Generalized Algebraic Data Type Exploration

RAY LI, Stanford University, USA

In this writeup, we seek to describe the pros, cons, and potential use cases of Generalized Algebraic Data Types (GADTs). We begin by introducing the basics of GADTs, followed by a literature review of relevant use cases and theoretical machinery for GADTs. We consider the performance benefits of GADTs by examining their use in practice and running several benchmark tests that show how, in very simple programs, GADTs achieve a significant performance improvement over programs without GADTs. We finish with a discussion on the limitations of GADTs and give a concrete example where GADTs seem to limit a program's power. In our appendices, we include relevant code samples, include solutions to a few exercises from a relevant book chapter.

Overall, we find that GADTs can be a valuable tool for ensuring correcting and achieving performance gain over large amounts of *simpler data*, while being more difficult to use for *more complex data*.

ACM Reference Format:
Ray Li. 2017. Generalized Algebraic Data Type Exploration. 1, 1 (December 2017), 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contents

Author's address: Ray Li, Stanford University, Stanford, California, USA, rayyli@stanford.edu.

XXXX-XXXX/2017/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 A simple example

Generalized Algebraic Datatypes (GADTs) were introduced independently by Cheney and Hinze [1] and Xi, Chen and Chen [2]. GADTs are a generalization of Algebraic Data Types (ADTs). In an Algebraic Data Type, we define a type using its constructors. For example,

```
type 'a list =
  Nil : 'a list
| Cons : 'a * 'a list -> 'a list
```

In this example, the outputs of the constructors all have the same type. In a GADT, the constructors can return values of different *types*.

To make `list` into a GADT, we index it by it's length, so that the length of the list is part of the type. In order to accomplish this, we establish types for natural numbers.

```
type z = Z : z
type 'n s = S : 'n -> 'n s
```

Now we can define a length-indexed GADT for lists, which we call `vec`.

```
type ('a, _) vec =
| VNil : ('a, z) vec
| VCons : 'a * ('a, 'n) vec -> ('a, 'n s) vec
```

Note that the values `VNil : ('a, z) vec` and `VCons(3, VNil) : (int, z s) vec` are incompatible, as they are different lengths.

GADTs are a way to enable richer typing in variables that gives the programmer greater control over a program's data. As a simple example, consider the following functions, which return the first and remaining elements of a list, respectively.

```
  val hd : 'a list -> 'a
  val tl : 'a list -> 'a list
```

Note that, for either of these function calls to be valid, the input list must be nonempty. However, there is nothing in the type signature of `hd` and `tl` that guarantee this. Consequentially, the implementation of `hd` and `tl` either raises runtime errors:

```
# List.hd [] ;;
Exception : Failure ‘‘hd’’ .
```

or requires the use of Options:

```
# List.hd [] ;;
- : 'a option = None
```

We can alternatively write this with GADTs.

```
val vhd : ('a, 'n s) vec -> 'a
val vtl : ('a, 'n s) vec -> ('a, 'n) vec
let vhd : type a n. (a, n s) vec -> a = function VCons(hd, _) -> hd
let vtl : type a n. (a, n s) vec -> (a, n) vec = function VCons(_, tl) -> tl
```

GADTs give us several benefits here. First, the signatures of the functions `vhd` and `vtl` tell us that the functions require *nonempty* lists as inputs. If we try to pass in an empty list `VNil` into either function, we will get a compile error. Additionally, the signature of `vtl` tells us the output of the function is a list of type `(a, n) vec`, which means it has one less element than the input.

As this simple example shows, GADTs can give us greater guarantees of program correctness (this may also go by the word *security* in the literature) during the *typechecking* phase. This is in contrast to the implementations `hd` and `tl` for `list`, whose signatures don't ensure anything about the output and could fail at runtime.

Additionally, notice that implementations of `hd` and `tl` for `list` would require an extra pattern matching step to account for the input list being empty. On the other hand, our implementations of `vhd` and `vtl` do not need to because the typechecking guarantees that the inputs have positive length. It turns out this will be reflected appropriately in the compiled OCaml, meaning that the `vec` implementation will also have a performance gain over `list` for these two functions.

1.2 Outline of this writeup

In this writeup, we explore the pros and cons of using GADTs. In §2, we survey some relevant literature on OCaml. In §3, we discuss some the performance tradeoffs in using GADTs. In §4, we discuss some of the disadvantages of using GADTs, and give a detailed discussion on one in particular, through the example of building a safe "reverse" function for `vec`. We also worked through explicit exercises in [3], which we include in Appendix B.

2 GADTS IN PRACTICE AND THEORY

GADTs have recently been making their way in modern functional languages such as OCaml [4] and Haskell [5]. GADTs have found a variety of real world applications guaranteeing runtime and memory performance as well as safety [6, 7]

One major advantage to using GADTs is that they allow the programmer to specify properties about programs using types. The programming concept derives from the so-called *Curry-Howard Isomorphism* [8, 9], which gives a direct correspondence between computer programs and mathematical proofs [10]. In this correspondence, types are identified with propositions and programs are identified with proofs. GADTs allow the programmer to encode propositions into the types of data, that are guaranteed to hold through the proof of the program. Examples of propositions include guarantees that lists or trees are nonempty, guarantees that lists are *sorted* for sorting programs, or guarantees that *device drivers* always obey correct protocols when interacting with hardware [5].

Sheard [5] shows how a number of GADT programming patterns were used in the language Ωmega.

A series of lecture notes [3] gives examples of a few simple data types that encode proofs. They introduce the `(a,b,c) max` type, which encodes that the maximum of natural numbers `a` and `b` is `c`, as well as `(a,b) eq`, which encodes `a=b`.

McBride [11] shows how to simulate *dependent types* using GADTs.

Often the proofs given by the Curry-Howard correspondence crucially require *singleton types*, i.e. types that only have one elements, and examples are considered by Eisenberg and Weirich [12].

3 PERFORMANCE BENCHMARKING

3.1 Benchmark 1: Trees

We ran a benchmark of GADTs performance on a few basic operations on *completely balanced trees*, given in [3]:

`top, incr, depth, swivel, zipTree`,

where `top` retries the top element of a tree, `incr` increments the top element of the tree by 1, `depth` returns a natural number for the depth, `swivel` performs a vertical reflection of the tree, and `zipTree` takes two trees of type `'a tree` and `'b tree` of the same shape and makes a new tree of pairs `('a * 'b) tree` by pairing the corresponding elements of the trees. We ran the following tests, with `n_iter = 50000000`.

(1) Run `top` on a depth-1 tree `n_iter` times
(2) Run `top` of four `incrs` on a depth-1 tree `n_iter` times
(3) Run `depth` on a depth-2 tree `n_iter` times
(4) Run `top` of four `incrs` on a depth-8 tree `n_iter` times
(5) Run `top` of four `swivels` on a depth-8 tree `n_iter/100` times
(6) Run `top` of `zipTree` on a depth-8 tree `n_iter/100` times

Below are our results, we compare the runtime of a regular tree `tree` against a tree GADT `gtree`. For the implementation details of `tree, gtree`, and the benchmarking code, see Appendix A.

Expectations We expected that `gtree` will run faster when its pattern matching is simpler than the corresponding implementation of `tree`. For example, for `topG`, as the input tree is of type `(a, n s) gtree`, we don't have to pattern match against the `EmptyG` case [3]. The same is true for the `incrG`. To see why `zipTreeG` should be faster, note that the GADT only has to pattern match against the *two* cases when the two inputs are both empty and both nonempty, while the regular tree must pattern match against *four* cases, checking two of the potentially invalid tree pairs. Our code also requires `zipTree` for regular trees to do a second pattern matching after the recursive iterations are completed, checking that the recursive iterations are valid, while the `zipTreeG` code does not need to do this as it knows the trees are the same shape.

We expected that for `depthG` and `swivelG`, where there do not seem to be any savings in GADT pattern matching, there is little difference between the performances of `tree` and `gtree`.

Results

Test #	Description	tree	gtree
1	`top`	0.886s	0.015s
2	`top incr^4`	1.412s	0.621s
3	`depth`	0.743s	1.495s
4	`top incr^4` big	1.304s	0.542s
5	`top swivel^4` big	2.784s	2.567s
6	`top zipTree` big	3.224s	1.282s

Analysis As expected, `gtree` is noticeably faster on Tests 1,2, and 4, which only use `top` and `incr`, as well as on Test 6, which uses `zipTree`. Also as expected, `tree` and `gtree` are comparable on test 5, which tests `swivel`.

To our surprise, `depthG` was slower than `depth` by a nontrivial amount. We conjecture that the reason is due to the slow manipulation of user defined natural numbers `S(S(Z))` versus ints, but didn't have time to verify this.

3.2 Benchmark 2: Vectors

For completeness, we also ran a benchmark performance of GADTs on a few basic operations on our `vec` GADT.

(1) Run `hd` 10^8 times.
(2) Sum the elements of a length 1000 `vec` or `list` 10^5 times.
(3) Sum the elements of a length 5 `vec` or `list` 10^8 times.

Our results are in the table below.

Test #	Description	list	vec
1	`hd` $\times 10^8$	0.218s	0.065s
2	`sum` length 1000 $\times 10^5$	0.188s	0.400s
3	`sum` length 5 $\times 10^8$	1.894s	1.609s

For implementation details of `vec`, see Appendix A.

As expected, because of better pattern matching, `hd` was faster for `vec`. Perhaps for similar reasons, `vec` may was slightly faster for summing lists of length 5. Perhaps surprisingly, `vec` was much slower for summing lists on lists of large size. We conjecture this is because the added length-indexed type constraints complicated the way `vec` is stored in memory, especially so when `vec` is large, and this may contribute to the performance hit, but we did not have time to verify this.

3.3 Real world example

We also found an example of the performance benefits of GADTs at scale. The company Issuu [7] used GADTs to speed up the processing of large amounts of JSON data. Just as in our toy examples, runtime savings are achieved by more efficient pattern matching:

> all pattern matching takes place after the first argument is received, and millions of result rows can then be printed without the overhead of doing the same pattern matches again for every row. [7]

Parsing JSON was also considered as a practical example where GADTs achieve performance improvement in a series of lecture notes [3].

3.4 Conclusion

Overall, we found that GADTs do achieve a performance advantage in many use cases, but slower in others, such as large lists. Furthermore, online articles such as [7] and [6] suggest that the extra type information of GADTs gives performance improvements for polymorphic inputs of known structure such as JSON or polymorphic lists. Together, these findings suggest that, while GADTs do not achieve universal performance improvement, there are many practical examples, perhaps especially when the data is simple, where GADTs perform better than the corresponding vanilla data type.

4 WHEN NOT TO USE GADTS

As we have seen, GADTs provide the programmer with a number of valuable benefits - stronger type safety and several very simple examples of better performance. However, as some articles [7, 13] illustrate, GADTs are not always helpful in practice. We outline several of the disadvantages of using GADTs below. Following the list, we elaborate on the last item, giving a concrete example of how GADTs can make programming cumbersome and in fact less efficient.

(1) "You cannot use record selectors on GADT values. You can declare GADT records, but you must extract values through pattern matching." [13]

(2) "Record fields cannot share names unless they also share a return type in the GADT constructor. This is not really an extra constraint, in a sense, because you could not do that with ADTs anyway, where all constructors shared a single return type. It is still annoying, though," [13]
(3) "Pattern matching on a GADT (the only way to extract information) requires the -XGADTs extension to be enabled at the pattern match site. This means that the implementation detail (that you used GADTs) leaks out to all clients of the code." This is a reason against designing public APIs around GADTs. [13]
Put another way, every type parameter in every module that uses a GADT must be explicitly passed around. [7]
(4) Larger data types, like long lists or large complicated JSON values (as opposed to many small lists or many small JSON values), may be less efficient with GADTs. See, for example Test 2 (`fold` of large list) in Benchmark 1 or Test 3 (`depth`) in Benchmark 2. Note also declaring large data types can be quite cumbersome. It does not seem possible to define a function `to_nat: int -> nat` that takes in a nonnegative integer and outputs one of our singleton natural numbers - such a function would output values of different types for different inputs! Consequently, to, for example, construct a `vec` of length 1000, we first need to construct the natural number 1000, which is done by writing `thousand = S(S(..S(Z)..))`.
(5) More complex data types require more complex proofs in types to ensure proper type inference.
In the words of the Issuu engineering team,

> [GADTs] immediately set us on a slippery slope of needing increasingly advanced type-system features. Just to recover basic functionality like parsing and pretty-printing, we needed existential types and polymorphic recursion. In other places in our codebase, we need advanced design patterns such as type-equality witnesses and various flavors of heterogeneous lists [7].

For the remainder of this section, we explicitly discuss Item (5). In doing so, we discuss some of heavy type-machinery needed to implement a `rev` (reverse) function for `vec`, and consider the performance and practicality of such an implementation.

4.1 The challenge: "rev" for "vec"

Recall the definition of `vec`.

```
type ('a, _) vec =
| VNil : ('a, z) vec
| VCons : 'a * ('a, 'n) vec -> ('a, 'n s) vec
```

Our goal is to implement a `rev` function that reverses a `vec` and returns the result. Naturally, the function rev has the following signature:

```
val rev : ('a, 'n) vec -> ('a, 'n) vec
```

A clear requirement encoded in this signature is that the input and output vectors of `rev` have the same length. However, ensuring this with GADTs is quite challenging, as we shall see.

Typically, reversing a list in a functional language is implemented by writing a helper function that builds a new list accumulating the elements of the list in reverse order. Naturally, we could follow the same procedure. We could use a `(_,_,_) add` to maintain the length of the vectors in our helper function.

```
type (_, _, _) add =
| AddZ : 'a -> ('a, z, 'a) add
| MoveS : ('a s, 'b, 'c) add -> ('a, 'b s, 'c) add

let rec add : type n m l. (n, m, l) add -> l = function
| AddZ x -> x
| MoveS x -> add x

let rev : type a n. (a, n) vec -> (a, n) vec = function
  | v ->
    let rec helper : type a n m l. (a, n) vec * (a, m) vec * (n,m,l) add
      -> (a, l) vec = function
      | VNil, acc, s -> acc
      | VCons (hd, tl), acc, s -> (helper (tl, VCons(hd, acc), (MoveS s)))
    in
    let len_v = len(v) in
    helper (v, VNil, AddZ(len_v))
```

The type `n` tracks the number of elements we have left in our list to reverse, and `m` tracks the number of elements we have accumulated, maintaining the invariant that `n` and `m` add to the input vector. The problem arises at the end: when our list reaches `VNil`, we return all the elements we have accumulated. However, it seems that our type constraints cannot *prove* that `acc` has the same length as the original input. We may start with an addition term of the form `(n, z, n) add`, and end with a term of the form `(z, m, n) add`. However, our propositions-as-proofs deductions don't allow to conclude from `(z,m,n) add` that `m=n`. Intuitively this seems trivial as we all know $0+m=n$ implies $m=n$, but from our inductive definition of propositions built from `AddZ` and `MoveS`, it's not clear that we could allow the type-checker to make such conclusions. Indeed, the OCaml compiler points to the line

```
      | VNil, acc, s -> acc
```

and gives the following error:

```
Error: This expression has type (a, m) vec
       but an expression was expected of type (a, l) vec
       Type m is not compatible with type l
```

One way to get around this seems to be to use existential types, but then we still don't have any guarantees on the length of the output of `rev`, which is what we were trying to ensure in the first place.

The articles such as [11], [12] that consider our example of vectors conveniently avoid an implementation of `rev`. As my exploration revealed, this was perhaps with good reason. It seemed to suggest that the implementing the `rev` function with GADTs is both (1) more complicated and (2) still comes at a cost to performance.

Before continuing, we make a minor digression to say a bit more about what this roadblock reveals about the limitations of the Curry-Howard correspondence. The challenge seems to come from a difference between "forward reasoning" and "backwards reasoning." Concretely, one way to prove that $0+m=n$ implies $m=n$ might be *proof by contradiction*. We start with $0+m=n$, and then stepping backwards we may end up with something of the form $0+m'=0$ or $0+0=n'$, and under our propositions $(n+0=n, (n+1)+m=l \implies n+(m+1)=l)$, these are invalid identities unless $m'=n'=0$ (this may take some work to show too). Of course, in this case, it may be possible to frame such a proof by contradiction as a "forward" proof that doesn't use contradiction, and indeed this is what [17] seems to do in Haskell.

However, in general, proofs by contradiction cannot be made "forward" [15, 16], and it seems like the "propositions as proofs" paradigm of Curry Howard does not cleanly allow for proofs that reason by contradiction. Put another way, the Curry-Howard isomorphism maps programs to proofs in an injective way, and perhaps for an appropriate definition of "proof" this map is also surjective. However, perhaps for a broader class of proofs this map is not surjective, meaning there are some proofs which do not admit programs, and it seems that proofs by contradiction are not always included in the set of proofs admitting programs.

4.2 Fix 1: Slow reversing

One workaround is to use a more naive implementation of the reverse function.

```
let rec rev_slow : type a n. (a, n) vec -> (a, n) vec = function
  | VNil -> VNil
  | VCons(hd, tl) ->
    let rec rapp : type a n. a -> (a, n) vec -> (a, n s) vec = function x ->
      (function
        | VNil -> VCons(x, VNil)
        | VCons(hd, tl) -> VCons(hd, rapp x tl))
    in
    rapp hd (rev_slow tl)
```

However, this code runs in time $O(n^2)$, where n is the length of the list. This seems like an inconvenient price to pay to guarantee the invariance of our list size, so an alternative solution would be preferable.

4.3 Fix 2: Proofs in types (still slow!)

The goal of our original approach was to try to encode an inductive proof that the output of `helper (VNil, acc, (z, m, l) add)` had size `l` (we know it has size `m`). This seems to be possible: Eisenberg shows how to so using Haskell in three successive lectures (see Lectures 13, 14, and 15 of [17]). However, after trying for several hours and with the help of additional sources such as [18], we were unable to convert this to OCaml.

Furthermore, even if we imagined that we successfully implemented a type-checked reverse function using the "propositions as types" correspondence and constructing a proof using types that the output has the correct length, there is still a problem: while the reverse function takes $O(n)$ "steps" (we use the word "steps" vaguely), the *proof* of length-correctness would take quadratic time! Indeed, the proof that $0 + m = n$ implies $m = n$ (or, in the case of the lecture notes, $0 + m = m$) requires building up addition statements inductively over all terms of the form $m_1 + m_2$ where $m_1 + m_2 \leq n$, of which there are $O(n^2)$. I don't have any kind of formal justification that such a programmatic proof requires $\Omega(n^2)$ lines of reasoning, but it did in [17], and proving such a result seems an interesting research direction, assuming nobody has already proved or disproved it. At the same time, it seems very counterintuitive that a simple function such as `rev`, with a natural constraint that the input and output vectors have the same length, would require more than linear time in this reasonable model of computation.

5 CONCLUSION

In this writeup, we considered Generalized Algebraic Data Types (GADTs) and saw several examples of how GADTs can provide greater guarantees of security and correctness as well as improve program performance. At the same time, we saw how the stricter type constraints introduced by GADTs requires us to introduce heavier type machinery and can make programs complex and potentially even cost us performance. Overall, we find that GADTs can be a valuable tool for ensuring correcting and achieving performance gain over large amounts of *simpler data*, while being more difficult to use for *more complex data*.

5.1 A bit of reflection

The prompt asks us to answer "How successful were you at achieving your goals" so I answer it now.

I felt like I got my hands dirty with several concrete examples of GADTs while also reading a lot about them, and obtained some intuition for the practicality and limitations of GADTs. Thus, overall I consider this a successful project.

Admittedly, I felt unable to learn all of the relevant PL background to understand the full spectrum of trade-offs (you'll noticed that I glossed over some of the details in §4), but I imagine that is what it means to be an "expert" in the area. Also, I had a hard time grasping some of the more technical material like all of the type machinery introduced in [17], as evidenced by my lack of "efficient" implementation for the `rev` function for `vec`. However, the time spent there was not wasted, as it revealed a major challenge in using GADTs that I otherwise may have overlooked.

REFERENCES

[1] James Cheney and Ralf Hinze. First-class phantom types. 2003.

[2] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235, 2003.

[3] Anil Madhavapeddy and Jeremy Yallop.

[4] Ocaml 4.00.1. [Online; accessed 11-December-2017].

[5] Tim Sheard. Putting curry-howard to work. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, pages 74–85, 2005.

[6] Yaron Minsky. Why gadts matter for performance, March 2015. [Online; accessed 11-December-2017].

[7] Jonas B. Jensen, Anders Fugmann, and Mads Hartmann Jensen. Practicalities and trade-offs in programming with gadts, September 2015.

[8] Haskell Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20:584–590, 1934.

[9] William A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[10] Wikipedia. Curry-howard correspondence — Wikipedia, the free encyclopedia, 2017. [Online; accessed 11-December-2017].

[11] Conor McBride. Faking it: Simulating dependent types in haskell. *J. Funct. Program.*, 12(4&5):375–392, 2002.

[12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 117–130, 2012.

[13] Gadts in api design, April 2013. [Online; accessed 11-December-2017].

[14] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 81–92, 2013.

[15] sonicboom (https://math.stackexchange.com/users/46347/sonicboom). Can every proof by contradiction also be shown without contradiction? Mathematics Stack Exchange.

URL:https://math.stackexchange.com/q/243770 (version: 2012-11-26).
[16] Timothy Gowers. When is proof by contradiction necessary?, March 2010.
[17] Richard Eisenberg. Lecture notes for cs 380: Modern functional programming, 2017. Lectures 13, 14, 15.
[18] NicholasT (https://gist.github.com/NicolasT). nel.ml. Github.

A CODE: PERFORMANCE BENCHMARKING

A.1 Tree code

```
type 'a tree =
  | Empty : 'a tree
  | Tree : 'a tree * 'a * 'a tree -> 'a tree

let incr : int tree -> int tree = function
| Empty -> Empty
| Tree (l, v, r) -> Tree (l, v+1, r)

let mx : int -> int -> int = function a -> (function b -> if a > b then a else b)

let rec depth : 'a . 'a tree -> int = function
| Empty -> 0
| Tree ( l ,_, r ) -> 1 + mx ( depth l ) ( depth r )

let top : 'a . 'a tree -> 'a option = function
Empty -> None
| Tree (_, v ,_) -> Some v

let rec swivel : 'a . 'a tree -> 'a tree = function
Empty -> Empty
| Tree ( l , v , r ) -> Tree ( swivel r , v , swivel l )

(* gtree *)
type ( 'a , _) gtree =
    EmptyG : ( 'a , z ) gtree
    | TreeG : ( 'a , 'n) gtree * 'a * ( 'a , 'n) gtree -> ( 'a , 'n s ) gtree

(* gtree functions *)

let incrG : type n . ( int , n s ) gtree -> ( int , n s) gtree=
function TreeG ( l, v , r) -> TreeG(l, v + 1, r)

let rec depthG : type a n . ( a , n) gtree -> n =
function
EmptyG -> Z
| TreeG ( l ,_,_) -> S (depthG l )

let topG : type a n . ( a , n s ) gtree -> a =
function TreeG (_, v ,_) -> v

let rec swivelG : type a n . ( a , n) gtree -> (a , n) gtree =
function
EmptyG -> EmptyG
| TreeG ( l , v , r ) -> TreeG ( swivelG r , v , swivelG l )
```

A.2 Tree benchmarking code

```
(* benchmarking *)
let n_iter = 50000000;;

(* top *)
let benchmark_top_tree_1 () =
  let t = Tree(Empty, 2, Empty) in
  for _ = 1 to n_iter do
    assert(top(t) = Some(2));
  done;;

let benchmark_top_gtree_1 () =
  let t = TreeG(EmptyG, 2, EmptyG) in
  for _ = 1 to n_iter do
    assert(topG(t) = 2);
  done;;

let benchmark_top_tree_2 () =
  let t = Tree(Empty, 2, Empty) in
  for _ = 1 to n_iter do
    assert(top(incr(incr(incr(incr(t))))) = Some(6));
  done;;

let benchmark_top_gtree_2 () =
  let t = TreeG(EmptyG, 2, EmptyG) in
  for _ = 1 to n_iter do
    assert(topG(incrG(incrG(incrG(incrG(t))))) = 6);
  done;;


let benchmark_top_tree_3 () =
  let t = Tree(Tree(Empty, 3, Empty), 2, Tree (Empty, 4, Empty)) in
  for _ = 1 to n_iter do
    (*assert(top(incr(incr(incr(incr(t))))) = Some(6));*)
    assert(depth(t) = 2)
  done;;

let benchmark_top_gtree_3 () =
  let t = TreeG(TreeG(EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 4, EmptyG)) in
  let two = S(S(Z)) in
  for _ = 1 to n_iter do
    (*assert(topG(incrG(incrG(incrG(incrG(t))))) = 6);*)
    assert(depthG(t) = two)
  done;;
```

```
let benchmark_top_tree_4 () =
  let t = Tree(Tree(Empty, 3, Empty), 2, Tree (Empty, 4, Empty)) in
  let t2 = Tree(t, 5, t)   in
  let t3 = Tree(t2, 5, t2) in
  let t4 = Tree(t3, 5, t3) in
  let t5 = Tree(t4, 5, t4) in
  let t6 = Tree(t5, 5, t5) in
  let t7 = Tree(t6, 100, t6) in
  for _ = 1 to n_iter do
    assert(top(incr(incr(incr(incr(t7))))) = Some(104));
  done;;

let benchmark_top_gtree_4 () =
  let t = TreeG(TreeG(EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 4, EmptyG)) in
  let t2 = TreeG(t, 5, t)   in
  let t3 = TreeG(t2, 5, t2) in
  let t4 = TreeG(t3, 5, t3) in
  let t5 = TreeG(t4, 5, t4) in
  let t6 = TreeG(t5, 5, t5) in
  let t7 = TreeG(t6, 100, t6) in
  for _ = 1 to n_iter do
    assert(topG(incrG(incrG(incrG(incrG(t7))))) = 104);
  done;;

let benchmark_top_tree_5 () =
  let t = Tree(Tree(Empty, 3, Empty), 2, Tree (Empty, 4, Empty)) in
  let t2 = Tree(t, 5, t)   in
  let t3 = Tree(t2, 5, t2) in
  let t4 = Tree(t3, 5, t3) in
  let t5 = Tree(t4, 5, t4) in
  let t6 = Tree(t5, 5, t5) in
  let t7 = Tree(t6, 100, t6) in
  for _ = 1 to n_iter do
    assert(top(swivel(swivel(swivel(swivel(t7))))) = Some(100));
  done;;

let benchmark_top_gtree_5 () =
  let t = TreeG(TreeG(EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 4, EmptyG)) in
  let t2 = TreeG(t, 5, t)   in
  let t3 = TreeG(t2, 5, t2) in
  let t4 = TreeG(t3, 5, t3) in
  let t5 = TreeG(t4, 5, t4) in
  let t6 = TreeG(t5, 5, t5) in
  let t7 = TreeG(t6, 100, t6) in
  for _ = 1 to n_iter do
    assert(topG(swivelG(swivelG(swivelG(swivelG(t7))))) = 100);
  done;;
```

```
let benchmark_top_tree_6 () =
  let t = Tree(Tree(Empty, 3, Empty), 2, Tree (Empty, 4, Empty)) in
  let t2 = Tree(t, 5, t)   in
  let t3 = Tree(t2, 5, t2) in
  let t4 = Tree(t3, 5, t3) in
  let t5 = Tree(t4, 5, t4) in
  let t6 = Tree(t5, 5, t5) in
  let t7 = Tree(t6, 100, t6) in
  for _ = 1 to n_iter/100 do
    assert(top(swivel(swivel(swivel(swivel(t7))))) = Some(100));
  done;;

let benchmark_top_gtree_6 () =
  let t = TreeG(TreeG(EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 4, EmptyG)) in
  let t2 = TreeG(t, 5, t)   in
  let t3 = TreeG(t2, 5, t2) in
  let t4 = TreeG(t3, 5, t3) in
  let t5 = TreeG(t4, 5, t4) in
  let t6 = TreeG(t5, 5, t5) in
  let t7 = TreeG(t6, 100, t6) in
  for _ = 1 to n_iter/100 do
    assert((topG (zipTreeG t7 t7)) = (100,100));
  done;;

let benchmark_top() = (
  Printf.printf "n_iter: %8d\n" n_iter;
  Printf.printf "Testing top_tree_1\n";
    time benchmark_top_tree_1();
  Printf.printf "Testing top_gtree_1\n";
    time benchmark_top_gtree_1();
  Printf.printf "Testing top_tree_2\n";
    time benchmark_top_tree_2();
  Printf.printf "Testing top_gtree_2\n";
    time benchmark_top_gtree_2();
  Printf.printf "Testing top_tree_3\n";
    time benchmark_top_tree_3();
  Printf.printf "Testing top_gtree_3\n";
    time benchmark_top_gtree_3();
  Printf.printf "Testing top_tree_4\n";
    time benchmark_top_tree_4();
  Printf.printf "Testing top_gtree_4\n";
    time benchmark_top_gtree_4();
  Printf.printf "Testing top_tree_5\n";
    time benchmark_top_tree_5();
  Printf.printf "Testing top_gtree_5\n";
    time benchmark_top_gtree_5();
  Printf.printf "Testing top_tree_6\n";
    time benchmark_top_tree_6();
  Printf.printf "Testing top_gtree_6\n";
    time benchmark_top_gtree_6();
  ) ;;


let () = benchmark_top()
```

A.3 Vec code

```
type ('a, _) vec =
| Nil : ('a, z) vec
| Cons : 'a * ('a, 'n) vec -> ('a, 'n s) vec

let head : type a n. (a, n s) vec -> a = function Cons(hd, _) -> hd
let tail : type a n. (a, n s) vec -> (a, n) vec = function Cons(_, tl) -> tl
let rec map : type a b n. (a -> b) -> (a, n) vec -> (b, n) vec = function
f -> (function
      | Nil -> Nil
      | Cons (hd, tl) -> Cons(f hd, map f tl)
)

let rec replicate : type a n. n nat -> a -> (a,n) vec = function
  | NatZ -> (function x -> Nil)
  | NatS(n) -> (function x -> Cons(x, replicate n x))

let rec crush : type a b n. b -> (a -> b -> b) -> (a, n) vec -> b =
  function init ->
    (function f ->
      (function
        | Nil -> init
        | Cons (hd, tl) -> crush (f hd init) f tl))
```

A.4 Vec benchmarking code

```
(* benchmarking *)

let n_iter_1 = 100000000;;
let n_iter_2 = 100000;;
let n_iter_3 = 100000000;;

let benchmark_list_1 () =
  let rec rpt n v =
    if n < 1 then [] else (v :: (rpt (n-1) v))
  in
  let l1 = rpt 5 1 in
  for _ = 1 to n_iter_1 do
    assert(List.hd l1 = 1)
  done;;
```

```
let benchmark_vec_1 () =
  let l1 = replicate (five) 1 in
  for _ = 1 to n_iter_1 do
    assert(head l1  = 1)
  done;;


let benchmark_list_2 () =
  let rec rpt n v =
    if n < 2 then [] else (v :: (rpt (n-2) v))
  in
  let l2 = rpt 1000 2 in
  for _ = 2 to n_iter_2 do
    assert((List.fold_left sum 0 l2) > 0)
  done;;


let benchmark_vec_2 () =
  let l2 = replicate (thousand) 2 in
  for _ = 1 to n_iter_2 do
    assert((crush 0 sum l2) > 0)
  done;;


let benchmark_list_3 () =
  let rec rpt n v =
    if n < 1 then [] else (v :: (rpt (n-1) v))
  in
  let l3 = rpt 5 1 in
  for _ = 1 to n_iter_3 do
    assert((List.fold_left sum 0 l3) > 0)
  done;;


let benchmark_vec_3 () =
  let l3 = replicate (five) 1 in
  for _ = 1 to n_iter_3 do
    assert((crush 0 sum l3) > 0)
  done;;


let benchmark_vec() = (
  Printf.printf "n_iter_1: %8d\n" n_iter_1;
  Printf.printf "Testing list_1\n";
    time benchmark_list_1();
  Printf.printf "Testing vec_1\n";
```

```
    time benchmark_vec_1();
  Printf.printf "n_iter_2: %8d\n" n_iter_2;
  Printf.printf "Testing list_2\n";
    time benchmark_list_2();
  Printf.printf "Testing vec_2\n";
    time benchmark_vec_2();
  Printf.printf "n_iter_3: %8d\n" n_iter_3;
  Printf.printf "Testing list_3\n";
    time benchmark_list_3();
  Printf.printf "Testing vec_3\n";
    time benchmark_vec_3();
  ) ;;

let () = benchmark_vec()
```

B EXERCISES

The following are solutions to Exercises 2, 3, and 4 in **??**, with appropriate tests.

```
(* Exercise 2 *)
let rec unit_gtree_of_depth : type n. n nat -> ( unit , n) gtree = function
  | NatZ -> EmptyG
  | NatS n -> TreeG(unit_gtree_of_depth n, (), unit_gtree_of_depth n)

let unit_gtree_of_depth_tests () =
  assert(unit_gtree_of_depth(NatZ) = EmptyG );
  assert(unit_gtree_of_depth(NatS(NatZ)) = TreeG(EmptyG, (), EmptyG));
  assert(unit_gtree_of_depth(NatS(NatS(NatZ))) = TreeG(TreeG(EmptyG, (), EmptyG), ()
let () = unit_gtree_of_depth_tests ()
(* http://yann.regis-gianas.org/mpri/01-gadts-checking.pdf*)

let test_fn : int -> (int * int) = function a -> (a,a)

(* Exercise 3 *)
(*
 * This function takes as input a natural number n of type 'n nat and returns a tree
 * populated with 0,\dots, 2^n - 1 from left to right
 *)
let int_gtree_of_depth : type n. n nat -> ( int , n) gtree =
  let rec int_gtree_of_depth_helper : type n. n nat -> int -> ((int , n) gtree) * in
    function
    | NatZ -> (function v -> (EmptyG, v))
    | NatS n -> function v ->
    let left_gt, v_1 = int_gtree_of_depth_helper n v in
    let right_gt, v_2 = int_gtree_of_depth_helper n (v_1 + 1) in
    (TreeG(left_gt, v_1, right_gt), v_2)
  in
  function n ->
    let gt, _ = int_gtree_of_depth_helper n 0 in
    gt

let int_gtree_of_depth_tests () =
  assert(int_gtree_of_depth(NatZ) = EmptyG );
  assert(int_gtree_of_depth(NatS(NatZ)) = TreeG(EmptyG, 0, EmptyG));
  assert(int_gtree_of_depth(NatS(NatS(NatZ))) = TreeG(TreeG(EmptyG, 0, EmptyG), 1, T
  assert(int_gtree_of_depth(NatS(NatS(NatS(NatZ)))) = TreeG(
    TreeG(
      TreeG(EmptyG, 0, EmptyG),
      1,
      TreeG(EmptyG, 2, EmptyG)
    ),
    3,
    TreeG(
      TreeG(EmptyG, 4, EmptyG),
      5,
      TreeG(EmptyG, 6, EmptyG)
    )
  ))

let () = int_gtree_of_depth_tests ()


(* Exercise 4 *)
let rec unzipTreeG : type n. ( 'a * 'b , n) gtree -> ( 'a , n) gtree * ( 'b , n) gtr
  function
  | EmptyG -> EmptyG, EmptyG
  | TreeG(l, (x1, x2), r) ->
    let l1, l2 = unzipTreeG l in
    let r1, r2 = unzipTreeG r in
    TreeG(l1, x1, r1), TreeG(l2, x2, r2)

type 'a egtree = E : ('a, 'n) gtree -> 'a egtree

let rec unnestify : type a. a ntree -> a egtree  =
  function
  | EmptyN -> E(EmptyG)
  | TreeN(v, t) ->
    let E(unnestified) = unnestify t in
    let l, r = unzipTreeG(unnestified) in
    E(TreeG (l, v, r))

let unnestify_tests () =
  let nt1 = TreeN (1 , TreeN (( 0 , 2 ) , EmptyN)) in
  let gt1 = int_gtree_of_depth(NatS(NatS(NatS(NatZ)))) in
  let gt2 = int_gtree_of_depth(NatS(NatS(NatS(NatS(NatS(NatZ)))))) in
  assert(unnestify(nt1) = E(
    TreeG(TreeG(EmptyG, 0, EmptyG), 1, TreeG(EmptyG, 2, EmptyG))));
  assert(unnestify(nestify(gt1)) = E(gt1));
  assert(unnestify(nestify(gt2)) = E(gt2))
let () = unnestify_tests ()
```