Lecture Notes on Data Abstraction

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 12
October 7, 2004

One of the most important ideas in programming is *data abstraction*. It refers to the property that clients of library code cannot access the internal data structures of the library implementation. The implementation remains *abstract*. Data abstraction is inherently a static property, that is, a property that must be verified before the program is run. This is because during execution the internal data structures of the library are, of course, present and must be manipulated by the running code. Hence, data abstraction is very closely tied to type-checking [Ch. 21].

Modern languages, such as ML and Java, support data abstraction, although the degree to which it is supported (or how easy it is to achieve) varies. Lower-level languages such as C do not support data abstraction because various unsafe constructs can be exploited in order to expose representations. This can have the undesirable effect that authors of widely used library code cannot change their implementations because such a change would break client code. Furthermore, ill-behaved clients can change the representation of a data type, potentially breaking the internal invariants of the library. Even the presence of a well-documented application programmers interface (API) is not much help if it can be easily circumvented due to weaknesses in the programming language.

In ML, abstraction is supported primarily at the level of modules. This can be justified in two ways: first, data abstraction is mostly a question of program interfaces and therefore it arises naturally at the point where we have to consider program composition and modules. Second, the ML core language has been carefully designed so that no type information needs to be supplied by the programmer: full type inference is decidable. In the presence of data abstraction this no longer makes sense since, as we will see, an implementation does not uniquely determine its interface.

So how is data abstraction enforced in ML? Consider the following skeletal signature, presenting a very simple interface to an implementation of queues containing only integers.

```
signature QUEUE =
sig
  type q
  val empty : q
  val enq : int * q -> q
  val deq : q -> q * int (* may raise Empty *)
end;
```

This signature declares a type `q` which is abstract (no implementation of `q` is given). It then presents three operations on elements of this type. An implementation of this interface is a structure that matches the signature. Here is an extremely inefficient one.

```
structure Q :> QUEUE =
struct
  type q = int list
  val empty = nil
  fun enq (x,l) = x::l
  fun deq l = deq' (rev l)
  and deq' (y::k) = (rev k, y)
    | deq' (nil) = raise Empty
end;
```

Note that we use *opaque ascription* `:> QUEUE`, which is Standard ML's way to guarantee data abstraction. No client can see the definition of the type `Q.q`. For example, the last line in the following example fails type-checking.

```
val q21 = Q.enq (2, Q.enq (1, Q.empty));
val (q2, 1) = Q.deq q21;
val _ = hd q21;   (* TYPE ERROR HERE *)
```

This is because `hd` can operate only on lists, while `q21` is only known to have type `Q.q`. The implementation of `Q.q` as `int list` is hidden from the type-checker in order to ensure data abstraction. This means we can replace `Q` with a more efficient implementation by a pair of lists,

```
structure Q :> QUEUE =
struct
  type q = int list * int list
  val empty = (nil, nil)
  fun enq (x, (back, front)) = (x::back, front)
  fun deq (back, x::front) = ((back, front), x)
    | deq (back as _::_, nil) = deq (nil, rev back)
    | deq (nil, nil) = raise Empty
end;
```

and any client code will continue to work (although it may now work much faster).

In order to avoid the complications of a full module system, we introduce *existential types* $\exists t.\tau$, where t is a bound type variable. t represents the abstract type and τ represents the type of the operations on t. Returning to the example, the signature

```
signature QUEUE =
sig
  type q
  val empty : q
  val enq : int * q -> q
  val deq : q -> q * int (* may raise Empty *)
end;
```

is represented by the type

$$\exists q.q \times (\mathtt{int} \times q \to q) \times (q \to q \times \mathtt{int}).$$

Except for the missing names `empty`, `enq`, and `deq`, this carries the same information as the signature.

A value of an existential type is a tuple whose first component is the implementation of the type, and the second component is an implementation of the operations on that type. We write this as $\mathtt{pack}(\sigma, e)$. For the sake of brevity, we show only part of the example:

```
structure Q :> QUEUE =
struct
  type q = int list
  val empty = nil
  ...
end;
```

is represented as

$$\texttt{pack}(\texttt{int list}, \texttt{pair}(\texttt{nil}, \ldots)) : \exists q. q \times \cdots$$

In contrast, the second implementation

```
structure Q :> QUEUE =
struct
  type q = int list * int list
  val empty = (nil, nil)
  fun enq (x, (back, front)) = (x::back, front)
  fun deq (back, x::front) = ((back, front), x)
    | deq (back as _::_, nil) = deq (nil, rev back)
    | deq (nil, nil) = raise Empty
end;
```

looks like

$$\texttt{pack}(\texttt{int list} \times \texttt{int list}, \texttt{pair}(\texttt{pair}(\texttt{nil}, \texttt{nil}), \ldots)) : \exists q. q \times \cdots$$

From these examples we can deduce the typing rules. First, existential types introduce a new bound type variable.

$$\frac{\Gamma, t \ \mathsf{type} \vdash \tau \ \mathsf{type}}{\Gamma \vdash \exists t. \tau \ \mathsf{type}}$$

Second, the package that implements an existential type requires that the operations on the type respect the definition of the type. This is modeled in the rule by substituting the implementation type for the type variable in the body of the existential.

$$\frac{\Gamma \vdash \sigma \ \mathsf{type} \quad \Gamma \vdash e : \{\sigma/t\}\tau}{\Gamma \vdash \texttt{pack}(\sigma, e) : \exists t. \tau}$$

For example, if we take the first implementation above, the first two lines below justify the third.

$$\frac{\begin{array}{l} \cdot \vdash \texttt{int list} \ \mathsf{type} \\ \cdot \vdash \texttt{pair}(\texttt{nil}, \ldots) : \texttt{int list} \times \cdots \end{array}}{\cdot \vdash \texttt{pack}(\texttt{int list}, \texttt{pair}(\texttt{nil}, \ldots)) : \exists q. q \times \cdots}$$

In the second implementation, we need the implementation of q to have type $\texttt{int list} \times \texttt{int list}$.

$$\frac{\cdot \vdash \texttt{int list} \times \texttt{int list}\ \mathsf{type} \qquad \cdot \vdash \texttt{pair}(\texttt{pair}(\texttt{nil}, \texttt{nil}), \ldots) : (\texttt{int list} \times \texttt{int list}) \times \cdots}{\cdot \vdash \texttt{pack}(\texttt{int list} \times \texttt{int list}, \texttt{pair}(\texttt{pair}(\texttt{nil}, \texttt{nil}), \ldots)) : \exists q. q \times \cdots}$$

Next we have to consider how make the implementation of an abstract type available. In ML, a structure is available when a definition `structure S = ...` is made at the top level. Here, we need an explicit construct to open a package to make it available to a client. Given a package $e : \exists t.\tau$, we write $\texttt{open}(e, t.x.e')$ to make e available to the client e'. Here, t is a bound type variable that refers to the abstract type (and remains abstract in e') and x is a bound variable that stands for the implementation of the operations on the type. In our example, $\texttt{fst}(e)$ denotes the implementation of `empty`, $\texttt{fst}(\texttt{snd}(e))$ stands for the implementation of `enq`, etc.

This leads us to the following rule:

$$\frac{\Gamma \vdash e : \exists t.\tau \quad \Gamma, t\ \mathsf{type}, x{:}\tau \vdash e' : \sigma \quad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \texttt{open}(e, t.x.e') : \sigma}$$

We have added the explicit premise that $\Gamma \vdash \sigma\ \mathsf{type}$ to emphasize that t must not occur already in Γ or σ: every time we open a package, or multiple package, we obtain a new type, different from all types already known. This *generativity* means that even multiple instances of the exact same structure are not recognized to have the same implementation type: any one of them could be replaced by another one without affecting the correctness of the client code.

The property of data abstraction can be seen in the rule above: the code e' can use the library code e, but during type-checking only a type variable t is visible, not the implementation type. This means the code in e' is *parametric* in t, which guarantees data abstraction.

The operational semantics is straightforward and does not add any new ideas to those previously discussed. This confirms that the importance of data abstraction lies in compile-time type-checking, not in the runtime properties of the language.

$$\frac{e \mapsto e'}{\texttt{pack}(\tau, e) \mapsto \texttt{pack}(\tau, e')} \qquad \frac{v\ \mathsf{value}}{\texttt{pack}(\tau, v)\ \mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{open}(e_1, t.x.e_2) \mapsto \texttt{open}(e_1', t.x.e_2)}$$

$$\frac{v_1\ \mathsf{value}}{\texttt{open}(\texttt{pack}(\tau, v_1), t.x.e_2) \mapsto \{v_1/x\}\{\tau/t\}e_2}$$

Observe that before the evaluation of the body of an `open` expression, we substitute τ for t, making the abstract type concrete. However, we know that e_2 was type-checked without knowing τ, so this does not violate data abstraction.

The progress and preservation theorems do not introduce any new ideas. For the type substitution we need a type substitution property that was given in Lecture 11 on *Parametric Polymorphism*.

Combining parametric polymorphism and data abstraction, that is, universal and existential types can be interesting and fruitful. For example, assume we would like to allows queues to have elements of arbitrary type s. This would be specified as

$$\forall a.\exists q.q \times (a \times q \to q) \times (q \to q \times a).$$

For example, the implementation of a queue by a single list would then have the form

$$\texttt{Fn}(a.\texttt{pack}(a\,\texttt{list}, \langle \texttt{Inst}(\texttt{nil}, a), \ldots \rangle))$$

Note that the type

$$\exists q.\forall a.q \times (a \times q \to q) \times (q \to q \times a)$$

would be incorrect, because we cannot choose the implementation type for q before we know the type a.

As another example, assume we want to widen the interface to also export double-ended queues `q'` with some additional operations that we leave unspecified here. Then the type would have the form

$$\exists q.\exists q'.q \times q' \times \cdots .$$

The implementation would provide definitions for both q and q', as in

$$\texttt{pack}(\texttt{int}\,\texttt{list}, \texttt{pack}(\texttt{int}\,\texttt{list}, \ldots)).$$

Next we return to the question of type-checking. Consider[1]

$$\texttt{pack}(\texttt{int}, \texttt{pair}(\texttt{fn}(\texttt{int}, x.x + 1), \texttt{fn}(\texttt{int}, x.x - 1))).$$

This package has 16 different types; we show four of them here:

$$\begin{array}{l}
\exists t.(t \to t) \times (t \to t) \\
\exists t.(\texttt{int} \to t) \times (t \to \texttt{int}) \\
\exists t.(t \to \texttt{int}) \times (\texttt{int} \to t) \\
\exists t.(\texttt{int} \to \texttt{int}) \times (\texttt{int} \to \texttt{int})
\end{array}$$

[1] using infix notation for addition and subtraction

While not all of these are meaningful, they are all different and the type-checker has no way of guessing which one the programmer may have meant. This is inherent: an implementation does not determine its interface. However, we can *check* an implementation against an interface, which is precisely what bi-directional type-checking achieves. We have not formally presented the technique in these notes and postpone its discussion for now.