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 e_1 e_2, 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)

eval : Expr → ℕ
eval = fold id _+_
```

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.

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

  unload⁺ : ℕ → Stack → ℕ
  unload v Top             = v
  unload v (Right v' stk) = unload⁺ (v' + v) stk
  unload v (Left 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:

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

- We give a verified proof of termination of tail-rec-eval using a carefully chosen *well-founded relation* (Sections 2 and 3). After redefining tail-rec-eval using this relation, we can prove the two evaluators equal in Agda.
- We generalize this relation and its corresponding proof of well-foundedness, inspired by McBride's work on *dissections* [McBride 2008], to show how to calculate an abstract machine from an algebra. To do so, we define a universe of algebraic data types and a generic fold operation (Section 4). Subsequently we show how to turn any structurally recursive function defined using a fold into its tail-recursive counterpart.
- Finally, we present how our proofs of termination and semantics preservation from our example are generalized to the generic fold (Sections 4.6 and 4.7).

Together these results give a verified function that computes a tail-recursive traversal from any algebra for any algebraic datatype. All the constructions and proofs presented in this paper have been implemented in and checked by Agda. The corresponding code is freely available online.[1]

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 unload function, that calls load on the expression stored on the top of the stack. From the definition of 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 load as follows:

```
load (Add e₁ e₂) stk = load e₁ (Left (f e₂) stk)
```

we might use some function `f : Expr → Expr` to push *arbitrary* expressions on the stack, potentially leading to non-termination.

The functions load and 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 load traverses the tree, until it finds the leftmost leaf; second when 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 load and unload must terminate – but how can we convince Agda's termination checker of this?

As a first approximation, we revise the definitions of load and 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.

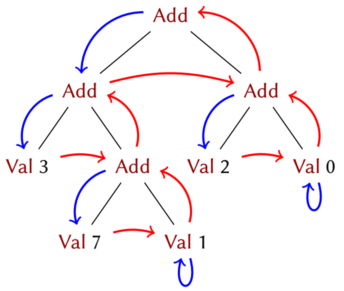

Figure 1. Traversing a tree with load and unload

The function unload is defined by recursion over the stack as before, but with one crucial difference. Instead of always returning the final result, it may also[2] return a new configuration of our abstract machine, that is, a pair `ℕ × Stack`:

```
unload : ℕ → Stack → (ℕ × Stack) ⊎ ℕ
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 other key difference arises in the definition of load:

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

Rather than calling unload upon reaching a value, it returns the current stack and the value of the leftmost leaf. Even though the function never returns an `inj₂`, its type is aligned with the type of unload so the definition of both functions resembles an 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 evaluator[3]:

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

Here we use 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 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

---

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

[2] `⊎` is Agda's type of disjoint union.

[3] We ignore load's impossible case, it can always be discharged with `⊥-elim : ∀ {X : Set} → ⊥ → X`.

have a relation between the states ℕ × Stack that decreases after every call to unload and a proof that the relation is well-founded – we know this function will terminate eventually. We now define the following version of the tail-recursive evaluator:

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
... | inj₁ (n , stk) = rec (n , stk) ☐₁
  where
    rec : (c : ℕ × Stack) → Acc _<_ c → ℕ
    rec (n , stk) (acc rs) with unload n stk
    ... | inj₁ (n' , stk') = rec (n' , stk') (rs ☐₂ )
    ... | inj₂ r           = r
```

To complete this definition, we still need to define a suitable relation _<_ between configurations of type ℕ × Stack, prove the relation to be well-founded ($\boxed{}_1$: Acc _<_ (*n* , *stk*)) and show that the calls to unload produce 'smaller' states ($\boxed{}_2$: (*n'* , *stk'*) < (*n* , *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* : 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 ℕ × Stack, are an instance of *dissections*, corresponding to a partially evaluated expression:

```
Config : Set
Config = ℕ × 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, 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 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 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 Config datatype:

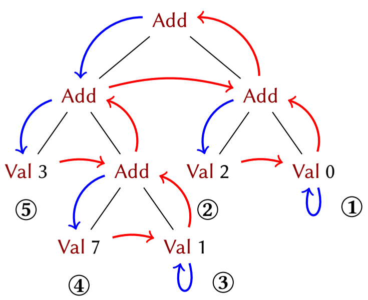

Figure 2. Numbered leaves of the tree

1. The 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 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 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 Config denotes a leaf in our input expression. In the previous example, the following Config corresponds to the third leaf:

As we observed previously, we would like to refine the type Config to capture the invariant that execution preserves: every 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 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 Stack type as follows:

```
data Stack⁺ : Set where
  Left  : Expr → Stack⁺ → Stack⁺
```

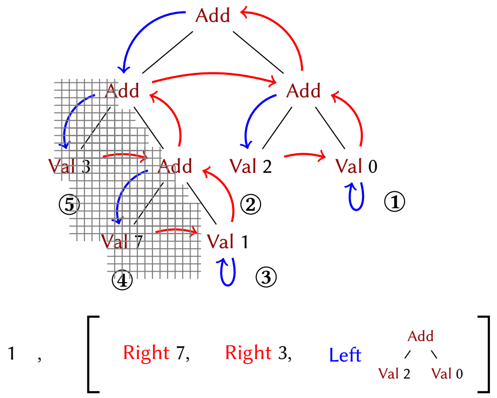

Figure 3. Example: *Configuration* of leaf number 3

```
Right : (n : ℕ) → (e : Expr) → eval e ≡ n → Stack⁺ → Stack⁺
Top   : Stack⁺
```

The 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 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 Config uses Stack⁺ as its right component:

```
Config = ℕ × Stack⁺
```

The function 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, Stack⁺:

```
unload⁺ : (n : ℕ) → (e : Expr) → eval e ≡ n → Stack⁺
        → Config ⊎ ℕ
unload⁺ n e eq Top            = inj₂ n
unload⁺ n e eq (Left e′ stk)  = load e′ (Right n e eq stk)
unload⁺ n e eq (Right n′ e′ eq′ stk)
   = unload⁺ (n′ + n) (Add e′ e) (cong₂ _+_ eq′ eq) stk
```

A value of type Config contains enough information to recover the input expression. This is analogous to the *plug* operation on zippers:

```
plug⇑ : Expr → Stack⁺ → Expr
plug⇑ e Top              = e
plug⇑ e (Left t      stk)  = plug⇑ (Add e t) stk
plug⇑ e (Right _ t _ stk)  = plug⇑ (Add t e) stk

plugC⇑ : Config → Expr
plugC⇑ (n , stk) = plug⇑ (Val n) stk
```

Any two terms of type 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 *statically* enforce such requirement by defining a new wrapper data type over Config that records the original input expression:

```
data Config⇑ (e : Expr) : Set where
  _,_ : (c : Config) → plugC⇑ c ≡ e → Config⇑ e
```

For a given expression e : Expr, any two terms of type Config⇑ e 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 Stack in the 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 Config corresponding to leaves with numbers 3 and 4 in our running example:

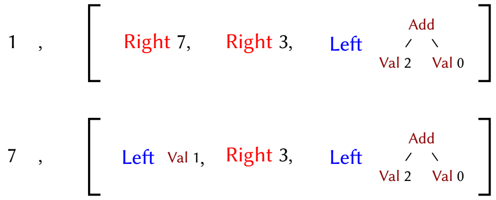

Figure 4. Comparison of *configurations* for leaves 3 and 4

The natural way to define the desired order relation is by induction over the 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 *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 *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 Add where one of the positions is in the left subtree and the other in the right.

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

```
plug⇓ : Expr → Stack⁺ → Expr
plug⇓ e Top              = e
plug⇓ e (Left t      stk)  = Add (plug⇓ e stk) t
plug⇓ e (Right _ t _ stk)  = Add t (plug⇓ e stk)

plugC⇓ : Config → Expr
plugC⇓ (n , stk) = plug⇓ (Val n) stk
```

We can convert freely between these two interpretations by reversing the stack. Furthermore, this conversion satisfies the plug$_{\Downarrow}$-to-plug$_{\Uparrow}$ property, relating the two variants of plug:

```
convert : Config → Config
convert (n , s) = (n , reverse s)

plug⇓-to-plug⇑ : ∀ (c : Config)
               → plugC⇓ c ≡ plugC⇑ (convert c)
```

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

```
data Config⇓ (e : Expr) : Set where
  _,_ : (c : Config) → plugC⇓ c ≡ e → Config⇓ e
```

As a corollary of the plug$_{\Downarrow}$-to-plug$_{\Uparrow}$ property, we can define a pair of functions to switch between Config$_{\Uparrow}$ and Config$_{\Downarrow}$:

```
Config⇓-to-Config⇑ : (e : Expr) → Config⇓ e → Config⇑ e
Config⇑-to-Config⇓ : (e : Expr) → Config⇑ e → Config⇓ e
```

3.3 Ordering configurations

Finally, we can define the ordering relation over values of type Config$_{\Downarrow}$. Even if the Config$_{\Uparrow}$ is still used during execution of our tail-recursive evaluator, the Config$_{\Downarrow}$ type will be used to prove its termination.

The ⌞_⌟_<_ type defined below relates two configurations of type Config$_{\Downarrow}$ e, that is, two states of the abstract machine evaluating the input expression e:

```
data ⌞_⌟_<_ : (e : Expr) → Config⇓ e → Config⇓ e → Set where
  <-StepR : ⌞ r ⌟ ((t₁ , s₁) , …) < ((t₂ , s₂) , …)
     → ⌞ Add l r ⌟ ((t₁ , Right l n eq s₁) , eq₁) < ((t₂ , Right l n eq s₂) , eq₂)
  <-StepL : ⌞ l ⌟ ((t₁ , s₁) , …) < ((t₂ , s₂) , …)
     → ⌞ Add l r ⌟ ((t₁ , Left r s₁) , eq₁) < ((t₂ , Left r s₂) , eq₂)
  <-Base : (eq₁ : Add e₁ e₂ ≡ Add e₁ (plugC⇓ t₁ s₁))
     →     (eq₂ : Add e₁ e₂ ≡ Add (plugC⇓ t₂ s₂) e₂)
     → ⌞ Add e₁ e₂ ⌟ ((t₁ , Right n e₁ eq s₁) , eq₁) < ((t₂ , Left e₂ s₂) , eq₂)
```

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 'smaller than' those in the left subtree.

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

```
<-WF : ∀ (e : Expr) → Well-founded (⌞ e ⌟_<_)
<-WF e x = acc (aux e x)
  where
    aux : ∀ (e : Expr) (x y : Config⇓ e)
       → ⌞ e ⌟ y < x → Acc (⌞ e ⌟_<_) y
    aux = …
```

The proof follows the standard schema[4] of most proofs of well-foundedness. It uses an auxiliary function, aux, that proves every configuration smaller than x is accessible.

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 e; without the type index, the subexpressions to the left e_1 and right e_2 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, inj$_2$ v, the entire input tree has been processed and the function terminates:

```
step : (e : Expr) → Config⇑ e → Config⇑ e ⊎ ℕ
step e ((n , stk) , eq)
  with unload⁺ n (Val n) refl stk
  … | inj₁ (n' , stk') = inj₁ ((n' , stk') , …)
  … | inj₂ v          = inj₂ v
```

We have omitted the second component of the result returned in the first branch, corresponding to a proof that plugC$_{\Uparrow}$ $(n', stk') \equiv e$. The crucial lemma that we need to show to complete this proof, demonstrates that the unload^{+} function respects our invariant:

```
unload⁺-plug⇑ :
  ∀ (n : ℕ) (e : Expr) (eq : eval e ≡ x) (s : Stack⁺) (c : Config)
   → unload⁺ n e eq s ≡ inj₁ c
   → ∀ (e′ : Expr) → plug⇑ e s ≡ e′ → plugC⇑ c ≡ e′
```

Finally, we can define the theorem stating that the step function always returns a smaller configuration:

---

[4] Most well-founded proofs in Agda standard library follow this pattern.

```
step-< : ∀ (e : Expr) → (c c' : Config⇑ e) → step e c ≡ inj₁ c'
       → ⌞ e ⌟ Config⇑-to-Config⇓ c' < Config⇑-to-Config⇓ c
```

Proving this statement directly is tedious, as there are many cases to cover and the expression *e* occurring in the types makes it difficult to identify and prove lemmas covering the individual cases. Therefore, we instead define another relation over non type-indexed configurations directly, and prove that there is an injection between both relations under suitable assumptions:

```
data _<_ : Config → Config → Set where
  <-StepR : (t₁ , s₁) < (t₂ , s₂)
          → (t₁ , Right l n eq s₁) < (t₂ , Right l n eq s₂)
  <-StepL : (t₁ , s₁) < (t₂ , s₂)
          → (t₁ , Left r s₁) < (t₂ , Left r s₂)
  <-Base  :   (e₁ ≡ plugC⇓ t₂ s₂) → (e₂ ≡ plugC⇓ t₁ s₁)
          → (t₁ , Right n e₁ eq s₁) < (t₂ , Left e₂ s₂)
to : (e : Expr) (c₁ c₂ : Config)
   → (eq₁ : plugC⇓ c₁ ≡ e) (eq₂ : plugC⇓ c₂ ≡ e)
   → c₁ < c₂ → ⌞ e ⌟ (c₁ , eq₁) < (c₂ , eq₂)
```

Thus to complete the previous theorem, it is sufficient to show that the function unload$^+$ delivers a smaller Config:

```
unload⁺-< : ∀ (n : ℕ) (s : Stack⁺) (e : Expr) (s' : Stack⁺)
          → unload⁺ n (Val n) refl s ≡ inj₁ (t′ , s')
          → (t′ , reverse s') < (n , reverse s)
```

The proof is done by induction over the stack supported; the complete proof requires some bookkeeping, covering around 200 lines of code, but is conceptually not complicated.

The function tail-rec-eval is now completed as follows[5]:

```
rec : (e : Expr) → (c : Config⇑ e)
    → Acc (⌞ e ⌟_<_) (Config⇑-to-Config⇓ c) → Config⇑ e ⊎ ℕ
rec e c (acc rs) = with step e c | inspect (step e) c
... | inj₂ n | _ = inj₂ n
... | inj₁ c' | [ Is ]
    = rec e c' (rs (Config⇑-to-Config⇓ c') (step-< e c c' Is))
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
... | inj₁ c = rec e (c , …) (<-WF e c)
```

Agda's termination checker now accepts that the repeated calls to rec are on strictly smaller configurations.

3.5 Correctness

As we have indexed our configuration datatypes with the input expression, proving correctness of it is relatively straightforward. The type of the function step guarantees that the configuration returned points to a leaf in the input expression.

Proving the function tail-rec-eval correct amounts to show that the auxiliary function, rec, that is iterated until a value is produced, will behave the same as the original evaluator, eval. This is expressed by the following lemma, proven by induction over the accessibility predicate:

```
rec-correct : ∀ (e : Expr) → (c : Config⇑ e)
            → (ac : Acc (⌞ e ⌟_<_) (Config⇑-to-Config⇓ c))
            → eval e ≡ rec e c ac
rec-correct e c (acc rs)
  with step e c | inspect (step e) c
... | inj₁ c' | [ Is ]
  = rec-correct e c' (rs (Config⇑-to-Config⇓ c') (step-< e c c' Is))
... | inj₂ n | [ Is ] = step-correct n e eq c
```

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$^+$:

```
unload⁺-correct : ∀ (n : ℕ) (e : Expr) (eq : eval e ≡ n) (s : Stack⁺)
                  ∀ (m : ℕ) → unload⁺ n e eq s ≡ inj₂ m
                   → eval (plug⇑ e s) ≡ 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:

```
correctness : ∀ (e : Expr) → eval e ≡ tail-rec-eval e
correctness e with load e Top
... | inj₁ c = rec-correct e (c , …) (<-WF e c)
... | inj₂ _ = ⊥-elim …
```

This finally completes the definition and verification of a tail-recursive evaluator.

4 A generic tail-recursive traversal

The previous section showed how to prove that our hand-written 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 U : Set describing the inhabitants of our universe together with its semantics,

---

[5] inspect is an Agda idiom needed to remember that *c'* is the result of the call step *e c*.

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

Types in this universe are formed from the empty type (𝟘), unit type (𝟙), and constant types (`K A`); the `I` constructor is used to refer to recursive subtrees. Finally, the universe is closed under both coproducts (`_⊕_`) and products (`_⊗_`). We could represent the *pattern* functor corresponding to the `Expr` type in this universe as follows:

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

Note that as the constant functor `K` takes an arbitrary type `A` as its argument, the entire datatype lives in `Set₁`. This could easily be remedied by stratifying this universe explicitly and parametrising our development by a base universe.

We can interpret the inhabitants of `Reg` as a functor of type `Set → Set`:

```
⟦_⟧ : Reg → Set → Set
⟦ 𝟘 ⟧ X        = ⊥
⟦ 𝟙 ⟧ X        = ⊤
⟦ I ⟧ X        = X
⟦ (K A) ⟧ X    = A
⟦ (R ⊕ Q) ⟧ X = ⟦ R ⟧ X ⊎ ⟦ Q ⟧ X
⟦ (R ⊗ Q) ⟧ X = ⟦ R ⟧ X × ⟦ Q ⟧ X
```

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

```
fmap : (R : Reg) → (X → Y) → ⟦ R ⟧ X → ⟦ R ⟧ Y
fmap 𝟘 f ()
fmap 𝟙 f tt              = tt
fmap I f x               = f x
fmap (K A) f x           = x
fmap (R ⊕ Q) f (inj₁ x) = inj₁ (fmap R f x)
fmap (R ⊕ Q) f (inj₂ y) = inj₂ (fmap Q f y)
fmap (R ⊗ Q) f (x , y)  = fmap R f x , fmap Q f y
```

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

```
data μ (R : Reg) : Set where
  In : ⟦ R ⟧ (μ R) → μ R
```

Next, we can define a *generic* fold, or *catamorphism*, to work on the inhabitants of the regular universe. For each code `R : Reg`, the `cata R` function takes an *algebra* of type `⟦ R ⟧ X → X` as argument. This algebra assigns semantics to the 'constructors' of `R`. Folding over a tree of type `μ R` corresponds to recursively folding over each subtree and assembling the results using the argument algebra:

```
cata : (R : Reg) → (⟦ R ⟧ X → X) → μ R → X
cata R ψ (In r) = ψ (fmap R (cata R ψ) r)
```

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

To address this, we fuse the `fmap` and `cata` functions into a single `map-fold` function [Norell 2008]:

```
map-fold : (R Q : Reg) → (⟦ Q ⟧ X → X) → ⟦ R ⟧ (μ Q) → ⟦ R ⟧ X
map-fold 𝟘        Q ψ ()
map-fold 𝟙        Q ψ tt         = tt
map-fold I        Q ψ (In x)     = ψ (map-fold Q Q ψ x)
map-fold (K A) Q ψ x            = x
map-fold (R ⊕ Q) P ψ (inj₁ x)  = inj₂ (map-fold R P ψ x)
map-fold (R ⊕ Q) P ψ (inj₂ y)  = inj₂ (map-fold Q P ψ y)
map-fold (R ⊗ Q) P ψ (x , y)   = map-fold R P ψ x , map-fold Q P ψ y
```

We can now define `cata` in terms of `map-fold` as follows:

```
cata : (R : Reg) (⟦ R ⟧ X → X) → μ R → X
cata R ψ (In r) = map-fold R R ψ r
```

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 `cata` function, we instantiate the catamorphism to work on the expressions and pass the desired algebra:

```
eval : μ exprF → ℕ
eval = cata exprF ϕ
  where ϕ : ⟦ exprF ⟧ ℕ → ℕ
        ϕ (inj₁ n)        = n
        ϕ (inj₂ (n , n')) = n + n'
```

In the remainder of this paper, we will develop an alternative 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 `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, ∇, computes a bifunctor for each element of our universe:

```
∇ : (R : Reg) → (Set → Set → Set)
∇ 𝟘        X Y = ⊥
∇ 𝟙        X Y = ⊥
∇ I        X Y = ⊤
∇ (K A)    X Y = ⊥
∇ (R ⊕ Q) X Y = ∇ R X Y ⊎ ∇ Q X Y
∇ (R ⊗ Q) X Y = (∇ R X Y × ⟦ Q ⟧ Y )
                ⊎ (⟦ R ⟧ X × ∇ Q X Y)
```

This operation generalizes the zippers, by defining a bifunctor $\nabla\ R\ X\ Y$. You may find it useful to think of the special case, $\nabla\ R\ X\ (\mu\ R)$ 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 ∇ is of particular interest: to *dissect* a product, we either *dissect* the left component pairing it with the second component interpreted over the second variable Y; or we *dissect* the second component and pair it with the first interpreted over X.

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

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

We can reconstruct Huet's zipper for generic trees of type $\mu\ R$ by instantiating both X and Y to $\mu\ R$.

Given a *dissection*, we can define a plug operation that assembles the context and current value in focus to produce a value of type $[\![\ R\ ]\!]\ Y$:

```
plug : (R : Reg) → (X → Y) → 𝒟 R X Y → ⟦ R ⟧ Y
plug 𝟘        η ((), x)
plug 𝟙        η ((), x)
plug I        η (tt , x)             = x
plug (K A)    η ((), x)
plug (R ⊕ Q) η (inj₁ r , x)          = inj₁ (plug R η (r , x))
plug (R ⊕ Q) η (inj₂ q , x)          = inj₂ (plug Q η (q , x))
plug (R ⊗ Q) η (inj₁ (dr , q) , x)  = (plug R η (dr , x) , q)
plug (R ⊗ Q) η (inj₂ (r , dq) , x)  = (fmap R η r , plug Q η (dq , x))
```

In the last clause of the definition, the *dissection* is over the right component of the pair leaving a value $r : [\![\ R\ ]\!]\ X$ to the left. In that case, it is only possible to reconstruct a value of type $[\![\ R\ ]\!]\ Y$, if we have a function η to recover Ys from Xs.

In order to ease things later, we bundle a *dissection* together with the functor to which it *plug*s as a type-indexed type.

```
data 𝒟ₓ (R : Reg) (X Y : Set) (η : X → Y) (tₓ : ⟦ R ⟧ Y) : Set where
  _,_ : (d : 𝒟 R X Y) → plug R η d ≡ tₓ → 𝒟ₓ R X Y η tₓ
```

4.3 Generic configurations

While the *dissection* computes the bifunctor *underlying* our configurations, we still need to take a fixpoint of this bifunctor. Each configuration consists of a list of *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 Stack^{+} datatype from the introduction, we also choose to store the original subtrees that have been visited and their corresponding correctness proofs:

```
record Computed (R : Reg) (X : Set) (ψ : ⟦ R ⟧ X → X) : Set where
  constructor _,_,_
  field
    Tree  : μ R
    Value : X
    Proof : cata R ψ Tree ≡ Value
```

```
Stackᴳ : (R : Reg) → (X : Set) → (ψ : ⟦ R ⟧ X → X) → Set
Stackᴳ R X ψ = List (∇ R (Computed R X ψ) (μ R))
```

A *stack* is a list of *dissections*. To the left we have the Computed results; to the right, we have the subtrees of type $\mu\ R$. Note that the Stack^{G} datatype is parametrised by the algebra ψ, as the Proof field of the Computed record refers to it.

As we saw in Section 3.5, we can define two different *plug* operations on these stacks:

```
plug-μ⇓ : (R : Reg) → {ψ : ⟦ R ⟧ X → X}
         → μ R → Stackᴳ R X ψ → μ R
plug-μ⇓ R t []          = t
plug-μ⇓ R t (h ∷ hs)   = In (plug R Computed.Tree h (plug-μ⇓ R t hs))
plug-μ⇑ : (R : Reg) → {ψ : ⟦ R ⟧ X → X}
         → μ R → Stackᴳ R X ψ → μ R
plug-μ⇑ R t []          = t
plug-μ⇑ R t (h ∷ hs)   = plug-μ⇑ R (In (plug R Computed.Tree h t)) hs
```

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

To define the configurations of our abstract machine, we are interested in *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 NonRec, stating when a tree of type $[\![\ R\ ]\!]\ X$ does not refer to the variable X, that will be used to represent recursive subtrees:

```
data NonRec : (R : Reg) → ⟦ R ⟧ X → Set where
  NonRec-𝟙  : NonRec 𝟙 tt
  NonRec-K  : (B : Set) → (b : B) → NonRec (K B) b
  NonRec-⊕₁ : (R Q : Reg) → (r : ⟦ R ⟧ X)
             → NonRec R r → NonRec (R ⊕ Q) (inj₁ r)
  NonRec-⊕₂ : (R Q : Reg) → (q : ⟦ Q ⟧ X)
             → NonRec Q q → NonRec (R ⊕ Q) (inj₂ q)
  NonRec-⊗  : (R Q : Reg) → (r : ⟦ R ⟧ X) → (q : ⟦ Q ⟧ X)
             → NonRec R r → NonRec Q q → NonRec (R ⊗ Q) (r , q)
```

As an example, in the pattern functor for the Expr type, $\mathsf{K}\ \mathbb{N} \oplus (\mathsf{I} \otimes \mathsf{I})$, terms built using the left injection are nonrecursive:

```
Val-NonRec : ∀ (n : ℕ) → NonRec (K ℕ ⊕ (I ⊗ I)) (inj₁ n)
Val-NonRec : n = NonRec-⊕₁ (K ℕ) (I ⊗ I) n (NonRec-K ℕ n)
```

This corresponds to the idea that the constructor *Val* is a leaf in a tree of type Expr.

On the other hand, we cannot prove the predicate NonRec for terms using the right injection. The occurences of recursive positions disallow us from framing the proof (The type NonRec does not have a constructor such as NonRec-I : $(x : X) \rightarrow$ NonRec I x).

This example also shows how ‘generic‘ leaves can be recursive. As long as the recursion only happens in the functor layer (code ⊕) and not in the fixpoint level (code I).

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

```
coerce : (R : Reg) → (x : ⟦ R ⟧ X) → NonRec R x → ⟦ 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:

```
Leaf : Reg → Set → Set
Leaf R X = Σ (⟦ R ⟧ X) (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:

```
Configᴳ : (R : Reg) → (X : Set) → (ψ : ⟦ R ⟧ X → X) → Set
Configᴳ R X ψ = Leaf R X × Stackᴳ R X ψ
```

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

```
plugC-μ⇓ : (R : Reg) {ψ : ⟦ R ⟧ X → X}
         → Configᴳ R X ψ → μ R → Set
plugC-μ⇓ R ((l , isl) , s) t = plug-μ⇓ R (In (coerce l isl)) s t
```

Note that the 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 load and unload given in the introduction (Section 2).

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

We write load^G by appealing to an ancillary definition first-cps, that uses continuation-passing style to keep the definition tail-recursive and obviously structurally recursive. If we were to try to define 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 first-cps function is daunting at first:

```
first-cps : (R Q : Reg) {ψ : ⟦ Q ⟧ X → X}
   → ⟦ R ⟧ (μ Q)
   → (∇ R (Computed Q X ψ) (μ Q) → (∇ Q (Computed Q X ψ) (μ Q)))
   → (Leaf R X → Stackᴳ Q X ψ → Configᴳ Q X ψ ⊎ X)
   → Stackᴳ Q X ψ
   → Configᴳ Q X ψ ⊎ X
```

The first two arguments are codes of type 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 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 Stack^G $Q\ X\ \psi$ is the current stack. The entire function will compute the initial configuration of our machine of type Config^G $Q\ X\ \psi$ [6]:

```
loadᴳ : (R : Reg) {ψ : ⟦ R ⟧ X → X} → μ R
      → Stackᴳ R X ψ → Configᴳ R X ψ ⊎ X
loadᴳ R (In t) s = first-cps R R t id (λ l → inj₁ ∘ _,_ l) s
```

We shall fill the definition of first-cps by cases. The clauses for the base cases are as expected. In 𝟘 there is nothing to be done. The 𝟙 and K A codes consist of applying the second continuation to the tree and the *stack*.

```
first-cps 𝟘 Q () _
first-cps 𝟙 Q x k f s     = f (tt , NonRec-𝟙) s
first-cps (K A) Q x k f s = f (x , NonRec-K A x) s
```

The recursive case, constructor I, corresponds to the occurrence of a subtree. The function 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.

```
first-cps I Q (In x) k f s = first-cps Q Q x id (λ c → inj₁ ∘ _,_ c) (k tt ∷ s)
```

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

```
first-cps (R ⊕ Q) P (inj₁ x) k f s = first-cps R P x (k ∘ inj₁) cont s
  where cont (l , isl) = f ((inj₁ l) , NonRec-⊕₁ R Q l isl)
first-cps (R ⊕ Q) P (inj₂ y) k f s = first-cps Q P y (k ∘ inj₂) cont s
  where cont (l , isl) = f ((inj₁ l) , NonRec-⊕₂ R Q l isl)
```

The interesting clause is the one that deals with the product. First the function 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 first-cps over the right component of the product. In case the continuation fails to to find a subtree, it returns the leaf as it is.

```
first-cps (R ⊗ Q) P (r , q) k f s = first-cps R P r (k ∘ inj₁ ∘ (, q)) cont s
  where cont (l , isl) = first-cps Q P q (k ∘ inj₂ ∘ _,_ (coerce l isl)) cont’
    where cont’ (l’ , isl’) = f (l , l’) (NonRec-⊗ R Q l l’ isl isl’)
```

---

[6] As in the introduction, we use a sum type ⊎ to align its type with that of unload^G.

Unload Armed with load^G we turn our attention to unload^G. 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 $\mathcal{D}\ R\ X\ Y$ or it shows that there are no occurrences of the variable left. In the latter case, it returns the functor interpreted over Y, $[\![\ R\ ]\!]\ Y$.

```
right : (R : Reg) → ∇ R X Y → X → ⟦ R ⟧ X ⊎ 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 $[\![\ R\ ]\!]\ X$.

The function unload^G is defined by induction over the *stack*. If the *stack* is empty the job is done and a final value is 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^G is dispatched, otherwise, if there is still a subtree to be processed the function load^G is called.

```
unloadᴳ : (R : Reg)
   → (ψ : ⟦ R ⟧ X → X)
   → (t : μ R) → (x : X) → cata R ψ t ≡ x
   → Stackᴳ R X ψ
   → Configᴳ R X ψ ⊎ X
unloadᴳ R ψ t x eq [] = inj₂ x
unloadᴳ R ψ t x eq (h ∷ hs) with right R h (t , x , eq)
unloadᴳ R ψ t x eq (h ∷ hs) | inj₁ r with compute R R r
... | (rx , rr) , eq' = unloadᴳ R ψ (In rp) (ψ rx) (cong ψ eq') hs
unloadᴳ R ψ t x eq (h ∷ hs) | inj₂ (dr , q) = loadᴳ R q (dr ∷ hs)
```

When the function right returns a inj$_1$ it means that there are not any subtrees left in the *dissection*. If we take a closer look, the type of the r in inj$_1$ r is $[\![\ R\ ]\!]$ (Computed $R\ X\ \psi$). The functor $[\![\ R\ ]\!]$ 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, $[\![\ R\ ]\!]\ X$, in order to apply the algebra; second, the functor interpreted over subtrees, $[\![\ R\ ]\!]\ (\mu\ R)$, 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 massages r to adapt the arguments for the recursive call to unload^G.

4.5 Relation over generic configurations

We can engineer a *well-founded* relation over elements of type Config$^G_{\Downarrow}$ 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 *stack*s.

To reduce clutter in the definition, we give a non type-indexed relation over terms of type Config^G. We can later use the same technique as in Section 3.4 to recover a fully type-indexed relation over elements of type Config$^G_{\Downarrow}$ t by requiring that the *zipper*s respect the invariant, plugC-$\mu_{\Downarrow}$ $c \equiv t$.

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

```
data _<ℂ_ : Configᴳ R X ψ → Configᴳ R X ψ → Set where
  Step : (t₁ , s₁) <ℂ (t₂ , s₂) → (t₁ , h ∷ s₁) <ℂ (t₂ , h ∷ s₂)
  Base : plugC-μ⇓ R (t₁ , s₁) ≡ e₁ → plugC-μ⇓ R (t₂ , s₂) ≡ e₁
       → (h₁ , e₁) <∇ (h₂ , e₂) → (t₁ , h₁ ∷ s₁) <ℂ (t₂ , h₂ ∷ s₂)
```

This relation has two constructors:

- The Step constructor covers the inductive case. When the head of both *stacks* is the same, i.e., both Config^{G}s share the same prefix, it recurses directly on tail of both stacks.
- The constructor Base accounts for the case when the head of the *stack*s 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 ⌞_⌟_<$_\nabla$_ that orders *dissections* of type $\mathcal{D}\ R$ (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 ⌞_⌟_<∇_ : (R : Reg) → 𝒟 R X Y → 𝒟 R X Y → Set where
  step-⊕₁ :   ⌞ R ⌟       (r , t₁)       <∇ (r' , t₂)
            → ⌞ R ⊕ Q ⌟ (inj₁ r , t₁) <∇ (inj₁ r' , t₂)

  step-⊕₂ :   ⌞ Q         (q , t₂)       <∇ (q' , t₂)
            → ⌞ R ⊕ Q ⌟ (inj₂ q , t₁) <∇ (inj₂ q' , t₂)

  step-⊗₁ :   ⌞ R ⌟       (dr , t₁)             <∇ (dr' , t₂)
            → ⌞ R ⊗ Q ⌟ (inj₁ (dr , q) , t₁) <∇ (inj₁ (dr' , q) , t₂)

  step-⊗₂ :   ⌞ Q ⌟       (dq , t₁)             <∇ (dq' , t₂)
            → ⌞ R ⊗ Q ⌟ (inj₂ (r , dq) , t₁) <∇ (inj₂ (r , dq') , t₂)

  base-⊗  :   ⌞ R ⊗ Q ⌟ (inj₂ (r , dq) , t₁) <∇ (inj₁ (dr , q) , t₂)
```

The idea is that we order the elements of a *dissection* in a left-to-right fashion. All the constructors except for base-⊗ simply follow the structure of the dissection. To define the base case, base-⊗, recall that the *dissection* of the product of two functors, $R \otimes Q$, has two possible values. It is either a term of type $\nabla\ R\ X\ Y \times [\![\ Q\ ]\!]\ Y$, such as inj$_1$ $(dr\ ,\ q)$ or a term of type $[\![\ R\ ]\!]\ X \times \nabla\ Q\ X\ Y$ like inj$_2$ $(r\ ,\ dq)$. The former denotes a position in the left component of the pair while the latter denotes a position in the right component. The base-⊗ 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, _<$_{\mathbb{C}}$_, has to be type-indexed by the tree of type $\mu\ R$ to which both *zipper* recursively plug through plugC-$\mu_{\Downarrow}$. The auxiliary relation, ⌞_⌟_<$_\nabla$_, needs to be type-indexed by the functor of type $[\![\ R\ ]\!]\ X$ to which both *dissections* plug:

```
data ⌞_⌟⌞_⌟_<∇_ {X Y : Set} {η : X → Y} : (R : Reg) → (tₓ : ⟦ R ⟧ Y)
   → 𝒟ₓ R X Y η tₓ → 𝒟ₓ R X Y η tₓ → Set where
```

```
data ⌞_⌟⌞_⌟_<_{ℂ⇓}_ {X : Set} (R : Reg) {ψ : ⟦ R ⟧ X → X} : (t : μ R)
  → Config^G_⇓ R X ψ t → Config^G_⇓ R X ψ t → Set where
```

The proof of *well-foundedness* of ⌞_⌟⌞_⌟_<$_{\mathbb{C}_\Downarrow}$_ is a straightforward generalization of proof given for the example in Section 3.3. The full proof of the following statement can found in the accompanying code:

```
<_ℂ-WF : (R : Reg) → (t : μ R) → Well-founded (⌞ R ⌟⌞ t ⌟_<_{ℂ⇓}_)
```

4.6 A generic tail-recursive machine

We are now ready to define a generic tail-recursive machine. To do so we now assemble the generic machinery we have defined so far. We follow the same outline as in Section 3.4.

The first point is to build a wrapper around the function unload^G that performs one step of the *catamorphism*. The function step^G statically enforces that the input tree remains the same both in its argument and in its result.

```
step^G : (R : Reg) → (ψ : ⟦ R ⟧ X → X) → (t : μ R)
    → Config^G_⇑ R X ψ t → Config^G_⇑ R X ψ t ⊎ X
```

We omit the full definition. The function step^G performs a call to unload^G, coercing the leaf of type $\llbracket\ R\ \rrbracket\ X$ in the Config$^G_\Downarrow$ argument to a generic tree of type $\llbracket\ R\ \rrbracket\ (\mu\ R)$.

We show that unload^G preserves the invariant, by proving the following lemma:

```
unload-plug^G_⇑ : ∀ (R : Reg) {ψ : ⟦ R ⟧ X → X}
                → (t : μ R) (x : X) (eq : cata R ψ t ≡ x) (s : Stack^G R X ψ)
                → (c : Config^G R X ψ)
                → ∀ (e : μ R) → plug-μ⇑ R t s ≡ e
                  → unload^G R ψ t x eq s ≡ inj₁ c → plug-μ⇑ R c ≡ e
```

Next, we show that applying the function step^G to a configuration of the abstract machine produces a smaller configuration. As the function step^G is a wrapper over the unload^G function, we only have to prove that the property holds for unload^G.

The unload^G function does two things. First, it calls the function right to check whether the *dissection* has any more recursive subtrees to the right that still have to be processed. It then dispatches to either load^G, if there is, or recurses otherwise. When there is a hole left, a new *dissection* is returned by right. Thus showing that the new configuration is smaller amounts to show that the *dissection* returned by right is smaller by ⌞_⌟_<$_\nabla$_. This amounts to proving the following lemma:

```
right-< : right R dr (t , y , eq) ≡ inj₂ (dr' , t')
        → ⌞ R ⌟ ((dr' , t')) <_∇ ((dr , t))
```

We have simplified the type signature, leaving out the universally quantified variables and their types.

Extending this result to show that the function unload^G delivers a smaller value is straightforward. By induction over the input stack we check if the traversal is done or not. If it is not yet done, there is at least one dissection in the top of the stack. The function right applied to that element returns either a smaller dissection or a tree with all values on the leaves. If we obtain a new dissection, we use the right-< lemma; in the latter case, we continue by induction over the stack. In this fashion, we can prove the following statement that our traversal decreases:

```
step^G-< : (R : Reg) (ψ : ⟦ R ⟧ X → X) → (t : μ R)
         → (c₁ c₂ : Config^G_⇑ R X ψ t)
         → step^G R ψ t c₁ ≡ inj₁ c₂ → ⌞ R ⌟⌞ t ⌟ c₂ _<_ℂ_ c₁
```

Finally, we can write the *tail-recursive machine*, 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:

```
rec : (R : Reg) (ψ : ⟦ R ⟧ X → X) (t : μ R)
    → (c : Config^G_⇑ R X ψ t)
    → Acc (⌞ R ⌟⌞ t ⌟_<_{ℂ⇓}_) (Config^G_⇑-to-Config^G_⇓ c) → X
rec R ψ t c (acc rs) with step^G R ψ t c | inspect (step^G R ψ t) c
... | inj₁ z′ | [ Is ] = rec R ψ t z′ (rs z′ (step^G-< R ψ t c z′ Is))
... | inj₂ x |  [ _ ] = x
tail-rec-cata : (R : Reg) → (ψ : ⟦ R ⟧ X → X) → μ R → X
tail-rec-cata R ψ x with load^G R ψ x []
... | inj₁ c = rec R ψ (c , ...) (<_ℂ-WF R c)
```

4.7 Correctness, generically

To prove our tail-recursive evaluator produces the same output as the catamorphism is straight-forward. As we did in the tail-rec-eval example (Section 3.5), we perform induction over the accessibility predicate in the auxiliary recursor. In the base case, when the function step^G returns a ground value of type X, we have to show that such value is the result of applying the *catamorphism* to the input. Recall that step^G is a wrapper around unload^G, hence it suffices to prove the following lemma:

```
unload^G-correct : ∀ (R : Reg) (ψ : ⟦ R ⟧ X → X)
                     (t : μ R) (x : X) (eq : cata R ψ t ≡ x)
                     (s : Stack^G R X ψ) (y : X)
                   → unload^G R ψ t x eq s ≡ inj₂ y
                   → ∀ (e : μ R) → plug-μ⇑ R t s ≡ e → cata R ψ e ≡ y
```

Our generic correctness result is an immediate consequence:

```
correctness^G : ∀ (R : Reg) (ψ : ⟦ R ⟧ X → X) (t : μ R)
              → cata R ψ t ≡ tail-rec-cata R ψ t
```

4.8 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 recap the *pattern* functor underlying the type Expr:

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

The Expr type is then isomorphic to tying the knot over exprF:

```
Expr^G : Set
Expr^G = μ exprF
```

The function eval is equivalent to instantiating the *catamorphism* with an appropriate algebra:

```
ψ : exprF ℕ → ℕ
ψ (inj₁ n)         = n
ψ (inj₂ (e₁ , e₂)) = e₁ + e₂
eval : Exprᴳ → ℕ
eval = cata exprF ψ
```

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

```
tail-rec-evalᴳ : Exprᴳ → ℕ
tail-rec-evalᴳ = tail-rec-cata exprF ψ
```

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 [Dybjer 1994], or inductive-recursive types [Dybjer 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 [Danielsson 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

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: constructing strictly positive types. *Theoretical Computer Science* 342, 1 (2005), 3–27.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A Functional Correspondence Between Evaluators and Abstract Machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP '03)*. ACM, New York, NY, USA, 8–19. https://doi.org/10.1145/888251.888254

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015).

Thorsten Altenkirch and Conor McBride. 2003. Generic programming within dependently typed programming. In *Generic Programming*. Springer, 1–20.

Thorsten Altenkirch, Conor Mcbride, and Peter Morris. 2007. Generic Programming with Dependent Types. In *Spring School on Datatype-Generic Programming*, Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). LNCS, Vol. 4719. Springer-Verlag.

Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *International Conference on Mathematics of Program Construction*. Springer, 52–67.

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1863543.1863547

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2364527.2364546

Olivier Danvy. 2009. *From Reduction-Based to Reduction-Free Normalization.* Springer Berlin Heidelberg, Berlin, Heidelberg, 66–164. https://doi.org/10.1007/978-3-642-04652-0_3

Peter Dybjer. 1994. Inductive families. *Formal Aspects of Computing* 6, 4 (01 Jul 1994), 440–465. https://doi.org/10.1007/BF01211308

Peter Dybjer and Anton Setzer. 1999. A Finite Axiomatization of Inductive-Recursive Definitions. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–146.

Gérard Huet. 1997. The zipper. *Journal of functional programming* 7, 5 (1997), 549–554.

Peter J Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* 6, 4 (1964), 308–320.

Per Martin-Löf. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Napoli.

Conor McBride. 2008. Clowns to the Left of Me, Jokers to the Right (Pearl): Dissecting Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 287–295. https://doi.org/

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.).

Keiko Nakata and Tarmo Uustalu. 2009. Trace-based coinductive operational semantics for while. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 375–390.

Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. 2008. A Lightweight Approach to Datatype-generic Rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP '08)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1411318.1411321

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.

Ulf Norell. 2008. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*. Springer, 230–266.

Nicolas Oury and Wouter Swierstra. 2008. The Power of Pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1411204.1411213

Wouter Swierstra. 2012. From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012*. 163–177. https://doi.org/10.4204/EPTCS.76.10

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1596550.1596585