A Functional Model-View-Controller Software Architecture for Command-oriented Programs

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

Abstract
Command-oriented functional programs are currently structured in an ad hoc way that makes the development of multiple user-interfaces difficult and error prone, and makes it difficult to abstractly understand a program’s command-oriented behavior. To rectify this, we propose a software architecture for such programs that we call functional model-view-controller (MVC), by a rough analogy with object-oriented MVC. In functional MVC, a program is structured as a model (domain-specific aspects), view (abstract user) and controller (command loops). In contrast to object-oriented MVC, a controller is active, consisting of a number of recursive functions. It 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 should be parameterized by its model and view, using a function or an ML-style functor. With this approach, one can write terminal and graphical views; one can also write views that do abstract scripting. One can understand and reason about a program’s command-oriented aspects at a high-level of abstraction by focusing on the controller. Of particular note is the way we are able to allow computations of the model to be monitored and aborted by view. We illustrate our approach with an case study of a complete program, written in Standard ML, and using Concurrent ML and the eXene X window system toolkit.

Categories and Subject Descriptors
D [J]; 1; D [2]; 11

General Terms
Design, Languages

Keywords
model-view-controller, user-interface, ML, Concurrent ML, eXene

1. Introduction
1.1 Command-oriented Programs
We are concerned with command-oriented functional programs, i.e., programs that a user interacts with by issuing commands. In a program with a terminal user-interface, this will be done by typing commands as sequences of letters, whereas in programs with graphical user-interfaces (GUIs), this may be done by clicking on buttons and/or typing in text fields. Such programs may have various kinds of command loops (primary, secondary, etc.). Typically, it will be possible to monitor and abort long-running computations.

In addition, we are mainly concerned with programs that could have both terminal and graphical user-interfaces. It sometimes takes creativity to see how one could recast an element of a terminal user-interface as one of a GUI, or vice versa. For example, consider a program with a set command for setting various parameters controlling the behavior of the program. In a GUI, the user might click on a Set button, and then be given a form whose parameter fields have default 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 program be structured? One possibility is for the set command to take the user to a subsidiary loop (not existing in the GUI version) in which the values of parameters may be queried and set, using subsidiary commands. There would also be subsidiary commands for confirming the changes or choosing to abort the process of making the changes, returning, in either case, to the parent loop.

Although most of our examples, as well as our case study, are in ML, the paper’s approach and results are applicable to functional programming languages in general. A number of command-oriented ML programs exist and are in use. Examples include:

- Many ML compilers. E.g., the Standard ML of New Jersey (SML/NJ) compiler (Appel et al., 2007) is a Standard ML (SML) program whose interactive front-end has a terminal user-interface.
- Some theorem proving environments. E.g., the Twelf theorem proving environment (Pfenning and Schürmann, 1999) is a Standard ML program with a terminal user-interface.
- The Concurrency Workbench of the New Century (CWB-NC, 2000) is a Standard ML program with both terminal and graphical user-interfaces.
- The Unison file synchronizer (Pierce et al., 2004) is an Objective Caml (OCaml) program with both terminal and graphical user-interfaces.

Terminal user-interfaces can be built using the facilities of the Standard ML Basis Library (Gansner and Reppy, 2002) or the OCaml library (Leroy, 2005). They are easy to build, and are often favored by expert users. Graphical user-interfaces (GUIs) can be built using a GUI toolkit, and are often favored by novice users. The Tcl/Tk GUI toolkit (Ousterhout, 1994) has Standard ML (smlTk (Lith et al., 1996)) and OCaml bindings (labltk (Leroy, 2005)) bindings, and the GTK+ GUI toolkit (GTK, 2007) has Standard ML (mGTK (Larsen and Niss, 2004)) bindings and OCaml bindings (LablGTK (Garrigue, 2006)) bindings. One can also build GUIs for Standard ML programs using eXene (Gansner and Reppy, 1993), a multi-threaded, higher-order user-interface toolkit for the X window system. eXene is built upon Concurrent ML (CML) (Reppy, 1999), which is an SML/NJ library.
Most functional languages have Tcl/Tk and/or GTK+ libraries, and some have other GUI libraries as well.

1.2 Architecture of Command-oriented Programs

Command-oriented functional programs are currently structured in an ad hoc way that makes the development of multiple user-interfaces difficult and error prone, and makes it difficult to abstractly understand a program’s command-oriented behavior. Most commonly, command-oriented functional programs come with only one kind of user-interface. And the few programs with both terminal and graphical user-interfaces are built in one of the following ways, both of which suffer from serious drawbacks.

• In the first approach, scripting is used to turn one kind of user-interface into another one. Typically, string-based scripting is used to turn a terminal-user interface into a graphical user-interface. E.g., this was the approach used in the Concurrency Workbench of the New Century (CWBN-C, 2000), which is turned into a GUI using Expect (Libes, 1991) and Tcl/Tk. But with some GUI toolkits, it is possible to do graphical-based scripting (simulating keyboard and mouse actions, responding to graphical events), to turn a GUI into a terminal user-interface. Perhaps this has been done with functional programs.

Unfortunately, doing scripting either in terms of strings or graphical actions and events is a low-level, error-prone business, typically done in a non-functional language without the benefit of static type-checking.

• In the second approach, terminal and graphical user-interfaces are written separately, only sharing the code for doing domain-specific work. E.g., this was the approach used in the implementation of Unison (Pierce et al., 2004), whose GUI is built using LablGTK (Garrigue, 2006).

Terminal user-interfaces typically consist of a collection of mostly tail-recursive functions, which prompt the user for input, obtain that input, pass it to the domain-specific code for processing, and then communicate the results of those calls back to the user. In contrast, graphical user-interfaces are typically constructed by combining a widget hierarchy with a set of event handlers. Consequently, terminal user-interfaces can be seen as “active”, whereas graphical user-interfaces can be seen as “reactive”.

Writing a second user-interface from scratch is an unappealing prospect. And this approach makes it hard to maintain consistency between a program’s terminal and graphical user-interfaces. Furthermore, it isn’t possible to analyze or reason about a program’s command-oriented aspects in an abstract way, because these aspects are implemented in very different ways in different user-interfaces.

Ideally, we would like to divide user-interfaces into two parts:

• the part that interacts with the user, or, more generally, is an abstraction of a user; and

• a collection of command loops, consisting of mostly tail-recursive functions, which interact with the abstract users.

Such a division would allow one to abstractly understand the command-oriented aspects of a program by focusing on the command loops. And, hopefully, the same command loops would be compatible with multiple abstract users.

Because graphical user-interfaces are typically reactive, not active, it may seem that this can’t be done for GUIs. But it turns out that graphical user-interfaces can be written in an active style, if one makes use of concurrency: the command loops can run in one thread, and the graphical toolkit can run in another thread or threads. When a command loop needs a command to execute, it can send a message to the main graphical thread asking for one. The graphical thread can activate certain buttons or fields, and prompt the user to select a command. Once the user has done this, the main graphical thread can send the resulting command back to the command loops thread. A similar approach can be used for communicating command results to the user.

With a multi-threaded graphical toolkit like eXene (Gansner and Reppy, 1993), which is built using CML (Reppy, 1999), doing this is relatively straightforward. Channels can be used to pass messages between the command loops thread and the main graphical thread (others are inside widgets), and selective communication can be used to allow the main graphical thread to be able to respond to one of a number of events, some consisting of receiving messages from the command loops thread, and others consisting of receiving data from graphical widgets. Typically, the main graphical thread consists of tail-recursive functions. Although not purely functional, such an approach feels functional. EXene and CML are very well suited to this kind of programming.

Furthermore, in an event loop-based toolkit such as Tcl/Tk (Ousterhout, 1994) or GTK+ (GTK, 2007), writing a main graphical thread that is compatible with command loops should also be possible. After initialization, this main graphical thread will handle the event loop. With the LablGTK OCaml binding (Garrigue, 2006) for GTK+, which provides support for OCaml’s threads, a command loop can generate special events for the event loop to process, and event handlers can use mailboxes (message queues) for communicating with the command loops thread. Because of the use of event handlers, which will need to consult and update mutable variables, such an approach won’t be especially functional. It is unclear whether it would be possible to use OCaml threads together with the OCaml binding for Tcl/Tk (labltk (Leroy, 2005)), although an undocumented threads module suggests the answer might be “yes”, or to use Concurrent ML’s (Reppy, 1999) threads in conjunction with the Standard ML bindings for Tcl/Tk (smlTk (Lüth et al., 1996)) and GTK+(mGTK (Larsen and Niss, 2004)). An event-loop-based approach should be possible with some other functional languages and graphical toolkits.

Borrowing terminology from object-oriented programming (Krasner and Pope, 1988; Apple, 2006), we will call a collection of command loops a controller, and call an abstract user a view. If, as usual, we call a program’s domain-specific aspects a model, then we are led to formulating a software architecture for command-oriented programs that we can call functional model-view-controller (MVC). In contrast to object-oriented MVC (Krasner and Pope, 1988; Apple, 2006), a controller is the locus of control, consisting of a number of recursive functions. 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. Also in contrast to the object-oriented case, views are much more abstract and self-contained. To increase adaptability, we will argue that a controller should be parameterized by its model and view, using a function or ML-style functor. In addition to terminal and graphical views, views for doing abstract-scripting can be written. Of particular note is the way we are able to allow computations of the model to be monitored and aborted by view. Although our architecture is not purely functional—communicating with users involves side-effects, and the widgets of graphical views have state—it is mostly functional.

1.3 Outline of Paper

In the following sections we:

• give a high-level description of the functional MVC architecture (Section 2);
• illustrate our approach with a case study of a complete program, written in Standard ML, and using Concurrent ML and the eXene X window system toolkit (Section 3);
• consider related work (Section 4); and
• summarize the paper's achievements and look ahead to future work (Section 5).

2. Functional MVC

2.1 Non-Parameterized Controllers

Let's start by supposing that the model, view and controller of a program live in the Standard ML (we use SML throughout) structures (i.e., modules) Model, View and Controller with signatures (i.e., interfaces) Model, View and Controller, respectively.

For example, the Model signature could include

```ml
val valid : params -> bool
val update : args * params -> args
```

the View signature could include

```ml
datatype primary_command =
  SetPC of params
| ...
```

```ml
datatype primary_response =
  ValidParamsPR
| InvalidParamsPR
| ...
```

```ml
val primaryInput : unit -> primary_command
val primaryOutput : primary_response -> unit
```

and part of the primary command loop of a controller might look like

```ml
fun primary args =
  case View.primaryInput() of
    View.SetPC params =>
      if Model.valid params
        then (View.primaryOutput
             View.ValidParamsPR;
                primary(Model.update
                           (args, params)))
        else (View.primaryOutput
               View.InvalidParamsPR;
                primary args)
      | ...
```

The arguments of the function `primary` can be whatever data the primary command loop needs to keep track of what it's doing. The function `primaryInput` of the View structure prompts the user to enter a command, and then gets a primary command from the user. This may involve repeated interactions with the user, e.g., if the user types an invalid command, or aborts from a subsidiary loop, as in the set command from Section I. The eventually returned primary command is then matched against the patterns of the case. If the command is `SetPC` (the set primary command), then the model is consulted to determine if the parameters chosen by the user are valid. If they are, then the view is given a primary response saying that the command succeeded, and then primary iterates, with arguments that take into account the supplied parameters. Otherwise, the view is given a primary response saying that the command failed, which the view passes on to the user in some way, and primary iterates with unchanged arguments.

The recursive calls of `primary` shown above are both tail calls, but it is also possible to use non-tail calls to allow commands to be undone, as shown in our case study. Some primary commands may take control to subsidiary command loops, as necessary. In graphical views, different command loops can be realized in different parts of the same window, or in different windows.

When a program is structured in this way, one can understand and reason about the program's command-oriented aspects at a high-level of abstraction by focusing on the controller. To keep the model and controller as abstract as possible, views should be responsible for putting user-supplied data into internal form.

2.2 Parameterized Controllers

To increase adaptability, a controller should be a function or ML functor (i.e., parameterized module), parameterized by its view and model. Let's start by seeing it as a functor, and then consider the alternative of using a function. Here are skeletons for the controller signature and functor:

```ml
signature CONTROLLER =
sig
  structure Model : MODEL
  structure View : VIEW
  sharing type Model.t = View.t
...
(* main's specification is in terms of Model and View *)
val main : string * string list -> OS.Process.status
functor ControllerFunc(structure Model : MODEL
  structure View : VIEW
  sharing type Model.t = View.t
  ...) :> CONTROLLER = struct ...
end
```

Here, `main` is the entry point to the program: it takes in the name of the program and its command-line arguments, and uses `Model` and `View` when doing its work; eventually, it returns an exit status back to the operating system.

Different views can then be written and supplied as arguments to the same controller. 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 SML/NJ's compilation manager (Appel et al., 2007).

It is important that a controller be a functor, instead of being a structure that uses a particular model and view. This way the controller functor may be typechecked independently from a model and view. In fact, it may be typechecked before a model and view are written. Of course, it is necessary that the `MODEL` and `VIEW` signatures be written and specified before the controller functor is written, or at least concurrently with its writing. Furthermore, the functor's signature makes clear the sharing constraints that relate the types of the model and view. For example, in our case study, the model and view must implement a type of symbols identically.

To the model and controller, this type is abstract, but to the existing views, it is implemented by the lowercase letters.

One can certainly build controllers that are ordinary functions, taking in models and views that are records. Then instead of model and view signatures, we would have model and view record types. But, models and views often provide datatypes, which would have to be declared before a controller function could be typechecked. Furthermore, such datatypes may involve abstract types (e.g., the type of symbols in our case study), and these abstract types would have to be partly implemented before typechecking could be done.

By making use of functors, we can avoid these problems: such types only need to be specified (not declared) in the `MODEL` and `VIEW` signatures, in order for typechecking to be possible.

A controller calls the functions of its view in a particular order, with particular values. A given view is free to make as much or as little use of this knowledge as is convenient. For example, a view might make use of the fact that there will never be a response to certain commands, or that instead of being given a response...
to a primary command, the controller might request a secondary command. Thus, although the VIEW signature must be written before (or concurrently with) the controller, an implementation of VIEW will often need to be written after the controller is written.

2.3 Model and View Data

One of the reasons that it is possible to write a single controller that works with different kinds of views is that the VIEW signature may contain a type vd of view data. This data can be anything that a view needs to do its work and maintain state, and is passed, in a single-threaded way, back and forth between the controller’s functions and the functions of the view. Furthermore, something similar can be done with the model, giving it a type md of model data. Thus the MODEL signature could include

```ocaml
type md
val init : unit -> md
val valid : md * params -> md * bool
val update : md * args * params -> md * args
```

and the VIEW signature could include

```ocaml
type vd
val primaryInput : vd -> vd * primary_command
val primaryOutput : vd * primary_response -> vd
val run : string * string list * (vd -> vd) -> OS.Process.status
```

Now (some of) the functions of the model and view take in the current model/view data and return possibly new model/view data. The function init of the MODEL signature is used to create initial model data. The function run of the VIEW signature takes in the name of the program, any command-line arguments not consumed by the controller, and a function f. If the command-line arguments are invalid, or if it can’t start the view up, it returns a failure exit status. Otherwise it starts up the view, creating the initial view data, and, in the graphical case, a main view thread, and calls f (which will come from the controller, and typically will start the controller’s primary command loop) with that view data. When f returns with new view data, run shuts the view down, and returns an exit status. Thus, the controller functor could include code like

```ocaml
fun primary(vd, md, args) =
  case View.primaryInput vd of
    (vd, View.SetPC params) =>
      case Model.valid(md, params) of
        (md, true) =>
          let val (md, args) =
            Model.update(md, args, params)
          in
            primary(View.primaryOutput
              (vd, View.ValidParamsPR), md, args)
          end
        | (md, false) =>
          primary(View.primaryOutput
            (vd, View.InvalidParamsPR), md, args)
        | ...
  end

fun main(cmd, args) =
  View.run(cmd, args, fn vd =>
    primary(vd, Model.init(), ...))
```

Sometimes, it is useful for an input function to be passed additional information that the view can display to help the user choose a command.

In a stateless, terminal view, like the one of our case study, the view data type can simply be unit, whereas in a graphical view with a main thread, like the one of our case study, the view data can be a constant record of channels for communicating with that thread. It is also possible to make use of changing view data. For example, in our case study, we have an automatic (batch) view that uses the view data to decide what command to issue next when the controller asks for one, and to decide how to respond to responses from the controller. This is an example of how alternative views can carry out abstract scripting.

2.4 Monitoring and Aborting Long-running Computations

Sometimes a model will initiate long-running computations, which the user, via the view, should be allowed to monitor and abort. This can be achieved using higher-order, polymorphic functions. For example, the MODEL signature might have a function

```ocaml
val computation : 'a * ('a -> 'a * bool) * md * input ->
  'a * md * output option
```

where input is the type of normal inputs to the function, and output is the type of normal outputs. The first two arguments, ab ("abortable computation data") and ca ("check abort" function), are intended to be used together, in a single-threaded way, to allow the computation to be aborted. The value ab consists of initial abortable computation data, and the function ca takes in the current abortable computation data, and returns the next version of the abortable computation data, plus a boolean, which is true iff abortion of the computation is being requested. The convention is that computation calls ca with the current abortable computation data at the beginning of each of its "steps". It aborts by returning (ab, md, NONE), where ab is the current abortable computation data and md is new model data. It returns normally by returning a value of the form (ab, md, SOME vd). The ca function should be fast, as it will be called many times.

Note that computation is a polymorphic, higher-order function. A model has no idea what the type of abortable computation data might be, or how the check abort function will work.

Then, the VIEW signature could have the following type and value:

```ocaml
閘 abortable =
  vd * int *
  (abort * (abort -> abort * bool) ->
    abort * 'a * 'b option) ->
  vd * 'a * 'b option
```

The function abortable is called with a triple (vd, n, f), where vd is the current view data and n ≥ 1. It uses vd to tell the user that an abortable computation is being begun, and then calls f with initial abortable computation data, consisting of vd plus the indication that 0 computation steps have been completed, and a function ca of type abort -> abort * bool that f calls with the current abortable computation data whenever it begins a step of its computation. The function f shouldn’t communicate with the view or do any input/output operations directly, it should terminate normally, and it should use the abortable computation data in a single-threaded manner.

When the number of computation steps recorded in the abortable computation data ab passed to ca is non-zero and divisible by n, then ca tells the user that this number of steps of the computation have been completed, and finds out whether the user wants to abort the computation; if the user does wish to abort, then it returns true, along with abortable computation data consisting of possibly new view data, as well as the number of steps in ab (no more compu-
A view has no idea what the type variables instantiated with.

in which the controller starts the model’s computation, but lets it potentially aborted by the view.

If \( f \) returns \((ab, u, \text{NONE})\), meaning it aborted with computation data \(ab\) and secondary value \(u\) but no primary value, then abortable returns a triple consisting of the view data of \(ab, u\) and \text{NONE}, after telling the user that the computation was aborted after completing the number of steps recorded in \(ab\). If \( f \) returns \((ab, u, \text{SOME} v)\), meaning that it terminated normally with abortable computation data \(ab\), secondary value \(u\) and primary value \(v\), then abortable returns the triple consisting of the view data of \(ab, u, v\), after telling the user that the computation terminated normally after the number of steps recorded in \(ab\).

Note that abortable is a higher-order, polymorphic function. A view has no idea what the type variables \('a\) and \('b\) may be instantiated with.

Thus, the controller could contain code of the form

```
case View.abortable
  (vd, 50000,
   f\n    (ab, ca) =>
    Model.computation(ab, ca, md, input)) of
  (vd, md, NONE) =>
    (* abortion occurred *) ...
  (vd, md, SOME v) =>
    (* v is the computation's output *) ...
```

in which the controller starts the model’s computation, but lets it be periodically (every 50,000 steps, in this case) monitored and potentially aborted by the view.

It is possible to implement abortable in both terminal and graphical views. In a terminal view, the user could be allowed to abort a computation by typing the interrupt character. In a graphical view, the user could be given a Cancel button by which to ask for abortion of the computation.

2.5 Guidelines

Here are our guidelines for functional MVC:

- The model, view and controller should be as functional as possible; mutable variables should not be used, except hidden within abstractions.

- The controller should be a functor, parameterized by structures with signature past and present and view. These signatures should be written and specified before, or concurrently with, the controller functor. The functor may need to specify sharing constraints relating the types of the model and view.

- The view signature should contain datatypes of different kinds of commands and command responses, along with functions for getting commands from the user, and displaying responses to the user. A function for getting some kind of command can pass supporting information to the view, which the view can display to the user to help him/her choose a command.

- Views are responsible for putting user-supplied data into internal form, before passing it to the controller. Multiple interactions with the user will often be needed to obtain a command: prompting the user for a command, activating/deactivating buttons, doing error checking, etc.

- The controller should consist of a set of command loop functions, and any data the controller needs to keep track of what it is doing should be passed as arguments to these functions. Non-tail calls can be used to allow commands to be undone.

- The model signature should include a type md of model data that is passed between the functions of the controller and (some of the functions of the) model, in a single threaded manner, and that allows a model to do its work and maintain state. In our case study, the model data is a random number seed. The model signature should also include a function init for creating initial model data.

- The view signature should contain a type vd of view data that is passed between the functions of the controller and view, in a single threaded manner, and that allows a view to do its work and maintain state. In the graphical case, the view data will include channels by which the functions of the view may communicate with the view’s main thread.

- The view signature should contain a run function for starting up the view, creating initial view data, and, in the graphical case, a main view thread, passing this view data to one of the controller’s functions, and shutting down the view when this function returns.

- Higher-order, polymorphic functions can be used to allow long-running computations in the model to be monitored and conditionally aborted by the view. Computation functions of the model use “check abort” functions, and the view implements abortable functions that supply the “check abort” functions.

- A view may make use of knowledge about the algorithm of the controller. Consequently, some views can only be written after the controller functor.

- Abstract scripting views can be written, i.e., non-traditional views that use the view data to automate certain interactions with the controller.

3. Case Study

This section consists of a case study of a program structured using functional MVC. The program is called crypto, and allows users to encode and decode simple cryptograms. Crypto is complex enough to raise the main issues involved in command-oriented programs, but is simple enough to be suitable as an example for research and teaching.

Crypto is implemented in Standard ML, and makes use of some special features of SML/NJ (Appel et al., 2007), including its signal handling facilities, first-class continuations, CML (Reppy, 1999) and eXen (Gansner and Reppy, 1993). It runs on Linux/Unix/Mac OS X. There are three view structures provided: a terminal-based one, an X window system-based one, and an automatic (batch-oriented) one. The carefully commented code and further documentation for crypto are available for downloading and browsing at http://people.cis.ksu.edu/~stough/crypto/.

3.1 Behavior of Crypto

Crypto makes use of a lexicon, 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 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, and msg’ can be formed by applying some renaming to msg.

First, we consider the behavior of the terminal and graphical versions of crypto. Crypto has primary and secondary command
loops. In the terminal-based version of crypto, a command’s arguments are typed after the command. In the graphical 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 quitting (quit), loading a lexicon from a file (lexicon), generating a random encoding of a message (encode), and interactively decoding a message (decode). In the graphical version, the user is allowed to edit faulty input (a filename in the case of lexicon; a message in the case of encode and decode), and try again. The encoding process first checks that the supplied message is the unique decoding of itself. (If each of its words is in the lexicon, then it will be a decoding of itself, but there could be others.) This checking can take a long time in the worst case, and the user is informed of its progress and allowed to abort it (by interrupting in the terminal version, and clicking on the Cancel button in the graphical version).

The decoding process begins with an abortable check that the supplied message, msg, 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. We also refer to lower- and uppercase letters as “old” and “new”, respectively.

There are secondary commands for exiting the program (quit), returning to the primary command loop (abort), checking whether the current pdm is decodable (check), i.e., whether the renamings performed so far are correct, 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 randomly broken) with the uppercase version of its decoding. The check and hint commands return to the primary command loop if the current pdm is decoded, and undo returns to the primary command loop if there are no replacements (by either replace or hint) to undo.

Figures 1 and 2 contain execution snapshots of similar stages of the executions of the terminal and graphical versions of crypto. Crypto is in its secondary command loop. The second-to-last pdm of the terminal version and the pdm of the graphical version are identical. In the last pdm of the terminal version, the user has just replaced d by b. In the graphical version, the first and second rows of buttons correspond to the commands of the primary and secondary command loops, respectively. Only certain buttons are active. In the snapshot, the user has already elected to replace the letter d 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.

The automatic user-interface version of crypto is invoked with two arguments: a lexicon filename and a mode, which may be encode or decode. At any point, the user may terminate crypto by interrupting. Crypto begins by loading its lexicon from the lexicon file.

If the mode is encode, then crypto reads a message from the standard input, complaining and exiting if the message contains any non-words. Otherwise, crypto checks whether the inputted message is the only decoding of itself, periodically informing the user of the status of this checking process. If the answer is “no”, then an error message is issued, and crypto exits. Otherwise, a random renaming is generated and then applied to the message, the resulting encoded message is printed on the standard output, and crypto exits.

If the mode is decode, then crypto reads a message from the standard input, as with encode, and then checks whether the inputted message has a unique decoding, saving this unique decoding if the answer is “yes”. The user is periodically informed of the status of this checking process. If the inputted message doesn’t have a unique decoding, then an error message is issued, and crypto exits. Otherwise, the unique decoding is printed on the standard output, and crypto exits.

3.2 Implementation of Crypto
3.2.1 Basic Data

Although all versions of crypto work with words made up of lowercase letters, everything but the existing views is written so as to allow other choices of basic symbols. There is a standard linear ordering signature, LIN_ORD, whose type is called elem, along with LEXICON (main type lexicon) and SET (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 define a signature DATA whose structures define the basic data used by crypto; see Figure 3. The value symbols consists of the elements of elem that may actually be used, e.g., in words. A symbol of a partially decoded message is tagged with 0ld when it has not yet been replaced, and is tagged with new when it is the result of a replacement. The functor DataFunce takes in a linear ordering of symbols and a list of allowed symbols, and forms a structure with signature DATA. This functor is applied to a linear ordering based on characters, plus the lowercase letters, to form the Data structure.

3.2.2 Model

Figure 4 contains the MODEL signature, which builds on the signature DATA. It contains a type md of model data, and a function init for creating initial model data. Some of the functions of the model take in model data and return possibly new model data (this could be done uniformly, but I chose to only do it when needed for the existing model). There are 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, turning all instances of an old symbol to a new one;
- running an abortable computation to get information about the decodings of a message relative to a lexicon;
- getting a hint about a pdm and what replacement should be done next, given the message from which it was formed, the old symbols of the pdm, and the decoding of the message;
- returning the words of a message that aren’t in a lexicon; and
- generating a random encoding of a message.

The functor ModelFunce makes a model out of a DATA structure. It defines md to be a type of random number seeds. Its implementation of the Decodings 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 SymLexicon structure to immediately return DecodingsNone when there is a partially decoded word of the pdm that isn’t consistent with any words of the lexicon. The check abort function is called on each call of this function. The model data (random seed) is used by Decodings when choosing which maximally occurring old symbol to replace next, as well as
Figure 1. Terminal Version Snapshot

Figure 2. Graphical Version Snapshot

Figure 3. DATA Signature
what to replace it with. The model data is also used by findHint to break ties when there are multiple maximally occurring old symbols of a pdm, as well as by encode to generate random encodings. ModelFunc is applied to our Data structure to form our standard model, Model.

3.2.3 View

The VIEW signature is listed in Figure 5. As with the MODEL signature, it builds on the DATA signature. Because crypto has both primary and secondary loops, there are input and output functions, as well as associated datatypes, for both loops. The secondary input 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.

3.2.4 Controller

The CONTROLLER signature is given by Figure 6. Model and View should be built from the same DATA structure. Hence the sharing constraints will be satisfied, allowing values to be passed back and forth between the two structures.

A controller with signature CONTROLLER is constructed by the functor ControllerFunc of Figures 7 and 8 from compatible MODEL and VIEW structures. The primary command loop keeps track of the view data, model data and lexicon, whereas the secondary command loop keeps track of the view data, model data, lexicon (which doesn’t change in this loop), message to be decoded (doesn’t change), pdm, the old symbols of the pdm, the new symbols of the pdm, and the message to be decoded’s decoding (doesn’t change). Although the primary command loop is tail-recurisve, the secondary command loop is not. Non-tail calls are used for the hint and replace secondary commands, abort to the primary command loop when the pdm is decoded. Both the encode and decode primary commands use the abortable function of the view to carry out abortable computations involving the decodings function of the model.

3.2.5 Terminal View

We conclude this subsection by considering the implementation of our three views, structures of signature VIEW. In the terminal view, vd is simply unit, as the view is stateless. Abortable computations may be terminated by interrupting. To handle interrupts, I used an Interrupts structure whose signature, INTERRUPTS, is:

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

The function ignore is used to run its argument function while ignoring interrupts. Such an argument function can call track, which 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 by an argument function of track to determine whether the user has signaled an interrupt so far. The check abort function can call check to determine whether it should indicate that the user wishes to abort. This structure is implemented using SML/NJ’s signal handling facilities and first-class continuations, plus a hidden mutable variable for keeping track of whether the user has signaled an interrupt so far.

3.2.6 Graphical View

In the graphical view, the view data is a record of channels for communicating with the main thread of the view. The run function:

- starts up CML,
- 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 view data vd,
- spawns the main thread of the view (others are embedded in widgets), giving this thread vd, and
- calls the function \(f\) it was given with \(vd\).

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 main view thread creates and realizes the widgets of the GUI and then enters its primary command loop, where it waits for an I-var (write once variable) to be sent to it on the primary command channel component of the view data record by
signature VIEW =
  sig
    include DATA
  datatype primary_command =
    QuitPC
    | LexiconPC of sym_lexicon
    | EncodePC of msg
    | DecodePC of msg
  datatype primary_response =
    (* responses from encode *)
    EncodeWordsNotInLexiconPR of (int * word)list (* int’s are line numbers in message *)
    | EncodeMultipleDecodingsPR
    | EncodeEncodingPR of msg (* the unique decoding *)
    (* responses from decode *)
    DecodeNoDecodingsPR
    | DecodeMultipleDecodingsPR
  datatype secondary_command =
    QuitSC
    | AbortSC
    | CheckSC
    | HintSC
    | ReplaceSC of sym * sym (* ReplaceSC(a, b) means replace Old a by New b *)
    | UndoSC
  datatype secondary_response =
    (* responses from check *)
    CheckNotDecodableSR
    | CheckDecodedSR
    | CheckDecodableButNotDecodedSR
    (* responses from hint *)
    | HintNotDecodableSR
    | HintDecodedSR
    | HintReplaceSR of sym * sym (* HintReplaceSR(a, b) means Old a has been replaced by New b *)

  type vd
  val primaryInput : vd -> vd * primary_command
  val secondaryInput : vd * pdm * sym_set * sym_set -> vd * secondary_command
  val primaryOutput : vd * primary_response -> vd
  val secondaryOutput : vd * secondary_response -> vd
  type abort
  val abortable :
    vd * int * (abort * (abort -> abort * bool) -> abort * 'a * 'b option) ->
    vd * 'a * 'b option
  val run :
    string * string list * (vd -> vd) -> OS.Process.status
end

Figure 5. VIEW Signature

signature CONTROLLER =
  sig
  structure Model : MODEL
  structure View : VIEW
    sharing type Model.SymLinOrd.elem = View.SymLinOrd.elem
    sharing type Model.pds = View.pds
    sharing type Model.SymSet.set = View.SymSet.set
    sharing type Model.SymLexicon.lexicon = View.SymLexicon.lexicon
  val main : string * string list -> OS.Process.status
end

Figure 6. CONTROLLER Signature
functor ControllerFunc(structure Model : MODEL
structure View : VIEW
  sharing type Model.SymLinOrd.elem = View.SymLinOrd.elem
sharing type Model.pds = View.pds
sharing type Model.SymSet.set = View.SymSet.set
sharing type Model.SymLexicon.lexicon = View.SymLexicon.lexicon) :>
    CONTROLLER =
    struct
      structure Model = Model
      structure View = View
      structure SymSet = Model.SymSet
      structure SymLexicon = Model.SymLexicon
    end

datatype secondary = Abort | Quit | Undo (* reason for secondary loop terminating *)

fun secondary(vd, md, lex, msg, pdm, olds, news, msg') =
  case View.secondaryInput(vd, pdm, olds, news) of
    (vd, View.QuitSC) => (vd, md, Quit)
  | (vd, View.AbortSC) => (vd, md, Abort)
  | (vd, View.CheckSC) =>
    secondary(View.secondaryOutput(vd, View.CheckDecodedSR), md, lex, msg, pdm, olds, news, msg')
      case Model.findHint(md, msg, pdm, olds, msg') of
        (md, Model.HintDecoded) =>
          secondary(View.secondaryOutput(vd, View.CheckDecodedSR), md, Abort)
        | (md, Model.HintNotDecodable) =>
          secondary(View.secondaryOutput(vd, View.CheckNotDecodableSR), md, lex, msg, pdm, olds, news, msg')
        | (md, Model.HintReplace(_, _)) =>
          secondary(View.secondaryOutput(vd, View.CheckDecodableButNotDecodedSR), md, lex, msg, pdm, olds, news, msg')
    | (vd, View.HintSC) =>
      secondary(View.secondaryOutput(vd, View.HintDecodedSR), md, Abort)
      case Model.findHint(md, msg, pdm, olds, msg') of
        (md, Model.HintDecoded) =>
          secondary(ViewsecondaryOutput(vd, View.HintDecodedSR), md, lex, msg, pdm, olds, news, msg')
        | (md, Model.HintNotDecodable) =>
          secondary(View.secondaryOutput(vd, View.HintNotDecodableSR), md, lex, msg, pdm, olds, news, msg')
        | (md, Model.HintReplace(a, b)) =>
          let val pdm' = Model.replace(a, b, pdm)
          in
            secondary(View.secondaryOutput(vd, View.HintReplaceSR(a, b)), md, lex, msg, pdm', SymSet.minus(olds, SymSet.fromList[a]), SymSet.union(news, SymSet.fromList[b]), msg')
          of
            (vd, md, Undo) => secondary(vd, md, lex, msg, pdm, olds, news, msg')
          | x => x
          end
    | (vd, View.ReplaceSC(a, b)) =>
      let val pdm' = Model.replace(a, b, pdm)
      in
        secondary(View.secondaryOutput(vd, View.ReplaceSR(a, b)), md, lex, msg, pdm', SymSet.minus(olds, SymSet.fromList[a]), SymSet.union(news, SymSet.fromList[b]), msg')
      of
        (vd, md, Undo) => secondary(vd, md, lex, msg, pdm, olds, news, msg')
      | x => x
      end
    | (vd, View.UndoSC) => (vd, md, Undo)
fun primary(vd, md, lex) =  
  case View.primaryInput vd of  
    (vd, View.QuitPC) => vd  
| (vd, View.LexiconPC lex) => primary(vd, md, lex)  
| (vd, View.EncodePC msg) =>  
    let val ps = Model.unknownWords(lex, msg)  
    in if null ps  
      then case View.abortable  
        (vd, 50000, fn (ab, ca) => Model.decodings(ab, ca, md, lex, msg)) of  
          (vd, md, NONE) => primary(vd, md, lex)  
        | (_, _, SOME Model.DecodingsNone) => raise Fail "cannot happen"  
        | (vd, md, SOME(Model.DecodingsUnique _)) =>  
          let val (md, msg') = Model.encode(md, msg)  
          in primary(View.primaryOutput(vd, View.EncodeEncodingPR msg'), md, lex)  
          end  
        | (vd, md, SOME Model.DecodingsMultiple) =>  
          primary(View.primaryOutput(vd, View.EncodeMultipleDecodingsPR), md, lex)  
      else primary(View.primaryOutput(vd, View.EncodeWordsNotInLexiconPR ps), md, lex)  
    end  
| (vd, View.DecodePC msg) =>  
  case View.abortable  
    (vd, 50000, fn (ab, ca) => Model.decodings(ab, ca, md, lex, msg)) of  
      (vd, md, NONE) => primary(vd, md, lex)  
    | (vd, md, SOME Model.DecodingsNone) =>  
      primary(View.primaryOutput(vd, View.DecodeNoDecodingsPR), md, lex)  
    | (vd, md, SOME(Model.DecodingsUnique msg')) =>  
      (case secondary  
        (md, lex, msg, Model.toPDM msg,  
          Model.symsMsg msg, SymSet.fromList nil, msg') of  
          (_, _, Quit) => vd  
        | (vd, md, Abort) => primary(vd, md, lex)  
        | (vd, md, Undo) => primary(vd, md, lex)  
        )  
    | (vd, md, SOME Model.DecodingsMultiple) =>  
      primary(View.primaryOutput(vd, View.DecodeMultipleDecodingsPR), md, lex)  
  end  
  fun main(cmd, args) =  
    View.run(Aux.lastPartOfPath cmd,  
      args,  
      fn vd => primary(vd, Model.init(), SymLexicon.empty))  
end

Figure 8. ControllerFunc Functor (Part II)

the primaryInput function. Once the I-var has been received, the main view thread begins the process of trying to fill the I-var with a primary command. Once this is done, the primaryInput function will sense that its I-var has been filled, and will return the I-var’s contents to its caller. First, the main view thread 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 do so, or to ask via the window manager for the program to exit. When a primary command button is clicked on, an attempt is made to get the data associated with the command from the user. If the user aborts this process, the main view thread restarts the process of trying to fill the I-var. Otherwise, it fills the I-var with a primary command, and returns to the beginning of the primary command loop.

Secondary commands and abortable computations are also handled using component channels of the view data record. For instance, the check abort function uses a channel of the view data record to communicate to the main view thread how many steps of the computation have been completed, as well as an I-var in which the main view thread can store an indication of whether the user wishes to abort the computation.

Although the view data record is constant, the view does have state: it’s the current expression of the view’s main thread combined with the state of its widgets. For example, when an EncodePC primary command results in primaryOutput being called with an indication of an error (the user’s message either contained one or more words that were not in the lexicon, or it had multiple decodings), the main thread knows that the erroneous message is still displayed in a text widget. As it is likely that the user will wish to edit this message and try encoding the result, it then waits to receive the I-var into which it must put a primary command, and then lets the user try again.

3.2.7 Automatic View

Finally, the automatic view is implemented using scripting, using the view data to keep track of what it should do next. For example, given a message to decode, its view generates a LexiconPC primary command, followed by a DecodePC primary command. If the controller responds with a primary response (indicating an error), then an error message is issued. Otherwise, the controller enters the secondary command loop, and the view issues a sequence of
4. Related Work

Lüth and Wolff (1999) have written an SML functor consisting of an smltk-based GUI for a theorem prover, parameterized by the theorem prover’s model.

Achten et al. (2004) use generic programming to construct graphical editors for values of arbitrary types in Clean, given models of those types.

Flatt and Felleisen (1998) give a small example of how a program can be parameterized by its user-interface. Their linking language allows a user-interface parameter and parameterized program to be mutually recursive. In our approach, we must use higher-order functions to achieve such mutual recursion.

5. Conclusions and Future Work

Command-oriented functional programs are currently structured in an ad hoc way that makes the development of multiple user-interfaces difficult and error prone, and makes it difficult to abstractly understand a program’s command-oriented behavior. This paper has described a software architecture—functional MVC—for such programs that rectifies these deficiencies. It was illustrated and tested by implementing a complete program that was chosen to be complex enough to raise the main issues involved in command-oriented programs, but to be simple enough to be suitable as an example for research and teaching. The program has three views—a terminal one, a graphical one, and an automatic (batch) one.

In order to get a better idea of the generality of our approach, more programs should be implemented using functional MVC. For example, it would be instructive to try refactoring the implementation of Unison (Pierce et al., 2004) so as to use our architecture. Because the Unison GUI is implemented using LablGTK (Garrigue, 2006), which is compatible with OCaml’s threads, it seems likely that this refactoring will be possible.

Another possibility would be to restructure the interactive font-end of an ML compiler such as SML/NJ (Appel et al., 2007) using our architecture, giving it a graphical user-interface (perhaps a full integrated development environment (IDE)), as well as a terminal user-interface. Of particular interest would be the facility for monitoring and aborting long-running computations. Support for debugging could also be added and controlled through both kinds of interfaces.

6. Acknowledgements

It is a pleasure to acknowledge helpful discussions with Andrew Appel, Matthias Blume, Matthias Felleisen, Matthew Fluet, 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 their work on some parts of eXene used by crypto’s implementation.

References
