Note on Conditional Compilation in Standard ML

Nicholas Haines   Edoardo Biagioni   Robert Harper
Brian G. Milnes

June 1993

CMU-CS-93-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-93-02

Abstract

In the Fox project we make frequent use of conditional code within Standard ML functors: code which depends on the functor arguments. The canonical example is instrumentation code for testing or debugging, only present when flags in the functor arguments are set. We would like the object code of instantiations of these functors to be as efficient as possible, *i.e.*, to omit the code when the argument flags are not set. This note considers compilation techniques to achieve this goal, and code styles which do well in the absence of such techniques.

Authors' electronic mail addresses are:
`Nicholas.Haines@cs.cmu.edu`, `Edoardo.Biagioni@cs.cmu.edu`, `Robert.Harper@cs.cmu.edu`, and `Brian.Milnes@cs.cmu.edu`.

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

Keywords: Standard ML, modules, functors, conditional compilation, code style, optimizing compilers

1 Introduction

This note discusses the issue of conditional code within functors in Standard ML[4]; *i.e.*, code paths which depend on the functor arguments. The canonical example is debugging or testing code, which is included or excluded depending on a boolean flag in the functor arguments:[1]

```
[a]     functor F (val debug : bool) =
            struct
                . . .
                fun f a = . . .
                          if debug then e1
                                   else e2
                          . . .
                . . .
            end

        structure Debug = F(val debug = true)

        structure Production = F(val debug = false)
```

This style of code is frequently used in situations which would see the use of preprocessor conditionals in C or C++:

```
[b]     #ifdef DEBUG
                printf("n = %d\n",n);
        #endif /* DEBUG */
```

However, no code is generated for [b] when the debugging flag is off. This is not true for [a], at least using release 0.93 of the Standard ML of New Jersey compiler[1] (the most recent at the time of writing). The code generated for the function `f` includes code to test the value `debug` and to evaluate expression `e1` or `e2` accordingly. This disadvantage may become important when Standard ML is considered for systems programming tasks, a goal of the Fox project[2].

The problem reduces to this: Standard ML has a way of *expressing* the opportunity for conditional compilation, namely the use of functors with boolean arguments, but Standard ML compilers do not currently *perform* any conditional compilation because of the manner in which they compile functor declarations and applications. The compilation technique used is semantically correct but generates inefficient code.

This note describes current Standard ML compilation techniques and explores possible coding styles and compiler optimizations that may improve the situation. Work is also proceeding on extending the Standard ML language in such a way as to enforce conditional compilation, but that is outside the scope of this note.

2 Code Generation for Functors

The use of a functor in [a] to express conditional code suggests that the decision on which version of the code to compile should be made at functor application time, when the flag `debug` is available.

[1] All code fragments in this note are identified by a single letter.

That is, the compilation of the structure declarations for `Production` and `Debug` should include code generation for those structures.

Unfortunately, typical compiling techniques for Standard ML compile each functor declaration as if it were a simple function taking a record argument. That is, functor `F` is compiled approximately as if it were this function:

```
[c]     fun F {debug : bool, ...} = { . . .,
                                       f = fn a => . . .
                                                   if debug then e1
                                                         else e2
                                                   . . . ,
                                       . . . }
```

When `F` is compiled, code is generated for `f` which obtains the value `debug` from the current closure and tests it, evaluating `e1` or `e2` accordingly. The applications of `F` to create the structures `Debug` and `Production` merely create a closure for `f` containing the value of `debug`.

Since functor application occurs in a very different context to function application (*i.e.*, functor applications are statically known), this compilation technique is not really suitable to the needs of programmers. It would be more useful if the compiler kept some intermediate representation of the functor (such as the abstract syntax) and generated code for the functor body at functor application time. In general this would shift compilation time from functor evaluation to functor application, equivalent in C to postponing optimization until link-time. A more sophisticated approach might consider compiling functor declarations as partially evaluated structure declarations, and specialize the evaluation at functor application time.

However, the Fox project's FoxNet software depends on currently available compilers, so we need a coding style that is compatible with the current functor compilation technique.

3 Possible Coding Styles

Given the current techniques used for functor compilation, how can we rewrite our code to produce the best runtime performance in functor bodies with conditional code?

Consider the example code [d], from our Ethernet driver testing and timing functor. The values `Debug.include_prints`, `Debug.do_prints` and `do_prints` are all functor arguments. Note that the conditional code here obscures the real purpose of the function `generic_receive`. Fragment [e] is a version without any instrumentation.

Assuming that all the debugging arguments are `false`, the code executed for [d] will be very similar to that for [e], except for either 2 or 3 boolean tests, depending on the path taken. Each test will take 3 or 4 instructions, and if we have decided to take some handicap, this is not an unacceptable one. But the code is very opaque. It can be trivially improved by combining two of the flags (code fragment [f]).

We can improve fragment [f] further by abstracting the instrumentation into a function (fragment [g]). This is now much more legible than [d], but we have introduced function calls to `debug_append` into the code. Function calls are usually more expensive than boolean tests because of register saving; even a call to an empty function may take a dozen or more instructions. Also this mechanism is not very general. It can be made more general, as shown in [h].

[d]

```
fun generic_receive from expected_data size history body p =
   (if Debug.include_prints andalso do_prints
       andalso ! Debug.do_prints
    then history := "." :: ! history else ();
    if sent_from from p then
     (if Debug.include_prints andalso do_prints
         andalso ! Debug.do_prints
      then history := "+" :: !history else ();
       (if valid_data p size expected_data then
        (if Debug.include_prints andalso do_prints
            andalso ! Debug.do_prints
         then history := "!" :: !history else ();
         body())
        else (* The message has invalid data. *)
          (if Debug.include_prints andalso do_prints
              andalso ! Debug.do_prints
           then history := "?" :: !history else ())))
    else (* The message is not from the server. *)
     (if Debug.include_prints andalso do_prints
         andalso ! Debug.do_prints
      then history := "-" :: !history else ()))
```

[e]

```
fun generic_receive from expected_data size history body p =
  if sent_from from p
     andalso valid_data p size expected_data
  then body ()
  else ()
```

[f]

```
val static_print = Debug.include_prints andalso do_prints

fun generic_receive from expected_data size history body p =
   (if static_print andalso ! Debug.do_prints
    then history := "." :: ! history else ();
    . . .
```

[g]

```
fun generic_receive from expected_data size history body p =
       let
         fun debug_append s =
                if static_print andalso !Debug.do_prints
                then history := s :: ! history else ();
       in
         (debug_append ".";
          if sent_from from p then
             (debug_append "+";
              . . .
       end
```

```
[h]     fun debug_action f arg = if static_print andalso !Debug.do_prints
                                 then (f arg) else ()

        fun generic_receive from expected_data size history body p =
               let
                 val debug_append =
                        debug_action (fn s => history := s :: ! history)
               in
                 (debug_append ".";
                  if sent_from from p then
                     (debug_append "+";
                      . . .
               end
```

The debug_action function can now be used around any instrumentation function call in the functor body: a very general mechanism. However, in the case that the debugging switches are off, we are now doing 3 or 4 function calls *and* 2 or 3 boolean tests each time we call generic_receive. This is substantially worse than in our original code, and seems an expensive price to pay. We can remove the boolean tests easily enough:

```
[i]     val debug_action = if static_print then
                                (fn f => (fn arg => if !Debug.do_prints
                                                    then (f arg)
                                                    else ()))
                           else (fn _ => (fn _ => ()))
```

generic_receive now only has an overhead of 3 or 4 function calls. If we uncurry debug_action we can reduce that by one, at the cost of some legibility in the body of generic_receive.

Note that this final version, [i], is crucially different from its predecessors in that it evaluates a condition at functor-application time. This difference, caused by "hoisting" the condition up through the surrounding lambdas, suggests the following solution to our problem:

```
[j]     local
          fun generic_receive_opt from expected_data size history body p =
            if sent_from from p
               andalso valid_data p size expected_data
            then body ()
            else ()

          fun generic_receive_debug from expected_data size history body p =
           (if ! Debug.do_prints
            then history := "." :: ! history else ();
            . . .)
        in
          val generic_receive = if static_print
                                then generic_receive_debug
                                else generic_receive_opt
        end
```

This code fragment expresses perfectly our intention: that the functor application should test the `static_print` flag and create a function `generic_receive` accordingly. However, it is very bad from the point of view of code modifications; there is no guarantee that `generic_receive_opt` and `generic_receive_debug` are equivalent, and therefore differences will inevitably arise in code like this when code is modified.

This code also incurs a space overhead under SML/NJ. The code generated for both functions `generic_receive_opt` and `generic_receive_debug`, will be placed in a single indivisible code object on the heap. Even if the functor is applied only once, the codes for both functions will remain 'live data' on the heap, since they both lie in the 'live' code object.

4 Possible Compiler Optimizations

The various code structures in section 3 are attempts to work around a problem which can be viewed as a failure in the compiler. Functor applications are static (in the sense that there is a finite number of them in the compilation of any piece of Standard ML, determined only by the number of occurrences of the functor-application syntax), and so the compiler should apply all the means at its disposal to optimize functor application. This is similar to the argument for link-time optimization of C or other languages, and the effect is similar: cross-module optimization. As discussed in section 2, such an implementation of functors is certainly possible.

Short of a major re-implementation of functors, what improvements can be made to the compiler to improve the compilation of conditional code?

The most obvious possibility is to introduce condition-hoisting, to change code [d] to code [j] automatically. This is akin to the techniques used in 'binding-time improvement' in partial evaluators[3, Chapter 12]. The case of interest to us is a static `if` expression inside a function declaration. In the Bare language, this has the following form:

`fn` *match*, where *match* contains the subexpression e_1:

  e_1 : `(fn true =>` e_2 `| false =>` e_3`)` `(`v`)`

  and v is a *var* free in *match*.

We transform this into:

`(fn true  => (fn` *match* $[e_2/e_1]$`)`
`  | false => (fn` *match* $[e_3/e_1]$`))` `(`v`)`

This transformation is in effect already performed outside of functors by constant-folding; adding it only affects functors by making this limited form of constant-folding explicit so that it can be carried out at functor-application time. Note that the transformation could lead to code size explosion (but in typical codes will not), and can be governed by heuristics based on the sizes of *match*, e_2 and e_3. If either e_2 or e_3 is a trivial expression (such as `()`), this transformation is a significant improvement for that side of the resulting expression.

We can generalize this transformation in two ways. Firstly if v is a general expression, the transformation can still be applied if the compiler can determine that the value of v cannot change after the `(fn` *match*`)` expression is evaluated. This is a property of expressions which can be useful to the compiler in other contexts (outside of functors, it indicates a value which can be computed at compile-time), so determining it for v does not present additional difficulty. Secondly the transformation can be generalized to types of v other than `bool`.

Note that the code style preferred in section 3 (as shown in code fragment [i]) will not benefit from this transformation. Indeed, adopting code style [i] will *reduce* efficiency in the presence of this transformation, since the function-call overhead will not be eliminated.

5 Our Decision

We have decided to use a code style which expresses our intention clearly, such as this:

```
[k]     functor F (val debug : bool
                  . . .) =
           struct
               . . .

               val f = if debug then fn a => . . .
                                 else fn a => . . .
               . . .
           end
```

This code is ready for an improved compiler that optimizes at functor application time. It will not benefit from minor improvements to the compiler, such as a condition-hoisting transformation. The software-engineering risk of writing several versions of many functions is removed by using this general construct:

```
[l]     val debug_action = if static_debug then
                                 (fn (f,arg) => if !dynamic_debug
                                                    then (f arg)
                                                    else ())
                           else (fn _ => ())
```

The `debug_action` function is then wrapped around any instrumentation code. It is not curried because SML/NJ doesn't optimize away the currying of functions.

6 Conclusions

In conclusion,

- Conditional code is supported in many other languages, such as C, C++, Modula-3 and Common Lisp, either by use of macros or by constant-folding `if` structures. Conditional code is required for code instrumentation or for ready portability.

- Standard ML supports conditional code through the use of functors taking switches as arguments. This technique is very natural, and similar to the Modula-3 `GENERIC` style of modular programming.

- Current Standard ML compilers do not efficiently compile instantiations of functors which include conditional code. They must do so if Standard ML is to have comparable speed to other languages used in systems programming.

- Code generation at functor application time is the most natural way to achieve this efficiency.

- There is a code transformation (presented in section 4) which achieves this efficiency in some cases, and should be simple to add to a Standard ML compiler. This transformation requires conditional code to be written in a particular style.

- We have chosen a code style which makes the most of current compilation techniques and will also take advantage of a compiler which generates code at functor application time. This style will not be optimized by the described code transformation.

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[2] Eric Cooper, Robert Harper, and Peter Lee. The Fox Project: Advanced development of systems software. Technical Report CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, August 1991.

[3] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[4] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.