

# Dynamic Applications From the Ground Up<sup>\*</sup>

Don Stewart   Manuel M. T. Chakravarty

Programming Languages and Systems  
School of Computer Science and Engineering  
University of New South Wales  
{dons,chak}@cse.unsw.edu.au

## Abstract

Some Lisp programs such as Emacs, but also the Linux kernel (when fully modularised) are *mostly dynamic*; i.e., apart from a small static core, the significant functionality is dynamically loaded. In this paper, we explore *fully dynamic* applications in Haskell where the static core is minimal and code is hot swappable. We demonstrate the feasibility of this architecture by two applications: Yi, an extensible editor, and Lambdabot, a plugin-based IRC robot. Benefits of the approach include hot swappable code and sophisticated application configuration and extension via embedded DSLs. We illustrate both benefits in detail at the example of a novel embedded DSL for editor interfaces.

**Categories and Subject Descriptors** D.3.3 [Programming Languages]: Language Constructs and Features

**General Terms** Design, Languages

**Keywords** Dynamic applications, Hot swapping, Dynamic update, Extension languages, Functional programming

## 1. Introduction

Dynamic applications are able to add, remove, and exchange code at runtime. This increases application flexibility by facilitating runtime configuration, user extension, hot code swapping, runtime meta programming, and application configuration by EDSLs (embedded domain specific languages).

Stallman's Emacs editor [27] is a widely known example of a dynamic application from the Lisp world—Stallman calls it an *on-line extensible system*. Emacs consists of a small core, implemented as a conventional C program, which is extended to a fully-fledged editor by a large body of Lisp code, which is byte-compiled and dynamically loaded. The application core contains the Lisp engine as well as those editor primitives that would not be sufficiently efficient in interpreted byte code. Such an application is *mostly dynamic*, but the code of the static core cannot change dynamically;

<sup>\*</sup>This work was funded by the Australian Research Council under grant number DP0211793.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'05 September 30, 2005, Tallinn, Estonia.  
Copyright © 2005 ACM 1-59593-071-X/05/0009...\$5.00.

thus, some functionality cannot change either. In the case of Emacs, the editor primitives and any other primitives provided by the Lisp engine are fixed. This includes the choice of user interfaces, which can be text-based or depend on one or more GUI toolkits (such as Xwindows).

An entirely different example of a mostly dynamic application is the Linux kernel. It can load and unload so-called *kernel modules* at runtime using an in-kernel module loader. Just like Emacs, Linux contains fixed functionality in its static core; some of this functionality (such as the process scheduler, the I/O scheduler, and the virtual memory subsystem) would benefit from being dynamically configurable. In contrast to Emacs, both the core as well as the modules of Linux are implemented in C. As a consequence, kernel configuration is achieved via mechanisms independent of dynamically loaded code—which is in contrast to Emacs, where configuration files are just stylised Lisp files.

In this paper we move beyond applications, such as Emacs and Linux, by exploring a *fully dynamic* software architecture whose static core is minimal; i.e., the core contains only what is needed to load the rest of the application at runtime. This was not a feasible design for Emacs, due to hardware and operating systems constraints, which required an interpreted Lisp implementation when Emacs was designed in the late 1970s. Interpreted byte code, in turn, was not sufficiently efficient for the implementation of performance-sensitive editor primitives. In contrast, we demonstrate by benchmarks of the extensible editor Yi that an editor in a modern, compiled functional language structured as a fully dynamic application can have excellent performance.

We experimented with two fully dynamic applications: the extensible editor Yi and the IRC<sup>1</sup> robot *Lambdabot*. We found two main advantages over applications that are only mostly dynamic. Firstly, with a smaller static core, we gain more flexibility and extensibility from dynamic code loading and code swapping, as fewer features depend on the static core. A case in point are user interfaces in Emacs and Yi. As already mentioned, the user interfaces supported by Emacs depend on the primitive operations realised in Emacs' static core. In contrast the minimal static core of Yi is independent of the choice of user interface, which enables completely dynamic interface configuration.

Secondly, fully dynamic applications enable hot code swapping. We demonstrate this by presenting a method for replacing the *currently executing* application code by a variant without losing the application state. Dynamic code replacement is always a tricky business; in a purely functional language, the semantic consequences are even more subtle. We achieve it by redoing the dynamic boot process while preserving a handle to the application state. This process is fast in practice and avoids a large range of semantic issues

<sup>1</sup>Internet Relay Chat (IRC) is a standardised online chat system.

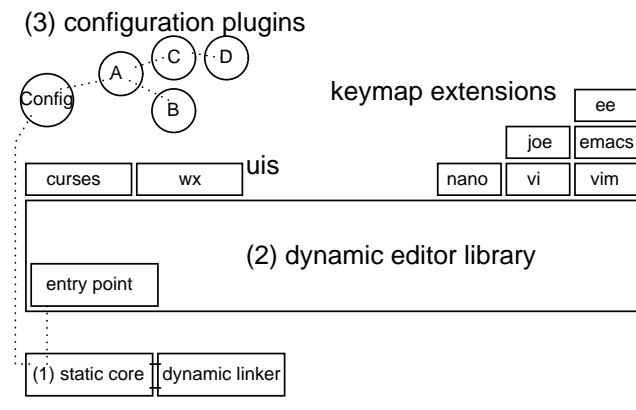


Figure 1. Structure of yi

that other approaches to dynamic software updates incur. Dynamic software updates, and also dynamic extension and configuration, are generally convenient, but they are of special value in 24/7 applications like servers that are in continual use.

Lisp applications, such as Emacs, pioneered the idea of configuration files based on the extension language itself—in this case, Lisp. This approach is lightweight as it avoids a parser and analyser for yet another language. Moreover, configuration languages have the tendency to slowly grow to include ad hoc abstraction facilities and control structures. These features are available in a systematic way right from the beginning if the extension language is used for configuration. This idea is even more powerful when combined with the concept of *embedded domain specific languages (EDSLs)* [13, 14]. Then, configuration files can be tailored to the task and be used for complex configurations. For example, the editor Yi uses self-optimising lexer combinators (i.e., an embedded language of regular expressions) to define editor command sequences such that different configuration files emulate the interface of various existing editors (e.g., Nano, Vim, Emacs). This approach is more lightweight than, e.g., Emacs’ keymaps. It has the added benefit that configuration files are statically type checked, and it illustrates how configurable components can be structured to achieve flexible, modular extensions.

In summary, our contributions are the following:

- a fully dynamic software architecture (Section 2);
- hot code swapping that preserves application state (Section 3);
- practical experience with the new software architecture in two applications, namely Yi and Lambdabot (Sections 2 and 4);
- application configuration and extension via plugins coded in embedded DSLs (Section 5) including a novel EDSL for editor interfaces; and
- performance evaluation of a fully dynamic editor (Section 6).

We discuss related work in Section 7. The source code for Yi<sup>2</sup> and Lambdabot<sup>3</sup> is publicly available.

## 2. Dynamic Architecture

The foundation for extensibility in Yi and Lambdabot is a dynamic architecture based on runtime code loading. Figure 1 illustrates this architecture, as implemented in the Yi editor. It is partitioned into three main components: (1) a static bootstrap core; (2) an editor library; and (3) a suite of plugins. Under this framework,

<sup>2</sup><http://www.cse.unsw.edu.au/~dons/yi.html>

<sup>3</sup><http://www.cse.unsw.edu.au/~dons/lambdabot.html>

the binary itself consists solely of a static core of around 100 lines of Haskell. The purpose of the static core is to provide an interface to the dynamic linker and to assist in hot swapping. The static core has *no* application-specific functionality. Lambdabot is structured similarly, with the difference that the dynamic application is a tree of dependent modules, rather than a single archive of objects like the Yi main library.

### 2.1 Overview

The static core, with the help of the dynamic linker, loads any configuration data (as plugins) followed by the main application code. Control is then transferred to the main application’s entry point—its `main` function—and the program proper starts up. In Yi files are opened, buffers allocated and the user may begin editing. Lambdabot connects to an IRC server node and begins serving user request. Note that the Yi configuration plugins and the main application are dynamically linked against each other so that plugins may call application code (and vice versa). For clarity we omit these details from Figure 1.

During execution the user may invoke the Haskell dynamic linker library [24] to recompile and reload any configuration files, or to reload the application itself. The state of the application is preserved during reloading (hot swapping), with the result that new code can be integrated or existing code be replaced, without having to quit the application. In the rest of this section, we discuss the process of booting in more detail; in the following section, we discuss hot swapping.

### 2.2 Dynamic Bootstrapping

The single purpose of the minimal static core is to initiate the dynamic bootstrapping process, by arranging the dynamic linking of the application proper. Hence, we seek a structure that is generic in its functionality (so that the structure is reusable), efficient and type safe. The following simplified code illustrates such a structure:

```
import System.Plugins

main = do
  status <- load "Yi.o" [] [] "main"
  case status of
    LoadFailure e   -> return ()
    LoadSuccess m main' -> main'
```

Here `Yi.o` is the main module of the editor library, which we load using the plugin infrastructure described in our previous work [24]. `Yi.o` is the root of a tree of modules determined by module dependencies. The plugin infrastructure takes care of recursively loading all required modules and packages, such that the entire program is loaded and linked. The final argument to `load`—i.e., `"main"`—is the symbol that represents the application’s entry point. If loading is successful, the plugin infrastructure presents us with the Haskell value represented by the given symbol. Hence, we transfer control to the dynamic portion of the application by evaluating that value.

On the dynamic side, the entry point is a conventional main function:

```
main = do
  setupLocale
  args <- getArgs
  mfiles <- do_args args
  ...
```

In this case, the Yi editor entry point is called, and execution continues as if the dynamic `main` had been invoked directly at

```

..
.. The may be invoked directly, or sometimes as arguments to other
.. /operator/ commands (like d).
..
cmd_move :: VimMode
cmd_move = (move_chr >|< (move2chrs +> anyButEsc))
  "meta" \cs st{st{acc<cnt} ->
    (with (msgClrE >> (fn cs cnt)), st{acc=[]}, Just $ cmd st)

  where move_chr = alt $ M.keys moveCmdFM
        move2chrs = alt $ M.keys move2CmdFM
        fn cs cnt = case cs of
          .. 'G' command needs to know Nothing count

*Yi/Keymap/Vim.hs* 175,19 20%
..
.. | The editor main loop. Read key strokes from the ui and interpret
.. them using the current key map. Keys are bound to core actions.
..
eventLoop :: IO ()
eventLoop = do
  fn <- Editor.getKeyBinds
  ch <- readEditor input
  let run km = catchDyn (sequence_ . km =<< getChanContents ch)
    (\(MetaActionException km') -> run km')
  repeatM_ $ handle handler (run fn)

*Yi/Core.hs* 241,1 30%
*Yi/Keymap/Vim.hs* Line 175 [20%]

```

Figure 2. Screenshot of Yi’s ncurses interface

startup. A screenshot of Yi running under its ncurses interface<sup>4</sup> is in Figure 2.

### 3. Hot Code Injection

Once the application is up and running there are two operations that we would like to be able to perform:

1. reloading of plugins and
2. hot swapping of application code.

The first operation means that changes to configuration files (implemented as plugins) can affect the running application without having to restart the program. The second is more profound: we want to replace application code while the editor is running, without losing state—allowing us to “fix-and-continue” or to incorporate new functionality on the fly. We will first consider how to reload configuration files in a dynamic architecture, which then leads us to the second, more difficult problem.

#### 3.1 Dynamic Reconfiguration

We follow the Lisp tradition of expressing configuration files as source code in the extension language, in our case Haskell. The reasons for using the extension language rather than special purpose configuration languages are very much the same as those for the use of embedded domain specific languages (EDSLs) over standalone DSLs [13, 14]. However, we also inherit the drawbacks of the EDSL approach, such as error messages that are harder to comprehend for end users. On the positive side, we shall see in Section 5 how configurations can naturally and conveniently grow into EDSLs in our approach.

In any case, users specify their own configuration settings by writing Haskell code in configuration files. These files are compiled and dynamically loaded. The values they provide are used to update a default configuration record type; in other words, we follow the scheme for application configuration via plugins described in our earlier work [24].

Configuration files are dynamically loaded when the static core of Yi begins executing. The user-defined configuration values are retrieved and passed to the dynamic editor core, which uses these values to set initial states for the various components of the editor. It is important to remember that it is the static core that performs

<sup>4</sup> A prototype interface based on wxHaskell [22] is also being developed

all dynamic linker operations, including plugin loading, in our architecture. All invocations of the dynamic linker by dynamic code go via the static core; i.e., the dynamic code calls into the static core, which in turn forwards the request to the dynamic linker. As we discuss soon, this structure is crucial for the reloading process.

To be able to pass configuration values through to the dynamic code, we need extend the simple dynamic main function illustrated in Section 2.2 slightly. Rather than jumping to a constant main function, we instead pass the configuration values as arguments to a modified entry point:

```
main' :: Config -> IO ()
```

To initiate reloading of configurations—or any other plugins—from the dynamic application, we need to interact with the dynamic linker. As already explained, the linker is not visible from the dynamic code: it is linked to the static core. This structure is crucial to enable the hot swapping of the entire dynamic application code, which we will discuss shortly. Hence, the static core needs to provide the dynamic application the required functions of the dynamic linker as function arguments to the dynamic entry point. Overall, the dynamic entry point has the following type:

```

type DynamicT =
  (Maybe State,           -- editor state
   Maybe Config,          -- configuration values
   Maybe State -> IO (),  -- 'reboot' function
   IO (Maybe Config))    -- 'reconf' function

```

```
dynmain :: DynamicT -> IO ()
```

The first component is an optional editor state value for when we wish to preserve state over hot code swapping (Section 3.2). The second field is a configuration record retrieved from the configuration plugins. The final two components constitute dynamic linker functions, which we call `reboot` and `reconf`, respectively.

The function `reconf` is a simple wrapper around the dynamic linker’s reloading primitive, `reload`, which checks for changes to the configuration files. If there are changes, `reload` triggers recompilation and reloading of the configuration modules. The reconfiguration function has the following type:

```
reconf :: IO (Maybe a)
```

The type is polymorphic so that we can enforce statically that `reconf` does not depend on the representation of the values extracted from configuration files—that is the concern of the consumers of configuration values.

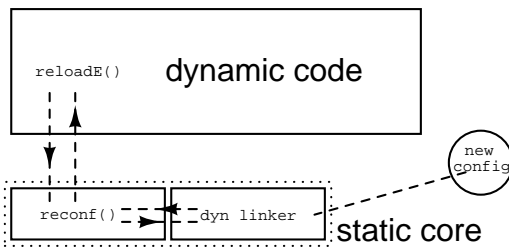
When `reconf` is called from the dynamic code a new config value is retrieved by the static core, which passes this value back to the caller in the dynamic application. The dynamic code then uses the new value to update the application’s state. For example, Yi calls `reconf` by defining an editor action in the dynamic code:

```

reloadE :: Action
reloadE = do
  modifyEditor_ $ \e -> do
    conf <- reconf
    return $ case conf of
      Nothing -> e
      Just (Config km sty) ->
        e { curkeymap = km, uistyle = sty }
  UI.initcolours

```

The function `reloadE` atomically (with respect to the dynamic application’s global state) invokes the static linker’s `reconf` function. Then, control is passed back to the static core where the actual recompilation checks and dynamic reloading takes place. Once the new configuration data is returned `reloadE` uses it to set the cur-



**Figure 3.** Reloading configuration files from dynamic code

rent user interface style and key mappings. In this way we can inject new (stateless) code dynamically, enabling changes in configuration files to be immediately reflected in the running application. This process is illustrated in Figure 3.

There is a question of type safety when importing code dynamically. We are able to check configuration files using techniques developed in our earlier work on type safe plugins [24]. Configuration files may be checked prior to loading either through the use of dynamic typing, or by employing the type checker at runtime to check the interface between static and dynamic code.

We use dynamic reconfiguration for similar purposes in `Lambdabot`. `Lambdabot` was originally developed as a static binary of some 1500 lines of Haskell; in addition, it used dynamically loaded plugins as an extension mechanism. We refactored `Lambdabot` to use a minimal static core which loads the actual `Lambdabot` application dynamically. In doing so, we needed to make all further loading of plugins go via the static core for the reasons just discussed. The new static core of `Lambdabot` is less than a 100 lines.

### 3.2 Pulling the Rug Out from Underneath

In the discussion so far, we neglected an important problem: during reloading a plugin, be it the main body of the dynamic application or an extension, we lose any state maintained in that plugin. In our experience, it is not unusual for plugins to require private state (for example, in the form of `IORefs`), which is reinitialised on reloading. The loss of this state may be acceptable for small plugins or configuration files, but is unacceptable if we attempt to dynamically reload large amounts of code, or indeed the application itself. Hence, we will now turn to discussing an approach to maintain plugin state across reloads by *state injection* from the static core.

#### 3.2.1 Keeping state

When updating code dynamically, we wish to continue execution from exactly the point at we reached prior to the dynamic update. To do this, we must reconstruct any relevant application state built by the code before the update. Reconstructing an application’s state in Haskell is usually simple. By default entire Haskell programs are purely functional, so simply re-entering the application via its normal entry point is enough to recreate the application’s state from a previous run. There is no need to actually store state between runs, as we are always able to reconstruct it.

However, some applications require large amounts of mutable state, and reproducing the mutations to that state may be infeasible. `Yi` is such an application. Editor buffers are potentially very large flat byte arrays, constructed over the entire running time of the application. Any copying is prohibitively expensive. Buffers in `Yi` are stored in a single global state component of the editor. What we need to do is capture this state during the update, and then inject it into the new updated code, continuing exactly where we left off.

This brings us to a central reason for using a dynamic application structure centred around a minimal static core; i.e., a reason

besides the desire to modify all functionality of the application at runtime. We use the static core to keep all global state while the dynamic linker reloads plugins. For this to work it is crucial that all requests to load or re-load plugins go via the static core, as mentioned previously. Ultimately, the static core is the only safe place to keep the global state during reloading, as the whole dynamic application code may be replaced. In other words, we structure the entire dynamic application so that it takes its state as an argument, making it purely functional again, and hence, enabling state-preserving hot swapping.

At this point, Haskell offers significant advantage over languages which encourage the use of global state. Unrestricted use of global state leads to a multitude of values scattered throughout the program; hence, bookkeeping of such a dispersed state becomes difficult and potentially infeasible. Hot swapping is only practical if the application already has disciplined use of global state—as is the case by default in Haskell programs. Without restricted state, runtime system or operating system support seems necessary to deal with the state problem.

#### 3.2.2 Redoing main

So far, we established that hot swapping needs to be via the static core, while passing any global state from the old to the new instance through the static core. We realise this by the `reboot` component of the configuration argument of type `DynamicT` passed to the dynamic entry point, as discussed in Section 3.1. The static core will pass the following function as a concrete value for `reboot`:

```
remain :: a -> IO ()
```

It takes a state value and ‘returns’ a nominal `()` value—in fact it never returns to the caller, it instead reloads all the application code (which is fast, c.f Section 6). The input state is then passed back to the new main function we just loaded. The new dynamic entry point then restarts the editor and uses the state parameter to restore the previous environment. The state value lets us keep any files and buffers open in `Yi`, for example.

We utilise polymorphism in `remain`, as with other interfaces to the static core. This simplifies state transfer, as we can be sure that the static core does not depend on the representation of the state component.

The complete sequence of operations performed by `remain` is:

1. the dynamic code calls `remain` with the current state as an argument,
2. static core unloads the dynamic application,
3. the (new) dynamic application is loaded, and
4. the core calls the application’s entry point, passing the old state as an argument.

In the static core, we implement this as follows:

```
remain :: a -> IO ()
remain st = do
  unloadPackage "yi"
  status <- load "Yi.o" [] [] "main"
  case status of
    LoadFailure e   -> return ()
    LoadSuccess m main' -> main' st
```

The function `unloadPackage` unloads the entire dynamic application (packaged as an archive). We then reload the code from object files, and reenter the code via its normal entry point, supplying the old state value as an argument. The dynamic code then inspects this value, and does a quick restart—avoiding the initial application start up—and instead directly entering the editor main loop.

In `Yi` we invoke the hot swap event by getting a handle to the current state, shutting down the editor (which involves terminating

some threads, and exiting the user interface). We then jump into the static code by calling `reboot` (i.e. calling into the static core), and passing down the state.

```
rebootE :: Action
rebootE = do
  st <- getEditorState
  Editor.shutdown
  reboot (Just st)
```

When the new application code is entered after the reload, the old state is injected back into the application, and we short circuit the usual initialisation steps.

We can increase the performance of dynamic reloading by structuring the dynamic code as a single library, rather than a tree of separate modules. Doing so means that the dynamic linker doesn't have to read interface files for each module it needs to load—to chase dependencies—and instead loads the entire dynamic linker in one load. In Section 6 we discuss the performance of this operation in detail.

### 3.3 Immortal Values

Replacing code dynamically while still preserving state works well when extending functions. New functionality, bug fixes and extensions can all be integrated without having to quit the application. However, reinjection of global state is only safe if the definition of the state data type is unchanged in the new code. If, for example, the buffer type changes to include an extra field and we restore the old state containing an old buffer value lacking this field, the new code will crash—it simply is not compiled to handle a value of the old type. What we need is a safe way of injecting an old state value into a new, different state type. This problem, state transfer, has been considered in work relating to operating systems supporting hot swapping [26, 5] as well as in the work of Hicks [12].

We tackle this problem by reducing it to a form of parsing: we define an error-handling parsing function to inject values of the old type into the new type. To transfer values between types we convert the state value into a binary representation, via serialisation. Prior to rebooting the application, we encode the old state value in this binary format. After injecting this binary state value into the new code, the value is parsed to construct a state value of the new type, where missing, or extra, fields are either ignored or replaced with default values. In this way global state values can be preserved over data type changes, and the state value is effectively immortal.

A more complex solution could use a version tag in the state data type and a class of injection functions to achieve binary compatibility between arbitrary versions of the state type.

### 3.4 Persistent State

The problem of preserving state during dynamic code reloading is tightly connected to the general problem of persistence. Applications that are able to extract and inject their entire state can be extended to support persistence via the usual mechanisms—e.g. serialisation of values to binary representations [30]. When passing the state value from dynamic code to the static core, the core can arrange to write the state to disk, and on rebooting, reread this state, passing it back to the dynamic main.

The problem of preserving state across code loading effectively forces us to separate and sanitise use of state in the application. Extending the state preservation to full persistence is relatively simple once we reach that point.

### 3.5 Static Applications

Dynamic linking is not supported in all environments that the compiler, GHC, runs in. For portability, it is convenient to be able to build a statically linked version of the application that provides

all the same features and is usable in less supported environments. Such an application will lack the ability to dynamically load configuration files or new code.

Producing a static version of the dynamically loaded application can be achieved quite simply by adding a conventional `main :: IO ()` to the main module of the dynamic application, and arranging for the compiler to treat this as the normal entry point of the program, leaving the boot stub out of the compilation process. Both Yi and Lambdabot can be compiled as static-only applications in this way.

This is possible as the static core is minimal—it contains no application code—so even when it is left out, we still arrive at a working application. This clean separation of all dynamic linking-related code into a single static core is another reason for forcing the dynamic application to delegate all invocations of the dynamic linker to the static core.

## 4. Adapting Lambdabot

Lambdabot is an Internet Relay Chat (IRC) [1] robot, or *bot* for short. IRC features messaging between individual users as well as *channels* where multiple users can meet to engage in conversation. In the chat network, bots appear as normal user clients that provide a wide range of services by reacting to commands sent to them directly or issued on a channel monitored by the bot. Some bots with special privileges authenticate users, award operator privileges to selected users, and stop network abuses. Other bots simply provide services to the users of a particular channel. Lambdabot belongs to the later category and is, for example, used on the `#haskell` channel of the `irc.freenode.net` network. Lambdabot services are implemented by a suite of dynamically loaded service plugins that include dictionary queries, weather forecasts, notification services, but also an online Haskell type checker and interpreter.

The first version of Lambdabot had a fairly large static core (about 1500 lines) implementing IRC protocol support. The service plugins amounted to another 5000 lines of Haskell. The main incentive for the use of plugins was ease of configuration and the ability to upgrade plugins without needing to shut down the bot. Channels, such as `#haskell`, almost always have active users who might want to use the bot's services. Hence, hot code updates are an attractive option. Unfortunately, the original architecture with the fairly large static core suffered from two drawbacks:

1. Patches to the core of Lambdabot required a complete restart—and the core is quite large (nearly 25% of the application's code).
2. GHC's runtime system requires all libraries that are both used in statically and dynamically linked code to be in memory twice. This is clearly a waste of resources.

We refactored Lambdabot to use the dynamic architecture described in this paper. Now, the static core has less than 100 lines of Haskell, which are linked statically against the `hs-plugins` linker. This static core when invoked dynamically loads the Lambdabot main module, triggering cascading dynamic loading of the rest of what was formerly the static binary and is now the dynamic application code. After loading, control is transferred to the dynamic entry point of Lambdabot, passing a set of linker functions as an argument. These functions are then used to dynamically load Lambdabot's suite of service plugins. At this point the application connects to the server. Under this new architecture, we avoid both of the above problems.

Refactoring Lambdabot proved to be quite painless, despite Lambdabot having been developed without any thought of structuring it as a dynamic application. The responsibility of loading plugins was transferred from Lambdabot's simple object loader, to the static core linked against the `hs-plugins` library. This was a

matter of exporting linker functions in a manner similar to Yi, and in fact reduced and simplified the main application code, as all dynamic loading glue was factored into the static core.

Our experience with Lambdabot suggests that adapting other Haskell applications to the dynamic architecture we propose is feasible. Dynamically loading any Haskell application can be achieved with only small modifications to the application. In particular, the static entry point is simply hoisted to become the dynamic entry point.

## 5. Configuration and Extension

As already mentioned in Section 3.1, the Lisp experience shows that it is attractive to realise configuration files in a dynamic environment as dynamically loaded code. After that, it is a small step from configuration files to extension modules and the use of embedded domain specific languages (EDSLs) [13, 14] for both application configuration as well as extension. Languages that are used to extend applications dynamically are called *extension languages*. Many programs written in conventional languages opt to implement extension languages by embedding interpreters for new application-specific languages—Vim can be configured and extended in *Vim script*, for example. An alternative is to embed an existing scripting or extension language in the application, for example Perl [29], Lua [16, 17], or Scheme.

Lisp programs, such as Emacs, allow configuration and extension of the application in the application language itself. Yi and Lambdabot are built following this model, as dynamically reloadable plugins directly provide extension and configuration via compiled code. Moreover, we use EDSLs for more complex configurations and extensions.

To illustrate our approach to creating modular, flexible extensions, we use as a running example the self-optimising lexer combinators used by Yi. These are an embedded regular expression language for implementing editor keystroke interfaces (or keymaps) based upon a lexer combinator library. When combined with dynamically reloadable configuration files and hot swapping, the use of a configuration and extension EDSL allows us to rapidly extend keystroke interfaces for Yi. Indeed, with very little programming effort, Yi is able to emulate, to differing degrees of completeness, several existing editors, including Vi [28] and Vim [8] (different editors of the Vi family) as well as Emacs [27], Nano [23] and Ee [15].

Configuration files in Yi are conventional Haskell modules which Yi arranges to compile and dynamically load, bringing user configuration data and code into the editor. The user may thus implement small interface extensions, or indeed completely new interfaces, by extending their configuration files. We use this EDSL to construct new lexers emulating existing editor interfaces, and make use of advanced lexer features, such as threaded lexer state storing nested lexer histories, and lexer table elements that trigger monadic lexer switching, to develop sophisticated interfaces.

### 5.1 The Lexer Language

The interface extension language of Yi is an embedded lexer combinator language based on the self-optimising combinators described in our earlier work [9]. We use these as the foundation of a novel EDSL for the construction of *key bindings*. User-written lexer fragments are written in configuration files which are then appended to, or replace, default key bindings at runtime. The use of a DSL for specifying key bindings allows full interfaces to be constructed by those unfamiliar with the implementation of the application. Additionally, interfaces written in a combinator style can be constructed from fragments spread over multiple files, allowing code reuse. This enables users to conveniently specify their own custom mappings.

The use of a domain-specific language for key bindings has enabled us to create specifications for the often obscure grammars of existing editor interfaces with little effort. We suspect that key binding construction via DSLs will allow cleaner and more intuitive interfaces to be constructed, as the gap from formal specification to implementation is less.

### 5.2 Overview of Key Bindings

Most editors have a fixed mapping of character sequences to editor actions. Users interact with the editor by typing character sequences which trigger specific functions in the editor. This method of interaction is analogous to an interpreter—keystrokes are lexed and parsed producing phrases in the *editor language* which specify editor state changes. These phrases are then executed, causing the corresponding changes to occur.

It is important to note that the input to the lexer is possibly infinite, making it essential to use a lexer that consumes input lazily (lexer generators that consume input strictly are not suitable). We can model a lazy keymap lexer as a function:

```
keymap :: [Char] -> [Action]
```

where `Action` is the type for editor actions. The return type of `keymap` is a list of such actions, as each key sequence generates a discrete action event, and there are an infinite number of such events.

In Yi the keybinding lexers have precisely this type. Multiple functions of this type can be loaded as a plugins to the editor, and lexer definitions can be implemented as separate combinator fragments spread across multiple modules. Different editor interfaces are emulated by writing different lexers. The result is an elegant combinator language for writing `[Char] -> [Action]` editor interfaces.<sup>5</sup>

We briefly summarise the lexer combinator language. It consists of a set of regular expression combinators, and a number of combinators for binding regular expressions to actions, to produce lexer fragments (`action`, `meta`), joining regular expressions (`>|<`), combining lexer fragments (`>||<`), and running lexers over input (`execLexer`). The semantics of these operations are described in [9].

Regular expressions	Construction and control
<code>epsilon</code>	<code>action</code>
<code>char</code>	<code>meta</code>
<code>string</code>	<code>&gt; &lt;</code>
<code>star</code>	<code>&gt;  &lt;</code>
<code>plus</code>	<code>execLexer</code>
<code>quest</code>	
<code>alt</code>	

### 5.3 A Simple Example

Here is a simple example of how Yi users would dynamically extend the editor interface via code fragments in configuration files. The user writes a configuration file, `Config.hs`. This file will be loaded by the static core when the application is invoked, and reloaded on demand as the user extends or modifies the code:

```
module Config where

import Yi.Yi
import Yi.Keymap.Vim
```

<sup>5</sup>The original lazy lexer combinators described in Chakravarty [9] have been modified in two ways to support key binding programming: firstly, lexers immediately return tokens once they are uniquely determined (not waiting till the next input character). Secondly, lexer composition with overlapping bindings is permitted, with the new bindings overriding previous bindings.

```
yi = settings { keymap = keymapPlus bind }
```

```
bind = char 'n' 'action' \_ ->
  Just $ mapM_ insertE "---"
```

In the above example, `settings` is a set of default configuration values, and `yi` is the distinguished value the dynamic loader expects to find in any config file. The `keymap` field specifies which lexer the editor is to use as its keystroke handler—in this case the Vim interface augmented with a binding for the character ‘n’, that when triggered inserts the Haskell comment token “--” into the buffer at the current point. The function `keymapPlus` augments the default binding with new lexer fragments, allowing us to compose the Vim lexer with new user-supplied code.

User’s may thus write their own lexer bindings in configuration files using the EDSL, gaining the safety and expressiveness of Haskell in the process. Configuration files that fail type checking are rejected, and default values are substituted in their place.

#### 5.4 Lexer Table Elements

Key bindings, under our scheme, are lexers from strings to action *tokens*. Keystroke interfaces use regular expressions to construct self-optimising lexer tables specifying what editor actions to invoke given a particular series of keystrokes. The elements of the lexer tables are functions combining primitive editor operations. These are built from a set of around 80 primitives providing the following functionality:

- Movement commands: left, right, up, down, goto, ...
- Buffer editing actions: insert, delete, replace
- Reading portions of the buffer: read, read line
- Multiple-buffer actions: next, prev, focus, unfocus, close, split
- Undo/redo, yank/paste, search/replace
- File actions: new, write
- Meta actions: reboot, reload, quit, suspend
- Higher-level actions: map, fold

These functions return `Action` type, a synonym for `I0 ()`. Our extension language thus allows us to construct bindings from character input to monadic expressions, specifying state changes to the editor. These actions values can be composed with `>>=` (or other monadic combinators), in the usual way.

When passed input, keymaps return a series of (unevaluated) editor actions. For example, in Vi or Vim emulation mode, user input of “j12x” generates the following list of actions:

```
[downE, leftE, replicateM_ 2 deleteE]
```

These actions are self-explanatory. As the user types characters a lazy list of these editor actions is produced. These actions in turn need to be forced, to generate their effects, so the main editor loop body is as follows:

```
sequence_ . keymap =<< getChanContents ch
```

The character input reader runs in a separate thread, returning a lazy list of keystrokes via a channel to the main thread. The list of keystrokes is passed as an argument to the current keymap, our (pure) key binding lexer. The resulting list of actions are evaluated as they are produced, causing immediate state changes in the application.

#### 5.5 A Complete Interface

We now present a basic keybinding definition for the `ee` [15] editor, implemented in our combinator lexer extension language. This interface can be written to a configuration file, replacing the

default interface, and may be extended while the editor is running. The brevity of the code provides an indication of the power of domain specific extension languages for this task, and we believe the lexer combinator EDSL to be of general utility for programming application keystroke interfaces:

```
keymap cs = fst3 $ execLexer lexer (cs, ())
```

```
lexer = insert >||< command
```

```
insert = any 'action' \[c] -> Just (insertE c)
```

```
command = cmd 'action' \[c] -> Just $ case c of
  '\^L' -> leftE
  '\^R' -> rightE
  '\^U' -> upE
  '\^D' -> downE
  '\^B' -> botE
  '\^T' -> topE
  '\^K' -> deleteE
  '\^Y' -> killE
  '\^H' -> deleteE >> leftE
  '\^G' -> solE
  '\^O' -> eolE
  '\^X' -> quitE
  -      -> undefined
```

Where `any` and `cmd` are patterns that match any character, and the set of `command` characters, respectively. At the top level, a key binding takes a lazy list of input characters, and runs the lexer on this list. This causes the lexer table to be dynamically constructed. The lexer is built from two lexer fragments: one for self-inserting characters, and another for a small set of control characters.

Partially defined extensions, such as those including `undefined`, or code that throws other exceptions [20], may be caught by the application, and dealt with in the usual manner. For example, Yi catches and prints exceptions via the message buffer, before resuming execution in the main loop. Furthermore, the effect of malicious extensions can be mitigated somewhat using techniques described in our previous work [24].

We now consider more sophisticated interfaces utilising threaded recursive state, lexer switching and finally monadic lexer switching.

#### 5.6 Threaded State

A powerful feature of the lexer combinator language upon which we base our extension language is the ability to thread state through the lexer as it evaluates input. This has proved to be invaluable, and we use the state, for example, to communicate results between different regular expressions, to implement command history and line editing, as well as stackable dynamic key mappings.

We take as a simple example the task of emulating the prefix repetition arguments to commands used in the Vi family of editors, including Vim and Vi. Many commands can be optionally prefixed with numerical repetition arguments. For example `3x` is the sequence to delete 3 characters. We need a way to parse any numerical prefix `n` in a digit lexing fragment, but make that value available later on, once we’ve decided which action to perform.

We implement this by threading an accumulator as a state component through the lexer. Digit key sequences can be appended to this lexer state by the digit lexer fragment (as well as, perhaps, echoed to a message buffer), until a non-digit key is pressed. Control is then transferred to a command character lexer. Once the full command has been identified, the digit state value is retrieved and the command replicated by that value. The digit lexing code is

```
nums :: Lexer String Action
```

```

nums = digit 'meta' \[c] s ->
  (msgE (s++[c]), s++[c], Just lexer)

```

The lexer state now consists of a `String` value. Rather than using the `'action'` combinator for binding regular expressions to actions, we instead access a lexer state component when the lexer table element is retrieved. This is achieved via the `'meta'` combinator, which allows us access to the lexer state component, as well as specifying (1) any token to return, (2) a new state value, and (3) a lexer to continue execution with.

In the above code, `s` is the state component of the lexer. When a `digit` is matched in the input stream, we extract the existing lexer state. We immediately echo the input value, and any previous input digits to the message buffer (via `msgE`), append the current character to the state, and continue execution with the default lexer. In this way digits will be accumulated, as well as being echoed to the screen each time they are pressed.

The state can be used later by the command lexer fragment:

```

command = cmd 'meta' \[c] s ->
  (msgClrE >> fn c (read s), [], Just lexer)
  where fn c i = case c of
    '\^L' -> replicateM_ i leftE
    '\^R' -> replicateM_ i rightE
    ...

```

Here we first clear the message buffer, then construct a new editor action using the digits stored in the state to specify the repetition. Finally, we return this action, along with a new empty state, and continue with the default lexer.

The state is convenient for other values we extract from the key input stream. For example, suppose we wish to maintain a command history. We can keep track of all editor input in a list value inside the lexer state, and later retrieve or index this history, when we receive user input to do so.

These simple examples illustrate the close fit between our lexer combinators, and the domain of interface keybindings. We continue by describing how to model more difficult features of key binding syntax.

## 5.7 Modes

Most editors consist of a set of *modes*, or distinct sets of key bindings. Usually only one such set is in operation at any point. Even in *modeless* editors there are submodes introduced when certain keys are pressed.

There is a direct encoding of modes as independent lexer fragments in our extension language, with the result that the distinction between moded and modeless editors evaporates—modeless editors just consist of one large default lexer, with smaller lexer fragments for submodes. The main problem is actually how to achieve mode switching. That is, how to bind a keystroke to an action that causes control to pass to a new lexer.

The usual way to switch modes is via a distinguished mode switching sequence. There are often multiple ways to enter a new mode, the difference being that particular actions are performed prior to the mode switch. These rules are directly implementable in the lexer framework, via the `meta` action.

For example, we may specify a lexer fragment for the Vi and Vim editors that specifies mode switching into `'insert'` mode (equivalent to the default self-insertion mode in Emacs):

```

switchChar 'meta' \[c] st ->
  let doI a = (with a, st, Just insert)
  in case c of {
    'i' -> doI nopE
    'I' -> doI solE
    'A' -> doI eolE

```

```

'a' -> doI $ rightOrEolE 1
'o' -> doI $ eolE >> insertE '\n'
'O' -> doI $ solE >> insertE '\n' >> upE
'C' -> doI $ readRestOfLnE >>= setRegE >> killE
}

```

These bindings all cause a mode switch to the insert mode lexer (`insert`) where keys are inserted into the buffer by default. The first binding (`'i' -> doI nopE`) causes a direct switch, whilst the other bindings all perform actions of various complexity prior to the switch. For example, `'A'` moves the cursor to the end of the line, and then enters insert mode.

We may also pass state from one mode to another via the state component of the lexer. State is sometimes useful over mode switches. For example, when exiting from a line editing mode, hitting `Ctrl-M` causes the final edited input to be passed from the line editor, to a sub-mode that can then interpret the edited string.

We now consider how to add new bindings that dynamically extend editor key bindings.

## 5.8 Dynamic Mappings

Many applications allow users to dynamically extend the table of key mappings by binding keys to a new sequence of keystrokes. In Vi or Vim this is achieved by `:map` and `:unmap` commands. We may also remap previously defined mappings, shadowing the former definition with a new definition.

In order to implement dynamically extensible mappings cleanly we would like to be able to update the application's lexer table at runtime. This is not possible if our lexer is either a hand-written parse function (i.e. the lexer table is compiled code) or if our lexer is an statically generated lexer table. One solution is to use dynamic reloading to recompile the lexer on the fly, however this is a rather sledgehammer approach when we simply want to update a lexer table.

An alternative is to construct new lexer combinators at runtime. This approach—implementing dynamically extensible key mappings via runtime constructed combinators—cleanly and elegantly allows us to implement dynamic mappings.

Let us consider user input in a vi-like editor of `:map zz d4j`. This binds the key sequence `zz` to the actions produced by typing `d4j`. This sequence deletes the next four lines down from the current line. How do we implement this using the lexer combinator extension language? What we need to be able to do is to inspect the current lexer table to find what actions the input `d4j` are bound to. We can then use the resulting actions to build a new lexer fragment dynamically.

To inspect the lexer table we need to run the current lexer, on the side, with `d4j` as input. This will produce a list of `Actions`. We can then use a `fold` to join the list of actions into a single action. This action we then use on the right hand side of a newly constructed lexer fragment (i.e. a new lexer table entry), bound to `zz`, which we use to augment the current lexer.

Assuming we wish to bind our `zz` to the command mode lexer, we can inspect the lexer table elements bound to an arbitrary input string, `inp`, like so:

```
lookupC inp = fst3 $ execLexer command (inp, [])
```

This builds a new lexer on the fly, with `inp` as input, returning a list of actions that are associated to the given input. We can construct a single action from the list of actions `as`. This can be achieved by folding the monadic `>>` over this list, with a `no-op` to start the fold off. We then construct a new lexer fragment binding `"zz"` to the resulting `Action`:

```
bind zzs = string zzs 'action' \_ ->
  Just (foldl (>>) nopE as)
```

The last step is to transfer control from the currently executing lexer to a new lexer augmented with the bind we have just constructed. The right hand side of the meta binding for the `:map` command is thus:

```
(msgClrE, [], Just $ command >||< bind)
```

where new bindings in the right argument of `>||<` replaces any existing binds in command.

One issue remains, however. When we combine two lexers we create a new lexer value to which we pass control. If we ever call `:map` again, we will add any new binding to the *original* lexer, losing any dynamic bindings. The solution to this is to store the most recent lexer, with any dynamic mappings it has, in the lexer state, so that bindings will not be lost on later mappings. We can extend the lexer state to store the current lexer, along with the accumulator, like so:

```
data State = State {
  acc  :: String,
  cmd  :: Lexer State Action
}
```

Now new key mappings will be bound to a meta action as before, and we also record this augmented lexer in the lexer state. Rather than hard coding which lexer control passes to a meta action, we instead transfer control to the lexer found in the threaded state. Now, we are in a position to consider how to implement command *unmapping*.

### 5.9 Nested Mappings

It is often desirable to allow users to unmap commands, and most extensible editors support this. Two main approaches may be considered. The first is to only allow one level of mapping to exist at any point. That is, if the sequence `zz` is mapped twice, the second mapping will overwrite the first mapping, and it will be forgotten – this is Vi’s behaviour. Unmapping `zz` will then revert to the mapping for `zz` found in the default lexer, i.e. no mapping at all.

We can implement this simple unmap similar to map. We run the default lexer with the sequence we wish to unmap as input. Three possible values will result, either:

- No actions are bound to the sequence
- Exactly one action
- Multiple actions are bound to the sequence

If no actions were bound to some sequence we are unmapping, then we can unmap this sequence from the current mapping by binding it to the empty action: `noPE`. If exactly one action is bound to the sequence, then that input did have a previous binding, and we restore this by binding the input to the action we have just retrieved. The difficult case is if some input string *s* results in multiple actions—this implies that components of the string are individually bound. In this case we really need to delete the entry for *s* from the table entirely (mapping *s* to `noPE` won’t do, as it adds a binding). Direct deletion of table elements is not yet possible, however we can emulate deletion by rebinding dynamically added bindings to the original lexer, filtering out the entry we wish to delete.

With this model it is possible to implement *stackable* dynamic mappings. Rather than new bindings overwriting previous dynamic bindings, we can instead revert to the last bind (which may have been the result of a `:map`) for a particular sequence. To do this we extend the lexer state to carry a table whose keys are input character sequences, and whose elements are lists of bindings. We can then implement an `:unmap` that reverts to the innermost binding for a particular sequence, by popping the top element of the list of bindings for that sequence. It would also be possible to browse the

history of key bindings or to write the current set of bindings to a file, so that dynamic mappings can persist beyond the current editor session.

### 5.10 Higher Order Keystrokes

A number of existing editor interfaces have commands that take other commands as arguments. For example, it is common for simple editing commands to be parameterised by movement commands that when executed define a region of the buffer for the editing command to operate over. We can implement such semantics in a similar way to dynamic mappings: we run a new lexer with the input key sequence to find the sequence of actions an input sequence maps to, and then use that to construct a composite action.

Extracting actions from the table is as before. For example, if we take the Vi or Vim key sequence `d4j` (delete four lines down), we then run a new lexer with `d4j` as input, returning a list of actions bound to `d4j`.

We can now take these actions and use them to construct a new expression of type `Action` to find the range of the buffer a command is to operate on. The following expression uses `as`, the actions bound to `d4j`, to calculate the start and end position in the buffer of a movement command:

```
do p <- getPointE
  foldr (>>) noPE as
  q <- getPointE
  when (p < q) $ gotoPointE p
  return (p,q)
```

We save the current point, and force the actions retrieved from the lexer. This yields a new point in the buffer. Then, we move back to where we started, and return a pair defining a buffer range. Finally, we can call the editor’s deletion primitive with the range as an argument:

```
deleteNE (max 0 (abs (q-p) + 1))
```

We are thus able to implement parameterised bindings via lexer table elements that are defined via the results of running other elements of the lexer table.

### 5.11 Monadic Lexer Switching

One problem that we have not approached yet is how to implement lexer switching where the switch is based on the result of evaluating an element of the lexer table—that is, an `Action`.

Switching lexers based on the lexer input is easy—it’s a pure operation, and we can use lexer meta actions to transfer control (and even pass control to lexers threaded through the lexer state). However, there are some features that require a mode switch based on the outcome of an editor action. That is, we need to pass control to a specific lexer based on the value returned when evaluating an element found in the current lexer table.

An example of such an action is text searching. Applications often provide a search function. If a user types `/` in many editors they enter a line editor submode for regular expression searching. On entering a complete search phrase, a buffer search takes place. If a search successfully finds a match, the editor displays a prompt and switches to a sub lexer where the user can answer *yes* or *no* to trigger individual text replacements. If there is no match, an error message is displayed, and control continues as normal.

The problem, then, is how to specify that if we find a match (the result of an IO event), control should be transferred to the search key binding lexer. We solve this by introducing the `metaM` combinator, an editor action for lexer continuations. It has the type

```
metaM :: ([Char] -> [Action]) -> IO ()
metaM km = throwDyn (MetaActionException km)
```

where the dynamic exception is defined as

```
newtype MetaActionException =
  MetaActionException ([Char] -> [Action])
  deriving Typeable
```

The `metaM` function connects the elements of the lexer table to the currently running lexer. Such an action allows a table element to cause control to be passed to a new lexer. When a `metaM` action is evaluated, it throws a dynamic exception which wraps the new lexer to which control is transferred.

To actually use this new lexer, we need to catch this exception in the main loop of the application, and replace the current keymap with the thrown keymap. The top-level key input code for Yi is thus:

```
do let run km = catchDyn
    (sequence_ . km <<< getChanContents ch)
    (\(MetaActionException km') -> run km')
  run dflt
```

The simple main loop code is now wrapped in a handler that passes control to a thrown keymap. We begin evaluation of user input with the default keybinding, `dflt`.

The `metaM` action is a powerful function—it allows elements of the lexer table to cause control transfers to lexers specified by the table element. We justify its inclusion by outlining some common editor behaviours only implementable in our framework via `metaM`.

### 5.12 Prompts and User Interaction

A common editor action requiring monadic lexer switch are command prompts. In the Nano editor [23], a user can begin a search by typing `Ctrl-W`. This causes a mode switch to a submode for searching, providing a line editor and some related commands (for continuing, or cancelling the search, for example). On hitting return, the search is performed.

After completing a search, any further searches behave as before, except that the previous search string is used to set a prompt, and a default search value. The default search value is a compiled regular expression produced as an intermediate result of a previous search. The problem is how to script this behaviour.

When switching to search mode we need to check if a previous expression has been searched for. In Yi, the global editor state stores the most recent compiled regular expression, and the string that it was generated from. When we switch to search mode the switch `action` extracts any previous regular expression and uses it to generate a prompt, which then dynamically creates a new lexer with the prompt as its default state. When this action is evaluated, we use `metaM` to switch control to the custom lexer generated by the action.

```
char '\^W' 'action' \_ -> Just (a >>= metaM)
  where
    a = do
      mre <- getRegexE
      let prompt = case mre of
          Nothing -> "Search: "
          Just (pat,_) -> "Search [\"++pat++\"]: "
      msgE prompt
      return (mkKM prompt)
```

Here, `mkKM` constructs a new search keymap via `execLexer`, with the prompt as its default state. In this way we are able to program lexer mode switches based on the result of monadic actions, enabling us to implement many common interface features. In effect we have lexers that build new lexers on the fly, whose elements are actions that trigger switches between lexers.

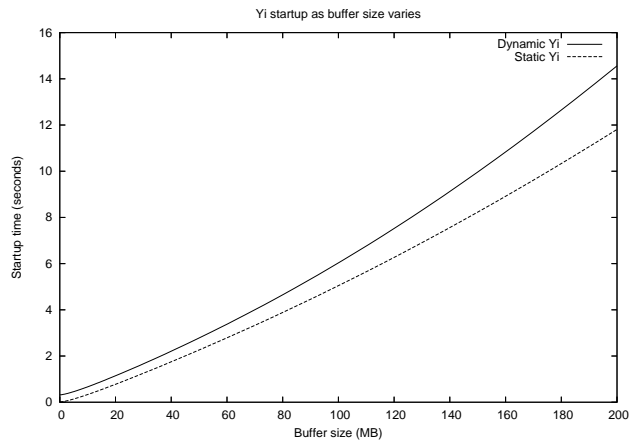


Figure 4. Dynamic architecture startup cost

## 6. Performance

We discuss the performance of our dynamic architecture by way of results obtained with the Yi editor. All measurements were made on a Pentium-M 1.6Ghz laptop running OpenBSD 3.7, with 256M RAM.

**Startup time.** We first evaluate the startup cost of the dynamic architecture by comparing the startup time for the Yi editor with and without the dynamic architecture in Figure 4. Caches were primed with a dummy run before each run.

We instrumented Yi to measure the time from entering the static main until the editor main loop, located in the dynamic application, commences. The startup time increases linearly with the buffer size (from the cost of loading the buffer into memory). The results demonstrate that the dynamic architecture has a *startup cost of around 30 milliseconds* on the benchmark platform for typical editing jobs.<sup>6</sup> In other words, the cost of loading buffers by far exceeds the startup costs of the dynamic architecture. As the buffer size increases, factors unrelated to dynamic linking come into play, in particular, after around 100M the amount of real memory has an impact as the machine begins to swap.

**Hot code swapping.** We display the costs of state-preserving hot code swapping in Figure 5. The displayed time includes hot swap the entire application code, reinjecting the preserved state, and returning to the dynamic main loop.

The results indicate a *constant cost of 10 milliseconds for hot swapping*, independent of the Yi state size (the state increases as the buffer size increases), up to buffer sizes of around 120M, at which point the test machine starts to run out of physical memory, and performance rapidly deteriorates as hot swapped code competes for space with the large buffer state. The cost of hot swapping is independent of the size of Yi buffers as we pass references to buffers to the static core, not buffers themselves.

**Comparison with other editors.** To evaluate Yi's overall performance, we compare it to a range of common editors in Figure 6. The benchmark involves the execution of a specific editing sequence submitted to the keystroke interfaces of each editor. We then test performance as buffer size increases. The measured time includes editor start, GUI initialisation, execution of the edit sequence, and editor shutdown. It might have been preferable to mea-

<sup>6</sup> We define *typical* as file under 20M in size. The average source file in the Haskell hierarchical libraries is 10k; so, 20M is a generous limit.

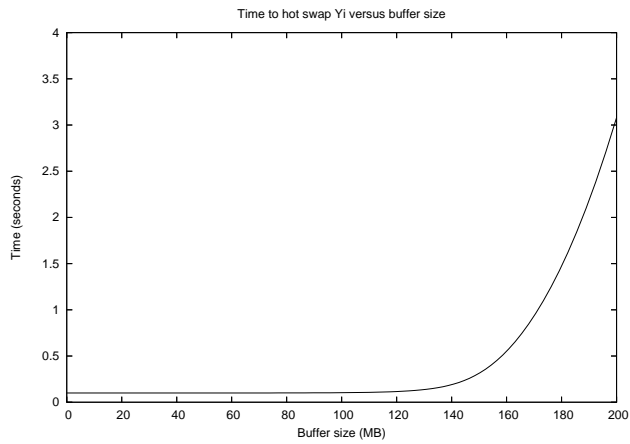


Figure 5. Performance of hot swapping

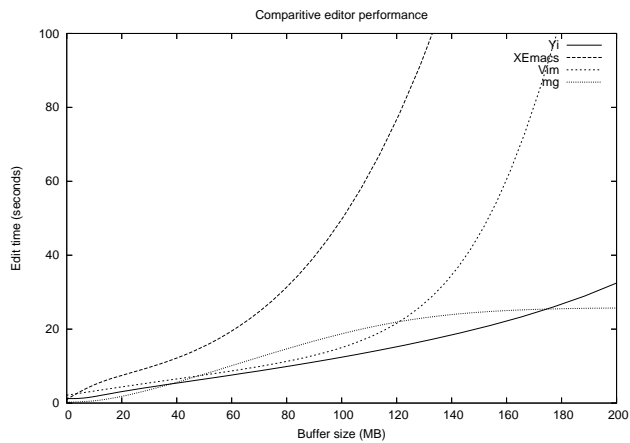


Figure 6. Comparative editor performance

sure each operation in isolation, but the interface of some editors does not support such isolated measurements.

It is interesting to observe that Yi (both dynamic and static) outperforms the mature extensible editors Vim and Emacs. It seems as if dynamic applications consisting of compiled functional code have a significant performance advantage over interpreted extensible systems. The small non-extensible editor mg also had good performance. We also tested several other small static editors (nano, ee and vi). They have good performance on small data sizes, but this advantage deteriorates rapidly as file sizes increase. We have not included them in the figure to improve readability.

## 7. Related Work

**Emacs.** Emacs [27] was the first editor whose design revolved around facilitating extensibility. This implied an extension language beyond simple editing macros and the ability to dynamically load extension code that can add new or override existing functionality. Limitations of the hardware and operating systems of the time, encouraged interpretation of extension code, and thus, required the most performance-sensitive functionality to be included in the static core (which is implemented in C in current flavours of Emacs). Stallman [27, Section 4], in fact, explicitly lists dynamic linking as an alternative approach, but dismisses it as impractical

due to lack of operating system support. Moreover, Emacs preloads the most essential Lisp byte-code at build time and dumps the resulting runtime image into a binary file for faster startup.

**Erlang.** Hot code swapping (aka dynamic software updating) is an important feature of Erlang [2, 3, 11, 18], which has been part of the language for several years. Hot swapping in Erlang is used, for example, in telephony applications that must have very high availability. Recent work has looked at formalising the hot code swapping capabilities of Erlang [6]. Hicks [12] cites Erlang’s functional nature as contributing to the simplicity of its hot swap implementation, making reasoning about global state much simpler.

**Java and Eclipse.** The architecture of the Eclipse workbench revolves around a hierarchy of dynamically loaded plugins [7]. At the core is the *Eclipse platform* together with a number of core plugins, which are always available. In recent versions of Eclipse, the core is close to minimal, permitting the use of the Eclipse plugin architecture for non-IDE applications. However, in contrast to Lisp applications and our proposal, Eclipse configuration is via XML meta data. Moreover, there seems to be no simple facility for hot swapping code. Hot swapping has also been used in the context of Java to implement dynamic profiling [10].

**Linux and other operating systems.** The Linux kernel can be compiled such that most of its functionality is in kernel modules than can be loaded and unloaded at runtime. When fully modularised, only functionality that is crucial during booting the kernel itself, such as memory management and process scheduling, are in the static core. Everything else, including support for the file system used for the root partition, is loaded at runtime; the root file system module comes from an initial RAM disk. Linux’ level of modularisation is limited as it began as a monolithic operating system with support for kernel modules added much later. In contrast, *micro kernel* operating systems provide a higher degree of modularisation and, in some cases, hot swapping capabilities [4, 5].

## 8. Further work

There still remains much work to be done in the area of extensible, statically-typed applications. Extensions to Yi and LambdaBot, for example, must currently be loaded as compiled plugins. In the case of Yi, the time taken to compile quick scripting jobs—such as evaluating a simple Haskell expression—is tedious when the code is immediately discarded once a result is produced. An alternative would be to embed a Haskell interpreter in the application and use that for small scripting jobs, whilst retaining compiled code for larger, more permanent application extensions. Haskell users already follow this pattern, writing throw-away code in an interpreter such as GHCi, but resorting to compilation for more long-lived code. Ideally we would like to be able to *mix* compiled code with interpreted code. To this end, we plan to integrate the GHCi interpreter, with its ability to mix compiled and interpreted code [25], into Yi, effectively adding a form of interactive runtime meta-programming.

When extending the state of a dynamic application, consumers of the state, in particular the state transfer functions (which must preserve state over changes in the state type) need to be modified to handle the new state type. We speculate that a state design based upon extensible generic programming may reduce the cost of state extension and allow extensible state transfer functions. Recent work on open, extensible, functions [21] seems promising.

When extending the state type whilst hot swapping, we are currently forced to step outside of the type system—by translating the state value into a binary representation—and then reconstructing this value after the swap. Ideally, we would like to be able to auto-

matically extend the state type, without requiring the binary encoding or requiring a new state type. Extensible records seem suitable in this regard [19]. The transfer of values as the state type is extended is achieved via a transform from a type  $\tau$  to the same type augmented with new record fields. The result would be that preserving state over extensions to the state type can be typed inside the language without resorting to untyped binary intermediaries.

We avoid a large number of semantic concerns by hot swapping all dynamic code, which obviates the need to worry about references into old code persisting in hot swapped code. However, more fine grained hot swapping may be desirable in some applications. We intend to investigate further issues relating to fine grained hot swapping in lazy languages.

Beyond the general research issues raised by this paper, there are a number of features we would like to see developed in Yi, including

- parser-based syntax highlighting and language-aware editing,
- support for embedding other Haskell applications in the editor, such as refactoring support or class derivation tools, and
- compiler support whilst editing; e.g., for incremental typing.

## 9. Conclusion

We presented an architecture for *fully dynamic* applications in Haskell, based upon a minimal static core. We discussed a practical design and implementation of hot swapping for such dynamic applications, and demonstrated the feasibility of this architecture with two applications, Yi and Lambdabot. We then explored sophisticated dynamic application configuration and extension via an EDSL, as implemented in Yi for dynamic interface extensions. Finally, we discussed some performance benchmarks that suggest that the costs of the dynamic architecture and hot swapping are negligible, and also that dynamic applications written in modern compiled functional languages can have excellent performance.

**Acknowledgements.** We are grateful to Gabriele Keller, Stefan Wehr and Simon Winwood for feedback on drafts. The development of Yi benefited from several discussions with Tuomo Valkonen, in particular the monadic lexer switching model.

## References

- [1] Internet Relay Chat (IRC) help archive. <http://www.irchelp.org/>, 2005.
- [2] J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, 1996.
- [3] J. L. Armstrong. The development of Erlang. In *International Conference on Functional Programming*, pages 196–203, 1997.
- [4] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Technical Conference*, pages 279–291. USENIX Association, 2005.
- [5] A. Baumann, J. Kerr, J. Appavoo, D. Da Silva, O. Krieger, and R. W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, Australia, Apr. 2005. To appear.
- [6] G. Bierman, M. Hicks, P. Sewell, and G. Stoye. Formalizing dynamic software updating. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE)*, April 2003.
- [7] A. Bolour. Notes on the Eclipse plug-in architecture. [http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin\\_architecture.html](http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html), 2003.
- [8] Bram Moolenaar. The Vim Editor. <http://www.vim.org/>.
- [9] M. M. T. Chakravarty. Lazy lexing is fast. *Fourth Fuji International Symposium on Functional and Logic Programming*, LNCS 1722:68–84, 1999.
- [10] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150, New York, NY, USA, 2004. ACM Press.
- [11] B. Hausman. Turbo Erlang: Approaching the speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer, Dordrecht, 1994.
- [12] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.
- [13] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.
- [14] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
- [15] Hugh Mahon. The Editor 'ee'. <http://mahon.cwx.net/>.
- [16] R. Ierusalimsky, L. H. de Figueiredo, and W. Celes. The evolution of an extension language: A history of Lua. In M. A. Musicante and E. H. Hausler, editors, *V Simpósio Brasileiro de Linguagens de Programação*, pages B–14–B–28, Curitiba, May 2001.
- [17] R. Ierusalimsky, L. H. de Figueiredo, and W. C. Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.
- [18] E. Johansson, M. Pettersson, and K. Sagonas. A high performance Erlang system. In *PPDP '00: Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 32–43, New York, NY, USA, 2000. ACM Press.
- [19] M. P. Jones and S. Peyton Jones. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell Workshop*. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, Sept. 1999.
- [20] S. P. Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 25–36, New York, NY, USA, 1999. ACM Press.
- [21] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. Draft, submitted to ICFP 2005. <http://www.cwi.nl/~ralf/syb3/>, Apr. 2005.
- [22] D. Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.
- [23] Nano Core Development Team. GNU Nano Text Editor. <http://www.nano-editor.org/>.
- [24] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging Haskell In. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 10–21. ACM Press, 2004.
- [25] J. Seward, S. Marlow, A. Gill, S. Finne, and S. P. Jones. Architecture of the Haskell Execution Platform (HEP). <http://www.haskell.org/ghc/docs/papers/>, 1999.
- [26] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.
- [27] R. M. Stallman. EMACS: the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, 1981.
- [28] Sven Verdoolaege and Keith Bostic. The Berkeley Vi Editor. <http://www.bostic.com/vi/>.
- [29] L. Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.
- [30] M. Wallace and C. Runciman. The bits between the lambdas: binary data in a lazy functional language. In *ISMM '98: Proceedings of the 1st International Symposium on Memory Management*, pages 107–117, New York, NY, USA, 1998. ACM Press.