DATA TYPES AND DATA STRUCTURES

by

David Toman

University of Waterloo

List of Slides

Data Type

- a *set of elements*: an universe

  $\Rightarrow$ objects, values, ...

- a set of *operations* on the elements

  $\Rightarrow$ an *algebra* (in the math sense)

Simple Data Types

Integers:

$\Rightarrow$ universe: set of integers

$\texttt{integer} = \{-32768, \ldots, -1, 0, 1, \ldots, 32767\}$

$\Rightarrow$ operations: integer arithmetic

$0 :\rightarrow \texttt{integer}$ (constants)

$+ : \texttt{integer} \times \texttt{integer} \rightarrow \texttt{integer}$

$- : \texttt{integer} \times \texttt{integer} \rightarrow \texttt{integer}$

... etc.

$\Rightarrow$ exceptions (an operation is not defined)

division by 0

overflow

Other Simple Types:

Reals (floats), Characters, ...

Not quite *integers* $(\mathbf{Z})$ in the math sense.

Implementation

Integers:

$\Rightarrow$ universe: set of bit representations of integers

$\mathtt{integer} = \{-32768, \ldots, -1, 0, 1, \ldots, 32767\}$

$\Rightarrow$ operations: integer arithmetic (in hardware)

$0 : \ \to \mathtt{integer}$ (constants)

$+ : \mathtt{integer} \times \mathtt{integer} \to \mathtt{integer}$

$- : \mathtt{integer} \times \mathtt{integer} \to \mathtt{integer}$

... etc.

$\Rightarrow$ exceptions:

division by 0

overflow

Comes with **performance** guarantees:

$\Rightarrow$ addition is (runs) in “constant time”

and “constant space”

Compound Data Types

IDEA: make more complex types out of simple types.

- Array of elements of type τ

  $\Rightarrow$ an integer-indexed set of elements of type τ

  ```
  array[1..100] of integer;
  ```

- Record

  $\Rightarrow$ a (fixed) set of identifier i-value of type τ_i pairs

  ```
  record
    sin:  integer;
    name: array [1..30] of char;
  end;
  ```

$\Rightarrow$ Compound types can be used "everywhere" simple types can.

Abstract Data Types

IDEA: Extend an existing programming language (e.g., PASCAL) with *new* data types:

- universe: set of “elements”

  structure not known/preset

- operations:

  functions on the universe
  serve as an *interface*

- “expected” behaviour

  laws the operations must obey

- exceptions

  operation is not defined/fails

$\Rightarrow$ can be used in programs similarly to data types built-in in the PL

Example: Arrays as ADTs

- universe: set of arrays (all arrays!) with elements τ
- operations:

$$\mathbf{new}_\tau : \mathbf{integer} \times \mathbf{integer} \to \mathbf{array}_\tau$$
$$\mathbf{get}_\tau : \mathbf{array}_\tau \times \mathbf{integer} \to \tau$$
$$\mathbf{put}_\tau : \mathbf{array}_\tau \times \mathbf{integer} \times \tau \to \mathbf{array}_\tau$$

- "expected" behaviour:

$$\forall a \in \mathbf{array}_\tau \forall x \in \tau . \mathbf{get}_\tau(\mathbf{put}_\tau(a, i, x), i) = x$$

whenever i is "in the range of indices" for a.

- exceptions:

— $\mathbf{get}_\tau(\mathbf{new}_\tau(1, 10), 20)$ (out of bounds)
— $\mathbf{get}_\tau(\mathbf{new}_\tau(1, 10), 2)$ (not initialized)

Example: Records as ADTs

- universe: set of records

$$\mathbf{record}\ \mathbf{id}_1 : \tau_1, \ldots \mathbf{id}_k : \tau_k\ \ \mathbf{end;}$$

- operations:

$$\mathbf{new}_{\tau_1,\ldots,\tau_k} :\ \to \mathbf{record}_{\tau_1,\ldots,\tau_k}$$
$$\mathbf{getid}_i : \mathbf{record}_{\tau_1,\ldots,\tau_k} \to \tau_i$$
$$\mathbf{setid}_i : \mathbf{record}_{\tau_1,\ldots,\tau_k} \times \tau_i \to \mathbf{record}_{\tau_1,\ldots,\tau_k}$$

- “expected” behaviour

$$\forall r \in \mathbf{record}_{\tau_1,\ldots,\tau_k} \forall x \in \tau_i . \mathbf{getid}_i(\mathbf{putid}_i(r, x)) = x$$

- exceptions

  none

Example: Stack as ADTs

- universe: set of stacks of type τ
- operations:

$$\mathbf{new}_\tau : \ \to \mathbf{stack}_\tau$$
$$\mathbf{empty}_\tau : \mathbf{stack}_\tau \to \mathbf{boolean}$$
$$\mathbf{pop}_\tau : \mathbf{stack}_\tau \to \tau \times \mathbf{stack}_\tau$$
$$\mathbf{push}_\tau : \mathbf{stack}_\tau \times \tau \to \mathbf{stack}_\tau$$

- “expected” behaviour

$$\forall s \in \mathbf{stack}_\tau \forall x \in \tau . \mathbf{pop}_\tau(\mathbf{push}_\tau(s, x)) = (x, s)$$
$$\forall s \in \mathbf{stack}_\tau \forall x \in \tau . \mathbf{empty}_\tau(\mathbf{push}_\tau(s, x)) = \mathrm{false}$$
$$\mathbf{empty}_\tau(\mathbf{new}_\tau) = \mathrm{true}$$

- exceptions

  out of space

Implementation: Data Structures

IDEA: use implementations of existing ADTs to implement new ADT

1. a *data structure* that represents abstract elements
2. *algorithms* that implement operations

Information about performance of existing ADTs allows us to derive such information for the *new* ADT

$\Rightarrow$ complexity of algorithms

Example: N-digit arithmetic ADT

- universe: N-digit natural numbers (in decimal)
- operations:

  $\Rightarrow$ constant 0 (**`zero`**)
  $\Rightarrow$ successor $(+1)$ function (**`succ`**)
  $\Rightarrow$ addition (**`add`**), multiplication (**`mult`**)
  $\Rightarrow$ ...
- laws: usual laws for arithmetic
- exceptions: overflow

Example: N-digit arith. (impl.)

- a data structure (for a constant **`N`**):

```
type number =
        array [1..N] of 0..9;
```

- constant 0:

```
function zero(): number;
begin
    var n: number,
        i: integer;
    for i:=1 to N do n[i]=0;
    zero := n;
end;
```

Example (cont.)

- successor function:

```
function succ(n: number): number;
begin
    var i:integer;
    i := N;
    while (i>0 and n[i]=9) do
    begin
        n[i] := 0;
        i := i-1;
    end;
    if (i=0) then ERROR;
    n[i] := n[i]+1;
    succ := n
end;
```

- addition (try as an exercise!)

Example (another data structure)

- a data structure (for a constant **`N`**):

```
type number = record
       num: array [1..N] of 0..9;
       valid: integer;
     end;
```

- constant 0:

```
function zero(): number;
begin
    var n: number;
    n.valid := 0;
    zero := n;
end;
```

- successor, addition (try as an exercise!)

Support for ADTs in PLs

- convention

  $\Rightarrow$ FORTRAN, Assembly

- convention/type system

  $\Rightarrow$ PASCAL, C, . . .

- modules

  $\Rightarrow$ Modula-x, . . .

- objects

  $\Rightarrow$ C++, JAVA, . . .

- expressive type systems

  $\Rightarrow$ SML (/NJ), Haskell, . . .

Bottom line: you can **always** use
ADT “techniques” by convention

What's to come?

Standard ADTs (commonly used in programming),
how to implement them (data structures and algorithms),
and how good they are (performance analysis)

$\Rightarrow$ in particular how to *store* things
and how to *search* for them later.

Prerequisites (reviewed in following lectures):

1. iteration vs. recursion
2. pointers (and how to live with them)
3. performance analysis basics

Summary

ADT *abstracts* behaviour of objects modeled in a program by a set of functions. The ADT can be used as if it was built-in the programming language to start with.

Why do we like ADTs:

- breaks problem(s) down to manageable pieces
- divides work (many people can work on the problem)
- allows reusing algorithms of other (usually extremely clever) people
  $\Rightarrow$ efficient implementation
- simplifies modification/debugging/. . .