Acta Univ. Sapientiae, Informatica, **3,** 1 (2011) 76–98

Implementing a non-strict purely functional language in JavaScript

László DOMOSZLAI
Eötvös Loránd University, Budapest, Hungary
Radboud University Nijmegen, the Netherlands
email: dlacko@gmail.com

Eddy BRUËL
Vrije Universiteit Amsterdam
the Netherlands
email: ejpbruel@gmail.com

Jan Martin JANSEN
Faculty of Military Sciences
Netherlands Defence Academy
Den Helder, the Netherlands
email: jm.jansen.04@nlda.nl

Abstract. This paper describes an implementation of a non-strict purely functional language in JavaScript. This particular implementation is based on the translation of a high-level functional language such as Haskell or Clean into JavaScript via the intermediate functional language Sapl. The resulting code relies on the use of an evaluator function to emulate the non-strict semantics of these languages. The speed of execution is competitive with that of the original Sapl interpreter itself and better than that of other existing interpreters.

1 Introduction

Client-side processing for web applications has become an important research subject. Non-strict purely functional languages such as Haskell and Clean have many interesting properties, but their use in client-side processing has been limited so far. This is at least partly due to the lack of browser support for these languages. Therefore, the availability of an implementation for non-strict

Computing Classification System 1998: D.1.1
Mathematics Subject Classification 2010: 68N18
Key words and phrases: web programming, functional programming, Sapl, JavaScript, Clean

purely functional languages in the browser has the potential to significantly improve the applicability of these languages in this area.

Several implementations of non-strict purely functional languages in the browser already exist. However, these implementations are either based on the use of a Java Applet (e.g. for Sapl, a client-side platform for Clean [8, 14]) or a dedicated plug-in (e.g. for HaskellScript [11] a Haskell-like functional language). Both these solutions require the installation of a plug-in, which is often infeasible in environments where the user has no control over the configuration of his/her system.

1.1 Why switch to JavaScript?

As an alternative solution, one might consider the use of JavaScript. A JavaScript interpreter is shipped with every major browser, so that the installation of a plug-in would no longer be required. Although traditionally perceived as being slower than languages such as Java and C, the introduction of JIT compilers for JavaScript has changed this picture significantly. Modern implementations of JavaScript, such as the V8 engine that is shipped with the Google Chrome browser, offer performance that sometimes rivals that of Java.

As an additional advantage, browsers that support JavaScript usually also expose their HTML DOM through a JavaScript API. This allows for the association of JavaScript functions to HTML elements through the use of event listeners, and the use of JavaScript functions to manipulate these same elements.

This notwithstanding, the use of multiple formalisms complicates the development of Internet applications considerably, due to the close collaboration required between the client and server parts of most web applications.

1.2 Results at a glance

We implemented a compiler that translates Sapl to JavaScript expressions. Its implementation is based on the representation of unevaluated expressions (thunks) as JavaScript arrays, and the just-in-time evaluation of these thunks by a dedicated evaluation function (different form the `eval` function provided by JavaScript itself).

Our final results show that it is indeed possible to realize this translation scheme in such a way that the resulting code runs at a speed competitive with that of the original Sapl interpreter itself. Summarizing, we obtained the following results:

- We realized an implementation of the non-strict purely functional programming language Clean in the browser, via the intermediate language Sapl, that does not require the installation of a plug-in.
- The performance of this implementation is competitive with that of the original Sapl interpreter and faster than that of many other interpreters for non-strict purely functional languages.
- The underlying translation scheme is straightforward, constituting a one-to-one mapping of Sapl onto JavaScript functions and expressions.
- The implementation of the compiler is based on the representation of unevaluated expressions as JavaScript arrays and the just-in-time evaluation of these thunks by a dedicated evaluation function.
- The generated code is compatible with JavaScript in the sense that the namespace for functions is shared with that of JavaScript. This allows generated code to interact with JavaScript libraries.

1.3 Organization of the paper

The structure of the remainder of this paper is as follows: we start with introducing Sapl, the intermediate language we intend to implement in JavaScript in Section 2. The translation scheme underlying this implementation is presented in Section 3. We present the translation scheme used by our compiler in two steps. In step one, we describe a straightforward translation of Sapl to JavaScript expressions. In step two, we add several optimizations to the translation scheme described in step one. Section 4 presents a number of benchmark tests for the implementation. A number of potential applications is presented in Section 5. Section 6 compares our approach with that of others. Finally, we end with our conclusions and a summary of planned future work in Section 7.

2 The Sapl programming language and interpreter

Sapl stands for **S**imple **A**pplication **P**rogramming **L**anguage. The original version of Sapl provided no special constructs for algebraic data types. Instead, they are represented as ordinary functions. Details on this encoding and its consequences can be found in [8]. Later a Clean like type definition style was adopted for readability and to allow for the generation of more efficient code (as will become apparent in Section 3).

The syntax of the language is the following:

⟨*program*⟩ ::= {⟨*function*⟩ | ⟨*type*⟩}+
⟨*type*⟩ ::= '::' ⟨*ident*⟩ '=' ⟨*ident*⟩ ⟨*ident*⟩* {'|' ⟨*ident*⟩ ⟨*ident*⟩*}*
⟨*function*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩* '=' ⟨*let-expr*⟩
⟨*let-expr*⟩ ::= ['let' ⟨*let-defs*⟩ 'in'] ⟨*main-expr*⟩
⟨*let-defs*⟩ ::= ⟨*ident*⟩ '=' ⟨*application*⟩ {',' ⟨*ident*⟩ '=' ⟨*application*⟩}*
⟨*main-expr*⟩ ::= ⟨*select-expr*⟩ | ⟨*if-expr*⟩ | ⟨*application*⟩
⟨*select-expr*⟩ ::= 'select' ⟨*factor*⟩ {'(' {⟨*lambda-expr*⟩ | ⟨*let-expr*⟩} ')'}+
⟨*if-expr*⟩ ::= 'if' ⟨*factor*⟩ '(' ⟨*let-expr*⟩ ')' '(' ⟨*let-expr*⟩ ')'
⟨*lambda-expr*⟩ ::= '\' ⟨*ident*⟩+ '=' ⟨*let-expr*⟩
⟨*application*⟩ ::= ⟨*factor*⟩ ⟨*factor*⟩*
⟨*factor*⟩ ::= ⟨*ident*⟩ | ⟨*literal*⟩ | '(' ⟨*application*⟩ ')'

An identifier can be any identifier accepted by Clean, including operator notations. For literals characters, strings, integer or floating-point numbers and boolean values are accepted.

We illustrate the use of SapI by giving a number of examples. We start with the encoding of the list data type, together with the `sum` function.

```
:: List = Nil | Cons x xs
sum xxs = select xxs 0 (λx xs = x + sum xs)
```

The `select` keyword is used to make a case analysis on the data type of the variable `xxs`. The remaining arguments handle the different constructor cases in the same order as they occur in the type definition (all cases must be handled separately). Each case is a function that is applied to the arguments of the corresponding constructor.

As a more complex example, consider the `mappair` function written in Clean, which is based on the use of pattern matching:

```
mappair f Nil         zs          = Nil
mappair f (Cons x xs) Nil         = Nil
mappair f (Cons x xs) (Cons y ys) = Cons (f x y) (mappair f xs ys)
```

This definition is transformed to the following SapI function (using the above definitions for `Nil` and `Cons`).

```
mappair f as zs
  = select as Nil (λx xs = select zs Nil (λy ys = Cons (f x y) (mappair f xs ys)))
```

SapI is used as an intermediate formalism for the interpretation of non-strict purely functional programming languages such as Haskell and Clean. The Clean compiler includes a SapI back-end that generates SapI code. Recently, the Clean compiler has been extended to be able to compile Haskell programs as well [5].

2.1 Some remarks on the definition of SapI

SapI is very similar to the core languages of Haskell and Clean. Therefore, we choose not to give a full definition of its semantics. Rather, we only say something about its main characteristics and give a few examples to illustrate these.

The only keywords in SapI are `let`, `in`, `if` and `select`. Only constant (non-function) `let` expressions are allowed that may be mutually recursive (for creating cyclic expressions). They may occur at the top level in a function and at the top level in arguments of an `if` and `select`. λ-expressions may only occur as arguments to a `select`. If a Clean program contains nested λ-expressions, and you compile it to SapI, they should be lifted to the top-level.

3 A JavaScript based implementation for SapI

Section 1 motivated the choice for implementing a SapI interpreter in the browser using JavaScript. Our goal was to make the implementation as efficient as possible.

Compared to Java, JavaScript provides several features that offer opportunities for a more efficient implementation. First of all, the fact that JavaScript is a *dynamic* language allows both functions and function calls to be generated at run-time, using the built-in functions `eval` and `apply`, respectively. Second, the fact that JavaScript is a dynamically *typed* language allows the creation of heterogeneous arrays. Therefore, rather than building an interpreter, we have chosen to build a compiler/interpreter hybrid that exploits the features mentioned above.

Besides these, the evaluation procedure is heavily based on the use of the `typeof` operator and the runtime determination of the number of formal parameters of a function which is another example of the dynamic properties of the JavaScript language.

For the following SapI constructs we must describe how they are translated to JavaScript:

- literals, such as booleans, integers, real numbers, and strings;
- identifiers, such as variable and function names;
- function definitions;
- constructor definitions;
- let constructs;
- applications;

- select statements;
- if statements;
- built-in functions, such as `add`, `eq`, etc.

Literals Literals do not have to be transformed. They have the same representation in Sapl and JavaScript.

Identifiers Identifiers in Sapl and JavaScript share the same namespace, therefore, they need not to be transformed either.

However, the absence of block scope in JavaScript can cause problems. The scope of variables declared using the `var` keyword is hoisted to the entire containing function. This affects the `let` construct and the λ-expressions, but can be easily avoided by postfixing the declared identifiers to be unique. In this way, the original variable name can be restored if needed.

With this remark we will neglect these transformations in the examples of this paper for the sake of readability.

Function definitions Due to JavaScript's support for higher-order functions, function definitions can be translated from Sapl to JavaScript in a straightforward manner:

`T⟦f x1 ... xn = body⟧` = **function** `f(x1, ..., xn) { T⟦body⟧ }`

So Sapl functions are mapped one-to-one to JavaScript functions with the same name and the same number of arguments.

Constructor definitions Constructor definitions in Sapl are translated to arrays in JavaScript, in such a way that they can be used in a `select` construct to select the right case. A Sapl type definition containing constructors is translated as follows:

```
T⟦:: typename = ... | Ck xk0 ... xkn | ...⟧
   = ... function Ck(xk0, ..., xkn) { return [k, 'Ck', xk0, ..., xkn]; } ...
```

where `k` is a positive integer, corresponding to the position of the constructor in the original type definition. The name of the constructor, `'Ck'`, is put into the result for printing purposes only. This representation of the constructors together with the use of the `select` statement allows for a very efficient JavaScript translation of the Sapl language.

Let constructs `Let` constructs are translated differently depending on whether they are cyclic or not. Non-cyclic lets in Sapl can be translated to `var` declarations in JavaScript, as follows:

`T⟦let x = e in b⟧ = var x = T⟦e⟧; T⟦b⟧`

Due to JavaScript's support for closures, cyclic lets can be translated from Sapl to JavaScript in a straightforward manner. The idea is to take any occurrences of `x` in `e` and replace them with:

`function () { return x; }`

This construction relies on the fact that the scope of a JavaScript closure is the whole function itself. This means that after the declaration the call of this closure will return a valid reference. In Section 3.1 we present an example to illustrate this.

Applications Every Sapl expression is an application. Due to JavaScript's eager evaluation semantics, applications cannot be translated from Sapl to JavaScript directly. Instead, unevaluated expressions (or *thunks*) in Sapl are translated to arrays in JavaScript:

`T⟦x0 x1 .. xn⟧ = [T⟦x0⟧, [T⟦x1⟧, ..., T⟦xn⟧]]`

Thus, a thunk is represented with an array of two elements. The first one is the function involved, and the second one is an array of the arguments. This second array is used for performance reasons. In this way one can take advantage of the JavaScript `apply()` method and it is very straightforward and fast to join such two arrays, which is necessary to do during evaluation.

select statements A `select` statement in Sapl is translated to a `switch` statement in JavaScript, as follows:

`T⟦select f (\x0 ... xn = b) ...⟧`

=

```
var _tmp = Sapl.feval(T⟦f⟧);
switch(_tmp[0]) {
    case 0: var x0 = _tmp[2], ..., xn = _tmp[n+2];
            T⟦b⟧;
            break;
    ...
};
```

Evaluating the first argument of a `select` statement yields an array representing a constructor (see above). The first argument in this array represents the position of the constructor in its type definition, and is used to select the right case in the definition. The parameters of the λ- expression for each case are bound to the corresponding arguments of the constructor in the `var` declaration (see also examples).

if statements An `if` statement in Sapl is translated to an `if` statement in JavaScript straightforwardly:

```
T⟦if p t f⟧ = if (Sapl.feval(T⟦p⟧)){ T⟦t⟧; } else { T⟦f⟧; }
```

This translation works because booleans in Sapl and JavaScript have the same representation.

Built-in functions Sapl defines several built-in functions for arithmetic and logical operations. As an example, the `add` function is defined as follows:

```
function add(x, y) { return Sapl.feval(x) + Sapl.feval(y); }
```

Unlike user-defined functions, a built-in function such as `add` has strict evaluation semantics. To guarantee that they are in normal form when the function is called, the function `Sapl.feval` is applied to its arguments (see Section 3.2).

3.1 Examples

The following definitions in Sapl:

```
:: List = Nil | Cons x xs

ones = let os = Cons 1 os in os
fac n = if (eq n 0) 1 (mult n (fac (sub n 1)))
sum xxs = select xxs 0 (λx xs = add x (sum xs))
```

are translated to the following definitions in JavaScript:

```
function Nil() { return [0, 'Nil']; }
function Cons(x, xs) { return [1, 'Cons', x, xs]; }

function ones() { var os = Cons(1, function() { return os; }); return os; }

function fac(n) {
    if (Sapl.feval(n) == 0) {
        return 1;
    } else {
        return [mult, [n, [fac, [[sub, [n, 1]]]]]];
    }
}
```

```
function sum(as) {
    var _tmp = Sapl.feval(as);
    switch (_tmp[0]) {
        case 0: return 0;
        case 1: var x = _tmp[2], xs = _tmp[3];
                return [add, [x, [sum, [xs]]]];
    }
}
```

The examples show that the translation is straightforward and preserves the structure of the original definitions.

3.2 The `feval` function

To emulate Sapl's non-strict evaluation semantics for function applications, we represented unevaluated expressions (thunks) as arrays in JavaScript. Because JavaScript treats these arrays as primitive values, some way is needed to explicitly reduce thunks to normal form when their value is required. This is the purpose of the `Sapl.feval` function. It reduces expressions to weak head normal form. Further evaluation of expressions is done by the printing routine. `Sapl.feval` performs a case analysis on an expression and undertakes different actions based on its type:

Literals If the expression is a literal or a constructor, it is returned immediately. Literals and constructors are already in normal form.

Thunks If the expression is a thunk of the form `[f, [xs]]`, it is transformed into a function call `f(xs)` with the JavaScript `apply` function, and `Sapl.feval` is applied recursively to the result (this is necessary because the result of a function call may be another thunk).

Due to JavaScript's reference semantics for arrays, thunks may become shared between expressions over the course of evaluation. To prevent the same thunk from being reduced twice, the result of the call is written back into the array. If this result is a primitive value, the array is transformed into a *boxed value* instead. Boxed values are represented as arrays of size one. Note that in JavaScript, the size of an array can be altered in-place.

If the number of arguments in the thunk is smaller than the arity of the function, it cannot be further reduced (is already in normal form), so it is returned immediately. Conversely, if the number of arguments in the thunk is larger than the arity of the function, a new thunk is constructed from the result of the call and the remainder of the arguments, and `Sapl.feval` is applied iteratively to the result.

Boxed values If the expression is a boxed value of the form `[x]`, the value `x` is unboxed and returned immediately (only literals and constructors can be boxed).

Curried applications If the expression is a curried application of the form `[[f, [xs]], [ys]]`, it is transformed into `[f, [xs ++ ys]]`, and `Sapl.feval` is applied iteratively to the result.

More details on evaluation For the sake of deeper understanding we also give the full source code of `feval`:

```
feval = function (expr) {
  var y, f, xs;
  while (1) {
      if (typeof(expr) == "object") {                  // closure
          if (expr.length == 1) return expr[0];        // boxed value
          else if (typeof(expr[0]) == "function") { // application -> make call
              f = expr[0];  xs = expr[1];
              if (f.length == xs.length) {             // most often occurring case
                  y = f.apply(null, xs);               // turn chunk into call
                  expr[0] = y;                         // overwrite for sharing!
                  expr.length = 1;                     // adapt size
              } else if (f.length < xs.length) {       // less likely case
                  y = f.apply(null,xs.splice(0, f.length));
                  expr[0] = y;                         // slice of arguments
              } else
                  return expr;                         // not enough arguments
          } else if (typeof(expr[0])=="object") {   // curried app -> uncurry
              y = expr[0];
              expr[0] = y[0];
              expr[1] = y[1].concat(expr[1]);
          } else
              return expr;                             // constructor
      } else if (typeof(expr) == "function")           // function
          expr = [expr, []];
        else                                           // literal
          return expr;
  }
}
```

3.3 Further optimizations

Above we described a straightforward compilation scheme from Sapl to JavaScript, where unevaluated expressions (thunks) are translated to arrays.

The `Sapl.feval` function is used to reduce thunks to normal form when their value is required. For ordinary function calls, our measurements indicate that the use of `Sapl.feval` is more than 10 times slower than doing the same call directly. This constitutes a significant overhead. Fortunately, a simple compile time analysis reveals many opportunities to eliminate unnecessary thunks in favor of such direct calls. Thus, expressions of the form:

`Sapl.feval([f, [x1, ..., xn])`

are replaced by:

`f(x1, ..., xn)`

This substitution is only possible if `f` is a function with known arity at compile-time, and the number of arguments in the thunk is equal to the arity of the function. It can be performed wherever a call to Sapl.feval occurs:

- The first argument to a `select` or `if`;
- The arguments to a built-in function;
- Thunks that follow a `return` statement in JavaScript. These expressions are always evaluated immediately after they are returned.

As an additional optimization, arithmetic operations are inlined wherever they occur. With these optimizations added, the earlier definitions of `sum` and `fac` are now translated to:

```
function fac(n) {
    if (Sapl.feval(n) == 0) {
        return 1;
    } else {
        return Sapl.feval(n) * fac(Sapl.feval(n) - 1);
    }
}

function sum(xxs) {
    var _tmp = Sapl.feval(xxs);
    switch(_tmp[0]){
        case 0: return 0;
        case 1: var x = _tmp[2], xxs = _tmp[3];
                return Sapl.feval(x) + sum(xs);
    }
}
```

Moreover, let's consider the following definition of the Fibonacci function, `fib`, in Sapl:

```
fib n = if (gt 2 n) 1 (add (fib (sub n 1)) (fib (sub n 2)))
```

This is translated to the following function in JavaScript:

```
function fib(n) {
    if (2 > Sapl.feval(n)) {
        return 1;
    } else {
        return (fib([sub, [n, 1]]) + fib([sub, [n, 2]]));
    }
}
```

A simple strictness analysis reveals that this definition can be turned into:

```
function fib(n) {
    if (2 > n) {
        return 1;
    } else {
        return (fib(n - 1) + fib(n - 2));
    }
}
```

The calls to `feval` are now gone, which results in a huge improvement in performance. Indeed, this is how `fib` would have been written, had it been defined in JavaScript directly. In this particular example, the use of eager evaluation did not affect the semantics of the function. However, this is not true in general. For the use of such an optimization we adopted a Clean like strictness annotation. Thus, the above code can be generated from the following Sapl definition:

```
fib !n = if (gt 2 n) 1 (add (fib (sub n 1)) (fib (sub n 2)))
```

But strictly defined arguments also have their price. In case one does not know if an argument in a function call is already in evaluated form, an additional wrapper function call is needed that has as only task to evaluate the strict arguments:

```
function fib$eval(a0) {
    return fib(Sapl.feval(a0));
}
```

As a possible further improvement, a more thorough static analysis on the propagation of strict arguments could help to avoid some of these wrapper calls.

Finally, the Sapl to JavaScript compiler provides simple tail recursion optimization, which has impact on not only the execution time, but also reduces stack use.

The optimizations only affect the generated code and not the implementation of `feval`. In the next section an indication of the speed-up obtained by the optimizations is given.

4 Benchmarks

In this section we present the results of several benchmark tests for the JavaScript implementation of Sapl (which we will call Sapljs) and a comparison with the Java Applet implementation of Sapl. We ran the benchmarks on a MacBook 2.26 MHz Core 2 Duo machine running MacOS X10.6.4. We used Google Chrome with the V8 JavaScript engine to run the programs. At this moment V8 offers one of the fastest platforms for running Sapljs programs. However, there is a heavy competition on JavaScript engines and they tend to become much faster. The benchmark programs we used for the comparison are the same as the benchmarks we used for comparing Sapl with other interpreters and compilers in [8]. In that comparison it turned out that Sapl is at least twice as fast (and often even faster) as other interpreters like Helium, Amanda, GHCi and Hugs. Here we used the Java Applet version for the comparison. This version is about 40% slower than the C version of the interpreter described in [8] (varying from 25 to 50% between benchmarks), but is still faster than the other interpreters mentioned above. The Java Applet and JavaScript version of Sapl and all benchmark code can be found at [2]. We briefly repeat the description of the benchmark programs here:

1. **Prime Sieve** The prime number sieve program, calculating the 2000th prime number.
2. **Symbolic Primes** Symbolic prime number sieve using Peano numbers, calculating the 160th prime number.
3. **Interpreter** A small Sapl interpreter. As an example we coded the prime number sieve for this interpreter and calculated the 30th prime number.
4. **Fibonacci** The (naive) Fibonacci function, calculating `fib 35`.
5. **Match** Nested pattern matching (5 levels deep) repeated 160000 times.
6. **Hamming** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 1000 times.
7. **Sorting** Tree Sort (3000 elements), Insertion Sort (3000 elements), Quick Sort (3000 elements), Merge Sort (10000 elements, merge sort is much faster, we therefore use a larger example)
8. **Queens** Number of placements of 11 Queens on a 11 x 11 chess board.

	Pri	Sym	Inter	Fib	Match	Ham	Qns	Kns	Sort	Plog	Parse
Sapl	1200	4100	500	8700	1700	2500	9000	3200	1700	1500	1100
Sapljs	2200	4000	220	280	2200	3700	11500	3950	2450	2750	4150
Sapljs nopt	4500	11000	1500	36000	6700	5500	36000	11000	4000	5200	6850
perc. mem.	58	68	38	0	21	31	37	35	45	53	41

Figure 1: Speed comparison (time in miliseconds).

9. **Knights** Finding a Knights tour on a 5 x 5 chess board.
10. **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating ancestors in a four generation family tree, repeated 100 times.
11. **Parser Combinators** A parser for Prolog programs based on Parser Combinators parsing a 3500 lines Prolog program.

For sorting a list of size n a source list is used consisting of numbers 1 to n. The elements that are 0 modulo 10 are put before those that are 1 modulo 10, etc.

The benchmarks cover a wide range of aspects of functional programming: lists, laziness, deep recursion, higher order functions, cyclic definitions, pattern matching, heavy calculations, heavy memory usage. The programs were chosen to run at least for a second, if possible. This helps eliminating start-up effects and gives the JIT compiler enough time to do its work. In many cases the output was converted to a single number (e.g. by summing the elements of a list) to eliminate the influence of slow output routines.

4.1 Benchmark tests

We ran the tests for the following versions of Sapl:

- Sapl: the Java Applet version of Sapl;
- Sapljs: the Sapljs version including the normal form optimization, the inlining of arithmetic operations and the tail recursion optimization. The strictness optimization is only used for the fib benchmark;
- Sapljs nopt: the version not using these optimizations.

We also included the estimated percentage of time spent on memory management for the Sapljs version. The results can be found in Figure 1.

4.2 Evaluation of the benchmark tests

Before analysing the results we first make some general remarks about the performance of Java, JavaScript and the Sapl interpreter which are relevant for a better understanding of the results. In general it is difficult to give absolute figures when comparing the speeds of language implementations. They often also depend on the platform (processor), the operating system running on it and the particular benchmarks used to compare. Therefore, all numbers given should be interpreted as global indications.

According to the language shoot-out site [3] Java programs run between 3 and 5 times faster than similar JavaScript programs running on V8. So a reimplementation of the Sapl interpreter in JavaScript is expected to run much slower as the Sapl interpreter.

We could not run all benchmarks as long as we wished because of stack limitations for V8 JavaScript in Google Chrome. It supports a standard (not user modifiable) stack of only 30k at this moment. This is certainly enough for most JavaScript programs, but not for a number of our benchmarks that can be deeply recursive. This limited the size of the runs of the following benchmarks: Interpreter[1] all sorting benchmarks, and the Prolog and Parser Combinator benchmark. Another benchmark that we used previously, and that could not be ran at all in Sapljs is: `twice twice twice twice inc 0`.

For a lazy functional language the creation of thunks and the re-collection of them later on, often takes a substantial part of program run-times. It is therefore important to do some special tests that say something about the speed of memory (de-)allocation. The Sapl interpreter uses a dedicated memory management unit (see [8]) not depending on Java memory management. The better performance of the Sapl interpreter in comparison with the other interpreters partly depends on its fast memory management. For the JavaScript implementation we rely on the memory management of JavaScript itself. We did some dedicated tests that showed that memory allocation for the Java Sapl interpreter is about 5-7 times faster than the JavaScript implementation. Therefore, we included an estimation of the percentage of time spent on memory management for all benchmarks ran in Sapljs. The estimation was done by counting all memory allocations for a benchmark (all creations of thunks) and multiplying it with an estimation of the time to create a thunk, which was measured by a special application that only creates thunks.

[1] The latest version of Chrome has an even more restricted stack size. We can now run Interpreter only up to the 18th prime number.

Results The Fibonacci and Interpreter benchmarks run (30 and 2 times resp.) significantly faster in Sapljs than in the Sapl interpreter. Note that both these benchmarks profit significantly from the optimizations with Fibonacci being more than 100 times faster and Interpreter almost 7 times faster than the non-optimized version. The addition of the strictness annotation for Fibonacci contributes a factor of 3 to the speed-up. With this annotation the compiled Fibonacci program is equivalent to a direct implementation of Fibonacci in JavaScript and does not use feval anymore. The original Sapl interpreter does not apply any of these optimizations. The Interpreter benchmark profits much (almost a factor of 2) from the tail recursion optimization that applies for a number of often used functions that dominate the performance of this benchmark.

Symbolic Primes, Match, Queens and Knights run at a speed comparable to the Sapl interpreter. Hamming and Sort are 40 percent slower, Primes and Prolog are 80 percent slower. Parser Combinators is the worst performing benchmark and is almost 4 times slower than in Sapl.

All benchmarks benefit considerably from the optimizations (between 1.5 and 120 times faster), with Fibonacci as the most exceptional.

The Parser Combinators benchmark profits only modestly from the optimizations and spends relatively much time in memory management operations. It is also the most 'higher order' benchmark of all. Note that for the original Sapl interpreter this is one of the best performing benchmarks (see [8]), performing at a speed that is even competitive with compiler implementations. The original Sapl interpreter does an exceptionally good job on higher order functions.

We conclude that the Sapljs implementation offers a performance that is competitive with that of the Sapl interpreter and therefore with other interpreters for lazy functional programming languages.

Previously [8] we also compared Sapl with the GHC and Clean compilers. It was shown that the C version of the Sapl interpreter is about 3 times slower than GHC without optimizer. Extrapolating this result using the figures mentioned above we conclude that Sapljs is about 6-7 times slower than GHC (without optimizer). In this comparison we should also take into account that JavaScript applications run at least 5 times slower than comparable C applications. The remaining difference can be mainly attributed to the high price for memory operations in Sapljs.

4.3 Alternative memory management?

For many Sapljs examples a substantial part of their run-time is spent on memory management. They can only run significantly faster after a more efficient memory management is realized or after other optimizations are realized. It is tempting to implement a memory management similar to that of the Sapl interpreter. But this memory management relies heavily on representing graphs by binary trees, which does not fit with our model for turning thunks into JavaScript function calls which depends heavily on using arrays to represent thunks.

5 Applications

Developing rich client-side applications in Clean We can use the Sapljs compiler to create dedicated client-side applications in Clean that make use of JavaScript libraries. We can do this because JavaScript and code generated by Sapljs share the same namespace. In this way it is possible to call functions within Sapl programs that are implemented in JavaScript. The Sapljs compiler doesn't check the availability of a function, so one has to rely on the JavaScript interpreter to do this. Examples of such functions are the built-in core functions like `add` and `eq`, but they can be any application related predefined function.

Because we have to compile from Clean to Sapl before compiling to JavaScript, we need a way to use functions implemented in JavaScript within Clean programs. Clean does not allow that programs contain unknown functions, so we need a way to make these functions known to the Clean compiler. This can be realized in the following way. If one wants to postpone the implementation of a function to a later time, one can define its type and define its body to be `undef`. E.g., `example` is a function with 2 integer arguments and an integer result with an implementation only in JavaScript.

```
example :: Int Int → Int
example = undef
```

The function `undef` is defined in the StdMisc module. An `undef` expressions matches every type, so we can use this definition to check if the written code is syntactically and type correct. We adapted the Clean to Sapl compiler not to generate code for functions with an undefined body. In this way we have created a universal method to reference functions defined outside the Clean environment.

We used these techniques to define a library in Clean for manipulating the HTML DOM at the client side. The following Clean code gives a demonstration of its use:

```
import StdEnv, SaplHtml

onKeyUp :: !HtmlEvent !*HtmlDocument → *(*HtmlDocument, Bool)
onKeyUp e d
    # (d, str) = getDomAttr d "textarea" "value"
    # (d, str) = setDomAttr d "counter" "innerHTML" (toString (size str))
    = (d, True)

Start
    = toString (Div [] [] [TextArea [Id "textarea", Rows 15, Cols 50]
                                    [OnKeyUp onKeyUp],
                           Div [Id "counter"] [] []])
```

It is basically a definition of a piece of HTML using arrays and ADTs defined in the `SaplHtml` module. What is worth to notice here are the definitions of the event handler function and the DOM manipulating functions, `getDomAttr` and `setDomAttr`, which are also defined in `SaplHtml`, but are implemented in JavaScript using the above mentioned techniques. The two parameters of the event handler function are effectively the related JavaScript `Event` and `Document` objects, respectively.

Compiling the program to JavaScript and running it returns the following string, which is legal HTML:

```
<div><textarea id="textarea"
          rows="15"
          cols="50"
          onKeyUp="Sapl.execEvent(event, 'onKeyUp$eval')">
     </textarea>
     <div id="counter"></div>
</div>
```

The event handler call is wrapped by the `Sapl.execEvent` function which is responsible for passing the event related parameters to the actual event handler. Including this string into an HTML document along with the generated JavaScript functions we get a client side web application originally written in Clean. Despite this program is deliberately very simple, it demonstrates almost all the basics necessary to write any client side application. Additional interface functions, e.g. calling methods of a JavaScript object, can be found in the `SaplHtml` module.

iTask integration Another possible application is related to the iTask system [13]. iTask is a combinator library written in Clean, and is used for the realization of web-based dynamic workflow systems. An iTask application consists of a structured collection of tasks to be performed by users, computers or both.

To enhance the performance of iTask applications, the possibility to handle tasks on the client was added [14], accomplished by the addition of a simple `OnClient` annotation to a task. When this annotation is present, the iTask runtime automatically takes care of all communication between the client and server parts of the application. The client part is executed by the Sapl interpreter, which is available as a Java applet on the client.

However, the approachability of JavaScript is much better compared to Java. The Java runtime environment, the Java Virtual Machine might not even be available on certain platforms (on mobile devices in particular). Besides that, it exhibits significant latency during start-up. For these reasons, a new implementation of this feature is recommended using Sapljs instead of the Sapl interpreter written in Java. Several feature were made to foster this modification:

- The Sapl language was extended with some syntactic sugar to allow distinguishing between constructors and records.
- Automatic conversion of data types like records, arrays, etc, between Sapl and JavaScript was added. In this way full interaction between Sapl and existing libraries in JavaScript became possible.
- Automatic conversion of JSON data structures to enable direct interfacing with all kinds of web-services was added.

6 Related work

Client-side processing for Internet applications is a subject that has drawn much attention in the last years with the advent of Ajax based applications.

Earlier approaches using JavaScript as a client-side platform for the execution of functional programming languages are Hop [15, 10], Links [1] and Curry [7].

Hop is a dedicated web programming language with a HTML-like syntax build on top of Scheme. It uses two compilers, one for compiling the server-side program and one for compiling the client-side part. The client-side part is only used for executing the user interface. The application essentially runs on the client and may call services on the server. Syntactic constructions are used for indicating client and server part code. In [10] it is shown that a reasonably

good performance for client-side functions in Hop can be obtained. However, contrary to Haskell and Clean, both Hop and the below mentioned Links are strict functional languages, which simplifies their translation to JavaScript considerably.

Links [1] and its extension Formlets is a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. Client-server communication is implemented using Ajax technology, like this is done in the iTask system.

Curry offers a much more restricted approach: only a very restricted subset of the functional-logic language Curry is translated to JavaScript to handle client-side verification code fragments only.

A more recent approach is the Flapjax language [12], an implementation of functional reactive programming in JavaScript. Flapjax can be used either as a programming language, compiling to JavaScript, or as a JavaScript library. Entire applications can be developed in Flapjax. Flapjax automatically tracks dependencies and propagates updates along dataflows, allowing for a declarative style of programming.

An approach to compile Haskell to JavaScript is YCR2JS [4] that compiles YHC Core to JavaScript, comparable to our approach compiling Sapl to JavaScript. Unfortunately, we could not find any performance figures for this implementation.

Another, more recent approach, for compiling Haskell to JavaScript is HS2JS [6], which integrates a JavaScript backend into the GHC compiler. A comparison of JavaScript programs generated by this implementation indicate that they run significantly slower than their Sapljs counterparts.

7 Conclusion and future work

In this paper we evaluated the use of JavaScript as a target language for lazy functional programming languages like Haskell or Clean using the intermediate language Sapl. The implementation has the following characteristics:

- It achieves a speed for compiled benchmarks that is competitive with that of the Sapl interpreter and is faster than interpreters like Amanda, Helium, Hugs and GHCi. This is despite the fact that JavaScript has a 3-5 times slower execution speed than the platforms used to implement these interpreters.

- The execution time of benchmarks is often dominated by memory operations. But in many cases this overhead could be significantly reduced by a simple optimization on the creation of thunks.
- The implementation tries to map Sapl to corresponding JavaScript constructs as much as possible. Only when the lazy semantics of Sapl requires this, an alternative translation is made. This opens the way for additional optimizations based on compile time analysis of programs.
- The implementation supports the full Clean (and Haskell) language, but not all libraries are supported. We tested the implementation against a large number of Clean programs compiled with the Clean to Sapl compiler.

7.1 Future work

We have planned the following future work:

- Implement a web-based Clean to Sapl (or to JavaScript) compiler (experimental version already made).
- Experimenting with supercompilation optimization by implementing a Sapl to Sapl compiler based on whole program analysis.
- Encapsulate JavaScript libraries in a functional way, e.g. using generic programming techniques.
- Attach client-side call-backs written in Clean to iTask editors. It can be implemented using Clean-Sapl dynamics [9] which make it possible to serialize expressions at the server side and execute them at the client side.
- Use JavaScript currying instead of building thunks. Our preliminary results indicate that using JavaScript currying would be significantly slower, but further investigation is needed for proper analysis.

Acknowledgements

The research of the first author was supported by the European Union and the European Social Fund under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

References

[1] E. Cooper, S. Lindley, P. Wadler, J. Yallop, Links: web programming without tiers, *Proc. 5th International Symposium on Formal Methods for Components and Objects (FMCO '06)* , *Lecture Notes in Comput. Sci.*, **4709** (2006) 266–296. ⇒94, 95

[2] L. Domoszlai, E. Bruël, J. M. Jansen, The Sapl home page, `http://www.nlda-tw.nl/janmartin/sapl`. ⇒88

[3] B. Fulgham, The computer language benchmark game, `http://shootout.alioth.debian.org`. ⇒90

[4] D. Golubovsky, N. Mitchell, M. Naylor, Yhc.Core – from Haskell to Core, *The Monad.Reader*, **7** (2007) 236–243. ⇒95

[5] J. van Groningen, T. van Noort, P. Achten, P. Koopman, R. Plasmeijer, Exchanging sources between Clean and Haskell – a double-edged front end for the Clean compiler, *Haskell Symposium,* Baltimore, MD, 2010. ⇒79

[6] T. Hallgren, HS2JS test programs, `http://www.altocumulus.org/~hallgren/hs2js/tests/`. ⇒95

[7] M. Hanus, Putting declarative programming into the web: translating Curry to JavaScript, *Proc. 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '07),* Wroclaw, Poland, 2007, ACM, pp. 155–166. ⇒94

[8] J. M. Jansen, P. Koopman, R. Plasmeijer, Efficient interpretation by transforming data types and patterns to functions, *Proc. Seventh Symposium on Trends in Functional Programming (TFP 2006),* Nottingham, UK, 2006. ⇒77, 78, 88, 90, 91

[9] J. M. Jansen, P. Koopman, R. Plasmeijer, iEditors:extending iTask with interactive plug-ins, *Proc. 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008),* Hertfordshire, UK, 2008, pp. 170–186. ⇒96

[10] F. Loitsch, M. Serrano, Hop client-side compilation, *Trends in Functional Programming (TFP 2007)*, New York, 2007, pp. 141–158. ⇒94

[11] E. Meijer, D. Leijen, J. Hook, Client-side web scripting with HaskellScript, *First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*, San Antonio, Texas, 1999, *Lecture Notes in Comput. Sci.*, **1551** (1999) 196–210. ⇒77

[12] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, Flapjax: a programming language for Ajax applications, *SIGPLAN Not.*, **44** (2009) 1–20. ⇒95

[13] R. Plasmeijer, P. Achten, P. Koopman, iTasks: executable specifications of interactive work flow systems for the web, *Proc. 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, Freiburg, Germany, 2007, ACM, pp. 141–152. ⇒94

[14] R. Plasmeijer, J. M. Jansen, P. Koopman, P. Achten, Declarative Ajax and client side evaluation of workflows using iTasks, *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '08)*, Valencia, Spain, 2008. ⇒77, 94

[15] M. Serrano, E. Gallesio, F. Loitsch, Hop: a language for programming the web 2.0, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, Portland, Oregon, 2006, pp. 975–985. ⇒94

Received: January 31, 2011 • Revised: March 23, 2011