Algrbraic Data Types

Luca Abeni
luca.abeni@santannapisa.it

Data Types

- Data types can be used to impose constraints on acceptable expressions
  - Expressions that do not type-check are invalid!
- To do this, we need (at least):
  - A set of *primitive* (pre-defined) types
  - Some way to create new types
  - Some rules to perform type-checking
- Informally speaking, a type system

Issues with Types

- Some type systems risk to compromise the Turing-completeness of the language
  - Think about typed lambda calculus...
- In particular, it important to have appropriate rules for defining new types
  - Again: “function types” are probably not enough
  - Expressions resulting in infinite recursion do not type check!
- We previously said we need “recursive types”, but...
  - What is a *recursive type*?
  - What is it useful for?
  - How can we use it?

More on Data Types

- Every programming language has a set of *primitive types*
  - And many languages allow to define new types
- Simple way to define new types: apply sum or product operations to existing types
  - Product $\mathcal{T}_1 \times \mathcal{T}_2$: type with possible values given by couples of values from $\mathcal{T}_1$ and $\mathcal{T}_2$
  - Sum $\mathcal{T}_1 + \mathcal{T}_2$: type with possible values given by values from $\mathcal{T}_1$ or values from $\mathcal{T}_2$
- Sum == disjoint union; Product == cartesian product
- If $|\mathcal{T}|$ is the number of values of type $\mathcal{T}$, then $|\mathcal{T}_1 \times \mathcal{T}_2| = |\mathcal{T}_1| \cdot |\mathcal{T}_2|$ and $|\mathcal{T}_1 + \mathcal{T}_2| = |\mathcal{T}_1| + |\mathcal{T}_2|$

Algebraic Data Types

- A set (the set of the language's data types), a sum operation and a product operation... It's an algebra!
  - Algebra of the data types; types are called Algebraic Data Types!
- Issue: the sum is a disjoint union...
  - Easy to do "float + bool" (type with possible values integers or booleans)...
  - ...But what about "int + int" (or similar)?
  - The types have to be tagged somehow...

Algebraic Data Types and Constructors

- Solution adopted by many programming languages: do not sum types directly, but first apply a *tagging function* to them
  - Constructor: function generating the values of the type to be summed
  - Summing types generated by different constructors, the issue is solved!
- Variant: set of values generated by a constructor
  - Different constructors generate disjoint variants
  - Hence, instead of “int + int” we can use “Left(int) + Right(int)”

Examples

- C unions are a special case of tagged sum
- “test = i(int) + f(float)” is

```
union example {
    int i;
    float f;
};
```

- Of course, algebraic data types are more generic (0-arguments or multi-argument constructors, etc...)
- All constructors with 0 arguments: enum type
- Haskell, ML and others fully support ADT

```
datatype test = i of int | f of real;

data Test = I Int | F Float
```

Example: Option Type

- Type containing a value or nothing
  - Two constructors: “Nothing” (without arguments) and “Just” (with one argument of the desired type)
- Example: integer or nothing $\rightarrow$ Option_int = Nothing + Just(int)
- Idea: instead of using a null pointer...
- ...Use an option type: Pointer_to_int = Nothing + Just(int *)
  - Advantage: only the “Just” variant can be dereferenced...
  - NULL pointer dereferences do not even compile!

Generic Data Types

- The definition of a new type might depend on a “type variable”
  - Parametric type, depending on another type “$\mathtt{T}$”, denoted by a variable
  - Type variables, generally indicated as greek letters
- Example: generic option type
  - Not “integer or nothing”, but “type α or nothing”
  - α: type variable
- In Haskell, something like

```
data Option a = Nothing | Just a
```

- Used for many other things too (lists, Monads, ...)