Bill Ackerman

structures

VIMVAL MINUTES -- 2/19/85

Participants: Jack Dennis, Bhaskar Guharoy, Suresh Jagannathan, Brad Kuszmual, Earl Waldin

Minutes by: Earl Waldin

The meeting began with a discussion of a paper (Earl Waldin, "Communication of 2/14/85) handed out before the meeting by Earl. This paper discussed some of the problems with modules as they are defined in the current proposal (Earl Waldin, "VimVal, Changes and Additions to the Val Preliminary Reference Manual," working paper of 6/12/85). The paper addresses the difficulty of writing modules for simple recursive functions, and using modules to define constants. Jack first pointed out that the example module on page 2 of the paper for the factorial function was not the only way to write this module. An alternative would be:

```
module (n) returns(integer)

    function factorial(i : integer) returns(integer);
        if i = 0 then 1 else n*factorial(i-1) endif
        endfun

    factorial(n)
    endmodule
```

This eliminates the need to invoke the module using `fact()(m)` as described in the paper. Instead, the above module would be invoked by `fact(m)`. However, Earl pointed out that in either case, the invocation requires two function invocations: one for the module, and one for the function defined within the module. Although the compiler may be able to eliminate this extra invocation under certain circumstances, the semantics of two function invocations would still be visible to the programmer.

The discussion turned to the section of the paper describing how modules are used to describe constants. Everyone agreed that the paper correctly outlined the problems in this area. The proposal that modules define objects instead of just functions was then addressed. Several examples from the paper were put on the blackboard to illustrate the proposal. Everyone agreed that allowing modules to define objects solves the problems raised in the paper, and that this should be incorporated into the VimVal language. Finally, Brad pointed out that it is irrelevant as to when the value of a module was actually computed. This would eliminate the distinction made by the paper between compile time constants and computed constants. As far as the programmer was concerned the expression giving the value of the module (i.e. the module's body) could be any expression. Whether the compiler evaluates constants at compile time, or the run time system computes constants at program initialization time, was independent of the semantics.

The discussion then turned to the role of anonymous functions. This issue was also addressed by Earl's paper. Everyone agreeded that, in principle, the paper's points in favor of anonymous functions were correct. In addition it was pointed out that, because VimVal encourages the use of function objects as arguments and results of functions, many functions would be used anonymously. The actual reference to a function at any moment would be via an identifier to which the function was assigned. If anyone wanted to refer to a function within some scope, then the **let** expression should be used to introduce a name by assigning an identifier to the function. It was also felt that this was the correct way to introduce recursive function definitions. The group then decided that anonymous functions would be the only mechanism for introducing function definitions in VimVal. The use of named functions as in Val would be eliminated. All that remained was to choose a syntax for doing so. Jack objected to the use of the word *lambda* for several reasons. To begin with it already had too many different meanings within the computer science literature. It was not only identified with the lambda calculus, but it was also used in lisp for introducing imperative procedures. Furthermore, the use of lambda varied from language to language. Using lambda in VimVal would only add to the confusion, especially for programmers who had no previous experience with functional languages. A more suitable name was needed. Of course, the word **function** immediately sprang to mind. Earl raised the issue that this might conflict with the use of **function** for a type specification. After a short discussion, it was decided that there is no conflict since the context in which a type specification could occur would make the grammar unambiguous, as well as readable for the programmer. The syntax adopted would be along the lines of:

function (*arguments*) **returns**(*results*) **is** *expression* **endfun**

where *arguments* lists the names of the formal parameters, *results* lists the types of the results, and *expression* is the body of the function. In keeping with the spirit of type inference, the types of the formal parameters as well as the **returns** part are optional.

The discussion then turned to the use of dummy type names and the role of the module header. It was agreed that type names are useful for partial type specifications where the programmer wishes to indicate constraints on the type without having to give a full type specification. Type names must be declared as such, just as free names in a module must be declared. Type names are introduced by the use of the keyword **type** as in:

type t;

For type names which appear in the header of a module, the name must be declared in the **where** clause. Those which appear in the body of a module must be declared in the type definition part of the module. The following example of the use of a type name was put on the board and discussed:

```
module function(t) returns(t)
   where type t;

   function(n) is n endfun
   endmodule
```

Brad also pointed out that the type inference system would also allow the following definition:

```
module t
   where type t;

   function(n) is n endfun
   endmodule
```

This would, of course, make the module header cryptic and useless to the user of the module. It was decided that the module header must contain the maximum amount of information about the type of the module, including any type constraints. For example, the following would be the header for a sort function which takes an array of some type and an ordering predicate on that type and returns a sorted array:

```
module function(collection, predicate) returns(collection)
  where
    type t;
    type collection = array[t];
    type predicate = function(t,t) returns(boolean);
```

It was pointed out that the compiler could infer enough type information from the module's body to be able to check that the header provided the correct level of information. It was then decided that the programmer must supply the maximum amount of type information in the header about the type of the module (i.e. the least general type), and that the compiler should enforce this.