CS 403: Types and Classes

Stefan D. Bruda

Fall 2021

Data Types

- Algorithms + data structures = programs
- Abstractions of data entities highly desirable
- Program semantics embedded in data types
- Data types enable semantic checking
- Data types can enhance design
- Data types can determine memory layout and allocation
- Issues:
  - Extent to which type information is represented in program
  - How types are constructed
  - ~~How types are checked~~ (done)
  - ~~When types are checked~~ (done)

DATA TYPES

- Algorithms + data structures = programs
- Abstractions of data entities highly desirable
- Program semantics embedded in data types
- Data types enable semantic checking
- Data types can enhance design
- Data types can determine memory layout and allocation
- Issues:
  - Extent to which type information is represented in program
  - How types are constructed
  - ~~How types are checked~~ (done)
  - ~~When types are checked~~ (done)
- Data type = set of values + operations on those values
  - A data type is an algebra
  - Set of values defined by enumeration, subrange, or mathematical construction
  - Set model with membership concept ($\in$)
  - Set model of types means language type constructor equivalents for the set operations $\in$, $\subset$, $\cup$, $\times$

- Determining types
  - Explicit typing $\rightarrow$ type determined at declaration, usually invariant throughout execution (Pascal, C, C++, Java)
  - Implicit typing $\rightarrow$ type determined by usage (Prolog, Lisp)
  - Mixed $\rightarrow$ implicit typing by default, but allows explicit declarations (Miranda, Haskell, ML)

Data Types (cont'd)

- Determining types
  - Explicit typing $\rightarrow$ type determined at declaration, usually invariant throughout execution (Pascal, C, C++, Java)
  - Implicit typing $\rightarrow$ type determined by usage (Prolog, Lisp)
  - Mixed $\rightarrow$ implicit typing by default, but allows explicit declarations (Miranda, Haskell, ML)
- Simple types
  - Pre-defined (int, bool, etc.)
  - Enumeration

    ```
    enum colour {red, green, blue};
    data Colour = Red | Green | Blue
    ```

  - Ordinal (discrete order on elements i.e., not real)

Data Types (cont'd)

- Determining types
  - Explicit typing $\rightarrow$ type determined at declaration, usually invariant throughout execution (Pascal, C, C++, Java)
  - Implicit typing $\rightarrow$ type determined by usage (Prolog, Lisp)
  - Mixed $\rightarrow$ implicit typing by default, but allows explicit declarations (Miranda, Haskell, ML)
- Simple types
  - Pre-defined (int, bool, etc.)
  - Enumeration

    ```
    enum colour {red, green, blue};
    data Colour = Red | Green | Blue
    ```

  - Ordinal (discrete order on elements i.e., not real)
- Type constructors: cartesian product
  - $U \times V = \{(u, v) : u \in U, v \in V\}$; $p_1 : U \times V \rightarrow U$; $p_2 : U \times V \rightarrow V$
  - Record implementation `struct icr {int i; char c; double r;};`
  - Tuple implementation `type Icr = (Int, Char, Double)`

Data Types (cont'd)

- Type constructors: union
  - Undiscriminated unions $\rightarrow$ uses names to distinguish components

    `union intOrReal {int i; double r};`
  - Direct algebraic definition `data Either a b = Left a | Right b`

DATA TYPES (CONT'D)

- Type constructors: union
  - Undiscriminated unions $\rightarrow$ uses names to distinguish components

    `union intOrReal {int i; double r};`
  - Direct algebraic definition `data Either a b = Left a | Right b`
- Type constructors: function types $U \rightarrow V$
  - Array types $\rightarrow$ `A[i]` is a mapping from `int` to the type of the array
  - Issues regarding the type of the index:
    - Restrict to int? (e.g., C)
    - Restrict to subrange? (e.g., Pascal)
    - Allow any ordinal type? (e.g., Perl)
    - Is the index part of the type?
  - Function types

```
typedef int (*fun) (int, int)
Int -> Int -> Int
```

Data Types (cont'd)

- Type constructors: union
  - Undiscriminated unions $\rightarrow$ uses names to distinguish components
    ```
    union intOrReal {int i; double r};
    ```
  - Direct algebraic definition `data Either a b = Left a | Right b`
- Type constructors: function types $U \rightarrow V$
  - Array types $\rightarrow$ `A[i]` is a mapping from `int` to the type of the array
  - Issues regarding the type of the index:
    - Restrict to int? (e.g., C)
    - Restrict to subrange? (e.g., Pascal)
    - Allow any ordinal type? (e.g., Perl)
    - Is the index part of the type?
  - Function types
    ```
    typedef int (*fun) (int, int)
    Int -> Int -> Int
    ```
- Type constructors: recursive types
  - Least fixed point: $\emptyset \cup \texttt{int} \cup \texttt{int} \times \texttt{int} \cup \texttt{int} \times \texttt{int} \times \texttt{int} \cup \cdots$
  - `struct intList {int key; intList next};` not legal in C++ (infinite recursion with no base case)
    - Faked in C++: `struct intList {int key; intList *next};`
  - `data IntList = Nil | Cons Int IntList`

Explicit Polymorphism

- Advantages of implicit typing:
  - Smaller code
  - Can use explicit types if desired/needed
  - Construction of a most general type $\rightarrow$ polymorphism
- Polymorphic data structures require explicit polymorphism
- Example: To avoid separate definitions for `IntStack`, `CharStack`, `StringStack`, etc., introduce a mechanism to parameterise the data type declaration
  - C++:

```
template <typename T> struct StackNode {
    T data;
    StackNode<T> *next;
};
template <typename T> struct Stack {
    StackNode<T> *theStack;
};
```

  - Haskell:

```
data Stack a = EmptyStack | StackNode a (Stack a)
```

ALGEBRAIC DATA TYPE SPECIFICATION

- Why data abstraction?
  - Easier to think about $\rightarrow$ hide what doesn't matter
  - Protection $\rightarrow$ prevent access to things you shouldn't see
  - Plug compatibility
    - Replacement of pieces often without recompilation, definitely without rewriting libraries
    - Division of labor in software projects
- An abstract data type specifies the allowed operations and consistency constraints for some type without specifying an actual implementation
- An abstract data type can be specified as an algebra of operations on entities
- An algebraic specification of a data type consists of signatures of available operations, and axioms defining behaviour
  - Algebraic types (implicitly) parameterized
  - Operations divided into constructors and inspectors
    - Axioms contain rules for each combination of constructor and inspector
  - Existence of error operations (or restrictions)

ABSTRACT DATA TYPES AND HASKELL

ADT *Stack*	**Haskell implementation**

type $Stack(Elm)$

Operators:

$\underline{empty} : \emptyset \to Stack$

$\underline{push} : Elm \times Stack \to Stack$

$pop : Stack \to Stack$

$top : Stack \to Elm$

$isEmpty : Stack \to Bool$

Axioms:

$pop(push(e, S)) = S$

$top(push(e, S)) = e$

$isEmpty(empty) = true$

$isEmpty(push(e, S)) = false$

Restrictions:

$pop(empty)$

$top(empty)$

□

```haskell
-- Stack.hs
module Stack (Stack, empty, push, pop,
              top, isEmpty) where

data Stack a = Empty | Push a (Stack a)
               deriving Show

empty :: Stack a
push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
isEmpty :: Stack a -> Bool

empty = Empty
push a s = Push a s
pop (Push e s) = s
pop (Empty) = error "pop (empty)."
top (Push e s) = e
top (Empty) = error "top (empty)"
isEmpty (Push e s) = False
isEmpty Empty = True
```

Abstract Data Types and Haskell (cont'd)

```haskell
-- main.hs
module Main where
import Stack

aStack = push 0 (push 1 (push 2 empty))
```

```
Main> aStack
Push 0 (Push 1 (Push 2 Empty))
Main> :type aStack
aStack :: Stack Integer
Main> isEmpty aStack
False
Main> pop aStack
Push 1 (Push 2 Empty)
Main> let aStack = push 0 empty in pop (pop aStack)

Program error: pop (empty)

Main>
```

Simplified Stack in Haskell

```
-- Stack.hs
module Stack where
data Stack a = Empty |
               Push a (Stack a)
           deriving Show

pop :: Stack a -> Stack a
top :: Stack a -> a
isEmpty :: Stack a -> Bool

pop (Push e s) = s
pop (Empty) = error "pop (empty)."
top (Push e s) = e
top (Empty) = error "top (empty)"
isEmpty (Push e s) = False
isEmpty Empty = True

-- main.hs
module Main where
import Stack
```

```
Main> aStack
Push 0 (Push 1 (Push 2 Empty))
Main> :r
Main> aStack
Push 0 (Push 1 (Push 2 Empty))
Main> :type aStack
aStack :: Stack Integer
Main> isEmpty aStack
False
Main> pop aStack
Push 1 (Push 2 Empty)
Main> let aStack = Push 0 Empty
         in pop (pop aStack)

Program error: pop (empty).
```

More Sample ADTs: Queues

type *Queue*(*Elm*)

Operators:

$\underline{create} : \emptyset \to Queue$

$\underline{enqueue} : Elm \times Queue \to Queue$

$dequeue : Queue \to Queue$

$front : Queue \to Elm$

$isEmpty : Queue \to Bool$

Axioms:

$front(enqueue(e, Q)) = \text{if } (isEmpty(Q)) \text{ then } e \text{ else } front(Q)$

$dequeue(enqueue(e, Q)) = \text{if } (isEmpty(Q)) \text{ then } Q$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } enqueue(e, dequeue(Q))$

$isEmpty(create) = true$

$isEmpty(enqueue(e, Q)) = false$

Restrictions:

$dequeue(empty)$

$front(empty)$

□

More Sample ADTs: Complex Numbers

type *Complex*

Operators:

$\underline{create} : Real \times Real \to Complex$
$real : Complex \to Real$
$imaginary : Complex \to Real$
$+ : Complex \times Complex \to Complex$
$\times : Complex \times Complex \to Complex$
...

Axioms:

$real(create(r, i)) = r$
$imaginary(create(r, i)) = i$
$real(x + y) = real(x) + real(y)$
$imaginary(x + y) = imaginary(x) + imaginary(y)$
$real(x \times y) = real(x) \times real(y) - imaginary(x) \times imaginary(y)$
$imaginary(x \times y) = imaginary(x) \times real(y) + real(x) \times imaginary(y)$
...

$\square$

Issues with Abstract Data Types

1. Problem: Early binding of ADT interface to implementation
    - Can have only one implementation of any given ADT
    - Solution: dynamic binding
        - `stack.pop()` selects the right function at run time depending on the specific implementation being used
        - caller needs not know what implementation is used

Issues with Abstract Data Types

1. Problem: Early binding of ADT interface to implementation
   - Can have only one implementation of any given ADT
   - Solution: dynamic binding
     - `stack.pop()` selects the right function at run time depending on the specific implementation being used
     - caller needs not know what implementation is used
2. Problem: Related ADTs are different ADTs
   - Example: `draw()`, `moveTo(x,y)` applicable to several graphical object ADTs (`Square`, `Circle`, `Line`, etc.)
     - Cumbersome code for drawing a list of objects of different types

```
case obj.tag is
    when t_square => Square.draw(obj.s);
    when t_circle => Circle.draw(obj.s);
    when t_line   => Line.draw(obj.s);
end case;
```

     - Unmaintainable code (what if we later add the `Rectangle` ADT?)
   - Solution: dynamic binding and subtyping
     - automatically select the right draw function for object depending on its type

Issues with Abstract Data Types (cont'd)

3. Problem: Why reimplement every ADT from scratch
   - ADT are often related, with substantial overlapping in implementation
   - Solution: inheritance
     - Define an ADT as a variant of another ADT and specify only the differences: "FilledSquare is like Square, except . . . "
     - Differential programming

Issues with Abstract Data Types (cont'd)

3. Problem: Why reimplement every ADT from scratch
   - ADT are often related, with substantial overlapping in implementation
   - Solution: inheritance
     - Define an ADT as a variant of another ADT and specify only the differences: "FilledSquare is like Square, except . . . "
     - Differential programming

- ADT + Dynamic binding + inheritance = object-oriented programming (OOP)
  - OOP can be added to an existing language (C++ added over C, CLOS added over Lisp)
  - Languages can support OOP but have the same structure and appearance of imperative languages (Eiffel, Java)
  - OOP can be integral part of the language yet a subset of the complete language (type classes in Haskell)
  - Pure OOP languages (Smalltalk)

OO CONCEPTS

- A class is an ADT defining
  - Format of object (instance variables = ”fields”)
  - Operations on objects (methods = functions)
  - Operations for creating new objects
- Objects are instances of a class
- A class may inherit all operations of another class, needs to override only those that it wants to change
  - In the simplest case, a class inherits all of the entities of its parents
- A class that inherits is a derived class or subclass
- The class from which another class inherits is the parent class or superclass
- Subprograms that define operations on objects are called methods or member functions
- The entire collection of methods of an object is called its message protocol or message interface
- Messages have two parts - a method name and the destination object (often called receiver)
- Dynamic dispatch: call specifies operation name (not implementation), system selects appropriate implementation (function) at runtime

OO Example

```
class GraphicalObject {
    private int x, y;
    public void draw() { ... }
    public void moveTo(int x, int y) { ... }
}
class line extends GraphicalObject {
    public void draw() { ... }
}
```

```
GraphicalObject obj;
Line line;
... // initialization, etc.

obj.draw();                 // dynamic dispatch
line.draw();                // dynamic dispatch
line.moveTo(0, 0);          // inheritance
obj = line;                 // ok
line = obj;                 // not ok -- why?
```

OO Polymorphism

- OO implements subtype polymorphism
  - Static type of `obj`: `GraphicalObject`
  - Dynamic type of `obj`: the static type of any subtype (actual type)
- A polymorphic variable references objects of the class as declared as well as objects of any of its descendants
  - When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method must be dynamic
  - This polymorphism simplifies the addition of new methods
- More polymorphic mechanisms:
  - A virtual or abstract method does not include a definition (only defines a protocol)
  - A virtual or abstract class is one that includes at least one virtual method
    - A virtual or abstract class cannot be instantiated
    - Extreme virtual classes: interfaces (Java), pure virtual classes (C++)

DESIGN ISSUES FOR OOPLS

- Single and multiple inheritance
- Allocation and deallocation of objects
- Dynamic and static binding
- Multiple levels of hiding
- The exclusivity of objects
- Are subclasses subtypes?
- Implementation and interface inheritance

MULTIPLE INHERITANCE

Multiple inheritance inherits from more than one class
Creates problems

- Conflicting definitions (e.g., two or more superclasses define a print method)
- Repeated inheritance: same class inherited more than once
  - If A is multiply inherited, should A's instance variables be repeated or not?
    - It depends: not allowed, can choose per field (Eiffel), can choose per class (C++)
  - Usefulness of multiple inheritance is not universally accepted
  - Smalltalk, Beta: single inheritance only
  - Java: single inheritance of classes but multiple inheritance of interfaces
  - C++, Eiffel: multiple inheritance

- Where are objects allocated?
  - Normal allocation for the language (stack + heap) or heap only
  - If they all live on the heap then references to them are uniform
- Is deallocation explicit or implicit?
  - Implicit: overhead (about 10%) and nondeterminism in running time
  - Explicit: no overhead, deterministic
    - Errors (as much as 40% of debugging time)
    - Whose responsibility to deallocate?

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?
- If dynamic binding is not allowed then one loses the advantages of dynamic binding
- If all are then the code is inefficient
  - C++ default is static but can use virtual functions to get dynamic binding
  - Java and Eiffel default is dynamic binding (except for final methods in Java)

Exclussivity of Objects

- Everything is an object
  - Advantage: elegance, purity, least surprise (everything works in the same way)
  - Disadvantage: potential slow operations on simple objects (e.g., float)
    - However all modern languages where everything is an object have implementation tricks that minimize the overhead
  - Notable examples: C++, Haskell
- Include an imperative-style typing system for primitive types, but make everything else objects
  - Advantage: fast operations on simple objects, relatively small typing system
  - Disadvantage: confusion caused by the existence of two separate type systems
  - Can also have class wrappers for primitive types (such as Integer, Float, Character, etc. in Java)
  - Single mainstream example: Java

Subclasses versus Subtypes

- Does an is-a relationship hold between a parent class object and an object of the subclass?
  - If so, then a variable of the derived type could appear anywhere that a variable of the parent type is requested
- Characteristics of a subclass that guarantees it is a subtype:
  - Can only add variables and methods, or redefine existing methods in a compatible way
- If the subclass hides some of the parent's variables and functions or modifies them in an incompatible way, then it does not create a subtype
- So what is a compatible way?
  - By imposing some restrictions on the use of inheritance, the language can ensure substitutability
  - Type extension: If the subclass is allowed to only extend the parent class, then substitutability is guaranteed (we have subtypes)
    - The subclass does not modify and does not hide any method
  - Overriding Methods: The redefined method(s)
    - must have the same number of parameters
    - must not enforce any more requirements on the input parameters
    - may require stronger requirements on the result

Overloaded Methods in Java

$$\begin{tikzcd} & \texttt{Cake} \arrow[r] & \texttt{ChockolateCake} \\ \texttt{Dessert} \arrow[ur] \arrow[dr] & & \\ & \texttt{Scone} \arrow[r] & \texttt{ButteredScone} \end{tikzcd}$$

- Definitions
  1. `Feast(Dessert d, Scone s) { ... }`
  2. `Feast(Cake c, Dessert d) { ... }`
  3. `Feast(ChockolateCake cc, Scone s) { ... }`
- Usage
  - `Feast(dessertref, sconeref)`
  - `Feast(chockolatecakeref, dessertref)`
  - `Feast(chockolatecakeref, butteredsconeref)`

Overloaded Methods in Java

$$
\begin{tikzcd}
 & \texttt{Cake} \arrow[r] & \texttt{ChockolateCake} \\
\texttt{Dessert} \arrow[ur] \arrow[dr] & & \\
 & \texttt{Scone} \arrow[r] & \texttt{ButteredScone}
\end{tikzcd}
$$

- Definitions
  1. `Feast(Dessert d, Scone s) { ... }`
  2. `Feast(Cake c, Dessert d) { ... }`
  3. `Feast(ChockolateCake cc, Scone s) { ... }`
- Usage
  - `Feast(dessertref, sconeref)` → uses definition 1
  - `Feast(chockolatecakeref, dessertref)` → uses definition 2
  - `Feast(chockolatecakeref, butteredsconeref)` → uses definition 3

CONTRAVARIANCE AND COVARIANCE

- Contravariance for input parameters: the input parameters of the subclass method must be supertypes of the corresponding parameters of the overridden method
- Covariance on the result: the result of the redefined method must be a subtype of the result of the overridden method
- Very few languages enforce both rules
  - C++, Java, Object Pascal and Modula-3 require exact identity of the two methods
  - Eiffel and Ada require covariance of both the input and result parameters

Subclass vs. Subtype

- Subclass = inheritance
  - Code reuse
  - Relationship between implementations
- Subtype = polymorphic code
  - Organization of related concepts
  - Relationship between types/interfaces
- Two different concepts but many languages unify them
  - Notable exception: Java
  - Class hierarchy (single inheritance, class B extends class A)
  - Interface hierarchy (multiple subtyping)
- Abstract (deferred) classes = classes with incomplete implementation
  - Cannot be instantiated
  - Provide partial implementation with "holes" that are filled in by subclasses
  - Pure virtual functions in C++
- Polymorphism may require dynamic type checking
  - Dynamic type checking is costly and delays error detection
  - But, if overriding methods are restricted to having the same type then the type checking can be static (provided there is no change in visibility in subclasses)
    - Example: C++ public inheritance

Java Interfaces

```
class Printable {
    void print() { /* default implementation */ } }
class Person extends Printable {
    void print() { /* overriding method */ } }

interface Printable { void print() }
class Person implements Printable {
    void print() { /* body of method */ }}
```

- With Printable as a class:
  - Actual parameter must be a subclass
  - Must inherit the implementation of Printable
  - Specify both interface and implementation
- With Printable as an interface:
  - Actual parameter must be an instance of a class that implements the Printable interface
  - Specifies only interface
  - Fewer constraints (more reusable)