Abstract Data Types

The use of abstract data types (**ADT**'s) as a structuring mechanism for programs is well-established practice.

To review, an ADT is a type together with some operators on that type, such that

- the operators have an external behavior specification
- the type has a private internal representation
- the operators have private internal implementations

Clients of the ADT see only the external specification, none of the internal details. This makes it possible for the **implementer** of the ADT to change the internals without affecting the clients.

In effect, the implementer and client agree to an **interface contract** specifying how values of the type may be used. Ideally, this contract should be completely expressible in our programming language so that the compiler can enforce it. Since operator behaviors are hard to specify, the best we get in practice is an approximation of the contract: a **signature** specifying the **types** of each operator.

Example: Sets

Consider an abstract type of **sets** of values. We might give it the following **signature**, using an ML-like notation:

```
type 'a Set
val empty : 'a Set
val insert : 'a Set -> 'a -> 'a Set
val remove : 'a Set -> 'a -> 'a Set
val member: 'a Set -> 'a -> bool
```

By itself, this doesn't tell us how the various functions should behave. We could specify the desired behavior (semi-formally) using some equations such as these:

```
∀ m,n:'a s:'a Set
member empty m = false
member (insert s n) m = (n = m) or member s m
member (remove s n) m = (n <> m) and member s m
```

We could add many more equations that we would hope to be true, but these three turn out to be sufficient to completely characterize the external behavior of the primitives.

(Of course, there are other possible signatures, including fundamentally more powerful ones, e.g., supporting union, intersection, etc.)

ML Implementation

Here's a simple implementation of sets in ML, using (unordered) lists (possibly with duplicates).

```
- type 'a Set = 'a list
type  'a Set = 'a list
- val empty = nil
val empty = [] : 'a list
- fun insert s x = x::s
val insert = fn : 'a list -> 'a -> 'a list
- fun remove s x = filter (fn y => y <> x) s
val remove = fn : ''a list -> ''a -> ''a list
- fun member s x = exists (fn y => y = x) s
val member = fn : ''a list -> ''a -> bool
```

This implementation is not at all abstract.

- `'a set` is just a synonym of `'a list`, allowing arbitrary lists to be treated like sets.
- Client can access and hence depend on internal representation of sets.
- Client can "spoof" the operators into thinking an arbitrary (bogus) list is a set.

(Note appearance of equality type variable `''a`.)

Towards Abstraction

We've already seen a way to distinguish the `'a set` type from `'a list`: make `set` a **datatype**, e.g.,

```
- datatype 'a Set = SET of 'a list
datatype  'a Set
  con SET : 'a list -> 'a Set
- val empty = SET [] : 'a Set
val empty = SET nil
- fun insert (SET s) x = SET(x::s)
val insert = fn : 'a Set -> 'a -> 'a Set
- fun remove (SET s) x = SET(filter (fn y => y <> x) s)
val remove = fn : ''a Set -> ''a -> ''a Set
- fun member (SET s) x = exists (fn y => y = x) s
val member = fn : ''a Set -> ''a -> bool
```

But the client of the type can still see **inside** a `set` by matching it against a pattern with the `SET` constructor. And the client can stil invent bogus sets by applying `SET` to arbitrary lists. So we still don't have proper abstraction

Key idea: since all access to the contents of a datatype depends on the constructors, obtain abstraction by **hiding** the constructors from the client!

Further Towards Abstraction

How might we hide the constructors? As a first approach, let's try using the `local` facility:

```
- local
    datatype 'a Set = SET of 'a list
  in
    val empty = SET [] : 'a Set
    fun insert (SET s) x = SET(x::s)
    fun remove (SET s) x = SET(filter (fn y => y <> x) s)
    fun member (SET s) x = exists (fn y => y = x) s
  end;
val empty = SET [] : 'a ?.Set
val insert = fn : 'a ?.Set -> 'a -> 'a ?.Set
val remove = fn : ''a ?.Set -> ''a -> ''a ?.Set
val member = fn : ''a ?.Set -> ''a -> bool
- SET;
std_in:12.1-12.3
     Error: unbound variable or constructor: SET
- insert empty 2;
val it = SET [2] : int ?.Set
- insert (insert empty 2) 2;
val b = SET [2,2] : int ?.Set
- a = b;
val it = false : bool
```

Abstypes

This comes close: client can no longer access internals of a SET. But there are still some problems:

- The `Set` type now has no proper name at all.
- Yet the top-level display still knows that the underlying representation uses lists.
- And the built-in equality operator is able to compare sets, which is inappropriate if they are really abstract.

It turns out that we need a special language mechanism to achieve the precise effect we want. This is the `abstype` declaration. It's just like a **datatype** except that it comes with a list of operator functions; they can see the datatype definition, but external clients cannot.

Abstype example

```
- abstype 'a Set = SET of 'a list
  with
    val empty = SET [] : 'a Set
    fun insert (SET s) x = SET(x::s)
    fun remove (SET s) x = SET(filter (fn y => y <> x) s)
    fun member (SET s) x = exists (fn y => y = x) s
  end;
type 'a Set
val empty = - : 'a Set
val insert = fn : 'a Set -> 'a -> 'a Set
val remove = fn : ''a Set -> ''a -> ''a Set
val member = fn : ''a Set -> ''a -> bool
- val a = insert empty 2;
val a = - : int Set
- val b = insert (insert empty 2) 2;
val b = - : int Set
- a = b;
std_in:24.1-24.5 Error: operator and
      operand don't agree (equality type required)
  operator domain: ''Z * ''Z
  operand:         int Set * int Set
  in expression:
    = (a,b)
```

Abstract types print as "-" and are **not** equality types. (In what way do they remain slightly non-abstract, though?)

One last objection: we still have no way of separating (in the program text or in time) the specification of the ADT interface from its implementation. We'll see how to do this with the module system soon.

An alternative implementation

We can now change the implementation of sets without any chance of invalidating client code that uses them. (Of course, client code does need to be recompiled.)

For example, we might arrange that the representation lists contain only unique elements. This will lower space requirements for sets into which the same elements are repeatedly inserted, and in general speed up `remove`s at the cost of slowing down `insert`s.

```
- abstype 'a Set = SET of 'a list
  with
    val empty = SET [] : 'a Set
    fun insert (SET s) x =
          let val s' =
                if exists (fn y => y = x) s then
                  s
                else x::s
          in SET s'
          end
    fun remove (SET s) x =
          let fun r (h::t) =
                    if x = h then t else h::(r t)
                | r nil = nil
          in SET(r s)
          end
    fun member (SET s) x = exists (fn y => y = x) s
  end;
```

An excessively alternative implementation

We might now be tempted to improve our `Set` implementation further by representing sets as **sorted trees**. This should improve the asymptotic time behavior of all the primitives. (Why not try sorted lists?)

But we have no $<$ operator with which to compare the order of arbitrary `'a` values.

In fact, our intended implementation only makes sense for sets of values that can be ordered, and depends on the choice of ordering relation. (This points up that our existing list-based implementations only make sense for sets of values on which the built-in equality predicate is valid – and may do the "wrong thing" even on these.)

One solution is to make the order predicate an explicit parameter of the ADT signature (a **new** signature!). It is convenient to specify the parameter (just) when creating a new set, and then carry it as part of each value.

```
  local
    datatype 'a bintree =
        Leaf | Node of 'a bintree * 'a * 'a bintree
  in
    abstype 'a Set = SET of ('a * 'a -> bool)
                          * 'a bintree
    with
      fun empty lt = SET(lt,Leaf)
      fun insert (SET(lt,s)) x =
          let fun f Leaf = Node(Leaf,x,Leaf)
                | f (n as Node(left,y,right)) =
                      if lt(x,y) then
                        Node(f left,y,right)
                      else if lt(y,x) then
                        Node(left,y,f right)
                      else n
          in SET(lt,f s)
          end
      fun remove (SET(lt,s)) x = ...
      fun member (SET(lt,s)) x = ...
    end
  end;
type 'a Set
val empty = fn : ('a * 'a -> bool) -> 'a Set
val insert = fn : 'a Set -> 'a -> 'a Set
val remove = fn : 'a Set -> 'a -> 'a Set
val member = fn : 'a Set -> 'a -> bool

- val a = empty Integer.<;
val a = - : int Set
```

A Function-based Implementation

ML provides no direct support for enforcing the equational specification that we gave originally – merely for making sure that the type signature is respected and that values of the ADT cannot be put together or taken apart by clients.

```
∀ m,n:'a s:'a set
member empty m = false
member (insert s n) m = (n = m) or member s m
member (remove s n) m = (n <> m) and member s m
```

But as a final implementation of (equality-based) `Set`, we'll use the equational spec quite directly. The idea is to **represent** as set by its own membership function!

```
abstype 'a Set = SET of 'a -> bool
with
  val empty = SET(fn n => false)
  fun insert (SET s) n =
              SET(fn m => (n = m) orelse s m)
  fun remove (SET s) n =
                SET(fn m => (n <> m) andalso s m)
  fun member (SET s) x = s x
end;
```

Note that this implementation approach extends well to infinite sets too.