A note on internal interfaces

Dinesh Katiyar

This document discusses the subtleties involved in adding a particular language feature to an existing framework. The target language is an object-oriented language which has a module system fairly similar to that of Standard ML, and a typing discipline that includes subtyping. The feature is what we call "the internal interface", and this is described in a later section.

1 The language

We describe the language by presenting the abstract syntax of the terms and types of the language. The expressions of the language can be described by the grammar below. The language contains specifications which are basically like the Standard ML signatures, and modules which provide implementations for these specifications.

$$
\begin{array}{rcll}
e & ::= & x & \text{(variable)} \\
& \mid & \lambda x{:}T.e & \text{(abstraction)} \\
& \mid & ee' & \text{(function application)} \\
& \mid & (i_{ext}, i_{int}) & \text{(module)} \\
& \mid & e.l & \text{(module constituent selection)} \\
& \mid & \Lambda t <: T.e & \text{(polymorphic term)} \\
& \mid & eT & \text{(type application)} \\
i & ::= & \langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle & \text{(submodule)}
\end{array}
$$

The type system has the following types.

$$
\begin{array}{rcll}
T & ::= & a & \text{(basic types)} \\
& \mid & T \rightarrow T' & \text{(functions)} \\
& \mid & (I_{ext}, I_{int}) & \text{(specifications)} \\
& \mid & \forall t <: T.T' & \text{(polymorphism)} \\
I & ::= & \langle \ell_1{:}T_1, \ldots, \ell_n{:}T_n \rangle &
\end{array}
$$

2 The feature

Probably the most unusual aspect of the language above is the fact that specifications have two components instead of one. We shall refer to these as the *external interface* and the *internal interface*. The latter is the feature that we shall inspect closely. This complexity naturally reflects in the structure of modules as well. The external interface describes the components of the specification which are visible to the external world. This corresponds to the familiar notion of methods of a class in an object-oriented world. The internal interface describes components that are visible to other instances of that specification, and instances of its subtypes. This concept is similar to that of *friends* in C^{++}.

As one would expect, a subtype of a specification can refine existing components and can have extra components in either interface. Less obviously, a subtype can also move components from the internal interface to the external. This makes sense because the subtype provides all the components of the supertype, except that it is less secretive about which of those can be seen by the external world.
The subtyping rule is

$$\frac{A_i <: A'_i \qquad B_j <: B'_j \qquad C_k <: C'_k}{(\,\langle a_i\colon A_i, b_j\colon B_j, e_n\colon E_n\rangle, \langle c_k\colon C_k, d_m\colon D_m\rangle\,) <: (\,\langle a_i\colon A'_i\rangle, \langle b_j\colon B'_j, c_k\colon C'_k\rangle\,)}$$

A very immediate use of this rule is that it allows one to provide a single implementation for different "views" of the same structure. For instance, given some basic structure with components a, b and c, one might want a type which allows only a and b to be visible to the external world, and another which allows only b and c to be visible. Such types would be very useful for providing restricted access to different users in many database applications. This could be easily achieved by declaring the following types :

$$V_1 = (\,\langle a\colon A, b\colon B\rangle, \langle c\colon C\rangle\,)$$
$$V_2 = (\,\langle b\colon B, c\colon C\rangle, \langle a\colon A\rangle\,)$$
$$V = (\,\langle a\colon A, b\colon B, c\colon C\rangle, \langle\rangle\,)$$

Now V is a subtype of both V_1 and V_2, and so instances of V can be used as instances of either V_1 or V_2, and would accordingly provide different extents of visibility.

3 The pros

It is often very convenient to be able to impose certain constraints on all implementations of a specification. This becomes fairly apparent in the implementation of operations that need an argument of the same type (binary operations, essentially). The object-oriented style of operations, where we have $x_1.bin\_op(x_2)$ instead of $bin\_op(x_1, x_2)$, imposes an inherent asymmetry on the two arguments of the binary operation. Only the internal representation of x_1 is available for the implementation of bin_op and no assumptions can be made about the representation of x_2. The internal interface is a mechanism that allows the implementation of a specification to make available some more information about itself to instances of that specification and its subtypes. This information is exactly the set of components that are in the internal interface. This also makes it easier for different implementations of the same specification to interact, since the internal interface provides a sort of peephole into the internal representation. The following examples motivate this concept further.

3.1 Time-stamped events

Consider time-stamped events. A specification for these might look as follows:
Specification `time_stamped_event` is

$$(\,\langle \mathit{event\_name}\colon \texttt{string}, \mathit{precedes}\colon \texttt{time\_stamped\_event} \rightarrow \texttt{bool}, \ldots\rangle, \langle\rangle\,)$$

Our implicit intention is that each structure of type `time_stamped_event` will have a local real variable which stores the time of the event. Since the only time-related operation on time-stamped events is `precedes`, this variable is not part of the external interface. However, if `ev1,ev2:time_stamped_event` are two events, then `ev1.precedes(ev2)` should compare the times of the two events, returning a boolean value. This requires `ev1` to have access to the time of `ev2`. If the function `precedes` must be implemented using only public operations on its argument, then the only solution is to make the time of an event public. This seems undesirable, since only other time-stamped events need to see the time of an event.
The internal interface solves this problem appropriately by letting us specify that every time-stamped event must have a `time` component. By putting this requirement in the internal interface, we make the `time` component of one event accessible to the function components of other events, without making `time` generally available. With an internal interface added, the specification will now appear as follows:

$$(\,\langle \mathit{event\_name}\colon \texttt{string}, \mathit{precedes}\colon \texttt{time\_stamped\_event} \rightarrow \texttt{bool}, \ldots\rangle, \langle \mathit{time}\colon \texttt{real}\rangle\,)$$

3.2 Sets with union

Suppose we wish to specify and implement a class of set objects with a union operation. In object-oriented style, the union of two sets is computed by sending one set as an argument to another.

$$
\begin{aligned}
\texttt{SET} = \lambda t.(\ \langle\ & \textit{empty} : \texttt{bool}, \\
& \textit{element} : \texttt{t} \to \texttt{bool}, \\
& \textit{insert} : \texttt{t} \to \texttt{SET(t)}, \\
& \textit{delete} : \texttt{t} \to \texttt{SET(t)}, \\
& \textit{union} : \texttt{SET(t)} \to \texttt{SET(t)}, \\
& \textit{intersect} : \texttt{SET(t)} \to \texttt{SET(t)}\ \rangle\ , \\
\langle\ \rangle\ )&
\end{aligned}
$$

If we try to implement the above specification, we run into difficulties with *union*(s:`SET(t)`). The problem is that the natural, efficient implementations of *union* need to iterate over the list of the elements in the argument `s`, but there is no apparent way to do this using only the public operations allowed on `s`. One not quite satisfactory approach is to implement sets in a *lazy* style. While this makes *union* easy to compute, it becomes difficult to implement *empty* properly in the presence of intersection. A more realistic approach is to extend the external interface with a function *element_list* which returns a list of all the elements in the set. This has the undesirable effect of making more operations public than required. A more subtle reason for keeping *element_list* out of the external interface is that we expect equality to be a congruence relation, but we do not require equal sets to have the same ordered element list.

We may solve this problem using internal interfaces. An internal interface requiring an *element_list* function forces each set to define a function *element_list*. However, this function is not publicly visible – it may only be used by other set objects. So our specification would be as follows.

$$
\begin{aligned}
\texttt{SET} = \lambda t.(\ \langle\ & \textit{empty} : \texttt{bool}, \\
& \textit{element} : \texttt{t} \to \texttt{bool}, \\
& \textit{insert} : \texttt{t} \to \texttt{SET(t)}, \\
& \textit{delete} : \texttt{t} \to \texttt{SET(t)}, \\
& \textit{union} : \texttt{SET(t)} \to \texttt{SET(t)}, \\
& \textit{intersect} : \texttt{SET(t)} \to \texttt{SET(t)}\ \rangle\ , \\
\langle\ & \textit{element\_list} : \texttt{List(t)}\ \rangle\ )
\end{aligned}
$$

4 The cons

The internal interface seems to provide a clean and reasonable mechanism for revealing as many details of the internal representation as may be useful for the implementation of the operations. However, a closer study reveals that everything isn't quite as nice as it seems. The structural approach to subtyping that we adopt leads to some unexpected behaviour with respect to internal interfaces.

Consider the following example.

$$S_1 = (\,\langle e_1 \!:\! T_1\rangle, \langle i \!:\! T_i\rangle\,)$$
$$S_2 = (\,\langle e_2 \!:\! T_2\rangle, \langle i \!:\! T_i\rangle\,)$$

Now some module M_1 implementing S_1 can actually access the i component of any instance of S_2. This happens because both S_1 and S_2 are subtypes of the specification S.

$$S = (\,\langle\rangle, \langle i \!:\! T_i\rangle\,)$$

This implies that any instance of S_2 is also an instance of S and therefore, its internal interface is visible to all instances of the subtypes of S, M_1 being one such instance.

So even though S_1 and S_2 can be radically different specifications, every common component in their internal interface is visible to instances of each other. This gets even more surprising when one uses the fact that a subtype can move components to the external interface. Consider the following.

$$S_3 = (\,\langle e_3 \!:\! T_3, i \!:\! T_i\rangle, \langle\rangle\,)$$

Now a module M_3 implementing S_3 can access the internal i component of any instance of S_2 just because it happens to have an i component itself. While this may be put off as one of the accidents that one would expect from structural subtyping, it does seem to be undesirable.
Another problem with using the internal interface is related to type inference. There is a certain degree of circularity here. To infer the type of an interface, one needs to infer the types of the components. But the components can involve the selection of components from the internal interface of some other module, and the well-formedness of this depends on the type of the interface, which is what we were inferring in the first place.

A Typing rules without the internal interface

The rules are fairly simple in this case. The subtyping rule allows for specialisation of existing components and addition of extra components. The typing rules for the specification and for component selection use the types of the components in a straightforward way.

$$(\mathit{spec\_subtype}) \qquad \frac{\vdash A_1 <: A_1' \qquad \ldots \qquad \vdash A_n <: A_n'}{\vdash (a_1 : A_1, \ldots, a_n : A_n, \ldots) <: (a_1 : A_1', \ldots, a_n : A_n')}$$

$$(\mathit{spec\_type}) \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \ldots \qquad \Gamma \vdash e_n : A_n}{\Gamma \vdash (a_1 = e_1, \ldots, a_n = e_n) : (a_1 : A_1, \ldots, a_n : A_n)}$$

$$(\mathit{comp\_selection}) \qquad \frac{\Gamma \vdash e_i : A_i}{\Gamma \vdash (a_1 = e_1, \ldots, a_n = e_n).a_i : A_i}$$

B Typing rules with the internal interface

The rules here are more involved. The subtyping rule allows for the "externalization" of internal components. One has to use the type of a specification in typing its components. So the environment now is a pair (Γ, T_I) where Γ is the usual environment consisting of assertions of the form $x{:}T$ and $t <: T$, and T_I is the type of the smallest enclosing module. And the selection of an internal component has to make sure that it is allowed, by looking at the types of the "caller" and the "callee".
For brevity of the rules, we use the abbreviations

$$\begin{array}{rcl} X & \equiv & (\,\langle l_1 : E_1, \ldots l_j : E_j \rangle, \langle m_1 : I_1, \ldots m_k : I_k \rangle\,) \\ x & \equiv & (\,\langle l_1 = e_1, \ldots l_j = e_j \rangle, \langle m_1 = i_1, \ldots m_k = i_k \rangle\,) \end{array}$$

$$(\mathit{spec\_subtype}) \qquad \frac{\vdash A_i <: A_i' \qquad \vdash B_j <: B_j' \qquad \vdash C_k <: C_k'}{\vdash (\,\langle a_i : A_i, b_j : B_j, e_n : E_n \rangle, \langle c_k : C_k, d_m : D_m \rangle\,) <: (\,\langle a_i : A_i' \rangle, \langle b_j : B_j', c_k : C_k' \rangle\,)}$$

$$(\mathit{spec\_type}) \qquad \frac{\begin{array}{c}(\Gamma, X) \vdash e_1 : E_1 \ \ldots\ (\Gamma, X) \vdash e_j : E_j \\ (\Gamma, X) \vdash i_1 : I_1 \ \ldots\ (\Gamma, X) \vdash i_k : I_k\end{array}}{(\Gamma, T) \vdash x : X}$$

$$(\mathit{ext\_comp\_selection}) \qquad \frac{(\Gamma, X) \vdash e_i : E_i}{(\Gamma, T) \vdash x.l_i : E_i}$$

$$(\mathit{int\_comp\_selection}) \qquad \frac{\begin{array}{c}(\Gamma, X) \vdash i_i : I_i \\ \vdash T <: X\end{array}}{(\Gamma, T) \vdash x.m_i : I_i}$$