Key Words in Context, an example

James Hook and Richard Kieburtz

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-012

August, 1990

Key Words in Context, an example

James Hook and Richard Kieburtz
Oregon Graduate Institute
hook@cse.ogi.edu

July 27, 1990

Abstract

This report presents the derivation of the "key words in context" example in the Extended ML framework of Sannella and Tarlecki[6, 7, 5, 8, 4, 9]. Experiences with the derivation are discussed.

1 Introduction

The production of verifiably correct programs without sacrifice of performance in execution has been a goal of computer scientists for years. Several methods for deriving programs from specifications have been proposed over the years, but few have demonstrated the capacity to scale up to realistic examples.

Specifications have generally followed one of two paradigms. A *logical* specification expresses logical relations over quantified object variables. The behavior entailed by a logical specification is inferred by constructing a realization if this is not manifest. Both first-order and higher-order logics have been used for specification. In principle, logics provide the means for composition of specifications, but this has not been very extensively developed linguistically. Composition of specifications seems clumsy in most logics.

An *algebraic* specification defines the operators of a many-sorted algebra, typically by imposing an equational theory. A carrier set for the algebra is usually not specified. In an algebraic specification, the behavior is manifest, but logical relations must be inferred in order to prove general properties. Algebraic composition is a natural and useful technique, but in a purely first-order algebraic system, new composition operators cannot themselves be specified.

Sannella and Tarlecki, building on extensive experience in algebraic specification languages, have recently developed Extended ML, a method for deriving ML programs that appears to provide sufficient tools for composition of component specifications to overcome the scaling problems. The method of developing a specification by refinement is still being fleshed out. This paper presents a non-trivial example developed according to their method. We offer comments on how this process might be automated and what demands it places on the logic.

In reviewing the text of this paper we have noticed, to our consternation, that the volume of words and symbols generated is at the verge of overwhelming the reader. The reader of

a specification must be able to concentrate his/her attention on specific details of concern. Having the visual field cluttered with text that dilutes the density of information content is distracting. There are several ways to mitigate this problem. (1) Some parts of component specifications are repeated many times in the linear text of the paper because they occur in several contexts. If one were reading the specification from the screen of a workstation, particularly one utilizing a hypertext-like environment, the duplications would be less distracting. (2) The text of specifications suffers from the wordiness of a programming notation. It would be easier to digest if it appeared in more familiar mathematical notation with abbreviated names and special symbols for operators such as universal quantification. Setting the text of axioms in an italic font in 'math mode' format would help to distinguish them from the declarations that appear in a signature. (3) At some points in the example, coercion functions are used to resolve otherwise overloaded operator names. Overloaded operator names are commonly used in mathematics, and a convention that permitted such overloading would be a boon to the readability of specifications.

In evaluating the specification language and methods we have used in this paper, the reader should also keep in mind that were a similar problem done in practice, the software designer would expect to have at his/her disposal libraries of existing module specifications and implementations. We have given everything needed for our example as if it were all created at one time, without benefit of an archive of prior work. Note that the module structure of the SML (and EML) language caters explicitly for the use of libraries by providing the interface structure and modes of composition that are needed. When this structure is superimposed on a modest example in which library modules are not used, it may appear cumbersome, but in practice it saves time and effort.

The example is not yet as complete as we would like. Specifically, it does not deal with I/O or display, character sets or fonts, or programmed exceptions. These can all be specified in EML and will be given in a future edition of this example.

1.1 Problem statement

The application we consider is one that has been used many times as an example. It is to generate, from a list of one-sentence titles, a list of permutations of the titles sorted by the occurrence of so-called key words. This is called a Key Words in Context (kwic) index.

More specifically, we require a sorted list of cyclic permutations of the original titles such that (1) each original title appears, (2) only those permutations appear that begin with a key word, and (3) only distinct titles and permutations appear in the output. A key word is any word that has not been explicitly noted as insignificant.

2 Specifying the problem

The Extended ML (EML) framework augments Standard ML's (SML) module declarations, a substructure intended to support programming in the large, to a calculus for specification and refinement. It generalizes the structures (large values) and signatures (large types) of

SML to include `axiom` declarations[3, 1, 2]. In the case of signatures, these axioms are viewed as specifications. In the case of structures they constrain the space of models in a manner consistent with the specification. Axioms are of use in structures during intermediate stages of the refinement process—once the structure is fully implemented with ML values there should be only one model. Extended ML also supports Standard ML's large functions—functors. This treatment is simply an extension of the treatment of structures to a parametric setting.

The Extended ML framework may be used with almost any logic. In the core of this development we use a first-order logic with equality, similar to that used in Sannella and Tarlecki[6]. Although EML is not implemented, this code does typecheck and run in Standard ML with the EML specific constructions commented out.

In our research we hope to develop a suite of related logics that can cope with richer subsets than the first-order logic in the main example. This would allow, among other things, the extension of the example to include its input-output specification. It would also support formal justification of the exploitation of higher-order functions used in section 3.1.

2.1 KWIC specification

To proceed from the informal problem description given above to a specification, we need to make many of the ideas more precise. We begin by typing the main function:

```
val kwic : string list list -> string list -> string list list
```

To express the ensemble of properties in the specification we will need:

1. a `Total_Order` structure, `ordered_words`, that captures the ordering of words,
2. a `Sorted_Total_Order` structure, `ordered_titles`, that encodes the ordering on titles,
3. a `Rotation` structure on titles, `permuted_titles`, that describes significant rotations, and
4. a `Repetition_Free_List` structure on titles, `unique_titles`, that describes the final output.

Given these structures we will develop axioms that specify:

1. `ordered_words` is a lexicographic ordering derived from the ASCII character code.
2. `ordered_titles` is a lexicographic ordering derived from `ordered_words`.
3. The output is sorted by the `ordered_titles` ordering and made unique by the `unique_titles` filter.
4. The output contains exactly the significant rotations (as defined in `permuted_titles`) of the input.

2.2 Supporting Concepts

To make our specification precise we need to build up a library of supporting concept specifications. These are the algebras of basic mathematical notions. In an EML specification, the operators of these algebras are expressed as programmed functions.

2.2.1 Order relations

One of the simplest component specifications in the system is that of an equivalence relation. It is expressed:

```
signature Equiv_Rel =
    sig
        type elem
        val eq : elem -> elem -> bool
        axiom (all x: elem. eq x x = true)
          and (all x,y:elem. eq x y = eq y x)
          and (all x,y,z:elem. eq x y and eq y z => eq x z)
    end;
```

This introduces a signature named `Equiv_Rel` and binds it to the signature that has a type called `elem`, an operation called `eq`, and three familiar axioms characterizing `eq` as an equivalence relation. In a nicer syntax these axioms appear as:

$$\forall x : elem \,.\, x \sim x$$
$$\forall x, y : elem \,.\, x \sim y = y \sim x$$
$$\forall x, y, z : elem \,.\, x \sim y \wedge y \sim z \rightarrow x \sim z$$

Where $\sim$ represents `eq`.

Total orders may be defined by incremental development on this basis, using the `include` signature construction operator. We extend equivalence relations to partial orders and partial orders to total orders as follows:

```
signature Partial_Order =
    sig
        include Equiv_Rel
        val less : elem -> elem -> bool
        axiom (all x: elem . less x x = false)
        and (all x,y: elem . less x y => not less y x)
        and (all x,y,z: elem . less x y and less y z => less x z)
   end;
```

Which augments the equivalence relation axioms with:

$$\forall x : elem \,.\, x \not\sqsubset x$$
$$\forall x, y : elem \,.\, x \sqsubset y \rightarrow y \not\sqsubset x$$
$$\forall x, y, z : elem \,.\, x \sqsubset y \wedge y \sqsubset z \rightarrow x \sqsubset z$$

Where we have used $\sqsubset$ for `less`. This is followed by the definition:

```
signature Total_Order =
    sig
        include Partial_Order
        axiom (all x,y: elem . less x y orelse less y x orelse eq x y)
   end;
```

which adds the restriction that $\forall x, y : elem \,.\, x \sqsubset y \vee y \sqsubset x \vee x \sim y$.

The reason we need total orders in this exercise is to produce a sorted list of output. Sorting is expressed by the following specification:

```
signature Sorted_Total_Order =
   sig
       include Total_Order

       val ordered : elem list -> bool
       axiom (all l:elem list.
              (all u,v:elem list.
               all x,y:elem.  l = u@[x]@[y]@v => less x y orelse eq x y)
              <=> ordered l)

       val permutation : elem list -> elem list -> bool
       axiom (permutation [] [] = true)
       and   (all l,u,v:elem list.
              all x:elem. permutation (x::l) (u@[x]@v) =
                            permutation l (u@v))

       val sort : elem list -> elem list
```

```
    axiom (all l,u:elem list.
            l = sort u => (permutation l u andalso
                           ordered l)
end;
```

The first axiom:

$$\forall l : \mathit{elem\ list}\,.\,(\forall u, v : \mathit{elem\ list}\,.\,\forall x, y : \mathit{elem}\,.\,l = u@[x]@[y]@v \rightarrow x \sqsubseteq y) \leftrightarrow \mathit{ordered}\,l$$

characterizes the meaning of ordered. The next two express permutations. Finally, a sort is characterized as an ordered permutation of the input.

2.2.2 Rotations

A list r is a *rotation* of a list l if r can be decomposed into two lists u and v such that $r = uv$ and $l = vu$. A rotation is *trivial* if either u or v is the empty string. A rotation is *significant with respect to a predicate* s if it is trivial or $s(\mathit{hd}\,r)$.

The formulation below focuses on the function `significant_rotations` which expects a significance predicate and a title and returns the list of all significant rotations of the title. This specification is captured in the signature:

```
signature Sig_Rotation =
    sig
        type elem

        val significant_rotations : (elem -> bool) -> elem list -> elem list list

        val rotation : elem list -> elem list -> bool
        val significant : (elem -> bool) -> elem list -> elem list -> bool

        axiom (all l, u, v: elem list.
                l = u@v => rotation l (v@u))

        and  (all l, r: elem list.
                rotation l r => exists u, v: elem list.
                                    u@v = l andalso v@u = r)

        and  (all sig_pred: elem -> bool.
              all l, r: elem list.
                 significant sig_pred l r
                    <=> l = r orelse
                        (rotation l r andalso sig_pred (hd r)))
```

```
        and  (all sig_pred: elem -> bool.
              all l, r: elem list.
                    significant sig_pred l r
                       <=> r is_in significant_rotation sig_pred l)
    end;
```

2.2.3 Sets

Since the significance predicate is going to be applied repeatedly, we choose to introduce a set abstraction for collections of insignificant words to support its implementation. This will allow us to easily change the module as more efficient implementations are required.

The EML signature for sets is:

```
signature Set =
   sig type elem
      type set
      val empty : set
      val is_member : elem -> set -> bool
      val insert : elem -> set -> set
      val list_members : set -> elem list
      axiom (all x: elem. is_member x empty = false)
        and (all x,y: elem.
             all s: set. (x=y) is_member x (insert y s) = true)
        and (all x,y: elem.
             all s: set. not (x=y) is_member x (insert y s) = is_member x s)
        and (all x: elem.
             all s: set.  x is_in list_member s <=> is_member x s)
   end;
```

2.2.4 Repetition free lists

Finally, to delete repetitions from the output, we specify lists without repetitions:

```
signature Repetition_Free_List =
    sig include Equiv_Rel
        val repetition_free: elem list -> bool

        axiom all l:elem list.
              repetition_free l <=>
                (all u,v: elem list,
                 all x,y: elem.
                     l = u@[x,y]@v => not eq x y)

        val remove_repetitions: elem list -> elem list
```

```
    axiom all l: elem list.
            repetition_free (remove_repetitions l)

    axiom all l:elem list, x:elem.
            x is_in l <=> x is_in remove_repetitions l

    axiom all u,v,w: elem list, x,y:elem.
            not eq x y =>
              exists u',v',w': elem list.
                u'@[x]@v'@[y]@w' =
                  remove_repetitions (u@[x]@v@[y]@w)
end
```

2.3 Sharing declarations

To integrate the specification it is necessary to relate the component structures to each other formally. In SML this is done with sharing declarations. Without an explicit sharing declaration, each occurrence of a type or structure declaration is taken to be distinct, even from another occurrence that is textually identical. In the ML nomenclature, this is called the *generative* property of declarations.

Standard ML (and EML) supports two sorts of sharing: type sharing and structure sharing. Either sort defines equivalences of names. Unfortunately, name equivalence is not rich enough to express the relationships in our example. Consider the relationship between `ordered_words` and `ordered_titles`. What we want to specify is:

```
sharing type ordered_titles elem = ordered_words.elem list
```

But the right hand side of this sharing specification is a type expression—not a name.

To accommodate to this restriction we shall augment the basic order types and order type functors by identifying derived structures and including in them the structure from which they are derived. To this end we define `Derived_Total_Order` and `Derived_Sorted_Total_Order` as:

```
signature Derived_Total_Order =
    sig
        include Total_Order
        structure elem_order:Total_Order

        val elem_to_list: elem -> elem_order.elem list
        and list_to_elem: elem_order.elem list -> elem
        axiom (all x:elem. x = list_to_elem (elem_to_list x))
        and   (all l:elem_order.elem list. l = elem_to_list(list_to_elem l))
```

```
        axiom (all y: elem_order.elem.
               all l: elem. less (list_to_elem [])
                                 (list_to_elem (y::(elem_to_list l))) = true)
        and   (all x,y: elem_order.elem.
               all xs,ys: elem.
               less (list_to_elem (x::(elem_to_list xs)))
                    (list_to_elem (y::(elem_to_list ys))) =
                         elem_order.less x y
                         orelse elem_order.eq x y
                                andalso less xs ys)
        and   (all l:elem. less l (list_to_elem []) = false)
    end;


signature Sorted_Derived_Total_Order =
    sig
        include Sorted_Total_Order
        structure elem_order:Total_Order
        val elem_to_list: elem -> elem_order.elem list
        and list_to_elem: elem_order.elem list -> elem
        axiom (all x:elem. x = list_to_elem (elem_to_list x))
        and   (all l:elem_order.elem list. l = elem_to_list(list_to_elem l))

        axiom (all y: elem_order.elem.
               all l: elem. less (list_to_elem [])
                                 (list_to_elem (y::(elem_to_list l)))
                             = true)
        and   (all x,y: elem_order.elem.
               all xs,ys: elem.
               less (list_to_elem (x::(elem_to_list xs)))
                    (list_to_elem (y::(elem_to_list ys)))
                 = elem_order.less x y
                   orelse elem_order.eq x y
                          andalso less xs ys)
        and   (all l:elem. less l (list_to_elem []) = false)
    end;
```

With these changes it is possible to express the relationship between the `Total_Order` `ordered_words` and the `Sorted_Derived_Total_Order` `ordered_titles` with the structure sharing constraint:

```
        sharing ordered_titles.elem_order = ordered_words
```

With these details completed, the specification can now be given in full.

```
signature KWIC_specification =
    sig
        structure ordered_words: Total_Order
            sharing type ordered_words.elem = string
        structure ordered_titles: Sorted_Derived_Total_Order
            sharing ordered_titles.elem_order = ordered_words
        structure permuted_titles: Sig_Rotation
            sharing type permuted_titles.elem = string
        structure unique_titles: Repetition_Free_List
            sharing type ordered_titles.elem = unique_titles.elem
        val kwic : string list list -> string list -> string list list

        axiom (all c:string ordered_words.less c "" = false)

        and (all c: string.
             all l: string list.
               (length (explode c) = 1) =>
                  (ordered_words.less "" (implode (c::l)) = true))

        and (all s,t:string.
             all x,y:string.
             all xs,ys: string list.
                ((explode s) = x::xs
                 andalso (explode t) = y::ys)
                => ordered_words.less s t
                     = (x < y) orelse (x = y)
                                   andalso ordered_words.less (implode xs)
                                                              (implode ys)

        and (all titles: string list list.
             all insignificant_words: string list.
             let val output = kwic titles insignificant_words
             in
                ordered_titles.sorted output
                andalso unique_titles.repetition_free output
             end)
```

```
    and (all titles: string list list.
         all insignificant_words: string list.
         all y: string list.
         y is_in kwic titles insignificant_words.
         <=> (y is_in titles
              orelse  (exists x : string list.
                         x is_in titles
                         andalso permuted_titles.rotation y x
                         andalso not (hd y) is_in insignificant_words)))
end
```

3 Deriving a solution

In the preceding section, we gave a signature with logical relations to specify the required behavior of the function `kwic`. The task is now to derive an SML structure that has the SML part of this signature and that satisfies all of the EML axioms given in the signature. The general approach of the EML method is to express specific implementations as structures and parametric implementations as functors. This example is specific, so we begin by giving the skeleton of a structure, using question marks to indicate the text to be filled in. This fully unspecified structure is nearly identical to the specifying signature; in most derivations we would expect an automated assistant to produce this text for us.

```
structure KWIC =
    struct
        structure ordered_words: Total_Order = ?
        structure ordered_titles: Sorted_Derived_Total_Order = ?
        structure permuted_titles: Rotation = ?
        structure unique_titles: Repetition_Free_List = ?

        val kwic : string list list -> string list -> string list list = ?

        axiom (all c:string ordered_words.less c "" = false)

        and (all c: string.
             all l: string list.
               (length (explode c) = 1) =>
                  (ordered_words.less "" (implode (c::l)) = true))
```

```
    and (all s,t:string.
         all x,y:string.
         all xs,ys: string list.
            ((explode s) = x::xs
             andalso (explode t) = y::ys)
            => ordered_words.less s t
                 = (x < y) orelse (x = y)
                                  andalso ordered_words.less (implode xs)
                                                             (implode ys)

    and (all titles: string list list.
         all insignificant_words: string list.
         let val output = kwic titles insignificant_words
         in
            ordered_titles.sorted output
            andalso unique_titles.repetition_free output
         end)

    and (all titles: string list list.
         all insignificant_words: string list.
         all y: string list.
         y is_in kwic titles insignificant_words.
         <=> (y is_in titles
              orelse  (exists x : string list.
                         x is_in titles
                         andalso permuted_titles.rotation y x
                         andalso not (hd y) is_in insignificant_words)))
end
```

The first piece required is the body of a `struct` for `ordered_words`. In fact, we guess from the first few axioms that `ordered_words` must be:

```
struct type elem = string
    fun less (x:elem) y = x < y
    fun eq (x:elem) y = x = y
end
```

Formally, the method allows us to retire the axioms specifying `ordered_words` at this stage, since the previous axioms are a consequence of this choice, according to the semantics of SML.

To refine the next `?` we postulate a functor that makes use of `ordered_words` to define `ordered_titles`. The functor, `words_to_titles`, is given the signature:

```
functor words_to_titles (T:Total_Order)
   : sig include Sorted_Derived_Total_Order
         sharing elem_order = T
     end
   = ?
```

Assuming this functor, the specification may be refined again, replacing the definition of `ordered_titles` with the functor application. The axioms constraining `ordered_titles` may now be omitted since they too have become derived rules.

Before continuing with the top level structure, we decide to refine the new functor into two parts—one that gives the lexical order and one that provides a sort function. These functors are `Lex` and `Derived_QSort`; their signatures are given:

```
functor Lex (T:Total_Order)
    : sig include Derived_Total_Order
          sharing T = elem.order
      end
= ?

functor Derived_QSort (L:Derived_Total_Order)
    : sig include Sorted_Derived_Total_Order
          sharing L.elem_order = elem_order
      end
= ?
```

In terms of these functors, `words_to_titles` may now be completely specified as:

```
functor words_to_titles (T:Total_Order)
    : sig
        include Sorted_Derived_Total_Order
        sharing elem_order = T
      end
    = Derived_QSort (Lex (T))
```

Since the EML signatures match exactly and the sharing specifications are propagated, no proof obligations are incurred.

We next refine `Lex`. It is sufficiently simple that it need not be decomposed further. By elaboration of the signature and sharing specification we obtain the outline below. Note that the axiomatization strongly suggests the use of `T.elem list` as the representation of the `elem` type.

```
functor Lex (T:Total_Order)
    : sig include Derived_Total_Order
          sharing T = elem.order
      end
= struct
    type elem = T.elem list
    structure elem_order = T

    val elem_to_list: elem -> elem_order.elem list
    and list_to_elem: elem_order.elem list -> elem
    axiom (all x:elem. x = list_to_elem (elem_to_list x))
    and   (all l:elem_order.elem list. l = elem_to_list(list_to_elem l))

    val less: elem -> elem -> bool = ?
    axiom (all x: elem . less x x = false)
    and   (all x,y: elem . less x y => not less y x)
    and   (all x,y,z: elem . less x y and less y z => less x z)
    and   (all y: elem_order.elem.
           all l: elem. less [] (y::l) = true)
    and   (all x,y: elem_order.elem.
           all xs,ys: elem.
               less (x::xs) (y::ys) = elem_order.less x y
                                      orelse elem_order.eq x y
                                             andalso less xs ys)
    and   (all l:elem. less l [] = false)

    val eq: elem -> elem -> bool = ?
    axiom (all x: elem. eq x x = true)
    and   (all x,y:elem. eq x y = eq y x)
    and   (all x,y,z:elem. eq x y and eq y z => eq x z)

    and   (eq [] [] = true)
    and   (all y: elem_order.elem.
           all l: elem. eq [] (y::l) = false)
    and   (all x,y: elem_order.elem.
           all xs,ys: elem.
               eq (x::xs) (y::ys) = elem_order.eq x y
                                    andalso eq xs ys)

    axiom (all x,y: elem . less x y orelse less y x orelse eq x y)
end
```

The first refinement step is to select `T.elem lists` for our elements. This gives identity functions for the ismorphism between elements and lists. The functor body can then be immediately refined using the simple recursive functions suggested by the axioms:

```
functor Lex (T:Total_Order)
    : sig include Derived_Total_Order
          sharing T = elem_order
      end
= struct
    type elem = T.elem list
    structure elem_order = T

    fun elem_to_list x = x
    and list_to_elem x = x;

    fun less [] [] = false
      | less [] (_::_) = true
      | less (x::xs) (y::ys) = T.less x y orelse
                               ((T.eq x y) andalso less xs ys)
      | less (_::_) [] = false

    fun eq [] [] = true
      | eq [] (_::_) = false
      | eq (_::_) [] = false
      | eq (x::xs) (y::ys) = (T.eq x y) andalso (eq xs ys)

end
```

This refinement requires that the axioms of the previous stage be derivable. While this is not simply signature checking, it is straightforward.

With `Lex` fully specified we turn to `QSort`—suggestively named after the quick sort algorithm which we intend to implement. We choose to refine a particular structure rather than compose functors. Elaborating the signature and sharing declarations yields the skeleton of the implementation:

```
functor Derived_QSort (L:Derived_Total_Order)
    : sig include Sorted_Derived_Total_Order
          sharing L.elem_order = elem_order
      end
=   struct
        structure elem_order = ?
        type elem = ?
```

```
        val elem_to_list: elem -> elem_order.elem list
        and list_to_elem: elem_order.elem list -> elem
        axiom (all x:elem. x = list_to_elem (elem_to_list x))
        and   (all l:elem_order.elem list. l = elem_to_list(list_to_elem l))

        val less: elem -> elem -> bool = ?
        axiom (all x: elem . less x x = false)
        and   (all x,y: elem . less x y => not less y x)
        and   (all x,y,z: elem . less x y and less y z => less x z)

        val eq : elem -> elem -> bool = ?
        and (all x: elem. eq x x = true)
        and (all x,y:elem. eq x y = eq y x)
        and (all x,y,z:elem. eq x y and eq y z => eq x z)

        and (all x,y: elem . less x y orelse less y x orelse eq x y)

        val ordered : elem list -> bool = ?
        axiom (all l,u,v,w: elem list.
               all x,y:elem. l = u @ [x] @ v @ [y] @ w
                             => (less x y orelse eq x y)

        val permutation : elem list -> elem list -> bool = ?
        axiom (permutation [] [] = true)
        and   (all l,u,v:elem list.
               all x:elem. permutation (x::l) (u@[x]@v) =
                             permutation l (u@v))

        val sort : elem list -> elem list
        axiom (all l,u:elem list.
                 l = sort u => (permutation l u andalso
                                ordered l)
    end
```

Since the goal is simply an extension of L, the components from L may be incorporated directly (using the open declaration) and all common axioms immediately discharged.

```
functor Derived_QSort (L:Derived_Total_Order)
    : sig include Sorted_Derived_Total_Order
          sharing L.elem_order = elem_order
      end
=   struct
        open L
```

```
    val ordered : elem list -> bool = ?
    axiom (all l,u,v,w: elem list.
           all x,y:elem. l = u @ [x] @ v @ [y] @ w
                         => (less x y orelse eq x y)

    val permutation : elem list -> elem list -> bool = ?
    axiom (permutation [] [] = true)
    and   (all l,u,v:elem list.
           all x:elem. permutation (x::l) (u@[x]@v) =
                permutation l (u@v))

    val sort : elem list -> elem list
    axiom (all l,u:elem list.
             l = sort u => (permutation l u andalso
                            ordered l)
end
```

The problem now is to guide the derivation from our intuition about the behavior of quicksort. The basic idea is to successively partition the set into separated subsets until it is decomposed to singletons. Then the subsets are glued together in order by list catenation. To start on this path, we introduce a predicate, `separated`, and function, `partition`, as follows:

```
val separated: elem -> elem list -> elem list -> bool
axiom (all a:elem.
       all u,v:elem list.
         separated a u v <=>
            all x:elem. member x u => less x a andalso
                        member x v => (less a x orelse eq a x))

val partition elem -> elem list -> (elem list * elem list)
axiom (all a:elem.
       all l:elem list.
         let val (u,v) = partition a l
         in permutation l (u@v) andalso
            separated a u v)
```

Since this only introduces new requirements, no verification is needed. The partition predicate is easily made concrete by the code:

```
fun partition a [] = ([],[])
  | partition a (x::l) = let val (u,v) = partition a l
                         in if less x a then (x::u, v)
                            else (u,x::v)
```

```
                        end;
```

Similarly, sort can be defined:

```
fun sort [] = []
  | sort [a] = [a]
  | sort (a::l) = let val (u,v) = partition a l
                  in (sort u)@[a]@(sort v)
                  end;
```

This step should be automatically justified. It is easily seen to follow from the facts:

```
ordered u andalso ordered v andalso separated a u v
    => ordered u@[a]@v
```

and

```
((u,v) = partition a l)
andalso permutation u u'
andalso permutation v v'
=> permutation (a::l) (u'@[a]@v')
```

While this completes the most interesting aspect of the definition, we are required to realize ordered, permutation and separated as well.

This completes the derivation of the `Derived_QSort` functor. It is important to note that there is more information about the logical development of the algorithm in its derivation than in its final presentation.

3.1 Significant rotations

We now return to the top level structure. The next task is to construct a significant rotation structure for the `permuted_titles` component. Since this structure deals with words, we refine the top level structure by postulating a functor, `Sig_Rotations`, mapping arbitrary structures to `Sig_Rotation` structures, and apply that to the `ordered_words` structure. Thus we add

```
permuted_titles = Sig_Rotations (ordered_words)
```

to our top level structure and declare the new functor:

```
functor Sig_Rotations (S:Triv)
    : struct include Sig_Rotation
             sharing type s.elem = elem
      end
= ?
```

The definition of significant rotations makes a special case of the trivial rotation. This complicates maters because it does not allow for a regular decomposition of the significant rotation

problem into rotations and filters, as seems most natural. Attempting to solve the problem directly is possible, but the result is not elegant. However, since duplicates are ultimately removed we can introduce duplicates in this module without changing the overall behavior of the program. This allows us to put in all of the trivial rotations and all of the rotations that start with a significant word.

To support the decomposition, the first step is to specify the interface between the components. In this case it is the Rotation signature.

```
signature Rotation =
    sig type elem
        val rotation : elem list -> elem list -> bool
        val all_rotations : elem list -> elem list list
        axiom all l, r: elem list.
            rotation l r => exists u,v:elem list . l = u@v andalso r = v@u
        axiom all l, r: elem list.
            rotation l r <=> r is_in all_rotations l
        axiom all l: elem list. length (all_rotations l) = length l
    end
```

This permits the functor decomposition:

```
functor Rotation (S:Triv)
    : sig include Rotation
          sharing type S.elem = elem
      end
= ?

functor Sig_Rotation' (R:Rotation)
    : sig include Sig_Rotation
          sharing type R.elem = elem
      end
= ?

functor Sig_Rotation (S:Triv)
    : sig include Sig_Rotation
          sharing type S.elem = elem
      end
= Sig_Rotation' (Rotation (S))
```

We first proceed with the Rotation functor. Rotations are cyclic permutations generated by the unit rotation. We begin coding the function by writing down the axioms from the signature and augmenting them with information about unit rotations. In an automated system this would probably be a two step process, the first done by the system and the second by the user.

```
functor Rotation (S:triv)
    : sig include Rotation
          sharing type S.elem = elem
      end
=   struct
        type elem = S.elem
        val rotation : elem list -> elem list -> bool = ?
        val unit_rotation : elem list -> elem list = ?
        val all_rotations : elem list -> elem list list = ?

        axiom all l, r: elem list.
            rotation l r => exists u,v:elem list . l = u@v andalso r = v@u
        axiom all l, r: elem list.
            rotation l r <=> r is_in all_rotations l
        axiom all a:elem, l:elem list.
            unit_rotation (a::l) = l@[a]
        axiom unit_rotation [] = []
        axiom all l: elem list. length (all_rotations l) = length l
    end
```

The `rotation` predicate and `unit_rotation` function are easily converted from axioms to code, yielding:

```
val rotation l r = r is_in (all_rotations l)

fun unit_rotation [] = []
  | unit_rotation (a::l) = l @ [a]
```

Given our knowledge that rotations are generated by iterating the unit rotation, a first cut at `all_rotations` would be:

```
fun  all_rotations l = l :: (all_rotations (unit_rotation l))
```

The only problem with this is that it diverges! Since we know there are only n distinct rotations of a sequence of length n, we can introduce the local function `n_rotations` to finitely iterate the `unit_rotation`.

```
val n_rotations : elem list -> int -> elem list list = ?
axiom all l : elem list. n_rotations l 0 = []
axiom all l : elem list, n: int.
    n_rotations l (n+1) = l::(n_rotations (unit_rotation l) n)
```

This specification is sufficiently specific that it should automatically be convertable to code. We then observe that `all_rotations` can be defined:

```
        fun all_rotations l = n_rotations l (length l)
```

The justification of this step may be non-trivial. Collecting all of these refinements together we discover that we have fully specified the rotation functor.

```
functor Rotation (S:Triv)
    : sig include Rotation
          sharing type S.elem = elem
      end
=   struct
        type elem = S.elem

        fun unit_rotation [] = []
          | unit_rotation (a::l) = l @ [a]

        fun n_rotations l =
             fn n => if n = 0
                        then []
                     else l::(n_rotations (unit_rotation l) (n - 1))

        fun all_rotations l = n_rotations l (length l)

        fun rotation l r = r is_in (all_rotations l)
    end
```

We now proceed to refine the `Sig_Rotation'` functor. Elaborating the body with the skeleton of the specification yields:

```
functor Sig_Rotation' (R:Rotation)
    : sig include Sig_Rotation
          sharing type R.elem = elem
      end
    = struct
        type elem = R.elem

        val significant_rotations : (elem -> bool) -> elem list -> elem list list = ?

        val rotation : elem list -> elem list -> bool = ?
        val significant : (elem -> bool) -> elem list -> elem list -> bool = ?

        axiom (all l, u, v: elem list.
                 l = u@v => rotation l (v@u))

        and  (all l, r: elem list.
```

```
            rotation l r => exists u, v: elem list.
                                u@v = l andalso v@u = r)

    and  (all sig_pred: elem -> bool.
          all l, r: elem list.
             significant sig_pred l r
                <=> l = r orelse
                    (rotation l r andalso sig_pred (hd r)))

    and  (all sig_pred: elem -> bool.
          all l, r: elem list.
                significant sig_pred l r
                   <=> r is_in significant_rotation sig_pred l)
end;
```

We can immediately discharge the `rotation` predicate by using `R.rotation`. It is then trivial to implement the significance predicate:

```
fun significant sig_pred l r
       = l = r orelse (rotation l r andalso sig_pred (hd r))
```

Now, to get significant rotations (with possible duplicates) we want to write:

```
fun significant_rotations sig_pred l
      = l :: (filter (sig_pred o hd) (R.all_rotations l))
```

Unfortunately `filter` is a higher-order function, and there is no way in the simple logic that we have attempted to use to discover the correctness of this refinement step. Research into logics for EML that support higher-order reasoning is being carried out in several places, including OGI, Bremen, and Edinburgh. We will continue with the development as if this step were justifiable.

Collecting these refinements we get:

```
functor Sig_Rotation' (R:Rotation)
    : sig include Sig_Rotation
          sharing type R.elem = elem
      end
    = struct
        type elem = R.elem

        val rotation = R.rotation

        fun significant sig_pred l r
               = l = r orelse (rotation l r andalso sig_pred (hd r))
```

```
    fun significant_rotations sig_pred l
          = l :: (filter (sig_pred o hd) (R.all_rotations l))
  end
```

3.2 Sets

To implement the `kwic` function at top level, we anticipate taking the list of insignificant words and building a function that characterizes membership in that set. To support this, we have introduced the set abstraction earlier. At this point we extend the top level `KWIC_specification` structure with the declaration below, and derive the `Set` functor:

```
structure word_set = Set(ordered_words)
```

The set abstraction may be provided for any structure with an equivalence predicate. We express this by the following EML functor:

```
functor Set(Rel:Equiv_Rel)
    : sig include Set
          sharing type elem = Rel.elem
      end
    = ?
```

We refine this by supplying the structure derived from the specification. Electing to use `elem lists` for our set type, the specification yields:

```
functor Set(Rel:Equiv_Rel)
    : sig include Set
          sharing type elem = Rel.elem
      end
    = struct
        type elem = Rel.elem
        type set = elem list
        val empty : set = ?
        val is_member : elem -> set -> bool = ?
        val insert : elem -> set -> set = ?
        val list_members : set -> elem list = ?
        axiom (all x: elem. is_member x empty = false)
          and (all x,y: elem.
                all s: set. (x=y) is_member x (insert y s) = true)
          and (all x,y: elem.
                all s: set. not (x=y) is_member x (insert y s) = is_member x s)
          and (all x: elem.
                all s: set.  x is_in list_member s <=> is_member x s)
      end
```

We then use the obvious implementation for sets as lists. An automated assistant would be expected to provide some level of help in translating these axioms to function definitions.

```
functor Set(Rel:Equiv_Rel) : Set =
   struct
      type elem = Rel.elem
      type set = elem list
      val empty = []
      fun is_member x [] = false
        | is_member x (y::ys) = Rel.eq x y orelse is_member x ys
      fun insert x s = x::s
      fun list_members s = s
   end
```

3.3 Repetition-free lists

We will obtain repetition-free lists by a functor application.

```
functor Delete_Repetitions (E: Equiv_Rel)
    : sig include Repetition_Free_list
          sharing type elem = E.elem
      end
    = ?
```

This will be used in the top level structure by including the declaration:

```
structure unique_titles = Delete_Repetitions (ordered_titles)
```

We refine the functor directly, simply elaborating the specification:

```
functor Delete_Repetitions (E: Equiv_Rel)
    : sig include Repetition_Free_list
          sharing type elem = E.elem
      end
    = struct
        open E
        val repetition_free: elem list -> bool = ?

        axiom all l:elem list.
              repetition_free l <=>
                (all u,v: elem list,
                 all x,y: elem.
                     l = u@[x,y]@v => not eq x y)

        val remove_repetitions: elem list -> elem list = ?
```

```
axiom all l: elem list.
        repetition_free (remove_repetitions l)

axiom all l:elem list, x:elem.
        x is_in l <=> x is_in remove_repetitions l

axiom all u,v,w: elem list, x,y:elem.
        not eq x y =>
          exists u',v',w': elem list.
            u'@[x]@v'@[y]@w' =
              remove_repetitions (u@[x]@v@[y]@w)

end
```

The `repetition_free` predicate is easily implemented:

```
fun repetition_free [] = true
  | repetition_free [_] = true
  | repetition_free (a::(l as (b::_))) =
      (not (eq a b)) andalso repetition_free l
```

To develop the `remove_repetitions` function we augment this with a function that takes an element and a list and returns a list that deletes all initial occurrences of the element.

```
val strip_prefix : elem -> elem list -> elem list

axiom all x:elem, u,v : elem list.
    strip_prefix x (u@v) = v
        <=> (all y:elem. y is_in u => eq y x)
            andalso all z:elem, w:elem list.
                v = z::w => not eq z x
```

We also augment the specification with a function that can handle intermediate results.

```
val remove_repetitions_2 : elem_list * elem_list -> elem_list

axiom all l:elem list. remove_repetitions_2 (l,[]) = l
axiom all l,u: elem list. all x:elem.
        remove_repetitions_2 (l,x::u) =
            remove_repetitions_2 (l@[x],strip_prefix x u)
```

Assuming this function, `remove_repetitions` is trivially implemented. The final functor body is:

```
functor Delete_Repetitions (E: Equiv_Rel)
    : sig include Repetition_Free_List
          sharing type elem = E.elem
      end
    = struct
        open E

        fun repetition_free [] = true
          | repetition_free [_] = true
          | repetition_free (a::(l as (b::_))) =
              (not (eq a b)) andalso repetition_free l

        fun strip_prefix x [] = []
          | strip_prefix x (l as (y::u)) =
              if not (eq x y) then l else strip_prefix x u

        fun remove_repetitions_2 (l,[]) = l
          | remove_repetitions_2 (l,x::u) =
                    remove_repetitions_2 (l@[x],strip_prefix x u)

        fun remove_repetitions l = remove_repetitions_2 ([],l)
      end
```

3.4 A solution

It is now time to assemble all of our top level refinements and examine the specification. At this point we have supplied all of the specified structures and one additional supporting structure. The current refinment is:

```
structure KWIC =
    struct
        structure ordered_words: Total_Order =
          struct type elem = string
              fun less (x:elem) y = x < y
              fun eq (x:elem) y = x = y
          end

        structure ordered_titles: Sorted_Derived_Total_Order =
          words_to_titles (ordered_words)

        structure permuted_titles: Sig_Rotation =
          Sig_Rotation (ordered_words)
```

```
structure unique_titles: Repetition_Free_List =
  Delete_Repetitions (struct type elem = string list
                              fun eq (x:string list) y = x = y
                      end)

structure word_set: Set =
  Set (ordered_words)

val kwic : string list list -> string list -> string list list = ?

and (all titles: string list list.
     all insignificant_words: string list.
     let val output = kwic titles insignificant_words
     in
        ordered_titles.sorted output
        andalso unique_titles.repetition_free output
     end)

and (all titles: string list list.
     all insignificant_words: string list.
     all y: string list.
     y is_in kwic titles insignificant_words.
     <=> (y is_in titles
          orelse  (exists x : string list.
                     x is_in titles
                     andalso permuted_titles.rotation y x
                     andalso not (hd y) is_in insignificant_words)))
end
```

At this point all that remains is to supply the top level function **kwic**. Intuitively, this will have four parts: construct the significance predicate, collect the rotations, sort the entries, and remove duplicates from the list. As we already have the machinery to build the last two, we will specify the first two:

```
val make_sig_pred: string list -> string -> bool = ?
axiom all l: string list, x:string.
         make_sig_pred l x
         <=> not x is_in l
```

Note that this function really is intended to be used as a higher-order function. If the development were being carried out in pure first-order EML, this would not be allowed. Instead sets could be passed around everywhere where the significance predicates are used in this derivation. That would satisfy the first-order constraints of this version of the logic.

The collection of entries is specified:

```
val collect_entries :
   (string -> bool) -> (string list list) -> (string list list)
   = ?
axiom (all titles: string list list.
       all sig_word_predicate: string -> bool.
       all y: string list.
       y is_in collect_entries sig_word_predicate titles
       <=> (y is_in titles
            orelse  (exists x : string list.
                       x is_in titles
                       andalso permuted_titles.rotation y x
                       andalso sig_word_predicate (hd y))))
```

With these functions, kwic is easily given:

```
fun kwic titles insig_words =
       let sig_word_pred = make_sig_pred insig_words
       in
         unique_titles.remove_repetitions
           (ordered_titles.sort
              (collect_entries sig_word_pred titles))
       end
```

The proof checker will have to verify that repetition removal does not reorder the elements and that all elements of the collected entries are present in the final function value, but these follow easily from the component specifications.

The two remaining functions are trivially implemented by higher-order list operations fold and concmap. They can, of course, be built without these functions as well. That development would proceed:

```
fun build_set [] = word_set.empty
  | build_set (a::l) = word_set.insert a (build_set l)

fun make_sig_pred l =
      let insig_set = build_set l
      in fn x => not (word_set.is_member x insig_set)
      end

fun collect_entries sig_pred [] = []
  | collect_entries sig_pred (t::l) =
         (permuted_titles.significant_rotations sig_pred t)
           @ (collect_entries sig_pred l)
```

The final, derived top level structure is:

```
structure KWIC =
    struct
        structure ordered_words: Total_Order =
          struct type elem = string
              fun less (x:elem) y = x < y
              fun eq (x:elem) y = x = y
          end

        structure ordered_titles: Sorted_Derived_Total_Order =
          words_to_titles (ordered_words)

        structure permuted_titles: Sig_Rotation =
          Sig_Rotation (ordered_words)

        structure unique_titles: Repetition_Free_List =
          Delete_Repetitions (struct type elem = string list
                                  fun eq (x:string list) y = x = y
                              end)

        structure word_set: Set = Set (ordered_words)

        fun build_set [] = word_set.empty
          | build_set (a::l) = word_set.insert a (build_set l)

        fun make_sig_pred l =
             let val insig_set = build_set l
             in fn x => not (word_set.is_member x insig_set)
             end

        fun collect_entries sig_pred [] = []
          | collect_entries sig_pred (t::l) =
                (permuted_titles.significant_rotations sig_pred t)
                   @ (collect_entries sig_pred l)

        fun kwic titles insig_words =
               let val sig_word_pred = make_sig_pred insig_words
               in
                 unique_titles.remove_repetitions
                   (ordered_titles.sort
                      (collect_entries sig_word_pred titles))
               end
end
```

Testing with the SML of New Jersey compiler yields the encouraging results:

```
- KWIC.kwic [["Tora","Tora","Tora"],
             ["It"],
             ["Crime","and","Punishment"]]
            ["it","and"];
val it = [["Crime","and","Punishment"],
          ["It"],
          ["Punishment","Crime","and"],
          ["Tora","Tora","Tora"]] : ?.E.elem list
```

4 Comments

This example represents our first significant experience with EML—and we have identified some problems, both with our approach and with the language and methodology. In doing the example we have made some observations which we feel are important.

4.1 Provable vs. testable equalities

In designing signatures, we have exported the defining concepts as well as their implementations. For example, in the `Sig_Rotation` signature, the `rotation` predicate was defined. Since there is no way to distinguish defining characteristics from executable Boolean predicates, this requires such predicates to be implementable. This dictates that rotations be over structures with equality.

Since provable, but untestable, defining characteristics are common in computer science, we think EML should support such a concept. In one draft we called these "`metaval`" declarations.

This observation also points out an anomaly in SML. There are two ways to get a structure with equality: an explicit structure, as used in our example, or an implicit structure exploiting `eqtypes`. It seems that there is no advantage to using the `eqtype` constuctor. Anything it can express can be expressed more accurately with the module facility.

We have also found, unsurprisingly, that the first-order logic used to illustrate EML does not support the higher-order programming style to which we are accustomed. It seems very unnatural to eschew higher-order, polymorphic functions while "programming" in an "extension" of ML. This is simply a deficiency in the logic. As has been noted elsewhere, the methodology and framework are largely independent of the details of the logic—alternative logics are being actively explored.

5 Conclusions

We are still debating which aspects of this derivation show strengths and which show weaknesses of the EML language and methodology. We believe, however, that EML is a major step along the path to an environment usable by programmers to formally derive real programs. However,

research is needed in (1) logics that allow us to express more of ML, (2) logics that allow us to reason elegantly about higher-order fragments, (3) frameworks in which such logics can be used together (if they differ) and (4) implementations that support the derivation process and the examination of derivations.

We are also investigating the application of term rewriting techniques to improve the code we have derived with this methodology. Our current vision is to support code improvement by transformation as a major component of an environment for producing correct code with satisfactory performance. We have notes on applying transformations to this example, and expect to generate a technical report on this topic soon.

References

[1] Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *TAPSOFT '87*, pages 308–319. Springer-Verlag, March 1987.

[2] David B. MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, 1986.

[3] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[4] Donald Sannella. Formal program development in Extended ML for the working programmer. Technical Report ECS-LFCS-89-102, Laboratory for the Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland, December 1989.

[5] Donald Sannella and Fabio da Silva. Syntax, typechecking and dynamic semantics for Extended ML. Technical Report ECS-LFCS-89-101, Laboratory for the Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland, December 1989.

[6] Donald Sannella and Andrzej Tarlecki. Program specification and development in standard ML. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 67–77. ACM, January 1985.

[7] Donald Sannella and Andrzej Tarlecki. Extended ML: an institution-indepdendent framework for formal program development. In David Pitt, Samson Abramsky, Axel Poigné, and David Rydeheard, editors, *Category Theory and Computer Programming Tutorial and Workshop*, volume 240 of *Lecture Notes in Computer Science*, pages 364–389, Berlin, 1986. Springer-Verlag. Meeting was held in September 1985 in Guildford, UK.

[8] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: foundations and methodology. Technical Report ECS-LFCS-89-71, Laboratory for the Founda-

tions of Computer Science, Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland, December 1989.

[9] Donald Sannella and Andrzej Tarlecki. The semantics of extended ML, March 1990. Draft, not for distribution.