The University of Birmingham
School of Computer Science

Handout 2: Introduction to data abstraction

The term abstraction is used in Programming Languages to denote the idea of ignoring the implementation details and providing an abstract view. Functional abstraction allows us to ignore the implementation (or code) of a function and use an abstract view of its input-output behaviour. **Data abstraction** is the parallel concept for data structures. An abstract data structure involves some notion of a type for the data structure and the **operations** for manipulating it. An implementation of the data structures involves a **representation** for the type and the implementations for all its operations. So data abstraction is an inherently more complex mechanism than functional abstraction.

Data abstraction is incorporated in functional programming through the use of **abstract data types**, which are based on using a type for representing the data structure. In imperative programming, on the other hand, of **objects** and **classes** are more popular. An object contains storage variables for representing the data structure as well as operations for manipulating those variables. In a program, however, we don't define objects, but rather "classes", and objects are created during the program execution as instances of classes.

1 Historical perspective

Historically, the concept of data abstraction dates back to Euclid's "Geometry", where he had to deal with primitive concepts like "point" and "straight line". Even though Euclid provided purported "definitions" for these concepts such as

- A point is that which has no parts.
- A straight line is that which has no width.

Euclid did realize that these were just intuitions, not mathematical definitions. Rather, the mathematical import of these "undefined" concepts is provided by the operations that one can perform with them, such as drawing a straight line between two points and other such operations. In our terminology, Euclid's point and straight line were "abstract types", whose effect is obtained through the operations on them. This fact was not fully understood until the 18th century, when non-Euclidean geometries began to be developed.

In the interim, the mathematicians developed a variety of number concepts such as negative numbers, rational numbers, real numbers and complex numbers, all of which could be represented in a variety of equivalent ways. So, the possibility of multiple representations for the same abstract concept was known, eventually leading to the development of modern abstract algebra in the 19th century.

In Programming Theory, the first ideas of data abstraction developed through David Parnas's groundbreaking ideas on "modules" (1964) and the introduction of the "class" concept in Simula 67 (1967). Tony Hoare combined the two ideas to evolve the notion of "abstract data type" (1972), which came to be regarded as the seminal concept of data abstraction. Objects and classes were ignored by programming theorists for about two decades, using abstract data types instead. Eventually, Bjarne Stroustrup reintroduced classes in C++ and popularised them. The programming language Java can be regarded as simplified version of C++.

It turns out that objects and classes are more suitable for imperative programming (because they are aimed at encapsulating storage variables), whereas abstract data types are more suitable for functional programming. We study abstract data types in this handout and consider objects and classes towards the end of the course.

2 Abstract data types

In Haskell, an abstract data type is defined by defining a module that exports a type. For example, here is a module for a data type of points in a 2D plane:

```
module PointADT (Point, point, xcoord, ycoord) where
  data Point = P (Real, Real)

  point :: Real -> Real -> Point
  point x y = P(x,y)

  xcoord :: Point -> Real
  xcoord (P(x,y)) = x

  ycoord :: Point -> Real
  ycoord (P(x,y)) = y
```

This module exports a type called `Point` and three operations `point`, `xcoord`, `ycoord` for creating a point from its coordinates, and extracting the x and y coordinates of a point respectively.

Any program that imports the module, which we call a **client** of the module, has access to the type `Point` and its operations. However, it has no access to the representation of the type, *i.e.*, it does not have access to the constructor `P` or the fact that a point is represented by a pair of the x and y coordinates. It can only use the `Point` type via the exported operations. So, it is important to export enough operations from the module so that the clients can make use of the type without knowing its representation.

The fact that the clients do not have access to the representation means that we can change the representation type in the module without affecting the clients. For example, we can rename the constructor to something else or we can choose to use a completely different representation such as polar coordinates. For illustration, here is the version of the module using a polar coordinate representation:

```
module PointADT (Point, point, xcoord, ycoord) where
  data Point = Polar (Real, Real)

  point x y = Polar (sqrt(x*x + y*y), arctan(y/x))

  xcoord (Polar(r,theta)) = r * (cos theta)

  ycoord (Polar(r,theta)) = r * (sin theta)
```

Points are represented here by the polar coordinates (the distance from origin r, and the angle θ subtended by the line connecting the origin and the point against the x axis). These quantities are related to the x and y coordinates by the usual formulas[1]

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \arctan(y/x) \end{aligned}$$

Conversely, x and y coordinates can be extracted from r and θ using the formulas: $x = r \cos\theta$ and $y = r \sin\theta$.

Note that replacing one PointADT by another will have no effect on the client programs. They continue to operate the same way and produce the same results (modulo some rounding differences).

[1]Technically, the expression `arctan`(y/x) works only when $x > 0$. Note that $x = 0$ gives a divide-by-zero problem. Morever, if a tangent is negative, `arctan` gives an angle in the bottom right quandrant (where $x > 0$ and $y < 0$). We never get angles in the top left and bottom left quandrants.

The IEEE floating point standard defines a two-argument function called `atan2`, which is also available in Haskell. So instead of `arctan`(y/x), we should write `atan2` y x. We ignore this technicality for the ease of presentation.

3 Data type invariants

An abstract data type can ensure that the exported operations maintain some chosen property of the representation type. For example, all points created by the polar version of the `PointADT` have an `r` component that is non-negative and a `theta` component that is between $-\pi$ and π.[2]

$$I(\texttt{Polar}\ (r,\theta)) \equiv (r \geq 0) \wedge (-\pi < \theta \leq \pi)$$

The functions that receive `point`-typed arguments can assume that the points given as arguments will satisfy this property. In effect, all points manipulated by the `PointADT` always satisfies the property. Hence, we call it the *data type invariant* of the ADT.

Example: Sorted List ADT

We illustrate the concept of data type invariants using a more sophisticated example of sorted lists below.

```
module SortedListADT (SortedList, empty, insert, delete, member) where

data SortedList a = SL [a]
-- invariant: for all SL xs, xs is sorted in ascending order

empty :: SortedList a
empty = SL []

insert :: (Ord a) => a -> SortedList a -> SortedList a
insert x (SL l) = SL (insert' x l)
insert' x [] = [x]
insert' x (a:as) = if x <= a then x:a:as
                   else a:(insert' x as)

delete :: (Ord a) => a -> SortedList a -> SortedList a
delete x (SL l) = SL (delete' x l)
delete' x [] = []
delete' x (a:as) = if x < a then a:as
                   else if x == a then as
                        else a:(delete' x as)

member :: (Ord a) => a -> SortedList a -> Bool
member x (SL l) = member' x l
member' x [] = False
member' x (a:as) = if x < a then False
                   else if x == a then True
                        else member' x as
```

`SortedLists` are represented as simply lists, via a constructor `SL`. We are stating the data type invariant that for all such Sorted Lists, the list will be sorted in ascending order. The operations `empty`, `insert` and `delete` that produce `SortedList` results should ensure that they produce only sorted lists. In return, all the operations that received `SortedList` arguments, *e.g.*, `insert`, `delete` and `member` can assume that they will receive only sorted lists. Notice this, for example, in the function `member`. If the value `x` that we are searching for is less than the first element of the list, we can

[2]This is ensured by the `atan2` function. Recall that π is 180 degrees in radians, taken to be in the anti-clockwise direction. $-\pi$ is 180 degrees in the clockwise direction. So both π and $-\pi$ are the same angle. We only need to include one of them in our range.

immediately return `False` without checking the rest of the list. This works correctly *because of* the data type invariant. If the invariant were not true for the input list, then the result `False` would be wrong.

To ensure that the data type invariant is actually an invariant, we have to prove *verification conditions* for the preservation of the invariant for each of the operations:

I`(empty)`
I`(sl)` $\Longrightarrow$ I`(insert x sl)`
I`(sl)` $\Longrightarrow$ I`(delete x sl)`
I`(sl)` $\Longrightarrow$ *true*

Here I stands for the data type invariant, the fact that the underlying list of the `SortedList` is sorted in ascending order. The second and third properties state that `insert` and `delete` preserve the invariant. The first property states that the constant `empty` satisfies the invariant. The last property, corresponding to the operation `member`, is a bit strange. Off-hand, it doesn't seem to say anything at all, because *true* is always true. However, the force of the statement is to say that *true* holds for the value of `member x sl`, which is to say that `member x sl` evaluates to a value without giving any errors.

The invariant preseration conditions above are derived directly from the types of the operations. Wherever there is a type of the form `SortedList a` in the type of an operation, we have the invariant I in the formula. Whenever there is a constant type (i.e., a type not involving `SortedLists`), we have no condition (i.e., we have *true*).

1. Exercise. Suppose we extend the point ADT's with the following operations:

```
translate: Point → Real → Real → Point
rotate : Point → Real → Point
```

Evidently, `translate` p dx dy translates a point p by dx and dy along the x and y-axes, and `rotate` p *theta* rotates a point p by angle *theta*. Write the invariant preservation conditions for the operations in the polar point ADT.

2. Exercise. Verify the invariant preservation formulas for all the operations of the sorted list ADT.

Example: Binary search trees

Data structures such as binary search trees, height-balanced trees (also called AVL trees), hash tables etc. all involve data type invariants similar to the Sorted List ADT above. As an example, the code for a binary search tree ADT is given in the file `BinarySearchTreeADT.hs` accessible from the module web page. Notice that it has operations very similar to that of the Sorted List ADT above. You are encouraged to think about the role played by the data type invariant in ensuring the correctness of the ADT's opeartions.

4 An ADT for queues

As a more sophisticated example of data abstraction, we treat a queue data structure.

First consider the following straightforward implementation of a queue data structure:

```
module SimpleQueueADT (Queue, empty, insert, delete, front) where

  data Queue a = L [a]
  -- the list of elements with the front of the queue at the head of the list

  empty :: Queue a
  empty = L []

  insert :: a -> Queue a -> Queue a
  insert x (L as) = as ++ [x]

  delete :: Queue a -> Queue a
  delete (L (a:as)) = L as

  front :: Queue a -> a
  front (L (a:as)) = a
```

Note that the list of elements is stored in the order of removal. So the front of the queue is at the head of list. Insertion of a new element is done at the end of the list.

This implementation of queues is inefficient because insertion of a new element takes $O(n)$ time. To achieve an $O(1)$-time insertion operation, we can use a representation with two lists:

```
module QueueADT (Queue, empty, insert, delete, front) where

  data Queue a = Q ([a], [a])
  -- the first list is towards the front of the queue
  -- the second list is towards the rear of the queue

  empty :: Queue a
  empty = Q ([], [])

  insert :: a -> Queue a -> Queue a
  insert x (Q (front, rear)) = Q (front, x:rear)

  reform :: Queue a -> Queue a
  reform (Q ([], rear)) = Q (reverse rear, [])
  reform (Q (front, rear)) = Q (front, rear)

  delete :: Queue a -> Queue a
  delete q = let Q (front, rear) = reform q
             in Q (tail front, rear)

  front :: Queue a -> a
  front q = let Q (front, rear) = reform q
            in head front
```

(Note that `reform` is a "private" operation in the module because it is not exported.)

In this representation, we use two lists `front` and `rear`, where

- the `front` list has elements stored *in the order of the deletion*, and
- the `rear` list has elements stored *in the order of insertion*.

"Order of deletion" means the order in which the elements are to be deleted, i.e., the front element of the list is at the head of the queue, the next element is at the second position and so on. "Order of insertion" means the last element inserted is at the front of the list, the one inserted before it is at the second position and so on. For example, a queue with the elements 1, 2, 3, 4, 5 may have various divisions into a `front` and `rear` lists as follows:

- `front` = [1,2], `rear` = [5, 4, 3]
- `front` = [1, 2, 3, 4, 5], `rear` = []
- `front` = [], `rear` = [5, 4, 3, 2, 1]

With this representation, we can do insertions and deletions efficiently. Insertion is done by adding the new element at the head of the `rear` list. Deletion is done by removing the head of the `front` list.

However, note that as we keep deleting front elements, eventually `front` list will become empty. All the inserted elements would be in the `rear` list. When the `front` list becomes empty, we use the operation called `reform`, which reverses the `rear` list and makes it the `front` list. Since the `reverse` operation can be done in $O(n)$ time, this means that the average time per insertion/deletion is $O(1)$, which is a significant speedup.

The two queue ADT's are *observationally equivalent*, i.e., in any client program, whether we use the `SimpleQueueADT` or the `QueueADT`, we obtain exactly the same results. We can convince ourselves of the truth of this statement by executing a few sample calls using each of the ADT's and observing that the results obtained are the same. But, is there a formal way of proving that the two ADT's are equivalent? This is in fact a deep problem. *We will see an answer towards the end of this course*.