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 2.4.1: 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:

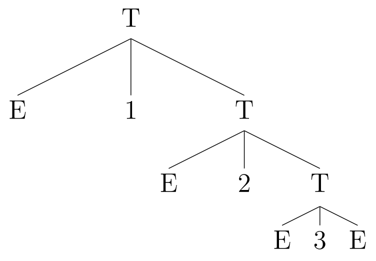
```
# List.hd [];;
Exception: Failure "hd".
```

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

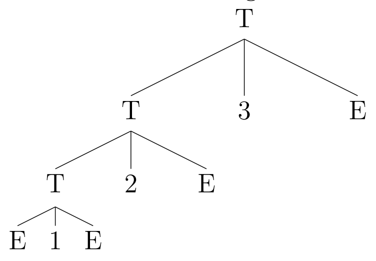
Towards the end of Chapter 2 we considered some different approaches to defining binary branching tree types. Under the following definition a tree is either empty, or consists of an element of type `'a` and a pair of trees:

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



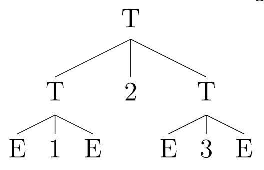
```
Tree
  (Empty,
   1,
   Tree
    (Empty,
     2,
     Tree (Empty, 3, Empty)))
```

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

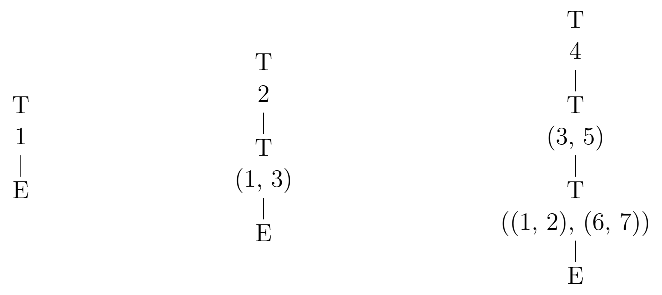
Using the constructors of `tree` we can build a variety of tree values. For example, we can build trees that are skewed to the right (Figure 8.1) or to the left (Figure 8.2), or whose elements are distributed evenly between left and right (Figure 8.3).

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.



```
TreeN             TreeN               TreeN
 (1, EmptyN)       (2, TreeN           (4, TreeN
                        ((1,3),             ((3,5),
                         EmptyN))            TreeN (((1,2),(6,7)),
                                                    EmptyN)))
```

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
```

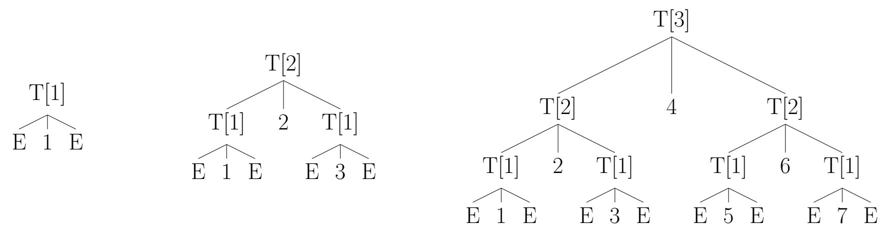
For each natural number `n`, the types `z` and `s` allow us to construct a type whose single inhabitant represents `n`. For example, the number three is represented by applying `S` three times to `Z`:

```
# 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.)

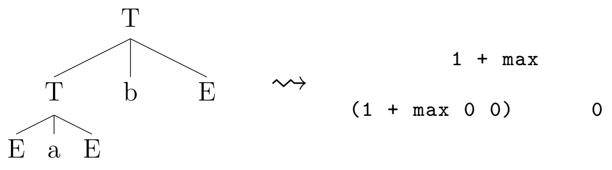
The definition of `gtree` corresponds to the definition of `tree`, but with an additional parameter for representing the depth of the tree. For the empty tree `EmptyG` the parameter is instantiated to `z`, reflecting the fact that empty trees have depth zero. For branching trees built with the `TreeG` 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*.

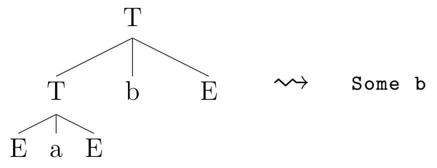
As with `ntree`, the constructors of `gtree` constrain the trees that we can build. Since both subtrees of each branch are constrained by the type of `TreeG` to have the same depth, the only values of type `gtree` are perfectly balanced trees. Figure 8.5 gives some examples.

8.1.1 Functions over GADTs

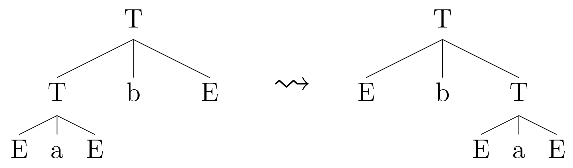
The `gtree` type illustrates how relaxing the regularity conditions on the types of data type constructors make it possible to impose interesting constraints on the shape of data. However, constructing data is somewhat less than half the story: the most interesting aspects of GADT behaviour are associated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

destructing values passed as function arguments. We will now consider a number of functions which destruct 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).

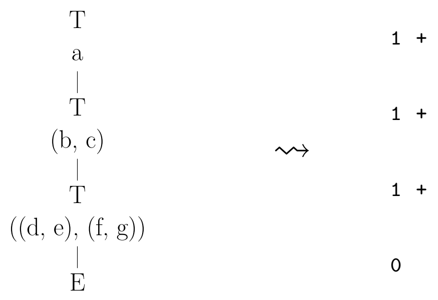
The implementation of `swivelN` is less straightforward, since it deals recursively with elements, and the element type changes as the depth increases. The auxiliary function `swiv` accepts a function `f` which can be used to swivel elements at a particular depth. At the point of descent to the next 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`.

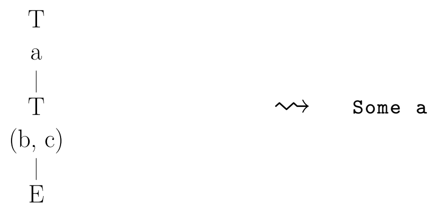
Locally abstract types The first thing to observe is the new syntax in the type signatures: the prefix `type a n` introduces two type names `a` and `n` that are in scope both in the signature and in the accompanying definition. These names denote so-called *locally abstract types*, and together with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

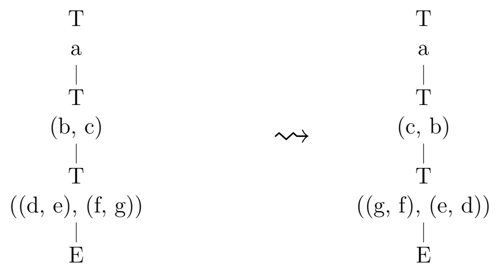
Type refinement extends standard pattern matching so that matching reveals facts both about the structure of the value and about its type. Since the constructors of a GADT value have different return types, determining which constructor was used to build a value reveals facts about the type of the value (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
```



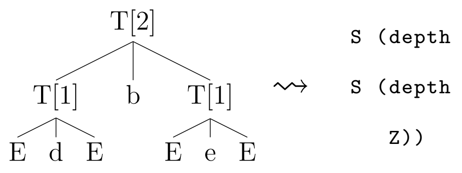
```
let rec topN : 'a.'a ntree → 'a option = function
   EmptyN → None
 | TreeN (v,_) → Some v
```



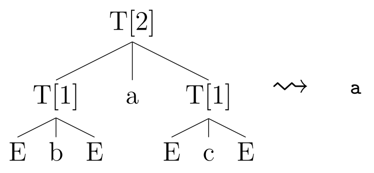
```
let rec swiv : 'a.('a→'a) → 'a ntree → 'a ntree =
 fun f t → match t with
   EmptyN → EmptyN
 | TreeN (v,t) →
   TreeN (f v, swiv (fun (x,y) → (f y, f x)) t)

let swivelN p = swiv id p
```

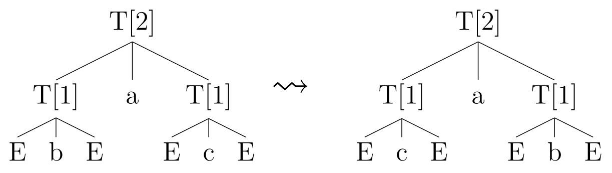
Figure 8.7: Functions over perfect trees



```
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 `z` 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:

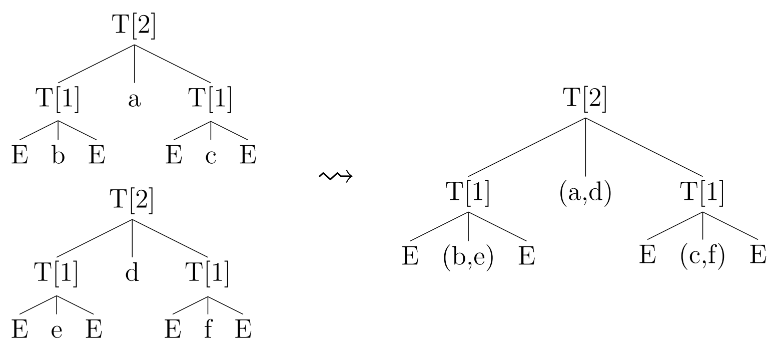
```
val depthG: ('a, 'n) gtree → 'n
```

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 which 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

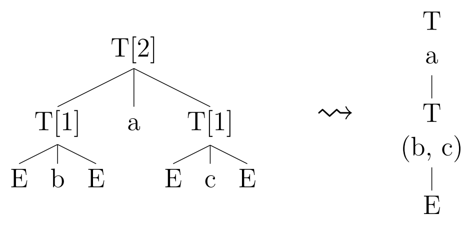
Finally, the type of `swivelG` once again tells us more than the types of `swivel` and `swivelN`:

```
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 swiveling: 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 130).

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.



```
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`

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` 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 which 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

```
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

need some way of constructing a type denoting this maximum.

Figure 8.11 defines a type `max`[2] that represents the maximum of two numbers. Following the propositions-as-types correspondence described in Chapter 4 we will interpret the type constructor `max` as a three-place predicate which we will write $\textsc{max}(-,-) = -$, the type `(a, b, c) max` as the proposition $\textsc{max}(a,b) = c$, 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.

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

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 $\textsc{max}(a,b) = c$.

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

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

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

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

[2]The definition of `max` used here is simpler and supports more efficient function definitions than the definition presented in lectures.

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 $\textsc{max}(1,3) = 3$ as follows:

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

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.

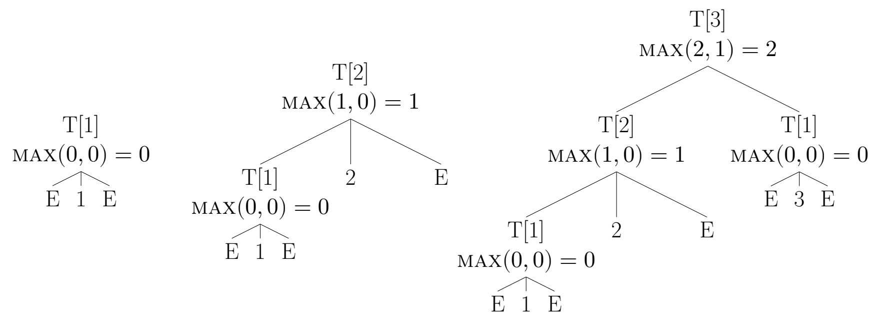


Figure 8.12: Depth-indexed unbalanced trees

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.

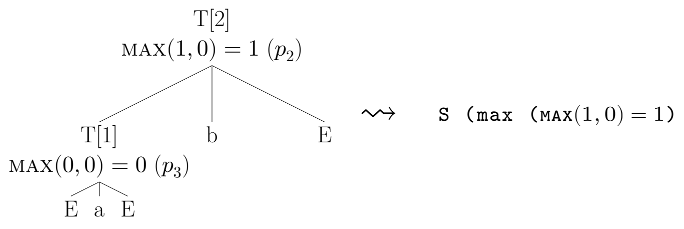
Here is the definition of `top` from Figure 8.6:

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

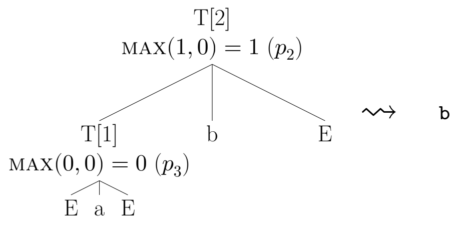
Passing the `-dlambda` option to the OCaml compiler causes it to print out an intermediate representation of the code[3]:

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

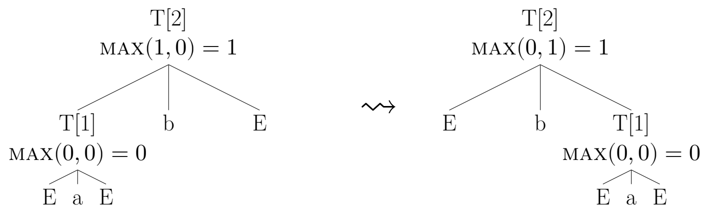
[3]Some names in the code below have been changed to improve legibility.



```
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

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:

```
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

```
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

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 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 ≡: if `a ≡ b` then `b ≡ a`.
- `trans` encodes the transitivity property of ≡: if `a ≡ b` and `b ≡ c` then `a ≡ c`.
- `Lift` lifts equality through type contexts: if `a ≡ b` then for any context `- t` we have `a t ≡ b t`
- Finally, the type of `cast` tells us that if `a ≡ 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 128).

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
```

```
json ::= string | number | true | false | null
         [] | [ json-seq ]
json-seq ::= json
            json , json-seq
```

Figure 8.16: A grammar for JSON, without objects

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.

In order to illustrate the various programming patterns we'll use the running example of representing JSON-like data[4]. Figure 8.16 gives a grammar for a subset of JSON expressions. For the sake of simplicity we have omitted object expressions.

Here is a typical JSON value:

```
["one", true, 3.4, [[ "four" ], [null]]]
```

We will start with the following "untyped"[5] representation of JSON data:

```
type ujson =
    UStr : string → ujson
  | UNum : float → ujson
  | UBool : bool → ujson
  | UNull : ujson
  | UArr: ujson list → ujson
```

We can use `ujson` to represent the JSON value above as follows:

```
# UArr [UStr "one"; UBool true; UNum 3.4;
      UArr [UArr [UStr "four"]; UArr [UNull]]];;
  - : ujson = ...
```

[4] http://json.org/

[5] The use of "untyped" to oppose the richly-typed data descriptions available with GADTs is a little tongue-in-cheek, but the `ujson` type actually bears quite a strong similarity to the kind of tagged universal data representation used in the implementations of "dynamically-typed" languages.

8.4.1 Pattern: richly typed data

Data may have finer structure than algebraic data types can express.
GADT indexes allow us to specify constraints more precisely.

Defining GADTs can be challenging because there are typically many ways to embellish a data type with additional information. In some cases we might wish to expose some particular aspect of a data structure, such as the depth of a tree, to the type level (Section 8.1). In other cases we may wish to use indexes of a type to propagate information about some other type, as we saw with `max` (Section 8.1.2), whose indexes store information about a relationship between triples of natural numbers, although the numbers themselves are not stored within `max` values. The `eql` type (Section 8.2) is perhaps the most striking example of the way that GADTs make it possible to define arbitrary relationships between types and data. At the data level its single nullary constructor gives `eql` the same representation as OCaml's `unit` type, which is so lacking in information that matching a value of type `unit` reveals nothing new at all. However, the type indexes of `eql` make it sufficiently powerful that it can be used to encode any other GADT (Section 8.2.1).

One common use of GADTs is giving rich types to data whose structure is specified externally. For such data the most common choice is to expose the general structure in the type indexes, after abstracting some details. Here is the example JSON datum again:

```
["one", true, 3.4, [[ "four" ], [null]]]
```

When defining a GADT to represent JSON we might like to expose some or all of the following facts about the structure of this value:

- The value contains `null`.
- The value contains five primitive subvalues
- When we ignore the distinction between primitive values, writing • in place of every primitive, the following structure remains:

  ```
  [•, •, •, [[ • ], [•]]]
  ```

- If we represent only the types of primitives, not their values, the following structure remains:

  ```
  [string, bool, number, [[ string ], [null]]]
  ```

There are many other possibilities, ranging from ignoring all distinctions and giving the same type to every value to attempting to give every JSON value a distinct type. We will take a middle ground, ignoring the distinctions between primitive values of a particular type, but reflecting the larger structure of values in the type level. Here are the resulting definitions for the types of JSON values and arrays of JSON values:

```
type _ tjson =
   Str : string → string tjson
 | Num : float → float tjson
 | Bool : bool → bool tjson
 | Null : unit tjson
 | Arr : 'a tarr → 'a tjson
and _ tarr =
   Nil : unit tarr
 | :: : 'a tjson * 'b tarr → ('a*'b) tarr
```

Using `tjson` we can represent our example JSON value and see its structure exposed in the types:

```
# Arr (Str "one" :: Bool true :: Num 3.4 ::
     Arr (Arr (Str "four" :: Nil) :: Null :: Nil) :: Nil);;
  - : (string * (bool * (float * (((string * unit) * (unit * unit)) * unit))))
    tjson
= ...
```

As we saw with functions over trees, we can learn more from the types of functions over richly typed data than from functions over types without indexes. For example, consider the following function type:

```
val utransform : ujson → ujson
```

We can learn almost nothing from this type about the behaviour of the function. Among the many other functions of this type are the identity function and the function which returns `Null` for every input. A similar function over `tjson` tells us a great deal more:

```
val ttransform : 'a tjson → 'a tjson
```

The fact that input and output indexes are the same allows us to deduce that `ttransform` preserves the general structure. It is not possible to write a function of this type that performs structure changing operations such as replacing numbers with strings or changing the lengths of list nodes.

8.4.2 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`.

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
  | UNum u → ETJson (Num u)
  | UBool b → ETJson (Bool b)
  | UNull → ETJson Null
  | UArr arr →
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Figure 8.17: Building typed JSON from untyped JSON using existentials

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))
```

Both the approach using existentials and the approach using universals extend well to more complex uses. Figure 8.17 shows how to use existential types to build a typed JSON representation from an untyped JSON representation. There are two new existential types, `etjson` and `etarr`, which hide the index of the `tjson` and `tarr` types, enabling us to return trees with different structure from the same function. The functions `tjson_of_json` and `tarr_of_uarr` traverse a `ujson` value to build a `tjson` value with the same structure.

Similarly, Figure 8.18 shows how to use polymorphism to building a typed JSON representation from an untyped JSON representation. There are two new polymorphic record types, `atjson` and `atarr`, whose single function members can accept arbitrary `tjson` and `tarr` values as arguments, enabling us to pass values with different indexes to the same function. The implementations of `tjson_of_json` and `tarr_of_uarr` in Figure 8.18 traverse a `ujson` value to build a `tjson` value with the same structure. The functions are written in a continuation-passing style: where there are multiple subnodes (as in the second case of `tarr_of_uarr`) the function passes the result of converting the first subnode to a continuation function which converts the second subnode and passes the result

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}

let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
  | UNum u → return (Num u)
  | UBool b → return (Bool b)
  | UNull → return Null
  | UArr arr →
    tarr_of_uarr arr {k = fun arr'
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Figure 8.18: Building typed JSON from untyped JSON using polymorphism

to a second continuation function, and so on until the last subnode is converted and the result of combining all the subnodes is passed to the function passed as argument.

8.4.3 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 which corresponding 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 which correspond to predicates like $\textsc{max}(1,3) = 3$, 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 132) lists a number of papers which explore how singletons support encoding dependently-typed programs in languages without dependent types.

8.4.4 Pattern: separating types and data

Entangling proofs and data can lead to redundant, inefficient code.
Separate proofs make data reusable and help avoid slow traversals.

The `tjson` type combines type-level descriptions of JSON data structures with constructors for building values of those structures. It is sometimes helpful to treat the type-level and value-level components separately, since the information they carry often becomes available at different times in a program's execution. For example, when we write bindings to an API that uses JSON we typically know the general structure of the values that will be passed back and forth, such as which fields are numbers or booleans. However, it is only when we actually make calls to the API that we have full information about the values that populate this structure.

We can separate out the the type-level and value-level components of `tjson` and `tarr` by eliminating the value arguments from constructor definitions, leaving only arguments which relate to type structure. The resulting type definitions have a similar structure and similar indexes to the original types, but have no way of storing primitive values:

```
type _ tyjson =
    TyStr : string tyjson
  | TyNum : float tyjson
  | TyBool : bool tyjson
  | TyNull : 'a tyjson → 'a option tyjson
  | TyArr: 'a tyarr → 'a tyjson
and _ tyarr =
    TyNil : unit tyarr
  | ::    : 'a tyjson * 'b tyarr → ('a * 'b) tyarr
```

We give the `tjson` style the slightly pejorative name *entangled* and call the `tyjson` style *disentangled*.

We have made one additional change to the type structure. Whereas `tjson` represents `null` as a primitive value, the `TyNull` constructor of the `tyjson` type describes *nullable* data. We are now indexing descriptions of data rather than particular data, and whereas a particular value is always `null` or not `null`, in a general description the idea of "possibly null" is much more useful than "definitely null".

We can use `tyjson` to represent our example JSON datum as a pair of values. The first describes the shape of the datum using the constructors and type indexes of `tyjson`. The second is built from standard OCaml types — bools, pairs, and so on — and contains the actual values of the datum. Comparing the types of the two constituents of the pair reveals that the type of the second matches the type index of the first:

```
# (TyArr (TyStr :: TyBool :: TyNum ::
          TyArr (TyArr (TyStr :: TyNil)
                 :: TyNull TyBool :: TyNil) :: TyNil),
  ("one", (true, (3.4, (((("four", ()), (None, ())), ())))));;
- : (string *
     (bool * (float * (((string * unit) * (bool option * unit)) * unit))))
    tyjson *
    (string *
     (bool * (float * (((string * unit) * ('a option * unit)) * unit))))
= ...
```

Functions in the entangled and disentangled styles As long as we deal with data descriptions and data together there is not a great deal of difference between the entangled and disentangled styles. Figures 8.19 and 8.20 show two implementations of a function which traverses a JSON structure negating every boolean node. The entangled implementation traverses the input value to find `bool` subnodes, passing everything else through unchanged. The disentangled implementation, `negateD`, is similar, but matches the description and the value in parallel, relying on type refinement to support writing patterns of different types, such as **`true`** and `None` in the value position.

The real difference between the entangled and disentangled styles appears when we need to write functions which examine the type structure before there is a value available. In the entangled style it is simply impossible to write such functions, since the type and value components are inextricably linked. In the

```
let rec negate : type a.a tjson → a tjson = function
   Bool b → not b
 | Arr arr → Arr (negate_arr arr)
 | v → v
and negate_arr : type a.a tarr → a tarr = function
   Nil → Nil
 | j :: js → negate j :: negate_arr js
```

Figure 8.19: The `negate` function, entangled

```
let rec negateD : type a.a tyjson → a → a =
  fun t v → match t, v with
   TyBool, true → false
 | TyBool, false → true
 | TyArr a, arr → negate_arrD a arr
 | TyNull j, Some v → Some (negateD j v)
 | TyNull _, None → None
 | _, v → v
and negate_arrD : type a.a tyarr → a → a =
  fun t v → match t, v with
   TyNil, () → ()
 | j :: js, (a, b) → (negateD j a, negate_arrD js b)
```

Figure 8.20: The `negate` function, disentangled

```
let id x = x
let map_option f = function None → None | Some x → Some (f x)

let rec negateDS : type a.a tyjson → a → a = function
   TyBool → not
 | TyArr a → negate_arrDS a
 | TyNull j → map_option (negateDS j)
 | _ → id
and negate_arrDS : type a.a tyarr → a → a = function
   TyNil → id
 | j :: js → let n = negateDS j
             and ns = negate_arrDS js in
                (fun (a, b) → (n a, ns b))
```

Figure 8.21: The `negate` function, disentangled and “staged”

```
let rec unpack_ujson :
  type a.a tyjson → ujson → a option =
  fun ty v → match ty, v with
    TyStr, UStr s → Some s
  | TyNum, UNum u → Some u
  | TyBool, UBool b → Some b
  | TyNull _, UNull → Some None
  | TyNull j, v → (match unpack_ujson j v with
                      Some v → Some (Some v)
                    | None → None)
  | TyArr a, UArr arr → unpack_uarr a arr
  | _ → None
and unpack_uarr :
  type a.a tyarr → ujson list → a option =
  fun ty v → match ty, v with
    TyNil, [] → Some ()
  | j :: js, v :: vs →
      (match unpack_ujson j v, unpack_uarr js vs with
         Some v', Some vs' → Some (v', vs')
       | _ → None)
  | _ → None
```

Figure 8.22: Building disentangled typed JSON from untyped JSON

disentangled style we can traverse the description of the structure to build a specialised function which operates only on data with the shape described.

Figure 8.21 gives a third implementation of `negate`, written in this style. The `negateDS` function has the same signature as `negateD`: its first argument is a description of data and its second argument a value matching the description. However, unlike `negateD`, `negateDS` is designed for partial application. Rather than matching on the description and the value in parallel it matches on the first argument as soon as that argument is received and returns a specialised function which operates on the type of data that the argument describes. Dividing the execution in this way offers potential for efficiency improvements, particularly when examining the description is an expensive operation.

We call the implementation of `negateDS` *staged* to reflect the fact that different parts of the computation potentially take place at different stages of the program's execution. We will return to this idea in Chapter 13, where we will examine staging as a first-class language feature.

Building typed values in the disentangled style There are further advantages to separating out descriptions of data from the actual data. It is typically easier to verify that data matches a particular shape than to determine the shape of arbitrary data. In particular, if we know the expected shape of data in advance then we can pass the shape as an argument to functions which construct typed data and avoid the tricks with existentials and polymorphism that we used in Section 8.4.2.

Figure 8.22 shows a function, `unpack_ujson`, which builds a typed JSON value from an `ujson` value and a description of the expected shape. As is common in the disentangled style, `unpack_ujson` matches description and value in parallel, relying on type refinement to support the differently-typed branches such as `Some s`, which has type `string option`, and `Some u`, which has type `float option`.

8.4.5 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.

Representing types built from strings and arrays The simple `is_zero` type is a useful replacement for `bool` in the return type of `is_empty`. As our predicates become more sophisticated our evidence becomes correspondingly more structured. The following example builds an inductive structure which offers evidence about the shape of a JSON value.

The `tyjson` type allows us to represent a wide variety of structures. It is sometimes helpful to work in a subspace of the possible representable values: for example, we might like to write functions that deal only with null-free JSON values, or with JSON values which do not contain numbers. The types `str_tyjs` and `str_tyarr` represent one such subset, namely the set of all JSON values which can be built from strings and arrays.

```
type _ str_tyjs =
    SStr : string str_tyjs
  | SArr : 'a str_tyarr → 'a str_tyjs
and 'a str_tyarr =
    SNil : unit str_tyarr
  | :: : 'a str_tyjs * 'b str_tyarr → ('a*'b) str_tyarr
```

The `str_tyjs` type has two constructors for representing the types of JSON strings and JSON arrays. The `str_tyarr`, like the `tyarr` type, has two constructors corresponding to an empty array and a cons operation for building arrays of JSON values. Using these constructors we can build a type that (for example) represents the structure of the value of an two element array which contains a string and an empty array:

```
# SArr (SStr :: SArr SNil :: SNil);;
- : (string * (unit * unit)) str_tyjs = ...
```

Figure 8.23 gives the implementation of a predicate which checks whether a JSON type is "stringy" — i.e. whether it involves only strings and arrays. If `is_stringy` is able to determine that the input type is stringy then it returns a value of the `is_stringy` type to serve as evidence. If the stringiness of the argument cannot be established then `is_stringy` returns `None`. The `stringyV` function in Figure 8.24 is similar to `is_stringy`, but operates on JSON values rather than on JSON types.

We can use the evidence returned by `is_stringy` and `is_stringyV` to reveal facts about a JSON value, just as we used `is_empty` to expose facts about trees. For example, suppose we have a function that operates only on stringy data:

```
val show_stringy : type a. a str_tyjs → a → string
```

Then we can pass a JSON value `v` to `show_stringy` only once we have established that the value is stringy.

```
let show_stringy : type a. a tyjson → a → string =
  fun ty v →
    match is_stringy ty with
      None → "(value cannot be displayed)"
    | Some evidence  → show_stringy evidence v
```

This time it is not type refinement that ensures that we can only call `show_stringy` in the branch, but the fact that only in the second branch is the evidence available.

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
   ```

```
let rec is_stringy : type a.a tyjson → a str_tyjs option =
 function
    TyStr → Some SStr
  | TyNum → None
  | TyBool → None
  | TyNull _ → None
  | TyArr arr → match is_stringy_array arr with
                None → None
              | Some sarr → Some (SArr sarr)
and is_stringy_array :
 type a.a tyarr → a str_tyarr option =
 function
    TyNil → Some SNil
  | x :: xs →
    match is_stringy x, is_stringy_array xs with
        Some x, Some xs → Some (x :: xs)
      | _ → None
```

Figure 8.23: Building evidence that a JSON type involves only arrays and strings

```
let rec is_stringyV : type a.a tjson → a str_tyjs option =
 function
    Str _ → Some SStr
  | Num _ → None
  | Bool _ → None
  | Null → None
  | Arr arr → match is_stringy_arrayV arr with
                None → None
              | Some sarr → Some (SArr sarr)
and is_stringy_arrayV :
 type a.a tarr → a str_tyarr option =
 function
    Nil → Some SNil
  | x :: xs →
    match is_stringyV x, is_stringy_arrayV xs with
        Some x, Some xs → Some (x :: xs)
      | _ → None
```

Figure 8.24: Building evidence that a JSON value involves only arrays and strings

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 $0 \dots 2^{n-1}$ 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. [★★★] The following code shows a common way of efficiently implementing queues in functional languages using a pair of lists:

   ```
   module type QUEUE =
   sig
     type _ t
     val empty: _ t
     val push : 'a → 'a t → 'a t
     val pop : 'a t → 'a option * 'a t
     val size: 'a t → int
   end

   module Queue : QUEUE =
   struct
     type 'a t = {
       inbox: 'a list;
       outbox: 'a list;
     }
     let empty = { inbox = []; outbox = [] }
     let push v q = {q with inbox = v :: q.inbox }
     let pop q = match q with
         { outbox = x :: xs } → Some x, { q with outbox = xs }
       | { outbox = [] } →
         match List.rev q.inbox with
           [] → None, empty
         | x :: xs → Some x, { outbox = xs; inbox = [] }
     let size {inbox;outbox} = List.length inbox + List.length outbox
   end
   ```

   Show how to implement the following signature of length-indexed queues by taking a similar approach, but using a pair of vectors (Question 5) rather than a pair of lists as the underlying data structure:

```
module type LQUEUE = sig
  type (_,_) t
  val empty: (_,z) t
  val push : 'a → ('a, 'n) t → ('a, 'n s) t
  val pop : ('a, 'n s) t → 'a * ('a, 'n) t
  val size : ('a, 'n) t → 'n
end
```

7. [★★★★] 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`?

8. [★★★★] 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)