UNIVERSITY OF GOTHENBURG

A language for functional web programming

Bachelor of Science thesis

Anton Ekblad

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering

Göteborg, Sweden, October 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

A language for functional web programming

Anton Ekblad

© Anton Ekblad, October 2011.

Examiner: Koen Lindström Claessen

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2011

Abstract

Computer programs are by far the most complex artifacts ever produced by humanity, and they get more complex year by year. As complexity grows, so does the need for better tools and higher level abstractions. In recent years, applications that run within the web browser have become increasingly popular, but the languages and tools used to develop them are archaic and error-prone as compared to the ecosystem available to developers of native applications. This thesis aims to bring a little more of this ecosystem to the field of web applications through the creation of a compiler for a purely functional prototype language that is able to run on any web browser that supports JavaScript.

Acknowledgements

I would like to thank my supervisor, Nicholas Smallbone, for his valuable guidance and feedback through the entire project; Koen Lindström Claessen, my examiner, for giving me the opportunity to do this project and for helping out with all the details that surround a bachelor's thesis; my girlfriend, Sofia Zaid; Jonathan Skårstedt; Peter Holm and Martin Falk Johansson for proof reading this thesis, providing valuable feedback and comments.

Without you guys, this thesis would not have been possible.

Contents

1 Introduction **1**

1.1 JavaScript as a (not quite) functional language 1

1.2 Other shortcomings 3

1.3 Wish list for an improved web language 5

1.4 Lambdascript to the rescue 6

2 Related work **7**

3 Background **7**

3.1 A quick introduction to JavaScript 7

3.2 Hindley-Milner type inference 10

3.3 Non-strict semantics and equational reasoning 11

4 Implementation **12**

4.1 Syntax and features 12

4.2 Data representation 15

4.3 Tail call elimination 18

4.4 Modules 19

4.5 Lexing and parsing 20

4.6 Desugaring 21

4.7 Type checking and annotation 23

4.8 Intermediate code generation 24

4.8.1 Generating patterns 24

4.8.2 Generating expressions 26

4.9 Optimization 27

4.10 Intermediate code to JavaScript 28

4.11 Runtime library 28

4.12 Output format and JavaScript interface 29

4.13 Performance 31

5 Future work **35**

6 Conclusions **37**

References **39**

1 Introduction

Functional programming has brought tremendous improvements in productivity and reliability to a wide range of fields within software development. However, there is one, increasingly important, area where it still has failed to make any substantial impression: client-side web development. In the web browser, the ECMAScript dialect known as JavaScript is still the only real cross-browser compatible way of developing rich client web applications. This is unfortunate, not only because of the "one size fits all" model thereby imposed on web developers, but also due to several problems with the JavaScript language itself.

This thesis assumes that the reader is at least somewhat familiar with Haskell and functional programming, and at least somewhat acquainted with the basic process of compilation. JavaScript knowledge is not required, but a reader who is not at least somewhat familiar with the language might want to read section 3.1 before continuing with the introduction.

1.1 JavaScript as a (not quite) functional language

While JavaScript admittedly does have closures and functions as first-class values and thus, according to some, would classify as a functional language, when it comes to actually enabling a functional programming style it is severely lacking. Consider, for example, the following code fragment.

```
function sumLists(lists) {
    return map(function(x) {
            return foldl(function(sum,x) {return sum+x;}, 0, x);
        }, someList);
}
```

Assuming the presence of essential functional primitives such as `map` and `foldl`, this fragment defines a function that takes as its input a list of lists of numbers, sums the numbers in each input list individually, and returns the resulting list of sums. This looks quite clunky; indeed, let's compare this

function to another code fragment, performing the same task in Haskell, a pure functional language.

```
sumLists = map (foldl (+) 0)
```

As we can see, the Haskell version is not only much shorter but also considerably clearer. In comparison, JavaScript becomes unnecessarily verbose when writing code in a functional style like this.

Moreover, the JavaScript specification makes tail call elimination, an optimization technique that turns tail recursive function calls into loops, impossible in the general case by demanding that functions be able to access their own call stack at run time. As an example, the following function runs in *O(1)* space with tail call elimination enabled; without it, the space bound becomes *O(n)* due to each recursive call needing to store its state on the stack.

```
foldl f a []     = a
foldl f a (x:xs) = foldl f (f a x) xs
```

Not only does the lack of tail call elimination severely impact performance for such code, but it makes using tail recursion for anything but trivial problem instances all but impossible since a recursive function would quickly run out of stack space without this optimization. The fact that many JavaScript implementations have very shallow call stacks further complicates matters to the point where the above implementation of `foldl`, if translated into JavaScript, would terminate due to a call stack overflow for any list with more than 200 elements when run on certain popular web browsers. Since client code needs to be able to run on just about any modern web browser, this obviously rules out the use of tail recursion in JavaScript code completely.

Furthermore, JavaScript completely lacks standard functional building blocks such as the `map` and `foldl` functions utilized in the previous examples. Of course, implementing these functions yourself would be a trivial affair, but not having a standard library of functional primitives virtually ensures massive duplication of effort as various projects invent their own functional libraries, all with subtly differing semantics and quirks. That there are no standard list or tuple types further aggravates the situation.

1.2 Other shortcomings

While the aforementioned issues make JavaScript less than ideal as a functional language, these are not the only problems when using it to develop larger applications. Weak typing[1] is one property of JavaScript which makes it all too easy to introduce subtle bugs. Consider the following function.

```
function agePlusOne() {
    var currentAge = prompt('How old are you?');
    var nextYearAge = currentAge + 1;
    alert('Next year, you will be ' + nextYearAge + ' old!');
}
```

At first glance, this function seems reasonable; it looks like it prompts the user for her current age, then tells her how old she'll be next year. Unfortunately, this is not what the function does. The `+` operator is overloaded to denote both addition (when both of its operands are numbers) and string concatenation (when both of its operands are strings.) However, in our example, the operands have different types as `prompt()` returns a string value. In this case JavaScript will silently convert the second operand to a value of the same type as the first. Thus, if the user gives her age as 24, our example function will perform string concatenation where we obviously intended addition and gleefully report that she'll soon be turning 241!

Another major pitfall is JavaScript's somewhat eccentric scoping, which can be exemplified using the following code fragment.

```
var x = 0;
for(var i = 0; i < 10; ++i) {
    var x = i;
}
```

Since JavaScript is a C-like language, one would expect it to obey roughly the same scoping rules as C, C#, C++, Java, PHP and the other C-likes, where

[1] A property of certain programming languages which, when the types of values to be used in an operation don't match, attempts to automatically convert one or more of the values into a type appropriate for the operation.

a pair of curly braces opens up a new scope and variables declared therein stay alive only within those braces. Not so with JavaScript; if one wants to open a new scope without creating an entirely new function one has to use the `with` keyword. Not only is this cumbersome, but also easy to overlook since the scoping semantics are so different from the other C-like languages. If the above code fragment were to be interpreted as C#, the variable `x` would have the value 0 after it finished executing; the `var x` in the loop body would declare a new variable named `x`, only in scope within the loop body, which would receive the assignments. C, C++, Java and PHP would all exhibit the same behavior, were one to make the minor syntax changes necessary to get the example to compile in those languages. However, interpreted as JavaScript, the variable `x` will end up equal to 9; since no new scope is ever opened the original value of `x` therefore gets overwritten with each iteration of the loop. Even if we embrace this way of handling scope, it would not be unreasonable to imagine that the two declarations of `x` within the same scope would trigger an error or at least a warning of some sort, something which is not the case; the re-declaration is silently ignored.

JavaScript also does not have any support for the concept of modules, or even C-style preprocessor include directives. To combine several source files into a larger program, the developer either has to resort to manually include all of the JavaScript source files used in the web page hosting the application, cluttering her HTML[2] code with `<script>` tags in the process, or write JavaScript code to dynamically add `<script>` tags to the HTML code at run time. This makes managing larger projects in JavaScript a fairly complicated affair as code cannot easily be split into several smaller files. It also complicates any sort of information hiding and separation of concerns between modules, as adding `<script>` tags to a web page essentially just pastes the contents of the referenced file into a global name space shared by all code executing on the page. Design patterns have emerged to provide partial workarounds to this problem,[Cherry 2010] but the fundamental problem still remains.

[2]Hyper Text Markup Language; the language commonly used to describe the semantic structure of web pages and applications

1.3 Wish list for an improved web language

There are also a few features which, while their omission don't really count as a shortcoming per se, would be nice to have in a modern language for client-side web development. Algebraic data types, and pattern matching to work on them, are one such feature. While it's perfectly possible to use nested `if`-statements and `switch-case`-blocks to manipulate symbolic values, such code tends to grow in complexity quite fast. Algebraic data types do away with the need to use numbers or manually defined constants to represent various non-numeric concepts, and also frees the developer from having to think about what fields of a compound value are currently active, preventing situations like the one in the following code fragment.

```
function foo(x) {
    if(x.hasValue) {
        return x.value * 2;
    } else {
        // x does not have a value - accessing it is a bug!
        return x.value + 1;
    }
}
```

As we can see in the above example, it's quite possible for a JavaScript developer to accidentally try to access a field of a compound type that's not currently active, leading to all sorts of interesting, possibly hard-to-find bugs.

Being able to enforce immutability at will, and somehow restrict side-effects for certain functions, would also be great boons in writing more robust software. As all JavaScript variables have reference semantics when used within a closure, keeping track of the current values of the variables involved in even simple calculations can be a daunting task for code which makes liberal use of closures. As daunting tasks are best avoided if one wants to produce robust code (or indeed, any substantial amount of code at all,) the ability to restrict mutation for certain values or types alone would make

be a prize worthy of pursuit. While being able to manipulate the DOM[3], an inherently side-effecting operation, is very important for JavaScript as the DOM is the main communications channel between the user and the program, there is no reason why the underlying calculations would need to make use of side effects, hence being able to enforce referential transparency when desirable would obviously be tremendously useful.

1.4 Lambdascript to the rescue

For these reasons, it would seem that an alternative is called for. While JavaScript's stranglehold on the web browser is loosened somewhat by projects such as [Google Web Toolkit], tools for developing web applications in a functional style are still lacking. In particular, all previous projects to enable Haskell to run within a web browser either produce prohibitively large output or are not mature enough to build upon. While large output may not be a problem when producing native code for today's machines with several gigabytes of memory and for all intents and purposes unlimited persistent storage, JavaScript code, which not only has to be retransmitted over a possibly slow network whenever the user's web browser decides that its cached copy of the code is too old, but also increasingly has to run on mobile devices where bandwidth is still quite expensive, can not afford this luxury.

This thesis details the development of a compiler from a Haskell-like language, named Lambdascript, to JavaScript. To restrict the scope of the project to something suitable for a one person bachelor's thesis, type classes are not implemented and "polish features" such as the ability to use binary functions as infix operators or define entirely new symbolic operators have been omitted, as has the ability to use pattern matching in local bindings (e.g. `let Foo x = bar`.) As a consequence of not supporting type classes, higher order type variables are also disallowed. Since the focus of this thesis is to actually produce running JavaScript for a non-strict, functional language, the language's syntax has been restricted to something quite similar

[3]Document Object Model; the fully mutable run-time representation of the structure of a web page

to Haskell's context-free syntax, in order to be parseable using an LR(1) parser. As code optimization is quite a large task on its own, producing blazingly fast code is not a priority for this thesis.

2 Related work

Web development being quite a hot topic lately, there exist several projects with similar aims. The now defunct ycr2js [Golubovsky 2007] project attempted to generate JavaScript code from the intermediate output of the also-defunct York Haskell Compiler. Unfortunately, the JavaScript files generated by this approach were quite large, and would sometimes not run at all. The Utrecht Haskell Compiler has taken a similar approach with a JavaScript back end scheduled for inclusion with the next release of their general purpose Haskell compiler, [Dijkstra 2010] and experimental work is being done on adding a similar backed to the de facto standard GHC compiler. [Mackenzie 2011] There is also a JavaScript compiler for a subset of Haskell to JavaScript by [Björnesjö, Holm 2011]. At the time of writing, none of these projects have reached maturity.

There are also efforts underway to compile a wealth of imperative languages to JavaScript. Some, such as [Google Web Toolkit] for turning Java code into JavaScript or [Pyjamas] for doing the same for Python, are quite mature, come with associated widget toolkits and development environments, and are already being used to develop production-quality applications. [GWT projects] As this thesis restricts itself to the compilation of functional languages, these projects may be interesting but are only tangentially related to this thesis.

3 Background

3.1 A quick introduction to JavaScript

The JavaScript language, which is both our target language and the language we're trying to replace, is a prototype-based, interpreted language with a C-like syntax. It uses weak dynamic typing, and has native support

for functions and closures as first-class values. The type of any value can be queried at run time and member fields can be added on the fly, both to individual objects and to entire types, also giving JavaScript reflective properties. Unstructured jumps are not allowed, although the **break** and **continue** keywords allows developers to break out of and jump to the head of any number of nested loops at once.

```
var i = 0;
outer:
for(;;) {
    ++i;
    inner:
    for(;;) {
        if(i % 3079 == 0) {
            break outer;
        } else {
            continue outer;
        }
    }
}
```

In the above example, the inner **for**-loop actually does nothing; each time it is entered, either the entire loop structure is broken out of (if 3079 divides **i**) or the outer loop is restarted using the labeled **continue** keyword.

All variables are mutable, and have reference semantics under closure. Functions are call-by-value, with complex types having their reference passed by-value; this means, perhaps counter-intuitively, that variable assignment behaves quite differently depending on the context.

```
var f = function(x) {return x+1;};
f = function(x) {return f(x)+1;};
```

Due to the reference semantics under closure property, the above example will not return; as **f** always has reference semantics when closed over, the second

closure will actually end up calling itself until the program terminates with a call stack overflow error.

In JavaScript, everything is either a primitive type or an object, and an object is simply a dictionary. Thus, object members may accessed either using dictionary syntax (**obj['member']**) or using more traditional C-style field access (**obj.member**.) The only primitive types are **number** and **boolean**. While the **string** type does not derive from the base **Object** type, it is still a reference type, behaviorally indistinguishable from **Object** subtypes. Functions and arrays are also objects, and can have fields added or removed, just like other objects.

```
function f(x) {
    f.lastX = x;
    return x+2;
}
f(f(10))
print(f.lastX);
```

The example above is quite legal, and will print **12**, as the outer invocation of `f` was called with 12 as its argument. Functions may also access their own call stack.

```
function f() {
    print(f.caller.name)
}
function g() {
    f();
}
g();
```

The above code will print `g`, as `g` is the caller of `f` in this example. Had we wanted to, we could have printed the name, arity, argument list, source code and so on of each function in the entire call stack, as the `caller` property of a function points directly to the calling function object. This is a major

reason why allowing tail call elimination or multi threading is impossible for the current incarnation of JavaScript.

Functions also have access to a special variable, called **this**. This variable is a reference to the object the function is being called as a method of, or **null** if the function is called as a function rather than a method, quite similar to the `this` pointer in object-oriented languages such as Java and C++.

```
function f() {
    print(this.bar);
}
var anObject = {foo: f, bar: "What? The curtains?"};
// This line prints "What? The curtains?"
anObject.foo();
// This line causes an error, as the this pointer in f is null
// when f is called as a function.
f();
```

3.2 Hindley-Milner type inference

Like many other functional languages, Lambdascript's type system is based on Hindley-Milner type inference. Hindley-Milner is attractive not only because it is always able to infer the most general type of any expression without any type annotations, but also because it does so in a simple and efficient manner. HM defines an algorithm for inferring types for the three basic lambda calculus expressions - variables, application and abstraction - as well as `let`-binding. This, however, is not enough to infer types for a Haskell-like language; we also need to be able to infer types for patterns, as used for pattern matching. For this reason, a slightly modified version of the extended algorithm used to check and infer types for Haskell code, presented in [Jones 1999], is used. As the algorithm is described in detail in both [Milner 1978] and [Jones 1999], this section is content to explain the general idea behind it.

The basic idea for both the extended algorithm and standard HM is to

take a function body[4], introduce a new type variable t for every identifier v, including the function itself,[5]adding the assumption that v has type t to a list of all such assumptions for the function we're currently inferring types for. Constraints are then placed upon these type variables depending on how the identifiers to which they are bound are used, these constraints are then solved to deduce the most general type for each variable. If any free type variables still remain in the type assigned to the function after all constraints imposed by the function's body have been taken into account, these are universally quantified to give the function a polymorphic type. The type of this function is then used to infer the type of any other function calling it.

3.3 Non-strict semantics and equational reasoning

Lambdascript disallows side effects for the same reason they are heavily restricted in Haskell; to enable developers to reason about the code as though it were pure math - to use *equational reasoning*. When a function is referentially transparent, substituting any call to a given function for its body obviously doesn't change the meaning of the program - or does it? In fact, it well might. Assuming strict semantics, consider the following function.

```
foo x y = if x then y else 0
```

If we apply foo as `foo True 10` or `foo False 10`, everything seems alright; we can replace those calls with the body of foo without changing the meaning of the expression. But what if we instead apply `foo` to `undefined`? In the case of `foo True undefined`, everything works as intended; `undefined` gets evaluated regardless of whether we apply the function or if we replace the application with its definition, an error is raised as expected. The problem appears with `foo False undefined`; in this case, if we use function application with strict semantics, `undefined` is evaluated before the call happens

[4]Actually, this process is performed on a per-group-of-mutually-recursive-functions basis, not per-function; for simplicity though, the algorithm is described disregarding mutually recursive function, as the simplification doesn't change the basic process.

[5]It is important to remember that variables and type variables are not the same thing; variables store values in the source language whereas type variables represent possibly unknown types during type inference.

and an error is raised, but if we choose to instead replace the application with the definition, `x` is evaluated to false, the `else`-branch is taken and `undefined` never gets evaluated - the inlining transformation no longer preserves the semantics of the expression!

To solve this problem, we introduce non-strict semantics: when applying a function to an argument the evaluation of that argument is delayed until the function actually uses it. It is easy to see then, that we have fixed the inlining transformation; `undefined` is not evaluated before the call, but has its evaluation suspended until the `else`-branch is taken which, in our example, doesn't happen. The inlined and the non-inlined version of the expression have the same semantics again.

The method used to allow code to have non-strict semantics consists of creating an object, known as a *thunk*, to contain the unevaluated expressions that make up the arguments to functions. When the value of such an expression is requested, the thunk evaluates its expression and memoizes it, to avoid having to perform the computation more than once, should it be needed again later. This mechanism is known as *lazy evaluation*.

4 Implementation

4.1 Syntax and features

Lambdascript borrows much of its feature set from Haskell. In fact, the implemented feature set is nearly a subset of Haskell, albeit with more annoying syntax. This section gives a brief rundown of the language's features and their syntax. Starting from the basics, functions and constants are defined just like in Haskell, including lambda functions.

```
a_constant = 10;
foo a b c = a * b + c;
square = \x -> x*x;
divide = \x y -> x / y;
nestedLambdas = \x -> \y -> x - 2*y;
```

Data type declarations look almost the same as in Haskell, the only difference being that multiple constructor arguments need to be separated by commas.

```
data Maybe a = Nothing | Just a;
data Pair a b = Pair a, b;
```

Functions may employ pattern matching and guard expressions for selection. Lambda functions may use pattern matching, but no guards and only one pattern per argument.

```
isJust (Just x) = True;
isJust _        = False;
isBig x | x > 10    = True;
        | otherwise = False;
fromJust = \(Just x) -> x;
```

While not mandatory, it's possible to specify type signatures as one would in Haskell. Function application works the exact same way, including partial application. Obviously, functions are first class values.

```
addThree :: Int -> Int -> Int -> Int;
addThree a b c = a + b + c;
addTwo :: Int -> Int -> Int;
addTwo = addThree 0;
addTwoList :: [Int] -> [Int] -> [Int];
addTwoList = zipWith addTwo;
```

`if` and `case` provide selection within expressions and, unsurprisingly, work the same as in Haskell.

```
safeDiv x y = if y != 0 then Just (x / y) else Nothing;
classifyNumber x = "For your information, " ++
  case x of
    0             -> "x is zero";
```

```
    n | n > 100    -> “x is big”;
      | otherwise -> “x is small”;
    ;
```

A pair of curly braces is used to denote local bindings. This corresponds to Haskell's `where` clause.

```
reverse xs = go [] xs {
    go a (x:xs) = go (x:a) xs;
    go a _      = a;
  };
```

Lambdascript is also, as previously mentioned, a lazy language, delaying evaluation of function arguments until they are actually used. The following functions are thus perfectly fine and will both return 6 as expected, as `undefined` is never used and so doesn't get evaluated.

```
foo = sum (take 3 [1,2,3,undefined]);
bar = alwaysReturnSix undefined {
        alwaysReturnSix x = 6;
      };
```

Despite not having type classes, basic arithmetic is still polymorphic. To accomplish this, rather than making the type of the basic numeric operators `Num a => a → a → a` as in Haskell, we give them the type `Num a → Num a → Num a` and define our numeric types as `Num Int` and `Num Double` rather than `Int` and `Double`. To see this in action, consider the following example.

```
addOne :: Num a -> Num a;
addOne x = x + 1;
```

A simple function that adds one to a numeric value. Now, when we generate the code for this function, we just use JavaScript's built-in `+` operator to create the following output. See section 4.8 for details about how code generation is done.

```
function addOne(x) {
    return eval(x) + 1;
}
```

The function's argument is evaluated[6] and then just added to one, the `+` operator basically just lifted as-is from the Lambdascript function to the output; as we can see there is no type arguments or other type class magic going on. This trick relies of the fact that JavaScript doesn't have separate integer or floating point types but rather a single `number` type, meaning that Lambdascript's addition operation, for example, can always be translated into `a + b`, regardless of whether `a` and `b` are of type `Num Int` or `Num Double`. This would also have worked with target a language like C, which does have separate numeric types but overloads its arithmetic operators, but obviously not if our target was assembly code.

With this basic Lambdascript-fu under the belt, we can now proceed to take a good look at the machinery that makes it all work.

4.2 Data representation

The algebraic data types and tuples used by Lambdascript have no equivalent in JavaScript. Similarly, thunks are also entirely foreign concepts. Therefore JavaScript representations of these concepts had to be defined. As a thunk represents a computation that returns a value and may or may not have been evaluated, it is quite similar to a nullary function with a static variable keeping track of whether the computation has been evaluated or not and, if the thunk has indeed been evaluated, what its value is. This is a nice and simple way to model thunks, but requires at least one function call to be performed when reading a thunk's value regardless of whether evaluation has already been performed or not. Instead, we make the evaluation status and value of the thunk available to the outside world by using a simple object. The thunk and evaluate operations can be described in pseudo code as follows.[7]

[6] See sections 4.2 and 4.8 for more information about thunks and evaluation.

[7] The actual generated code is quite a bit shorter in order to conserve space, but the short member names actually generated would make for a rather cryptic example and so

```
Thunk(x) = {
  evaluated: false,
  value: function() {
      this.evaluated = true;
      this.value = x.performComputation();
      return this.value;
    }
  };
Eval(t) = if t.evaluated then t.value else t.value();
```

Note that Thunk and Eval are not actual functions but low-level descriptions of the code emitted for those operations and that `x` and `t` in the above example denote entire expressions; the thunk creation function is actually impossible to create in JavaScript as the computation `x` would be reduced to a value before entering Thunk's function body. Even if it were possible, performing a function call every time a thunk is created or evaluated would add quite a lot of computational overhead to the generated code. This example also uses `this.value = x.performComputation()` to express that the computation described `x` is evaluated and the result assigned to `this.value`; this is a simplification, as in the actual implementation, this method call is inlined into the thunk itself, to avoid making unnecessary calls.

While it might seem odd or even stupid to reuse the `value` field of the thunk to store both the suspended computation and its value, this is the result of a conscious design decision. After the thunk is evaluated, the closure allocated for the computation needs to be freed so that the garbage collector may reclaim it. While it would be quite possible to add a `closure` field and extra code to clear that field upon evaluation, thunks are created frequently enough that any extra code added to their creation would add a considerable amount of bulk to the generated code, as well as computation overhead.

In JavaScript, complex values can be represented either using arrays or objects. Since arrays have *O(1)* random access complexity, it made sense to use those as much as possible. Algebraic data types are represented by

are here replaced by longer names that better describe the purpose of the fields.

an array where the first element contains the ID of the constructor used to create the value, and subsequent elements contain thunks of the constructor's arguments. Consider the following code.

```
data Foo = Foo | Bar Int | Baz Double, Foo;
foo = Foo;
bar = Bar 10;
baz = Baz 1.1 Foo;
```

The function `foo` will be represented in JavaScript by the value `[0]` - the first constructor (ID 0) with no arguments. The function `bar` will be represented by `[1, Thunk(10)]` and `baz` will turn into `[2, Thunk(1.1), Thunk([0])]`.

Tuples are implemented nearly the same way; a tuple of n elements is simply an array of n thunks. While one would be correct to argue that a tuple is nothing but a special case of algebraic data types, that Lambdascript uses tuples quite heavily internally and that, being single constructor types, tuples can have only one constructor ID motivates giving them this special attention.

Functions, while perfectly representable in JavaScript, also have a special representation to facilitate partial application without having to generate quite a lot of boilerplate code. A function `a → b` is simply represented by a JavaScript function with a single argument of type `Thunk a`, returning a value of type b. A function `a → b → c`, however, is represented by a JavaScript closure that takes a single argument of type `Thunk a` and returns a JavaScript function of type `b → c`. To see this scheme in action, let's look at the following example.

```
foo :: Int -> Int -> Int;
foo a b = a + b;
```

This nice, simple example will turn into roughly the following JavaScript when compiled.

```
function foo(a) {
    return function(b) {
```

```
            return eval(a) + eval(b);
        };
    }
```

The main benefit of this representation is that partial application becomes trivial to implement. The downside of course is that, disregarding inlining, total application of an n-ary Lambdascript function results in n JavaScript function calls.

4.3 Tail call elimination

JavaScript doesn't have a `goto` keyword or any other method of jumping to arbitrary locations in code, which means that implementing tail call elimination in a performant way is not possible for the general case. Instead, we have to resort to a method called trampolining whereby all functions are entered via a helper function, called the trampoline. The trampoline takes the function to call as an argument and calls it. If that function returns a value, the trampoline returns that value in turn. If, however, a function wants to tail call another function, it instead returns an object, containing a reference to the function to be tail called and a list of arguments to pass on to said function. When such a return value is detected, the trampoline calls the returned function with its accompanying parameters. In Lambdascript, this is implemented in the runtime library, and looks as follows.

```
function trampoline(f, as, thisptr) {
    var result = {func: f, args: as};
    for(;;) {
        result = result.func.apply(thisptr, r.args);
        if(!result.func) {
            return result;
        }
    }
}
```

This implementation quite closely follows the description outlined above. The only thing that sticks out is the `thisptr` parameter. This parameter is needed because evaluating a previously unevaluated thunk is a method call rather than a function call since the result of the evaluation needs to be stored in an object, which means that the call must be made as a method call on the closure representing the thunk.

4.4 Modules

A Lambdascript program consists of one or more named modules, the name of each module determined by the name of the file in which it resides. Hierarchical modules are not supported. In a break with Haskell conventions, module names follow the same naming rules as variables; names must start with an underscore or a lower case letter, the rest of the name may upper and lower case letters, underscores and single quotes ("primes.") The reason for this is a completely arbitrary decision that single word module names just look nicer in lowercase.

One module, the one passed to the `lsc` compiler on the command line, is considered the "main" module of the program. Only one module may be specified on the command line, any other modules making up the program are inferred by recursively following `import` statements in the main module. Imported modules are searched for on the compiler's library paths; for instance, if the main module imports a module called `std`, the first file called *std.ls* found on any library path will be assumed to contain this module. Should any imported module not be found, compilation is halted immediately. This mode of operation is quite similar to invoking GHC with the `--make` parameter. The main module only differs from any other modules making up a program in that it's the root node in the dependency tree, as mentioned, and that it may be renamed using a command line parameter whereas all other modules have their name fixed by their file name.

The list of modules to be compiled is built using the following algorithm.

```
function AddModules(Modules, CurrentModule):
    For each ImportedModule in imports(CurrentModule):
```

```
            prepend ImportedModule to Modules
            Modules := AddModules(Modules, ImportedModule)
        End
        Return Modules
    End
    ModuleList := AddModules([], MainModule)
    For each Module in ModuleList:
        Remove all instances of Module but the first from ModuleList
    End
```

As we can see, a dependency cycle will lead to the algorithm recursing until it runs out of stack space. Adding safeguards against this hasn't been a priority and so hasn't been done even though doing so would be fairly trivial.

After the list of modules is generated, the modules are lexed and parsed. Assuming that there are no syntax errors, the modules are then desugared, turning higher level niceties into equivalent expressions using simpler building blocks, and type checked in order, beginning with the first module in the list. All functions exported from all of a module's imports are added to the initial list of assumptions used to type check that module. During type checking, every subexpression in every module's abstract syntax tree is annotated with its inferred type. After type checking is complete, the annotated AST is passed on to intermediate code generation. The intermediate representation is then run through an optimization pass, and the resulting code is then finally turned into JavaScript and written to file. The following sections explain each step along this road in further detail.

4.5 Lexing and parsing

The first step in the compilation of each module is to split the source text into tokens for further analysis, called "lexical analysis" or "lexing", and to parse those tokens. We initially considered using the *haskell-src* package[8] for these tasks, but decided against it; as the full Haskell feature set was not

[8]A Haskell library for lexing and parsing Haskell code

to be implemented, *haskell-src* would have been far too lenient in parsing unsupported constructs such as type class contexts, and dealing with them post-parsing seemed like a clunky, cumbersome solution.

Instead, we chose the BNF Converter [Forsberg, Ranta 2005]. Similar to *haskell-src*, BNFC defines an abstract syntax tree as well as a lexer and a parser, but does so for a user-specific LR(1) labeled BNF grammar rather than standard Haskell. After eliminating *haskell-src* from the selection process, we decided to restrict Lambdascript to a context-free grammar to enable the use of an LR(1) parser generator such as BNFC, as context sensitive languages are trickier to parse and doing so would have taken time that would have been better spent realizing the project's main goals.

4.6 Desugaring

Desugaring is the process by which we turn high level, nice-to-have constructs into labyrinthine combinations of simpler language constructs. This is done to make the type checking and code generation phases of compilation quick and easy without the user ever having to know what his shiny function definitions *really* look like. The desugaring done by the Lambdascript compiler mainly concerns itself with turning the nice, clean mathematical-looking function definitions written by the user and converting them into lambda functions with `cases`. There are three main steps to this; casefication, case merging and lambdafication. To illustrate this process, let's start with a simple example.

```
foo 0 0 0              = -1;
foo a b c | a > b      = 10;
          | otherwise = a*b*c;
```

Since pattern matching works the same way in both function definitions and `case` expressions, we don't want to write code for type checking and generating code for both of them. We want to somehow turn one into the other and just deal with that one; this is what I'm calling casefication. We turn each definition of the function into a simple assignment, with a case expression for the value being assigned.

```
foo = case (v0, v1, v2) of
        (0, 0, 0) -> -1;
        ;
foo = case (v0, v1, v2) of
        (a, b, c) | a > b     -> 10;
                  | otherwise -> a*b*c;
        ;
```

The observant reader will note that this is no longer valid code. Not only does the first definition of `foo` shadow the second one, but we've also introduced a set of unbound variables. This is where the second step comes in; we merge all alternatives of case expressions assigned to an identifier of the same name (in this example, `foo`) into one big `case` expression.

```
foo = case (v0, v1, v2) of
        (0, 0, 0)             -> -1;
        (a, b, c) | a > b     -> 10;
                  | otherwise -> a*b*c;
        ;
```

That's better; now there is only one definition and so the `case` expression would be perfectly valid, if just the variables `v0`, `v1` and `v2` were in scope. This makes the third major step of the desugaring fairly obvious. `foo` was originally a function with three arguments, and our partially desugared version has three free variables. To fix things up, we simply wrap the `case` expression in one lambda function for each free variable.

```
foo = \v0 -> \v1 -> \v2 ->
          case (v0, v1, v2) of
            (0, 0, 0)             -> -1;
            (a, b, c) | a > b     -> 10;
                      | otherwise -> a*b*c;
            ;
```

This is the final form that our function takes; now we only have one pattern matching construct to deal with, and one kind of function definition. The only thing left to do is to do the same to local function definitions, something which is accomplished by simply traversing the abstract syntax tree of every expression to find all local bindings, applying the same transformation wherever one is found.

4.7 Type checking and annotation

Type checking and annotation of Lambdascript quite closely follows the algorithm described in [Jones 1999], the reference implementation of Hindley-Milner type checking and inference as used by Haskell, in order to ensure compatibility with Haskell as far as possible with Lambdascript's reduced features set. The only substantial differences are that no checking of class contexts is performed, the monomorphism restriction is not implemented and kinds other than * are not allowed, all direct consequences of Lambdascript not supporting type classes.

The type checking process is done within a monad that keeps track of the current substitution and the creation of fresh type variables. Modules are checked one by one, with any functions imported by a module added to its initial list of assumptions before type checking commences. Top level definitions are then divided into bind groups, where a bind group consists of all mutually recursive functions; no polymorphism is possible between functions in the same bind group. These groups are then sorted and type checked in dependency order, the assumptions and substitutions from each group being added to the global environment as checking progresses. During type checking, every subexpression of the module is annotated with its inferred type. After the entire module has been checked, all type annotations are further resolved into the most concrete type possible using the module's complete substitution.

4.8 Intermediate code generation

When the code is guaranteed to be type correct and thusly annotated, we get to the phase where we actually do something with the code. While it would be quite possible to generate the final Lambdascript directly from the AST, this would not be a good idea. In general we want to perform at least some sort of simplifications and other transformations to the program code before writing its final form to a file, and the AST is not a good representation for making these transformations. Not only would it mean that any change in the AST would require all such transformations to be updated and retested, which due to the use of BNFC means that any minor change in the source grammar would propagate all the way through the compiler, but many of the transformations we want to make only make sense in the context of our target language, rather than the relatively high-level Lambdascript. Applying these transformations to the generated JavaScript itself is of course possible, but would require an enormous amount of text-juggling, including a complete parser and analysis framework for the JavaScript language.

Much better then to introduce an intermediate representation as an abstraction layer. This intermediate representation maps more or less 1:1 onto JavaScript, but has the advantage of being stored in a symbolic format that's easy to work with rather than text. The IR is primarily made up of two algebraic data types, one representing expressions and one for statements, and contains operations for expressing creation and evaluation of thunks, creation and indexing of arrays, calls and tail calls to functions as well as returning from and creating them, `if`-statements, loops, basic arithmetic and boolean logic. In the IR, data is represented in the same way as in the target language; see section 4.2 for an in-depth discussion of this.

4.8.1 Generating patterns

The IR generated from a pattern match, which when we get to the point of IR generation only occur in `case` expressions, has two components: a boolean expression determining whether the pattern matches the checked expression and a block statement binding any local variables used in the

pattern. Any accompanying guard expression gets translated into another boolean expression, coupled with its parent pattern, denoting whether the expression holds or not. These expressions and statements are then used to generate a series of nested `if`-statements, choosing the appropriate case to return. To avoid completely confusing the reader with a long-winded, abstract text, let's look at a concrete example instead.

```
case exp of
  (Just x) | x < 2  -> "less than two";
           | x > 10 -> "more than ten";
  (Just 7)          -> "seven";
  ;
```

The above example is a fairly standard `case`-expression with two patterns, one of which also has two guards. Let's look at what's generated from it.

```
var result;
for(;;) {
    if(eval(exp)[0] == 1) {
        if(eval(exp)[1] < 2) {
            result = "less than two";
            break;
        }
        if(eval(exp)[1] > 10) {
            result = "more than ten";
            break;
        }
    }
    if(eval(exp)[0] == 1 && eval(exp)[1] == 7) {
        result = "seven";
        break;
    }
    error("Non-exhaustive pattern in case expression!");
}
```

At the end of each `if`-statement, a `break` statement is inserted, and all `if`-statements for the current case expression are then inserted into an infinite loop, to implement the fall-through behavior necessary to select the proper case in an expression such as the following. The loop body is only ever executed once, as the `break` inserted into every case acts as a forward-goto, immediately exiting the loop.

By looking at the order of the emitted `if`-statements, we can see that patterns are checked first, in order. If a pattern matches, then any accompanying guard expressions are checked. If none of them match, control falls through to the next pattern; the `break` at the end of each case body ensures that control is transferred to the end of the loop as soon as one guarded pattern is matched. This is quite obviously the intended fall-through behavior.

4.8.2 Generating expressions

Compared to its pattern counterpart, IR generation for expressions is trivial as most Lambdascript expressions map directly onto their JavaScript equivalents. There are two concepts here that are slightly more interesting though; thunk handling and statement expressions.

Thunks are created when a function or a constructor is applied to an expression, and when an expression is bound to an identifier either at the top level or local to another function; in both cases, the thunk is bound to an identifier. A thunk is evaluated whenever the identifier it is bound to is dereferenced, memoizing and returning the value yielded by the computation represented by the thunk.

Some expressions, can't stand on their own; they require one or more statements to be evaluated before they can be computed. This is the case for `case` and `if` expressions, both of which are implemented using JavaScript `if`-statements. However, thunks return a value and are thus considered expressions, but if we can't stuff these supporting statements into the same thunk as the expression they support, then they will be executed prematurely, breaking our non-strict semantics! The solution to this problem is the *statement expression* which, quite simply, is an expression consisting of

another expression and a list of supporting statements. Using this construct, we can fit both expressions and statements into thunks.

4.9 Optimization

After the AST is converted into a low-level, manageable, symbolic intermediate representation, the resulting IR is run through an optimization pass. Optimization is a bit of a misnomer though, considering that it mainly concerns itself with making the compiler's output a little more compact and readable, the one exception being performing tail call elimination. To this end, a framework for traversing the IR, applying a function to each expression or statement encountered. While this spot-transformation approach doesn't do very well for higher level optimizations, such as tail call elimination which actually has to be done outside this traversal framework, its purpose isn't to be a comprehensive infrastructure for all sorts of performance enhancements, but merely to allow rules for local transformations to be written and applied without having to deal with the larger structure of things.

The transformations applied are quite simple affairs; zero comparisons (`x == 0` and `x != 0`) are turned into `!x` and `x` respectively in order to make the code a little more compact. This has a surprisingly large effect, as checking whether a constructor ID is zero or not is a fairly common operation. Boolean expressions are reduced to get rid of any constants, which are liberally introduced during IR generation. Always-true conditional branches, that is, expressions such as `if(true) {...}` are reduced to just the body of the `if`-statement and any `else`-branches thereof of always-false branches are removed as they will obviously never be reached. Constant indices into constant arrays (for example `([foo, bar, baz])[2]`, also liberally introduced during IR generation) are replaced by the value at the given index. Finally, any code that is otherwise unreachable, for example due to coming after an unconditional `break` or `return`, is also removed.

While the performance effects of this optimization pass are negligible, the size of the generated code is reduced by a factor of 1/3 for some non-trivial examples. The JavaScript produced is also quite a bit more readable with

these optimizations than without.

4.10 Intermediate code to JavaScript

Upon completion of the optimization pass, our IR has reached its final form and we're ready to actually translate it into our target language. Since all IR concepts map fairly directly onto JavaScript, this is a trivial affair. For each module in the program, JavaScript code to create an object is emitted to the output stream. Within each module object, every function of that module is then omitted by traversing their respective function bodies, emitting the direct JavaScript translation of each IR construct encountered. If the user has requested tail call elimination to be disabled, tail calls are replaced by "normal" calls and all trampolining instructions are ignored during this stage.

After this step has been performed, the entire output stream is written to the output file, and the user finally has a Lambdascript program, ready for inclusion on a web page or loading into a command line JavaScript interpreter.

4.11 Runtime library

To simplify code generation, certain Lambdascript operations are delegated to a runtime library, written in JavaScript. These operations include trampolining, constructing values of algebraic data types, marshalling data across the JavaScript/Lambdascript boundary, and comparing complex values.[9] List concatenation is also handled by the runtime library as the `++` operator is a built-in rather than a function, which would make reliance on the standard library for its existence a bit unwieldy. During compilation, all comments and blank lines are stripped from the standard library, and its code is then copied verbatim to the beginning of the resulting .js bundle.

[9]Remember, JavaScript has no integer type. This counter-intuitively means that in Lambdascript, integer division is actually slower than its floating point counterpart.

4.12 Output format and JavaScript interface

The code produced by the the compiler of course needs to be callable from native JavaScript code, or it's not much use to anyone. To this end, a nice, clean interface needed to be defined for exporting Lambdascript functions to the outside world. At a high level, the compiler handles this by creating a JavaScript object for each module, with a method for each exported function. The name of each object is determined by the file name of the module it represents; for instance, functions exported from the module *foo.ls* will reside in a JavaScript object called `foo`. These objects are then all dumped to a single .js file, named *a.out.js* unless the name is overridden by command line arguments. While writing all modules to the same file leads to quite a lot of code duplication in the case where two independently compiled Lambdascript programs are used on the same web page, it also makes life easier on the developer by not requiring her to hunt down and write `<script>` tags for every single module used in her program. Adding a compiler switch to enable writing each module to its own file would be a nice and easy solution to get the best of both worlds, but not really a priority here. Let's assume that we have the following module, called *lists.ls*.

```
export reverse, head;
reverse :: [a] -> [a];
reverse = ...
head :: [a] -> a;
head = ...
```

As previously stated, this module compiles into a JavaScript object. The structure of this object is fairly simple; each exported function appears as a member twice, once with its name preceded by an underscore, and once under its actual name. The underscore-prefixed version is the raw, unmarshalled function, using Lambdascript's calling convention, for use by other Lambdascript modules. The member sharing a function's actual name is a marshalled version, ready to be called by native JavaScript.

```
var lists = {
```

```
        _reverse: function(xs) {...},
        _head: function(xs) {...},
        reverse: export(_reverse),
        head: export(_head)
    };
```

After including the resulting *a.out.js*, which contains one object like the one above for each module making up the program, in our web page, we can then call these functions from native JavaScript.

```
var name = prompt("What's your name?");
alert("Neat! Your name backwards is " + lists.reverse(name) +
    "and the first letter of your name is " + lists.head(name));
```

Obviously, "create objects, dump to file" is not the whole story here; Lambdascript supports lazy evaluation and algebraic data types, something JavaScript has no notion of. To free the developer from having to think about these differences, arguments and return values passed between Lambdascript and JavaScript are marshalled by the runtime library.

Not all Lambdascript concepts make sense in JavaScript though, and conversely, not all JavaScript values have a place in a Lambdascript context. What's the JavaScript value of `Left "Oh snap!" :: Either String ()`, for example? Similarly, `{foo: 10, bar: "I like kittens"}` doesn't really have a sensible Lambdascript counterpart. For this reason, the only values marshalled to and from Lambdascript are lists, strings (which are just lists of characters in Lambdascript, but a distinct type in JavaScript) and primitive types. The following table details the type conversions performed.

LS	JS
[Char]	string
[a]	Array
Int	Number
Double	Number

Lambdascript ↔ JavaScript type mappings.

Returned functions are not marshalled, as they are never returned from a function exported to JavaScript. The type `a → (b → c)` quite obviously is

equivalent to `a → b → c`, which implies that any m-ary Lambdascript function returning another n-ary function will get marshalled into a $m+n$-ary JavaScript function. Functions are not marshalled from JavaScript to Lambdascript, as arbitrary untyped functions could not only decide to return data that is not well typed, possibly crashing the entire program, but also decide to perform any side effect imaginable, which might lead to quite subtle bugs in the presence of lazy evaluation.

4.13 Performance

While this thesis doesn't concern itself with generating code that runs like the wind, and performance sensitive number crunching is not really Lambdascript's intended problem domain anyway, it would still be nice to get a feel for how the generated code performs. To this end, a few microbenchmarks were quickly coded up, implementing simple operations in both idiomatic Lambdascript and idiomatic JavaScript. The scripts were then run using a standalone binary build of Mozilla's SpiderMonkey JavaScript engine, comparing the lowest obtained running times of the two scripts over several runs.[10] While microbenchmarks are a poor tool for measuring performance characteristics, they can at least give a pointer or two as to where further optimizations might be worthwhile.

For these microbenchmarks, with the default compiler flags, the code generated by the Lambdascript compiler runs in somewhere between 19 and 65 times the time of the respective JavaScript counterparts, depending on the benchmark. While this might seem incredibly slow, it is important to note that the operations benchmarked are heavily exercising Lambdascript's thunk implementation without performing much in the way of actual work. Interestingly, execution time drops by nearly half when tail call elimination is disabled; the trampolining required to work around JavaScript not having a `goto` keyword really takes its toll on performance.

The first microbenchmark calculates the first 100 000 Fibonacci numbers

[10] Any deviation from the lowest run time is a result of external factors such as scheduling, page faults, etc. so it doesn't make much sense to include those factors in the comparisons.

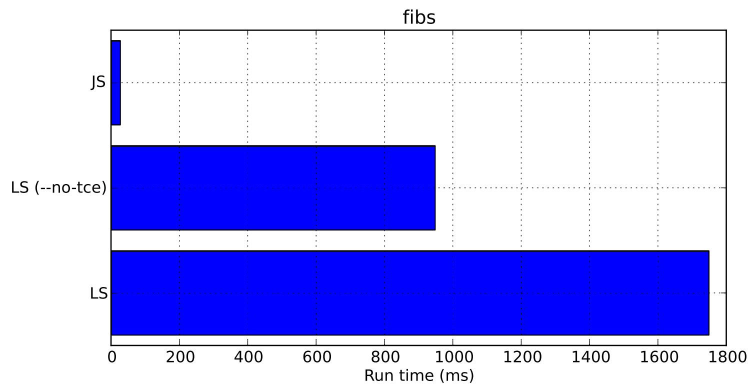
using a memoizing algorithm. The Lambdascript version uses the more or less canonical Haskell way of doing this; an infinite list using `zipWith` on itself.

```
import std;
main = take 100000 fibs;
fibs = 1:1:zipWith (\a b -> a+b) fibs (tail fibs);
```

The JavaScript version uses the same technique, but replaces the lazy list with an array and a loop.

```
var fibs = [1,1];
for(var i = 0; i < 99998; ++i) {
    fibs.push(fibs[i] + fibs[i+1]);
}
```

This was the microbenchmark that showed the worst results. The Lambdascript version, when compiled with trampolining enabled, is *more than 60 times slower* than the native JavaScript one, by far the slowest of the tests. This is perhaps unsurprising, as there is quite a lot of thunking and very little else going on in this test. Even the most conservative inlining would likely have worked wonders for these numbers. Another indication that the major part of the program's run time is spent creating and evaluating thunks, which amounts to creating and calling lots of JavaScript closures, is that the program runs in roughly half the time with trampolining disabled. As the unoptimized trampolining used by Lambdascript effectively turns every function call into *two* function calls, disabling it cuts away half of the function calling penalty; it's quite likely that just as much of the run time left is also wasted calling functions, most of which could have been inlined.



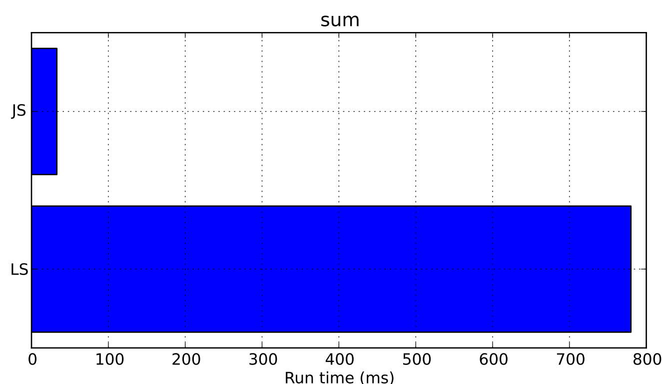
The second microbenchmark creates a list of 100 000 numbers and sums them. The Lambdascript version uses a rather cumbersome construction involving the `case`-keyword to force strictness in the accumulator, making it rather slower than what could have been accomplished with explicit strictness annotations.

```
sum (x:xs) n = case n of
                 0  -> sum xs x;
                 n' -> sum xs n';
               ;
sum _ n      = n;
main = sum (replicate 100000 1) 0;
replicate 0 _ = [];
replicate n x = x : replicate (n-1) x;
```

The JavaScript version creates the list beforehand, to reflect that under normal circumstances, the list of numbers would have to have been created separately from the *sum* function.

```
var a = [];
for(var i = 0; i < 100000; ++i) {
    a.push(1);
}
var sum; for(var i = 0; i < 100000; ++i) {
    sum += a[i];
}
```

Our Lambdascript program does a bit better in this test, taking roughly 24 times as long to run as the native JavaScript version. While the `case` construction in the `sum` function ensures that we don't get a huge chain of thunks built up, thunks are still created and evaluated though; considering this, it's a little surprising that this program performs so much better, relatively speaking, than the *fibs* test. This can likely be attributed to the fact that the generating function, `replicate`, does less thunking than its *fibs* counterpart.



The third and final microbenchmark filters all occurrences of the letter *a* out of a fairly large string. In Lambdascript, this is done using the standard `filter` function.

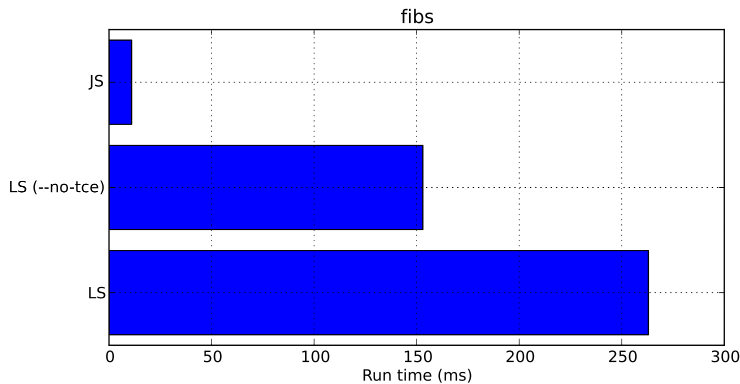
```
import std;
str = ...; -- A string of about 30 000 characters
main = filter (\x -> x != 'a') str;
```

The JavaScript version instead uses JavaScript's built-in string manipulation functions.

```
var str = ...; // The same large string
var out = '';
for(var i = 0; i < str.length; ++i) {
    var c = str.charAt(i);
    if(c != 'a') {
        out += c;
    }
```

```
}
```

This test gives us just about the same performance characteristics as the *sum* one; Lambdascript is about 24 times slower and, just as with the *fibs* microbenchmark, disabling trampolining does away with almost half the run time.



These microbenchmarks all hint at the constant creation and evaluation of thunks as the major culprit when it comes to performance issues. As no inlining or strictness analysis is performed on the intermediate code, this is rather expected. Interestingly, an earlier version of the compiler, when modified to emit `thunk` and `eval` operations as no-ops, effectively turning Lambdascript into a strict language, performance on most tests suddenly came within 5-10 % of each respective native JavaScript counterpart, giving further support to the theory that measures to minimize thunk usage, such as strictness analysis, would have quite an impact on these numbers.

5 Future work

While a compiler for a functional language is a first step towards bringing the benefits of functional programming to the web development community, there is still much to be done if Lambdascript is to make a serious contribution to the field.

- Adopting actual Haskell syntax. Allowing actual Haskell code to be compiled and executed on any JavaScript-capable web browser would

obviously bring greater benefits than just another functional language; not only would it get rid of the somewhat clunky concessions made in order to be able to parse Lambdascript using a LR parser, but also enable a large number of libraries already written in Haskell to be compiled and used to develop web applications.

- Adding type classes. In order to do anything truly interesting with a Haskell-like languages, type classes are a hard requirement. Adopting Haskell syntax would also be a fruitless endeavor unless this major feature is supported.

- Replacing BNFC with *haskell-src-exts*. This is most likely the easiest way to move to Haskell syntax. It would also enable better error reporting and an abstract syntax tree that can be a bit more separated from the concrete syntax than what BNFC allows, resulting in a friendlier compiler for the user and cleaner code overall.

- Introducing basic optimizations. While Lambdascript is already fast enough for most web development tasks, the performance of the generated code still leaves a lot to be desired. As previously noted, trampolining and laziness make up the vast majority of the performance gap between Lambdascript and native JavaScript. Although trampolining would seem to be a necessary evil to implement tail call elimination within the constraints of JavaScript for the foreseeable future, it can be easily eliminated in the common case of a function tail calling itself. Laziness can be minimized by adding some form of strictness analysis to the optimization pass, and both problems would likely reap large benefits by even a conservative use of inlining. Another interesting possibility would be to work closer with the optimizer built into JavaScript engines by generating more idiomatic code.

- Adopting an interface to native JavaScript code, allowing Lambdascript code to call back into JavaScript. Not only is this necessary to enable IO within Lambdascript, enabling entire web applications to be written without having to fall back to JavaScript at all, but also allows

JavaScript code to provide function callbacks to Lambdascript code.

- Developing a functional web application client library. While developing web applications in Lambdascript may be nicer and safer than using pure JavaScript, it is still essentially the same imperative reading-and-writing-the-global-state-all-over-the-place style DOM manipulation as before. A better approach is clearly desirable, and a functional language with an expressive type system would make an excellent basis for such a library. Obviously, the implementation of type classes would be a prerequisite for any attempt at such a library, and Haskell extensions like GADTs, existential types and type families might also be quite beneficial.

6 Conclusions

While the decision to use BNFC for lexing and parsing enabled the project to get up and running quickly, a series of problems with the approach became apparent as work went on. As BNFC generates an abstract syntax definition from the source LBNF grammar, the abstract syntax tree becomes tightly coupled not only with the language's concrete syntax, but also with the specific way the source grammar file is structured. When Lambdascript's grammar needed to be updated along the road the abstract syntax definition changed as well, forcing a lot of maintenance work for the type checker and intermediate code generator, maintenance that could have been avoided by using a more flexible parser generator. Desugaring of monadic syntax might also have been done better in the parser itself had a less general tool been used, rather than between parsing and type checking as it is done in the Lambdascript compiler. The most problematic drawback from using BNFC, however, is that it doesn't pass source context such as line numbers on to the later stages of compilation, making it impossible to provide useful error messages when type checking fails. As type errors are by far the most common errors when writing code in any statically typed language, this is a huge problem. In retrospect, this problem alone makes it quite clear that

choosing *haskell-src* and dealing with the unsupported language constructs post-parsing would have been the better decision.

Although the performance numbers obtained may look disappointing, it's important to remember that they come from microbenchmarks, comparing very simple basic operations such as iterating through a list doing very little work on its elements, using standard functional code and similarly idiomatic native JavaScript. In fact, considering that the main performance penalties are incurred by laziness and trampolining, both relatively low hanging fruit with regards to optimization, the generated code performs surprisingly well. While admittedly no tests that seriously stress the memory allocator have been written, Lambdascript seems to be getting along fairly well with JavaScript garbage collectors that are used to programs that mutate a small set of long-lived values rather than allocate a massive amount of new ones continuously. There are likely also incremental gains to be made by tweaking the thunk format somewhat.

While there are indeed still problems to be solved with Lambdascript, they are not intrinsic to the problem at hand, the language or the compiler, but the consequences of either a poor choice of tools, in the case of the BNFC related problems, or a deliberate restriction of the problem domain. The Lambdascript compiler provides a quite solid base for future improvements and meets its goal of producing working, cross-browser compatible, relatively compact JavaScript. Adding the most important features the compiler is currently lacking - FFI, Haskell source compatibility, useful error messages and better performance - can be accomplished with a modest effort, to make Lambdascript a truly useful tool for functional web development.

References

[Cherry 2010] http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth, August 2011

[Golubovsky 2007] YCR2JS, a Converter for YHC Core to JavaScript, http://www.haskell.org/haskellwiki/Yhc/JavaScript, August 2011

[Dijkstra 2010] Haskell to JavaScript Backend, http://utrechthaskellcompiler.wordpress.com/2010/10/18/haskell-to-javascript-backend/, August 2011

[Mackenzie 2011] Experimental GHC with GHCJS built in and Cabal support for JavaScript files, http://www.haskell.org/pipermail/haskell-cafe/2011-May/091897.html, August 2011

[Björnesjö, Holm 2011] JSHC - JavaScript Haskell Compiler, https://github.com/evilcandybag/JSHC/blob/master/report/project-report-2011-06-16.pdf, August 2011

[Google Web Toolkit] http://code.google.com/webtoolkit/, August 2011

[Pyjamas] http://pyjs.org/, August 2011

[GWT projects] http://code.google.com/webtoolkit/examples/#real-world-projects, August 2011

[Forsberg, Ranta 2005] The Labeled BNF Grammar Formalism, http://www.cse.chalmers.se/research/group/Language-technology/BNFC/doc/LBNF-report.pdf, August 2011

[Jones 1999] Typing Haskell in Haskell, Haskell Workshop, Paris, France, October 1999

[Milner 1978] A Theory of Type Polymorphism in Programming. Journal of Computer and System Science (JCSS) 17