A Functorial, Mostly Functional Approach to the Model-View-Controller Software Architecture in Standard ML

Alley Stoughton
Kansas State University
stough@cis.ksu.edu

August 16, 2006

Abstract

We report on our attempt to transfer the model-view-controller software architecture to a mostly functional setting in Standard ML (SML). In our approach, a controller's algorithm is apparent; it doesn't have to be mentally pieced together from a set of event handlers. Furthermore, what would be the state of a model (domain-specific part of program) in an object-oriented setting becomes arguments to the functions of the controller in our setting. For us, a controller mediates between its view (user-interface) and model. The controller calls its view to get user input, and to display results to the user; it calls its model to do domain-specific work. To increase adaptability, a controller is an SML functor, parameterized by its view and model. In our experience, one can write controllers that work simultaneously with both terminal- and graphical-based views. Of particular note is the way we are able to allow certain computations of the model to be monitored and aborted by the user, via the view. Our approach makes much use of SML's module system and higher-order functions, and we illustrate it with a case study of a complete application. The application's code is available for downloading.

1 Background

Most Standard ML (SML) programs have user-interfaces. Terminal user-interfaces can be built using the facilities of the Standard ML Basis Library [1]. Alternatively, one can build graphical user-interfaces (GUIs) using a GUI toolkit, e.g., eXene [2] or sml_tk [3]. Surprisingly, and in stark contrast to the situation with object-oriented programming, it seems that very little has been written about the software architecture of SML programs with user-interfaces. How should the module system be used when writing such programs? How can the programmer facilitate the switching from one kind of user-interface to an-

other. How should a program with multiple user-interfaces best be structured? SML programmers must answer these questions for themselves.

In the object-oriented world, programs are typically structured according to the *model-view-controller* (MVC) software architecture [4, 5]. According to this architecture, a program should be structured as three objects/classes:

- a *model*, which is concerned with the simulation of the application domain;
- a *view*, which handles the presentation of the model to the user; and
- a *controller*, which determines how the user may interact with the model.

In the original Smalltalk-80 MVC scheme [4], user input wasn't seen as coming from the view, but more recent presentations of MVC typically consider the view to be both an input and output medium. In MVC, user input is passed from the view to the controller, which may ask the model and view to update themselves. When the model updates itself, it notifies registered observers—e.g., the controller and view—of the changes; the observers may ask the model for more information about its state. Changes in the model may trigger changes in the appearance of the view and the behaviour of the controller.

Although models are not supposed to know about controllers and views, there is typically a fairly close dependence between controllers and views. Nonetheless, the separation of concerns leads to programs that are better structured and more adaptable. Some more recent presentations of MVC, e.g., that of Apple's Cocoa framework [5], specify that any communication between views and models should be mediated by their controllers, claiming better modularity and adaptability as consequences of this choice.

Although MVC is typically used for constructing graphical user-interfaces, terminal-based users-interfaces can also be structured according to this architecture. Whereas the controller of a program with a GUI typically consists of event handlers, responding to events generated from a user's interaction with the view, the controller of a program with a terminal-based user-interface would typically be the locus of control.

2 Functorial, Mostly Functional MVC in SML

This paper reports on our attempt to transfer MVC to a mostly functional setting in SML. Although it would be possible to structure a controller in SML as a set of event handlers, we reject this approach as unnecessarily imperative. In our approach, the algorithm of the controller is apparent; it doesn't have to be mentally pieced together from a set of event handlers. Furthermore, what would be the state of a model in an object-oriented setting more naturally should be arguments to the functions of the controller in our setting. In our approach, a controller mediates between its view and model. The controller calls its view to get user input, and to display results to the user; it calls its model to carry out domain-specific work.

To increase adaptability, a controller in our setting is an an SML functor, parameterized by its view and model. Different views (user-interfaces) can then be written and supplied as arguments to the same controller. Our experience indicates one can write controllers that work simultaneously with both terminal- and graphical-based views. Different models can be written too, carrying out the domain-specific computations using different algorithms. This makes it easy to generate multiple versions of a program, e.g., by using the conditional compilation feature of Standard ML of New Jersey's (SML/NJ's) compilation manager [6].

It is important that a controller be a functor, instead of a structure that uses a particular view and model. This way the controller may be typechecked independently from a model and view. In fact, it may be typechecked before a model and view are written. Furthermore, the functor's type makes clear the sharing constraints that relate the types of the model and view.

Of particular note is the way we are able to allow certain computations that are carried out by the model to be monitored and aborted by the user, via the view. Our approach makes much use of SML's module system and higher-order functions.

3 Related Work

C. Lüth and B. Wolff have contructed an SML functor transforming a model for a theorem-prover-like application into an sml_tk-based GUI for that application [7]. Their functor can be seen as a combined controller and view.

The Unison [8] file synchronizer is an example of an ML (OCaml) program that can be built with both terminal and graphical user-interfaces. However, this program isn't structured in the way we propose, with a shared controller. I was unable to find examples of other ML programs with multiple user-interfaces, or examples of programs with controllers parameterized by their models and views.

4 Case Study: A Cryptogram Encoder/Decoder

In this section, we illustrate our functorial, mostly functional approach to the MVC architecture using a case study of a complete application. We consider the implementation of Crypto, a Linux/Unix/Mac OS X program for encoding and decoding cryptograms. The decoding process is structured as a game. The are two versions of the program, one with a terminal-based user-interface, and one with an X window system graphical user-interface. The program makes use of some special features of the Standard ML of New Jersey (SML/NJ) implementation of SML [6], including its signal processing facilities, Concurrent ML (CML) [9] and the X window system toolkit eXene [2]. The carefully commented code and further documentation for crypto are available at:

`http://www.cis.ksu.edu/~stough/crypto/`

Crypto's implementation should be useful as an example in teaching.

4.1 Specification

Crypto makes use of a *lexicon* *lex*, consisting of a set of *words*, which are nonempty sequences of lowercase letters. A *message* is a list of *lines*, each of which is a list of words. A *renaming* *ren* is a bijection over the lowercase letters. We *apply* a renaming *ren* to a message by applying *ren* to each letter of each word of each line of the message. A *decoding* of a message *msg* is a message msg' such that

- each word of msg' is in the lexicon *lex*;
- msg' can be formed by applying some renaming to *msg*.

Crypto has primary and secondary command loops. In the terminal-based version of Crypto, a command's arguments are listed after the command. In the GUI version of the program, commands are selected by clicking on buttons, after which the user is prompted to enter or select any command arguments.

Upon invocation, Crypto enters its *primary command loop*. There are primary commands for quiting (quit), loading a lexicon from a file (lexicon), generating a random encoding of a message (encode), and interactively decoding a message (decode). The encoding process first checks that the supplied message is the unique decoding of itself. This checking can take a long time in the worst case, and the user is allowed to abort it (by interrupting in the terminal version, and clicking on the Cancel button in the GUI version).

The decoding process begins with an abortable check that the supplied message has a unique decoding. If it does, that decoding is saved but not reported to the user. Then, Crypto enters its *secondary command loop*, in which the user attempts to decode the message. At each iteration of the secondary command loop, Crypto first displays the current *partially decoded message* (*pdm*) consisting of a sequence of *partially decoded lines*, each of which consists of a sequence of *partially decoded words*, i.e., nonempty sequences of upper- and lowercase letters. Initially, this partially decoded message is *msg*. The uppercase letters represent the renamings already performed by the user and program.

It will always be the case that the current partially decoded message, *pdm*, is *consistent* with the message *msg* being decoded, i.e.,

- *msg* and *pdm* have the same shape, and
- for all lowercase letters a, either
  - a appears in *msg* at exactly the same positions as it appears in *pdm*, or
  - there is an uppercase letter b such that a appears in *msg* at exactly the same positions that b appears in *pdm*.

Suppose *pdm* is a partially decoded message that is consistent with the message *msg* being decoded, and that msg' is the decoding of *msg*. We say that *pdm* is *decodable* iff each occurrence of an uppercase letter in *pdm* appears

```
-----
iEEe Id THE zOjEST Id THE jAeIiLs iEkjEASIdq LIqHT
IT tAS iIzzIkmLT TO TELL tHO tAS AeejOAkHIdq mS
-----
secondary command >> hint
replacing d by n
-----
iEEe IN THE zOjEST IN THE jAeIiLs iEkjEASINq LIqHT
IT tAS iIzzIkmLT TO TELL tHO tAS AeejOAkHINq mS
-----
secondary command >> replace j r
-----
iEEe IN THE zOREST IN THE RAeIiLs iEkREASINq LIqHT
IT tAS iIzzIkmLT TO TELL tHO tAS AeeROAkHINq mS
-----
secondary command >>
```

Figure 1: Terminal Version Snapshot

in the same position in msg', but in its lowercase form. We say that *pdm* is *decoded* iff *pdm* is decodable and has no lowercase letters. Given a lowercase letter a that appears in *pdm*, we say that the *decoding* of a is the lowercase letter b that appears in msg' at the positions corresponding to the positions at which a appears in *pdm* (i.e., the positions at which a appears in *msg*).

There are secondary commands for exiting the program (quit), returning to the primary command loop (abort), checking whether the current pdm is decodable (check), asking for a hint (hint), replacing a lowercase letter by a fresh uppercase letter (replace) and undoing the last replacement (undo). The hint command complains if the current pdm isn't decodable; otherwise, it replaces the lowercase letter of the pdm that occurs most often (ties are broken by favoring earlier letters in the alphabet) with the uppercase version of its decoding. The check and hint commands return to the primary command loop if the current pdm is decoded. The undo command returns to the primary command loop if there are no replacements (either carried out by replace or hint) to undo.

Figures 1 and 2 contain execution snapshots of similar stages of the executions of the terminal and GUI versions of Crypto. The second-to-last pdm of the terminal version and the pdm of the GUI version are identical. In the last pdm of the terminal version, the user has just replaced j by r. In the GUI version, the first and second rows of buttons correspond to the commands of the primary and secondary command loops, respectively. Only certain buttons are enabled/active. In the snapshot, the user has already elected to replace the letter j of the pdm, and is being asked to select one of the indicated letters as its replacement (these are the only letters not already in uppercase in the pdm), or to cancel the replacement.

4.2 Basic Data

Although both versions of Crypto work with words made up of lowercase letters, everything but the views/user-interfaces is written so as to allow other choices

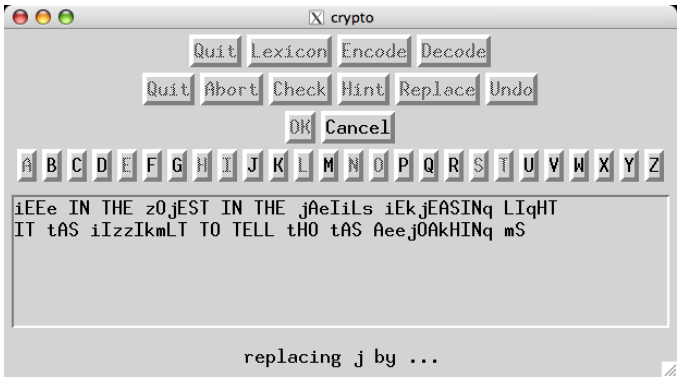

Figure 2: GUI Version Snapshot

of basic symbols. There is a standard linear ordering signature, `LIN_ORD`, whose type is called `elem`, along with lexicon (`LEXICON`, with main type `lexicon`) and set (`SET`, with main type `set`) signatures based on a linear ordering. The lexicon signature includes a function for checking whether there is a word in a lexicon matching a certain kind of pattern.

These signatures are used to defined a signature `DATA` whose structures define the basic data used by Crypto; see Fig. 3. The value `symbols` consists of the elements of `sym` that may actually be used, e.g., in words. The functor `DataFunc` takes in a linear ordering and a list of symbols, and forms a data structure. Here, as elsewhere, we use opaque ascription to a signature in which the non-abstract types have been specified using `where type`. This functor is then applied to a linear ordering based on characters, plus the lowercase letters, to form the `Data` structure.

4.3 Model/Coding

Figure 4 contains the coding signature, functor and structure, which constitute the model of our program. The signature `CODING` builds on the signature `DATA`. It contains functions for:

- returning the symbols of a message;
- converting a message to a pdm in which all symbols are old;
- making a replacement in a pdm;
- getting information about the decodings of a message;
- getting a hint about a pdm and how it should be altered, given the message from which it was formed, the old symbols of the pdm, and the decoding of the pdm;
- returning the words of a message that aren't in the lexicon; and

```
signature DATA =
sig
  structure SymLinOrd : LIN_ORD
  structure SymLexicon : LEXICON where type LinOrd.elem = SymLinOrd.elem
  structure SymSet : SET where type LinOrd.elem = SymLinOrd.elem

  type sym = SymLinOrd.elem
  type sym_lexicon = SymLexicon.lexicon
  type sym_set = SymSet.set

  val symbols : sym_set  (* symbols actually allowed *)

  type word = sym list
  type line = word list
  type msg = line list

  structure WordSet : SET where type LinOrd.elem = word  (* sets of words *)

  type word_set = WordSet.set

  datatype pds = Old of sym | New of sym  (* partially decoded symbol *)
  type pdw = pds list  (* partially decoded word *)
  type pdl = pdw list  (* partially decoded line *)
  type pdm = pdl list  (* partially decoded message *)
end

functor DataFunc(structure LinOrd : LIN_ORD
                 val symbols : LinOrd.elem list) :>
          DATA where type SymLinOrd.elem = LinOrd.elem =
struct
  ...
end

structure Data =
            DataFunc(structure LinOrd = CharLinOrd
                     val symbols = explode "abcdefghijklmnopqrstuvwxyz");
```

Figure 3: Data Signature, Functor and Structure

```
signature CODING =
sig
  include DATA

  val symsMsg : msg -> sym_set
  val msgToPDM : msg -> pdm
  val replacePDM : sym * sym -> pdm -> pdm

  datatype decodings =
            DecodingsNone | DecodingsUnique of msg | DecodingsMultiple

  val decodings :
        (IntInf.int -> bool) * sym_lexicon * msg ->
        IntInf.int * decodings option

  datatype hint = HintDecoded | HintNotDecodable | HintReplace of sym * sym

  val findHint : msg * pdm * sym_set * msg -> hint
  val unknownWordsMsg : sym_lexicon * msg -> word_set
  val encode : msg -> msg
end

functor CodingFunc(structure Data : DATA) :>
          CODING
            where type SymLinOrd.elem     = Data.SymLinOrd.elem
            and   type pds                = Data.pds
            and   type SymSet.set         = Data.SymSet.set
            and   type SymLexicon.lexicon = Data.SymLexicon.lexicon
            and   type WordSet.set        = Data.WordSet.set =
struct
  open Data
  ...
end

structure Coding = CodingFunc(structure Data = Data);
```

Figure 4: Coding Signature, Functor and Structure

- generating a random encoding of a message.

As expected, the `decodings` function takes in the lexicon *lex* and a message *msg*. Ideally, it would simply return a value of type `decodings`. But making the necessary determination can take a long time in the worst case (try "the quick brown fox jumps over the lazy dog"), at least with the algorithm we are using, and so it must be possible to abort this computation, at an observer's discretion, and to provide the observer with feedback as to the progress of the computation. For this purpose, `decodings` takes in a function *ca* (for "check abort"), and returns an optional value of type `decodings` along with an integer. It periodically calls *ca* with the number of "steps" it has completed so far, and uses the value returned by ca to determine whether it should abort (`true` means abort, `false` means don't abort). When `decodings` aborts, it returns the number of steps completed plus `NONE`; when it terminates normally, it returns the number of steps completed, plus `SOME` of the answer. The *ca* function may be called a very large number of times, and so should be fast. In another

application, *ca* might return some other indication of how much progress had been made so far.

The `encoding` function needs a source of random numbers. As can be inferred from the signature, this source must be stored in a mutable variable, encapsulated in the model. This is the one instance in Crypto's design when it seemed best for the model to have state. In other applications, the model might have more or less state.

The functor `CodingFunc` makes a model out of a `DATA` structure. Its implementation of the `decoding` function uses a recursive function that takes in a pdm and returns an answer that is relative to that pdm. It uses the matching function of the lexicon structure to immediately return `DecodingsNone` when there are no words in the lexicon that are consistent with the pdm. The function *ca* is called on each call of this function, and a "step" of its computation corresponds to a call of the function. `CodingFunc` is applied to our `Data` structure to form our standard model, `Coding`.

4.4 View/User-interface

The signature of our view/user-interface is listed in Fig. 5. As with the signature (`CODING`) of our model, it builds on the `DATA` signature. The `ui` type consists of whatever data the functions of the user-interface need in order to do their work. In the graphical user-interface described below, this type consists of several CML channels, which can only be allocated once CML is running.

The function `run` takes in the name of the program, its command line arguments, and a function f (which in, practice, comes from the main processor/controller). It processes the command line arguments, initializes the user-interface, producing user-interface data ui, and then calls f with ui. Once f returns, it finalizes the user-interface, and then returns a status of success. If the command line arguments are inappropriate, or if initializing the user-interface fails, then it outputs an error message and returns a status of failure, without first calling f.

The `primaryInput` function is used to get a primary command from the user. The argument to a lexicon command is a lexicon, not the file from which the lexicon was obtained. The `primaryOutput` function is used to tell the user about a response to a primary command. Only the encode and decode commands have responses.

The `secondaryInput` function is called with a pdm and its sets of old and new symbols; it displays this pdm to the user, and then gets a secondary command from the user. The arguments (a, b) to a replace command are required to be consistent with the pdm: a must be an old symbol of the pdm, and b must not be a new symbol of the pdm. The `secondaryOutput` function is used to tell the user about a response to a secondary command.

If the value of type `ui` needed to change over time, it could be returned by the functions of the user-interface.

The function `abortable` is called with the user interface data ui, an integer n and a function f. It uses ui to tell the user that an abortable computation is

```
signature USER_INTERFACE =
sig
  include DATA

  datatype primary_command =
             QuitPC | LexiconPC of sym_lexicon | EncodePC of msg
           | DecodePC of msg

  datatype primary_response =
             EncodeWordsNotInLexiconPR of word_set  (* responses from encode *)
           | EncodeMultipleDecodingsPR
           | EncodingPR of msg
           | DecodeNoDecodingsPR                    (* responses from decode *)
           | DecodeMultipleDecodingsPR

  datatype secondary_command =
             QuitSC | AbortSC | CheckSC | HintSC | ReplaceSC of sym * sym
           | UndoSC

  datatype secondary_response =
             CheckNotDecodableSR            (* responses from check *)
           | CheckDecodedSR
           | CheckDecodableButNotDecodedSR
           | HintNotDecodableSR             (* resonses from hint *)
           | HintDecodedSR
           | HintReplaceSR of sym * sym

  type ui

  val primaryInput    : ui -> primary_command
  val secondaryInput  : ui * pdm * sym_set * sym_set -> secondary_command
  val primaryOutput   : ui * primary_response -> unit
  val secondaryOutput : ui * secondary_response -> unit

  val abortable :
        ui * int * ((IntInf.int -> bool) -> IntInf.int * 'a option) ->
        'a option

  val run : string * string list * (ui -> unit) -> OS.Process.status
end
```

Figure 5: User-interface Signature

```
signature MAIN =
sig
  structure UserInterface : USER_INTERFACE
  structure Coding : CODING

  sharing type UserInterface.SymLinOrd.elem =
               Coding.SymLinOrd.elem
  sharing type UserInterface.pds = Coding.pds
  ...

  val main : string * string list -> OS.Process.status
end
```

Figure 6: Main Signature

being begun. It then calls f with a function *ca* (check abort) that f can use to communicate (using *ui*) to the user how many "steps" it has completed so far, as well as find out whether the user has asked for it to abort its computation (`true` means abort, `false` means don't abort). The integer n controls how often calls to *ca* actually communicate with the user: this happens every n calls; in all other calls, *ca* quickly returns `false`.

If f returns $(m, \texttt{NONE})$, meaning it aborted after completing m steps, then `abortable` returns `NONE`, after telling the user that the computation was aborted after m steps; if f returns $(m, \texttt{SOME}\, v)$, meaning that it terminated normally with value v after m steps, then abortable returns $\texttt{SOME}\, v$, after telling the user that the computation terminated normally after m steps.

Our controller/main processor calls the user-interface functions in a specific order. A given view/user-interface may make as much use of this order as it wishes. Views may be stateless or stateful; our terminal-based user-interface is stateless, whereas our graphical user-interface is stateful.

4.5 Controller/Main Processor

The signature `MAIN` given in Fig. 6 is the signature of our controller/main processor. It consists of a main processing function that uses the view/user-interface and model/coding structures to do its work. The sharing constraints require that the basic data types of the two structures are identical; otherwise, values couldn't be passed back and forth between the two structures.

A controller with signature `MAIN` is constructed by the functor `MainFunc` of Figs. 7 and 8. Note how the controller's algorithm is apparent. It is mostly functional, and doesn't have to be mentally stitched together out of a set of event handlers.

The sharing constraints in the functor's parameter list are necessary; otherwise, the functor fails to compile. The function `main` uses the `run` function of the supplied `UserInterface` structure to process the command line arguments, start up the user-interface, and then call the `primary` function with the user-interface data and an empty lexicon, entering the primary command loop. When `primary` returns to `run`, the user-interface will shut itself down, returning

```
functor MainFunc(structure UserInterface : USER_INTERFACE
                 structure Coding : CODING
                 sharing type UserInterface.SymLinOrd.elem =
                         Coding.SymLinOrd.elem
                 ...) :>
          MAIN =
struct
  structure UserInterface = UserInterface
  structure Coding = Coding
  structure UI = UserInterface
  structure Cg = Coding
  ...

  exception Abort
  exception Quit

  fun secondary(data as (ui, lex, msg, pdm, olds, news, msg')) =
        case UI.secondaryInput(ui, pdm, olds, news) of
             UI.QuitSC          => raise Quit
           | UI.AbortSC         => raise Abort
           | UI.CheckSC         => ...
           | UI.HintSC          =>
               (case Cg.findHint(msg, pdm, olds, msg') of
                     Cg.HintDecoded       =>
                       (UI.secondaryOutput(ui, UI.HintDecodedSR); raise Abort)
                   | Cg.HintNotDecodable  =>
                       (UI.secondaryOutput(ui, UI.HintNotDecodableSR);
                        secondary data)
                   | Cg.HintReplace(a, b) =>
                       let val pdm = Cg.replacePDM (a, b) pdm
                       in UI.secondaryOutput(ui, UI.HintReplaceSR(a, b));
                          secondary(ui,
                                    lex,
                                    msg,
                                    pdm,
                                    SymSet.minus(olds, SymSet.fromList[a]),
                                    SymSet.union(news, SymSet.fromList[b]),
                                    msg');
                          secondary data
                       end)
           | UI.ReplaceSC(a, b) => ...
           | UI.UndoSC          => ()
```

Figure 7: Main Functor, Part 1

```
  fun primary(ui, lex) =
        case UI.primaryInput ui of
             UI.LexiconPC lex => primary(ui, lex)
           | UI.QuitPC        => ()
           | UI.EncodePC msg  =>
               let val xs = Cg.unknownWordsMsg(lex, msg)
               in if WordSet.size xs = 0
                  then case UI.abortable
                            (ui, 50000, fn ca => Cg.decodings(ca, lex, msg)) of
                            NONE                       => primary(ui, lex)
                          | SOME Cg.DecodingsNone      =>
                            raise Fail "cannot happen"
                          | SOME(Cg.DecodingsUnique _) =>
                            let val msg' = Cg.encode msg
                            in UI.primaryOutput(ui, UI.EncodingPR msg');
                               primary(ui, lex)
                            end
                          | SOME Cg.DecodingsMultiple  =>
                            (UI.primaryOutput(ui,
                                              UI.EncodeMultipleDecodingsPR);
                             primary(ui, lex))
                  else (UI.primaryOutput(ui, UI.EncodeWordsNotInLexiconPR xs);
                        primary(ui, lex))
               end
           | UI.DecodePC msg  =>
               (case UI.abortable
                     (ui, 50000, fn ca => Cg.decodings(ca, lex, msg)) of
                     NONE                          => primary(ui, lex)
                   | SOME Cg.DecodingsNone         =>
                     (UI.primaryOutput(ui, UI.DecodeNoDecodingsPR);
                      primary(ui, lex))
                   | SOME(Cg.DecodingsUnique msg') =>
                     if (secondary(ui, lex, msg, Cg.msgToPDM msg,
                                   Cg.symsMsg msg, SymSet.fromList nil,
                                   msg');
                         true)
                          handle Quit  => false
                               | Abort => true
                     then primary(ui, lex)
                     else ()
                   | SOME Cg.DecodingsMultiple     =>
                     (UI.primaryOutput(ui, UI.DecodeMultipleDecodingsPR);
                      primary(ui, lex)))

  fun main(cmd, args) =
        UI.run(Aux.lastPartOfPath cmd,
               args,
               fn ui => primary(ui, SymLexicon.empty))
end
```

Figure 8: Main Functor, Part 2

a status of success back to `main`, which will return that status. (If something goes wrong when processing the command line arguments or starting the user-interface, then `run` will immediately return a status of failure, which will be returned by `main`.)

The function `primary` is tail-recursive and keeps track of the user-interface data and the current lexicon. In response to a `decode` command, it uses the `abortable` function of the user-interface to tell the user that an abortable computation is being begun. The `abortable` function then calls the `decodings` function of the `Coding` structure with:

- a function *ca* (check abort) that `decodings` uses to communicate to the user how many "steps" it has completed so far, as well as to find out whether the user has asked for it to abort its computation;
- the current lexicon;
- the message of interest.

Because of the second argument to `abortable`, the calls to *ca* only actually communicate with the user every 50,000 times.

If *decodings* returns $(m, \texttt{NONE})$, meaning it aborted after completing m steps, then `abortable` returns `NONE`, after telling the user that the computation was aborted after m steps. This causes `primary` to iterate. Alternatively, `decodings` returns a value with form $(m, \texttt{SOME}\, v)$, meaning that it terminated normally with value v after m steps, so that abortable returns $\texttt{SOME}\, v$, after telling the user that the computation terminated normally after m steps. If v is `DecodingsNone` or `DecodingsMultiple` of the `Coding` structure, then the user is informed, using the function `primaryOutput` of `UserInterface`, that an error occurred, before `primary` iterates.

Otherwise, v has the form $\texttt{Cg.DecodingsUnique}\, msg'$, meaning that msg' is the unique decoding of msg. In this case, we enter the secondary command loop by calling the function `secondary` with the user-interface data, the current lexicon, the message msg to be decoded, the result of turning this message into a pdm, the old symbols of this pdm (which is the same as the symbols of msg), the new symbols of this pdm (i.e., the empty set), and the unique decoding msg' of the message. If `secondary` returns normally or raises `Abort`, then `primary` will iterate. But if `secondary` raises `Quit`, then `primary` will return.

Some of the details of `secondary` have been elided. Each recursive call of this function is a tail-call, except for those carried out in response to the hint and replace (not shown) commands. This allows the undo command to be implemented by simply returning.

Finally, our controller/main processor is created by calling `MainFunc` with `UserInterface` and `Coding`, as in Fig. 9. We use the conditional compilation feature of the compilation manager to define the appropriate version of `UserInterface`.

```
structure Main =
            MainFunc(structure UserInterface = UserInterface
                     structure Coding = Coding);
```

Figure 9: Main Structure

```
signature INTERRUPTS =
sig
  val ignore : (unit -> 'a) -> 'a
  val track  : (unit -> 'a) -> 'a
  val check  : unit -> bool
end
```

Figure 10: Interrupts Signature

```
fun run(cmd, args, f) =
      (if null args
       then (print "at a prompt, type \"help\" for help\n";
             Interrupts.ignore f; OS.Process.success)
       else (print("usage: " ^ cmd ^ "\n"); OS.Process.failure))
```

Figure 11: Run Function of Terminal User-Interface

4.6 Terminal View/User-interface

The terminal user-interface is stateless, and consequently the user-interface data type `ui` is `unit`.

To handle interrupts, we use an `Interrupts` structure whose signature, `INTERRUPTS`, is given in Fig. 10. The function `ignore` is used to run its argument function while ignoring interrupts. The function `track` is used to run its argument function while keeping track of whether an interrupt has been signaled by the user. Finally, the function `check` is used to determine whether the user has signalled an interrupt so far.

The `run` function of the user-interface is given in Fig. 11. Note that nothing need be done to "start up" and "shut down" the user-interface.

The functions `primaryInput` and `secondaryInput` prompt the user for a command, and do the work of turning the user's input into the required form. In the case of the lexicon command, this involves turning the contents of a file into a lexicon. In the case of the encode and decode commands, this involves prompting the user to enter a message. If the user types an invalid command, or aborts the process of typing in a message, then the function simply returns to its beginning and tries again. In the case of a replace command, the user-interface is responsible for checking that the argument letters are legal ones; if they are not, it issues an error message, and jumps back to its beginning.

The functions `primaryOutput` and `secondaryOutput` simply output the responses they are given to the user.

The `abortable` function is given in Fig. 12. Note how a mutable variable is used to keep track of when the `checkAbort` function should actually communi-

```
local
  val delayRef = ref 0
in
  fun abortable(_, n, f) =
        let fun checkAbort m =
                  if !delayRef = 1
                  then (delayRef := n;
                        print("\rcompleted " ^ IntInf.toString m ^
                              " steps ...");
                        Interrupts.check())
                  else (delayRef := !delayRef - 1; false)

            val _ = delayRef := n
            val _ = print "computing ..."
        in case Interrupts.track(fn () => f checkAbort) of
                (m, NONE) =>
                  (print("\rinterrupted after completing " ^
                         IntInf.toString m ^ " steps\n");
                   NONE)
              | (m, x)    =>
                  (print("\rterminated after completing " ^
                         IntInf.toString m ^ " steps\n");
                   x)
        end
end
```

Figure 12: Abortable Function of Terminal User-Interface

```
type ui =
       {primaryCommandChan    : primary_command SV.ivar chan,
        secondaryCommandChan  :
          (pdm * sym_set * sym_set * secondary_command SV.ivar)chan,
        primaryResponseChan   : primary_response chan,
        secondaryResponseChan : secondary_response chan,
        abortableStartChan    : unit chan,
        abortableCheckChan    : (IntInf.int * bool SV.ivar)chan,
        abortableStopChan     : (IntInf.int * bool)chan}
```

Figure 13: User-interface Data for Graphical User-interface

cate with the user.

4.7 Graphical View/User-interface

The graphical user-interface is built using eXene and CML. In contrast to the terminal user-interface, the graphical user-interface does have state. The user-interface data type `ui` consists of a record of channels that allow the functions of the user-interface to communicate with the main thread of the GUI; see Fig. 13 for the details. In this figure, as elsewhere in the graphical user-interface, `SV` is an abbreviation for the `SyncVar` structure, and an `'a SV.ivar` is an incremental (write-once) variable (I-var) of type `'a`.

The `run` function of the user-interface:

- starts up CML,

```
fun loop() = loop'(recv(#primaryCommandChan ui))

and loop' iVar =
      (sync delFlushEvt;
       primSetActive[true, true, true, true];
       messageSet "select primary command";
       select
       [wrap(primLabEvt,
             fn "Quit"    => quitPC(iVar, guiData, loop)
              | "Lexicon" => lexiconPC(iVar, guiData, loop, loop')
              | "Encode"  => encodePC(ui, iVar, guiData, loop, loop')
              | "Decode"  => decodePC(ui, iVar, guiData, loop, loop')
              | _         => raise Fail "cannot happen"),
        wrap(delEvt, fn () => quitPC(iVar, guiData, loop))])
```

Figure 14: Primary Command Loop of Graphical User-interface

```
fun primaryInput({primaryCommandChan, ...} : ui) =
      let val ivar : primary_command SV.ivar = SV.iVar()
      in send(primaryCommandChan, ivar);
         SV.iGet ivar
      end
```

Figure 15: Primary Input Function of Graphical User-interface

- processes the command line arguments (which can be used to control, e.g., which X display will be opened),
- opens a connection to the X display,
- creates the channels of the user-interface data *ui*,
- spawns the main thread of the user-interface (others are embedded in widgets), giving this thread *ui*, and
- calls the function f it was given with *ui*.

Once f returns, `run` closes the connection to the X server, shuts down CML, and returns a status of success. If there are problems processing the command line arguments or opening the X display, then `run` returns a status of failure without calling f.

The user-interface thread creates and realizes the widgets of the GUI and then executes its primary command loop, which is listed in Fig. 14. The GUI thread begins by waiting for an I-var to be sent to it on the primary command channel. This will happen when the `primaryInput` function is called—see Fig. 15, and the GUI thread is responsible for filling the I-var with a primary command obtained from the user.

Once the I-var has been received, `loop'` clears any requests from the window manager for the program to exit, makes all of the primary command buttons active, prompts the user to click on one of them, and then waits for the user to click on one of them, or to ask via the window manager for the program to exit.

```
local
  val delayRef = ref 0
in
  fun abortable(ui : ui, n, f) =
        let fun checkAbort m =
                  if !delayRef = 1
                  then let val iVar = SV.iVar()
                       in delayRef := n;
                          send(#abortableCheckChan ui, (m, iVar));
                          SV.iGet iVar
                       end
                  else (delayRef := !delayRef - 1; false)

            val _ = delayRef := n
            val _ = send(#abortableStartChan ui, ())
        in case f checkAbort of
                 (m, NONE) =>
                   (send(#abortableStopChan ui, (m, false)); NONE)
               | (m, x)    =>
                   (send(#abortableStopChan ui, (m, true)); x)
        end
end
```

Figure 16: Abortable Function of Graphical User-Interface

When a command is received, the corresponding function is called, and is passed among other things both `loop` and `loop'`, so that the function can return to `loop` or `loop'`, depending upon whether it is able to obtain a primary command from the user. The secondary command loop, which will be entered via `decodePC`, works similarly.

Once a command's function delivers an obtained command in an I-var, it has enough context to sensibly respond to what happens next. E.g., the `encodePC` function could receive a response on the primary response channel listing the words of the previously supplied message that are not in the lexicon; it can then tell the user what those words are, as well as remind the user what the message is. This is an example of why it is useful for the GUI to have state.

When all of the words of a message to be encoded are in the lexicon, the `encodePC` function is told via the abortable start channel that the process of verifying that this message has a unique decoding has begun. The message originated from a call to the `abortable` function, which is listed in Fig. 16. The `encodePC` function responds to the message on the abortable start channel by calling `abortableServer`, which is listed in Fig. 17.

The `abortableServer` function begins by telling the user that an abortable computation has begun. It then enables the Cancel auxiliary button, and calls the `abortSer` function with `false`. The `abortSer` function keeps track of whether the user has already asked for the computation to be aborted. At each iteration, it can:

- Respond to the Cancel auxiliary button being clicked on, which causes it to record that the user has asked for the computation to be aborted.

```
fun abortableServer(ui : ui, {auxLabEvt, auxSetActive, messageSet,
                  textSet, ...} : gui_data) =
      let fun abortSer aborted =
                select
                [wrap(auxLabEvt, fn _ => abortSer true),
                 wrap(recvEvt(#abortableCheckChan ui),
                      fn (m, iVar) =>
                           (SV.iPut(iVar, aborted);
                            messageSet("completed " ^ IntInf.toString m ^
                                       " steps ...");
                            abortSer aborted)),
                 wrap(recvEvt(#abortableStopChan ui), fn x => x)]

          val _      = messageSet "computing ..."
          val _      = auxSetActive[false, true]
          val (m, b) = abortSer false
          val _      = auxSetActive[false, false]
      in if b
         then (messageSet("terminated after completing " ^
                          IntInf.toString m ^ " steps");
               sleep 2000; messageSet "")
         else (messageSet("interrupted after completing " ^
                          IntInf.toString m ^ " steps");
               sleep 2000; messageSet ""; textSet "");
         b
      end
```

Figure 17: Abortable Server of Graphical User-interface

- Receive on the abortable check channel notification that m steps of the computation have been completed, along with an I-var that it must fill in with a boolean indication of whether the user has asked that the computation be aborted. This message originated with a call to the `checkAbort` function of the `abortable` function. The user is informed by `abortSer` that m steps of the computation have been completed.

- Receive a notification (m, b) on the abortable stop channel, where m is the number of steps completed when the abortable computation terminated, and the boolean b is `true` if the termination was normal, and `false` if it was an abortion. The notification originated in the `abortable` function. The pair (m, b) is then returned by `abortSer`.

Once the `abortSer` returns its result (m, b) back to `abortableServer`, that function disables the Cancel button, lets the user know what happened, and returns b.

Back in `encodePC`, if b is `false` (the computation was aborted), then a jump is made back to the function `loop` of the primary command loop, so that the GUI thread will await a request for another primary command. Otherwise, it awaits a primary response, which will either tell it that the message to be encoded had multiple decodings, or what random encoding was chosen.

5 On the General Approach

As illustrated in our case study, the cornerstone of our approach is a controller that is truly in control. It mediates between the view/user-interface and model, sometimes asking the view to get input from the user, sometimes asking the view to display results to the user, and sometimes asking the model to carry out domain-specific work. What would be the state of a model in an object-oriented setting is the arguments to the functions of the controller in our setting.

That controllers of this sort can be written for applications with terminal user-interfaces is unsurprising. But before working through our case study, the reader may have thought that applications with graphical user-interfaces would have to have controllers that were reactive, not active, or that a single controller couldn't work with both terminal and graphical views. Hopefully, the case study has at least partly dispelled such worries.

On the other hand, the case study was only concerned with a few ways of communicating with the user, and the reader might wonder which communication patterns are consistent with our architecture. It seems unclear how to answer this question in general, so we will content ourselves with sketching how one more communication pattern can be accommodated.

Suppose that an application's primary command loop has a set command for setting various parameters controlling the behavior of the application. In a GUI, the user might click on a Set button, and then be given a form whose parameter fields have current values that may be overridden. After resetting the values of selected fields, the user would have the option of clicking on OK or Cancel buttons, to either confirm the changes or abort making the changes.

How would a terminal user-interface for such an application be structured? One possibility is for the set command to take the user to a secondary loop in which the values of parameters may be queried and set, using secondary commands. There would also be secondary commands for confirming the changes or choosing to abort the process of making the changes, returning, in either case, to the primary command loop.

The user-interface signature for such an application would have a new primary command constructor, `SetPC`, whose argument would be a record of parameter values. The controller's primary command loop would have to keep track of the current value of this record, and the function, `primaryInput`, for getting a new primary command would take this record as an argument, so that the user-interface would know the current values of the parameters, for the case in which the user elects to run the `set` command. In both the graphical and terminal user-interfaces, aborting the `set` command is internal to the user-interface, and doesn't result in a primary command being returned yet to the controller.

In order to get a better idea of the generality of our approach, more case studies should be carried out. For example, it would be instructive to try refactoring the implementation of Unison [8], which comes with both terminal and graphical user-interfaces, so as to use our architecture.

6 Acknowledgments

It is a pleasure to acknowledge helpful discussions with Brian Howard, Benjamin Pierce, Dave Schmidt and the students in my graduate programming languages course. Thanks are due to Dustin deBoer, Dominic Gélinas and Cole Hoosier for contributions to eXene that made it possible to improve the functionality and appearance of Crypto's GUI. Referees' comments on an earlier version of this paper were also very helpful.

References

[1] Gansner, E.R., Reppy, J.H., eds.: The Standard ML Basis Library. Cambridge University Press (2002)

[2] Gansner, E.R., Reppy, J.H.: A multi-threaded higher-order user interface toolkit. In Bass, Dewan, eds.: User Interface Software. Volume 1 of Software Trends. Wiley (1993)

[3] Lüth, C., Westmeier, S., Wolff, B.: sml_tk: Functional programming for graphical user interfaces. Technical Report 8/96, FB 3, Universität Bremen (1996)

[4] Krasner, G., Pope, S.: A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. Journal of Object Oriented Programming **1**(3) (1988) 26–49

[5] Apple Computer, Inc.: The Model-View-Controller Design Pattern. (2006) Apple Developer Connection Reference Library.

[6] Appel, A., Blume, M., Gansner, E., George, L., Huelsbergen, L., MacQueen, D., Reppy, J., Shao, Z.: Standard ML of New Jersey. `www.smlnj.org` (2006)

[7] Lüth, C., Wolff, B.: Functional design and implementation of graphical user interfaces for theorem provers. Journal of Functional Programming **9**(2) (March 1999) 167–189

[8] Pierce, B., Balasubramaniam, S., Vouillon, J.: The unison file synchronizer. `http://www.cis.upenn.edu/~bcpierce/unison/` (2004) Version 2.13.15.

[9] Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press (1999)