19. ALGEBRAIC DATA TYPES

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

January 2025

In programming, types are a useful instrument for obtaining correctness guarantees and ensuring robustness.

In this lecture, we will study the concept of Algebraic Data Types; the term "algebraic" comes from the fact that the *constructors* of these types form an **algebra**. We won't concern ourselves much with this aspect; for us, ADTs are just types with one or more *constructors*, each constructor having zero or more *arguments*.

Instead of discussing abstractly about how these types look in general, let's dive directly into a very common example, lists.

1 The "List" ADT

The `List` ADT has two constructors, which capture the idea that a list is either empty, or consists of some element added to another list:

`Empty`: $List$
`Cons`: $\mathbb{N} \times List \to List$

`Empty` is a *nullary* constructor (no arguments), and `Cons` is a *binary* constructor (two arguments).

The list consisting of the numbers `1, 2, 3`, in this order can be expressed as:

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

It might help to read it from the right. First we have the empty list, which contains no elements.

Then, `Cons(3, Empty)` is the list consisting of the element 3 added to the empty list.

Then, `Cons(2, Cons(3, Empty))` is the list consisting of the element 2 followed by the list above.

Lastly, `Cons(1, Cons(2, Cons(3, Empty)))` is the list consisting of the element 1, followed by the list above.

> **Note 19.1.** The expression above is a well-formed mathematical object. The fact that it refers to the list `[1, 2, 3]` is an *interpretation* and it is somewhat arbitrary. We could just as well claim that it refers to the list `[3, 2, 1]`, as long as we're consistent with this interpretation and modify our subsequent definitions accordingly.

Now that we have defined a data type, we can introduce *operators*. "Operator" is just a fancy name for a function; it has a domain, a codomain and a definition. The definition of the operators is grouped in one or more *axioms*. Our first operator, `isEmpty` ranges over lists, mapping them to a boolean value which shows whether they are the empty list or not.

`isEmpty`: $List \to \{FALSE, TRUE\}$

($IE1$) `isEmpty`$(Empty) = True$
($IE2$) `isEmpty`$(Cons(x, xs)) = False$

The first line tells us the name, the domain and the codomain of our operator.

Next, we have the two defining axioms. We choose to name these axioms "`IE1`" and "`IE2`". These names will comes in handy when writing proofs, as we will need to refer to the axioms. We will get to the that in the next lecture, when we will study structural induction.

The first axiom, IE1, tells us that the list Empty is empty. The second axiom, IE2, tells us that some list consisting of an element x added to a list xs[†] is not empty.

Who are x and xs in the last axiom? Well, x is some element, a member of E and xs is some list, a member of $List$; that's all that is relevant. The operator isEmpty is not concerned with the particular elements of a list, just with the overall "structure" of he list. Thus, you should view axiom IE2, as being preceded by: $\forall x \in E, \forall xs \in List$. We will usually omit such quantifiers, for brevity.

At this point, you might well doubt the usefulness of operator isEmpty. The implicit usefulness, relating to programming, would be to offer the programmer a way to check whether some arbitrary list l is empty or not. Couldn't a programmer just check if $l = Empty$?

Well, what does "=" mean here? What does it mean for two lists to be equal? You might have an intuition about this and you might even deem it obvious, but the important fact is that we have not *defined* what equality is. All we have is a data type. An equality test should come in the form of an explicitly defined operator:

---

listEqual : $List \times List \to \{FALSE, TRUE\}$

(EQ1) $\texttt{listEqual}(Empty, l) = isEmpty(l)$
(EQ2) $\texttt{listEqual}(l, Empty) = isEmpty(l)$
(EQ3) $\texttt{listEqual}(Const(x, xs), Cons(y, ys)) = (x = y \wedge \texttt{listEqual}(xs, ys))$

The first two axioms, EQ1 and EQ2, regard the equality between the empty list and an arbitrary list l (again, you can think of this axiom as being preceded by a $\forall l \in List$). It hinges on the definition of isEmpty.

The third axiom, $EQ3$, concerns the equality between two non-empty lists, whose heads are x and y and whose tails are xs and ys, respectively. These two lists are equal only if they have the same first element and their tails are equal. Notice that we compare the heads using the operator "="; here, this has the meaning of the ordinary equality comparison on natural numbers, with which we're familiar. What's interesting is the second operand of the logical and: here we compare the tails of the lists using the very operator that we're defining! This makes the definition of our listEqual *recursive*. Recursive definitions are very common for lists, and for ADTs in general.

Here's another example, of an operator that maps a list to its length:

---

length : $List \to \mathbb{N}$

(LEN1) $\texttt{length}(Empty) = 0$
(LEN2) $\texttt{length}(Cons(x, xs)) = 1 + \texttt{length}(xs)$

The first axiom, LEN1, tells us that the empty list has no elements. The second one is more exciting: it tells us that the length of a list with head x and tail xs is one more than the length of the tail, using the very operator that we're defining!

To append a list to another:

---

append : $List \times List \to List$

(APP1) $\texttt{append}(Empty, l) = l$
(APP1) $\texttt{append}(Cons(x, xs), l) = \texttt{Cons}(x, \texttt{append}(xs, l))$

For example, given lists $[1, 2, 3]$ and $[4, 5, 6]$, the result of appending the second to the first is $[1, 2, 3, 4, 5, 6]$ (try working out "step by step", based on the two axioms)

2 Booleans

In the previous section we've used the boolean values FALSE and TRUE; but we can also treat these as two nullary constructors of a new ADT:

False : $Bool$
True : $Bool$

[†]The separation of a list into two parts with names like x and xs, or y and ys, is a common idiom. The name xs is is actually the English plural of x, and should be read as "/exes/".

We can then define familiar operators:

$\texttt{and}: \; Bool \times Bool \to Bool$

(AND1) $\texttt{and}(False, False) = False$
(AND1) $\texttt{and}(False, True) = False$
(AND1) $\texttt{and}(True, False) = False$
(AND1) $\texttt{and}(True, True) = True$

---

3 Tuples

A tuple is an n-ary collection of elements. The difference between a list and a tuple is that, while all lists are values of the same ADT, tuples are a family, with a separate data type for each n.

For example, for $n = 2$, we call such tuples "*pairs*". A pair

---

$\texttt{P}: \; \mathbb{N} \times \mathbb{N} \to Pair$

P is our first example of an *external constructor*; i.e. a non-nullary constructor (so it has some arguments), but neither argument is a member of the ADT that it's defining. All other non-nullary constructors up until now have been *internal*. This distinction will be relevant for us during the next lecture, when we will be studying structural induction.

We can define *getters* that retrieve the elements of a pair:

$\texttt{first}: \; Pair \to \mathbb{N}$

(FST) $\texttt{first}(P(x, y)) = x$

$\texttt{second}: \; Pair \to \mathbb{N}$

(SND) $\texttt{second}(P(x, y)) = y$

We can also define other operations, such as summing the elements of the pair:

$\texttt{pairSum}: \; Pair \to \mathbb{N}$

(PSUM) $\texttt{pairSum}(P(x, y)) = x + y$

---

4 Natural numbers

Up until now, we have used natural numbers and some arithmetical operations in the definitions of our ADTs. We can also define natural numbers as a recursive ADT. A natural number is either the number zero, or *the successor* of another natural numbers. This definition is based on Giuseppe Peano's axioms for defining natural numbers.

$\texttt{Zero}: \; Natural$
$\texttt{Succ}: \; Natural \to Natural$

The number one, is simply $Succ(Zero)$; the number two is $Succ(Succ(Zero))$ and so on.

We can now define arithmetical operations, such as addition and multiplication:

$\texttt{add}: \; Natural \times Natural \to Natural$

(ADD1) $\texttt{add}(Zero, n) = n$
(ADD2) $\texttt{add}(Succ(m), n) = Succ(\texttt{add}(m, n))$

---

mult : $Natural \times Natural \to Natural$

(MUL1) $\texttt{mult}(Zero, n) = Zero$
(MUL2) $\texttt{mult}(Succ(m), n) = \texttt{add}(n, \texttt{mult}(m, n))$

---

5 Binary trees

A binary tree is either a node with two children that contains a particular value, or the nil tree, containing no value and having no children.

Nil : $BTree$
Node : $\mathbb{N} \times BTree \times BTree \to BTree$

Note that a "leaf" under this definition is simply a node whose children are both Nil.

Like for lists, we can start with an operator which checks whether a tree is nil or not:

---

isNil : $BTree \to \{FALSE, TRUE\}$

(IN1) $\texttt{isNil}(Nil) = True$
(IN2) $\texttt{isNil}(Node(x, l, r)) = False$

We can write more interesting operators, for example to calculate the size of the tree, i.e. the number of nodes. Note that this is similar to `length` for lists:

---

size : $BTree \to \mathbb{N}$

(SZ1) $\texttt{size}(Nil) = 0$
(SZ2) $\texttt{size}(Node(x, l, r)) = 1 + size(l) + size(r)$

---

height : $BTree \to \mathbb{N}$

(H1) $\texttt{height}(Nil) = 0$
(H2) $\texttt{height}(Node(x, l, r)) = 1 + \max(size(l), size(r))$

---

6 Queue

QEmpty : $Queue$
Enqueue : $\mathbb{N} \times Queue \to Queue$

---

peak : $Queue \to \mathbb{N}$

(PK1) $\texttt{peak}(\texttt{Enqueue}(x, \texttt{QEmpty})) = x$
(PK2) $\texttt{peak}(\texttt{Enqueue}(x, \texttt{Enqueue}(y, ys)) = \texttt{peak}(\texttt{Enqueue}(y, ys))$

We cannot peak at the top element if there are no elements; in other words, the `peak` operator is undefined for `QEmpty`.

---

dequeue : $Queue \to Queue$

(DQ1) $\texttt{dequeue}(\texttt{Enqueue}(x, QEmpty)) = QEmpty$
(DQ2) $\texttt{dequeue}(\texttt{Enqueue}(x, xs)) = \texttt{Enqueue}(x, \texttt{dequeue}(xs))$

---

7 Map

A "*map*", or "*dictionary*" or "*associative array*" is a structure that stores (key, value) pairs.

In our example, both the key and the value are natural numbers. Although usually, this data structure is interesting for its implementation that have faster-than-linear insertion and searching, we will not concern ourselves with efficiency; search will be done in linear time, by potentially going through each entry, one by one.

MEmpty : Map
Insert : $\mathbb{N} \times \mathbb{N} \times Map \to Map$

---

exists : $Map \times \mathbb{N} \to \mathbb{N}$

$(EX1)$ $\texttt{exist}(\texttt{Insert}(k', k, v, \texttt{MEmpty}) = FALSE$
$(EX2)$ $\texttt{exist}(\texttt{Insert}(k', k, v, m) =$ if $k' = k$ then $TRUE$ else $\texttt{exist}(k', m)$

---

get : $Map \times \mathbb{N} \to \mathbb{N}$

$(GT1)$ $\texttt{get}(\texttt{Insert}(k, v, m), k') =$ if $k' = k$ then v else $\texttt{get}(k', m)$

---

8 A note on the term "algebraic"

The term "algebraic" refers to the fact that the methods of constructing these data types (multiple constructors/-multiple arguments) form an algebra.

Consider a type with a single constructor that has two arguments, modelling pair of booleans:

BoolP : $Bool \times Bool \to BoolPair$

There are two possible $Bool$ *values*, so there are $2 \times 2 = 4$ possible $BoolPair$ *values*:

```
BoolP(False, False)
BoolP(False, True)
BoolP(True, True)
BoolP(True, False)
```

In other words, the two-argument constructor ends up creating the *cartesian product* between its two types; and that also applies for types with infinite values, such as natural numbers.

Adding another constructor extends our type with those values produced by the new constructor, which is akin to addition. The next, rather artificial ADT "$BoolOrBoolPair$", models either a single boolean or a pair of booleans, employing two constructors:

BSingle : $Bool \to BoolOrBoolPair$
BPair : $Bool \times Bool \to BoolOrBoolPair$

There are now $2 + 4 = 6$ possible values – the union between the sets of values generated by each constructor.

```
BSingle(False)
BSingle(True)
BPair(False, False)
BPair(False, True)
BPair(True, True)
BPair(True, False)
```

---

9 References and further reading

The historical survey *A History of Haskell: Being Lazy with Class*[‡] by Paul Hudak, John Hughes, Simon Peyton Jones and Philip Wadler [1] traces the origins of algebraic data type to the works of Rod Burstall and John Darlington: the 1969 *Proving properties of programs by structural induction* [2] and the 1977 *A transformation system for developing recursive programs* [3].

Embedding ADTs in a programming language is attributed to the language HOPE, presented in the 1980 article: *Hope: An Experimental Applicative Language* [4], by Rod Burstall, David MacQueen and Donald Sannella.

Independent researcher Li-yao XIA traces the first occurrence of the term "algebraic data type"[§] to David Turner's 1985 article: *Miranda: A non-strict functional language with polymorphic types* [5].

Bibliography

[1] Paul Hudak et al. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 12–1.

[2] Rod M Burstall. "Proving properties of programs by structural induction". In: *The Computer Journal* 12.1 (1969), pp. 41–48.

[3] Rod M Burstall and John Darlington. "A transformation system for developing recursive programs". In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 44–67.

[4] Rod M Burstall, David B MacQueen, and Donald T Sannella. "HOPE: An experimental applicative language". In: *Proceedings of the 1980 ACM conference on LISP and functional programming*. 1980, pp. 136–143.

[5] David A Turner. "Miranda: A non-strict functional language with polymorphic types". In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1985, pp. 1–16.

[‡]Haskell is a functional programming language that employs ADTs; you will study it on the second semester, at the *Programming Paradigm* course

[§]`https://blog.poisson.chat/posts/2024-07-26-adt-history.html`