Reactive Web Programming with SMLtoJs

Martin Elsman

IT University of Copenhagen

DIKU TOPPS Talk, Nov. 27, 2007

Motivation

Type safe programming on the Web client:

- Compile **full Standard ML** to JavaScript.
- Allow developers to build **AJAX applications** in a high-level statically typed programming language.

Reactive Web Programming:

- Replace the DOM event handler architecture with library support for **behaviors** and **event streams**.
- Allow behaviors to be installed directly in the DOM tree.

Outline of the Talk:

- Features of SMLtoJs (pronounced SML Toys)
- SMLtoJs in action — **a demo**
- The inner workings of SMLtoJs
- A **Reactive Web Programming** library for SMLtoJs
- Related projects

Features of SMLtoJs

Supports all Browsers:

- SMLtoJs compiles Standard ML programs to JavaScript for execution in all main Internet browsers.

Compiles all of Standard ML:

- SMLtoJs compiles **all of SML**, including higher-order functions, pattern matching, generative exceptions, and modules.

Basis Library Support:

- SMLtoJs supports most of the Standard ML Basis Library, including:

```
Array2 ArraySlice Array Bool Byte Char CharArray
CharArraySlice CharVector CharVectorSlice Date
General Int Int31 Int32 IntInf LargeWord ListPair
List Math Option OS.Path Pack32Big Pack32Little
Random Real StringCvt String Substring Text Time
Timer Vector VectorSlice Word Word31 Word32 Word8
Word8Array Word8ArraySlice Word8Vector
Word8VectorSlice
```

Features of SMLtoJs — continued

JavaScript Integration:

- ML code may **call** JavaScript functions and **execute** JavaScript statements.

DOM Access:

- SMLtoJs has support for **simple DOM access** and for **installing ML functions** as DOM event handlers and timer call back functions.

Optimizing Compilation:

- All ML module language constructs, including functors, functor applications, and signature constraints, are **eliminated** by SMLtoJs at compile time.
- Further optimizations include **function inlining** and **specialization** of higher-order recursive functions, such as map and foldl.
- As a result, SMLtoJs generates **fairly efficient** JavaScript code.

SMLtoJs in Action — a Demonstration

Example: Compiling the Fibonacci Function (fib.sml):

```
fun fib n = if n < 2 then 1 else fib(n-1) + fib(n-2)
val _ = print(Int.toString(fib 23))
```

Resulting JavaScript Code:

```
var fib$45 =
 function fib$45(n$48){
  if (n$48<2) { return 1; }
  else { return SmlPrims.chk_ovf_i32(
                  fib$45( SmlPrims.chk_ovf_i32(n$48-1) ) +   // Overflow
                  fib$45( SmlPrims.chk_ovf_i32(n$48-2) )     // checking
                ); };
 };
var __dummy =
 document.write(basis$Int32$.toString$447(fib$45(23)));      // Printing
```

Notice:

Compilation of an **sml-file** or an **mlb-file** (a project) results in an **html-file** mentioning a series of **js-scripts**.

Library for Manipulating the DOM and Element Events

```
signature JS = sig
  type elem                                    (* dom *)
  val getElementById : string -> elem option
  val value          : elem -> string
  val innerHTML      : elem -> string -> unit

  datatype eventType = onclick | onchange      (* events *)
  val installEventHandler : elem -> eventType
                            -> (unit->bool) -> unit
  type intervalId
  val setInterval   : int -> (unit->unit) -> intervalId
  val clearInterval : intervalId -> unit
  val onMouseMove   : (int*int -> unit) -> unit
end
```

Example: Temperature Conversion — temp.sml

```
val _ = print ("<html><body><h1>Temperature Conversion</h1>" ^
               "<table border='1'>" ^
                 "<tr><th align='left'>Temp in Celcius:</th>" ^
                    "<td><input type='text' id='tC'></td></tr>" ^
                 "<tr><th align='left'>Temp in Fahrenheit:</th>" ^
                    "<td><div id='tF'>?</div></td></tr>" ^
               "</table></body></html>")

fun get id = case Js.getElementById id of
      SOME e => e
    | NONE => raise Fail ("Missing id: " ^ id)

fun comp () =
 let val v = Js.value (get "tC")
     val res = case Int.fromString v of
                 NONE => "Err"
               | SOME i => Int.toString(9 * i div 5 + 32)
 in Js.innerHTML (get "tF") res; false
 end

val () = Js.installEventHandler (get "tC") Js.onchange comp
```

Towards Type-Safe AJAX Programming

- By integrating SMLtoJs with **SMLserver**, for server-side Web programming, a service API (a signature) may be implemented natively on the server and as a PROXY on the client.
  - The two implementations may make use of the **same signature** file, which facilitates **cross-tier type-safety**.
- Several serialization possibilities:
  - XML.
  - JSON.
  - Low-bandwidth type-safe serialization using combinators.

The Inner Workings of SMLtoJs

- SMLtoJs compiles SML to JavaScript through an **MLKit** IL.
- SML **reals**, **integers**, **words**, and **chars** are implemented as JavaScript **numbers** with explicit checks for overflow.
- SML **variables** are compiled into JavaScript **variables**.
- SML **functions** are compiled into JavaScript **functions**:

$$[\![\textbf{fn}\ x \Rightarrow e]\!]_{\text{exp}} \;=\; \textbf{function}(x)\{[\![e]\!]_{\text{stmt}}\}$$

- SML **variable bindings** compiles to JS **function applications**:

$$[\![\textbf{let val}\ x\ \textbf{=}\ e\ \textbf{in}\ e'\ \textbf{end}]\!]_{\text{exp}} \;=\; \textbf{function}(x)\{[\![e']\!]_{\text{exp}}\}([\![e]\!]_{\text{exp}})$$

- When compilation naturally results in a JavaScript statement, the statement is converted into an expression:

$$\frac{[\![e]\!]_{\text{stmt}} \;=\; \textit{stmt}}{[\![e]\!]_{\text{exp}} \;=\; \textbf{function}()\{\textit{stmt};\}()}$$

SMLtoJs Issues and Related Work

Future Optimizations:

- **Unboxing** of certain datatypes, such as lists.
- Transform simple recursive functions into **while-loops**.
- Proper implementation of **tail-calls** using trampolines; none of the important JavaScript interpreters implements tail-calls efficiently.

Notice:

- Important JavaScript implementations of the future (such as Adobe's Tamarin JavaScript compiler) deals with tail-calls efficiently.

SMLtoJs Related Work:

- The Google Web Toolkit project (GWT).
- The Scm2Js project by Loitsch and Serrano.
- The Links project. Wadler et al. 2006.
- The AFAX F# project by Syme and Petricek, 2007.

Basic RWP Concepts:

- A **behavior** denotes a value that may change over time:

```
open RWP
val t : Time.time b =
   timer 100   (* time updated every 100ms *)
```

- An SML function may be lifted to become a behavior **transformer**:

```
val bt : Time.time b -> string b =
   arr (Date.toString o Date.fromTimeLocal)
```

- Behaviors of type **string b** may be installed in the DOM tree:

```
val _ = print ("<html><body><h2>Time: <span id='time'>?"
              ^ "</span></h2></body></html>")
val _ = insertDOM "time" (bt t)
```

- **Example behaviors**: mouse position, time, form field content.
- An **event stream** is another RWP concept — mouse clicks...

Reactive Web Programming is based on Arrows:

```
signature ARROW = sig
  type ('b,'c,'k) arr
  (* basic combinators *)
  val arr : (''b -> ''c) -> (''b,''c,'k) arr
  val >>> : (''b,''c,'k)arr * (''c,''d,'k)arr -> (''b,''d,'k)arr
  val fst : (''b,''c,'k)arr -> (''b*''d,''c*''d,'k)arr
  (* derived combinators *)
  val snd : (''b,''c,'k)arr -> (''d*''b,''d*''c,'k)arr
  val *** : (''b,''c,'k)arr * (''d,''e,'k)arr -> (''b*''d,''c*''e,'k)arr
  val &&& : (''b,''c,'k)arr * (''b,''d,'k)arr -> (''b,''c*''d,'k)arr
end
```

Notice:

- The ARROW signature specifies combinators for creating **basic** arrows and for **composing** arrows.
- Specifically, we model **behavior transformers** and **event stream transformers** as arrows.
- The **'k**'s are instantiated either to **B** (for behavior) or to **E** (for event stream).

The RWP library

Building Basic Behaviors and Event Streams:

```
signature RWP = sig
  type B type E (* kinds: Behaviors (B) and Events (E) *)
  type ('a,'k)t
  type 'a b = ('a, B)t
  type 'a e = ('a, E)t
  include ARROW where type ('a,'b,'k)arr = ('a,'k)t -> ('b,'k)t
  val timer     : int -> Time.time b
  val textField : string -> string b
  val mouseOver : string -> bool b
  val mouse     : unit -> (int*int) b
  val pair      : ''a b * ''b b -> (''a * ''b) b
  val merge     : ''a e * ''a e -> ''a e
  val delay     : int -> (''a,''a,B)arr
  val calm      : int -> (''a,''a,B)arr
  val fold      : (''a * ''b -> ''b) -> ''b -> ''a e -> ''b e
  val click     : string -> ''a -> ''a e
  val changes   : ''a b -> ''a e
  val hold      : ''a -> ''a e -> ''a b
  val const     : ''a -> ''a b
  val insertDOM : string -> string b -> unit
end
```

Example: Adding the Content of Fields

Code:

```
open RWP infix *** &&& >>>

val _ = print ("<h1>Add Content of Fields</h1>" ^
               "<input id='a' value='0'/> + <input id='b' value='0'/>" ^
               " = <span id='c'>?</span>")

val si_t : (string,int,B)arr = arr (Option.valOf o Int.fromString)

val form = pair( textField "a", textField "b" )

val t = (si_t *** si_t) >>> (arr op +) >>> (arr Int.toString)

val _ = insertDOM "c" (t form)
```

Notice:

- `t` takes a behavior of pairs of integers and returns an integer behavior.

Example: Reporting the Mouse Position

Code:

```
val _ = print ("<h1>Mouse Position</h1>" ^
               "<span id='mouse0'>?</span><br />" ^
               "<span id='mouse1'>?</span><br />" ^
               "<span id='mouse2'>?</span><br />")
val t : (int*int,string,B) arr =
  arr (fn (x,y) => ("[" ^ Int.toString x ^ ","
                    ^ Int.toString y ^ "]"))
val bm = mouse()
val t10 : (int*int,int*int,B) arr =
  arr (fn (x,y) => (x div 10 * 10, y div 10 * 10))
val bm2 = (t10 >>> t) bm
val _ = insertDOM "mouse0" (t bm)
val _ = insertDOM "mouse1" (calm 400 bm2)
val _ = insertDOM "mouse2" (delay 400 bm2)
```

Notice:

- **calm** waits for the underlying behavior to be stable.
- **delay** transforms the underlying behavior in time.

Implementation Issues

- Behaviors and event streams are implemented using "listeners":

```
type ('a,'k) t =
      {listeners: ('a -> unit) list ref,
       newValue : 'a -> unit,
       current: 'a ref option}
```

- Behaviors (of type `('a,B)t`) always have a *current* value, whereas event streams do not.
- Installing a behavior b in the DOM tree involves adding a listener to b that updates the element using `Js.innerHTML`.
- The implementations of **calm** and **delay** make use of `Js.setTimeout`.
- The implementation of **textField** makes use of `Js.installEventHandler`.
- The implementation of **mouse** makes use of `Js.onMouseMove`.

Related and Future Work

Related Work:

- The **Flapjax** language and JavaScript library by Shriram Krishnamurthi et al.
- John Hughes. Generalising Monads to Arrows. Science of Computer Programming 37. Elsevier 2000.
- The **Fruit** Haskell library by Courtney and Elliott.

Future Work:

- Build a DOM combinator library where elements are behaviors.
- Build a library of high-level composable widgets (list widgets, etc.)
- Build support for controlling (start and stop) basic event triggers.