

Principles of Programming Languages

Data Types and Operations on Data

In previous sections, we reviewed tools a programming language provides that help programmers design set of values that have common properties. The concrete form is a type language with which we can express type annotations. These type annotations denote sets of values. They are associated to variables or function parameters and function return values.

Type definitions help us:

- Enable type checking
- Document our intentions
- Structure the code that operates on complex values in a way that reflects the structure of the data type.

Reversely, programmers design and name types for sets of values that will be processed by the same set of functions. That is, type definitions allow the definition of uniform functional interfaces.

We illustrate this point through 4 examples:

- Homogeneous array types and the sequence interface (**map**, **filter**, **reduce**)
- Modeling trees
- Mutable data types in FP
- Disjoint types and disjoint unions to enable uniform functional interfaces

Array values can be homogeneous (all items have the same type) or heterogeneous (items of different types appear).

The *natural* way to operate over arrays is to use the sequence interface - that is, the set of higher-order functions which can be applied on arrays. For example:

- `map(f, array)`
- `filter(pred, array)`
- `reduce(reducer, initial, array)`

Homogeneous Array Types and the Sequence Interface

When we analyze the type of these functions, we realize that such operations will be easy and natural if the type of the function passed as a parameter is a simple type. For example:

```
import { map } from "ramda";
```

```
let arr = [1, 2, 3];  
map(x => x * x, arr);  
// => [1, 4, 9]
```

Homogeneous Array Types and the Sequence Interface

This operation over the array *works well* because the type of the mapper function is simple and can be derived from the type of the array:

- `arr` is of type `number[]`
- `x => x * x` is a function of type `number => number`, so it can be applied to all elements in `arr`.
- the resulting array is also of type `number[]`

Homogeneous Array Types and the Sequence Interface

Similarly, if we use a mapper function of type `number => boolean` - we know the return value of `map` will be `boolean[]`:

```
import { map } from "ramda";
```

```
let arr = [1, 2, 3];
```

```
map(x => x > 2, arr);
```

```
// => [false, false, true]
```

Operating over a heterogeneous array like `[1, "a", true]` would make the usage of these functions much more challenging - because the mapper function would need to know what to do for each type of parameter.

Homogeneous array types encourage the use of a simple functional interface - including **map**, **filter**, **reduce**. This functional interface (set of functions which operate on the same data structure) receive a function parameter with a simple type signature and abstract many forms of loops over repeated values of the same type.

Let us review the definition of a binary tree:

```
interface BinTree<T> {  
    root: T;  
    left?: BinTree<T>;  
    right?: BinTree<T>;  
}
```

When we write operations that operate over such trees, we must write code that operates according to the expected structure of the values in this type.

For example, let us write a function that traverses a **BinTree** in Depth-First order:

Modeling Trees with Types

```
const dfs: <T>(t: BinTree<T>) => void = t => {  
  console.log(t.root);  
  if (t.left !== undefined) {  
    dfs(t.left);  
  }  
  if (t.right !== undefined) {  
    dfs(t.right);  
  }  
};
```

The structure of this function follows the structure of the type definition - when we process a value of type **BinTree**, we know that accessing the field **t.root** is safe (will not return **undefined** or throw an exception). To access **t.left** we must first check whether it is **undefined** since the type allows for this (base case of the recursion). If it is not, we know it must be a value of type **BinTree**.

In addition to these assurances, we also know that after checking these conditions we have checked all possible configurations of values (that is, the function exhaustively covers all possible **BinTree** values).

Let us consider a variant task where we actually
build values of type **BinTree**:

Modeling Trees with Types

```
type N = number;

const square: (x: N) => N = x => x * x;
const sqrTree: (t: BinTree<N>) => BinTree<N> = t =>
  t.left !== undefined && t.right !== undefined ?
    { root: square(t.root),
      left: sqrTree(t.left),
      right: sqrTree(t.right) }
  : t.left !== undefined && t.right === undefined ?
    { root: square(t.root), left: sqrTree(t.left) }
  : t.left === undefined && t.right !== undefined ?
    { root: square(t.root), right: sqrTree(t.right) }
  : { root: square(t.root) };
```

Modeling Trees with Types

```
let t = {  
  root: 2,  
  left: {  
    root: 3,  
    left: { root: 4}  
  },  
  right: {  
    root: 5,  
    right: { root: 6 }  
  }  
};
```

```
sqrTree(t);
```

Modeling Trees with Types

```
{ root: 4,  
  left: { root: 9, left: { root: 16 } },  
  right: { root: 25, right: { root: 36 } } }
```

The function `sqrTree` operates over a `BinTree` value and creates a new `BinTree` return value. It considers all possible configurations of `BinTree` and invokes the function `sqrTree` recursively on each child value accordingly.

In this case as well - type analysis allows us to verify that all accesses to the fields of the **BinTree** value are *safe* and that all recursive calls pass values of the right type as parameters. In addition, we can verify that the function is *exhaustive* in checking all possible configurations for the value.

We can *relax* the type checking - and accept to receive values of type **undefined** in addition - yielding slightly shorter and more readable code:

Modeling Trees with Types

```
type N = number;

const square: (x: N) => N = x => x * x;
const sqrTree2: (t?: BinTree<N>) => BinTree<N> | undefined =
  t => t === undefined ? undefined
    : { root: square(t.root),
        left: sqrTree2(t.left),
        right: sqrTree2(t.right) };
```

Modeling Trees with Types

```
let t = {  
  root: 2,  
  left: {  
    root: 3,  
    left: { root: 4}  
  },  
  right: {  
    root: 5,  
    right: { root: 6 }  
  }  
};
```

```
sqrTree2(t);
```

Modeling Trees with Types

```
{  
  root: 4,  
  left: { root: 9,  
    left: { root: 16,  
      left: undefined,  
      right: undefined },  
    right: undefined },  
  right: { root: 25,  
    left: undefined,  
    right: { root: 36,  
      left: undefined,  
      right: undefined } }  
}
```

Observe that in this version:

- We extended the expected values of type `BinTree` to also include the value `undefined`.
- We explicitly test for this value as the first base case in the recursive function `sqrTree2`.

Observe that in this version:

- The recursive calls are now simplified as we don't need to avoid the recursive calls with a value **undefined**.
- The return value has values marked explicitly as **undefined** - these are semantically equivalent to absent values - but in the syntax of the object, they still appear.

We can get rid of the `undefineds` using a simple idiom:

```
JSON.parse(JSON.stringify(sqrTree2(t)));
```

The difference between the two versions is a matter of style preference. In general, the presence of undefined values complicates type analysis, but it is difficult to avoid dealing with it explicitly.

We indicated earlier that FP encourages immutable variables and data structures to achieve the goals of determinism (calling the same function with the same arguments should return the same value) and safe concurrency (avoid shared mutable data across threads).

Yet, some data types are **mutable** in their definition. Consider the example of a **stack**. It is defined as a container of values that enforces a specific access pattern through an interface:

- **push(x)**: modifies the stack by adding a new value **x** on top of it.
- **pop()**: modifies the stack by removing the top value of the stack and returning its value.
- **empty()**: determines whether the stack is empty.

As was discussed in SPL, in OOP, it is useful to split the Stack interface into **Queries** (functions that only return information about the data structure without changing it) and **Commands** (functions which only modify the data structure and do not return any value). Such distinction makes writing tests much easier.

To adopt this distinction, we split the **pop()** method into two distinct methods:

- **peek()**: returns the value of the top element in the stack (this is a query method).
- **pop()**: modifies the stack by removing the top element (this is a command method - that just has a side effect and no return value).

This definition is inherently procedural - as it defines the data type in terms of mutation (with the commands **push()** and **pop()**) in addition to the queries **peek()** and **empty()**.

Note that according to this methodology, even the queries are **not** pure functions - because they are not deterministic. Let us illustrate these points with a simple implementation of Stacks in TypeScript:

Mutable (Persistent) Data Types in FP

```
type Stack<T> = T[];
```

```
const makeStack: <T>(a: T[]) => Stack<T> =  
  a => a;
```

```
const peek: <T>(s: Stack<T>) => T =  
  s => s[0];
```

```
const empty: <T>(s: Stack<T>) => boolean =  
  s => s.length === 0;
```

```
const push: <T>(s: Stack<T>, t: T) => void =  
  (s, t) => { s.unshift(t); };
```

```
const pop: <T>(s: Stack<T>) => void =  
  s => { s.shift(); };
```

This implementation relies on an array encoding for Stack values. It relies on generic data types in TypeScript so that we can use it for Stacks of any type - as long as it is a homogeneous stack.

It relies on the fact that arrays in JavaScript are mutable - and implements `push()` using the primitive `unshift(x)` operation on arrays, and `pop()` using the primitive `shift()` operation.

(We skipped checking the preconditions to simplify the presentation.)

Note the specific style: commands (functions that have a side-effect - in this case **push()** and **pop()**) have no return value - we mark them as **void** in TypeScript.

In contrast, queries (functions that have no side-effect and only return information about the data structure - in this case **peek()** and **empty()**) return a value.

Given that the underlying data type is mutable,
we cannot obtain deterministic behavior:

Mutable (Persistent) Data Types in FP

```
let s = makeStack([1, 2, 3]);  
push(s, 0);  
console.log(peek(s)); // => 0  
pop(s);  
console.log(peek(s)); // => 1
```

The same operation `peek(s)` on the same variable `s` returns different values when mutation has occurred between the two calls.

Can we define a **functional data structure** for the Stack data type?

For example, can we define a Stack data structure that operates as an immutable data structure, while still offering the same interface to its clients?

The key change that is required to obtain such immutable functional data types is to modify the **commands** so that instead of mutating the existing data structure and “returning” **void**, the commands will return a new copy of the data structure.

This imposes first a change on the type of the functions, next a change on the client side. Let us illustrate this first round of changes (which we will find out is necessary but not sufficient):

Functional Stack: Step 1

```
type Stack<T> = T[];
```

```
const makeStack: <T>(a: T[]) => Stack<T> =  
  a => a;
```

```
const peek: <T>(s: Stack<T>) => T =  
  s => s[0];
```

```
const empty: <T>(s: Stack<T>) => boolean =  
  s => s.length === 0;
```

```
const push: <T>(s: Stack<T>, t: T) => Stack<T> =  
  (s, t) => { s.unshift(t); return s; };
```

```
const pop: <T>(s: Stack<T>) => Stack<T> =  
  s => { s.shift(); return s; };
```

This modification in the signature of the commands requires clients to change as well: each time a command is invoked, we must bind the return value to a new variable so that it can be used further:

Functional Stack: Step 1

```
let s1 = makeStack([1, 2, 3]);  
let s2 = push(s1, 0);  
console.log(peek(s2)); // => 0  
let s3 = pop(s2);  
console.log(peek(s3)); // => 1
```

In this new style, we do not observe direct mutation - the calls seem to be deterministic.

Unfortunately, this is an illusion:

```
const s1 = makeStack([1, 2, 3]);  
const s2 = push(s1, 0);  
const s3 = pop(s2);  
console.log(peek(s1)); // => 1  
pop(s1);  
console.log(s1); // => [2, 3]
```

The implementation relies on JavaScript arrays
- which are internally mutable. We did not prevent this mutation by just changing the signature of the methods, because in the body of the commands we still call mutators on the internal representation of the stack.

The situation is even worse because we have created a very risky situation called **variable aliasing** - the stacks **s2** and **s3** share in memory cells that are used by **s1**. As a result, operations on **s1** end up modifying the state of **s2**.

The reason the stacks `s2` and `s3` were changed when we applied mutation on `s1` is because the 3 stacks actually share parts of their value in memory - because arrays in JavaScript behave like pointers to values in C++.

The solution to this problem is to require that commands actually **copy** the data structure when they need to modify it - so that each returned value is indeed a new value - and not an **alias** of the previous value.

Functional Stack: Step 2

```
type Stack<T> = T[];
```

```
const makeStack: <T>(a: T[]) => Stack<T> =  
  a => a;
```

```
const peek: <T>(s: Stack<T>) => T =  
  s => s[0];
```

```
const empty: <T>(s: Stack<T>) => boolean =  
  s => s.length === 0;
```

```
const push: <T>(s: Stack<T>, t: T) => Stack<T> =  
  (s, t) => [t].concat(s);
```

```
const pop: <T>(s: Stack<T>) => Stack<T> =  
  s => s.slice(1);
```

Let us verify that this new implementation provides deterministic behavior for the Stack functions and no aliasing:

Functional Stack: Step 2

```
let s1 = makeStack([1, 2, 3]);  
let s2 = push(s1, 0);  
console.log(peek(s2)); // => 0  
let s3 = pop(s2);  
console.log(peek(s3)); // => 1  
pop(s1);  
console.log(s1); // => [ 1, 2, 3 ]  
console.log(s2); // => [ 0, 1, 2, 3 ]  
console.log(s3); // => [ 1, 2, 3 ]
```

This implementation is **safe** - it does not introduce unexpected side-effects to the data structures, and the data structures remain immutable.

The cost of this implementation, though, is that each mutation requires a full copy of the data structure. This is inefficient in RAM (we obtain many copies of the same objects) and in CPU (the copying operations are expensive).

We have highlighted the perspective of Data Types as denoting sets of values over which common operations can be performed. On the basis of this understanding, we defined operations over types which correspond to set operations - such as union and intersection.

Disjoint Types and Disjoint Union

Such operations are provided for example in TypeScript, and we can define types such as:

```
type NorS = number | string;  
type SorB = string | boolean;  
type S = NorS & SorB;
```

Type union can be used for example if we want to model the set of values that can be denoted by the JSON notation - as a recursive union of possible values:

Disjoint Types and Disjoint Union

```
type Json =  
  // Atomic values  
  | string  
  | number  
  | boolean  
  | null  
  // Compound values - maps and arrays  
  | Json[]  
  // A map where the keys are all strings  
  | { [property: string]: Json };
```

We have also defined that the type system implemented in TypeScript follows structural subtyping as opposed to nominal subtyping.

Disjoint Types and Disjoint Union

For example, if we define two types:

```
interface Person {  
  name: string;  
  address: string;  
}
```

```
interface Variable {  
  name: string;  
  address: string;  
}
```

```
let p: Person = { name: "Ben", address: "BS" };  
let v: Variable = p; // OK!
```

Under **nominal typing** (like it exists in Java for example), these two types would be disjoint - values of type **Person** and values of type **Variable** would be different.

Under **structural typing** (like it exists in TypeScript), these two types are actually equal - they describe the same set of values.

When modeling data types, we are often interested in distinguishing such types - so that the values we describe are distinct, and we cannot confuse a **Person** value with a **Variable** value.

Disjoint Types and Disjoint Union

The way to obtain this behavior is to add a discriminant field - called a **tag** - to distinguish values that are intended of being of different types.

```
interface Person {  
  tag: "person";  
  name: string;  
  address: string;  
}
```

```
interface Variable {  
  tag: "variable";  
  name: string;  
  address: string;  
}
```

Disjoint Types and Disjoint Union

```
let p: Person = {  
  tag: "person",  
  name: "Ben",  
  address: "BS"  
}
```

```
let v: Variable = p; // Does not compile!
```

Disjoint Types and Disjoint Union

In this example, the type of the **tag** field is a set of a single value - the string "**person**" or the string "**variable**".

With the addition of the **tag** field with these specifications, the two types **Person** and **Variable** have become **disjoint** - that is, the set of values these type annotations denote are disjoint.

The possibility to define disjoint types can be combined into a very common pattern called **disjoint union**.

In set theory, the disjoint union of two sets A and B is a binary operator that combines all distinct elements of a pair of given sets, while retaining the original set membership as a distinguishing characteristic of the union set.

$$A \uplus B = (A \times \{0\}) \cup (B \times \{1\})$$

For example:

$$\{0, 1, 2\} \uplus \{2, 3\} = \{(0, 0), (1, 0), (2, 0), (2, 1), (3, 1)\}$$

We identify in that operation that the elements $(0, 1)$ added to each pair play a role similar to the **tag** field we added to map types to make them disjoint.

In type descriptions, in order to define a disjoint union type, we define the union of two (or more) map types which are made disjoint by using the same **tag** field.

Disjoint Union

For example:

```
interface Point2D { x: number; y: number }  
type Shape = Circle | Rectangle | Triangle;
```

```
interface Circle {  
  tag: "circle";  
  center: Point2D;  
  radius: number;  
}
```

```
interface Rectangle {  
  tag: "rectangle";  
  upperLeft: Point2D;  
  lowerRight: Point2D;  
}
```

```
interface Triangle {  
  tag: "triangle";  
  p1: Point2D;  
  p2: Point2D;  
  p3: Point2D;  
}
```

These type definitions allow this type of processing - which is “case by case” processing of all the options in the disjoint union:

```
const area: (s: Shape) => number = s => {  
  switch (s.tag) {  
    case "circle": return ...  
    case "rectangle": return ...  
    case "triangle": return ...  
    // No need for "default" case  
    // because our cases are exhaustive  
  }  
}
```

The tool of disjoint union together with the corresponding **switch** construct achieves an effect similar to sub-classes with virtual classes in Object-Oriented Programming. It allows the function to dispatch to different computations based on the type of the actual value received as a parameter.

We prefer in the course the following functional notation instead of switch, using the simple conditional expression:

```
const area = (s: Shape): number =>
  s.tag === "circle" ? ... :
  s.tag === "rectangle" ? ... :
  s.tag === "triangle" ? ... :
  s // s is understood as a variable of type "never" here -
    // it can be returned without affecting the return
    // type of the function which remains `number`
```

The conditional expression when it is chained as in this example has the same semantic as a **switch** case where all the cases are exclusive.

The difference is that in the last step, the syntax of the expression requires an **else** case which is a-priori not needed.

The type inference mechanism of TypeScript recognizes that after each test, the type of the variable `s` has less and less possible options. In the last `else` option, however, `s` is known to have no possible values. In this case, it is understood to have a type called **`never`** (which denotes the empty set, the opposite of the type **`any`** which denotes all possible values).

In this case, we can just return **s**, which does not contradict the return type of the function, because we know that the function will return either one of the cases in the first 3 clauses (which is **number**) or **never** – altogether, the function returns **number** union **never** which is **number**.

Both the usage of `switch` and the conditional expression are safe against future changes in the code: if the programmer extends the definition of the type `Shape` with another option, then the code for the function `area` will not compile anymore.

The definition of the union type in this specific context makes sense because it expresses the intention of the programmer:

- these are disjoint types - they have no commonality in structure
- but they have a similar functional interface
 - we *consume* them in a similar manner.

Note that the type checker can determine that the **switch** construct covers all possible options - based on the structure of the type union - and for each case, it can check the expected keys based on the value of the **tag** key.