Chapter 8

Programming with GADTs

ML-style variants and records make it possible to define many different data types, including many of the types we encoded in System Fω in Chapter 2.3.2: booleans, sums, lists, trees, and so on. However, types defined this way can lead to an error-prone programming style. For example, the OCaml standard library includes functions `List.hd` and `List.tl` for accessing the head and tail of a list:

```
val hd : 'a list → 'a
val tl : 'a list → 'a list
```

Since the types of `hd` and `tl` do not express the requirement that the argument lists be non-empty, the functions can be called with invalid arguments, leading to run-time errors:

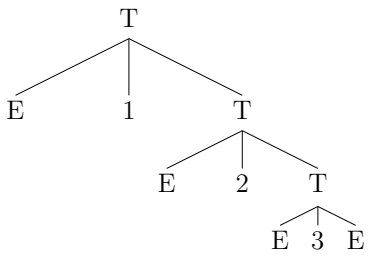
```
# List.hd [];;
Exception: Failure "hd".
```

In this chapter we introduce generalized algebraic data types (GADTs), which support richer types for data and functions, avoiding many of the errors that arise with partial functions like `hd`. As we shall see, GADTs offer a number of benefits over simple ML-style types, including the ability to describe the shape of data more precisely, more informative applications of the propositions-as-types correspondence, and opportunities for the compiler to generate more efficient code.

8.1 Generalising algebraic data types

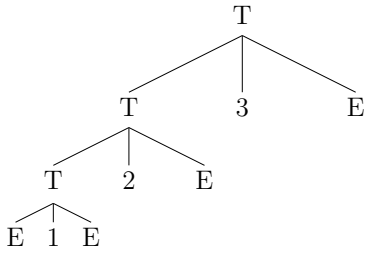
Towards the end of Chapter 2 we considered some different approaches to defining binary branching tree types. Under the following definition a tree is either empty, or consists of an element of type `'a` and a pair of trees:

```
type 'a tree =
  Empty : 'a tree
| Tree : 'a tree * 'a * 'a tree → 'a tree
```



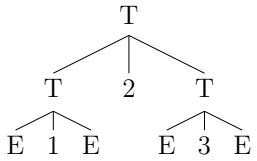

```
Tree
  (Empty,
   1,
   Tree
    (Empty,
     2,
     Tree (Empty, 3, Empty)))
```

Figure 8.1: A right skewed tree value




```
Tree
 (Tree
   (Tree (Empty, 1, Empty),
    2,
    Empty),
  3,
  Empty)
```

Figure 8.2: A left skewed tree value




```
Tree
 (Tree (Empty, 1, Empty),
  2,
  Tree (Empty, 3, Empty))
```

Figure 8.3: A balanced tree value

Using the constructors of tree we can build a variety of tree values. For example, we can build trees that are skewed to the right (Figure 8.1) or to the left (Figure 8.2), or whose elements are distributed evenly between left and right (Figure 8.3).

Alternatively we can give a definition under which a tree is either empty, or consists of an element of type 'a and a tree of pairs[1]:

```
type _ ntree =
  EmptyN : 'a ntree
| TreeN : 'a * ('a * 'a) ntree → 'a ntree
```

The constructors of ntree severely constrain the shape of trees that can be built. Since the element type of each subtree 'a*'a duplicates the element type 'a of the parent , the number of elements at each depth precisely doubles, and

[1]The perfect type of Chapter 2 defined trees with labels at the leaves rather than at the branches. The definition of ntree given here makes it easier to compare the various tree types in this chapter.

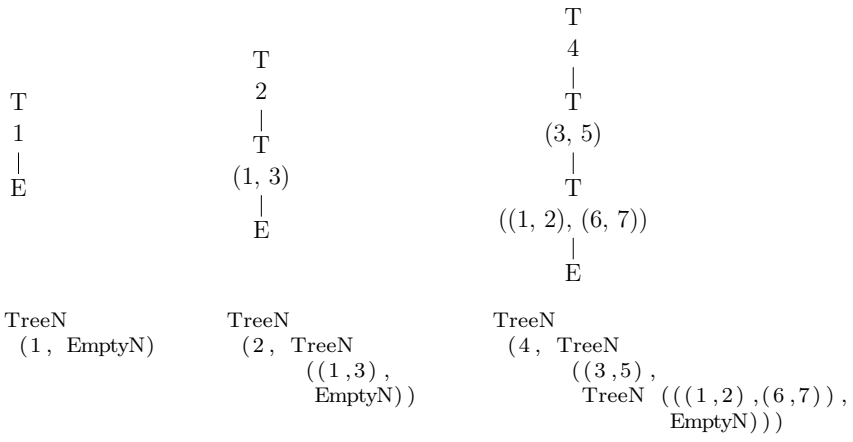


Figure 8.4: Perfectly-balanced trees

the elements are distributed evenly to the left and to the right. As a result, the only trees that can be built are perfectly balanced trees whose elements number one less than a power of two (Figure 8.4).

The definition of ntree is *non-regular* because the type constructor it defines, ntree, is not uniformly applied to its type parameters in the definition: instead, it is instantiated with 'a * 'a in the argument of TreeN. We call such non-regular types *nested*. Allowing the *return types* of constructors to vary in a similar way gives us a variety of non-regular types known as *generalized algebraic data types* (GADTs).

Our first example of a GADT definition involves a couple of auxiliary types for natural numbers:

```
type z = Z : z
type 'n s = S : 'n → 'n s
```

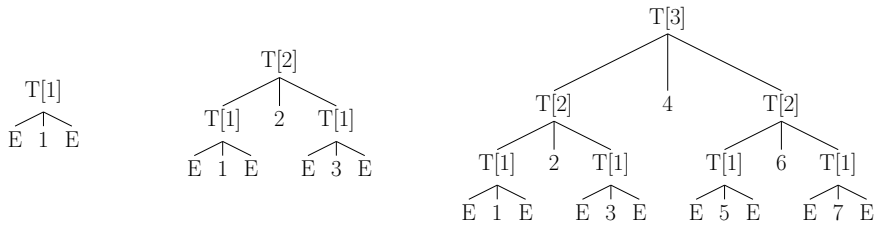
For each natural number n, the types z and s allow us to construct a type whose single inhabitant represents n. For example, the number three is represented by applying S three times to Z:

```
# S (S (S Z));;
- : z s s s = S (S (S Z))
```

Initially we will be mostly interested in the types built from z and s rather than the values which inhabit those types.

The types z and s are not themselves GADTs, but we can use them to build a GADT, gtree, that represents perfect trees:

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




```
TreeG
  (EmptyG,
   1,
   EmptyG)
```

```
TreeG
  (TreeG (EmptyG,
          1,
          EmptyG),
   2,
   TreeG (EmptyG,
          3,
          EmptyG))
```

```
TreeG
  (TreeG
    (TreeG (EmptyG, 1, EmptyG),
     2,
     TreeG (EmptyG, 3, EmptyG)),
   4,
   TreeG
    (TreeG (EmptyG, 5, EmptyG),
     6,
     TreeG (EmptyG, 7, EmptyG))
```

Figure 8.5: Perfect trees using GADTs

(The bracketed numbers indicate the depth of the subtree.)

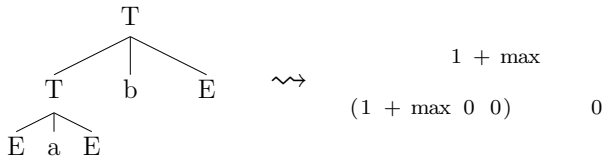
The definition of gtree corresponds to the definition of tree, but with an additional parameter for representing the depth of the tree. For the empty tree EmptyG the parameter is instantiated to z, reflecting the fact that empty trees have depth zero. For branching trees built with the TreeG constructor the depth parameter is instantiated to 'n s, where 'n is the depth of each of the two subtrees. There are two constraints introduced by this second instantiation: first, the subtrees are constrained to have the same depth 'n; second, the depth of the tree built with TreeG is one greater than the depth of its subtrees.

It is the different instantiations of the second type parameter in the return types of EmptyG and TreeG which make gtree a GADT. We call parameters which vary in constructor return types *indexes*.

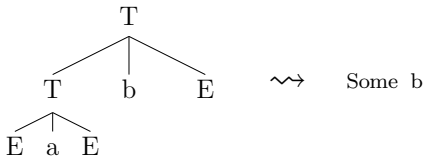
As with ntree, the constructors of gtree constrain the trees that we can build. Since both subtrees of each branch are constrained by the type of TreeG to have the same depth, the only values of type gtree are perfectly balanced trees. Figure 8.5 gives some examples.

8.1.1 Functions over GADTs

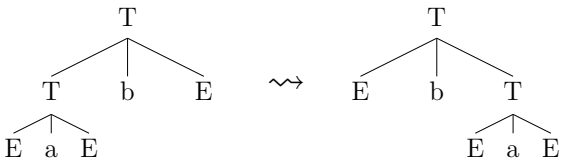
The gtree type illustrates how relaxing the regularity conditions on the types of data type constructors makes it possible to impose interesting constraints on the shape of data. However, *constructing* data is somewhat less than half the story: the most interesting aspects of GADT behaviour are associated with




```
let rec depth : 'a.'a tree → int = function
    Empty → 0
  | Tree (l,_,r) → 1 + max (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)
```

Figure 8.6: Functions over unconstrained binary trees

analysing values passed as function arguments. We will now consider a number of functions that analyse trees passed as arguments to illustrate how GADTs have certain clear advantages over regular and nested data types.

Functions over tree Figure 8.6 shows the implementations of three functions over the regular tree type tree. The first, depth, computes the depth of a tree, defined as zero if the tree is empty and the successor of the maximum of the depths of the left and right subtrees otherwise. The second, top, retrieves the element nearest the root of the argument tree, and returns an option value in order to account for the case where argument is empty. The third, swivel, rotates the tree around its central axis.

The types of depth, top and swivel are straightforward, but fairly uninformative. The type of depth tells us that the function accepts a tree and returns an int, but there is nothing in the type that indicates how the two are related.

It is possible to write functions of the same type that compute the number of elements in the tree rather than the depth, or that compute the number of unbalanced branches, or that simply return a constant integer. The type of swivel is slightly more informative, since we can apply parametricity-style reasoning to conclude that every element in the output tree must occur in the input tree, but we cannot say much more than this. It is possible to write functions with the same type as swivel that return an empty tree, ignoring the argument, or that duplicate or exchange nodes, or that simply return the argument unaltered.

Functions over ntree Figure 8.7 shows the implementation of functions corresponding to depth, top and swivel for the ntree type.

The implementations of depthN and topN correspond quite directly to their counterparts for the tree type. Since all values of type ntree are perfectly balanced, it is sufficient for depthN to measure the spine rather than computing the maximum depth of subtrees. One additional point is worth noting: since a non-empty value of type 'a ntree has a subtree of type ('a * 'a) ntree, depthN is an example of polymorphic recursion (Section 3.4.2).

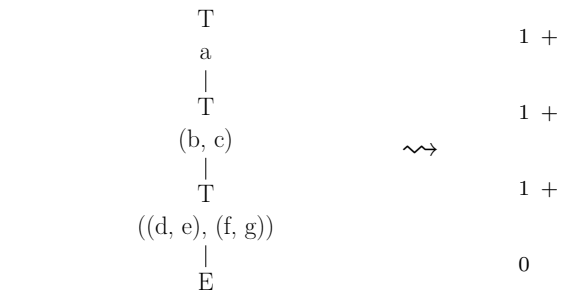
The implementation of swivelN is less straightforward, since it deals recursively with elements, and the element type changes as the depth increases. The auxiliary function swiv accepts a function f which can be used to swivel elements at a particular depth. At the point of descent to the next level, f is used to construct a function **fun** (x,y) → (f y,f x) that can be used to swivel elements one level deeper.

Functions over gtree Figure 8.8 shows the implementation of functions corresponding to depth, top and swivel for gtree.

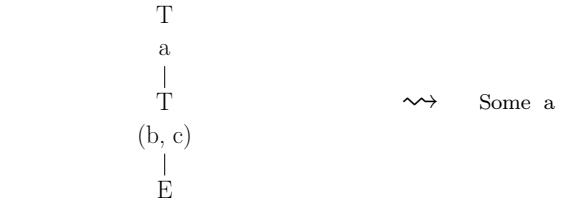
Locally abstract types The first thing to observe is the new syntax in the type signatures: the prefix **type** a n introduces two type names a and n that are in scope both in the signature and in the accompanying definition. These names denote so-called *locally abstract types*, and together with GADTs they support type-checking behaviour known as *type refinement*.

Type refinement Under standard pattern matching behaviour, matching a value against a series of patterns reveals facts about the structure of the value. For example, in the depth function of Figure 8.6, matching the argument determines whether it was constructed using Empty or Tree. It is only possible to extract the element from the tree in the Tree branch, since the variable v which binds the element is not in scope in the Empty branch.

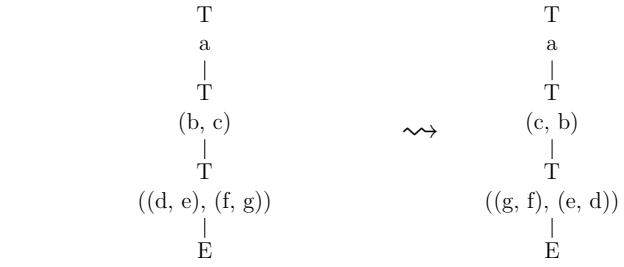
Type refinement extends standard pattern matching so that matching reveals facts both about the structure of the value and about its type. Since the constructors of a GADT value have different return types, determining which constructor was used to build a value reveals facts about the type of the value (and sometimes about other values, as we shall see later). Let's consider the implementation of depthG to see how refinement works.




```
let rec depthN : 'a.'a ntree → int =
  function
     EmptyN → 0
   | TreeN (_,t) → 1 + depthN t
```



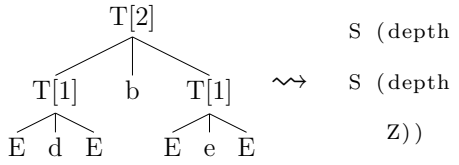

```
let rec topN : 'a.'a ntree → 'a option = function
    EmptyN → None
 | TreeN (v,_) → Some v
```



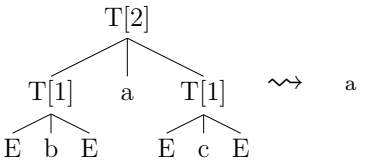

```
let rec swiv : 'a.('a→'a) → 'a ntree → 'a ntree =
 fun f t → match t with
    EmptyN → EmptyN
 | TreeN (v,t) →
    TreeN (f v, swiv (fun (x,y) → (f y, f x)) t)

let swivelN p = swiv id p
```

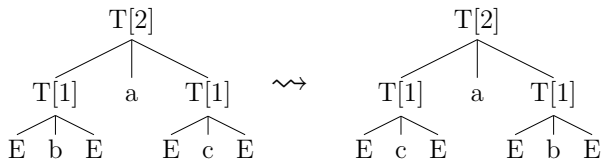
Figure 8.7: Functions over perfect trees




```
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)
```

Figure 8.8: Functions over perfect trees using GADTs

Here's the signature of depthG:

```
type a n.(a, n) gtree → n
```

Knowing the interpretation of the second type parameter, we might read this as follows: depthG takes a gtree with element type a and depth n and returns the depth n. Thus for an empty tree we expect depthG to return Z, and for a tree of depth three we expect it to return S (S (S Z)). However, these have different (and incompatible) types, and the normal type checking rules require that every branch of a **match** have the same type. Type refinement addresses exactly this difficulty. The first branch is executed if the value was built with EmptyG:

```
EmptyG → Z
```

Looking back to the definition of EmptyG (page 97) we find the depth parameter instantiated with z:

```
EmptyG : ('a,z) gtree
```

It is therefore reasonable to draw the following conclusion: if the first branch is executed, then the type equality n ≡ z must hold, and we can freely exchange n and z in the types of any expression within this branch. In particular, the expression Z, which has type z, can also be given the type n, which is exactly what is needed to satisfy the signature.

Similarly, the second branch is executed if the value was built with TreeG:

```
| TreeG (l,_,_) → S (depthG l)
```

In the definition of TreeG the depth parameter is instantiated with 'n s:

```
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree → ('a,'n s) gtree
```

If this branch is executed then we can draw the following series of conclusions:

1. The type equality n ≡ 'n s must hold for some unknown type variable 'n and we can freely exchange n and 'n s in the types of any expression within this branch
2. According to the type of the TreeG constructor the first constructor argument l has type (a, 'n) gtree.
3. The recursive call depthG l therefore has type 'n.
4. The application of the S constructorS (depthG l) therefore has type 'n s.
5. Since n ≡ 'n s, the expression S (depthG l) can be given the type n, which is exactly what is needed to satisfy the signature.

There's quite a lot going on to type check such a simple definition! It is only because we have specified the expected type of the function that type checking succeeds; there is no hope of inferring a principal type. There are at least three reasons why the type inference algorithm of Chapter 3 cannot be expected to determine a type for depthG:

1. The definition is *polymorphic-recursive*, since the argument passed to depthG has a different type to the parameter in the definition. We saw in Section 3.4.2 that polymorphic recursion is incompatible with type inference.

2. The type of the variable l is *existential*, which is a more formal way of saying the same thing as "for some unknown type variable 'n"' above. We saw in Chapter 6 that type inference for general existential types is undecidable.

3. *Type refinement* is not generally compatible with inference, since the type checker needs to know in advance what is being refined. We will cover this point in more detail in Section 8.3.

The second function over gtree values, topG, illustrates an additional benefit of type refinement. Although the gtree type has two constructors, the definition of topG matches only TreeG. A pattern match that matches only a subset of constructors for the matched type is usually a programming mistake which leads to a warning from the compiler, as we see if we try to give a similar one-branch definition for tree:

```
# let top : 'a.'a tree → 'a option =
  function Tree (_,v,_) → Some v;;
  Characters 38-69:
    function Tree (_,v,_) → Some v;;
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Empty
```

However, the OCaml compiler accepts topG without warning. An analysis of the type refinement that takes place in the definition shows why. Here is the type of topG:

```
type a n.(a,n s) gtree → a
```

As before, matching the TreeG branch refines the depth index type to 'n s for some unknown type variable 'n. Combining this with the depth index n s in the signature gives the type equality 'n s ≡ n s, (which is equivalent to the simpler equation 'n ≡ n, since the type constructor s is injective). Similarly, if we had a case for EmptyG we would again see the index type refined to z to give the equation z ≡ n s. However, this last equation clearly has no solutions: there is no value of n which can make the two sides the same! Since it is therefore impossible to pass the value EmptyG to topG, there is no need to include a case for EmptyG in the match.

In fact, the OCaml compiler goes a little further than simply accepting the incomplete match without a warning. Since the EmptyG case is clearly impossible, OCaml treats its inclusion as an error:

```
# let topG : type a n.(a,n s) gtree → a = function
    TreeG (_,v,_) → v
  | EmptyG → assert false;;
Characters 75-81:
```

```
  | EmptyG → assert false;;
Error: This pattern matches values of type (a, z) gtree
       but a pattern was expected which matches values
       of type (a, n s) gtree
       Type z is not compatible with type n s
```

The final function in Figure 8.8, swivelG, illustrates building GADT values in a context in which type equalities are known to hold. As with depthG, the compiler deduces from the types of the constructors in the pattern match that the equalities n ≡ z and n ≡ 'n s (for some 'n) hold in the EmptyG and TreeG branches respectively. The following type assignments are therefore justified:

- The expression EmptyG can be given the type (a, n) gtree in the EmptyG branch (since the EmptyG constructor has the type ('a, z) gtree for any 'a, and we know that n ≡ z).
- The bound variables l and r are each given the types (a, 'n) gtree, since the whole TreeG pattern has the type (a, 'n s) gtree.
- These types for l and r, together with the type of swivelG, lead to the type (a,'n) gtree for the recursive calls swivelG r and swivelG l.
- The expression TreeG (swivelG r, v, swivelG l) can be given the type (a, n) gtree using the types of the arguments, the type of the TreeG constructor and the type equality n ≡ 'n s.

The types for depthG, topG and swivelG are a little more complex than the types for the corresponding functions over unconstrained trees (tree) and nested trees (ntree). It is worth considering what we can learn about the functions from their types alone.

While the types of depth and depthN told us relatively little about the behaviour of those functions, the type of depthG tells us precisely what the function returns:

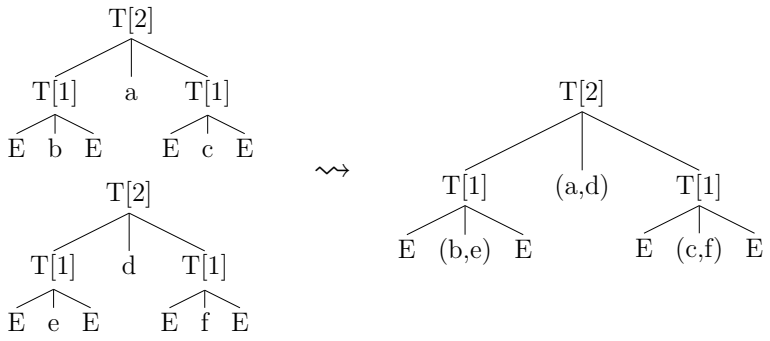
```
val depthG: ('a, 'n) gtree → 'n
```

Since the index 'n describes the depth of the tree, and the function returns a value of type 'n, we can be sure that the value returned by depthG represents the depth. For each value of type ('a, 'n) gtree there is exactly one corresponding value of type 'n.

The type of topG also tells us more than the types of its counterparts top and topN:

```
val topG : ('a,'n s) gtree → 'a
```

The instantiation of the depth index to 'n s tells us that only non-empty trees can be passed to topG. The return type, 'a, tells us that topG always returns an element of the tree, in contrast to top and topN, which might return None. Unlike the type of depthG, however, the type of topG does not completely specify the function. For example, a function that returned the leftmost or rightmost element of a gtree would have the same type.




```
let rec zipTree :
  type a b n.(a,n) gtree → (b,n) gtree → (a * b,n) gtree =
  fun x y → match x, y with
    EmptyG, EmptyG → EmptyG
  | TreeG (l,v,r), TreeG (m,w,s) →
    TreeG (zipTree l m, (v,w), zipTree r s)
```

Figure 8.9: Zipping perfect trees using GADTs

Finally, the type of swivelG once again tells us more than the types of swivel and swivelN:

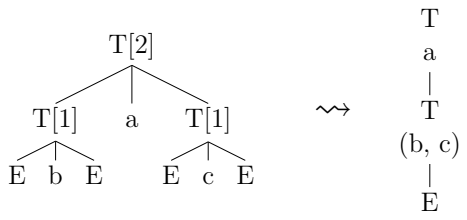
```
val swivelG : ('a,'n) gtree → ('a,'n) gtree
```

Since the depth index is the same in the parameter and the result we can conclude that swivelG preserves the depth of the tree passed as argument. Since trees of type gtree are always balanced, we can also conclude that the tree returned by swivelG always has the same number of elements as the input tree. As with topG, however, the type is not sufficiently precise to completely specify the behaviour. For one thing, the type of swivelG tells us nothing about swivelling: the identity function can be given the same type!

Conversions between the two representations of perfect trees We have shown two ways of representing perfect trees, using nested types and using GADTs. We can demonstrate the interchangeability of the two representations by defining an isomorphism between them. We will only give one half of the isomorphism here; the other half is left as an exercise for the reader (Question 4, page 120).

Figure 8.9 shows the implementation of a function zipTree, which turns a pair of gtree values into a gtree of pairs. There are two cases: first, if both input trees are empty, the result is an empty tree; second, if both trees are branches, the result is built by pairing their elements and zipping their left and right subtrees.

As with topG, type refinement relieves us of the need to specify the other cases, in which one tree is empty and the other non-empty. The type of zipTree




```
let rec nestify : type a n.(a,n) gtree → a ntree =
  function
    EmptyG → EmptyN
  | TreeG (l, v, r) →
    TreeN (v, nestify (zipTree l r))
```

Figure 8.10: Converting a gtree to an ntree

specifies that the two input trees have the same depth n. Consequently, if one tree matches EmptyG, we know that the depth of both trees must be z. Similar reasoning leads to the conclusion that if one tree is non-empty the other must also be non-empty.

Figure 8.10 shows how zipTree can be used to build a function nestify, which converts a gtree to an ntree. The depth information is discarded in the output, and so the types do not guarantee that the structures of the input and output trees correspond: we must examine the implementation of the function to convince ourselves that it is correct.

8.1.2 Depth-indexing imperfect trees

The trees we have seen so far fall into two categories: trees without balancing constraints implemented using standard variants, and perfect trees implemented either with nested types or with GADTs. At this point the reader may be wondering whether GADTs are only useful for representing data with improbable constraints that are unlikely to be useful in real programs. In this section we provide evidence to the contrary in the form of a second implementation of unbalanced trees, this time with an additional type parameter to track the depth. As we shall see, this implementation combines benefits of both the unconstrained and the depth-indexed trees from earlier in the chapter: the depth-indexing provides extra information about the types to improve correctness and performance, but we will be able to represent arbitrary binary branching structure.

The depth of an unbalanced tree is determined by the maximum depth of its branches. In order to implement a depth-indexed unbalanced tree we will need some way of constructing a type denoting this maximum.

Figure 8.11 defines a type max that represents the maximum of two numbers. Following the propositions-as-types correspondence described in Chapter 4 we

```
type (_,_,_) max =
    MaxEq : 'a → ('a, 'a, 'a) max
  | MaxFlip : ('a, 'b, 'c) max → ('b, 'a, 'c) max
  | MaxSuc : ('a, 'b, 'a) max → ('a s, 'b, 'a s) max

let rec max : type a b c. (a, b, c) max → c = function
    MaxEq x → x
  | MaxSuc m → S (max m)
  | MaxFlip m → max m
```

Figure 8.11: A max function for type-level natural numbers

will interpret the type constructor max as a three-place predicate which we will write $\text{MAX}(-,-) = -$, the type (a, b, c) max as the proposition $\text{MAX}(a,b) = c$, and a value of the type as a proof of the proposition. Viewed this way, the types of the three constructors for max become inference rules for constructing proofs. There are three rules, each of which is consistent with the notion that MAX defines a notion of maximum.

The first rule, max-eq, says that the maximum of a value a and the same value a is a.

$$\frac{a}{\text{MAX}(a,a) = a}\ \text{max-eq}$$

The premise a in the inference rule, corresponding to the argument of the constructor MaxEq, stipulates that the max-eq rule only applies if we have evidence for a; this will make it easier to write the max function described below, which produces the maximum value c from a proof that $\text{MAX}(a,b) = c$.

The second rule, max-flip, says that max is commutative.

$$\frac{\text{MAX}(a,b) = c}{\text{MAX}(b,c) = a}\ \text{max-flip}$$

The third rule, max-suc, says that the maximum of two values remains the maximum if we increment it.

$$\frac{\text{MAX}(a,b) = a}{\text{MAX}(a+1,b) = a+1}\ \text{max-suc}$$

These rules represent just one of many possible ways of defining the maximum predicate. The definition given here is convenient for our purposes, and allows us to build proofs for the maximum of any two natural numbers. For example, we can construct a proof for the proposition $\text{MAX}(1,3) = 3$ as follows:

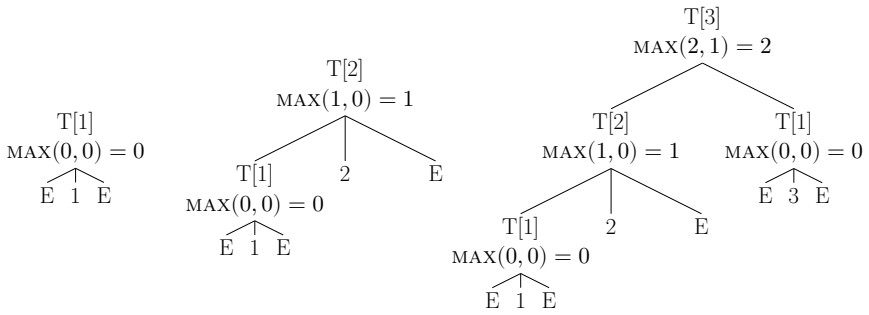


Figure 8.12: Depth-indexed unbalanced trees

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1}}{\textsc{max}(1,1) = 1}\ \text{max-eq}}{\textsc{max}(2,1) = 2}\ \text{max-suc}}{\textsc{max}(3,1) = 3}\ \text{max-suc}}{\textsc{max}(1,3) = 3}\ \text{max-flip}$$

Translating the proof back to OCaml, turning each rule application into a constructor application gives us a value of type `(z s, z s s s, z s s s) max`:

```
# MaxFlip (MaxSuc (MaxSuc (MaxEq (S Z))));;
- : (z s, z s s s, z s s s) max = ...
```

Figure 8.11 also defines the function `max`, which builds a value of type `c` from a value of type `(a,b,c) max`. For example, when given our proof of $\textsc{max}(1,3) = 3$ the `max` function will return `3`:

```
# max (MaxFlip (MaxSuc (MaxSuc (MaxEq (S Z)))));;
- : z s s s = S (S (S Z))
```

Now that we have defined the `max` type we can move on to the definition of the trees themselves. Starting from the `tree` type that represents unbalanced trees two changes are needed. First, we must add a type parameter representing the depth. Second, we must store in each non-empty tree a value which relates the depths of the subtrees to the depth of the tree. Here is the definition:

```
type ('a,_) dtree =
  EmptyD : ('a,z) dtree
| TreeD : ('a,'m) dtree * 'a * ('a,'n) dtree * ('m,'n,'o) max
      → ('a,'o s) dtree
```

The `EmptyD` constructor is straightforward: an empty tree has depth zero. In the definition of `TreeD` the depth indexes of the subtrees (`'m` and `'n`) and in the return type (`'o`) are all different, but the relation between them is represented by an additional value of type `('m,'n,'o) max`. As we have seen, this value represents the fact that `'o` is the maximum of `'m` and `'n`; further, since the depth index

in the return type of TreeD is 'o s we have captured the desired property that the depth of a non-empty tree is one greater than the maximum depth of its subtrees (Figure 8.12).

Figure 8.13 shows the implementation of functions corresponding to depth, top and swivel for dtree.

The depthD function computes the depth of a depth-indexed unbalanced tree. The max value stored in each non-empty node supports a relatively efficient implementation, since it allows us to retrieve the depth of the deeper subtree without inspecting the subtrees themselves.

The topD function retrieves the topmost element of a non-empty tree. As with topG, the type refinement that takes place when the function matches the argument determines that only the TreeD constructor can occur.

The swivelD function rotates a tree around its central axis. Exchanging the left and right subtrees requires updating the max value which records which subtree is deeper: we must replace a proof $\textsc{max}(l, r) = t$ with a proof $\textsc{max}(r, l) = t$.

8.1.3 GADTs and efficiency

As we saw when considering topG (page 104), the extra type information introduced by GADT indexes enable the compiler to detect **match** cases that can never be executed. In addition to allowing the programmer to omit unreachable branches, this analysis also makes it possible for the compiler to generate more efficient code. If type checking a **match** reveals that only one of the constructors of the scrutinee value type can ever occur then the generated code need not examine the value at all, leading to simpler generated code and faster execution.

Here is the definition of top from Figure 8.6:

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

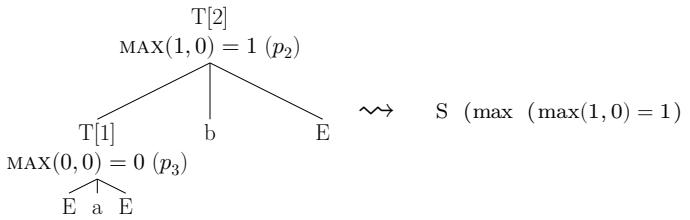
Passing the -dlambda option to the OCaml compiler causes it to print out an intermediate representation of the code[2]:

```
(function p
  (if p
     (makeblock 0 (field 1 p))
     0a))
```

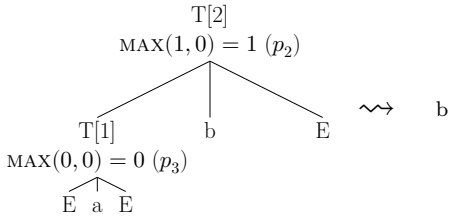
This intermediate representation corresponds quite closely to the source. The value of top is a function with a parameter p. There is a branch if p to determine whether the constructor is Tree or Empty; if it is Tree then makeblock is called to allocate a Some value, passing the first field (called v in the definition of top) as an argument. If p is determined to be Empty then the function returns 0a, which is the intermediate representation for None.

Here is the definition of topG from Figure 8.8:

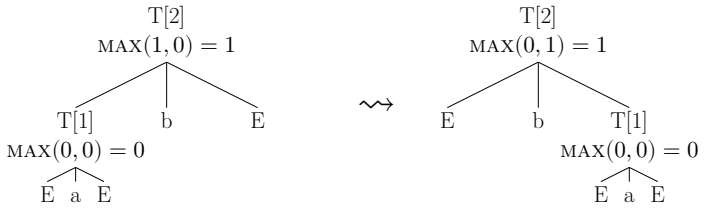
[2]Some names in the code below have been changed to improve legibility.




```
let rec depthD : type a n.(a,n) dtree → n = function
    EmptyD → Z
  | TreeD (l,_,r,mx) → S (max mx)
```




```
let topD : type a n.(a,n s) dtree → a =
  function TreeD (_,v,_,_) → v
```




```
let rec swivelD : type a n.(a,n) dtree → (a,n) dtree = function
    EmptyD → EmptyD
  | TreeD (l,v,r,m) → TreeD (swivelD r, v, swivelD l, MaxFlip m)
```

Figure 8.13: Functions over depth-indexed unbalanced trees

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

This time the code printed out when we pass -dlambda is much simpler:

```
(function p
  (field 1 p))
```

The call to makeblock has disappeared, since the more precise typing allows topG to return an a rather than an a option. Additionally, the compiler has determined that the Empty constructor can never be passed as an argument to topG, and so no branch has been generated. The source language types have made it possible for the compiler to emit significantly simpler and faster code.

Let's consider one more example. The zipTree function of Figure 8.10 traverses two gtree values in parallel. The lambda code for zipTree is a little more complicated than the code for topG, mostly as a result of the recursion in the input definition:

```
(letrec
  (zipTree
    (function x y
      (if x
        (makeblock 0
          (apply zipTree (field 0 x) (field 0 y))
          (makeblock 0 (field 1 x) (field 1 y))
          (apply zipTree (field 2 x) (field 2 y)))
        0a)))
  (apply (field 1 (global Toploop!)) "zipTree" zipTree))
```

Of particular interest is the generated branch if x. A match for two values, each with two possible constructors, will typically generate code with three branches to determine which of the four possible pairs of constructors has been passed as input. However, the type for zipTree constrains the types of the arguments so that both must be EmptyG or both TreeG. As there are only two cases to distinguish the compiler can generate a single branch if x to determine the constructors for both arguments.

8.2 GADTs and type equality

We have seen the central role that type equalities play in programs involving GADTs. The type refinement that is associated with pattern matching on GADT values introduces type equalities that follow the types of the constructors in the match. We now consider the definition of a GADT eql which captures the idea of a type equality, and outline how every other GADT value can be straightforwardly encoded using non-GADT variant types together with eql.

Figure 8.14 gives a definition of the eql type, which has two type parameters and a single constructor with no arguments. The parameters of eql are instantiated to the same type 'a, enforcing the constraint that Refl can only be used with

```
type (_, _) eql = Refl : ('a, 'a) eql
```

Figure 8.14: The equality GADT

```
let symm : type a b.(a, b) eql → (b, a) eql =
  fun Refl → Refl


let trans : type a b c.(a, b) eql → (b, c) eql → (a, c) eql =
  fun Refl Refl → Refl


module Lift (T : sig type _ t end) :
sig
  val lift : ('a,'b) eql → ('a T.t,'b T.t) eql
end =
struct
  let lift : type a b.(a, b) eql → (a T.t, b T.t) eql =
    fun Refl → Refl
end


let cast : type a b.(a,b) eql → a → b =
  fun Refl x → x
```

Figure 8.15: Some properties of type equality

a type (a,b) eql when the types a and b are known to be equal. For example, we can use Refl at type (int,int) eql, since the types int and int are known to be equal:

```
# (Refl : (int, int) eql);;
- : (int, int) eql = Refl
```

Similarly, since a type alias declaration simply introduces a new name for an existing type, we can instantiate the parameters of Refl with a type and its alias:

```
# type t = int;;
type t = int
# (Refl : (t, int) eql);;
- : (t, int) eql = Refl
```

However, if we hide the representation of t behind the signature of a module M then the fact that t is equal to int is hidden and attempting to build a value of type (M.t, int) eql meets with failure:

```
# module M : sig type t end = struct type t = int end;;
module M : sig type t end
# (Refl : (M.t, int) eql);;
Characters 1-5:
  (Refl : (M.t, int) eql);;
   ^^^^

Error: This expression has type (M.t, M.t) eql
       but an expression was expected of type (M.t, int) eql
       Type M.t is not compatible with type int
```

This last example gives a clue to how we might use eql. Since modules and other abstraction mechanisms make different type equalities visible at different parts of a program it is useful to have a means of passing equalities around as values. From a Curry-Howard (Propositions as Types) perspective, we can view a value of type (a,b) eql as a *proof* of the proposition that a and b are equal. Looked at this way, eql and other GADTs are a convenient way of programming with proofs as first-class objects.

We first introduced a representation for type equalities in Chapter 2, where we used Leibniz's principle of substitution in a context to construct values representing equality of types. In addition to the equality type itself we introduced a number of functions representing various properties of type equality: symmetry, transitivity, and so on. Figure 8.15 gives implementations of a number of these properties for the eql GADT:

- symm encodes the symmetry property of $\equiv$: if a $\equiv$ b then b $\equiv$ a.
- trans encodes the transitivity property of $\equiv$: if a $\equiv$ b and b $\equiv$ c then a $\equiv$ c.
- Lift lifts equality through type contexts: if a $\equiv$ b then for any context - t we have a t $\equiv$ b t
- Finally, the type of cast tells us that if a $\equiv$ b then we can convert a value of type a to a value of type b.

As the figure shows, the implementations of these properties for the eql GADTs are significantly simpler than the corresponding implementations of equality in System Fω. In System Fω we had to find a suitable type context argument to pass to the encoding of equality; with GADTs we simply match on Refl and rely on type refinement to ensure that the types match up. It is worth examining the type checking of one of these equality property functions to see the type refinement in action. The signature for symm is as follows:

```
type a b.(a, b) eql → (b, a) eql
```

The signature dictates a simple implementation: we only have a single constructor Refl for eql, which we must use in as the pattern and the body of the function:

```
fun Refl → Refl
```

In the signature for symm we have two distinct locally abstract types a and b. Matching against Refl reveals that we must have a ≡ b, since the two type parameters in the definition of Refl are the same. The type equality a ≡ b justifies giving the Refl in the body the type b ≡ a, which is just what is needed to satisfy the signature.

8.2.1 Encoding other GADTs with eql

We can use eql together with a standard (non-GADT) OCaml data type to build a data type that behaves like gtree. Here is the definition:

```
type ('a,'n) etree =
  EmptyE : ('n,z) eql → ('a,'n) etree
| TreeE : ('n,'m s) eql *
    ('a,'m) etree * 'a * ('a,'m) etree → ('a,'n) etree
```

Each constructor of etree has an additional argument which represents an instantiation of the second type parameter. The EmptyE constructor has an argument of type ('n,z) eql, reflecting the instantiation of the depth parameter to z in the original definition of EmptyG. Similarly the TreeE constructor has an additional argument of type ('n, 'm s) eql, where the existential type variable 'm is the depth of the two subtrees, reflecting the instantiation of the depth parameter to 'n s in the definition of TreeG.

For each function involving gtree we can write a corresponding function for etree. Here is an implementation of the depth operation:

```
let rec depthE : type a n.(a, n) etree → n =
  function
    EmptyE Refl → Z
  | TreeE (Refl, l,_,_) → S (depthE l)
```

In contrast to depthG, no type refinement takes place when EmptyE and TreeE are matched. However, matching the GADT constructor Refl introduces the same type equalities as for depthG, namely n ≡ z in the first branch and n ≡ 'm s in the second.

Implementing equivalents of topG and swivelG is left as an exercise (Question 1, page 120).

8.3 GADTs and type inference

We have mentioned in passing that it is not possible in general to infer types for functions involving GADTs, as we shall now show with a simple example. The following function matches a value of type eql and returns an int:

```
let match_eql = function Refl → 3
```

We can ascribe a number of types to match_eql, including the following:

```
let match_eql1 : type a.(int,a) eql → a = function Refl → 3
let match_eql2 : type a b.(a,b) eql → int = function Refl → 3
```

However, neither of these is a substitution instance (Section 3.3) of the other, and there is no valid type for match_eql that generalises both. Without the principal types property we cannot infer types without sacrificing generality.

8.4 GADT programming patterns

Up to this point we have focused on what GADTs are, and on how to understand the behaviour of the OCaml compiler on programs involving GADTs. However, there is more to programming than simply understanding the mechanics of type checking: using GADTs effectively requires a rather different programming style than programming with simpler types. We will now look at a number of programming patterns that emerge when using GADTs in real programs.

8.4.1 Pattern: building GADT values

> It's not always possible to determine index types statically.
> For example, the depth of a tree might depend on user input.

In the first part of this chapter we considered various functions whose arguments are trees defined using GADTs. Each of these functions follows a similar approach: the input tree has a type whose depth index involves universally quantified type variables which may also occur in the function's return type. Pattern matching on the input tree reveals equalities between the type indexes which can be used when constructing the return value. In this way we can write a wide variety of functions whose types connect together the shapes of the input and result types in some way.

The combination of polymorphism in the depth index together with type refinement makes it straightforward to write functions which accept and scrutinise trees of different depths. However, writing functions which return trees of different depths introduces new difficulties. Here is a function which builds a tree without depth constraints from an option value:

```
let tree_of_option : 'a. 'a option → 'a tree =
  function
    None → Empty
  | Some v → Tree (Empty, v, Empty)
```

If we try to write a corresponding function for depth-indexed trees we soon run into difficulty. As a first attempt we might start to write

```
let dtree_of_option : type a n. a option → (a, n) dtree =
```

but we will soon discover that this is not what we want: the type says that the function returns a value of type (a,n) dtree for *all* n, but the specification of the function requires that it return a tree indexed by *some* particular depth — either (a,z) dtree or type (a,z s) dtree — not a value that is polymorphic in the depth.

To state the problem is, in this case, to give the solution. The return value has *some* depth, not *all* depths, so the appropriate quantifier is existential, not universal. We have seen how to define existential types in OCaml in Chapter 6: we must define a variant type whose definition involves more type variables than type parameters. It is the depth index that we want to conceal, so we define the following type, which has 'a but not 'n as a parameter:

```
type 'a edtree = E : ('a, 'n) dtree → 'a edtree
```

Hiding the depth of trees with edtree allows us to return trees of different depths in different branches, and so to write dtree_of_option:

```
let dtree_of_option : type a n. a option → a edtree =
 function
   None → E EmptyD
 | Some v → E (TreeD (EmptyD, v, EmptyD, MaxEq Z))
```

The depth information is hidden, but not gone forever. Other parts of the program can recover the depth by unpacking the existential and matching on the dtree constructors in the usual way.

Chapter 6 highlighted the duality of existential and universal quantification that allows us to move between abstraction and parametricity. It is therefore no surprise to discover that there is a second approach to building depth-indexed trees of unknown depth using universals rather than existentials. We can pass a value of polymorphic type as an argument using a record with a polymorphic field:

```
type ('a, 'k) adtree = { k: 'n. ('a, 'n) dtree → 'k }
```

The type of adtree might be read as follows: a value of type ('a,'k) adtree is a (record containing a) function which accepts a tree of any depth with element type 'a and returns a value of type 'k.

Equipped with adtree we can give an alternative implementation of dtree_of_option which accepts a function wrapped as an adtree to which it passes the constructed tree. The depth polymorphism in the definition of adtree ensures that it is able to accept any tree, regardless of depth.

```
let dtree_of_option_k : type a k. a option → (a, k) adtree → k =
 fun opt {k} → match opt with
   None → k EmptyD
 | Some v → k (TreeD (EmptyD, v, EmptyD, MaxEq Z))
```

8.4.2 Pattern: singleton types

> Without dependent types we can't write predicates involving data.
> Using one type per value allows us to simulate value indexing.

We saw in Chapter 4 that the types in non-dependently-typed languages such as System F correspond to propositions in a logic without quantification over objects. The System F type language has no constructs for referring to individual values, so there is a syntactic barrier to even forming types that correspond to propositions involving individuals.

However, we have seen in Section 8.1.2 that we *can* apparently form propositions involving individuals. For instance, we can use max to form types that correspond to predicates like $\text{MAX}(1,3) = 3$, which mention the individual numbers 1 and 3, not just sets like $\mathbb{N}$. This appears to conflict with our claims above, and might lead us to wonder whether the extra expressive power that comes with GADTs allows us to write dependently typed programs in OCaml.

In fact there is no conflict. GADTs do not allow us to write dependently-typed programs and types like (z, z s s, z s s) max correspond to propositions in a logic without quantification over individual objects. The key to understanding types like max is the observation that types such as z and z s s are so-called *singleton types* — i.e. they each have a single inhabitant. When there is only one value of each type the type can act as a proxy for the value in type expressions and we can simulate the quantification over individuals which the type language does not support directly.

Here is an additional example. We can represent equations of the form $a + b = c$ using the following GADT definition:

```
type (_,_,_) add =
    AddZ : 'n → (z, 'n, 'n) add
  | AddS : ('m, 'n, 'o) add → ('m s, 'n, 'o s) add
```

As we saw in Section 8.1.2 we can read the types of the constructors of add as inference rules for constructing proofs:

$$\frac{\text{n}}{0 + n = n}\ \text{add-z} \qquad\qquad \frac{\text{m+n=0}}{(1 + m) + n = 1 + o}\ \text{add-s}$$

Then each value of type add corresponds to a proof of some fact about addition. For example, we can build a value corresponding to a proof that $2 + 1 = 3$:

```
# AddS (AddS (AddZ (S Z)));;
- : (z s s, z s, z s s s) add = AddS (AddS (AddZ (S Z)))
```

The singleton pattern works well for simple data such as natural numbers, and can sometimes be extended to more complex data. The further reading section (page 122) lists a number of papers that explore how singletons support encoding dependently-typed programs in languages without dependent types.

8.4.3 Pattern: building evidence

> With type refinement we learn about types by inspecting values.
> Predicates should return useful *evidence* rather than **true** or **false**.

In a typical program many constraints on data are not captured in the types. The programmer might ensure through careful programming that a certain list is always kept in sorted order or that a file handle is not accessed after it is closed, but since the information is not made available to the type checker there is no way for the compiler either to ensure that the constraint is maintained or to make use of it to generate more efficient code.

For example, if we wish to ensure that our program never attempts to retrieve the top element of an empty tree we might write a predicate that test for emptiness

```
let is_empty : 'a .'a tree → bool =
  function
    Empty → true
  | Tree _ → false
```

and then use the predicate to test trees before passing them to top:

```
if not (is_empty t) then
  f (top t)
else
  None
```

There is potential both for error and for inefficiency here. Although the programmer knows that the bool returned by is_empty is intended to indicate whether the tree was determined to be empty, the type checker does not, and so would have no cause for complaint if we were to switch the two branches of the **if** expression or omit the call to the not function. Further, there is nothing in the types which allows the top function to skip the test for emptiness, so the generated code tests for emptiness twice, once in the condition of the **if** and once in top.

GADTs offer a solution to this unsatisfactory state of affairs. The problem lies in the type of the predicate function, which tells us nothing about the facts that the predicate was able to discover. If we arrange for our predicates to have return types more informative than bool then the facts which the predicates discover can flow through the types to the rest of the program.

In the example above is_empty checks whether a tree is empty or non-empty — that is, whether its depth is zero or non-zero. We can capture this property in a type that, like bool, has two nullary constructors like bool but, unlike bool, is indexed by what could be determined about the depth:

```
type _ is_zero =
  Is_zero : z is_zero
| Is_succ : _ s is_zero
```

We can use the is_zero type to write a predicate that builds *evidence* for the emptiness or non-emptiness of its argument:

```
let is_emptyD : type a n.(a,n) dtree → n is_zero =
  function
    EmptyD → Is_zero
  | TreeD _ → Is_succ
```

As with is_empty, we can branch on the result of is_emptyD to determine whether it is safe to call topD:

```
match is_emptyD t with
  Is_succ → f (topD t)
| Is_zero → None
```

Whereas calling is_empty communicated no information to the type checker about the depth of the tree, examining the result of is_emptyD reveals whether the depth is z or 'n s (for some type 'n). It is only in the second case that the type checker will allow the call to topD. Switching the two branches leads to a type error, since Is_zero reveals that the depth of the tree is z, and the type of topD demands a non-zero depth index. Further, as we have seen in Section 8.1.3, there is no need for topD to repeat the test for emptiness once we have captured the fact that the tree is non-empty in its type.

8.5 Exercises

1. [★] Implement the functions corresponding to top and swivel for etree (Section 8.2.1).

2. [★] It is sometimes convenient to work with a GADT for natural numbers, indexed by z and s:

   ```
   type _ nat =
       Z : z nat
     | S : 'n nat → 'n s nat
   ```

   Use nat to write a function of the following type:

   ```
   val unit_gtree_of_depth : 'n nat → (unit, 'n) gtree
   ```

3. [★★] Write a second function

   ```
   val int_gtree_of_depth : 'n nat → (int, 'n) gtree
   ```

   which, for a given natural number n, builds a tree populated with the numbers $0 \ldots 2^{n-1}$ in left-to-right order.

4. [★★] Write a inverse for zipTree:

   ```
   val unzipTree : ('a * 'b,n) gtree → ('a,'n) gtree → ('b,'n) gtree
   ```

   and use it to write an inverse for nestify that converts an ntree to a gtree, using an existential for the return type.

5. [★★] Define a type of length-indexed vectors using GADTS:

```
type ('a, _) vec = ...
```

A vector is either empty, in which case its length is zero, or consists of a cons cell with an element and a tail, in which case its length is one greater than the tail.

Write analogues of the list-processing functions head, tail, map and rev for your vec type.

6. [★★★★] Here is an alternative definition of the equality type of Figure 8.14:

```
type ('a, 'b) eql_iso = {
  a_of_b : 'b → 'a;
  b_of_a : 'a → 'b;
}
```

It is possible to define a number of the equality operations for eql_iso, including refl and symm:

```
let refl : 'a. ('a, 'a) eql_iso =
  { a_of_b = (fun x → x); b_of_a = (fun x → x) }

let symm : 'a 'b. ('a, 'b) eql_iso → ('b, 'a) eql_iso =
  fun { a_of_b; b_of_a } → { a_of_b = b_of_a; b_of_a = a_of_b }
```

Is it possible to define analogues of all the functions in Figure 8.15 for eql_iso? Is it possible to encode arbitrary GADT types in the style of Section 8.2.1 using eql_iso instead of eql?

7. [★★★★] Here is a function which turns a proof of equality for list types into a proof of equality for their element types:

```
let eql_of_list_eql : type a b.(a list,b list) eql → (a,b) eql =
  fun Refl → Refl
```

Here is a similar function for option:

```
let eql_of_option_eql : type a b.(a option,b option) eql → (a,b)
    eql =
    fun Refl → Refl
```

Rather than writing such a function for every type constructor we would like to give a single definition which could be reused. However, the following attempt is rejected by OCaml. Can you explain why?

```
module Eql_of_t_eql(T: sig type 'a t end) =
struct
   let eql_of_t_eql : type a b.(a T.t,b T.t) eql → (a,b) eql =
      fun Refl → Refl
end
```

Further reading

- The following paper describes a number of GADT programming patterns realised in the language Ωmega. Features similar to those used in the paper, namely GADTs and extensible kinds, have found their way into recent versions of the Glasgow Haskell Compiler:

  Putting Curry-Howard to work
  Tim Sheard
  Haskell Workshop (2005)

- A number of papers investigate how to simulate dependently-typed programming using GADTs and other features of functional programming languages (typically Haskell). Here are a few examples:

  Faking It (Simulating Dependent Types in Haskell)
  Conor McBride
  Journal of Functional Programming (2003)

  Dependently typed programming with Singletons
  Richard A. Eisenberg and Stephanie Weirich
  Haskell Symposium (2012)

  Hasochism: the pleasure and pain of dependently typed Haskell programming
  Sam Lindley and Conor McBride
  Haskell Symposium (2013)

- The following paper shows how to encode dynamic types in a statically-typed functional language using many of the techniques described in this chapter and elsewhere in the notes. Since the paper predates the introduction of GADTs into functional languages it uses an encoding of Leibniz equality to perform a similar function.

  Typing dynamic typing
  Arthur I. Baars and S. Doaitse Swierstra
  International Conference on Functional Programming (2002)

- There is an interesting correspondence between various number systems and tree types, which can be realised using nested types, as the following paper shows:

  Numerical Representations as Higher-Order Nested Datatypes
  Ralf Hinze
  Technical Report (1998)

- There have been a number of papers over the last decade or so proposing algorithms for type-checking GADTs. One of the more straightforward, which describes the approach taken in OCaml, is described in the following paper:

Ambivalent types for principal type inference with GADTs
Jacques Garrigue and Didier Rémy
Asian Symposium on Programming Languages and Systems (2013)

- As types become richer, the inhabitants (i.e. the terms having those types) becomes fewer; and in many cases (such as the polymorphic compose function (Chapter 2) and the trans function (page 113) the type is sufficiently descriptive that there is only a single inhabitant. The following paper investigates the question of when a type has a unique inhabitant:

  Which simple types have a unique inhabitant?
  Gabriel Scherer and Didier Rémy
  International Conference on Functional Programming (2015)