

Experience Report: Scheme in Commercial Web Application Development

Noel Welsh

School of Computer Science
University of Birmingham
Birmingham
B15 2TT
UK

nhw at cs dot bham dot ac dot uk

David Gurnell

Untyped Ltd
111 Durley Dean Road
Birmingham
B29 6RY
UK

dave at untyped dot com

Abstract

Over the past year Untyped has developed some 40'000 lines of Scheme code for a variety of web-based applications, which receive over 10'000 hits a day. This is, to our knowledge, the largest web-based application deployment of PLT Scheme. Our experiences developing with PLT Scheme show that deficiencies in the existing infrastructure can be easily overcome, and we can exploit advanced language features to improve productivity. We conclude that PLT Scheme makes an excellent platform for developing web-based applications, and is competitive with more mainstream choices.

Categories and Subject Descriptors D. Software [D.1 Programming Techniques]: D.1.1 Applicative (Functional) Programming

General Terms Economics, Languages

Keywords Web, Scheme

1. Introduction

Languages such as Java, Python, and Ruby are popular for web-application development in large part due to their well developed libraries that target this domain. Modern functional programming languages have powerful features that make them attractive to developers but these same languages typically have many fewer libraries than mainstream alternatives. A developer choosing a functional language is gambling that the cost of library development is lower than the benefits accrued from a more expressive language. Our experience with PLT Scheme shows this is the case.

Over the last year we have delivered web-based applications to the School of Biological and Chemical Sciences, Queen Mary University of London (QMUL) to produce customisable staff and course home pages, record coursework marks and attendance, record student registration information, and handle incoming student applications. The combined applications receive about 10,000 requests per day in peak periods. All server-side code, some 40,000 lines, is written in PLT Scheme, with some client-side (web browser) code using the Flapjax functional-reactive framework for

Javascript (Meyerovich et al. 2007). We believe we are the first people to deploy a web application using PLT Scheme on such a scale.

2. Architecture of the QMUL software

The high level architecture for the QMUL software is a pipeline, a fairly standard structure for a web application. The pipeline starts with a HTTP request received from a web browser, and ending with an HTTP response send to the browser. A typical pipeline consists of the following stages:

1. choose computation: resume a previous one, or start a new one;
2. authentication;
3. extraction and parsing of form data;
4. validation;
5. query database;
6. process query results;
7. generation of continuations;
8. rendering user interface.

We have developed two libraries to implement most of this functionality. Lylux, with support from the PLT web server, implements all the stages that do not involve the database, whilst Snooze handles all the database interaction.

2.1 Architecture of Lylux

The main abstractions in Lylux are *controllers*, *pipelines*¹, and *components*.

A *controller* is the start point for a web computation. Each distinct action in an application (for example, updating course information) is associated with a controller, and each controller is associated with a unique URL and a unique processing pipeline. Controllers and pipelines are first class values so they can be composed in the usual ways. For example, the following code defines a controller called *update-course*, which is a function of a single argument, and associates a pipeline that allows only superuser access.

```
(define-controller update-course
  (make-request-pipeline #t allow-superusers)
  (lambda (course)
    ...))
```

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.

ICFP'07, October 1–3, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-815-2/07/0010...\$5.00

¹ Note that a pipeline in the Lylux sense starts processing after a computation is chosen in step 1 of the list above.

Pipelines allow sharing of common preprocessing steps between controllers, and querying of pipeline stages. For example, code embedding a link to a controller can query the controller's pipeline for its security predicate and insert a greyed-out link if the current user is not able to access the controller. A pipeline is simply a list of pipeline stages. For example:

```
;; make-request-pipeline : boolean (→ boolean) → (list-of stage)
(define (make-request-pipeline use-idcheck? security-predicate)
  (list init-log-stage
        catch-exn-stage
        ... ;; Other stages
        (make-security-stage security-predicate)
        log-entry-stage))
```

A pipeline stage is in turn a name and a function of any number of arguments: a success continuation (which does not return) plus any arbitrary number of additional arguments unique to a given pipeline. We use this continuation passing style so pipeline stages can install values, such as exception handlers and parameters, in the dynamic extent of the succeeding stages.

```
;; init-log-stage : pipeline-stage
(define init-log-stage
  (make-stage
   `init-log
   (lambda (success . args)
     (with-log-preamble
      default-log-preamble
      (apply success args))))))
```

Having defined a controller and a pipeline we can install a controller at a specific URL using code such as:

```
(define dispatch-table
  (list
   ... ;; Other controllers
   (make-rule
    update-course
    (make-pattern root-uri "/admin/courses/edit/"
                  (course-arg))))))
```

Note that the URL includes a *course-arg*, which associates a fragment of the URL with a course, automatically retrieving the URL fragment to a course when starting the controller, and automatically converting a course to a URL fragment when a link to the controller is embedded in a page. We dispatch on the request URL with a single call to the *dispatch* function.

Typically, code within a controller assembles a page from a number of components, and renders the page with a call to *respond/page*. A component is a coherent and reusable unit of functionality which usually encapsulates some state. A page is the primary container component, and we have developed a number of widget components such as forms, text boxes, and lists. Each component is an object, written in PLT Scheme's object system (Flatt et al. 2006), and implementing an interface with methods to render a component to HTML, determine if the component has changed since the last rendering, and update internal state from a new request. State is stored in web cells (McCarthy and Krishnamurthi 2006), which interact with continuations in a sensible manner. With a slight modification to the web cell API we are able to automatically determine if a value has changed since the last rendering.

2.2 Architecture of Snooze

The principle abstraction in Snooze is the *persistent structure*, which extends PLT Scheme's structure definitions with database storage types. For example, a University course might be defined as follows:

```
(define-persistent-struct course
  ((code type:symbol)
   (name type:text)
   (credits type:integer)
   (rating type:real)
   (active type:boolean)
   (start type:time-tai)))
```

Creating a persistent structures and saving it to the database is a few lines of code. Snooze also provides an SQL-like query languages, implemented as a combinator library. For example, to find all courses ordered by name we can write:

```
(s:select (s:alias 'a entity:course)
          (s:order (s:asc 'a 'name))
          (s:limit limit offset))
```

Query results are represented as *generators* (Liskov 1996), functions of no arguments that yield a new result from the query on each application. Generators are memory efficient, and we have developed the usual iterator functions (*map*, *fold*, and so forth) to make working with them as easy as working with lists.

We presently support the PostgreSQL and SQLite database engines, generating SQL customised to each.

3. Library Support

Our software relies on many other libraries in addition to Lylux and Snooze. The PLT Scheme *web server* (Krishnamurthi et al. 2007) is perhaps the most critical library we use. It stands between the web browser and our application code, and provides an API that hides the complexity of parsing HTTP requests and generating HTTP responses. More importantly it supports continuation based web applications, which provides several advantages we discuss later.

Many other libraries in the PLT Scheme distribution are used in our daily coding. The principle ones are:

- the *net* library, which provides functions to encode and decode `form-urlencoded` data, and make HTTP GET and POST requests;
- keyword and optional-argument functions, implemented as macros, allow us to provide flexible interfaces to functions;
- a number of SRFIs (Sperber et al. 2007), particularly string and date libraries, *cut*, and eager comprehensions;
- pattern matching, based on (Wright and Duba 1995);
- the contracts library (Findler and Felleisen 2002), used to enforce invariants in our programs.

Two external libraries are crucial to our system. The SPGSQL driver for the PostgreSQL database (Culpepper 2007) provides the back end we currently use to connect Snooze to our database. The SSAX XML parser (Kiselyov and Lisovsky 2002) is used to process XML data we receive from external systems.

4. Advantages of PLT Scheme

We were motivated to use PLT Scheme by the rich set of abstractions at our disposal. We now analyse how these abstractions are used in the software system we have described above.

Continuations allow us to program without the inversion of control symptomatic of conventional web development. The advantages are well documented; see (Krishnamurthi 2003) and (Bridgen et al. 2003) for example. The primary benefit is that we completely remove a class of bugs brought about by the manual continuation-passing style transformation forced upon conventional web applications.

Bookmarking URLs can be an issue in continuation based web applications. The URL contains a key to a continuation stored on the server. Continuations consume memory on the web server, and so they must be expired at regular intervals to ensure bounded memory consumption. This means bookmarked URLs will become stale with time. Recall that we associate each controller with a unique URL. If we receive a request for an expired continuation we simply redirect the user to the starting point of the controller associated with the URL. This provides an acceptable solution. Choosing when and which continuations to expire is a more difficult problem. We discuss our experiences in Section 5.

First class functions are a natural accompaniment to continuations. “Request handler” procedures can be embedded directly in links in HTML: the web server dispatches requests to the appropriate handler (Hopkins 2003). We also find many other uses for first class functions. For example the combinators in Snooze’s query language provide better abstraction than if we were to use strings, and prevents SQL injection by correctly quoting all data in the query.

Parameters are a form of thread- and continuation-local dynamically scoped variable (Moreau 1998) that provide us with a convenient way to pass configuration and request data to an application. This prevents unwieldy argument lists, and race conditions that would result from using global variables.

Hygienic macros provide extendable syntax (Dybvig 1992). Macros are hugely important in our programming. For example, we extend the PLT Scheme object system (itself written using macros) in Lylux to include special syntax for declaring cells and child components, and we use this information to automatically propagate changes up the component hierarchy. The *define-persistent-struct* form in Snooze is also implemented as a macro.

PLT Scheme’s *module system* (Owens and Flatt 2006) supports both internal linking and external linking (module parameterised by other modules). The former is standard, but the latter is not found in many languages. Parameterised modules are useful in a number of situations. For example, Snooze is parameterised by the database backend it uses, and the web server can be easily assembled in many different configurations using this technology.

5. Issues Encountered During Development

There were a number of set-backs during development of the QMUL software, which fall into two categories. Firstly, we had a number of issues with libraries that had not been tested up load. Secondly, many of our libraries are missing features due to time constraints.

5.1 Problems Under Load

We have struggled with memory consumption in the PLT web server. Continuations in the web server are freed based on a least recently used (LRU) used policy. Each continuation is assigned a set number of “life points” when it is created. Points decrease over time, and the rate of change increases when memory consumption is high. When there are no remaining points, references to the continuation are dropped and it may be reclaimed by the garbage collector. In most cases this policy keeps our memory consumption stable around 300MB with no adverse effects. However we have come across situations this policy fails to behave well.

Large file uploads are problematic as the web server automatically decodes the entire file into memory. Thus a large file causes all continuations to be freed, and users receive an error messages. This is aggravated by the LRU policy, as the most recently created continuation, the file upload process, is the one that is taking up the most memory. This is an example of a failure of user isolation, a classic problem in operating systems. The file upload user should not be able to impact other users of the system. So we see the in-

teraction of two problems: a poor design choice, and a failure to implement user isolation in continuation management leading to a bad behaviour.

There is a path to fix both problems. PLT Scheme provides a *port* abstraction for file handles, sockets, and other similar devices. The memory consumption of a port is dominated by its small internal buffer and there are efficient port copying procedures. It will be relatively simple, though tedious, to change the web server to present file uploads as ports that can then be decoded straight to disk. PLT Scheme also provides a memory accounting system (Wick and Flatt 2004) which will allow proper user isolation to be implemented in the LRU policy.

We have had another major issue with the web server crashing every two or three days. This was a major concern (and cause of embarrassment), and it took several weeks, with the support of Matthew Flatt, the primary developer of MzScheme, to isolate the fault to the logging module of the web server. What should have been a tail recursive loop consumed stack space by installing a new exception handler with every recursion, exhausting the stack after a few thousand iterations. This error would have been uncovered by any demanding use of the web server; we just had the bad luck to be the first. We lost between two and four weeks fixing this error.

Our earliest versions of Snooze used the SQLite database. We experienced problems with concurrent access to the database under high load, and switched to the PostgreSQL database. This involved adding SSL support to the SPGSQL driver and took several weeks of development.

5.2 Missing Features

We are happy with the architecture of Snooze but there are many features we have not had time to implement, of which the most important are:

- relationships between persistent structures;
- subtyping of persistent structures;
- our current query language is quite expressive but a touch baroque. We would like to reimplement it with a solid grounding in the relational algebra, and add support for additional features such as stored procedures.

Lylux is less mature than Snooze. We are on our second major rewrite of Lylux, and still feel we have not found the right abstractions. In particular the control flow is too complicated and the level of abstraction too low.

OO style event handlers invert the flow of control, making program comprehension difficult with a concomitant increase in debugging time. We would like to implement a functional reactive system along the lines of Yampa (Nilsson et al. 2002) and FrTime (Cooper and Krishnamurthi 2006). Synchronous systems like Yampa are relatively limited in their expressiveness but fast and simple to implement. Asynchronous systems like FrTime are expressive but relatively computationally expensive, and more complex to implement. We expect we would benefit from a hybrid system: AJAX systems are best viewed as asynchronous, but traditional web applications are synchronous. We haven’t had the time to complete a prototype implementation to properly evaluate our needs.

Many parts of a web application follow a repetitive cycle of view/edit/commit pages. For these systems it should be possible to generate most of the code. There is some prior work on this problem (Achten et al. 2004), but we have not had time to fully explore the literature.

6. Comparison to Other Systems

It is clear that by choosing to develop in PLT Scheme we have had to develop deal with issues that would not arise with a more mature platform. However, these costs are a one off. More important is to consider how our fundamental model compares against that offered by other systems. We do this here, comparing against J2EE and Ruby on Rails.

Continuations greatly simplify a number of issues in web programming. Take, for example, the issue of storing data between requests. This is trivial in a continuation based system: the data is bound to a name with the usual scoping rules of the programming language. The Java Servlets and Ruby on Rails solution require the user to set a location, keyed by string, in a framework provided object that lasts for the request, session, or application lifetime. Using strings as keys is prone to misspelling errors, and collisions between common names. Furthermore these bindings are not statically apparent in the code and so not amenable to static analysis. Finally, these solutions do not interact gracefully with the back button.

Snooze has obvious comparisons to Hibernate (Java) and ActiveRecord (Ruby on Rails). We start by noting that Snooze lacks some features found in these other libraries, solely due to lack of time. Both Java and Scheme require that all names are statically known at run-time², and must solve the problem of introducing record names from record definitions. Hibernate solves this problem by generating Java code from a XML specification, which requires the programmer learn a new language and introduces another step to the compilation process. The *define-persistent-struct* macro is a convenient alternative. ActiveRecord has a similarly succinct style, which is possible as Ruby allows run-time code to introduce names, at the expense of compile time checking. Given that we lose no important flexibility with the Scheme system we prefer it over ActiveRecord.

Ruby on Rails has poor support for components. Its primitive system only allows components to return HTML and there is no support for event handlers or other forms of dependency tracking. JavaServer Faces is a more advanced system, providing an object-oriented framework that is comparable to classic OO GUI frameworks and so can be seen as a more developed version of Lylux. We have already noted that we feel FRP style is a superior solution.

7. Conclusions

Our experience developing web applications in PLT Scheme compares favourably to our experience developing with other languages. We have spent much effort developing libraries and fixing errors that would not have been an issue with a more mature platform, but we can use a range of language features not available elsewhere. After an initial one-off ‘startup’ cost we feel this tradeoff has worked in our favour. We have shown that continuation based web applications developed in PLT Scheme are feasible and robust. The basic technology is mature and what is required now is refinement of the libraries. This will take time but the problems are not insurmountable and in many cases straightforward to solve. Our experiences suggest the end result will be a major improvement over presently available solutions.

Acknowledgments

We wish to thank the developers of PLT Scheme, particularly Matthew Flatt and Jay McCarthy, for support and patience in dealing with our many questions, and Matthew Jadud and the anonymous reviewers for their helpful comments.

² We take run-time to be the time after macro expansion time. Macros may introduce new names but run-time code may not.

References

- Peter Achten, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. Arrows for generic graphical editor components. Technical Report NIII-R0416, Radboud University Nijmegen, 2004.
- Michael Bridgen, Noel Welsh, and Matthias Radestock. Scheme in the real world: A case study. In *International Lisp Conference 2003*, August 2003.
- Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006.
- Ryan Culpepper. SPGSQL. <http://planet.plt-scheme.org/>, April 2007.
- R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report TR-356, Computer Science Department, Indiana University, June 1992.
- Robby Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.
- Matthew Flatt, Robby Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2006.
- Peter Walton Hopkins. Enabling complex ui in web applications with send/suspend/dispatch. In *Scheme Workshop 2003*, November 2003.
- Oleg Kiselyov and Kirill Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. In *International Lisp Conference (2002)*, September 2002. URL <http://okmij.org/ftp/papers/SXs.pdf>.
- Shriram Krishnamurthi. The Continue server (or, how I administered PADL 2002 and 2003). In *Practical Aspects of Declarative Languages (PADL'03)*, January 2003.
- Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 2007.
- Barbara Liskov. A history of CLU. In *History of programming languages—II*, pages 471–510, New York, NY, USA, 1996. ACM Press. ISBN 0-201-89502-1.
- Jay McCarthy and Shriram Krishnamurthi. Interaction-safe state for the web. In *Scheme and Functional Programming, 2006*, September 2006.
- Leo Meyerovich, Michael Greenberg, Gregory Cooper, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax. <http://flapjax-lang.org>, April 2007.
- Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, December 1998. ISSN 1388-3690.
- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.
- Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *International Conference on Functional Programming (ICFP)*, 2006.
- Mike Sperber, Francisco Solsona, David Van Horn, Donovan Kolbly, Shriram Krishnamurthi, Dave Mason, and David Rush. Scheme requests for implementation. <http://srfi.schemers.org/>, April 2007.
- Adam Wick and Matthew Flatt. Memory accounting without partitions. In *International Symposium On Memory Management ISMM'04*, October 2004.
- A. Wright and B. Duba. Pattern matching for scheme. Technical Report TX 77251-1892, Rice University, May 1995.