Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

Problem Set 7 Solutions

1. Provide in ML an abstract type for scalar physical quantities such as mass or angular momentum which have magnitudes and dimensions in SI units. Implement a pure abstract type (one where values are not updatable) and use the ML module construct, not the ML `abstype` construct, which is obsolete. Discuss briefly the design decisions you make.

   Note that there are several different so-called metric systems, but only one internationsl standard metric system. This system of units is called SI, which stands for *Système International* in French. For information on SI see `http://www.kumc.edu/ASP/si_units.html`.

   [Solution due to George Xylomenos.] First, let us look at the concrete data type needed for this problem. A scalar physical quantity can be characterized by a scalar *value* and a set of *dimensions*. Since we use SI, we know that all such dimensions can be assembled using the basic physical entities of *mass* (in kilograms), *length* (in meters) and *time* (in seconds), with some of them present and some missing, and each one raised to an integer power, positive or negative. For example, acceleration is measured in m/s^2, which is actually length raised to the first power, time raised to the negative second power, and mass raised to zero, so that it disappears. We can then think of a physical quantity as a four-tuple of values: the scalar value and the three integer exponents for the dimensions, with zero signifying that the corresponding dimension is not used for a value. Note that a pure number, i.e. a scalar with no dimensions, which may be used as a multiplicative factor in an expression using physical quantities, can be also represented in this scheme by having all the exponents set to zero. This will prove quite useful in extending our implementation.

   The second thing we have to do is decide on the operations that we want to provide on values of the type. Since by enclosing the type in a module we make its users unable to access its implementation directly, bypassing the defined operations for the abstract data type, we must provide enough functions for the users so that they can manipulate the values of the type as they please without rewriting the module to satisfy their particular needs. On the other hand, we probably do not know what exactly the users want for their applications, and we also do not want to overload the type with more functions than the average user will need, so as not to inconvenience users needing simple things. The most economical solution is to provide all functions that the user absolutely needs to manipulate the values of the type, and avoid providing functions that can be synthesized from simpler ones. Thus, if a user wants extra functionality, he can build it over the base functions provided, and if he does not want it, he does not have to put up with many operations taking space in his machine.

   If we take this approach, we must find what is the minimal set of operations. We must include at least one function to construct values of the type and one function to display these values, since there is no way to use the type if we cannot build some instances of it and if we cannot see somehow what are the values that we manipulate. Then, we need some functions that produce new values by combining existing ones (since the data type does not provide mutable objects, these functions do not modify existing values but produce new ones) using the arithmetic operators. We need addition and subtraction or negation for simple manipulations. Since we can produce a negated value by subtracting the initial value from a zero value of the same type, we choose to include subtraction only. For these functions, the values must be compatible, that is, expressed in the same physical dimensions, so our functions must make this check, raising an exception when provided with incompatible values. The second set of operations is multiplication and division or reciprocal, where again the last two can simulate one another, with our verdict favoring division (dividing a value of one using a simple scalar, i.e. with dimensional exponents set to zero, provides the reciprocals). Here, no dimensional checks are needed, and the result of an operation is a value with new dimensions in general. Note that we do not need multiplication with scalars since we can build scalar values in our type (using zero exponents) and use them to get results from the standard multiplicative operations.

It is conceivable that users will not like these choices between operations that can emulate each other; however, they can use the provided functions to build a new one that implements the operation they need using a more convenient interface. The key point is that the functionality can be extended using the basic operations. Normally, most users will be satisfied with these operations, probably augmented in straightforward ways, so they will not have to change the type. Note that the semantics are what one would expect, since we do "type checking" for dimensions when this is needed. There can arise problems though, for example with negative masses. Users must deal with these semantic problems in *ad hoc* ways, since there can be no provision for everything, especially for semantic peculiarities of particular physical quantities, or their combinations. Now, let us see the code for this implementation:

```
signature SIsig =
   sig
      exception SITypeError;
      eqtype SI;
      val make : (real*int*int*int) -> SI;
      val add  : SI*SI -> SI;
      val sub  : SI*SI -> SI;
      val mult : SI*SI -> SI;
      val divd : SI*SI -> SI;
      val show : SI -> unit;
   end;
```

We begin with the signature for the abstract data type, using it both because we will hide some details irrelevant to the users and because we will use the ML `abstraction` mechanism in the definition, which requires that a specific signature be provided in a structure definition. The elements should be self-explanatory; we define an exception to report type errors in addition and subtraction, we specify that the type can be tested for equality, and provide a function that prints values of the type using side-effects. The definition of the type is:

```
abstraction SIstruct:SIsig =
   struct
      exception SITypeError;
      datatype SI = ValSI of real * int * int * int;
      fun make (s,m,l,t) = ValSI (s,m,l,t);
      fun compat (ValSI x, ValSI y) =
         (#2 x = #2 y andalso
          #3 x = #3 y andalso
          #4 x = #4 y)
      fun add (ValSI x, ValSI y) =
         if compat(ValSI x, ValSI y)
         then
            make(#1 x + #1 y,#2 x,#3 x,#4 x)
         else
            raise SITypeError;
      fun sub (ValSI x, ValSI y) =
         if compat(ValSI x, ValSI y)
         then
            make(#1 x - #1 y,#2 x,#3 x,#4 x)
         else
            raise SITypeError;
      fun mult (ValSI x, ValSI y) =
         make(#1 x * #1 y, #2 x + #2 y,
              #3 x + #3 y, #4 x + #4 y);
      fun divd (ValSI x, ValSI y) =
         make(#1 x / #1 y, #2 x - #2 y,
              #3 x - #3 y, #4 x - #4 y)
      fun show (ValSI x) =
         (print(#1 x); print(" kg^"); print(#2 x);
          print("*m^"); print(#3 x); print("*sec^");
          print(#4 x); print("\n"));
  end;
```

We use `abstraction` instead of `structure` so that ML will not report the actual form of values of type `SI`. This is non-portable (it is an enhancement supplied by SML/NJ) but equivalent to a normal definition, apart from the requirement that an explicit signature must be provided. The function definitions are what one would expect, with the multiplication ones providing values with new dimensions and the addition ones making dimension checks using a function that is *not* exported. Inside the definitions we use the tuple selector operators for compactness. Note that since we hide the type definition completely, the only way to look at the values is through the `show` function, which is a simple pretty-printer. Now, a simple example of the use of these constructs:

```
open SIstruct;
- val s = make(2.0,2,3,4);
val s = - : SI
- val t = make(4.0,2,3,4);
val t = - : SI
- show(add(s,t));
6.0 kg^2*m^3*sec^4
val it = () : unit
```

Note how ML hides the datatype completely when we use `abstraction`.

What is bad about this implementation and what could, or should, change? The basic problem is that there is no direct access to the dimensions and to the scalar value itself; this means that a program cannot check a value for negativity, for example. The user's program cannot really tell what is inside a value: only the user can see a printed version in the output.

Providing inspector functions to show the scalar value and the exponents of the dimensions would not sacrifice information hiding, since users already know what the characteristics are of values of this type (the four-tuples used to construct them). An alternative interface would provide a function to build values of the type and four selector functions to inspect the components. Normally, the user would then decide what functions he wants and implement them himself. In this solution the data type would provide a set of low level primitives only, so that other programmers would define their own high level constructs (like addition and multiplication) in a desired way, by inspecting what is in the values themselves.

However, this is not a user-friendly approach, and it should *not* be mixed with user-friendly functions, like the ones in the provided implementation, since these would then be redundant. The preferred way to use such a low-level structure would be to include it as a substructure in a user-oriented structure definition that would provide what a specific application area needs. This is possible in ML, although it is not straightforward to implement, since it needs type sharing and similar advanced constructs to work. In fact this is a form of limited inheritance: the specific interface is derived from the primitive interface that we began with.

Similarly, the basic interface could be parameterized with respect to the data type of the scalar value (to use double precision arithmetic for example), which would then be instantiated using a `functor`. Unfortunately, ML is not good at supporting substructures, especially combined with functors, due to its lack of easily understandable mechanisms to support inheritance.

Note that dimensions could be built into the ML type system instead of supplied through an abstract data type. See *Dimension types* by Anthony Kennedy, Proceedings of European Symposium on Programming (ESOP'94), Edinburgh, pp. 348–62, Springer Verlag, 1994.

2. A Thcheme meta-interpreter is an interpreter for Thcheme written in Thcheme. Dr. Lisp says "Thcheme themanticth ith tho eathy," because he believes that the meanings of programs written in Thcheme can be defined by a Thcheme meta-interpreter. Unfortunately, Dr. Lisp is wrong, because most Thcheme meta-interpreters are ambiguous.

   Consider the Thcheme meta-interpreter below. Assume that the meaning of all predefined functions such as `car` and of all syntactic forms such as `(if` $\mathcal{C}$ $\mathcal{X}$ $\mathcal{Y}$`)` is well-defined, including whether they are strict and/or lenient. Explain why the meta-interpreter nevertheless does not specify the semantics of Thcheme unambiguously.

```
(define eval (exp envt fundefs)
    (if (number? exp) exp
    (if (symbol? exp) (assoc exp envt)
    (if (= (car exp) 'quote) (cadr exp)
    (if (= (car exp) 'if)
        (if (null? (eval (cadr exp) envt fundefs))
            (eval (cadddr exp) envt fundefs)
            (eval (caddr exp) envt fundefs))
    (if (userfun? (car exp) fundefs)
        (apply-userfun
            (assoc (car exp) fundefs)
            (evallist (cdr exp) envt fundefs)
            fundefs)
    (if (= (length exp) 2)
        (apply-unary-op (car exp) (eval (cadr exp) envt fundefs))
        (apply-binary-op
            (car exp)
            (eval (cadr exp) envt fundefs)
            (eval (caddr exp) envt fundefs))))))))

(define userfun? (f fundefs) (assoc f fundefs))

(define apply-userfun (fundef args fundefs)
    (eval (cadr fundef)
          (mkassoc* (car fundef) args '())
          fundefs))
```

   [Solution due to Nicholas Mitchell.] The main point to remember in answering this question is that the task is to discover *semantic* ambiguities, not *syntactic* ambiguities. A syntactic ambiguity results from a valid expression having two valid parse trees; the canonical $3+4+5$ is a good example. That is, the *specification* of the grammar does not indicate which parse tree is the correct choice.

   A semantic ambiguity, on the other hand, occurs when the language specification does not dictate uniquely some aspect of the behavior of some programs. Language characteristics that may be semantically ambiguous include scoping (static versus dynamic), the mode of parameter passing (call by value, reference, name, value-result, etc.), and order of evaluation (eager order, normal order, lazy, etc.). Here, the interpreter determines the mode of parameter passing unambiguously. Given an application of a function, `eval` always *eventually* (see below for this caveat), evaluates the argument expressions down to either numbers or symbols. Therefore, the interpreter dictates call-by-value semantics.

However, the interpreter fails to dictate order of evaluation. That is, the given interpreter code gives us no idea as to whether it uses, for example, eager order, normal order or lazy evaluation. Does this mean that the interpreter is non-functional? No, the evaluation order implemented by the meta-interpreter depends on the evaluation order of the language in which the interpreter is written. If we try

```
(eval '(<meta-interpreter code>) <global_env> <global_funs>)
```

in a interpreter which uses a certain evaluation order, the resulting meta-interpreter will use the same evaluation order.