Universiteit Utrecht

Master Thesis in Computing Science

# Verified tail-recursive folds through dissection

Carlos Tomé Cortiñas

# Abstract

The functional programming paradigm advocates a style of programming based on higher-order functions over inductively defined datatypes. A fold, which captures their common pattern of recursion, is the prototypical example of such a function. However, its use comes at a price.

The definition of a fold is not tail-recursive which means that the size of the stack during execution grows proportionally to the size of the input. McBride [2008] has proposed a method called *dissection*, to transform a fold into its tail-recursive counterpart. Nevertheless, it is not clear why the resulting function terminates, nor it is clear that the transformation preserves the fold's semantics.

In this thesis, we formalize the construction of such tail-recursive function and prove that it is both terminating and equivalent to the fold. In addition, using McBride's dissection, we generalize the tail-recursive function to work on any algebra over any regular datatype.

# Acknowledgements

say something nice

# Contents

# 1 Introduction

Folds, or *catamorphisms*, are a pervasive programming pattern. Folds generalize many simple traversals over algebraic data types [Meijer et al. 1991]. Functions implemented by means of a fold are both compositional and structurally recursive. The fold associated with a datatype, however, is not a tail-recursive function.

We start this chapter with a detailed description of the problem that motivates the research conducted within this master thesis (Section 1.1), and subsequently, we formulate the concrete research questions (Section 1.2). Lastly, in section 1.3 we outline the organization of the rest of this document.

## 1.1 Description of the problem

The foldr function is one of the first higher-order functions that any functional programmer learns [Hutton 2016]. Many simple functions over lists such as map, reverse, take, sum, and more can be expressed in terms of foldr. However, if not used carefully, foldr may cause a *well-typed program go wrong* by dynamically failing with a stack overflow. In order to understand the problem, let us review the definition of foldr:[1]

```
foldr : (α → β → β) → β → List α → β
foldr f e []       = e
foldr f e (x :: xs) = f x (foldr f e xs)
```

In the second clause of the definition, the parameter function *f* cannot reduce further before the result of the recursive call on the argument *xs* is available. This is a problem of both strict languages and non-strict languages with strict functions.

If we think about it in terms of the execution of a stack machine, before the control flow is passed to the recursive call, a new frame has to be allocated on the top of the stack to resume with the reduction of *f*. Only a finite number of frames may be ever pushed on the stack before it reaches its limit. Performing a few steps of the evaluation of adding a very big list of numbers illustrates the issue:

```
foldr _+_ 0 [ 1 ..1000000 ]
      ⇝ 1 + (foldr _+_ 0 [ 2 ..1000000 ])
      ⇝ 1 + (2 + (foldr _+_ 0 [ 3 ..1000000 ]))
```

[1] Code snippets within this thesis are written in the dependently typed programming language *Agda* [Norell 2007]

```
  ⇝ 1 + (2 + (3 + (foldr _+_ 0 [ 4 ..1000000 ])))
  ⇝ 1 + (2 + (3 + (4 + (foldr _+_ 0 [ 5 ..1000000 ]))))
  ⇝ ...
```

At each step of the reduction, denoted by $\rightsquigarrow$, the size of the expression being evaluated reflects the size of the underlying machine's stack. Before any addition can actually reduce, the function foldr has to reach the end of the list; during the evaluation, foldr needs to allocate every intermediate application of _+_ on the stack. The linear dependency between the size of the input and the size of the stack, can potentially lead to a stack overflow on large inputs.

To solve this problem, we can rewrite the function to be *tail-recursive*. The definition of a tail-recursive function does not allow for another function to post process the result of the recursive calls. In each clause, either a value is returned or a recursive call is the final action to be performed. Modern compilers typically map tail-recursive functions to machine code that runs in constant stack space [Steele 1977].

In the case of the function foldr, a possible implementation of an equivalent tail-recursive function would be a left fold, foldl. Its definition is as follows:

```
foldl : (α → β → β) → β → List α → β
foldl f e []       = e
foldl f e (x :: xs) = foldl f (f x e) xs
```

Using foldl, the addition of the previous list of numbers runs in constant stack space:

```
foldl _+_ 0 [ 1 ..1000000 ]
  ⇝ foldl _+_ (1 + 0) [ 2 ..1000000 ]
  ⇝ foldl _+_ 1 [ 2 ..1000000 ]
  ⇝ foldl _+_ (2 + 1) [ 3 ..1000000 ]
  ⇝ foldl _+_ 3 [ 3 ..1000000 ]
  ⇝ foldl _+_ (3 + 3) [ 4 ..1000000 ]
  ⇝ foldl _+_ 6 [ 4 ..1000000 ]
  ⇝ ...
```

However, for inductive datatypes with constructors that have more than one recursive subtree, recovering a tail-recursive fold from the regular fold is not as straightforward.

**Folds for binary trees** As an example of a datatype with more than one recursive subtree, we consider the type of binary trees with natural numbers in the leaves:

```
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. The algebra is the pair of functions that assigns semantics to both constructors of Expr:

```
fold_expr : (ℕ → α) → (α → α → α) → Expr → α
fold_expr ϕ1 ϕ2 (Val n)     = ϕ1 n
fold_expr ϕ1 ϕ2 (Add e1 e2) = ϕ2 (fold_expr ϕ1 ϕ2 e1) (fold_expr ϕ1 ϕ2 e2)
eval : Expr → ℕ
eval = fold_expr id _+_
```

Unfortunately, the definition of eval suffers from the same shortcomings as the List function foldr. The operator _+_ needs both of its parameters to be fully evaluated before it can reduce further, thus the stack used during execution grows *again* linearly with the size of the input.

To address the problem, we can *manually* rewrite the evaluator to be *tail-recursive*. 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 e stk)   = load e (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 whether the tail-recursive evaluator, tail-rec-eval, produces the same result as our original eval function.

## 1.2 Research questions

As previously shown, it is not obvious how to write a *provably* terminating and correct tail-recursive evaluator for the type of binary trees. A necessary prerequisite to show correctness, is to convince *Agda*'s termination checker that the tail-recursive evaluator terminates. Thus, we are ready to spell the research questions that this master thesis is set out to answer:

1. **Termination** *Agda*'s termination checker cannot verify that the functions load and unload, as previously defined, terminate for every possible input. How can we demonstrate that the functions terminate, so that as a corollary the tail-recursive evaluator terminates?

2. **Correctness** In the case the tail-recursive evaluator terminates, it is correct? By correct it is understood that both the evaluation function, eval, and its tail-recursive counterpart, tail-rec-eval, are equivalent: for any input both functions compute the same result.

3. **Generalization to the *regular* universe** McBride proposes a method, *dissection* [2008], for calculating the type of the *stack* from the definition of any type that can be generically expressed in the *regular* universe. Can we generalize the results of **termination** and **correctness** from Expr to the generic case through *dissection*?

We answer questions one and two in chapter 3, where we show how to *manually* write a tail-recursive evaluator for the type of Expr. We subsequently prove the evaluator to be both terminating and correct with respect to the fold.

In chapter 4, we answer the third research question. Particularly, we generalize the result from chapter 3 and develop a terminating tail-recursive evaluator that works for any algebra over any regular datatype. Additionally, we prove the evaluator to be correct with regard to the fold associated with the datatype.

## 1.3 Organization

This master thesis is divided in four chapters.

We start in chapter 2 giving the reader a broader perspective on folds in programming languages to justify the importance of our our work. Furthermore, we revisit the available literature on techniques to assist the termination checker to accept functions that are not defined by strictly structural recursion. We end the chapter with an introduction to generic programming in *Agda* using the *regular* universe, which forms the basis of our generic tail-recursive evaluator.

Chapters 3 and 4 contain the main contributions of this master thesis: a provably terminating and correct tail-recursive function equivalent to the fold over Expr, and its generalization to the regular universe.

We conclude this document, in chapter 5, with a few remarks about the work presented here and discuss possible future directions to pursue.

**Style** Before dwelling into the content, we have to remark a few conventions that this document follows. The purpose of the code snippets present in this thesis is to guide the reader through the ideas of our construction, thus in many cases only

the type signature of the relevant functions/theorems/datatypes is given, and the body is omitted altogether. All code snippets use *Agda* syntax, although not all of them typecheck directly. In the type signatures, any mentioned variable of type Set is taken as implicitly universally quantified. To differentiate between functions and theorems (in dependent type theory they are the same) we choose to prepend the type signatures of the latter with a explicit $\forall$ quantifier. The full *Agda* formalization is freely available online in:

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

# 2 Background

In this chapter, we introduce some of the concepts that are mandatory prerequisites for understanding the main parts of this master thesis. The chapter is organized into three different sections, whose content is seemingly unrelated. We begin in section 2.1 with a broad overview of semantics in programming languages and its relation to the content of this master thesis. In section 2.2, we revisit the literature about techniques for assisting the termination checker of *Agda*. Lastly, in section 2.3 we quickly overview generic programming in the context of this thesis.

## 2.1 A broader perspective

There are three main approaches for formalizing the semantics of a programming language: small-step operational semantics, where for each construct of the language it is specified how the abstract machine, which is evaluating the program, evolves; denotational semantics, where each construct is mapped by a *mathematical* function to the value it evaluates in the denotational domain of the language; and big-step operational semantics, where the overall results of the computation are characterized by a relation over the terms of the language.

The connection between the first two styles of formalizing the semantics has been extensively exploited, for example see Ager et al. [2003], to derive well-known abstract machines, such as the Krivine [2007] machine, from a denotational semantics specification and vice versa. However, as they state in the article:

> Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language.

A dependently typed programming language such as *Agda* is a perfect vehicle for the study and implementation of programming languages. Dependent types can be fruitfully leveraged for defining both the language, and correspondingly formalize and verify its semantics either in a small-step or a denotational style. For example, Swierstra [2012b] shows how to derive the Krivine machine in *Agda* starting from a small-step evaluation semantics for lambda terms.

However, implementing both a small-step and a denotation function and proving that they are related is a tedious work that requires some particular expertise.

From a high level perspective, the denotational semantics of a language is a function that is both compositional and structurally recursive. For each term, it

specifies how the values that the subterms denote are combined together into a value for the term on focus. In disguise, the denotation function is just a fold! The fold is inherent to the inductive structure of the datatype that represents the language, thus everything is needed is the algebra that for each constructor determines how to combine the values.

On the other hand, the small-step semantics of a language require some extra work. As a start, the programming language engineer has to define the configurations, or internal states, of the machine. Then, she has to specify the transition rules that govern the machine and finally show, if the language is strongly normalizing, that iterating the small-step evaluation function terminates.

The work on this master thesis can be regarded as a step towards exploiting, in a formal environment such as *Agda*, the connection between high-level denotational functions—in the form of folds—and low-level abstract machines. Given a language in terms of its generic representation and an algebra, we construct a generic tail-recursive function, i.e. the low-level abstract machine, that we later formally proof equal to the fold induced by its structure.

## 2.2 Termination in type theory

*Agda* is a language for describing total functions. General recursive functions are not allowed as they would render the logic inconsistent. It is not possible to decide in general if a recursive function terminates. To ensure that any defined function terminates, *Agda* uses a termination checker based on foetus [Abel 1998], that syntactically checks whether the recursive calls of a function are performed over **structurally** smaller arguments. The termination checker, however, is not complete: there are programs that terminate but the termination checker classifies as possibly non terminating.

Many interesting and terminating functions that we would like to define do not conform to the pattern of being defined by structural recursion. For instance, the naive tail-recursive evaluator presented in the introduction (Section 1.1).

In this section, we explore several available techniques which allow us to reason within *Agda* about termination conditions. Particularly, we revisit *sized types* (Section 2.2.1), *Bove-Capretta* predicates (Section 2.2.2) and *well-founded* recursion (Section 2.2.3).

As a running example, we consider the sorting function *quicksort* implemented in a functional style:

```
quickSort : (α → α → Bool) → List α → List α
quickSort p [] = []
quickSort p (x :: xs) = quickSort p (filter (p x) xs)
                        ++ [ x ] ++
                        quickSort p (filter (not ∘ (p x)) xs)
```

*Agda*'s termination checker marks the function as possibly non terminating. In the second clause of the definition, the arguments to both recursive calls, `filter (p x) xs` and `filter (not ∘ (p x))` are not structurally smaller than the input list `x :: xs`. The termination checker has no reason whatsoever to believe that the application of the function `filter` to the list does not increase its size. As a contrived example, we

could have made a mistake in the definition of filter, replicating elements, so that quickSort as defined above may diverge:

```
filter_bad : (α → Bool) → List α → List α
filter_bad p [] = []
filter_bad p (x :: xs) = if p x then x :: x :: filter_bad p xs
                         else filter_bad p xs
```

On its own, $\text{filter}_{bad}$ is a perfectly valid function that the termination checker classifies as terminating for any possible input. However, if any other function uses its result as a recursive argument, such function could diverge.

The termination checker only uses *local* information of the definition of a function to classify it as terminating or possibly non terminating. Concretely, it constructs a graph between the parameters of the function and the arguments passed to recursive calls and tries to find a well-founded order amongst them. In any case, calls to other functions are treated as a blackbox. The reason is that syntactically checking termination conditions is not modular: two terminating functions when combined together may give rise to a divergent function.

The rest of this section is devoted to show how using the aforementioned techniques, we can convince *Agda* that quicksort terminates on any input.

### 2.2.1 Sized types

Sized types [Abel 2010] is a type system extension that allows to track structural information on the type level. Terms can be annotated with a type index that represents an **upper bound** of the actual *size* of the term being annotated. The size of a term is the number of constructors used to build it.

Functions can quantify over size variables to relate the size of its parameters to the size of its result. A term is only well-typed if the type system can ensure during type checking that the size type annotation of a term conforms to its actual size. Size annotations are gathered during typechecking and passed to a linear inequality solver to check their validity. The type Size used to annotate sizes can be understood as the type of ordinal numbers without a base element. Its definition is built-in in the *Agda* compiler, and corresponds to the following:

```
postulate
  Size : Set
  ω    : Size
  ↑_   : Size → Size
```

Usually, sized types are used to relate the size of the input of a function to the size of its result. Therefore, sizes in functions are universally quantified variables that index both the type of the domain and the codomain. For instance, the identity function over a sized type would be written as:

```
id_S : {A : Size → Set} → {i : Size} → A i → A i
id_S x = x
```

In the next example we explain in more detail how to program with sized types in order to show that the *quicksort* function always terminates.

**Example 2.2.1**

We start the example by defining the type of *sized* lists. The difference with the regular definition of list, List, is that its signature has a new type index of type Size. The return type of every constructor explicitly instantiates the Size index in such a way that the size of the recursive occurrences is related to the size of the value being constructed. The definition of sized lists is as follows:

```
data SList (α : Set) : Size → Set where
  SNil  : {i : Size} → SList α i
  SCons : {i : Size} → α → SList α i → SList α (↑ i)
```

In the constructor SNil there are no recursive occurrences, thus the Size type-index is universally quantified. On the other hand, in the constructor SCons the returned Size is the size of the recursive parameter, $i$, increased by one, $\uparrow i$. Indeed, the constructor is adding a new 'layer' on top of its parameter.

Using the sized type SList we define a filter function that is guaranteed to preserve the size of its input list: the result list does not gain new elements. We do so by explicitly declaring in its type signature that the size of the result does not exceed—recall that the size is an upper bound—the size of the parameter:

```
filterS : {i : Size} → (α → Bool) → List α i → List α i
filterS {_ i}     p (SNil {i})          = SNil
filterS {_ (↑ i)} p (SCons {i} x xs) = if p x then SCons x (filterS {i} p xs)
                                       else filterS {i} p xs
```

The second clause of the definition is interesting. In the else branch the type of the recursive call is List $\alpha$ $i$ but the expected type, from the signature of the function, is List $\alpha$ $(\uparrow i)$. Sized types come with a subtyping relation which states that List $\alpha$ $i \leqslant_S$ List $\alpha$ $(\uparrow i)$, thus the recursive call is well-typed.

With the definition of $\text{filter}_S$ in hand, we are ready to define the function quicksort over the type of sized lists such that it is catalogued as terminating by the termination checker:

```
quickSortS : {i : Size} → (α → α → Bool) → List α i → List α ω
quickSortS {_ i} p (SNil {i}) = SNil
quickSortS {_ (↑ i)} p (SCons {i} x xs)
  = quickSortS {i} p (filterS {i} (p x) xs)
      ++ [ x ] ++
    quickSortS {i} p (filterS {i} (not ∘ p x) xs)
```

The first thing to note is the use of $\omega$ as the size annotation in the type of the resulting list. The concatenation function, _++_, has type {$i$ $j$ : Size} → List $\alpha$ $i$ → List $\alpha$ $j$ → List $\alpha$ $\omega$, where $\omega$ indicates that we know nothing about the *size* of the resulting list. In fact this is a limitation, both theoretically and at the implementation level, of sized types. The system is not sufficiently expressive to give the function _++_ a more precise type such as {$i$ $j$ : Size} → List $\alpha$ $i$ → List $\alpha$ $j$ → List $\alpha$ $(i + j)$. Nevertheless, it is enough to demonstrate to *Agda* that the implementation of quicksort

terminates. Specifically, in the second clause of the definition, the information about the size of the input, $\uparrow i$, is propagated to the function $\text{filter}_S$ that it is known to preserve the size of its input. The recursive call is now provably terminating.

If we try to reimplement the bogus version of filter, $\text{filter}_{bad}$, using sized types, its definition is not *well-typed* and the typechecker raises a compile-time error.

---

A termination check based on sized types represents an improvement over a termination check that works purely in the syntactic level. Sized types allow the programmer to introduce semantic annotations about sizes both in types and functions so they can be exploited for a more accurate assessment of termination. As we showed in the previous example, termination based on sized types is modular because it works across the boundaries of function definitions. However, the expressivity of the system is somewhat limited and in general sized types are not first class entities in the language, but rather built-in objects with special treatment subject to some restrictions.

### 2.2.2 Bove-Capretta predicate

Another commonly used technique in type theory to encode general recursive functions is the Bove-Capretta [Bove and Capretta 2001] transformation. The call graph of any function, even if is not defined by structural recursion, poses an inductive structure that can be exploited to show termination. Instead of directly defining the function, the call graph of the original function is added as a new parameter so the function can be defined by structural recursion over it.

The call graph of a function of type f : $\alpha \to \beta$, can be made explicit as a predicate over the input type $\alpha$, P : $\alpha \to$ Set. Thus, the possibly non terminating function f is transformed into another function f' : $(x : \alpha) \to$ P $x \to \beta$ that uses the argument P $x$ as the recursive structure. The domain predicate P outlines the conditions for which the function f is known to terminate.

However, not everything in the garden is rosy. Every time we want to call function f' we have first to prove that the predicate holds on the argument we supply. Showing that the function terminates for every possible input, amounts to construct a proof that the predicate is true for every element of type $\alpha$, that is $\forall\, (x : \alpha) \to$ P $x$.

**Example 2.2.1**

We now turn our attention to encode the function quicksort using the Bove-Capretta technique. The first step is to define the call graph of the function as a predicate:

```
data qsPred (p : α → α → Bool) : List α → Set where
  qsNil  : qsPred p []
  qsCons : {x : α} → {xs : List α}
                   → qsPred p (filter (p x) xs)
                   → qsPred p (filter (not ∘ p x) xs)
```

$$\rightarrow \text{ qsPred } p\ (x :: xs)$$

As explained before, the predicate qsPred encodes the conditions on which the function quickSort terminates. The constructor qsNil represents the base case: quicksort always terminates if the input list is empty. In the inductive case, constructor qsCons, the termination of quicksort on the input list $x :: xs$ depends solely on the termination of quicksort on the inputs filter $(p\ x)\ xs$ and filter $(\text{not} \circ p\ x)\ xs$.

Thus we can define now a version of the function quickSort that is accepted by the termination checker. We introduce a new parameter, the predicate qsPred applied to the input list, and recurse over it:

```
quickSort_BC : (p : α → α → Bool) → (xs : List α)
                                  → qsPred p xs → List α
quickSort_BC p . [] qsNil = []
quickSort_BC p . (x :: xs) (qsCons lesser greater)
  = quickSort_BC p ((filter (p x) xs)) lesser
      ++ [ x ] ++
    quickSort_BC p (filter (not ∘ (p x)) xs) greater
```

Every time $\text{quickSort}_{BC}$ is called with a list $xs$, also a proof that the predicate holds, qsPred $p\ xs$ has to be supplied. We know that quickSort terminates for every possible input, thus it should be possible to define a theorem that states that the qsPred predicate is true for any list of any type:

```
qsPred-true : ∀ (p : α → α → Bool) → (xs : List α) → qsPred p α
qsPred-true = ...
```

Proving the previous theorem, however, is not possible just by structural recursion. Basically, we would fall in the same problem as before but this time manipulating lists at the type level. In order to complete the proof, we need a more advanced technique, such as *well-founded* recursion (Section 2.2.3).

---

The Bove-Capretta transformation allows the programmer to decouple the task of defining a function with proving its termination. First, it is enough to outline the definition of the wanted function and identify its call graph. The construction of the domain predicate is a fully automatic matter. Nevertheless, the programmer is required to show every time the function is called that the input satisfies the predicate. Even if the function obviously terminates for every input, showing that the domain predicate holds in general cannot be done by pattern matching and structural recursion.

### 2.2.3 Well-founded recursion

The last technique we will discuss is *well-founded* recursion. Amidst the three, it is the most relevant for this work because the results of this master thesis heavily rely on its use.

The main idea is simple: define a relation over the type of the parameter that gets 'smaller' in each invocation of a function, and show that the relation has the property of not decreasing indefinitely.

Formally, for a given binary relation over elements of type $\alpha$, $\_<\_ : \alpha \to \alpha \to \mathsf{Set}$, an element $x : \alpha$ is *accessible* if there are no infinite descending chains starting from it by repeated decrements, $x_0 < x_1 < \ldots < x_{n-1} < x_n < x$. A more constructive characterization of the accessibility predicate in type theory, due to Nordström [1988], is the following type:

```
data Acc (_<_ : α → α → α) (x : α) : Set where
  acc : (∀ (y : α) → y < x → Acc _<_ y) → Acc _<_ x
```

An element $x : \alpha$ is accessible, if every smaller element by the relation is also accessible. The type Acc is inductively defined in such a way that we can only construct a particular proof of Acc _<_ $x$ if transitively we can show that any smaller element is also accessible. A given element, for which no smaller elements exists, i.e. $y < x \to \bot$, constitutes the base case and trivially satisfies the predicate.

The recursive structure of the accessibility predicate can be used to turn a non structurally recursive function into one that is accepted by the termination checker. A possibly non terminating function f : $\alpha \to \alpha$, is transformed into another function f' : $(x : \alpha) \to$ Acc _<_ $x \to \alpha$, that takes as extra parameter a proof that the input satisfies the accessibility predicate. The recursive structure of the Acc datatype can be exploited only, if for each invocation of f, we can prove the result to be smaller than the input.

When another function calls f', it has to explicitly supply a proof that the concrete input is accessible. If for every element in the relation there are no infinite descending chains, the relation is *well-founded*. Thus, for every possible input of a function, defined by structural recursion over the accessibility predicate, the initial proof needed to kick off the computation can be algorithmically constructed. We express that a relation is well-founded as follows:

```
Well-founded : (α → α → Set) → Set
Well-founded _<_ = ∀ x → Acc _<_ x
```

**Example 2.2.1**

We proceed to show how to encode the quickSort function using well-founded recursion. The first step is to define a relation over the input type List:

```
data _<ₗ_ : List α → List α → Set where
  Base : (x : α) (xs : List α)                → []       <ₗ (x :: xs)
  Step : (x y : α) (xs ys : List α) → xs <ₗ ys → (x :: xs) <ₗ (y :: ys)
```

The relation has two constructors:

- The constructor Base represents the base case: the empty list is always smaller than any list built up with _::_.
- The inductive case is provided by Step. A list $x :: xs$ is smaller than a list $y :: ys$ if inductively the tail of the former is smaller than the tail of the latter.

In the next step, we define quickSort as a function that takes the accessibility predicate over the relation as an extra argument and recurse over it:

```
quickSort_WF : (α → α → Bool) → (xs : List α)
                              → Acc _<L_ xs → List α
quickSort_WF p [] (acc rs)          = []
quickSort_WF p (x :: xs) (acc rs) =
  quickSort_WF p (filter (p x) xs) (rs (filter (p x) xs) □1)
    ++ [ x ] ++
  quickSort_WF p (filter (not ∘ p x) xs) (rs (filter (not ∘ p x) xs) □2)
```

The holes that are left, $\square_1$ : filter (p x) xs <L (x :: xs) and $\square_2$ : filter (not ∘ p x) xs <L (x :: xs), necessitate of an ancillary lemma expressing that the function filter always returns a smaller list by the relation:

```
filter-<L : ∀ (p : α → Bool) (x : α) (xs : List α) → filter p xs <L (x :: xs)
filter-<L p x [] = Base x []
filter-<L p x (y :: xs) with p y
... | false = lemma-<-:: x (filter p xs) (y :: xs) (filter-<L p y xs)
... | true  = Step y x (filter p xs) (y :: xs) (filter-<L p y xs)
```

The definition of lemma-<-:: shows that for any lists *ys* and *ys*, if *ys* is smaller than *xs*, *ys* <L *xs*, then regardless of how many elements are prepended to *xs*, *ys* remains smaller:

```
lemma-<-:: : ∀ (x : α) (ys xs : List α) → ys <L xs → ys <L (x :: xs)
```

The lemma is easily completed by induction over the argument *ys*.

Every time the programmer wants to call $\text{quickSort}_{WF}$, she has to produce a proof that the input is accessible by _<L_. It is burdensome to impose such requirement, specially when is clear that quickSort terminates for every possible input. To solve this undesirable situation, we can show once and for all that every element is accessible. The constructive nature of the *well-foundedness* proof (it is an algorithm) serves as the procedure to build the accessibility predicate proofs for every input in the domain. The proof of the theorem is as follows:

```
<L-Well-founded : Well-founded _<L_
<L-Well-founded x = acc (aux x)
  where aux-Step : ∀ (x : α) (xs : List α) → Acc _<L_ xs
                   → ∀ (y : List α) → y <L (x :: xs) → Acc _<L_ y
        aux-Step x xs (acc rs) ._ []          (Base ._ ._)
          = acc λ { _ () }
        aux-Step x xs (acc rs) ._ (y :: ys) (Step y ._ ys ._ p)
          = acc (aux-Step y ys (rs ys p))
        aux : ∀ (x : List α) → (y : List α) → y <L x → Acc _<L_ y
        aux ._ (x :: xs) ._ (Base x xs) = acc λ { _ () }
```

```
aux _ (y :: ys) _ (x :: xs) (Step x y xs ys p)
  = acc (aux-Step x xs (aux ys xs p))
```

The proof follows the usual structure of well-foundedness proofs that can be found in the Agda standard library. An auxiliary function aux is used, whose definition is by induction over the proof. In the base case, there are no smaller lists than [], thus the proof is discharged by appealing to the impossible pattern. In the inductive case, where two lists built up with _::_ are compared, we need another ancillary lemma, aux-Step. This lemma says that if the tail of a list *xs* is accessible, then any list that results from prepending elements to it is accessible too. It is noteworthy to mention that the proof relies on showing the termination checker that something *structurally* decreases. In the case of aux-Step, the proof decreases, while in the function aux both the proof and the input get smaller. Nevertheless, when the proof and the input are not related, i.e their type does not depend on a common argument, this is not the case.

Finally, the quicksort function is defined as a wrapper over $\text{quickSort}_{WF}$:

```
quickSort : (α → α → Bool) → List α → List α
quickSort p xs = quickSort_WF p xs (<_L-Well-founded xs)
```

---

The previous example is well engineered to be straightforward. We declare a relation over lists and the proof of well-foundedness follows almost immediately from the definition of the relation. Well-founded proofs are not always that simple, in the next example we examine how the proof is very dependent on the inductive structure of the relation.

**Example 2.2.2**

Let us consider the natural numbers and two equivalent definitions of the $<$ (strict less than) relation:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data _<₁_ (m : ℕ) : ℕ → Set where
  Base₁ :                    m <₁ suc m
  Step₁ : (n : ℕ) → m <₁ n → m <₁ suc n

data _<₂_ : ℕ → ℕ → Set where
  Base₂ : (n : ℕ)                → zero <₂ suc n
  Step₂ : (n m : ℕ) → n <₂ m → suc n <₂ suc m
```

In the first relation, constructors are peeled off from the first argument until there is a difference of one which constitutes the base case. On the other hand, in the second relation, the constructors are removed from both arguments until the left reaches zero.

It should be clear that both definitions are equivalent. However, the first is

more suitable to prove well-foundedness due to the explicit *structural relation* between both arguments.

```
<₁-Well-founded : Well-founded _<₁_
<₁-Well-founded x = acc (aux x)
  where
    aux : ∀ (x : ℕ) → ∀ (y : ℕ) → y <₁ x → Acc _<₁_ y
    aux .(suc y) y Base₁       = <₁-Well-founded y
    aux .(suc n) y (Step₁ n p) = aux n y p
```

Pattern matching on the relation allows us to refine both arguments. The recursive call to the well-foundednees proof in the Base case is allowed because $y$ is structurally smaller than suc $y$. In the step case we can recurse using aux because the proof $p$ is structurally smaller than Step $p$.

However, things are not that easy with the second definition. As an attempt we might try the following:

```
<₂-Well-founded : Well-founded _<₂_
<₂-Well-founded x = acc (aux x)
  where
    aux : (x : ℕ) → ∀ (y : ℕ) → y <₂ x → Acc _<₂_ y
    aux zero y ()
    aux (suc x) .zero Base₂          = acc λ { _ () }
    aux (suc x) .(suc y) (Step₂ y p) = aux x (suc y) □
```

The $\text{Base}_2$ case is effortless: there are no natural numbers smaller than zero, thus it is eliminated using the impossible pattern. In the inductive case, $\text{Step}_2$, the refined patterns are not adequate to use in a recursive call, the arguments are not structurally related.

To tackle the problem, we have to introduce an auxiliary lemma that links the inductive structure of both inputs. Such the lemma states the following:

```
lemma₂ : ∀ (n : ℕ) (m : ℕ) → n <₂ suc m → n ≡ m ⊎ n <₂ m
```

Finally, we can complete <₂-Well-founded proof by appealing to $\text{lemma}_2$ and dispatching a recursive call in the inductive case:

```
<₂-Well-founded : Well-founded _<₂_
<₂-Well-founded x = acc (aux x)
  where
    aux : (x : ℕ) → ∀ (y : ℕ) → y <₂ x → Acc _<₂_ y
    aux zero y ()
    aux (suc x) y p with lemma₂ _ _ p
    aux (suc x) .x p | inj₁ refl = <₂-Well-founded x
    aux (suc x) y p  | inj₂ i    = aux x y i
```

This example illustrates how the proof of well-foundedness is totally dependent on the choice of the relation. In the first relation, _<₁_, the proof

follows directly by induction from the definition, but the second relation, $\_<_2\_$, necessitates some extra work and a bit of insight to complete the proof.

---

Using well-founded recursion the programmer can write a non structurally recursive function directly in *Agda*. Before writing such a function, a suitable relation over the type of elements has to be defined. Moreover, it is necessary to prove that the argument decreases, by the relation, with each application of the function.

Then, there are two options: each time the function is called a proof that the input is accessible by the relation is explicitly supplied, or, the relation is proven to be well-founded and the proof is used to produce the required evidence.

## 2.3 Generic programming

There are many opinions on what the term "generic programming" means, depending on whom you ask. For a thoroughly account of its different flavours, we recommend the reader to the material by Gibbons [2007]. Nevertheless, a central idea prevails: find a common ground in the implementation details that can be abstracted away such that when instantiated can be applied over and over.

The second part of this master thesis presents a generalization of the tail-recursive evaluator from part one (Chapter 3), on the type Expr, to the "generic" case. What is meant by generic in this context? it means **datatype generic**, which Gibbons refers to as *shape genericity*: abstract over the shape of a datatype, or its inductive structure.

In the rest of this section, we give a fast-track introduction to generic programming with dependent types. We put special interest on the *regular* universe, for which we later, chapter 4, construct the generic tail-recursive evaluator.

### 2.3.1 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 1984a], that is, a data type U : Set describing the inhabitants of our universe together with its semantics, and a function, 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 (_⊗_). Note that as the constant functor K takes an arbitrary type $A$ as its argument, the entire

datatype lives in $\mathsf{Set}_1$. This could easily be remedied by stratifying this universe explicitly and parametrising our development by a base universe.

We can interpret the inhabitants of 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 law abiding fmap operation:[1]

```
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
```

**Example 2.3.1**

We can encode the type of booleans, Bool, in the *regular* universe. Such type is represented by a code built out of the combination of two unit functors, $\mathbb{1}$, using the coproduct $\_\oplus\_$. The lack of the constructor I in the code allow us to interpret it over any type we like:

```
Bool^Reg : Reg
Bool^Reg = 𝟙 ⊕ 𝟙
Bool^G : Set
Bool^G = ⟦ Bool^Reg ⟧ ⊤
```

---

The universe as-is forbids the representation of inductive types. Simple recursive datatypes can be expressed as their underlying pattern functor and a fixed point that ties the recursion explicitly. In *Agda*, the least fixed point of a functor associated with an element of our universe is defined as follows:

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

A functor layer given by the code $R$ is interpreted by substituting the recursive positions, marked by the constructor I, with generic trees of type $\mu\ R$. The definition

[1] The proof is left as an exercise.

of the fixed point is constrained to functors built within the universe. In general, the fixed point of a non-positive[2] type can be used to build non-normalizing terms, leading to inconsistency.

**Example 2.3.2**

As a first example of a recursive datatype, we show how to encode the usual type of cons-lists in the regular universe. The construction is simple: first, we express the *pattern functor* underlying the constructors, _::_ and [], as a generic code of type Reg, then the fixed point delivers the representation of List:

```
List^Reg : Set → Reg
List^Reg α = 𝟙 ⊎ (K α ⨁ I)
List^G : Set → Set
List^G α = μ (List^Reg α)
```

The type $\mathsf{List}^G$ is the generic representation of the List type, and we can witness their equivalence by writing a pair of embedding-projection functions:

```
from : {α : Set} → List α → List^G α
from []       = In (inj₁ tt)
from (x :: xs) = In (inj₂ (x , from xs))
to : {α : Set} → List^G α → List α
to (In (inj₁ tt))      = []
to (In (inj₂ (x , xs))) = x :: to xs
```

That satisfy the following roundtrip properties:

```
from-to : ∀ {α : Set} → (xs : List α ) → to (from xs) ≡ xs
to-from : ∀ {α : Set} → (xs : List^G α) → from (to xs) ≡ xs
```

---

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 $[\![ R ]\!]\ X \to X$ as argument. This algebra assigns semantics to the constructors of $[\![ R ]\!]\ X$. Folding over a tree of type $\mu\ R$ 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, 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.

[2]The type being defined appears in negative positions, that is as a function argument, in its own constructors.

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 2.3.3**

We can take the type of expressions from the introduction, section 1.1, and encode it in the *regular* universe in two steps: first, we define the code of the *pattern functor* underlying the constructors; second, the generic representation of Expr arises from tying the knot over the pattern functor:

```
ExprR : Reg
ExprR = K ℕ ⊕ (I ⊗ I)
Expr^G : Set
Expr^G = μ ExprR
```

The type $\text{Expr}^G$ is equivalent to Expr, so we can define a embedding-projection pair:

```
to : Expr^G → Expr
to (In (inj₁ n))         = Val n
to (In (inj₂ (e₁ , e₂))) = Add (to e₁) (to e₂)
from : Expr → Expr^G
from (Val n)     = In (inj₁ n)
from (Add e₁ e₂) = In (inj₂ (from e₁ , from e₂))
```

The example evaluator, eval, is equivalent to a function defined using the generic catamorphism, cata, that instantiates the code argument with ExprR and the algebra with _+_ and id:

```
eval : Expr^G → ℕ
eval = cata ExprR ψ
  where ψ : ⟦ ExprR ⟧ ℕ → ℕ
        ψ (inj₁ n)       = n
        ψ (inj₂ (n , n')) = n + n'
```

Generic programming within the regular universe was first explored by Noort et al. [2008] in the context of a generic rewriting system written in *Haskell* [Hudak et al. 1992]. The implementation in *Haskell* differs from the one we have presented here because the language has no support for first class dependent types (yet). Each code in the universe we defined, Reg, is encoded as a different datatype. Generic functions are written as methods of a typeclass [Wadler and Blott 1989] that is then instantiated to every datatype in the 'universe'. In addition, the library provides a typeclass $Regular$ that uses associated type synonyms [Chakravarty et al. 2005] to witness the isomorphism, i.e. embedding-projection pair, between a datatype and its generic representation.

The regular library has, however, a rather limited expressivity. As the authors acknowledge:

> One of the most important limitations of the library described in this paper is that it only works for datatypes that can be represented by means of a fixed-point. Such datatypes are also known as regular datatypes. This is a severe limitation, which implies that we cannot apply the rewriting library to nested datatypes or systems of (mutually recursive) datatypes.

Indeed, the regular universe can only represent simple algebraic datatypes. Datatypes that contain functions—exponentials—[Meijer and Hutton 1995]; that are nested [Bird and Meertens 1998]; or that are type indexed [Dybjer 1994] cannot be encoded in the universe.

# 3 A verified tail-recursive evaluator

In this chapter, we present the termination and correctness proof of a tail-recursive fold equivalent to the evaluation function eval, introduced in section 1.1. As a starting point, section 3.1, we take the definitions of the functions load and unload and reformulate them: the problem of termination is reduced into finding a suitable well-founded relation. In the next section, section 3.2, we show how to step-by-step construct such relation and prove its well-foundedness. Section 3.3, presents the terminating tail-recursive evaluator, and finally, in section 3.4, we prove its correctness with regard to the eval function. We conclude in section 3.5 with a discussion about the pros and cons of our evaluator and point out other possible solutions in the design space.

## 3.1 Setting the stage

In the first place, we recapitulate the definitions of load and unload from the introduction (Section 1.1):

```
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)
```

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 fig. 3.1.

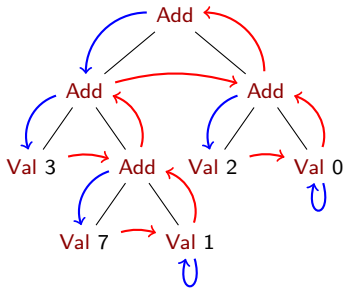


Figure 3.1: Traversing a tree with load and unload

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 perform one 'step' of the tail-recursive fold.

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[1] 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 $\mathrm{inj}_2$, its type is aligned with the type of unload so the definition of the 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 the functions to define the following evaluator:

[1] ⊎ is *Agda*'s type of disjoint union.

```
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 have a relation between the states $\mathbb{N} \times \text{Stack}$ that decreases after every call to unload and a proof that the relation is well-founded—we know this function will terminate eventually, we 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 $\mathbb{N} \times \text{Stack}$, prove the relation to be well-founded ($\square_1$ : Acc _<_ $(n , stk)$) and show that the calls to unload produce 'smaller' states ($\square_2$ : $(n' , stk') < (n , stk)$). In the next sections, we define such a relation and prove it is well-founded.

## 3.2 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 $\mathbb{N} \times \text{Stack}$, 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—i.e. the first leaf found after the initial call to load—is the greatest element; the rightmost leaf is the smallest. In our example expression, fig. 3.1, we would number the leaves as follows:

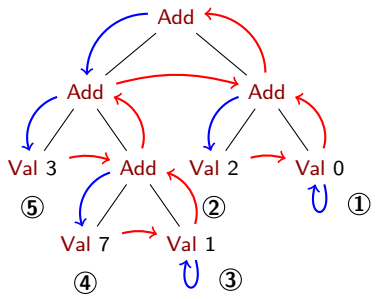


Figure 3.2: Numbered leaves of the tree

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:

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 desired 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 3.2.1). Secondly, we explore a different representation of stacks, as paths from the root, that facilitates the definition of the desired order relation (Section 3.2.2). Subsequently, we will define the relation over configurations, section 3.2.3, and sketch the proof that it is well-founded.

### 3.2.1 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:

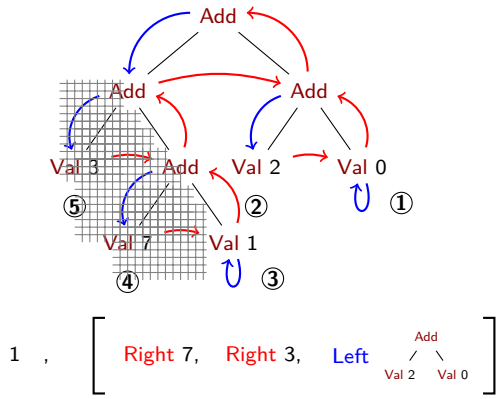


Figure 3.3: *Configuration* of leaf number 3

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 computing 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 fig. 3.3, 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⁺
  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 runtime, but only required for the proof of well-foundedness – a point we will return to in our discussion (Section 3.5). From now onwards, the type Config uses $\text{Stack}^+$ as its right component:

```
Config = ℕ × Stack⁺
```

The function $\text{unload}^+$ was previously defined by induction over the stack (Section 3.1), thus, it needs to be modified to work over the new type of stacks, $\text{Stack}^+$:

```
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 $\text{Config}_{\Uparrow}\ e$ are configurations of the same abstract machine during the tail-recursive fold over the expression $e$.

### 3.2.2 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:

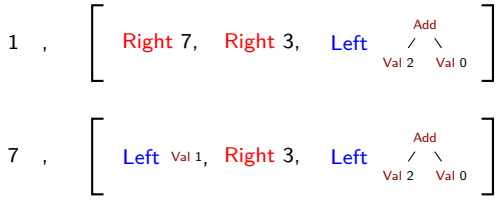


Figure 3.4: Comparison of *configurations* for leaves 3 and 4

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 *bottom* 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 $\text{plug}_{\Uparrow}$ 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 $\text{plug}_{\Downarrow}$-to-$\text{plug}_{\Uparrow}$ 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 the Config type to denote 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 $\text{plug}_{\Downarrow}$-to-$\text{plug}_{\Uparrow}$ property, we can define a pair of functions to switch between $\text{Config}_{\Uparrow}$ and $\text{Config}_{\Downarrow}$:

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

### 3.2.3 Ordering configurations

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

The $\llcorner\_\lrcorner\_<\_$ type defined below relates two configurations of type $\text{Config}_{\Downarrow}$ $e$, 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. To ensure that both configurations represent positions in the same expression, the <-Base constructor takes as a parameter a pair of equalities such that: the leaf pointed by the tail of the stack, $(t_1 , s_1)$, coincides, $e_2 \equiv \mathsf{plugC}_{\Downarrow}\ t_1\ s_1$, with the subtree stored in the top of the stack of the other configuration $(t_2 , \mathsf{Left}\ e_2\ s_2)$.

### 3.2.4 Well-founded relation

Now we turn out attention 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 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 the 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 $e$.

Following the same layout of example 2.2.1, the proof uses two lemmas that propagate the property of well-foundedness from structurally smaller configurations, i.e. with less elements in the stack:

```
accR : ∀ (l : Expr) (r : Expr) (x : ℕ) (s : Stack⁺) (n : ℕ) (eq : eval l ≡ n)
     → Acc (⌞ r ⌟_<_) (x , s)
     → ∀ (y : Config⇓ (Add l r)) → ⌞ Add l r ⌟ y < (x , Right l n eq s)
                                  → Acc (⌞ Add l r ⌟_<_) y
accL : ∀ (l : Expr) (r : Expr) (x : ℕ) (s : Stack⁺)
     → Well-founded (⌞ r ⌟_<_)
     → Acc (⌞ l ⌟_<_) (x , s)
     → ∀ (y : Config⇓ (Add l r)) → ⌞ Add l r ⌟_<_ y < (x , Left r s)
                                  → Acc (⌞ Add l r ⌟_<_) y
```

The first lemma, accR, follows directly from induction over the accessibility predicate. In the second lemma, accL, the proof is done by induction over the argument $y$. There are two cases to consider: the inductive case, <-StepL, proceeds by recursion over the accessibility predicate on the left subexpression Acc (⌞ l ⌟_<_) (x , s). However, the non-inductive case, constructor <-Base, poses a technical challenge: for the relation to be well-founded on the expression Add l r depends on itself being well-founded on the right subtree $r$. The former lemma, accR, handles this case if we

can supply a proof that the right subtree is accessible Acc (⌞ r ⌟_<_) (x , s), which indeed, accL can produce using its argument of type Well-founded (⌞ r ⌟_<_). This is only only possible because in the auxiliary function, aux, the initial call to accL, can recursively use the well-foundedness proof <-WF. Pattern matching reveals that the input expression e is a node Add l r, thus the recursive call is done on a structurally smaller input. Acceptance by the termination checker certifies it.

The type ⌞_⌟_<_, more than being a single relation over configurations, is a family of relations, one for every possible value of type Expr. Although indexing the relation, and the configurations, is *necessary* to prove that it is well-founded, it is not amenable to prove properties of functions regarding the relation. For instance, a proof that the function unload$^+$ returns a smaller configuration would require a lot of bookkeeping for the type index.

Instead of working directly with ⌞_⌟_<_, we define another auxiliary relation over non type-indexed configurations, and prove that there is an injection between both 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)
```

The definition of _<_ is an exact blueprint of its type-indexed counterpart. The only difference is that all the refined type indices stripped off the constructors.

## 3.3 A terminating tail-recursive evaluator

We now have almost all the definitions in place to revise our tail-recursive evaluator, 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 and complete the definition of our tail-recursive evaluator.

**Decreasing recursive calls** To define our tail-recursive evaluator, we will begin by defining an auxiliary step function that performs a single step of the 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 $\mathsf{plugC}_{\Uparrow}\ (n' , stk') \equiv e$. The crucial lemma, which we need to complete this proof, states that the $\mathsf{unload}^{+}$ 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′
```

The proof proceeds by induction over the stack part of the configuration. In the case the stack is empty, there is nothing to show, $\mathsf{unload}^{+}$ returns a natural number wrapped in $\mathsf{inj}_2$. In case the stack is not empty, depending on the element in the top, either Right or Left, it calls itself recursively or uses a lemma showing that the function load honors the invariant too:

```
load-plug⇑ : ∀ (e : Expr) (s : Stack⁺) (c : Config)
   → load e s ≡ inj₁ c
   → ∀ (t : Expr) → plug⇑ e s ≡ t → plugC⇑ c ≡ t
```

The lemma is proven by induction on the expression $e$.

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

```
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. We can simplify things by appealing to the non type-indexed relation _<_ and the lemma $\mathsf{unload}^{+}\text{-}\mathsf{plug}_{\Uparrow}$. Thus to complete the theorem, it is sufficient to show that the function $\mathsf{unload}^{+}$ delivers a smaller Config with the stacks reversed:

```
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:[2]

```
rec : (e : Expr) → (c : Config⇑ e)
    → Acc (⌞ e ⌟_<_) (Config⇑-to-Config⇓ c) → Config⇑ e ⊎ ℕ
```

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

```
rec e c (acc rs) = with step e c | inspect (step e) c
... | inj₂ n | _      = inj₂ n
... | inj₁ c' | [ ls ] = rec e c' (rs (Config⇑-to-Config⇓ c') (step-< e c c' ls))
```

The auxiliary recursor rec is defined by structural recursion over the accessibility predicate, thus, it provably terminates. Using the ancillary lemma step-<, we demonstrate that repeated invocations of the function step are done on strictly smaller configurations. Therefore, *Agda*'s termination checker accepts the function as terminating.

The tail-recursive evaluator, tail-rec-eval, is then defined as a wrapper over rec: it uses the fact that the relation is well-founded to feed the initial input and a proof that is accessible:

```
tail-rec-eval : Expr → ℕ
tail-rec-eval e with load e Top
... | inj₁ c = rec e (c , ...) (<-WF e c)
```

## 3.4 Correctness

Indexing the datatype of configurations is useful when proving correctness of the tail-recursive evaluator. The type of the function step guarantees by construction that the input expression never changes during the fold: the invariant consistently holds. Because the input expression remains constant across invocations, the result of eval does so also.

Proving the function tail-rec-eval correct amounts to showing that the auxiliary function, rec, iterated until a value is produced, behaves as eval. The auxiliary function rec is defined by recursion over the accessibility predicate, thus the proof is done by induction over the same argument:

```
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' | [ ls ]
  = rec-correct e c' (rs (Config⇑-to-Config⇓ c') (step-< e c c' ls))
... | inj₂ n | [ ls ] = step-correct n e eq c
```

While the proof by induction covers the recursion, we still have to prove the base case: when there are no more subexpressions left to fold, the resulting natural number is equal to evaluating the input expression using eval. The lemma step-correct precisely states that:

```
step-correct : ∀ (e : Expr) → (c : Config⇑ e)
             → ∀ (r : ℕ) → step e c ≡ inj₂ r → eval e ≡ r
```

As step is a wrapper around the function unload⁺, 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
```

```
unload⁺-correct e n eq Top ._ n refl    = sym eq
unload⁺-correct e n eq (Left x s) r p   = ⊥-elim ...
unload⁺-correct e n eq (Right r' e' eq' s) r p
  = unload⁺-correct (r' + r) (Add e' e) (cong₂ _+_ eq' eq) s r p
```

The proof follows immediately by induction over $s$ : Stack$^+$ using the fact that equality is congruent.

The main correctness theorem now shows 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 ...
```

The definition and verification of a tail-recursive evaluator is completed.

## 3.5 Discussion

In this chapter, we have seen how to define and verify a tail-recursive evaluator for the type of expressions Expr. Before wrapping up the evaluator, we address some open questions and issues:

- Our construction relies on two key points: type-indexed configurations and a well-founded relation. The former is essential for the latter, without the type-index in the configuration type, Config$_\Downarrow$, is not possible to prove well-foundedness. However, enlarging the type of the stacks to prove the required properties comes at a cost: the runtime impact of the function tail-rec-eval is larger than the pair of mutually recursive functions load and unload, section 1.1, that we took as starting point.

- Our tail-recursive evaluator is tied to a concrete algebra composed of the functions _+_ and id, however, a tail-recursive machine capable of computing the fold for any algebra over any Expr would be preferable.

- Alternatively, we can formulate a provably terminating tail-recursive fold using continuations. The idea consists in storing a partially applied recursive call – the continuation– at the point where the argument is known to be structurally smaller than the input. In such approach, however, the execution stack is no longer a first-order object, thus, the tail-recursive function cannot be longer understood as the formalization of an abstract machine.

- The tail-recursive evaluator we developed exchanges space in the execution stack for space in the heap. The runtime environment where the function is executed has to explicitly allocate space in the heap to hold the Stack argument of tail-rec-eval. On the practical level, it is not clear what we gain from the transformation.

- Previous work by Danvy [2009] has focused on constructing abstract machines from a one step reduction function. Our tail-recursive evaluator is an example of an abstract machine that uses a reduction function, the algebra. Both machines are definitely related.

- The step function, which our evaluator iterates, performs several reductions each time it is applied. However, the interpretation of a tail-recursive function as an abstract machine fits more naturally when the function it iterates reduces at most one redex at a time.

In the next paragraphs we discuss each of these points.

**Irrelevant arguments** *Agda* allows the programmer to identify the arguments of a function or the parameters of a constructor as *computationally irrelevant*. Code marked as irrelevant is erased, or interpreted as the unit type, when extracted into a *Haskell* executable. The typechecker has ensured that evaluation does not depend on irrelevant code.

If we compare the type of stacks, Stack and $\text{Stack}^+$, the constructor Right in the latter additionally stores the already evaluated subexpressions and the associated proofs:

```
data Stack : Set where
  Right : (n : ℕ) → Stack → Stack
  ...
data Stack⁺ : Set where
  Right : (n : ℕ) → (e : Expr) → (eval e ≡ n) → Stack⁺ → Stack⁺
  ...
```

The purpose of the expression and the proof is only to aid proving termination and correctness, and they should not produce any runtime overhead if we compare it with the naive tail-recursive function. We can address the issue and remove the extra cost, by marking both parameters as irrelevant in the type of of $\text{Stack}^+$:

```
data Stack⁺ᵢ : Set where    -- does not typecheck!
  Right : (n : ℕ) → .(e : Expr) → .(eval e ≡ n) → Stack⁺ᵢ → Stack⁺ᵢ
  ...
```

In the above definition, the expression $e$ : Expr and the proof eval $e \equiv n$ are irrelevant —marked with a preceding . (dot). Unfortunately, the datatype $\text{Stack}^+_\iota$ does not typecheck, the function eval expects a non irrelevant argument, which is necessary since we defined it by pattern matching on its argument. We can tackle this obstacle by reifiying the graph of the function eval as a datatype:

```
data Eval : .Expr → ℕ → Set where
  Eval-Val : (n : ℕ) → Eval (Val n) n
  Eval-Add : (e₁ e₂ : Expr) → (n n′ : ℕ)
           → Eval e₁ n → Eval e₂ n′ → Eval (Add e₁ e₂) (n + n′)
```

Because we marked the first index of Eval, of type Expr, as irrelevant, we can define the type of irrelevant stacks as follows:

```
data Stack⁺ᵢ : Set where
  Right : (n : ℕ) → .(e : Expr) → .(Eval e n) → Stack⁺ᵢ → Stack⁺ᵢ
  ...
```

If we assume that we can adapt the rest of the facilities, such as auxiliary datatypes, functions, etc, to use $\text{Stack}^+_\iota$ then the tail-recursive evaluator would have the same runtime impact as the pair of mutually recursive functions load and unload from the introduction (Section 1.1).

**Tail-recursive fold** We constructed a tail-recursive evaluator equivalent to a fold over expressions for a concrete algebra composed of _+_, for the constructor Add, and id for Val. The presentation in this simple terms is meant to reduce the overall clutter and let the reader focus on the ideas driving the construction. Nonetheless, we can easily generalize tail-rec-eval to a tail-recursive fold equivalent to fold (Section 1.1) that works for any algebra and for any Expr. First, we define an algebra over Expr as the following triple:

```
record Exprϕ : Set1 where
  field
    α  : Set
    ϕ1 : ℕ → α
    ϕ2 : α → α → α
```

The folding function fold takes the algebra as a parameter rather than a pair of functions:

```
fold : (alg : Exprϕ) → Expr → α alg
fold alg (Val n)     = ϕ1 alg n
fold alg (Add e1 e2) = ϕ2 alg (fold alg e1) (fold alg e2)
```

The rest of the construction accounts for the algebra by augmenting every datatype and function with a new parameter, the algebra. We can, instead, use a module parametrized by the algebra:

```
module _ (alg : Exprϕ) where
  ...
```

Using this approach, the definitions become more simple and clear; we do not have to keep track of the algebra in every type signature. However, at the formal level the formalizations are equivalent.

The reader can find the formalization of the tail-recursive fold that uses $\text{Expr}_\phi$ in the repository under the file `src/Tree/Indexed.agda`.

**Continuations** We can write a tail-recursive version of the evaluator from the introduction (Section 1.1) using continuations. In order to do so, we define an auxiliary function, go, by structural recursion over the expression passing the continuation as a parameter:

```
tail-rec-cont : Expr → ℕ
tail-rec-cont = go id
  where go : (ℕ → ℕ) → Expr → ℕ
        go k (Add e1 e2) = go (λ n → go (k ∘ (n +)) e2) e1
        go k (Val n)     = k n
```

In the first clause of the definition of go, the continuation passed as an argument to the recursive call uses the result of left subexpression, $e_1$, to recurse over the right subexpression, $e_2$. *Agda*'s termination checker classifies the function as terminating because the recursive call is done at a point where the argument is structurally smaller.

As an alternative design, we could redefine the type of stacks to explicitly store the continuation. In the Left constructor instead of saving the right subexpression

for later processing, we cache the continuation corresponding to a call of unload over such expression. The type of stacks with continuations would be as follows:

```
data Stack_cps : Set where
  Right_cps : ℕ                          → Stack_cps → Stack_cps
  Left_cps  : (ℕ → Stack_cps → ℕ) → Stack_cps → Stack_cps
  Top_cps   : Stack_cps
```

Accordingly, the pair of functions load and unload have to change to account for the continuations. The new load function, $\mathsf{load}_{\mathsf{cps}}$, creates a continuation on the right subexpression, and saves it on the stack, before it proceeds by recursion over the left subexpression. The replacement of the unload function, $\mathsf{unload}_{\mathsf{cps}}$, applies the continuation once it finds a Left constructor in the stack. The definition of both functions is the following:

```
unload_cps : ℕ → Stack_cps → ℕ
unload_cps n Top_cps             = n
unload_cps n (Right_cps n' stk)  = unload_cps (n + n') stk
unload_cps n (Left_cps k stk)    = k n stk

load_cps : Expr → Stack_cps → ℕ
load_cps (Add e1 e2) stk = load_cps e1 (Left (λ n → load_cps e2 ∘ (Right n)) stk)
load_cps (Val n) stk     = unload_cps x stk

eval_cps : Expr → ℕ
eval_cps e = load_cps e Top_cps
```

There is a problem, though, with this last approach. The $\mathsf{Stack}_{\mathsf{cps}}$ datatype is not strictly positive, indeed, *Agda*'s positivity checker highlights the non strictly positive occurrences in pink: the type $\mathsf{Stack}_{\mathsf{cps}}$ appears as an argument to the continuation in its own constructor $\mathsf{Left}_{\mathsf{cps}}$.

Using continuations explicitly, the evaluators are obviously terminating: they are solely defined by structural recursion over their input. Nonetheless, we cannot understand the tail-recursive functions tail-rec-cont and $\mathsf{eval}_{\mathsf{cps}}$ as first-order stack-based abstract machines anymore. The function tail-rec-cont does not manipulate explicitly a stack but rather uses functions in the host language, in this case *Agda*, to implement tail-recursion. On the other hand, the function $\mathsf{eval}_{\mathsf{cps}}$ does use a stack, but *again* relies on using functions from the host language to do tail-recursion. We have traded a first-order formulation in exchange for tail-recursion and termination.

**Space trade-off** Our tail-recursive evaluator uses a Stack argument as the reification of the underlying execution stack. The evaluator, indeed, is tail-recursive because it uses such argument to perform the tail calls. A explicit Stack, however, is not for free; the runtime system still has to allocate and manage the stack.

For instance, in a functional language such as *Haskell*, GHC [Marlow et al. 2004] –the *de facto Haskell* implementation– uses the same memory region for the allocation of the heap and the stack; the former grows upwards while the latter grows downwards. Then, what do we gain by transforming a fold into its tail-recursive counterpart?

We can mark the values saved in the stack with strictness [Wadler and Hughes 1987] annotations. At the point where the function $\mathsf{unload}^{+}$ stores a term $n : \mathbb{N}$

in the stack, such term is a fully evaluated value. The runtime system can reclaim the space that otherwise would occupy as a thunk.

In a language like *Haskell*, however, we have to take some extra care because of its non-strict semantics. Our tail-recursive evaluator is not exactly equivalent to a fold associated with a non inductive, possibly infinite, datatype. For instance, if we pass to the $fold$ function on Expr a function const $x\ y\ =\ x$ and the right subtree is an infinite value, then, our tail-recursive function does not terminate while the original $fold$ does.

**Decompose, contract, recompose** Danvy [2009] has previously shown how to derive a reduction-free evaluation function beginning from a small step reduction semantics. Given a term language, a specification of redexes in the language –i.e. terms the can be immediately reduced in one step–, and a one-step contraction function, Danvy shows how to construct an abstract machine to evaluate terms, that later turns into a reduction-free evaluation function.

The high level idea of his construction consists of applying a series of functions: *decompose* a term into potential redex and its evaluation context, *contract* the redex, and *recompose* the term by plugging back the result into the context. He then obtains the abstract machine by finding a fixed point of the composition of the three functions. He later observes [Danvy and Nielsen 2004] that the decomposition step always happens right after a recomposition, thus, he further optimizes the machine by deforesting the intermediate terms. He dubs the fusion of both functions, recompose and decompose, *refocusing*.

This concept of a machine is not very dissimilar to how our tail-recursive evaluator, tail-rec-eval, operates. Danvy's abstract machine iterates the one-step reduction function – decompose, contract, recompose– until the term is fully evaluated, while, our tail-recursive evaluator iterates the function step. A question is to wonder: how are both machines related for the type of expressions, Expr?

The first problem that arises, when formalizing Danvy's machine in a total language such as *Agda*, is that the decomposition step essentially corresponds to the pair of mutually recursive functions load and unload. As we previously saw in section 1.1, the termination checker classifies them as possibly non terminating. As we did for our tail-recursive machine, we could define a well-founded relation to show that traversing an expression to find its leftmost redex terminates.

In Danvy's machine, once *decompose* finds a redex, *contract* reduces it to a value. Our machine, on the other hand, uses the algebra, passed as a parameter to unload, to combine the results of previously evaluated subexpressions.

After Danvy's machine contracts the redex, it recomposes expression by plugging the value into the context. Our function unload, instead, recursively traverses the stack looking for the next subexpression to load.

We proved that our tail-recursive evaluator finds the fixed point of the one-step function, step, because we carefully engineered such function to deliver a smaller value by the well-founded relation over configurations. However, to iterate *decompose-contract-recompose* we would need to define a well-founded relation over elements of type Expr and prove it decreases with each invocation.

Up to this point, we need to have two different relations just to construct Danvy's machine in *Agda*: one to prove that decomposition terminates, one to prove that iterating the one step function terminates. Surprisingly, the optimization that Danvy applies to the machine, *refocusing*, which removes any intermediate expression by

fusing the decomposition and recomposition steps, makes its more amenable to construct in *Agda*. Indeed, it is a variation of our tail-recursive evaluator with one difference: our one step function contracts several redexes at once while Danvy's contracts only one at a time. In the next paragraph, we explore the ramifications of modifying our one-step function to match Danvy's abstract machine.

**Fine-grained reduction** Our tail-recursive evaluator, tail-rec-eval, iterates the function unload until it completely consumes the input expression and returns the result of the fold. We can argue that unload completes an excessive amount of work: while traversing the stack in search for the next subexpression, it might perform *several reductions* before dispatching a call to load or returning a value. Figure 3.5 shows an example; the function unload, starting from the configuration corresponding to the leaf Val *1*, traverses the *spine* at once while accumulating and reducing all partial results.

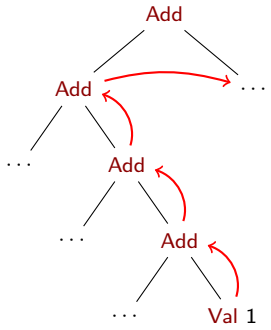


Figure 3.5: unload traverses the spine of an expression

We should be able to rewrite the tail-recursive evaluator, such that iterates a function that performs at most one reduction at a time. The evaluator would match more closely the concept of an abstract machine designed as single step transition system, but, as we will see, it would also increase the complexity of the construction.

There is one fundamental idea in the definition of our tail-recursive evaluator: the intermediate states, or configurations of the abstract machine, always represent locations of leaves in the input expression. If unload is implemented not to consume the spine at once, we will have to reconsider what constitute a valid configuration; aside from leaves, a not yet contracted redex will also be a possible internal state:

```
data Config : Set where
  Leaf  : ℕ → Stack⁺ → Config
  Redex :  (n : ℕ) → (e₁ : Expr) → eval e₁ ≡ n
        → (n′ : ℕ) → (e₂ : Expr) → eval e₂ ≡ n′
        → Stack⁺ → Config
```

The leaves of the input expression remain the same as before: a natural number and the stack pointing to its position. The new constructor, Redex, represents a *redex* that is ready to be reduced. The definition of the function unload clarifies its purpose:

```
unload : (n : ℕ) → (e : Expr) → eval e ≡ n → Stack⁺ → Config ⊎ ℕ
unload n e₁ eq (Left e₂ stk)          = load e₂ (Right n e₁ eq stk)
```

```
unload n e₁ eq₁ (Right n′ e₂ eq₂ stk) = inj₁ (Redex n′ e₂ eq₂ n e₁ eq₁ stk)
unload n _ _ Top                      = inj₂ n
```

In the second clause, instead of recursing over the stack and applying _+_ to $n$ and $n'$, the function unload returns the Redex immediately. The function step will be, in this case, the responsible of triggering the reduction:

```
step : Config → Config ⊎ ℕ
step (Leaf n stk)
  = unload n (Val n) refl stk
step (Redex n e₁ eq₁ n′ e₂ eq₂ stk)
  = unload (n + n′) (Add e₁ e₂) (cong₂ _+_ eq₁ eq₂) stk
```

The key ingredient to build our tail-recursive evaluator was a well-founded relation that decreases with every invocation of step. Accordingly, we will have to find a suitable relation over elements of type Config (we omit the type-indexed relation for the sake of the argument), prove it is well-founded, and show it decreases with step. For most of it, the relation can be defined as _<_: comparing two leaves or redexes in a common subexpression is done inductively; comparing them if one is located on the left subexpression and the other on the right constitutes the base case. However, two more situations will need to be considered:

- Between two redexes, how do we determine which one is smaller if both belong to the same spine?
- Between a redex and a leaf, how do we encode that the leaf is bigger, if it is located at the end of the spine where the redex stands?

The definition of the type Config, increases the diversity of possibilities that have to be dealt with, thus the complexity of functions and proofs. In overall, we are trading a simple formulation that takes advantage of the fact that the function unload$^+$ provably terminates—it is defined by structural recursion over the stack—for a more complicated one that demands explicit evidence of the termination.

In this part of the thesis, the main objective is not just to develop a tail-recursive evaluator for binary trees, but, to prepare the stage for the generic solution that we further present in chapter 4. The simplicity of our approach pays off, as it later will become clear that it has a straightforward generalization. However, it is not clear how changing the function unload$^+$ to one, that reduces at most one redex at a time, fits in the construction as a whole, nor how it scales to the generic case. Certainly, it should be possible but we have not explored such direction any further.

# 4 A verified generic tail-recursive catamorphism

In the previous chapter, chapter 3, we showed how to *manually* construct a tail-recursive evaluation function for the type of binary trees, and prove that is both terminating and equal to the original fold.

In this chapter, we build upon this work and we define a terminating tail-recursive function that we prove equivalent to any fold over any (simple) algebraic datatype that can be generically expressed in the *regular* universe. We begin in section 4.1, recapitulating the idea of *dissection*, due to McBride [2008], and show how it leads (Sections 4.2 and 4.3) to the definition of generic configurations of the abstract machine. Subsequently, in section 4.4, we introduce the generic version of the functions load and unload, which compute one step of the fold. In section 4.5 we set up the relation over generic configurations and present its well-foundedness proof. Finally, in section 4.6, we define the terminating tail-recursive abstract machine as the iteration of the one step function fueled by well-founded recursion. The correctness proof, section 4.7, follows directly from the construction. In section 4.8, we present some examples of the generic tail-recursive fold in action. We conclude this chapter (Section 4.9) discussing some open issues about the construction.

## 4.1 Dissection

The configurations of the abstract machine, which computes the tail-recursive fold for the type Expr, are instances of a more general concept: McBride's dissections [2008]. We briefly recap this construction, showing how it allow us to calculate the type of configurations of the abstract machine that computes the catamorphism of any type expressible in the regular universe (Section 2.3.1).

The key definition of *dissections* is a new interpretation function over the regular universe, $\nabla$, that maps elements of the universe into bifunctors:

```
∇ : (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)
```

Following the metaphor of a functor as a container of things, the reader may find useful to think of its dissection as tearing apart the container in two subcontainers. The elements contained in the left subcontainer do not need to be of the same type as those stored in the right. The $\nabla$ operation applied to a code $R$ : Reg considers all the possible ways in which exactly one of the recursive positions—code I, inhabited by terms of type $X$—is in focus and serves as the breaking point. Because only one variable is specially distinguished, the recursive positions appearing to its left can be interpreted over a different type than those on its right: this is where $\nabla$ differs from a *zipper*.

The last clause of the definition of $\nabla$ 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 right component and pair it with the first interpreted over $X$.

A *dissection* is then formally defined as the pair of the context, resulting from *dissecting* a concrete code $R$, and the missing value that fits in it:

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

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

```
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 dissected value is the right component of the pair, leaving $r$ : $\llbracket R \rrbracket X$ to the left. In such case, it is only possible to construct a term of type $\llbracket R \rrbracket Y$ if we have a function $\eta$ to recover $Y$s from the $X$s contained in $r$.

Using a type-indexed type, we can bundle together a dissection with the value of type $\llbracket R \rrbracket Y$ to which it *plug*s:

```
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
```

## 4.2 Generic stacks

While the *dissection* computes the bifunctor underlying the functorial layer of the generic tree, we still need to take the fixed point of this bifunctor to obtain the type of stacks of the generic abstract machine.

A generic tree, $\mu\ R$, is a recursive structure formed by layers of the functor with code $R$ interpreted over generic trees, $[\![\ R\ ]\!]\ (\mu\ R)$. A dissection calculates how a concrete layer can be decomposed into a subtree in focus and its context, but, on its own it does not account for the recursivity induced by the fixed point. In order to focus on a subtree that may be deeply buried within the generic tree, we need to take a list of dissections, where each element of the list is a particular dissection of the corresponding functorial layer. The type of generics stacks is as follows:

```
Stack : (R : Reg) → (X Y : Set) → Set
Stack R X Y = List (∇ R X Y)
```

Huet's zipper corresponds with instantiating both $X$ and $Y$ to generic trees of type $\mu\ R$. Given such instantiation, we can define a pair of functions that reconstruct the tree by traversing the stack:

```
plug-μ↓ᶜ : (R : Reg) → μ R → Stack R (μ R) (μ R) → μ R
plug-μ↓ᶜ R t []       = t
plug-μ↓ᶜ R t (h :: hs) = In (plug R id (h , plug-μ↓ᶜ R t hs))
plug-μ↑ᶜ : (R : Reg) → μ R → Stack R (μ R) (μ R) → μ R
plug-μ↑ᶜ R t []       = t
plug-μ↑ᶜ R t (h :: hs) = plug-μ↑ᶜ R (In (plug R id (h , t))) hs
```

We can interpret the zipper both as the path starting from the root and descending to the subtree in focus, plug-$\mu\downarrow^C$, or beginning from the position of the subtree and navigating up to the root, plug-$\mu\uparrow^C$. We pass the identity function to plug because the left side of the dissection already stores generic trees.

An abstract machine, which computes the tail-recursive catamorphism, traverses a generic tree from left to right. The stack of such machine is a list of dissections of type $\nabla\ X\ (\mu\ R)$: for each of the subtrees that have been already processed we store a value of type $X$, while we save those that still have to be visited untouched—type $\mu\ R$.

As we did in the concrete tail-recursive evaluator for the type Expr, section 3.2.1, we have to keep extra information in the stack to assist *Agda*'s termination checker and later prove correctness of the construction. For such matter, we define a record type that stores values, subtrees, and the corresponding correctness proofs:

```
record Computed (R : Reg) (X : Set) (ψ : [[ R ]] X → X) : Set where
  constructor _,_,_
  field
    Tree  : μ R
    Value : X
    Proof : Catamorphism R ψ Tree Value
```

**Digression**

Compared to the stack of the tail-recursive evaluator, tail-rec-eval, the type of correctness proofs is not anymore propositional equality, but a custom relation that reifies the function cata:

```
data Catamorphism (R : Reg) (ψ : ⟦ R ⟧ X → X) : μ R → X → Set where
  Cata : {i : ⟦ R ⟧ (μ R)} {o : ⟦ R ⟧ X} → MapFold R ψ R i o
                                         → Catamorphism R ψ (In i) (ψ o)

data MapFold (Q : Reg) (ψ : ⟦ Q ⟧ X → X) : (R : Reg)
       → ⟦ R ⟧ (μ Q) → ⟦ R ⟧ X → Set where
  MapFold-𝟙  : MapFold Q ψ 𝟙 tt tt
  MapFold-I  : {i : ⟦ Q ⟧ (μ Q)} {o : ⟦ Q ⟧ X}
             → MapFold Q ψ Q i o → MapFold Q ψ I (In i) (ψ o)
  MapFold-K  : {a : A} → MapFold Q ψ (K A) a a
  MapFold-⊕₁ : {R P : Reg} {i : ⟦ R ⟧ (μ Q)} {o : ⟦ R ⟧ X}
             → MapFold Q ψ R i o → MapFold Q ψ (R ⨁ P) (inj₁ i) (inj₁ o)
  MapFold-⊕₂ : {R P : Reg} {i : ⟦ P ⟧ (μ Q)} {o : ⟦ P ⟧ X}
             → MapFold Q ψ P i o → MapFold Q ψ (R ⨁ P) (inj₁ i) (inj₂ o)
  MapFold-⊗  : {R P : Reg} {i₁ : ⟦ R ⟧ (μ Q)} {i₂ : ⟦ P ⟧ (μ Q)}
               {o₁ : ⟦ R ⟧ X} {o₂ : ⟦ P ⟧ X}
             → MapFold Q ψ R i₁ o₁ → MapFold Q ψ P i₂ o₂
             → MapFold Q ψ (R ⨂ P) (i₁ , i₂) (o₁ , o₂)
```

The reason for choosing a relation over propositional equality is practical: most of the functions and theorems are inductively defined over the generic code. A datatype indexed by the same code facilitates building proofs for each specific case. Nonetheless, from a value of the reified function we are able to recover the propositional equality proof:

```
MapFold-mapFold : ∀ (Q : Reg) → (ψ : ⟦ Q ⟧ X → X) → (R : Reg)
                → (t : ⟦ R ⟧ (μ Q)) → (x : ⟦ R ⟧ X)
                → MapFold Q ψ R t x → map-fold R Q ψ t ≡ x
```

```
Cata-cata : ∀ (R : Reg) → (ψ : ⟦ R ⟧ X → X) → (t : μ R) → (x : X)
          → Catamorphism R ψ t x → cata R ψ t ≡ x
Cata-cata R ψ ._ (In i) ._ (ψ o) (Cata {i} {o} x)
  = cong ψ (MapFold-mapFold R ψ R i o x)
```

In the rest of this chapter, we use propositional equality to indicate equality whereas in the accompanying code, for every function that is involved in a equality proof, we use a datatype that reifies the call graph of such function.

---

Finally, the type of stacks of the generic abstract machine is defined as a list of *dissections*: on the left we have the Computed results; on the right, we have the subtrees of type $\mu\ R$:

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

Note that the $\mathsf{Stack}^G$ datatype is parametrised by the algebra $\psi$ as the Proof field of the Computed record refers to it.

Given a stack, $\mathsf{Stack}^G$, and the subtree in focus, $\mu\ R$, we define two different *plugging* operations: one top-down, another bottom-up:

```
plug-μ↓ : (R : Reg) → {ψ : ⟦ R ⟧ X → X}
        → μ R → Stack^G 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 → Stack^G 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 pass the projection Computed.Tree as an argument to plug to extract the subtree from the Computed record.

## 4.3 Generic configurations

Recapitulating from the tail-recursive evaluator, tail-rec-eval, the type of configurations of the abstract machine represent locations within the expression that is being evaluated. However, we are not interested in any location within the generic tree, but only on those paths that lead to a leaf. A question, then, is to be asked: what constitutes a leaf in the generic setting?

First, let us recall the two different levels of recursion present in a generic tree:

1. At the functorial layer, because the universe allows functors to be combined: the (co)product of two functors is also a functor.

2. At fixed point level, because positions marked with the constructor I are interpreted over generic subtrees.

It would be troubled to enforce that a leaf is truly non-recursive value, thus, we consider only to be leaves those values of the functor layer that do not contain subtrees, but otherwise might be recursive because of (1).

To describe a generic leaf, we introduce the following predicate:

```
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)
```

Given a value of type $t : \llbracket R \rrbracket X$, the predicate is only true, i.e. we can build a term of type NonRec R t, *iff* it has no occurrences of elements of type $X$.

As an example, in the pattern functor for the Expr type, K ℕ ⊕ (I ⊗ I), terms built using the left injection are not recursive:

```
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 that the predicate NonRec holds for terms using the right injection. The occurrences of recursive positions disallow us from constructing the proof (The type NonRec does not have a constructor such as NonRec-I : $(x : X) \to$ NonRec I $x$).

Now, we define the notion of leaf generically; it is a value of the functor layer that does not have recursive subtrees:[1]

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

A leaf is independent of the type $X$, the predicate NonRec proves it, thus we can coerce it to a different type:

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

The function is defined by induction over the proof NonRec $R\,x$. The case for the code I is eliminated which means we do not have to produce a value of type $Y$ out of thin air.

Just as before, a generic configuration is given by the current leaf in focus and the stack that stores partial results and unprocessed subtrees—or points to it:

```
Config^G : (R : Reg) → (X : Set) → (ψ : ⟦ R ⟧ X → X) → Set
Config^G R X ψ = Leaf R X × Stack^G R X ψ
```

From a configuration of the abstract machine, $\text{Config}^G$, we should be able to recover the input generic tree that is being folded. Crucially, we can embed the value of the leaf into a larger tree by coercing the type $X$ in the leaf to $\mu\ R$. In a similar fashion as in the concrete case, we define a pair of *plugging* functions that recompute the input tree:

```
plugC-μ↓ : (R : Reg) {ψ : ⟦ R ⟧ X → X} → Config^G R X ψ → μ R → Set
plugC-μ↓ R ((I , isI) , s) t = plug-μ↓ R (In (coerce I isI)) s t
plugC-μ↑ : (R : Reg) {ψ : ⟦ R ⟧ X → X} → Config^G R X ψ → μ R → Set
plugC-μ↑ R ((I , isI) , s) t = plug-μ↑ R (In (coerce I isI)) s t
```

Moreover, to ensure that the configurations preserve the invariant—the input tree does not change during the evaluation of the tail-recursive catamorphism—we define a pair of datatypes indexed by the input tree:

```
data Config↓^G (R : Reg) (X : Set) (ψ : ⟦ R ⟧ X → X) (t : μ R) : Set where
  _,_ : (c : Config^G R X ψ) → plugC-μ↓ R c ≡ t → Config↓^G R X ψ t
data Config↑^G (R : Reg) (X : Set) (ψ : ⟦ R ⟧ X → X) (t : μ R) : Set where
  _,_ : (c : Config^G R X ψ) → plugC-μ↑ R c ≡ t → Config↑^G R X ψ t
```

## 4.4 One step of a catamorphism

In this section, we show how to define the generic operations that correspond to the functions load and unload given in section 3.1. Moreover, we outline the proofs of several properties we later require to show correctness and termination.

[1] $\Sigma$ is *Agda*'s type for dependent pair.

### 4.4.1 Load

The function $\mathsf{load}^G$ traverses the input term to find its leftmost leaf. Any other subtrees the $\mathsf{load}^G$ function encounters are stored on the stack. Once the $\mathsf{load}^G$ function encounters a constructor, without subtrees, it is has found the desired leaf. Its definition is as follows:[2]

```
load^G : (R : Reg) {ψ : ⟦ R ⟧ X → X} → μ R
       → Stack^G R X ψ → Config^G R X ψ ⊎ X
load^G R (In t) s = first-cps R R t id (λ l → inj₁ ∘ _,_ l) s
```

We write $\mathsf{load}^G$ by appealing to an auxiliary definition first-cps, that uses continuation passing style (CPS) to keep the definition tail-recursive and obviously structurally recursive. If we were to try to define $\mathsf{load}^G$ by recursion directly, we would need to find the leftmost subtree and recurse on it—but this subtree is not syntactically smaller for the termination checker.

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 → Stack^G Q X ψ → Config^G Q X ψ ⊎ X)
  → Stack^G Q X ψ
  → Config^G 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 are 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 $\mathsf{Stack}^G\ Q\ X\ \psi$ is the current stack.

We shall fill the definition of first-cps by cases. The clauses for the base cases are as expected. In $\mathbb{0}$ there is nothing to be done. The $\mathbb{1}$ and $\mathsf{K}\ A$ 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:

[2] As in the introduction, we use a sum type ⊎ to align its type with that of $\mathsf{unload}^G$.

```
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. 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-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')
```

Using continuations in the definition of first-cps might seem overkill, however, they are necessary to keep the function tail-recursive. We will discuss this issue further at the end of the chapter (Section 4.9).

There is one important property that the function $\mathsf{load}^G$ satisfies: it preserves the input tree. In the concrete case, we proved such property directly by induction over the stack, however, in the generic case we need a more involved construction due to the genericity and CPS nature of the auxiliary function, first-cps. The signature of the property is spelled as follows:

```
load-plug⇑ : ∀ (R : Reg) {ψ : ⟦ R ⟧ X → X} → (r : μ R)
           → (s : Stack^G R X ψ) → (c : Config^G R X ψ)
           → load^G R r s ≡ inj₁ c
           → ∀ (t : μ R) → plug-μ⇑ R r s ≡ t → plugC-μ⇑ R c ≡ t
```

The function $\mathsf{load}^G$ directly calls first-cps, so proving the above lemma amounts to show that it holds for first-cps. However, from its type it is not clear what property we need. We start with the obvious skeleton:

```
first-cps-plug⇑ : (R Q : Reg) {ψ : ⟦ Q ⟧ X → X}
  → (r : ⟦ R ⟧ (μ Q))
  → (k : ∇ R (Computed Q X ψ) (μ Q) → ∇ Q (Computed Q X ψ) (μ Q))
  → (f : Leaf R X → List (∇ Q (Computed Q X ψ) (μ Q)) → Config^G Q X ψ ⊎ X)
  → (s : Stack^G Q X ψ) → (c : Config^G Q X ψ)
  → first-cps R Q r k f s ≡ inj₁ c
  → ∀ (t : μ Q) → □ → plugC-μ⇑ Q c ≡ t
```

Naively, we could try to fill the hole with the type plug-$\mu\Uparrow$ $R\ r\ s \equiv t$, however, the recursive subtrees in $r$ are of type $\mu\ Q$ while the outermost layer is a functor of a different code $R$; the equality does not typecheck. The type of the hole, □, has to relate both continuations, $f$ and $k$, to the value $r$ that is subject to recursion and the stack $s$.

Given a term of type $⟦\ R\ ⟧\ (\mu\ Q)$, for any $R$ and $Q$, there are two possibilities: either the term is a leaf and recursive subtrees do not occur; or the term can be *dissected* into a context and the subtree that fits in it. We express this in *Agda* as a view[Wadler 1987][McBride and McKinna 2004]:

```
view : (R Q : Reg) → {ψ : ⟦ Q ⟧ X → X} → (r : ⟦ R ⟧ (μ Q))
     → (Σ (∇ R (Computed Q X ψ) (μ Q))
          λ dr → Σ (μ Q)
                   λ q → plug R Computed.Tree (dr , q) ≡ r)
       ⊎ (Σ (⟦ R ⟧ X)
            λ leaf → (NonRec R leaf × [ R ]-[ μ Q ] r ≈[ X ] leaf))
```

The value $r : ⟦\ R\ ⟧\ (\mu\ Q)$ either decomposes into a dissection, *dr*, and the subtree *q*, such that plugged together recompose to *r*; or there is a leaf, *leaf*, *equal* to *r*. The variables *r* and *leaf* are not of the same type, thus, we cannot assert they are equal using propositional equality. Instead, we need a different notion of equality: heterogeneous equality. Its definition is as follows:

```
data [_]-[_]_≈[_]_ : (R : Reg) → (X : Set) → ⟦ R ⟧ X
                                → (Y : Set) → ⟦ R ⟧ Y → Set₁ where
  ≈-𝟙  : {X : Set} {Y : Set}                          → [ 𝟙 ]-[ X ] tt ≈[ Y ] tt
  ≈-K  : {A : Set} {a : A} {X : Set} {Y : Set}        → [ K A ]-[ X ] a ≈[ Y ] a
  ≈-I  : {X : Set} {x : X}                            → [ I ]-[ X ] x ≈[ X ] x
  ≈-⊕₁ : {R Q : Reg} {X Y : Set} {x : ⟦ R ⟧ X} {y : ⟦ R ⟧ Y}
       → [ R ]-[ X ] x ≈[ Y ] y → [ R ⊕ Q ]-[ X ] (inj₁ x) ≈[ Y ] (inj₁ y)
  ≈-⊕₂ : {R Q : Reg} {X Y : Set} {x : ⟦ Q ⟧ X} {y : ⟦ Q ⟧ Y}
       → [ Q ]-[ X ] x ≈[ Y ] y → [ R ⊕ Q ]-[ X ] (inj₂ x) ≈[ Y ] (inj₂ y)
  ≈-⊗  : {R Q : Reg} {X Y : Set} {x₁ : ⟦ R ⟧ X} {x₂ : ⟦ R ⟧ Y}
                                 {y₁ : ⟦ Q ⟧ X} {y₂ : ⟦ Q ⟧ Y}
       → [ R ]-[ X ] x₁ ≈[ Y ] x₂ → [ Q ]-[ X ] y₁ ≈[ Y ] y₂
       → [ R ⊗ Q ]-[ X ] (x₁ , y₁) ≈[ Y ] (x₂ , y₂)
```

Two functors with the same code can be interpreted over different types, *X* and *Y*, as long as the code is not I. In that case, constructor ≈-I, both types must coincide. Heterogeneous equality is an equivalence relation as expected:

```
≈-refl  : ∀ {X : Set} {R : Reg} {x} → [ R ]-[ X ] x ≈[ X ] x
≈-sym   : ∀ {X Y : Set} {R : Reg} {x y}
        → [ R ]-[ X ] x ≈[ Y ] y → [ R ]-[ Y ] y ≈[ X ] x
≈-trans : ∀ {X Y Z : Set} {R : Reg} {x y c}
        → [ R ]-[ X ] x ≈[ Y ] y → [ R ]-[ Y ] y ≈[ Z ] c → [ R ]-[ X ] x ≈[ Z ] c
```

In the particular case of both types agreeing it turns into plain propositional equality:

```
≈-to-≡ : ∀ {X : Set} {R : Reg} {x y} → [ R ]-[ X ] x ≈[ X ] y → x ≡ y
```

Continuing with the lemma first-cps-plug$_{\Uparrow}$, we apply the view on the input *r* and for each case define a sensible property:

```
Prop : ∀ (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 → Stack^G Q X ψ → Config^G Q X ψ ⊎ X)
     → Stack^G Q X ψ → μ Q → Set
Prop {X} R Q r k f s t with view {X} R Q r
... | inj₁ (dr , q , _)
```

```
    = Σ (⟦ Q ⟧ (μ Q))
        λ e → plug Q Computed.Tree (k dr , q) ≡ e × plug-μ⇑ Q (In e) s ≡ t
... | inj₂ (l , isl , _) with f (l , isl) s
  ... | inj₁ c = plugC-μ⇑ Q c ≡ t
  ... | inj₂ _ = ⊥
```

When the value can be decomposed into a dissection, $dr$, and a subtree $q$, there exists a tree $e : \mu\ q$, such that applying the continuation $k$ to the dissection and plugging back $q$ results in $e$. Moreover, recursively plugging $e$ to the stack yields $t$. On the other hand, when $r$ is a leaf, $l$, we apply the second continuation $f$, and in case it returns another configuration, $c$, it should plug to the tree $t$.

Using Prop, we can complete the type signature of the lemma first-cps-plug$_{\Uparrow}$. The proof is done by decomposing the input with view, induction on the code, and using properties of heterogeneous equality.

Other properties about how the function load$^G$ behaves follow the same pattern. First, state the property for load$^G$ and, subsequently, for first-cps using the view to differentiate the possible cases.

### 4.4.2 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 $X$, $⟦\ R\ ⟧\ X$.

```
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, $R \bigotimes Q$, that needs another ancillary definition to look for the leftmost occurrence of the variable position within the left component of type $⟦\ R\ ⟧\ X$.

We define the function unload$^G$ 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.

```
unload^G : (R : Reg) → (ψ : ⟦ R ⟧ X → X)
         → (t : μ R) → (x : X) → cata R ψ t ≡ x → Stack^G R X ψ
         → Config^G R X ψ ⊎ X
unload^G R ψ t x eq [] = inj₂ x
unload^G R ψ t x eq (h :: hs) with right R h (t , x , eq)
... | inj₁ r with compute R R r
  ... | (rx , rr) , eq' = unload^G R ψ (In rp) (ψ rx) (cong ψ eq') hs
... | inj₂ (dr , q)     = load^G R q (dr :: hs)
```

When the function right returns a inj$_1$ it means that there are not 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 the variable positions both values, subtrees and proofs.

What is needed for the recursive call, however, is: first, the functor interpreted over values, $⟦\ R\ ⟧\ X$, to apply the algebra; second, the functor interpreted over

subtrees, $[\![\ R\ ]\!]\ (\mu\ R)$, to use the unevaluated subtree for termination; 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 $\text{unload}^G$:

```
compute : (R Q : Reg) {ψ : [[ Q ]] X → X}
  → [[ R ]] (Computed Q X ψ)
  → Σ ([[ R ]] X × [[ R ]] (μ Q)) λ {(r , t) → map-fold Q ψ R t ≡ r}
```

To conclude, we can prove the expected property that $\text{unload}^G$ satisfies: it preserves the input tree:

```
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)
              → unload^G R ψ t x eq s ≡ inj₁ c
              → ∀ (e : μ R) → plug-μ⇑ R t s ≡ e → plugC-μ⇑ R c ≡ e
```

The proof follows directly by induction over the stack.

## 4.5 Relation over generic configurations

We can engineer a *well-founded* relation over elements of type $\text{Config}^G_{\Downarrow}\ t$, 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 *dissection*s 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 $\text{Config}^G$. We can later use the same technique as in section 3.2 to recover a fully type-indexed relation over elements of type $\text{Config}^G_{\Downarrow}\ t$ by requiring that the configurations respect the invariant, $\text{plugC-}\mu_{\Downarrow}\ c \equiv t$. We define inductively the relation over the $\text{Stack}^G$ part of the configurations as follows:

```
data _<C_ : Config^G R X ψ → Config^G 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 $\text{Config}^G$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 *stacks* is different. This means that the paths given by the configurations denote different subtrees of the same node. In such case, the relation we are defining relies on an auxiliary relation, $\llcorner\_\lrcorner\_<_\nabla\_$, that orders *dissections* of type $\mathcal{D}\ R\ (\text{Computed}\ R\ X\ \psi)\ (\mu\ R)$.

We can define this relation on dissections directly; we do not need to consider the recursivity due to the fixed point. The definition of the relation over dissections, interpreted on *any* sets $X$ and $Y$, is the following:

```
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 inductive 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 $\mathsf{inj}_1\ (dr\ ,\ q)$; or a term of type $[\![\ R\ ]\!]\ X \times \nabla\ Q\ X\ Y$ like $\mathsf{inj}_2\ (r\ ,\ dq)$. The former denotes a configuration pointing to the left subtree of the pair while the latter denotes a position in the right subtree. The base-⊗ constructor states that positions to the right are *smaller* than those to 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 *configurations* recursively plug through plugC-$\mu_{\Downarrow}$. Likewise, the auxiliary relation on *dissections*, $\ulcorner\_\urcorner\_<_{\nabla}\_$, 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 ⌞_⌟⌞_⌟_<℃ₛ_ {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 that the first relation is well-founded follows from induction over the code. Like the proof in the relation for expressions, section 3.2.4, it necessitates several lemmas covering each of the constructors. Writing an indexed relation is, again, crucial to prove the lemma. Otherwise, the proof cannot recursively call itself because the inputs are not structurally related.

The proof of *well-foundedness* of $\ulcorner\_\urcorner\_\ulcorner\_\urcorner\_<_{\mathbb{C}_s}\_$, on the other hand, is not as straightforward. The recursive subtrees occurring in the functor layer are not directly accessible, thus, recursive calls are rejected by the termination checker. We tackle this issue in three steps: first, we define a predicate over functors that states whether a property holds for all the values in variable positions; second, we use recursion to build the proof that all such values are well-founded; third, if the property holds, then it also holds for any subtree resulting from a dissection over the value. The three definitions are as follows:

```
data All {A : Set} (P : A → Set) : (R : Reg) → ⟦ R ⟧ A → Set₁ where
  ...

all-is-WF : ∀ (R Q : Reg) (ψ : ⟦ R ⟧ X → X) → (t : ⟦ Q ⟧ (μ R))
          → All (μ R) (λ r → Well-founded (⌞ R ⌟⌞ r ⌟_<℃ₛ_)) Q t
```

```
all-to-plug : ∀ {X : Set} {R Q : Reg} {η : X → μ Q} {P : μ Q → Set}
            → (t : ⟦ R ⟧ (μ Q)) → All (μ Q) P R t
            → ∀ (r : μ Q) (dr : ∇ R X (μ Q)) → plug η R dr r ≡ t → P r
```

The predicate, *All*, is defined by induction over the code. In the particular case of the constructor I, we require a proof that the predicate holds for the concrete value of the type variable. Both lemmas, all-is-WF and all-to-plug, follow directly by induction over the code.

The full proof that the relation is well-founded can be found in the accompanying code. Here we only spell its signature:

```
<ℂ-WF : (R : Reg) → (t : μ R) → Well-founded (⌞ R ⌟ t ⌟_<ℂ_)
```

## 4.6 A generic tail-recursive machine

Finally, we are ready to define a generic tail-recursive machine. To do so we assemble the generic machinery we have developed so far, following the same outline as in section 3.3.

The first point is to write a wrapper around the function unload$^G$ that performs one step of the *catamorphism*:

```
step^G : (R : Reg) → (ψ : ⟦ R ⟧ X → X)
       → Config^G R X ψ → Config^G R X ψ ⊎ X
step^G R ψ ((x , nr-x) , s) = unload^G R ψ (In (coerce x nr-x)) (ψ x) (...) s
```

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 $\llbracket\ R\ \rrbracket\ (\mu\ R)$. Moreover, it supplies a proof, here omitted with elipsis, stating that applying the catamorphism over a *coerced* leaf is the same as directly evaluating the algebra on the leaf, $\psi\ x$. Next, we define a type-indexed step function that statically enforces the configurations, both in its argument and in its result, to be states of the catamorphism over the same generic tree:

```
step^G⇑ : (R : Reg) → (ψ : ⟦ R ⟧ X → X) → (t : μ R)
        → Config^G⇑ R X ψ t → Config^G⇑ R X ψ t ⊎ X
step^G⇑ R ψ t (c , s) with step^G R ψ (c , s) | inspect (step^G R ψ) (c , s)
  ... | inj₁ c | [ ls ] = inj₁ (c , unload-plug^G⇑ ... c t ls ...)
  ... | inj₂ x | _     = inj₂ x
```

The key ingredient of our construction consists of proving that the function step$^G_{\Uparrow}$ delivers a smaller configuration, by $\lfloor\_\rfloor\_\lfloor\_\rfloor\_<_{\mathbb{C}}\_$, each time the function is called. The required lemma has the following signature:

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

As the function step$^G$ is a wrapper over unload$^G$ (Section 4.4), it suffices to prove a similar property for such function. The function unload$^G$ does two things: first,

it calls the function right to check whether the *dissection* has any more recursive subtrees to the right, which still have to be processed; second, it dispatches to either $\mathsf{load}^G$, if there is a subtree left, or recurses over the stack otherwise. In the former circumstance, a new *dissection* is returned by right. Proving that the new configuration is smaller, amounts to showing that the returned *dissection* is smaller by $\llcorner\_\lrcorner\_<_\nabla\_$. The lemma states:

```
right-< : ∀ (R : Reg) (t : μ R) (x : X) (eq : cata R ψ t ≡ x)
            (dr : ∇ (Computed R X ψ) (μ R))
          → (t' : μ R) (dr' : ∇ (Computed R X ψ) (μ R))
          → right R dr (t , x , eq) ≡ inj₂ (dr' , t') → ⌞ R ⌟ (dr' , t') <∇ (dr , t)
```

The proof of this lemma follows by induction over the code.

Extending this result to show that the function $\mathsf{unload}^G$ 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.

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 ⌟_<℃* _) (Config↑-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 x | [ ls ] = rec R ψ t x (rs x (step↑-< G R ψ t c x ls))
... | inj₂ y | [ _ ] = y


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 , ...) (<℃-WF R c)
```

## 4.7 Correctness

The proof that our tail-recursive function produces the same output as the catamorphism is uncomplicated. The function $\mathsf{step}^G$ is type-indexed by the input generic tree which remains constant across invocations, thus, the result of the catamorphism does so as well. As we did in the tail-rec-eval evaluator, section 3.4, we use an ancillary definition indicating that when the result of $\mathsf{step}^G_{\Uparrow}$ is an $\mathsf{inj}_2$, the final value, then it equates to applying the catamorphism to the input:

```
step↑-correct : ∀ (R : Reg) (ψ : ⟦ R ⟧ X → X) (t : μ R)
              → (c : Config↑ G R X ψ t)
                → ∀ (x : X) → step↑ G R ψ t c ≡ inj₂ x → cata R ψ t ≡ x
```

Recall that $\mathsf{step}^G_{\Uparrow}$ is a wrapper around $\mathsf{unload}^G$, 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
```

The correctness of our generic tail-recursive function is an immediate consequence of the above lemmas:

```
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 the construction of the generic tail-recursive evaluator, we show how to use the generic machinery to implement two tail-recursive evaluators: one for the type of Expr from the previous chapter (Chapter 3); and another for a problem dubbed "the balancer of Calder mobiles" by Danvy [2004]. By doing so, we demonstrate how we get a correct-by-construction tail-recursive machine almost for free.

**Expressions** First, we remind the *pattern* functor underlying the type Expr:

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

The type Expr type is 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 : Expr^G → ℕ
eval = cata exprF ψ
```

Finally, we can define a tail-recursive machine *equivalent* to the one we derived in section 3.3, tail-rec-eval:

```
tail-rec-eval^G : Expr^G → ℕ
tail-rec-eval^G = tail-rec-cata exprF ψ
```

**Calder mobiles** We define a Calder mobile inductively as an object of a certain weight or a bar of a certain weight and two sub-mobiles:

```
data Mobile : Set where
  OBJ : ℕ → Mobile
  BAR : ℕ → Mobile → Mobile → Mobile
```

For instance, $m_1$ and $m_2$ are two Mobiles:

```
m₁ : Mobile
m₁ = BAR 1 (BAR 1 (OBJ 2)
                  (OBJ 2))
           (OBJ 5)
m₂ : Mobile
m₂ = BAR 1 (OBJ 6)
           (BAR 1 (OBJ 2)
                  (OBJ 9))
```

The weight of a Mobile is the sum of the weight of its objects and its bars. The following function computes recursively the weight of a Mobile:

```
weight : Mobile → ℕ
weight (OBJ n)         = n
weight (BAR n m₁ m₂) = n + weight m₁ + weight m₂
```

For example, the weight of $m_1$ is 11 and the weight of $m_2$ is 19:

```
prop₁ : weight m₁ ≡ 11
prop₁ = refl
prop₂ : weight m₂ ≡ 19
prop₂ = refl
```

A Mobile is in equilibrium if it is an OBJ, or if it is a BAR and its sub-mobiles are of the same weight and also in equilibrium. The following function determines whether a Mobile is in equilibrium:

```
equil : Mobile → Bool
equil (OBJ _)         = true
equil (BAR _ m₁ m₂) = weight m₁ =ℕ weight m₂ ∧ equil m₁ ∧ equil m₂
```

This solution is highly inefficient because it repeatedly traverses the Mobiles to compute the weight and the equilibrium. In order to reduce the number of traversals we can fuse together the weight and the equilibrium. Before defining another recursive function over Mobile, however, let us express the type in the regular universe:

```
MobileF : Reg
MobileF = K ℕ ⊕ (K ℕ ⊗ I ⊗ I)
Mobile^G : Set
Mobile^G = μ MobileF
```

And the embedding from Mobile into its generic representation:

```
from : Mobile → Mobile^G
from (OBJ n)         = In (inj₁ n)
from (BAR n m₁ m₂)   = In (inj₂ (n , from m₁ , from m₂))
```

Now, we can define a much more efficient solution in terms of performance and code size using the generic tail-recursive evaluator. First and foremost, we define an algebra of the functor MobileF interpreted over Maybe $\mathbb{N}$. A just $n$ denotes that the Mobile is in equilibrium and has weight $n$, while nothing means the Mobile is not in equilibrium. Its definition is as follows:

```
ψ : ⟦ MobileF ⟧ (Maybe ℕ) → Maybe ℕ
ψ (inj₁ n)                       = just n
ψ (inj₂ (n , just m₁ , just m₂)) = if m₁ =ℕ m₂ then just (n + m₁ + m₂)
                                                else nothing
ψ (inj₂ (_ , _ , _))             = nothing
```

We define the tail-recursive function that traverses each Mobile only once using the generic tail-recursive evaluator, tail-rec-cata:

```
equil^G : Mobile → Bool
equil^G = is-just ∘ tail-rec-cata MobileF ψ ∘ from
```

Using the generic equilibrium function, we show that the Mobile $m_1$ is in equilibrium, but, $m_2$ is not:

```
prop₃ : equil^G m₁ ≡ true
prop₃ = refl
prop₄ : equil^G m₂ ≡ false
prop₄ = refl
```

There is still an inefficiency in the code. In case the left sub-mobile is not in equilibrium, it is not necessary to check whether the right is in equilibrium or not. Danvy [2004] proposes to either use exceptions (in *ML*) or transform the evaluator to continuation passing style to overcome this inefficiency.

Unfortunately, the tail-rec-cata function has to traverse the full Mobile term to obtain an answer: we cannot short-circuit the catamorphism at any point using the algebra.

## 4.9 Discussion

In this chapter, we have explained how to derive a generic machine that computes the catamorphism of any algebra over any regular datatype. Adhering to the steps we followed in the concrete case, chapter 3, we derived an abstract machine that we proved to be both terminating and correct. Before concluding the chapter there are some open questions that are worth discussing:

**Choice of universe** The generic tail-recursive machine that we implemented in this chapter works over a rather limited universe. The motivation behind this choice was practical: the universe is expressive enough to implement many simple algebraic datatypes, but, is sufficiently simple to to transport 'directly' the ideas from the concrete example, Expr type, to the generic setting.

Nevertheless, our work is generalizable to other universes. The landmark of every approach to generic programming is to show that is possible to define Huet's notion of *zipper* generically. Because dissections are a generalization of zippers, the steps we follow to construct our generic tail-recursive machine can be taken as a guide to implement terminating and correct-by-construction tail-recursive machines for those universes.

**The function load$^G$ written in continuation passing style** The function load$^G$, as we defined it in section 4.4, uses the ancillary function first-cps to look for the leftmost leaf in the input tree. Such function is defined in continuation passing style, which makes its definition looks overly complicated. However, it is necessary to keep the machine tail-recursive.

The function is defined by induction over the code. When the code is a product of codes, $R \otimes Q$, the input tree has the shape of $(x , y)$ for some $x : [\![ R ]\!]\ (\mu\ (R \otimes Q))$ and $y : [\![ Q ]\!]\ (\mu\ (R \otimes Q))$. There are three possible situations: the value $x$ is not a leaf, $x$ is a leaf but $y$ is not, or $x$ and $y$ are leaves and the product is a leaf itself. In the first case, it is necessary to perform recursion over $x$, while storing $y$ on the stack; in the second case, the recursion is on the right component, $y$, saving $x$ on the stack; and in the last case, there is no recursion involved and the leaf $(x , y)$ is immediately returned. The problem is that checking whether $x$ or $y$ are leaves requires already to perform recursion over them. If the function first-cps was to wait until the result of the recursive call is available to decide which case is met, the function would not be tail-recursive anymore.

**Irrelevance** The generic tail-recursive machine should not have extra runtime impact due to termination and correctness proofs. The inclusion of subtrees and proofs along with Computed values in the stack indeed incur memory overhead during execution. We could use *again* computational irrelevance to identify the parts of the stack not needed during runtime so they are automatically removed. However, it is not clear how to do so in Agda due the narrowness of irrelevance as we previously discussed in section 3.5.

# 5 Conclusions and future work

In this master thesis, we presented the derivation of a generic tail-recursive machine capable of computing the catamorphism of any algebra for any regular datatype. Moreover, we proved our tail-recursive machine to be both terminating and correct with respect to the catamorphism. We formalized the work discussed in this thesis in the dependently typed programing language *Agda*, only using the sound and complete parts of the language: our construction does not require the utilization of any exotic termination flags or extraneous postulates.

We have developed two abstract machines, one for the type of binary trees with natural numbers in the leafs and another for any datatype representable in the regular universe. We have shown how the construction of the latter machine follows the same steps to those of the former. By doing so we tried to motivate the design choices made in the generic setting; it is always hard to reason alone about generic constructions because of their abstract nature. The most complicated part of our development was to find the appropriate predicates and lemmas that allowed us to show termination; once the properties were correctly spelled, they mostly followed by induction over their arguments.The correctness of the tail-recursive machine was immediately obvious from the usage of type-indexed configurations and a type-indexed step function.

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 thesis, 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*.

The use of *Agda* as the formalization language is not casual. Skipping parts of a proof is a standard procedure in hand-written mathematics. However, in a theorem prover such as *Agda* we have to be completely honest: to prove every theorem and lemma we have to reason up to the most concrete detail. In return, we can be certain—as certain as we trust *Agda*'s implementation to be correct—that when a program (or proof) typechecks then it is *mathematically* true. We know once and

for all that the abstract machine terminates and is correct; no amounts of testing can ever provide a more definite and convincing argument. In addition, the type theory underlying Agda is constructive. A theorem can be interpreted as function that transforms inputs into outputs. Within our work, we can appreciate this in the fact that the proof of a relation being *well-founded* implies that we can construct the accessibility predicate mechanically for any term in the domain of the relation.

However, using *Agda* has also its drawbacks. The experience working with it as an interactive proof assistant is far from being ideal: typechecking big modules is rather slow; most the theorems and functions depend on the inductive structure of the input and a simple change in the definition of the datatype results in massive changes to the codebase; irrelevance in Agda is very primitive, for example we cannot have functions that purely work on the irrelevant side, thus, its application is limited.

There are several directions in which this thesis could be further developed. First, the choice of universe. As we mentioned in section 2.3.1 and chapter 4, we chose to build our generic tail-recursive machine in the regular universe because of practical reasons. However, the development presented in this thesis can be taken as a recipe to build tail-recursive machines for other universes. The main insight we provide is: restrict the *zippers*, or configurations of the abstract machine, to leaves of the generic tree; assume that the stack has to be interpreted both top-down or bottom-up depending whether its used for computing or proving; define a suitable relation over configurations that can be proven to be well-founded because is type-indexed with the input tree. Moreover, correctness of the machine follows almost directly from having the type of configurations indexed by the input tree.

The universe of regular types used in this thesis 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 Martin-Löf [1984b]. 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 [Oury and Swierstra 2008, Chapman et al. 2010]. Doing so would allow us to exploit dependent types further in the definition of our evaluators. A long term goal of our work would be to export our development to a generic universe capable of representing well-typed lambda calculus terms, and their evaluation as a simple fold over the syntax. In such environment, we could derive a tail-recursive evaluator automatically, rather than verifying the construction by hand [Swierstra 2012a].

Moreover, it would be worthwhile to explore how to use a well-founded argument to show that other variety of recursion schemes, such as hylomorphisms, histomorphisms, paramorphisms, etc [Meijer et al. 1991], can be turned into provably terminating and correct tail-recursive functions. Another possible path would be to derive a tail-recursive machine equivalent to an effectful fold where the algebra determines the order of the effects involved. A common method to encode effects in pure functional languages is to use monads [Wadler 1998], thus, a monadic fold would be the self-evident choice.

Marking some parts of the code as computationally irrelevant, such as the relation or the proofs, is important to keep the resulting abstract machine both tail-recursive and overhead free. The tail-recursive function that we derived is 'morally' tail-recursive but not practically: to show termination the step function is executed

by the recursor, but its result is then used to show termination before actually recursing on the accessibility predicate. Ideally, the derived machine should have the same runtime impact as if it was implemented in a general purpose functional programming language, such as *Haskell*. At the end of both chapters 3 and 4 we discussed about the shortcomings of using irrelevance directly in *Agda*. However, it should be possible to export our construction to a more mature proof system such as *Coq* where the distinction between the parts of the code used for proving and those used for computing can be clearly separated. We could use the impredicative universe **Prop** for the former while using the predicative universe, **Type**, for the latter. Nevertheless, it is well-known that *Coq* as a theorem prover excels for its capability of using the dependently typed part of the language to prove properties about programs expressed in the simply typed fragment. The generic machinery relies upon dependent types, thus, it is not unambiguous how suitable is Coq for its implementation.

# Bibliography


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

Andreas Abel. foetus – termination checker for simple functional programs, 1998.

Andreas Abel. Miniagda: Integrating sized and dependent types. *arXiv preprint arXiv:1012.4896*, 2010.

Agda. standard library. URL `https://github.com/agda/agda-stdlib`.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 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, pages 8–19, New York, NY, USA, 2003. ACM. ISBN 1-58113-705-2. doi: 10.1145/888251.888254. URL `http://doi.acm.org/10.1145/888251.888254`.

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

Thorsten Altenkirch, Conor Mcbride, and Peter Morris. Generic programming with dependent types. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007.

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

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

Ana Bove and Venanzio Capretta. *Nested General Recursion and Partiality in Type Theory*, pages 121–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44755-9. doi: 10.1007/3-540-44755-5_10. URL `https://doi.org/10.1007/3-540-44755-5_10`.

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

Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN Notices*, volume 40, pages 241–253. ACM, 2005.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863547. URL http://doi.acm.org/10.1145/1863543.1863547.

Nils Anders Danielsson. Operational semantics using the partiality monad. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 127–138, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364546. URL http://doi.acm.org/10.1145/2364527.2364546.

Olivier Danvy. Sur un exemple de patrick greussay. *BRICS Report Series*, 11(41), Dec. 2004. doi: 10.7146/brics.v11i41.21866. URL https://tidsskrift.dk/brics/article/view/21866.

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

Olivier Danvy and Lasse R Nielsen. Refocusing in reduction semantics. *BRICS Report Series*, 11(26), 2004.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, Jul 1994. ISSN 1433-299X. doi: 10.1007/BF01211308. URL https://doi.org/10.1007/BF01211308.

Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 129–146, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48959-7.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, pages 1–71, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76786-2.

Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

Gérard Huet and Inria Rocquencourt France. The zipper. *JFP*, 1997.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2016.

Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation*, 20(3):199–207, 2007.

Simon Marlow, Simon Peyton Jones, et al. The glasgow haskell compiler, 2004.

Per Martin-Löf. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984a.

Per Martin-Löf. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984b.

Conor McBride. 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, pages 287–295, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328474. URL http://doi.acm.org/10.1145/1328438.1328474.

Conor McBride and James McKinna. The view from the left. *Journal of functional programming*, 14(1):69–111, 2004.

Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 324–333. ACM, 1995.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In Christine Paulin-Mohring Jean-Christophe Filliatre and Benjamin Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, Lecture Notes in Computer Science, 2006.

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

Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-060-9. doi: 10.1145/1411318.1411321. URL http://doi.acm.org/10.1145/1411318.1411321.

Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28 (3):605–619, 1988.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

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

Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411213. URL http://doi.acm.org/10.1145/1411204.1411213.

Guy Lewis Steele, Jr. Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 153–162, New York, NY, USA, 1977. ACM. ISBN 978-1-4503-3921-6. doi: 10.1145/800179.810196. URL http://doi.acm.org/10.1145/800179.810196.

Wouter Swierstra. 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.*, pages 163–177, 2012a. doi: 10.4204/EPTCS.76.10. URL https://doi.org/10.4204/EPTCS.76.10.

Wouter Swierstra. From mathematics to abstract machine: a formal derivation of an executable krivine machine. *arXiv preprint arXiv:1202.2924*, 2012b.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313. ACM, 1987.

Philip Wadler. The marriage of effects and monads. In *ACM SIGPLAN Notices*, volume 34, pages 63–74. ACM, 1998.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

Philip Wadler and R John M Hughes. Projections for strictness analysis. In *Conference on Functional Programming Languages and Computer Architecture*, pages 385–407. Springer, 1987.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 233–244, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596585. URL http://doi.acm.org/10.1145/1596550.1596585.