SMLtoJs: Hosting a Standard ML Compiler in a Web Browser

Martin Elsman
SimCorp, Copenhagen, Denmark
martin.elsman@gmail.com

Abstract

Standard ML is a strictly typed functional programming language that provides programmers with many powerful features, including parametric polymorphism, pattern matching, and a rich module system. In this paper, we present SMLtoJs, an optimizing compiler that compiles Standard ML programs into JavaScript to be executed in any JavaScript-supported Web browser. SMLtoJs supports all of Standard ML and most of the Standard ML Basis Library, which allows for the SMLtoJs compiler itself to be compiled and executed in a browser. We present the overall structure of SMLtoJs, including the non-JavaScript aware front-end and the JavaScript-specific backend. We also present SMLtoJs's type safe interface for interacting with native JavaScript, such as the DOM api. Finally, we present the details of the JavaScript-specific optimizations, including how SMLtoJs deals with tail recursion, which enable complex Standard ML programs to be compiled into efficient JavaScript.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]

General Terms Design, Performance

Keywords Standard ML, JavaScript

1. Introduction

Recently, there has been a large number of attempts by compiler writers to target JavaScript to allow for programming languages other than JavaScript to be used for programming Web clients [4, 14, 16, 19, 22]. This paper describes a successful attempt at compiling Standard ML programs into JavaScript. One of the success criteria has been for the compiler itself to run within a Web browser, which has been possible because the compiler itself is written in Standard ML.

There are many good reasons for allowing other languages than JavaScript to be used for developing software for Internet clients. For instance, higher-order and typed (HOT) programming languages may ease development and maintenance of advanced Web browser libraries (e.g., Reactive Web Programming libraries, similar to Flapjax [17]) and allow developers to build AJAX applications (without tiers) in a single HOT language, for which the same language is used for programming the server and the client [4, 14, 19, 22].

Another benefit from targeting JavaScript is to allow for existing program code (developed for non Web browser purposes) to execute in Web browsers. One example is the Standard ML compiler mentioned here (about 80.000 lines of Standard ML), which can be compiled due to full support for the Standard ML programming language and the almost full support for the Standard ML Basis Library [13]. Yet another possibility is to provide access to libraries, written in the source language, from JavaScript. A concrete example is the Standard ML IntInf library that provides support for integer arithmetic for non-bounded integers.

In this paper, we present SMLtoJs, a compiler from Standard ML to JavaScript that uses the frontend and optimization steps of the MLKit Standard ML compiler [1, 21] and translates the intermediate MLKit language into optimized JavaScript. The backend targets a small functional fragment of JavaScript and makes use of a non-trampoline implementation of tail calls for generating efficient JavaScript code.

SMLtoJs provides the following key features:

- SMLtoJs compiles Standard ML programs to JavaScript for execution in all main Internet browsers, including IE, Firefox, and Chromium.
- SMLtoJs compiles all of Standard ML, including higher-order functions, pattern matching, generative exceptions, and modules.
- Support for most of the Standard ML Basis Library [13], including full support for the following modules:

```
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
```

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.
PLASTIC'11, October 24, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-1171-7/11/10. . . $10.00

- ML code may call JavaScript functions and execute JavaScript statements through the special `JsCore` semi-type-safe module.
- SMLtoJs has support for simple DOM access and for installing ML functions as DOM event handlers and timer call back functions.

The paper proceeds as follows. In the following section, we demonstrate, by example, how a simple Standard ML program compiles into JavaScript and outline how multiple fragments of generated JavaScript code are joined together to form complete programs or libraries. In Section 3, we present a semi-type-safe interface for interacting with native JavaScript and for making Standard ML code accessible from JavaScript programs. In Section 4, we present a simple type-safe library for accessing the DOM and related browser functionality from within Standard ML programs and give an example of a use of the library.

In Section 5, we present the architecture and design for the browser-hosted Standard ML compiler. The implementation allows for library code to be shared between the compiler itself and programs compiled in the browser. Technically, the implementation allows for precompiled library code, which may be utilized by compiled programs via deserialization of compactly generated symbol table information [7]. In Section 6, we describe the inner workings of the compiler from an intermediate language of the MLKit compiler to JavaScript. We present the overall translation function and describe in details how tail-calls are compiled into efficient JavaScript.

In Section 7, we compare, across a number of benchmark programs, SMLtoJs execution with native execution. We also investigate execution performance on a selection of browsers. In Section 8, we report on other uses of SMLtoJs. First, SMLtoJs comes with a Flapjax-like Reactive Web Programming library [17], written entirely in Standard ML, but utilizing SMLtoJs's possibilities for accessing a browser's DOM implementation. Much like the Google Web Toolkit (GWT), which compiles Java into JavaScript, SMLtoJs may be used in concert with SMLserver [9] for type safe engineering of AJAX-like applications that are composed of code running on a server and code running in the user's browser.

Finally, in Section 9, we present related work and in Section 10, we conclude. Readers may point their Google Chromium or Firefox browsers to

`http://www.smlserver.org/smltojs_prompt/`

for trying out the compiler, or to

`http://www.smlserver.org/smltojs/`

for finding more information about the SMLtoJs project.

2. SMLtoJs in Action

As a first simple example of SMLtoJs compilation, consider the following Standard ML version of the Fibonacci function:

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

The above Standard ML code is compiled into the following JavaScript code by SMLtoJs:

```
var fib$45 = function fib$45(n$48) {
  if (n$48<2) { return 1; }
  else { return Prims.chk_ovf_i32(
    fib$45( Prims.chk_ovf_i32(n$48-1) ) +
    fib$45( Prims.chk_ovf_i32(n$48-2) ) );
  };
};
General.print$156(Int32.toString$447(fib$45(23)));
```

Notice the explicit checks for overflow, which are required on all integer operations, and the reference to the two Standard ML Basis libraries `General` and `Int32`.

2.1 Composing JavaScript Fragments

For programming in the large, Standard ML programs may be split up in a series of files, which again may be organized, by reference, in ML Basis Files (MLB files) [3, 8]. SMLtoJs takes as input a single `sml`-file or an `mlb`-file and generates a series of `js`-scripts and an `html`-file that loads the `js`-scripts; see Figure 1.

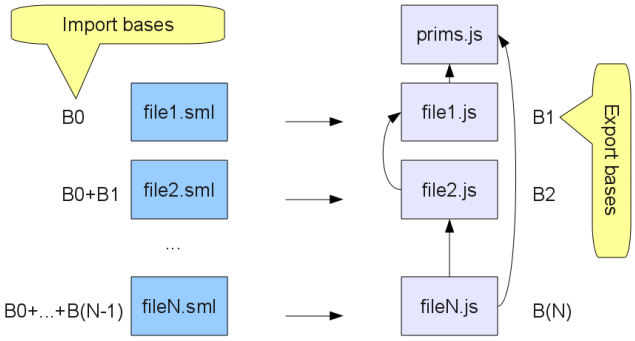

Figure 1. Composing JavaScript Fragments.

Each `sml`-file is compiled with respect to a certain import basis (i.e., symbol table) and results in a `js`-script and an export basis (a symbol table for the exported identifiers). The initial import basis (i.e., B_0) contains bindings for all basic primitives for which JavaScript is either in-lined when referred to or appears in the `prims.js` file, which contains a series of primitive hand-written JavaScript functions.

Symbol tables in the framework (i.e., bases) are implemented as functional persistent data structures based on so-called Patricia trees [18], which, once loaded, may be merged in almost constant time.

The implementation supports precompilation of libraries, which are made available to programs, without recompilation, by reading export bases. For detailed information about the separate compilation framework used by SMLtoJs, the reader is referred to [8].

3. Interfacing with JavaScript

Native JavaScript code can be executed with the `JsCore : JS_CORE` module:

```
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 ensure proper interfacing. Here is an example use of the `JsCore.exec2` function:

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

Here the execution of the statement `stmt` occurs in a context where `d` is bound to the first parameter of `documentWrite`, an object of type `foreignptr`, and `s` is bound to the second parameter, a value of type `string`. In this case, since the `stmt` string is known statically, the statement will be inlined as JavaScript code by SMLtoJs. When the string is not known statically (e.g., if it is the result of a dynamic string concatenation), a JavaScript function object is created and the statement will be resolved and executed at runtime.

The `JsCore` module contains a series of `exec`N definitions that allow for execution of statements with other than two parameters. The module also contains library functions for reading and writing JavaScript object properties.

A natural way of using the `JsCore` module is to limit its use to isolated and specially trusted modules containing typed wrapper functions for JavaScript executable statements and functions. Following this style of interaction with JavaScript, application programmers should not make use of the `JsCore` functionality, directly from within application code.

4. The Document Object Model API

Interaction with the DOM and other browser functionality is implemented using the `JsCore` module. SMLtoJs implements most of the functionality in the `Js : JS` module, for which the signature is shown in Figure 2.

4.1 Example: Temperature Conversion

Figure 3 lists an example client application for converting temperatures in degrees Celsius into Fahrenheit.
The first part of the code inserts, in a newly opened window, HTML containing a table with an input field (id='tC')

```
signature JS = sig
  (* dom *)
  eqtype win and doc and elem
  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

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

Figure 2. The `JS` signature.

and a `div`-element for output (id='tF'). Using the function `Js.installEventHandler`, the program executes the `comp` function whenever there is a change in the input field. Once called, the `comp` function will compute a new Fahrenheit temperature value and store it in the output `div`-element. When compiled and running in a browser, the example shows as displayed in Figure 4.

An important observation here is that the Standard ML code listed in Figure 3, is not much different than how it would appear in plain JavaScript. Later in Section 8.1, we shall see how reactive programming with SMLtoJs allows for programming without explicit side-effecting event handlers.

5. Hosting SMLtoJs in a Browser

In this section, we describe how SMLtoJs itself can be made to run in a browser, as seen in Figure 5.

Because SMLtoJs itself is written in Standard ML, all SMLtoJs sources, including code for a read-eval loop, and code for a simple IDE is compiled into JavaScript with the offline version of SMLtoJs.

During the offline compilation, it is arranged that export bases for the basis library are serialized and written into compact JavaScript string declarations in separate `js`-script files. Now, when a browser visits the SMLtoJs online site, the export bases for the basis library are computed by first loading the `js`-script files and then deserializing the contained strings. Notice that the serialization and deserialization code is used both by the offline SMLtoJs compiler (when serializing) and by the online SMLtoJs compiler

```
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 Celsius:</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
```

Figure 3. Standard ML code for a complete temperature conversion application.

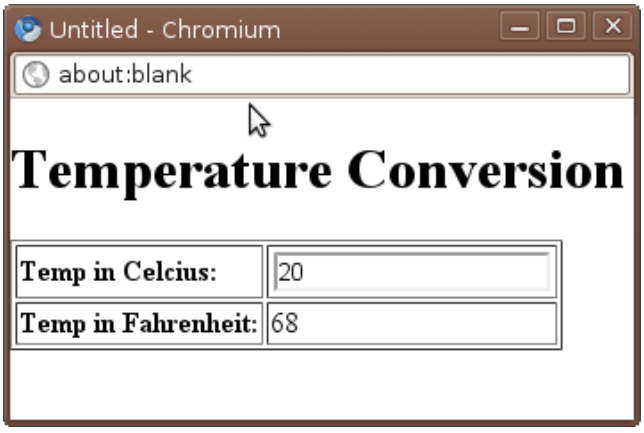

Figure 4. Running the compiled temperature example in a browser.

(when deserializing). The serialization format is highly specialized and supports recursive data structures with a high degree of sharing and compactness [7]. An important point to be made here is that the format can be kept local to the Standard ML serialization module; no other code needs to know the details of the format.

6. Optimizing Compilation

The SMLtoJs compiler makes use of the MLKit frontend, which compiles Standard ML source code into an explicitly, polymorphically typed intermediate language. Making use of the MLKit frontend has a number of benefits.

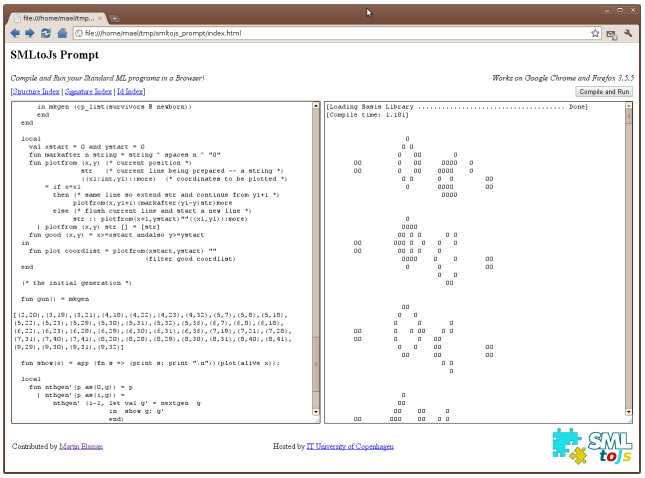

Figure 5. Compiling and running the Game-of-life in a browser.

First, all module constructs, including functors, functor applications, and signature constraints, are eliminated at compile time [6]. Second, pattern matching is compiled into simple case-expressions on fundamental datatypes. Moreover, a number of aggressive semantics-preserving optimizations are performed at the intermediate language level, including function in-lining, constant propagation, constant folding, specialization of higher-order recursive functions (e.g., `List.map` and `List.foldl`), and unboxing of certain datatypes (e.g., lists and certain kinds of trees).

6.1 The Source Language

The MLKit intermediate language is an expression language, in the sense that all constructs are expressions. For the sake of presentation, we consider here only the essential part of the MLKit intermediate language. The complete language also has support for mutually recursive functions, exceptions, user-defined data types, tuples and records, and a number of primitive data types and operations, that we shall not consider here.

In the following, we use x, y, and f to range over program variables and d to range over integer constants. We also use *op* to range over a set of primitive operators.

$$\begin{array}{rcl} e & ::= & x \mid d \mid e_1\ op\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{true} \mid \texttt{false} \\ & \mid & \texttt{fn}\ x \Rightarrow e \mid e_1\ e_2 \mid \texttt{fix}\ f\ (x) = e_1\ \texttt{in}\ e_2 \end{array}$$

For the `let` and `fn` constructs, x is bound in e_2 and e, respectively. Moreover, for the `fix` construct, f and x are bound in e_1 and f is bound in e_2. For any expression e, we write $\mathrm{fv}(e)$ to denote the free variables in e.

6.2 The JavaScript Target Language

The SMLtoJs backend compiles the MLKit intermediate language into a JavaScript abstract syntax tree, for which certain simple peep-hole optimizations may be performed before pretty printing. The subset of the translation presented

here targets only a small subset of possible JavaScript constructs:

$$
\begin{array}{rcl}
e & ::= & x \mid d \mid e_1 \; op \; e_2 \mid \texttt{function}(x)\,\{\, s \,\} \\
& \mid & e\,\texttt{?}\,e_1 : e_2 \mid \texttt{true} \mid \texttt{false} \\
& \mid & e_1(e_2) \mid e.x \mid e[e] \mid [e_1, ..., e_n] \\
\\
s & ::= & \texttt{var}\; x; \mid \texttt{var}\; x = e; \mid e_1 = e_2; \\
& \mid & s\; s \mid \texttt{if}\; e\; \texttt{then}\; s_1\; \texttt{else}\; s_2 \mid \texttt{return}\; e; \\
& \mid & \texttt{continue}\; x; \mid x : \texttt{while}(e)\,\{\, s \,\}
\end{array}
$$

The complete translation also targets a number of other constructs, including JavaScript's support for exceptions and makes use of JavaScript's support for multi-parameter functions.

6.3 The Translation

Because programming in a functional language, such as Standard ML makes highly use of recursion to program loops (there are, essentially, no other loop construct), it is important to deal with tail-recursion efficiently. Moreover, none of the major browsers implements tail calls efficiently and the ECMAScript Specification (ano 2009) says nothing about tail calls!

There are a number of useful techniques for compiling tail calls, including the use of trampolines, which provides for a general treatment of tail-calls, but which also incurs a serious overhead, as all calls are implemented as returns to a trampoline function [20].

The translation that we present here implements tail-recursion using JavaScript's continue statement. Although the technique leads to an efficient implementation of a large class of tail calls, only certain tail calls, which we shall call straight tail calls, are dealt with. A *straight* tail call is a recursive call to the nearest enclosing function such that the call does not appear under another function abstraction. This approach is also used in the Hop project [16].

A naïve translation of the source language into JavaScript would translate `let`-constructs into immediately applied function declarations, which would not work well with the suggested approach for compiling tail-recursion. Instead, `let`-constructs are compiled into JavaScript `var`-statements, which again requires closures to be implemented using function properties and access to free variables of a function through the function object. For instance, consider the ML function

```
fun f (n,a) =
  if n = 0 then a
  else let val i = n+1
           in f (n-1, (fn()=>i) :: a)
           end
```

Here we must make sure that the functions accumulated in the list refer to their distinct value for `i`. To ensure that this property holds, we transform the function as follows:

```
function f(n,a) {
 loop:
 while(true) {
  if (n == 0) { return a; }
  else {
    var i = n + 1;
    n = n - 1;
    var g = function() { return this.i; };
    g.i = i;
    a = [g,a];
    continue loop;
  }
 }
}
```

Notice that in the body of g, access to `i` is done through *this*. Moreover, the closure for the function is explicitly created by updating the function property `g.i` at the time the function is created.

The translation makes use of a notion of *continuation*, defined as follows:

$$k \quad ::= \quad \texttt{Ret} \mid \texttt{Ret}(f, x) \mid \texttt{Var}(x)$$

Continuations are used in the translation to specify how the translation should turn a final expression into a statement. For this use, the translation makes use of an auxiliary function for resolving continuations:

$$\text{WrapCont}(k)(e) \quad = \quad \begin{cases} x = e; & \text{if } k = \texttt{Var}(x) \\ \texttt{return}\; e; & \textit{otherwise} \end{cases}$$

The $\texttt{Var}(x)$ continuation specifies that the result of the final expression should be assigned to the variable x. The `Ret` continuation specifies that the result of the final expression should be returned by the enclosing function. The $\texttt{Ret}(f, x)$ continuation specifies that if the final expression is a tail call to f, the argument should be assigned to x and the JavaScript code should continue at label f; when the final expression is not a tail call to f, the result of the expression should be returned by the enclosing function.

The translation also makes use of a function Prim() for translating primitives. Prim() takes as arguments an operator *op* and a series of JavaScript expressions. We shall not completely specify the Prim() function here, but here is the entry for addition on integers:

$$\begin{array}{l} \text{Prim}(\texttt{+}, e_1, e_2) \quad = \\ \quad \texttt{Prims.chk\_ovf\_i32}(e_1 + e_2) \end{array}$$

For presentation, the translation is given as judgments of the form $C \vdash e \Rightarrow e'$ and $k, C \vdash e \Rightarrow s$, where C ranges over sets of variables, e is a source language expression, e' a target language expression, and s a target language statement. Judgments of the form $C \vdash e \Rightarrow e'$ are read: Given a set of closure-stored variables C, the source language expression e may translate into the target JavaScript expression e'. Similarly, judgments of the form $k, C \vdash e \Rightarrow s$ are read:

Given a continuation k and a set of closure-stored variables C, the source language expression e may translate into the target JavaScript statement s.

Expressions $\boxed{C \vdash e \Rightarrow e'}$

$$\frac{x \notin C}{C \vdash x \Rightarrow x}$$

$$\frac{x \in C}{C \vdash x \Rightarrow \mathit{this}.x}$$

$$\frac{}{C \vdash d \Rightarrow d}$$

$$\frac{C \vdash e_1 \Rightarrow e_1' \quad C \vdash e_2 \Rightarrow e_2'}{C \vdash e_1 \mathit{op}\, e_2 \Rightarrow \mathrm{Prim}(\mathit{op}, e_1', e_2')}$$

$$\frac{C \vdash e \Rightarrow e' \quad C \vdash e_1 \Rightarrow e_1' \quad C \vdash e_2 \Rightarrow e_2'}{C \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow e'\ \texttt{?}\ e_1'\ \texttt{:}\ e_2'}$$

$$\frac{C, \texttt{Ret} \vdash e \Rightarrow s \quad \mathrm{fv}(e) \setminus \{x\} = \emptyset}{C \vdash \texttt{fn}\ x \Rightarrow e \Rightarrow \texttt{function}(x)\{\ s\ \}}$$

$$\frac{C \vdash e_1 \Rightarrow e_1' \quad C \vdash e_2 \Rightarrow e_2'}{C \vdash e_1\ e_2 \Rightarrow e_1'(e_2')}$$

$$\frac{C, \texttt{Ret} \vdash e \Rightarrow s}{C \vdash e \Rightarrow (\texttt{function()}\{s\})()}$$

Statements $\boxed{C, k \vdash e \Rightarrow s}$

$$\frac{\begin{array}{c} f \text{ fresh} \quad C, \texttt{Ret} \vdash e \Rightarrow s \\ \mathrm{fv}(e) \setminus \{x\} = \{x_1, ..., x_n\} \quad C \vdash x_i \Rightarrow e_i \ \ i = [1..n] \end{array}}{\begin{array}{ll} C, k \vdash \texttt{fn}\ x \Rightarrow e \Rightarrow & \texttt{var}\ f = \texttt{function}(x)\{\ s\ \}; \\ & f.x_1 = e_1;\ \ldots\ f.x_n = e_n; \\ & \mathrm{WrapCont}(k)(f) \end{array}}$$

$$\frac{C, \texttt{Var}(x) \vdash e_1 \Rightarrow s_1 \quad C, \texttt{Var}(y) \vdash e_2 \Rightarrow s_2}{\begin{array}{ll} C, k \vdash e_1\ e_2 \Rightarrow & \texttt{var}\ x; \texttt{var}\ y; s_1\ s_2; \\ & \mathrm{WrapCont}(k)(x(y)) \end{array}}$$

$$\frac{C, \texttt{Var}(x) \vdash e_1 \Rightarrow s_1 \quad C \vdash e_2 \Rightarrow e_2'}{C, k \vdash e_1\ e_2 \Rightarrow \texttt{var}\ x; s_1; \mathrm{WrapCont}(k)(x(e_2'))}$$

$$\frac{C \vdash e \Rightarrow e' \quad C, k \vdash e_1 \Rightarrow s_1 \quad C, k \vdash e_2 \Rightarrow s_2}{C, k \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow \texttt{if}\ (e')\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}}$$

$$\frac{C \vdash e \Rightarrow e'}{C, k \vdash e \Rightarrow \mathrm{WrapCont}(k)(e')}$$

$$\frac{C, \texttt{Var}(x) \vdash e_1 \Rightarrow s_1 \quad C, k \vdash e_2 \Rightarrow s_2}{C, k \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{var}\ x; s_1; s_2}$$

$$\frac{\begin{array}{c} C' = \mathrm{fv}(e_1) \setminus \{x, f\} = \{x_1, ..., x_n\} \\ C', \texttt{Ret}(f, x) \vdash e_1 \Rightarrow s_1 \quad C, k \vdash e_2 \Rightarrow s_2 \\ C \vdash x_i \Rightarrow e_i' \ \ i = [1..n] \end{array}}{\begin{array}{l} C, k \vdash \texttt{fix}\ f(x) = e_1\ \texttt{in}\ e_2 \Rightarrow \\ \quad \texttt{var}\ f = \texttt{function}\ f(x)\{f : \texttt{while(true)}\{s_1\}; \}; \\ \quad f.x_1 = e_1'; \ldots f.x_n = e_n'; \\ \quad s_2; \end{array}}$$

$$\frac{C, \texttt{Var}(x) \vdash e \Rightarrow s}{C, \texttt{Ret}(f, x) \vdash f\ e \Rightarrow s; \texttt{continue}\ f;}$$

For the presentation, the translation considers only closed source program fragments. At top-level, a source language expression is translated into a JavaScript statement using the judgment $\emptyset \vdash e \Rightarrow s$.

For the complete translation, Standard ML reals, integers, booleans, words, and chars are implemented as JavaScript numbers with explicit checks for overflow when appropriate. Moreover, the complete translation, including the tail-call optimization, supports functions and functions calls with multiple parameters.

Consider the following Standard ML source program:

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

This function is translated into the following JavaScript code:

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

Notice the argument reassignment before the `continue` statement. There are still some room for improvements by removing some unnecessary variable bindings; we will leave this improvement to future work on the peep-hole optimizer.

Finally, we mention that the approach to dealing with tail calls can be generalized to work for mutually recursive functions by merging the mutually recursive functions into a single function containing an outer while loop and an inner switch statement with a branch for each function. Besides shuffling arguments correctly, a tail call needs to specify the switch index for the function to be called.

7. Experiments

In this section, we present a number of experiments performed with SMLtoJs to measure its performance, with regards to generated code efficiency and compile times.

Figure 6 compares running times for a series of Standard ML benchmark programs. Running times are measured[1] for the Firefox 6.0, Firefox 8.0a2, and Chromium[2] browsers and compared against native code generated with the native MLKit compiler [21].

	LoC	FF 6	FF 8	Chromium	Native
fib35	7	1.80	1.85	1.37	0.69
kkb	617	6.14	6.13	9.06	0.28
life	211	8.02	5.92	3.65	0.25
simple	1064	7.9	–	5.04	0.85
mandel	74	7.35	5.20	11.04	0.42
boyer	933	7.27	6.48	9.93	0.60
prof	277	0.93	0.97	0.59	0.61

Figure 6. Comparison of running times for a series of Standard ML benchmark programs.

The first column of Figure 6 shows the number of source code lines for each benchmark program. In general, browser running times are much larger than native running times, which may partly be due to lazy JIT compilation by the browser.

Compile times for the different browsers and for the native code compiler are presented in Figure 7. Times measured for the native code compiler include times for performing a series of optimizations and transformations that are not performed by the JavaScript targeting compiler (e.g., region inference and register allocation).

One interesting observation is that compile times for the benchmark programs across Firefox versions are very unstable. Recently, much effort has gone into improving and experimenting with the Firefox JavaScript engine, which may be the reason for the instability.

[1] Measurements done on a Thinkpad T42, 1GB RAM, Ubuntu 11.04.

[2] Chromium version 12.0.742.112.

	FF 6*	FF 8*	Chromium*	Native
fib35	0.12	0.12	0.04	0.64
kkb	–	54.8	5.50	1.59
life	2.39	2.23	2.51	3.14
simple	464	–	22.94	5.29
mandel	0.30	0.44	0.35	0.84
boyer	–	–	9.19	4.62
prof	–	2.40	2.45	1.46

(∗) Compilation in the browser.

Figure 7. Comparison of compile times for a series of Standard ML benchmark programs.

8. Other Uses of SMLtoJs

In this section, we present some other uses of SMLtoJs. An obvious use of SMLtoJs would be to use it for type-safe AJAX programming in concert with SMLserver [10–12], a Web server module for programming server-side Web applications in Standard ML. This approach to programming AJAX applications could benefit from the type-safe low-bandwidth serialization support, also used for serializing compiler symbol table information, as described in Section 5.

In the next section, we present a Reactive Web Programming (RWP) library for use with SMLtoJs. The library extends the DOM event handler architecture with library support for creating and transforming *behaviors* (i.e., time-changing values) and event streams, as well as support for installing behaviors directly in the DOM tree.

8.1 Reactive Web Programming (RWP)

Figure 8 lists the source code for a simple example that illustrates the use of the `Rwp` module.

```
open Rwp
val t : Time.time b = timer 100
val bt : Time.time b -> string b =
   arr (Date.toString o Date.fromTimeLocal)
val _ = print (
  "<html><body><h2>Time: <span id='time'>?" ^
  "</span></h2></body></html>")
val _ = insertDOM "time" (bt t)
```

Figure 8. A Reactive Web application that shows time updates every 100ms.

The example creates a time behavior `t` (which updates every 100ms) and a *behavior transformer* `bt`, which takes a time behavior and computes a string behavior by lifting a function of type `Time.time` → `string` to work on behaviors. Finally, the example stores the computed string behavior in the DOM using the `Rwp.insertDOM` function.

Reactive Web Programming is based on Hughes' Arrows [15], a generalization of Monads:

```
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
```

The ARROW signature specifies combinators for creating basic arrows and for composing existing arrows. The Reactive Web Programming library models behavior transformers and event stream transformers as arrows.

The signature for the Rwp : RWP library is given in Figure 9. As we shall illustrate in two simple examples, the library contains a number of useful functions for creating and manipulating behaviors and event streams.

```
signature RWP = sig
  type B type E (* Behaviors (B), 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
```

Figure 9. The Rwp library: Building Basic Behaviors and Event Streams.

The first example creates a behavior that is always the sum of two input fields:

```
  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)
```

The behavior transformer t takes a behavior of pairs of integers and returns a string behavior, which is used for insertion in the DOM. The example makes highly use of the arrow combinators, for transforming behaviors, and the basic textField function for referring to the content of an input field.

The following example illustrates elegantly the power of functional reactive programming. The example continuously reports the mouse position and continuously computes and reports two other positions:

```
  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 bm2 = t bm
  val _ = insertDOM "mouse0" bm2
  val _ = insertDOM "mouse1" (calm 400 bm2)
  val _ = insertDOM "mouse2" (delay 400 bm2)
```

Besides the precise mouse position, using the calm operator, the example shows the last mouse position for which the mouse has rested for 400ms. Moreover, using the delay combinator, the example shows the position of the mouse 400ms earlier.

The Rwp library implements behaviors and event streams using the "listener" pattern and functionality from the Js module from Section 4. For instance, installing a behavior b in the DOM tree involves adding a listener to b that updates the element using the function Js.innerHTML. Further, the implementations of calm and delay make use of the function Js.setTimeout and the implementation of textField and mouse make use of the two functions Js.installEventHandler and Js.onMouseMove, respectively.

There is more to be said about the implementation of the Rwp library, but the full story is beyond the scope of this paper.

9. Related Work

Examples of compiler frameworks that seeks to compile existing languages into JavaScript include the Google Web

Toolkit (GWT) [14], O'Browser [2], and earlier work on the Hop language (Scm2Js) [16].

Another group of related work include languages that have been developed for programming the Web using a single language. This group of work include ML5 [22], which uses modal logic for controlling code running in different "worlds". Similarly, Links [4], and the AFAX F# proposal [19], seek to compile certain code into JavaScript to execute in a browser and other code into code to be executed on a server.

The Hop project by Loitsch and Serrano [16], 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. This implementation also limits the implementation of tail calls to direct recursive function calls [23].

Another branch of related work includes Flapjax [17], which, together with the Fruit Haskell library by Courtney and Elliott [5], has served as inspiration for the `Rwp` reactive Web Programming library.

10. Conclusion

The main contribution of this work has been to demonstrate that compiling a full standardized programming language into JavaScript is feasible. The implementation deals correctly with all aspects of the Standard ML programming language, including polymorphic equality, generative exceptions, pattern matching, and modules. Moreover, all of the Standard ML Basis Library modules that make no assumptions about an underlying operating system are supported, including the `IntInf` module, which allows for operations on unbounded integers. The ultimate test has been the compilation of the SMLtoJs compiler itself into JavaScript, which allows for Standard ML programs to be compiled and executed in a browser.

References

[1] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (Version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

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

[3] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. Formal specification of the ML Basis System, January 2005. Available from `http://www.mlton.org`.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *In 5th International Symposium on Formal Methods for Components and Objects (FMCO*. Springer-Verlag, 2006.

[5] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 Haskell Workshop*, September 2001.

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

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

[8] Martin Elsman. A framework for cut-off incremental recompilation and inter-module optimization. Technical report, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark, April 2008.

[9] Martin Elsman and Niels Hallenberg. *SMLserver—A Functional Approach to Web Publishing*. The IT University of Copenhagen, February 2002. (154 pages). Available via `http://www.smlserver.org`.

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

[11] 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`.

[12] 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.

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

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

[15] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

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

[17] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–20, New York, NY, USA, 2009. ACM.

[18] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *1998 ACM International Workshop on ML (ML'98)*, pages 77–86, 1998.

[19] Tomas Petricek and Don Syme. AFAX: Rich client/server web applications in F#, 2007.

[20] Guy L. Steele, Jr. Rabbit: A compiler for scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[21] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical Report TR-2001-07, IT University of Copenhagen, October 2001.

[22] Tom Murphy Vii, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *In Trustworthy Global Computing 2007*, November 2007.

[23] Jérôme Vouillon. Js_of_ocaml. Documentation at http://ocsigen.org/js_of_ocaml/manual/.