Hosting a Standard ML Compiler in a Web Browser

Martin Elsman
SimCorp A/S
ML Workshop 2010, Baltimore, MD.
September 25, 2010
Compiling and Running the Game of Life in a Browser

Hosting a Standard ML Compiler in a Web Browser

Martin Elsman

ML Workshop 2010
Outline of the Talk

- Motivation for SMLtoJs — A Compiler from Standard ML to Javascript
- SMLtoJs — Inner workings and examples
- The online version of SMLtoJs
  - http://www.smlserver.com/smltojs_prompt
- Other uses of SMLtoJs
  - A Reactive Web Programming library for SMLtoJs
  - AJAX programming
- Related work
Motivation for SMLtoJs — a Compiler from SML to Javascript

**Higher-Order and Typed (HOT) Web Browsing:**
- Easy development and maintenance of advanced Web browser libraries (e.g., Reactive Web Programming libraries, similar to FlapJax)
- Allow developers to build **AJAX applications** in a HOT language

**Allow for Existing Code to Execute in Browsers:**
- Programs (e.g., SMLtoJs itself — it is itself written in SML)
- Libraries (e.g., The **IntInf** Basis Library module)
- Support all of SML and (almost) all of the SML Basis Library

**Web Programming Without Tiers:**
- Allow the same code to run both in the browser and on the server (e.g., complex serialization code)
- AJAX applications using a single language
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:**
- Support for 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
- **Additional Libraries:** JsCore Js Html Rwp
Features of SMLtoJs — continued

**Javascript Integration and DOM Access:**

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

**Optimizing Compilation:**

- Module constructs, including functors, functor applications, and signature constraints, are **eliminated** at compile time.
- Further optimizations include
  - **Function inlining** and **constant propagation**
  - **Specialization** of higher-order recursive functions (map, foldl)
  - **Tail-call optimization** of so-called **straight tail calls**
  - **Unboxing** of certain datatypes (lists, certain trees, etc.)
- SMLtoJs uses the **MLKit** frontend.
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
    ); }
  }

basis$General$.print$156(basis$Int32$.toString$447(fib$45(23))); // Printing
Interfacing with Javascript

“Native” Javascript code can be executed with **JsCore** module:

```val
signature JS_CORE = sig
  type 'a T
    val unit : unit T
    val int : int T
    val string : string T
    val fptr : foreignptr T
    val exec2 : {stmt: string, arg1: string * 'a1 T, arg2: string * 'a2 T, res: 'b T} -> 'a1 * 'a2 -> 'b
... end
```

Phantom types are used to ensure proper interfacing:

```fun
fun documentWrite d s =
  J.exec2 {stmt="return d.write(s);", arg1=("d",J.fptr), arg2=("s",J.string), res=J.unit} (d,s)
```

SMLtoJs inlines **stmt** if it is known statically; otherwise a **Function object** is created and **stmt** resolved and executed at runtime.
Library for Manipulating the DOM and Element Events

Interaction with the DOM and other Javascript libraries is implemented using the `JsCore` module.

```ml
signature JS = sig
  eqtype win and doc and elem (* dom *)
  val openWindow : string -> string -> win
  val document : doc
  val windowDocument : win -> doc
  val documentElement : doc -> elem
  val getElementById : doc -> 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
```
val win = Js.openWindow "" "height=200,width=400"
val doc = Js.windowDocument win
val elem = Js.documentElement doc
val _ = Js.innerHTML elem
("<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 doc 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
The Inner Workings of SMLtoJs (no tail calls)

- 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:
  \[
  \[\text{fn } x \Rightarrow e\]_{\text{exp}} = \text{function}(x)\{\[e\]_{\text{stmt}}\}
  \]
- SML variable bindings compiles to JS function applications:
  \[
  \[\text{let val } x = e \text{ in } e' \text{ end}\]_{\text{exp}} = \text{function}(x)\{\[e'\]_{\text{exp}}\}(\[e\]_{\text{exp}})
  \]
- When compilation naturally results in a Javascript statement, the statement is converted into an expression:
  \[
  \[e\]_{\text{stmt}} = stmt
  \]
  \[
  \[e\]_{\text{exp}} = \text{function}()\{stmt;\}()\]
Optimizing Straight Tail Calls

- A *straight tail call* is a tail call to the nearest enclosing function with a tail call context containing no function abstractions.
- Example SML code:

  ```sml
  fun sum (n, acc) = if n <= 0 then acc else sum (n-1, acc + n)
  val _ = print (Int.toString (sum (10000, 0)))
  ```

- Generated Javascript code:

  ```javascript
  var sum$45 = function(v$54, v$55) {
    lab$sum:
    while (true) {
      if (v$54 <= 0) { return v$55; }
      else {
        var t$89 = SmlPrims.chk_ovf_i32(v$54 - 1);
        var t$90 = SmlPrims.chk_ovf_i32(v$55 + v$54);
        var v$54 = t$89;
        var v$55 = t$90;  // Argument reassignment
        continue lab$sum; }
    }
  }
  basis$General$.print$156(basis$Int32$.toString$449(sum$45(10000, 0)));
  ```

- No major browser implements tail calls efficiently and the ECMAScript Specification (ano 2009) says nothing about tail calls!
Composing Javascript Fragments

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

- Basis library files are precompiled and available to user programs without recompilation (by reading export bases).
Hosting SMLtoJs in a browser

Compile SMLtoJs sources, including code for a **read-eval loop**, with offline version of SMLtoJs.

During offline compilation, arrange that export **bases for the basis library are serialized**, written into Javascript strings, and stored in js-script files.

Once a browser visits the SMLtoJs online site, the export bases for the basis library are loaded and deserialized.

**NOTICE:**

The serialization and deserialization code is used both by the offline SMLtoJs compiler (when serializing) and by the online SMLtoJs compiler (when deserializing).
**Benchmarks — Running Times**

<table>
<thead>
<tr>
<th></th>
<th>Firefox*</th>
<th>Chromium*</th>
<th>Native**</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib35</td>
<td>36.65</td>
<td>3.93</td>
<td>0.69</td>
</tr>
<tr>
<td>kkb</td>
<td>25.94</td>
<td>2.75</td>
<td>0.28</td>
</tr>
<tr>
<td>life</td>
<td>12.34</td>
<td>1.15</td>
<td>0.48</td>
</tr>
<tr>
<td>simple</td>
<td>88.01</td>
<td>6.66</td>
<td>0.85</td>
</tr>
</tbody>
</table>

(* Running in the browser.

(**) Running in an OS shell.

Measurements done on a Thinkpad T42, 1GB RAM, Ubuntu 10.04.

Firefox version 3.6.10.

Chromium version 6.0.472.53.
### Benchmarks — Compile Times

<table>
<thead>
<tr>
<th></th>
<th>Firefox*</th>
<th>Chromium*</th>
<th>Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib35</td>
<td>0.29</td>
<td>0.04</td>
<td>0.64</td>
</tr>
<tr>
<td>kkb</td>
<td>16.27</td>
<td>2.70</td>
<td>1.59</td>
</tr>
<tr>
<td>life</td>
<td>6.47</td>
<td>1.04</td>
<td>0.85</td>
</tr>
<tr>
<td>simple</td>
<td>70.23</td>
<td>8.24</td>
<td>5.29</td>
</tr>
</tbody>
</table>

#### Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>fib35</th>
<th>kkb</th>
<th>life</th>
<th>simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib35</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kkb</td>
<td>617</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>life</td>
<td>211</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple</td>
<td>1064</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(*) Compilation in the browser.
Other Uses (and Possible Uses) of SMLtoJs

Reactive Web Programming (RWP):

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

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.
- The two tiers may communicate data using low-bandwidth serialization implemented in only one language.
Reactive Web Programming (RWP)

**Reactive Web Programming:**

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

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

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

```sml
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:

```sml
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 (behavior) or to E (events).
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 flatten : ''a b b -> ''a b
  val insertDOM : string -> string b -> unit
end
Example: Adding the Content of Fields

**Code:**

```ml
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:**

```ml
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":

  ```ml
  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 Work

**Related Compiler 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.
- O’Browser (ML’08).

**Related Reactive Programming Work:**
- The Flapjax language and Javascript library by Shriram Krishnamurthi et al.
- The Fruit Haskell library by Courtney and Elliott.