Abstract Datatypes in Standard ML

Handout F

26 June 2007

Sebesta defines Abstract Data Types (ADTs) as data types that satisfy the following properties:

(a) "The declarations of the type and the protocols of the operations on objects of the type [...] are contained in a single syntactic unit."

(b) "The representation of objects of the type is hidden from the program units that use the type."

Standard ML provides ADT support by means of a specialised *module language*. This module language provides five features:

(a) *signatures*[1], which describe a type or several types as well as operations over the type or types.

Signatures capture the idea behind Sebesta's first concept above.

(b) *structures*[2], which collect types and structures in a single syntactic unit.

(c) *signature constraints*, which allow us to hide implementation details from prying eyes: a signature constraint takes a signature and a structure, and hides all parts of the structure except for the parts that are listed in the structure.

(d) *structure/signature renaming and inclusion*, which allows us to save some typing work.

(e) *functors*, which transform structures into other structures. For example, the SML/NJ library uses functors to transform a structure that describes a datatype into a structure that describes a hashmap over the datatype.

Functors are a very powerful feature– less expressive than C++ templates, but still quite expressive. However, since they are specific to Standard ML, we will not discuss them here. Similarly, structure and signature inclusion, while extremely handy for practical programming, are only convenience features that I will leave out.

[1] These are different from *method signatures* and *procedure signatures*. For clarity, I wil call them SML signatures.

[2] Not to be confused with C *structs*, which are records.

1 Structures

SML structures are similar to structures and modules found in other languages (such as Pascal or CLU): Each structure collects definitions of types and values (including functions) in a specific namespace. For example, below is the definition of a simple STACK structure:

```
structure Stack =
struct
    type 'a stack = 'a list

    val empty = []
    val push = op::
    fun pop []          = NONE
      | pop (tos::rest) = SOME tos
end
```

This structure collects the definition of a type constructor stack and of operations definitions on it:

```
    val empty : 'a stack
    val push : 'a * 'a stack -> 'a stack
    val pop : 'a stack -> 'a option
```

We can now use this structure in our SML programming. For example, we can push an element onto the stack:

```
- Stack.push (1, Stack.empty);
```

The above can be a bit inconvenient to type. SML allows us to import structures directly, but we will not use this feature; instead, we will introduce an alias to the structure Stack:

```
- structure S = Stack
```

Now we can write

```
- S.push (1, S.empty);
```

Alas, what SML reports back to us is not quite what we wanted:

```
val it = [1] : int list
```

The type of our result is int list, not int stack! Let us try to see whether we can convince the type inference system to give us the type we wanted:

```
- S.push (1, S.empty) : int S.stack;
val it = [1] : int S.stack
```

Well, we can– but the result is still not satisfying. Our explicit type annotation convinced the type inference system to use a different type, but all this means is that SML considers Stack.stack to be just an alias for list: we can use both interchangeably whereever we want. In fact, as far as SML is concerned, the types of our stack operations are the same as the following:

```
    val empty : 'a list
    val push : 'a * 'a list -> 'a list
    val pop : 'a list -> 'a option
```

In summary, structures, by themselves, are useful for collecting related definitions into a common *namespace*[3], but it does little by itself to help us with abstract datatypes.

2 Signatures

As we discussed above, structures are insufficient to give us abstract datypes. What we need are *abstract* descriptions of the type we want to provide (hence the term *abstract* data type). In SML, these abstract descriptions come about in the shape of *signatures*. Signatures can be given names, though this is not neccessary. Below is a named signature, with the name STACK, which neatly matches the stack datatype we want.

```
signature STACK =
sig
    type 'a stack

    val empty : 'a stack
    val push  : 'a * 'a stack -> 'a stack
    val pop   : 'a stack -> 'a option
end
```

Note that we didn't specify what the 'a stack looks like. This is precisely the point where we abstract over the concrete implementation: We explicitly do not give any implementation details. In fact, we may have many different implementations– our list-based implementation is simple and straightforward, but we could use a different datatype, or use some of SML's imperative features to log stack usage.

Now, let us apply the above signature to our STACK:

```
structure ListStack = Stack :> STACK
```

The above is similar to an aliasing, except that we now introduce a *signature constraint*: We force Stack to match the signature STACK, using the operator :>[4]. As a result, we get a structure ListStack, which we abbreviate as

```
structure LS = ListStack
```

Let's try to push something to this stack:

```
- LS.push (1, LS.empty);
val it = - : int ListStack.stack
```

[3] A namespace is a scope that has a name attached to it, such that we can refer to the scope by name.

[4] This operator should not be confused with the subtype relation, even though the two are related.

SML allowed us to push the value, but now the type we got back is the type we wanted. Let's see whether this type is still compatible to int list:

```
- val l : int list = LS.push (1, LS.empty);
(* ERROR: Types don't match! *)
```

Our new stack type is now distinct from the type of lists. Thus, we have managed to implement an ADT: First, we managed to contain the description of the ADT in a "single syntactic unit", namely our signature STACK. Secondly, we have managed to provide an implementation of the stack that did not reveal its inner workings, in our structure ListStack.

Signature constraints not only restrict types, but also eliminate definitions that were not promised by the signature. For example, consider the alternative (and somewhat silly) implementation of STACK below:

```
structure SlowStack =
struct
    datatype 'a stack = TOS
                      | ELT of 'a * 'a stack

    fun replace_tos v TOS            = v
      | replace_tos v (ELT (e, tl))  = ELT (e, replace_tos v tl)

    val empty = TOS
    fun push (elt, stack) = replace_tos (ELT (elt, TOS))
                                        stack
    fun pop (TOS)           = NONE
      | pop (ELT (v, TOS)) = SOME v
      | pop (ELT (_, tl))   = pop tl
end :> STACK
```

This implementation manages to replace the efficient $O(1)$ push and pop operations from our list-based implementation by $O(n)$ operations. The interesting part, however, is that, after the type constraint, SlowStack looks and behaves exactly like ListStack (apart from being slower, of course).

In particular, the signature constraint did the following:

(a) The signature hid the fact that stack is implemented as a user-defined datatype

(b) The signature hid both of the constructors, TOS and ELT, which would normally be visible in the structure, and

(c) The sinature hid the auxiliary function replace_tos.

3 Summary

- SML signatures (signature SIG = sig ... end) allow us to specify the abstract description of an ADT.
- SML structures (structure Str = struct ... end) allow us to collect type and value definitions in a single namespace.
- The signature constraint operator, :>, allows us to restrict a structure to precisely the information promised by a signature. The result of such a restriction is again a structure.
- SML signature and structure aliases allow us to define shortcuts for signature and structure names. Simultaneously, in combination with the :> operator, they allow us to give names to structures that we have restricted by a signature.
- We need structures, signatures, and :> to build ADTs in Standard ML.