Experience Report: Type-Safe Multi-Tier Programming with Standard ML Modules

Martin Elsman
Department of Computer Science,
University of Copenhagen, Denmark
mael@di.ku.dk

Philip Munksgaard
iAlpha AG, Switzerland
philip@ialpha.xyz

Ken Friis Larsen
Department of Computer Science,
University of Copenhagen, Denmark
kflarsen@di.ku.dk

Abstract

We describe our experience with using Standard ML modules and Standard ML's core language typing features for ensuring type-safety across physically separated tiers in an extensive web application built around a database-backed server platform and advanced client code that accesses the server through asynchronous server requests.

1. Introduction

The iAlpha Platform is an advanced asset management system featuring a number of analytics modules for combining trading strategies suggested by fund managers. Technically, the application is comprised of approximately 40.000 lines of Standard ML and is built around the SMLserver [9, 10, 11] and SMLtoJs [7] compiler frameworks, which, respectively, allows for Standard ML code to be executed in a web server context and in web browsers.

Our approach to type-safe cross-tier communication is simple: We make sure that all communication between a client and the web server happens through a shared module interface, that is, through a Standard ML signature, which is located in a shared file that is part of both the code base for clients (compiled with SMLtoJs) and the code base for the web server (compiled with SMLserver). Whereas the server implementation of a particular function specified by a signature will often access the underlying database, the client implementation will instead operate as a proxy for the server-provided function. But there is a catch! Because the client functionality needs to be asynchronous, the type of the client function will have to be slightly different from the type of the server counterpart. Here type abstraction and parameterised types come to the rescue. By abstracting over function result types in the shared signature, using `where type` constraints, the server code and the client code can provide sufficiently different types for the functions.

So how does the actual communication happen? The underlying serialisation and deserialisation of data is implemented using type-indexed pickle combinators [6, 15], which requires some unfortunate, albeit type-safe, boilerplate programming. A module, which is compiled both for the server and for the client provides the necessary picklers for both the arguments and the result value for each function together with a unique name for the function. Using this shared module, it is now possible, on the server, to implement a module that for each function responds to a request by first deserialising the argument into a value, calls the particular function and replies to the client with a serialised version of the result value. Dually, on the client side, for each function, the client code can serialise the argument and send the request asynchronously to the server. When the server responds, the client will deserialise the result value and apply the handler function to the value. Both for the server part and for the client part, the code can be implemented in a type-safe fashion. In particular, if an interface type changes, the programmer will be notified about all type-inconsistencies at compile time.

To decrease drastically the amount of boilerplate code that a user needs to manage, we have developed a tool, named SMLexpose, which reads the `SERVICES` signature and writes the various boilerplate structures.[1] We shall discuss this tool further in Section 6.

In general, a large number of generic libraries are part both of the client code base and the web server code base. Such code is often functorised and can be tested in isolation from the application using a traditional Standard ML compiler.

In the next section, we briefly present the two Standard ML compiler frameworks involved, namely SMLserver and SMLtoJs. We then present two libraries that are essential for the solution, namely a *deferred library* for managing asynchronous requests on clients and the type-indexed serialisation library that is used both on the server and on clients for serialising and deserialising data. We then demonstrate how the pieces are composed through a concrete example, for which the server exposes to clients a functionality for obtaining daily stock closing quotes for a particular ticker, such as the Carlsberg stock. Finally, we present related work and conclude.

2. The Tools

SMLserver is a web server platform and compiler for Standard ML programs, also called scripts. It provides access to a variety of RDBMSs through an efficient generic implementation of connection pooling. It also supports type-safe data caching and script scheduling. Scripts, which may share common library code, are compiled into bytecode, which is loaded only once but may be executed many times upon client requests. The framework is built around an Apache web server module [16] and provides a multi-threaded execution model, which allows multiple requests to be served simultaneously. Scripts are executed without reference tracing garbage collection. Instead, SMLserver applies region-based memory management [8, 9, 21], which works well, for instance, in a web-server context where scripts run shortly but are executed often.

[1] SMLexpose is hosted at `https://github.com/melsman/smlexpose`.

ML Family Workshop (ML '2018) September 28, 2018, St. Louis, Missouri, USA
Copyright © 2018 Held by the authors.

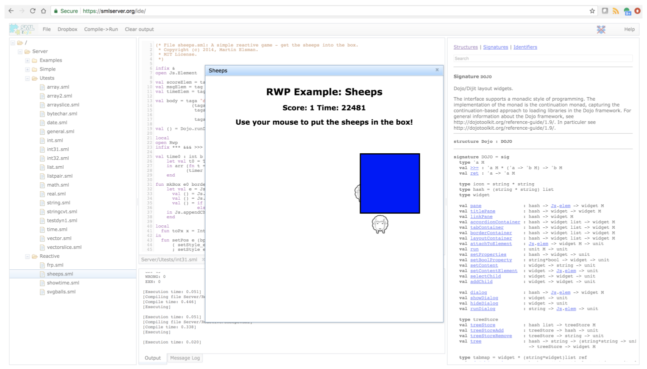

Figure 1. Hosting a Standard ML compiler in a browser.

SMLtoJs is a compiler that compiles Standard ML programs into efficient JavaScript code. In essence, it allows developers to construct client-based web applications in a Higher-Order and Typed (HOT) language. SMLtoJs compiles large programs and features a rich set of libraries, including most of the Standard ML Basis Library [12] and a library for functional reactive programming; it can even compile itself, which provides for a Standard ML compiler working in a browser, as illustrated in Figure 1.
SMLtoJs also supports easy JavaScript integration, which allows for encapsulating a number of JavaScript libraries as Standard ML modules accessible on the client. Such libraries include GUI libraries, such as the Dojo toolkit[2] and various charting functionality from Highcharts.[3]

Both SMLserver and SMLtoJs are built around the MLKit infrastructure, which makes use of a number of crucial optimisation techniques for generating efficient code, including elimination of polymorphic equality [4], heavy function inlining, specialisation of higher-order functions [20], and static interpretation of modules [5].

3. A Deferred Library

Inspired by the OCaml Core Async library [14], it is possible in portable Standard ML to define a module that implements so-called *deferred values*, which are values that may be filled once and that, when filled, may trigger other computations. Important parts of the signature for such a `Deferred` module is given in Figure 2. The module provides standard monadic operations, including `ret` and `>>=`, for composing deferred values, but it also features, for instance, an `all` combinator, which allows programmers to compose deferred values so that clients can make multiple server requests in parallel. The library also handles exceptions properly. If the computation associated with a deferred value results in an exception, the exception is raised when the deferred value is peeked with the `peek` function. Another more proper way for code to react to modifications of deferred values is for the code to listen to changes using the `upon` function. This function takes as arguments a deferred value and two functions, which are executed once the deferred value is filled or when the associated computation results in a raised exception.

One JavaScript object that is directly exposed to Standard ML programmers through a Standard ML structure is the object `XMLHttpRequest`, which is supported by most web browsers. Based on this module and the `Deferred` module, functionality

[2] The Dojo toolkit is available from `http://dojotoolkit.org`.

[3] Highcharts is available from `http://highcharts.com`.

```
structure Deferred : sig
  type 'a t
  val ret    : 'a -> 'a t
  val >>=    : 'a t * ('a -> 'b t) -> 'b t
  val >>|    : 'a t * ('a -> 'b) -> 'b t
  val both   : 'a t * 'b t -> ('a * 'b) t
  val any    : 'a t list -> 'a t
  val all    : 'a t list -> 'a list t
  val throw  : exn -> 'a t
  val upon   : 'a t -> ('a->unit)
               -> (exn->unit) -> unit
  (* Low-level operations *)
  val new    : unit -> 'a t
  val set    : 'a t -> 'a -> unit
  val setexn : 'a t -> exn -> unit
  val peek   : 'a t -> 'a option
end
```

Figure 2. Important parts of the `Deferred` module.

```
structure Pickle : sig
  type 'a pu
  val int    : int pu
  val string : int pu
  val pair   : 'a pu -> 'b pu -> ('a*'b)pu
  val list   : 'a pu -> 'a list pu
  ...
  val pickle   : 'a pu -> 'a -> string
  val unpickle : 'a pu -> string -> 'a
end
```

Figure 3. A simple library for type-indexed serialisation.

for asynchronous HTTP requests can be expressed directly as it is done in the following `Async` module:

```
structure Async : sig
  val httpRequest :
     {method: string,
      binary: bool,
      url: string,
      headers: (string*string)list,
      body: string option}
     -> string Deferred.t
  ...
end
```

The function `httpRequest` takes as argument a record specifying HTTP request information and returns a deferred value, which identifies the asynchronous task. As mentioned, an asynchronous task may be executed in parallel with other asynchronous tasks, using the client's parallel capabilities. Compared to the OCaml Core library, which is implemented using low-level system programming, the SMLtoJs `Async` module is implemented based on browser capabilities.

4. Type-Indexed Serialisation

Another essential functionality is for values at one tier to be serialised into a byte stream and deserialised into an equivalent value at the opposite tier. Such a functionality can be implemented using so-called *pickle combinators* [6, 15], which are type-indexed functions that are composed from basic picklers for base types and combinators, which take other picklers as arguments and produces picklers for structural types, such as tuples. For the example we are going to present here, it suffices to show the simple signature for the pickling library in Figure 3.

Pickled values can be very compact due to the fact that type information need not be serialised. Moreover, the approach supports even cyclic data structures to be serialised (e.g., through `ref`-values). One aspect to mention here is that due to the lack of JavaScript proper tail-calls, the library depends on a number of advanced optimisations being performed by SMLtoJs, including heavy inlining and specialisation of higher-order functions, as mentioned earlier. With these optimisations in place, the `Pickle` library supports very large data structures to be communicated between tiers.

Another issue is the potential inconsistency between different versions of pickle code occuring on clients and servers, for instance during code deployment. The `Pickle` library allows for a representation of the type of the serialized data (or a hash of the type) to be serialised and checked against the type at the receiving end. In this way, consistency can be checked and managed dynamically.

5. An Example: Fetching Stock Quotes

As an example of how cross-tier type-safety is achieved, consider an example where a client needs to fetch stock closing quotes from a server given a ticker symbol identifying the stock. The following `SERVICES` signature, which is shared by the client code and the server code, specifies the service:

```
signature SERVICES = sig
  type 'a res
  type ticker = string and isodate = string
  val quotes : ticker -> (isodate*real)list res
end
```

The signature specifies a function `quotes`, which takes a ticker value and returns a result value of type `(isodate*real)list res`. Notice how the `res` type constructor wraps the actual result type of the function. We shall make use of this parameterisation to support both asynchronous and direct implementations.

Besides from the service signature, the server code and the client code also share a `ServiceDefs` module, which provide type-indexed serialisation code for the services:

```
structure ServiceDefs = struct
  structure P = Pickle
  type ('a,'b)t =
    {id:string, arg:'a P.pu, res:'b P.pu}
  val quotes =
    {id="quotes", arg=P.string,
     res=P.list(P.pair(P.string,P.real))}
end
```

Using the shared code parts, we can now construct the appropriate client code for a proxy for the functionality. The client code is shown in Figure 4. Notice how the `res` type constructor is specified to be the type constructor for deferred values. Also notice that the signature ascription asserts that the types for the pickle values in the `ServiceDefs` module are consistent with the type of the service in the `SERVICES` signature.

A simple server implementation of the service is listed in Figure 5. The `wrap` helper function takes a service definition and a function of appropriate type and returns a function of type `string->string`. The `expose` function extracts the service name from the request header, which is used to identify and call the associated wrapped service. Upon a request to the `service` url, SMLserver may be set up to execute the `expose` function. The function calls the functions `Web.getHeader` and `Web.getRequestData` functions to retrieve information about the request. Using this request data, the proper function can be evaluated and the result send back to the client using the function `Web.returnBinary`.

```
signature CLIENT_SERVICES =
  SERVICES where type 'a res = 'a Deferred.t

structure ClientServices : CLIENT_SERVICES =
struct
  structure P = Pickle
  type 'a res = 'a Deferred.t
  type ticker = string
   and isodate = string
  fun mk (sd: ('a,'b)ServiceDefs.t)
         (arg:'a) : 'b res =
    let val m =
          Async.httpRequest
            {method="POST",
             binary=true,
             url="http://mysite.com/service",
             headers=[("serviceid",#id sd)],
         body=SOME (P.pickle(#arg sd)arg)}
    in Deferred.>>| (m,P.unpickle (#res sd))
    end
  val quotes = mk ServiceDefs.quotes
end
```

Figure 4. Client code for implementing an asynchronous proxy for the server functionality.

```
signature SERVER_SERVICES =
  SERVICES where type 'a res = 'a

structure Services :> SERVER_SERVICES =
struct
  type 'a res = 'a
  type ticker = string
   and isodate = string

  fun quotes ticker =
    [("2018-01-01",23.1)] (* Or: ask DB *)
end

structure ServerExpose : sig
  val expose : unit -> unit
end =
struct
  structure P = Pickle

  fun wrap {id,arg,res} (f:'a -> 'b) =
    P.pickle res o f o P.unpickle arg

  fun expose () =
    case Web.getHeader "serviceid" of
       NONE => raise Fail "missing header"
     | SOME id =>
       let val data = Web.getRequestData()
           val f : string -> string =
             case id of
                "quotes" =>
                   wrap ServiceDefs.quotes
                        Services.quotes
              | _ => raise Fail "no service"
       in Web.returnBinary(f data)
       end
end
```

Figure 5. Server code for implementing the quotes functionality.

The scheme is naturally extended to support and manage simple client-side caching. Often, however, more complex caching is needed, for instance for ensuring that multiple data items are fetched using a single query. We are currently experimenting with various interfaces for facilitating such caching. Orthogonally to caching, authentication and authorisation can be controlled using additional arguments.

6. Automatic Generation of Boilerplate Code

As the critical reader may suggest, the approach is somewhat heavy on writing boilerplate code for picklers. Moreover, when adding a new service function or when altering the type of an existing service function, several files need to be updated.

To decrease this writing and modification of boilerplate code, we have constructed a simple tool, named SMLexpose, which takes as input a `SERVICES` signature from which it generates the `ServiceDefs` structure, the `ClientServices` structure, and the `ServerExpose` structure. With proper integration of SMLexpose in the build process, using simple `make` targets, adding a new service or modifying an existing service now only amounts to creating a type for the service along with a server implementation, after which the service is readily available at the client side.

7. Related Work

There is a large body of related work focusing on tier-less web programming. Such work include ML5 [18], which uses modal logic for controlling code running in different "worlds". Similarly, Links [3], Eliom [19], and Ur/Web [2], compile certain code into JavaScript to execute in a browser and other code into code to be executed on a server. Whereas these systems provide different elegant solutions to constructing multi-tier web applications in a single language, our approach has been to resist the temptation of adding additional domain-specific features to the Standard ML language.

Another line of related work includes compiler frameworks that seek to compile existing languages into JavaScript, such as the Google Web Toolkit [13], O'Browser [1], and earlier work on the Hop language (Scm2Js) [17] by Loitsch and Serrano, which implements tail-calls for their Scheme implementation similar to how tail calls are implemented for SMLtoJs. Another related project is the `js_of_ocaml` project, which compiles OCaml bytecode into JavaScript. Similar to SMLtoJs, this implementation also limits the implementation of tail calls to direct recursive function calls [22].

8. Conclusions and Future Work

We have outlined a simple solution for ensuring cross-tier type-safety in web applications using the SMLserver and SMLtoJs tools. The approach scales well to large applications with many performance critical communication points. There are various possibilities for future work, including proper integrated treatments of caching, authentication, and authorisation. Whereas the SMLexpose tool that we have implemented decreases the amount of boilerplate code needed for serialisation, the simple approach to type-safe multi-tier programming works well even without such a tool. All in all, it turns out that the time for writing the boilerplate code is not dramatic and that the type-safety obtained is invaluable.

References

[1] Benjamin Canou, Vincent Balat, and Emmmanuel Chailloux. O'Browser: Objective Caml on browsers. In *2008 ACM International Workshop on ML (ML'08)*, 2008.

[2] Adam Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 153–165, New York, NY, USA, 2015. ACM.

[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Martin Elsman. Polymorphic equality—no tags required. In *Second International Workshop on Types in Compilation (TIC'98)*, March 1998.

[5] Martin Elsman. Static interpretation of modules. In *Procedings of Fourth International Conference on Functional Programming (ICFP'99)*, pages 208–219. ACM Press, September 1999.

[6] Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[7] Martin Elsman. SMLtoJs: Hosting a Standard ML compiler in a web browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 39–48, New York, NY, USA, 2011. ACM.

[8] Martin Elsman and Niels Hallenberg. A region-based abstract machine for the ML Kit. Technical Report TR-2002-18, Royal Veterinary and Agricultural University of Denmark and IT University of Copenhagen, August 2002. IT University Technical Report Series.

[9] Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.

[10] Martin Elsman, Niels Hallenberg, and Carsten Varming. *SMLserver—A Functional Approach to Web Publishing (Second Edition)*, April 2007. (174 pages). Available via `http://www.smlserver.org`.

[11] Martin Elsman and Ken Friis Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*. Springer-Verlag, June 2004.

[12] Emden R. Gansner and editors John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[13] Google. Google Web Toolkit (GWT). Documentation at `http://code.google.com/webtoolkit/`.

[14] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. *Real World OCaml*. O'Reilly & Associates, 2014.

[15] Andrew J. Kennedy. Functional pearl: Pickler combinators. *Jounal of Functional Programming*, 14(6):727–739, November 2004.

[16] Nick Kew. *The Apache Modules Book: Application Development with Apache (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[17] Florian Loitsch and Manuel Serrano. Hop client-side compilation. In *Proceedings of the 8th Symposium on Trends on Functional Languages*, 2007.

[18] Tom Murphy, VII., Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC'07, pages 108–123, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Gabriel Radanne and Jérôme Vouillon. Tierless web programming in the large. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 681–689, 2018.

[20] Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Second International Symposium on Static Analysis*, pages 366–381, September 1995.

[21] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation (HOSC)*, 17(3):245–265, September 2004.

[22] Jérôme Vouillon and Vincent Balat. From bytecode to JavaScript: the js_of_ocaml compiler. *Softw., Pract. Exper.*, 44(8):951–972, 2014.