ADVANCED PROGRAMMING LANGUAGES
Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

Algebraic Data Types

With the development of the last few lectures, we have most of our language fundamentals in place: functions, primitive types, their reduction rules and a sound type system. Now it's time to look at compound types, to account for richer representations of data structures such as those available in practical programming languages.

We start with tuples, records and variants, conventionally referred to as **algebraic data types – ADTs**.
The two fundamental mechanisms that support datatypes are **constructors** and **selectors**:

- a datatype **constructor** creates the values of the datatype by assembling multiple components;
- datatype **selector** extracts the components assembled in the datatype.

As we will discuss shortly, *algebraic* refers to the constructors employer to assemble their components, which are are qualified as *algebraic* as they are reminiscent of the product and sum operators found in algebra.

Datatype Constructors

Constructors are the fundamental mechanism to build datatypes: they are best understood as functions that *inject* values from one or more domain (type) into the domain of the datatype being defined.

Tuples, records and variants are all created by corresponding constructors. We look at each of them below.

Tuples

Tuples are the familiar structures we are used to see in mathematics. In formal systems as well as in programming languages, they are introduced as **structural types** which, for each $n \in 0, 1, 2, \ldots,$

are built by the n-ary constructor $(\cdot)_n : T_1 \times \cdots \times T_n \to (T1, \ldots, T_n)$ that collects the n components (of the respective types) and injects then in a new compound value structured as an n-tuple. Thus we have:

$(\mathsf{true}, \mathsf{false}) : (\mathsf{Bool}, \mathsf{Bool})$
$(\text{"}\mathsf{aaa}\text{"}, \mathsf{true}) : (\mathsf{String}, \mathsf{Bool})$
$(1, 5, 2, 10) : (\mathsf{Int}, \mathsf{Int}, \mathsf{Int}, \mathsf{Int})$
$(3, \mathsf{true}, 2.3, \text{"}\mathsf{aaa}\text{"}) : (\mathsf{Int}, \mathsf{Bool}, \mathsf{Double}, \mathsf{String})$
$(\text{"}\mathsf{bb}\text{"}, \lambda(\mathsf{x} : \mathsf{Bool}).\mathsf{x} \,\|\, \mathsf{true},) : (\mathsf{String}, \mathsf{Bool} \to \mathsf{Bool})$

It does not take long to realize that for each n, we must either presuppose that a type-indexed family of constructors exists which includes a specific constructor for each of the infinitely many possible combination of the component types, or else assume that for each n there exists a single, polymorphic n-ary constructor that may be applied to any combination of the component types.

In either case, the typing relation may be formalized by the following rule:

$$\frac{\Gamma \vdash M_1 : T_1 \quad \cdots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash (M_1, \ldots, M_n) : (T_1, \ldots, T_n)} \ (\text{T-Tuple})$$

Alternatively, we may take n-tuples as being built as the composition of multiple applications of a (type indexed, or polymorphic) constructor for *pairs*. For instance, the triple type (T_1, T_2, T_3) may be constructed as the type $((T_1, T_2), T_3)$ or else as the type $(T_1, (T_2, T_3))$: in any case, the three types are different and, as we discuss shortly, the different ways they are constructed, affects the way we define the selector operators, to access the tuple components.

Records

Records are a generalization of tuples, in which each component (field) is associated with a label drawn from a predefined set L (in programming languages are simply identifiers). Formally, given n and a set of *distinct* labels $\ell_1, \ldots, \ell_n$, a record is built by a constructor $\{\ell_1, \cdots, \ell_n\} : T_1 \times \cdots \times T_n \to \{\ell_1 : T_1, \cdots, \ell_n : T_n\}$, which collects the n components (of the respective types) and associates them with the corresponding labels within the record.

As for tuples, we must presuppose that the record constructors are either type indexed, or polymorphic. In either case, the typing rule has the following general format:

$$\frac{\Gamma \vdash M_1 : T_1 \quad \cdots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash \{\ell_1 = M_1, \ldots, \ell_n = M_n) : \{\ell_1 : T_1, \ldots, \ell_n : T_n\}} \ (\text{T-Rec})$$

Given that components are identified by their labels, and the labels are all distinct, their order is irrelevant: as a result, records and record types that are equal up to the reordering of their fields are equated. Before proceeding with a detailed discussion on datatypes, we take a short pause to to examine one fundamental stylistic difference between two different approaches to introducing such datatypes. This difference concerns the status of *type names*.

Variants

In many situations we deal with compound structures that may have different, alternative shapes. For example:

- a node in a binary tree can be either a leaf or an interior node with two children;
- a list cell can be either nil or a *cons cell* carrying a head and a tail;
- a node of an abstract syntax tree in a compiler can represent a variable, an abstraction, an application, etc.

The foundational mechanisms that support this kind of programming are *variants* and variant types. We illustrate the idea with a simple example.

Suppose we want to define a new type of *shapes*, where a *shape* can be either a *circle* or a *rectangle*. We introduce the new type with the structural definition below:

$$\mathsf{Shape} = \langle\ \mathsf{Circ}(\mathsf{Double}) \mid \mathsf{Rect}(\mathsf{Double}, \mathsf{Double})\ \rangle$$

Here Shape is just an alias for the new variant type expression $\langle\ \mathsf{Circ}(\mathsf{Double}) \mid \mathsf{Rect}(\mathsf{Double}, \mathsf{Double})\ \rangle$: the type expression, in turn, introduces a new domain as the disjoint union of two subdomains built by the two data constructors Circ and Rect .

When it comes to typing, the purely structural presentation of variant types we adopted above poses the problem of associating the elements of the different subdomain to the correct variant.

Unlike records and tuples, whose structure is entirely determined by they syntactic form, each constituent of a variant only accounts for one of the subdomains and provides no information on the other subdomains. In other words, each subdomain can be thought of as part of different, possibly unrelated variants.

To see the problem, simply consider that in a purely structural setting, the following all look like perfectly valid judgements:

$$\begin{array}{lcl}
\mathsf{Circ}(3.0) & : & \langle\ \mathsf{Circ}(\mathsf{Double})\ \rangle \\
\mathsf{Circ}(3.0) & : & \langle\ \mathsf{Circ}(\mathsf{Double}) \mid \mathsf{Rect}(\mathsf{Double}, \mathsf{Double})\ \rangle \\
\mathsf{Circ}(3.0) & : & \langle\ \mathsf{Circ}(\mathsf{Double}) \mid \mathsf{Square}(\mathsf{Double})\ \rangle \\
\mathsf{Circ}(3.0) & : & \langle\ \mathsf{Circ}(\mathsf{Double}) \mid \ldots\ \rangle
\end{array}$$

In a system with subtyping, this could be acceptable, as we could reasonably assume that variant types are ordered, and take $\langle\ \mathsf{Circ}(\mathsf{Double})\ \rangle$ as the minimal type for $\mathsf{Circ}(3.0)$. But without subtyping, all the above types are unrelated and the fact that the judgements are all equally reasonable leaves us no way to elect a unique (or at least preferred) type. To recover type uniqueness, we resort to a mechanism known as type *ascription*, a syntactic device that annotates variants values with the intended variant type to drive type checking.

As a result, variant values must always carry a type tag, e.g. as in $\mathsf{Circ}(3.0)@\langle\mathsf{Circ}(\mathsf{Double}) \mid \mathsf{Square}(\mathsf{Double})\rangle$, and the structural typing rule for variant types may then be defined as follows:

$$\frac{\Gamma \vdash M : T_j \quad (j \in 1..n)}{\Gamma \vdash \mathsf{C}_j(M)@\langle\ \mathsf{C}_1(T_1) \mid \ldots \mid \mathsf{C}_n(T_n)\ \rangle : \langle\ \mathsf{C}_1(T_1) \mid \ldots \mid \mathsf{C}_n(T_n)\ \rangle}\ (\text{T-Variant})$$

Nominal presentations, such as the ones we will look at in our next lecture , solve the problem without any need for annotations, as they can count on the type declarations to associate the datatype values with the correct variant.

Pattern Matching — A uniform selection mechanism

We've already seen pattern-matching at work in a few specific cases in our overview of Scala. Now, we present it for what it really is, that is a fully general mechanism for selecting datatype components.

Fundamentally, pattern-matching is a mechanism for taking apart a value by *finding out which constructor* it was built with and *which components the constructor was applied to*. Based on that, it provides a uniform (in some cases the *only*) way to select the constituents of a compound datatype.

Patterns

Given a datatype constructor, a *pattern* is essentially a *semi-constructed* term (value), in which the structure of some of the components are left unspecified and represented by variables. Formally, we represent patterns with the notation $P[x_1, \ldots, x_n]$ indicating a term P some of whose sub-terms are the variables $x_1, \ldots, x_n$: the pattern is meant to represent all the terms

$P[M_1, \ldots, M_n]$ that result from substituting the pattern variables with *proper* terms $M_1, \ldots, M_n$.

This leads rather directly to a general definition of *pattern matching*: we say that a term M *matches* a pattern $P[x_1, \ldots, x_n]$, if there exist terms $N_1, \ldots, N_m$ such that $M \equiv P[N_1, \ldots, N_m]$.

We distinguish two main kinds of patterns.

Variable Patterns. A variable x is a special case of pattern that represents (hence is matched by) all terms. Variable patterns may also be structured as *wildcard patterns* using the anonymous variable (the wildcard) "_". As we will see shortly, using wildcards in place of variables affects the semantics of matching expressions

Constructor Patterns. Given any constructor, any semi-constructed term built with the constructor is a pattern. Examples:

- $(\mathsf{false}, \mathsf{x}, 1, \_, \mathsf{z})$ is a 5-tuple pattern, matched by any 5-tuple that has false and 1 in the first and third position respectively. Notice that wildcards and variables may coexist within a pattern.
- $(\mathsf{false}, \mathsf{x}, 1, (\mathsf{y}, 3))$ is a 4-tuple (nested) pattern, matched by any quadruple whose first, third and fourth element are false, 1 and a pair whose second element is 3 respectively;
- $\{\mathsf{fst} = 5, \mathsf{snd} = \mathsf{x}\}$ is a record pattern, matched by all pairs whose fst field is the integer 5
- $\mathsf{Cons}(5, \mathsf{Cons}(\mathsf{x}, \mathsf{Cons}(3, \mathsf{xs})))$ is a $\mathsf{IntList}$ pattern matched by all lists with at least three elements, whose first and third elements are the integers 5 and 3.

Existing programming languages provide further pattern forms, which we will discuss in the context of Scala in the next lecture. Here, we move on to complete the discussion on pattern matching by introducing matching expressions.

Matching expressions

The formal device for putting patterns and composite datatype at work is a special-purpose expression which we refer to as *matching expressions*. Different programming languages employ different syntax and naming for such expressions, but the fundamental mechanism is the same: a term M is matched against a pattern $P[x_1, \ldots, x_n]$ and if the match is successful, a new binding is created for the rest of the computation between each variable in P and the corresponding sub-term in M.

The general form of a matching expression is as follows:

$$M \text{ match } P_1[x_{1,1}, \ldots, x_{1,k1}] \Rightarrow M_1 \mid \cdots \mid P_n[x_{n,1}, \ldots, x_{n,kn}] \Rightarrow M_n$$

The variables occurring in each pattern may occur free in the corresponding terms to the right for the arrow. In Then, when the expression is evaluated:

- the term M is matched against each of the patterns $P_1, P_2, \ldots P_n$
- if a matching pattern P_i is found, then $M \equiv P_i[N_{i,1}, \ldots, N_{i,ki}]$ for suitable terms $N_{i,1}, \ldots, N_{i,ki}$
- then the evaluation proceeds with the term obtained from M_i by substituting all the free occurrences of the pattern variables with the corresponding subterms found in M.

The following reduction formalizes this behavior. For brevity, we present the simpler case in which patterns have all just one variable. The generalization to patterns with multiple variables is straightforward.

$$P_i[N_i] \text{ match } (P_1[x_1] \Rightarrow M_1 \mid \cdots \mid P_n[x_n] \Rightarrow M_n) \;\Rightarrow\; [N_i/x_i]M_i$$

If no pattern is found that matches the term being tested, the whole matching expression is stuck. Notice, however, that in principle the term could match more than one pattern, resulting in different possible reductions for the matching expressions. Since this would clearly break one of the fundamental properties of reduction (the first Church-Rosser theorem) foundational calculi require that the patterns occurring in a matching expressions be *disjoint*, so that the term being tested matches at most one pattern.

Programming languages, instead, take a more pragmatic approach in which (i) the patterns are considered for matching sequentially, following the order in which they occur in the matching expression, and (ii) only the first matching clause is selected for reduction. If no pattern is found, the expression returns an error: this situation is ruled out in practice, as checking that the patterns *cover* the term can easily be accomplished with a static check if the type of the term to be tested is known.

The typing rule for matching expressions is so defined as to ensure that types are preserved by the reduction. Again, in the interest of brevity, we present the rule for the case of single variable patterns:

$$\frac{\Gamma \vdash M : T \qquad \Gamma, x_1 : T_1 \vdash (P_1, M_1) : (T, T) \quad \cdots \quad \Gamma, x_n : T_n \vdash (P_n, M_n) : (T, T)}{\Gamma \vdash M \text{ match } (P_1[x_1] \Rightarrow M_1 \mid \cdots \mid P_n[x_n] \Rightarrow M_n) : T} \text{(T-Match)}$$

We conclude the discussion on pattern matching with two simple Scala examples that manipulate the variant datatypes introduced earlier.

```
def area(s:Shape) = s match
   case Circle(r) => Math.pow(r,2) * Math.PI
   case Rectangle(b,h) => b * h
```

```
def len(as : IntList) : Int = as match
   case Empty => 0
   case Cons(a,as) => 1 + length(as)
```

The function `area` takes as a parameter the type `Shape` and it uses pattern matching to figure out whether it's a `Circle` or a `Rectangle`) to compute the area accordingly. The `len` function operates similarly on type `IntList` to compute the list length.

Datatype-specific selectors

Though pattern matching provides a uniform, fully general scheme for selecting the constituents of compound datatypes, tuples and records come equipped with specific selectors in most popular programming languages.

In Scala and various dialects of ML, for instance, the tuple components can be accessed by the positional selectors `_n` we have illustrated in our brief overview of Scala. Haskell is an exception in that it only provides the **fst** and **snd** selectors for pairs, and pattern matching for general tuples.

As for records, most languages, including Scala, adopt the familiar *dot notation* to extract the values associated with the record fields. Haskell is again the exception: given a record with fields $\ell_i : T_i$, each label ℓ_i is construed as function whose domain is the record type and the codomain is the type T_i. Thus, given the type:

```
data Address = Addr { city :: String, street :: String, number :: Int }
```

Haskell generates three functions:

```
city   :: Address => String
street :: Address => String
number :: Address => Int
```

Then given the record `a = Addr { city = "Boston", street = "Worcester", number = 15 }`, the fields are accessed by the function applications `city a`, `street a` and `number a`.

Exercises

1. Reduction and typing for datatype-specific selectors. Assume we extend our foundational presentation of datatypes with syntactic forms to express specific operators for selecting the components of tuples and records.

$$M ::= \cdots \mid M.\_j \mid M.\ell_j \mid \cdots$$

Define the reduction and typing rules to capture the intended semantics and typing relation for the new syntactic forms.

2. Option types. One very useful idiom involving variant types is optional values. For example, an element of the type

$$\mathsf{OptionNat} = \langle \mathsf{NaN} \mid \mathsf{Some}(\mathsf{Nat}) \rangle$$

is either the trivial NaN value or else natural number wrapped inside the constructor Some. Define the Scala implementation of the type OptionNat.

3. Computing with Options. Based on the datatype definition of the previous exercise, define a Scala function

$$\mathsf{succ} : \mathsf{OptionNat} \to \mathsf{OptionNat}$$

that operates as the conservative extension of the successor function to the type OptionNat.

As further use of optional values, consider the following type representing represents finite mappings from numbers to numbers

$$\mathsf{FiniteMap} = \mathsf{Nat} \to \mathsf{OptionNat}$$

The domain of such a mapping is the set of inputs for which the result is $\mathsf{Some}(\mathsf{n})$. Define the Scala implementation of the type FiniteMap and of the following two functions:

- emptyMap : FiniteMap representing the map with empty domain
- extendMap : FiniteMap $\to$ Nat $\to$ Nat $\to$ FiniteMap that, given a finite map M and and two natural numbers n, v, returns a new map which extends (or modifies) M with an entry mapping the n to v

CREDITS

The discussion on new types and data types in the lambda calculus is adapted from Chapter 11 of Prof. Benjamin's Pierce's book on Types and Programming Languages. MIT Press. 2001. The section of nominal vs structural typing is taken from Chapter 19.3 of the same book.