Structure-Editing in Poly and ML

David C.J. Matthews

This note is a suggestion for the way to proceed with structure-editing in Poly and/or ML. It should be relevant to both the Programming Logics project and to other work in Poly and ML. N.B. throughout this note I have used structure to mean a data structure, typically a tree. When the word is used in the sense of a module in ML this is made clear in each case.

1 Type System

Existing structure-editors, such as the Mentor system from INRIA, have a type-system for the structures to ensure some consistency. One wants to be able to declare, that when editing a parse-tree for example, a tree denoting a declaration cannot be moved to a place in the tree where an expression is expected. The editor can inform the user that a particular operation would break the type-rules in the same way that a compiler will refuse to compile a statement in a programming language which is type-incorrect.

Operations performed directly by the user on the tree require dynamic type-checking, that is each operation must be type-checked when it is invoked. It is not possible for a programmer to decide in advance what sequence of operations a particular user is going to want, so it is impossible to type-check it beforehand.

There is however a place for static type-checking in such a system. It is often convenient to be able to write sequences of editing commands together and treat them as subroutines to be invoked from the console. Since we have a dynamic type-checking system for the primitive commands we could use that for the subroutines by simply checking each primitive command as it is applied. It is much better, though, to check the subroutine when it is created to ensure that it is type-correct. It should be possible, with an appropriate type-system, to check that the application was valid in a particular context, i.e. that its arguments are correct, and then the internals need not be checked on each application.

The structures we create and manipulate with the editor are going to be used by other programs. If we are editing the structure of some proof we want to be able to run the proof-checker on the resulting structure. The data structure used by the proof-checker will probably be an ML datatype. It would be sensible if the structure editor, rather than operating on a parallel data structure, and having the ML program translate it into the proof-checker's structure, could use the same structure.

This has several advantages. The immediate saving is in the cost of translating the data structures. It also provides us with a type-system for the structure. Transformations are allowed only if they conform to the ML type declarations. Operations on the tree are performed by ML functions, usually executed dynamically from the console, but it is possible to construct more complicated functions by composing the primitives. Because these would be written in ML the normal type-checking rules would be applied and in addition they could perform arbitrary computations.

Editing ML datatypes is relatively easy because the datatype definition gives a list of the constructors and their types. So, given a datatype value and its type definition the editor can break the structure down and display it.

There are a few disadvantages. The most obvious is that ML datatypes cannot in general be destructively updated. There are many reasons for not allowing the editor to have a privilege forbidden to other programs, so instead we must settle for allowing only copying of structures with replacement. This should not be too expensive since the editor can keep track of which parts of a tree have been modified and which have not.

2 User Display Functions

The editor can use the datatype definitions to print out the structure, but this will only be in a general fashion. It would be much better if the user could write a function to display a particular structure which the editor could then invoke. Unlike Poly, ML has no way of attaching functions directly to datatypes so the only way of doing this would be either by using a name such as `T_display` to display a value of a type called `T`, or having a structure (in the sense of a module in ML) `T` containing a function `display`. As well as the value to be displayed the function would have to be given the size of the window on the screen in which the value is to be displayed. In this way it could calculate how much of the structure to show.

User display functions also present problems with "picking" values from the screen. We want the ability to select a node by pointing at something with the mouse. This means translating screen co-ordinates into a node, requiring an inverse of the display function. Since the user's display function laid out the text on the screen, only a similar function can interpret some screen co-ordinates as a node in the structure. Apart from the problem of actually finding the node from the co-ordinates there is also the problem of the type of the value returned.

```
datatype expression =
  num of int | ident of string | let_in of declaration * expression
and declaration = Declaration of string * expression
```

This is the parse tree for a simple language with numbers and identifier declarations. It contains values of various types any of which should be able to be selected by the "pick" function. If we selected a number we should be able to get that number out of the structure. We could create a new union type containing all the types in the structure and declare the "pick" function as returning a value of that type. Alternatively we could introduce the idea of a universal type similar to `dynamic` in Amber, and coerce the value to that type. While the idea of introducing dynamic type-checking in ML seems a step backwards it is really the only solution to this problem.

We introduce two infinitely overloaded functions, `dynamic,` which can be applied only to values of types where all the type constructors have global scope and and `coerce,` whose result must have been unified to such a type. `coerce` raises an exception if the value cannot be unified to the result type.

```
dynamic: 'a -> dynamic
coerce: dynamic -> 'a
exception coerce: unit
```

Dynamic values contain a value together with its type and are the objects that the compiler uses when top-level declarations are made. Picking a value on the screen would return a dynamic value. It could probably also return a function which would accept a value of the appropriate type and rebuild the structure with it. The editor can then ensure that a structure is only rebuilt with the appropriate values.

3 Structure Editing in Poly

In Poly the implementation of a type is hidden and the only information available is the signature of the operations. A type defined as a union is initially created with injection, projection and tag-test operations, but it is easy to redefine these, add new operations or hide them. Unlike a datatype in ML it is not possible to examine the type to decide how to take a value apart. However in Poly it is relatively easy to associate a function with a type so we could insist that only structures with the appropriate functions can be edited, and not provide any default display behaviour based on the type definition. As with ML we will have to introduce dynamic type-checking. Rather than have `dynamic` and `coerce` as functions these would be reserved words with a syntax similar to function applications. This allows the restriction that only values involving types with global scopes to be enforced. The other reason is that because type checking is not done by unification the signature that `coerce` is to return must also be passed as an argument.

4 Operation of the Editor

The editors working on ML or on Poly will be fairly similar apart from the way they construct functions from a type when given a value to edit, so for the moment we will draw the examples from ML. We can think of the editor as a front end to the ML compiler. It generates strings of ML which are passed through the compiler and executed. While in practice it would be more tightly bound into the compiler so as to speed up the process, it is useful to think of the operation in this way.

A structure is edited by passing a value of type `dynamic` and a window to the editor. Normally this will be done by giving the name of a structure stored in the global name space and creating a new top-level window. The editor examines the type of the value it has been given and then generates a call to the appropriate display function, passing it the value and the window.

```
T_display(ROOT, window);
```

The display function recurses down the structure displaying it in some suitable form. Since it is given the area of the window it has to work in it can choose to display as much as will fit and elide the remainder. If the mouse button is clicked in this window the pick function for this type is called. This either returns the node selected or raises an exception if the co-ordinates do not lie in an identifiable part of a node.

```
val PICK =
   let val picked = T_pick(ROOT, window, mouse_x, mouse_y)
   in coerce(picked): type_of(picked)
   end
```

The above would not be valid ML even with the addition of dynamic types but it illustrates the kind of code that would be generated. The point is that the result held in PICK is a value of a particular type depending on the type of the value selected, not a dynamically typed value. If a number is selected the result will be that PICK is declared as the value of the number and type int. This is rather like "it" in ML which holds the value of the last expression evaluated. Its type depends on the type of the result of the expression.

There are two reasons for selecting a node, we may want the node itself or we may want to replace the node with a value from somewhere else. Selecting a node can be used as a way of zooming in on part of a structure because having selected a node we can then display the structure in the same or another window using the selected node as the root.

Replacing a node cannot be done directly, instead we have to generate a new structure by copying the old one with the replacements. Regeneration must in general be done by a user function because there may be other constraints on the data as well as type-consistency which must be preserved. If we are editing an ordered list of integers we cannot replace a particular node by an arbitrary value. We have an analogue of the "pick" function, called "poke" which rebuilds the structure with a particular node replaced by a given value. An alternative would be to have the "pick" function return a function to rebuild the structure as well as returning the current value. Replacing the currently selected node involves applying this function to the new value.

```
val ROOT =
  T_poke(ROOT, dynamic NEW_VALUE, window, mouse_x, mouse_y);
```

The new value is passed in as a dynamic because the type required depends on the position in the structure where the new value is to be placed. The result is a new root for the structure.

5 Conclusions

1. We have to introduce dynamic types into ML and Poly.

2. For each structure we wish to edit we must provide display, pick and poke functions. These are responsible for the layout of the structure and any consistency checks on the data. They will typically be written in terms of display, pick and poke functions of other types.

3. The editor will take care of overall screen management. It will have to have some information about the data structures used to represent ML and Poly types in order to be able to generate appropriate function calls. It might be

possible to abstract some of this out to make the editor more independent of the type system but I expect there will still have to be some dependencies on the type system.