THESIS TITLE: RacketScript: A Racket to JavaScript Compiler

AUTHOR: Vishesh Yadav

M.S. Thesis Approved as an elective towards the Master of Science Degree in Computer Science.

Thesis Advisor

Date

Thesis Reader

Date

Thesis Reader

Date

Thesis Reader

Date

Thesis Reader

Date

GRADUATE SCHOOL APPROVAL:

Director, Graduate School

Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature

Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
RacketScript: A Racket to JavaScript Compiler

Vishesh Yadav
Northeastern University
Boston, MA
vyadav@ccs.neu.edu

Abstract. We present RacketScript, a Racket-to-JavaScript compiler that incorporates features and data structures from both Racket and JavaScript, enabling RacketScript programs to interoperate with libraries from either language. In addition, RacketScript aims to generate human-readable code with performance and memory usage comparable to handwritten JavaScript. To achieve such goals, our compiler must approximate the semantics of some features such as tail-call optimization and continuation marks. This dissertation describes these and other key design decisions. To evaluate the effectiveness of our design, we used RacketScript to implement an event-driven game library, several popular games, and a sandbox web app (both client and server) that allows other programmers to experiment with RacketScript.

1 Introduction

JavaScript is the assembly language of the web. Almost all major browser based applications are implemented in, or compiled to JavaScript. In recent years, JavaScript has also been used in environments other than browsers, including server-side web applications, game development, desktop applications, and embedded languages. To be relevant, a modern language must compile to JavaScript.

Racket is a modern Lisp-descendant that has a uniquely diverse range of applications. It is widely used as a teaching language, is leveraged by academics as a tool for creating new languages, and has also been successfully deployed in industry. It has not gained much traction in web contexts, however, despite several attempts, due to some features that do not interoperate well with JavaScript.

We present RacketScript, a new Racket-to-JavaScript compiler. It aims to make Racket an effective replacement for all environments where JavaScript is typically used such as web browsers and Node.js applications. In addition, it strives to use existing JavaScript and Racket libraries whenever possible, allowing Racket programmers to leverage both ecosystems and providing tight interoperability between them.

Doing so, however, requires various trade-offs in terms of features, performance, memory, code size and ease of use. On one hand, semantically preserving all features of Racket in JavaScript comes with non-trivial performance
and memory costs. On other hand, JavaScript libraries utilize objects, prototypical inheritance and asynchronous I/O, making their use awkward in Racket.

At a high level, RacketScript relaxes Racket’s semantics in order to improve many of the aforementioned factors such as memory footprint. The following sections describes how RacketScript deals with specific features of both Racket and JavaScript, including some discussion about the pros and cons of these design decisions.

1.1 Outline of this thesis

- Section 2 describes the main differences between Racket and JavaScript, and section 3 discusses the directions that compilers can take to address these differences.
- Section 4 provides a high-level overview of our compiler architecture. Section 5 to section 7 discusses the translation of Racket expressions, bindings and modules.
- Section 8 presents the RacketScript runtime library which implements a subset of Racket primitives and core data structures (e.g. pairs, hash, vectors etc.).
- Section 9 describes how we handle tail calls in JavaScript. Specifically, we show how we translate self-tail-calls to JavaScript’s native loops.
- Section 10 presents our implementation of continuation marks in JavaScript. JavaScript does not allow low-level access to the stack, and thus prevents implementation of continuation marks directly. We also discuss the implementation of Racket parameters, which are based on continuation marks.
- Section 11 describes RacketScript’s foreign function interface to interoperate with JavaScript libraries and data structures.
- Section 12 presents an evaluation of our compiler in terms of practicality and performance. Our evaluation experiments includes (1) an implementation of HTDP’s [10] image and big-bang library that utilizes RacketScript’s FFI, (2) ports of various big-bang games to RacketScript, and (3) comparisons of code-size and performance with previous efforts at Racket-to-JavaScript compilation.
- Finally, section 13 wraps our thesis with a discussion of related work, and the directions RacketScript can take in future.

2 Racket vs JavaScript

JavaScript is standardized in the ECMAScript language specification, and the latest standard is commonly known as ECMAScript 6 (abbreviated ES6) [15]. The last revision, ECMAScript 5 (abbreviated ES5) [14], is ubiquitously supported while ES6 adoption is ongoing. JavaScript was historically inspired by Scheme and hence supports various features found in Scheme. Most notably they are both dynamically typed with automatic memory management and
first-class functions. However, this is where major similarities end and differences start showing up. We highlight some differences below, considering both ES5 and ES6.

- **Proper Tail Calls** Racket guarantees proper tail call implementation, which means that tail calls do not consume stack space. This removes the need for any special loop primitive, as loops can instead be expressed in terms of recursion. ¹ JavaScript on other hand does not support proper tail calls, making a naive compilation of Racket loops susceptible to stack overflow.

- **Continuations** Racket supports first-class continuations by providing control operators such as `call/cc`. This can be used to implement various other language features such as exception handling, co-routines, generators, threads etc. JavaScript does not support continuations but does have exception handling, which could be used to simulate continuations at a huge runtime cost.

- **Scoping** Unlike Racket, JavaScript has a weak notion of lexical scoping. A declared variable is visible throughout its nearest enclosing function, regardless where it was declared in the function and is removed from the environment only when the function returns. ES6, however, introduces a `let` statement for declaring lexically scoped variables.

- **Runtime Safety** JavaScript is known for its weak type system and runtime safety. Types are often coerced implicitly during execution in awkward ways. For some illegal Racket operations that would produce runtime exceptions, JavaScript produces a special primitive value called `undefined` instead. Examples of such operations include reading or writing of an undeclared variable, division by zero, and applying a function to the wrong number of arguments.

- **Numbers** Racket inherits Scheme’s numeric tower, featuring exact and inexact numbers. Exact numbers can be arbitrarily large and won’t overflow. JavaScript numbers, by contrast, are all double precision floating point numbers.

- **Multiple Return Values** An expression in Racket can produce multiple values, while JavaScript functions result in only one value.

- **Function Application** As discussed earlier (in runtime safety), function application in JavaScript does not check the number of arguments passed. Racket on other hand not only checks it, but also allows variable arguments and `case-lambda`, i.e., lambdas with different bodies for different argument arities. ES6 supports the former but not latter, while ES5 supports none. However, in both revisions, when entering a function, a binding named `arguments` is added to the environment which is essentially an array of arguments passed to a function.

¹ Racket’s `do` loops and comprehension forms are compiled into tail recursive functions.
3 Compilation Strategies

Although JavaScript includes higher-order functions, the absence of continuations in JavaScript and its lack of proper tail-call implementation obscures how to compile Racket into this language. This section describes two philosophically opposing approaches to compiling Racket to JavaScript.

3.1 Semantics-preserving compilation

A semantics-preserving compiler produces a JavaScript program that behaves exactly like its Racket source program. Certain features such as continuations and tail calls are implemented by emulating stacks and running programs via a trampolining machine. While this approach is ideal, because it requires no adjustments to programmer’s minds, it suffers from several disadvantages:

- **Poor performance** Preserving Racket’s semantics typically involves implementing an interpreter of some kind in JavaScript, which then evaluates programs. Unsurprisingly, this approach leads to slow programs and is thus infeasible for deployment of real-world applications. [16, 24]
- **Large memory requirements** This is a large contributing factor towards the poor performance. The generated code is typically huge, due to a large runtime library, which is undesirable in the web world where developers prefer lightweight libraries for faster network delivery. Also, load time plays a crucial role in user experience and code generated by these compilers often incur a noticeable delay.
- **Non-human-readable generated code** Although this may seem like a minor issue, in practice when things fail, non-readable compiler output significantly affects the cost and ease of debugging. Also, non-human-readable code renders the ecosystem for dealing with JavaScript nearly useless. With a plethora of tools available for JavaScript, one should ideally be able to leverage their benefits.

Whalesong [24], a previous Racket-to-JavaScript compiler designed for educational purposes, suffers from all the above disadvantages. The size of the generated code is frequently cited on the Racket mailing list as the reason why it is not practical. In addition, on several occasions, we tried to investigate performance bottlenecks using both in-browser profilers and Vincent St-Amour’s Optimization Coach [20] for SpiderMonkey [19], but neither produced useful results due to the obfuscated compiler output.

We also investigated another approach, spending one month to port the Pycket [2, 23] Racket runtime to a JavaScript backend using Emscripten [25] and pypyjs [18]. Again, the result suffered from the described disadvantages. The runtime alone produced an 80mb JavaScript, which is unacceptable in practice.

---

2 https://groups.google.com/d/msg/racket-users/Kkff1CVyZSM/BtlZAbxTCAAJ
3 https://groups.google.com/d/msg/racket-users/FwqeqVp2KE/NEicJYmKQAJ
3.2 Non-semantics preserving compilation

This approach more closely resembles a one-to-one syntactic translation from the source language to JavaScript, a process sometimes called transpiling. Primitive forms in the source typically compile directly to their corresponding JavaScript primitives, without any major global transformation such as continuation-passing style. As a result, the semantics of the source language are retained only on a "best effort" basis. With this tradeoff, however, compiler writers gain more control over the performance and code size of the JavaScript output. This approach also produces human-readable JavaScript, and thus interoperation with JavaScript tools and libraries is much easier. Most JavaScript compilers used in industry seem to follow this approach. ClojureScript [8] and BuckleScript [26] are some of the more widely known examples.

Our compiler follows this approach, favoring performance and smaller code size over Racket semantics.

4 Compiler Implementation

The RacketScript compiler adopts a non-semantics-preserving strategy. Figure 1 shows the phases of the compiler. As the first phase, RacketScript leverages the Racket expander to fully expand a program. Thus RacketScript programs may use all the features of Racket’s macro system, though they will require pre-compilation before they can run in a JavaScript environment.

Unlike previous JavaScript compilers [24], RacketScript does not compile programs to bytecode because Racket bytecode is fairly low-level with no obvious one-to-one resemblance to JavaScript. For example, instead of binding values with names, Racket VM pushes them to a stack and refers to them by relative index. Thus, it is difficult to produce “native” and readable JavaScript from bytecode.

At the other end of the compilation pipeline, the compiler emits ES6 code, leveraging its new features to generate more compact code. Utilizing ES6 does not pose compatibility issues since several ES6 features such as classes, modules, default argument values and rest arguments are merely syntactic sugar over ES5. We take advantage of this relation in the last step, using Google’s Traceur 4 or Babel 5 to convert RacketScript’s output to an older JavaScript standard to improve compatibility.

In between macro expansion and JavaScript generation, RacketScript compilation utilizes two tree-based intermediate representations. The first, called AST, has a one-to-one correspondence with a fully-expanded Racket program. The second, an intermediate language called IL, approximates JavaScript’s abstract syntax. These two representations help to bridge the largely expression-based Racket to a statement-based JavaScript. The rest of the section summarizes the compilation phases. Section 5 through section 7 explain these steps in more detail.

4 https://github.com/google/traceur-compiler
5 https://babeljs.io/
Figure 1: Compiler phases in RacketScript

1. **Expansion to basic forms** This phase first parses Racket source code and expands it to one that only contains core forms using Racket’s `expand` function [12].

2. **Freshening** This phase gives a fresh new name to each local binding. As discussed in section 2, scope rules are different for Racket and JavaScript. Thus, without this phase, `let` bindings would not behave properly.

3. **Conversion to AST** This phase takes in the fully-expanded program produced by the last phase and translates it to AST structures. We convert to AST rather than use the existing syntax objects to simplify translation of some forms, such as `case-lambda`, that do not have an immediate counterpart in JavaScript.

4. **Conversion to IL** The AST structures are then transformed to IL. This intermediate language tries to bridge Racket and JavaScript and simplifies generation of JavaScript code. This phase also performs small optimizations. For example, we insert JavaScript binary operators when possible instead of their corresponding lambdas.

5. **(Best Effort) Proper tail-call implementation** This phase tries to translate tail calls to JavaScript loops when possible. Since, JavaScript does not implement tail calls properly, this phase is important to achieve some degree of stack-space saving. Note that RacketScript makes no guarantees and is not safe for space with respect to tail calls [7]. It behaves as expected, however, for the common scenarios such as self-recursion and thus this deficiency does not seem to be detrimental in practice.

   To enable proper tail-call implementation, IL first goes through `Return Lifting`, which merges return statements with previous statements when possible.

---

6 [http://docs.racket-lang.org/reference/syntax-model.html#%28part._fully-expanded%29](http://docs.racket-lang.org/reference/syntax-model.html#%28part._fully-expanded%29)
6. **Assemble to JavaScript** Each IL form is transformed to JavaScript ES6 forms. Each Racket module produces one JavaScript file. The compiled output is linked with the runtime written in JavaScript ES6. Finally, we use the Gulp build system and Google’s Traceur \(^7\) to transform ES6 to ES5. The resulting code thus executes even in old web browsers or NodeJS environments.

5 **Freshening**

Unlike Racket, JavaScript combines lexical scoping with function scope. Hence, the lifetime of a variable is not within its enclosing block, but the enclosing function. For example, in this JavaScript code fragment below, the inner \(x\) shadows the outer \(x\) for the entire function body instead of just in the inner block.

```javascript
(function () {
    var x = "outer";
    {
        var x = "inner";
        console.log(x);  // => "inner"
    }
    console.log(x);  // => "inner"
}());
```

Since, an exact one-to-one translation to JavaScript would fail to produce the expected output, each Racket identifier is given a fresh name. For example, the following code:

```racket
((lambda ()
    (let ([x "outer"])
        (let ([x "inner"])
            (displayln x)) ;;; => "inner"
            (displayln x)))) ;;; => "outer"
)();
```

compiles to:

```javascript
(function () {
    var x1 = "outer";
    var x2 = "inner";
    console.log(x1);  // => "outer"
    console.log(x2);  // => "inner"
}());
```

where both \(xs\) are given fresh new names \(x1\) and \(x2\).

ES6’s `let` statements could also be used to create lexically scoped bindings. However, `let` bindings seem to be slower than older `var` bindings \(^8\) \(^9\) so RacketScript does not use `let`.

\(^7\) Other compilers such as BabelJS would also suffice.
\(^8\) https://esdiscuss.org/topic/performance-concern-with-let-const
\(^9\) https://jsperf.com/let-vs-var-performance/50
6 Intermediate Languages: From AST to IL

The AST language, shown in figure 2, roughly corresponds to Racket’s Fully Expanded grammar. The grammar indicates modules-specific forms, which are the subject of section 7. In contrast to AST, IL is almost identical to JavaScript’s abstract syntax. The grammar for IL is described in figure 3. AST expressions are translated to IL statements and expressions. The translation function recursively traverses the syntax tree, and returns a list of IL statements and an IL expression which represents the value of that AST expression. The list of statements returned also contains the list of statements produced by any nested expression.

Compiling Racket modules to JavaScript presents another area of difficulty. Section 7 explains modules, imports, and exports in more detail but briefly, JavaScript has less-expressive import/export forms and thus the RacketScript compiler must convert Racket requires and provides to lower-level lists of individual identifiers, particularly for exports where the RacketScript compiler must manually convert all-defined and prefix-all-defined forms, renaming if necessary, by tracking the define-values forms in a module. The macro expansion compiler pass creates another problem, since Racket macros may use unexported identifiers in scope at the definition site, so the RacketScript compiler must explicitly ensure that these identifiers are exported and available at the macro use site.

6.1 Translation of Values

Simple literal values such as strings, numbers and booleans map directly to their corresponding JavaScript values. RacketScript numbers are mapped directly to JavaScript numbers and thus behave differently from Racket numbers. We use JavaScript’s typed array named Uint8Array which is an array of 8-bit unsigned integers to represent byte arrays. Racket’s void is mapped to JavaScript null primitive. Regular expressions are currently unsupported.

Lists, vectors, hashes, bytes, symbols and keywords are implemented as ES6 classes and constructors provided by RacketScript’s runtime library (described in section 8). Bindings introduced by let-values, letrec-values and define-values are translated to var statements. set! is translated to JavaScript assignment. Lastly, multiple values are represented by another ES6 class which wraps all items.

6.2 Translation of forms

Binding multiple values Racket functions can return multiple values. let-values, letrec-values and define-values all take a sequence of bindings that should match the number of values returned by the body.

\[
\text{let-values } \{ ((a \ b \ c) \ (values \ 1 \ 2 \ 3)) \} \\
\text{displayln } \{ \text{list } a \ b \ c \} \\
\]
\[
\langle ModuleForm \rangle \ := \ (\text{Module} \ \langle id \rangle \ \langle path \rangle \ (\text{setof} \ \langle ModulePath \rangle) \\
\langle ModuleLevelForm \rangle \ldots) \\
\langle ModuleLevelForm \rangle \ := \ (\langle GeneralTopLevelForm \rangle) \\
\text{Provide} \\
\langle ModuleForm \rangle \\
\langle GeneralTopLevelForm \rangle \ := \ (\langle Expr \rangle) \\
\text{DefineValues} \ (\langle id \rangle \ldots) \ (\langle Expr \rangle) \\
\text{JSRequire} \ (\langle id \rangle \langle ModulePath \rangle) \\
\langle Expr \rangle \ := \ \text{Ident} \\
\text{TopId} \ (\langle id \rangle) \\
\text{Quote} \ \langle any \rangle) \\
\text{Begin} \ (\langle expr \rangle \ldots^+) \\
\text{Begin0} \ (\langle Expr \rangle) \ (\langle Expr \rangle \ldots) \\
\text{PlainApp} \ (\langle Expr \rangle) \ (\langle Expr \rangle \ldots) \\
\text{PlainLambda} \ (\langle Forms \rangle) \ (\langle Expr \rangle \ldots^+) \\
\text{CaseLambda} \ (\langle PlainLambda \rangle \ldots^+) \\
\text{If} \ (\langle Expr \rangle) \ (\langle Expr \rangle) \ (\langle Expr \rangle) \\
\text{LetValues} \ (\{\langle \langle id \rangle \ldots \} \cdot \langle Expr \rangle \ldots) \ (\langle Expr \rangle \ldots^+) \\
\text{Set} \ (\langle id \rangle) \ (\langle Expr \rangle) \\
\text{WithContinuationMark} \ (\langle Expr \rangle) \ (\langle Expr \rangle) \ (\langle Expr \rangle) \\
\langle Ident \rangle \ := \ (\langle LocalIdent \ (\langle id \rangle) \rangle) \\
\text{TopLevelIdent} \ (\langle id \rangle) \\
\langle Formals \rangle \ := \ (\langle id \rangle) \\
\text{TopId} \ (\langle id \rangle) \\
\text{TopId} \ (\langle id \rangle) \\
\langle Provide \rangle \ := \ (\langle SimpleProvide \ (\langle id \rangle) \rangle) \\
\text{RenamedProvide} \ (\langle id \rangle) \ (\langle id \rangle) \\
\text{AllDefined} \ (\text{setof} \ (\langle id \rangle) \rangle) \\
\text{PrefixAllDefined} \ (\text{setof} \ (\langle id \rangle) \rangle) \\
\langle ModulePath \rangle \ := \ (\langle id \rangle) \\
\text{Path} \\
\langle id \rangle \ := \ \text{Any valid Racket identifier.}
\]

Figure 2: "The AST language"
\(\langle\text{ModuleForm}\rangle \equiv (\langle\text{Module}\rangle \langle\text{ModulePath}\rangle \langle\text{Provide}\rangle \langle\text{Require}\rangle \langle\text{Statement}\rangle \ldots+)\)

\(\langle\text{Require}\rangle \equiv (\langle\text{RequireDirect}\rangle \langle\text{id}\rangle)\)
| (\langle\text{RequireDefault}\rangle \langle\text{id}\rangle) |
| (\langle\text{RequireAll}\rangle \langle\text{id}\rangle) |

\(\langle\text{Provide}\rangle \equiv (\langle\text{SimpleProvide}\rangle \langle\text{id}\rangle)\)
| (\langle\text{RenamedProvide}\rangle \langle\text{id}\rangle \langle\text{id}\rangle) |

\(\langle\text{Expr}\rangle \equiv (\langle\text{Lambda}\rangle \langle\text{id}\rangle \langle\text{Statement}\rangle \ldots)\)
| (\langle\text{Application}\rangle \langle\text{Expr}\rangle \langle\text{Expr}\rangle \ldots) |
| (\langle\text{Array}\rangle \langle\text{Expr}\rangle \langle\text{Expr}\rangle \ldots) |
| (\langle\text{Object}\rangle (\langle\text{ObjectKey}\rangle \langle\text{Expr}\rangle \ldots)) |
| (\langle\text{Ref}\rangle \langle\text{Expr}\rangle \langle\text{id}\rangle) |
| (\langle\text{Index}\rangle \langle\text{Expr}\rangle \langle\text{Expr}\rangle) |
| (\langle\text{New}\rangle \langle\text{LValue}\rangle) |
| (\langle\text{New}\rangle (\langle\text{Application}\rangle \langle\text{Expr}\rangle \langle\text{Expr}\rangle \ldots)) |
| (\langle\text{InstanceOf}\rangle \langle\text{Expr}\rangle) |
| (\langle\text{Operator}\rangle \langle\text{Ops}\rangle \langle\text{Expr}\rangle \ldots+) |
| (\langle\text{Value}\rangle \langle\text{Datum}\rangle) |
| (\langle\text{id}\rangle) |

\(\langle\text{Statement}\rangle \equiv (\langle\text{Expr}\rangle)
| (\langle\text{VarDec}\rangle \langle\text{id}\rangle \langle\text{option}\rangle \langle\text{Expr}\rangle)) |
| (\langle\text{LetDec}\rangle \langle\text{id}\rangle \langle\text{option}\rangle \langle\text{Expr}\rangle)) |
| (\langle\text{Assign}\rangle \langle\text{LValue}\rangle \langle\text{Expr}\rangle) |
| (\langle\text{If}\rangle \langle\text{Expr}\rangle \langle\text{Statement}\rangle \ldots \langle\text{Statement}\rangle \ldots) |
| (\langle\text{While}\rangle \langle\text{Expr}\rangle \langle\text{Statement}\rangle \ldots) |
| (\langle\text{Return}\rangle \langle\text{Expr}\rangle) |
| \langle\text{Label}\rangle \langle\text{id}\rangle |
| \langle\text{Continue}\rangle \langle\text{id}\rangle |
| \langle\text{ExnHandler}\rangle \langle\text{Statement}\rangle \langle\text{id}\rangle \langle\text{Statement}\rangle \ldots \langle\text{Statement}\rangle \ldots) |
| (\langle\text{Throw}\rangle \langle\text{Expr}\rangle) |

\(\langle\text{ObjectKey}\rangle \equiv (\langle\text{id}\rangle) \langle\text{string}\rangle)

\(\langle\text{LValue}\rangle \equiv (\langle\text{id}\rangle) \langle\text{Ref}\rangle \langle\text{Expr}\rangle \langle\text{id}\rangle) \langle\text{Index}\rangle \langle\text{Expr}\rangle \langle\text{Expr}\rangle)

\(\langle\text{Ops}\rangle \equiv \text{All JavaScript arithmatic, comparison and logical operators.}

\(\langle\text{ModulePath}\rangle \equiv (\langle\text{id}\rangle)
| \langle\text{path}\rangle |

\(\langle\text{id}\rangle \equiv \text{Any valid JavaScript identifier.}

---

Figure 3: "Intermediate Representation"
<table>
<thead>
<tr>
<th>Racket</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(())</td>
<td>[]</td>
</tr>
<tr>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>42.2</td>
<td>42.2</td>
</tr>
<tr>
<td>#t</td>
<td>true</td>
</tr>
<tr>
<td>#f</td>
<td>false</td>
</tr>
<tr>
<td>#\a</td>
<td>Core.Char.make(&quot;a&quot;)</td>
</tr>
<tr>
<td>&quot;hello world&quot;</td>
<td>&quot;hello world&quot;</td>
</tr>
<tr>
<td>(void)</td>
<td>null or undefined</td>
</tr>
<tr>
<td>'(1 2)</td>
<td>Core.Pair.make(1, 2)</td>
</tr>
<tr>
<td>(box v)</td>
<td>Core.Box.make(v)</td>
</tr>
<tr>
<td>(bytes 1 2 3)</td>
<td>new UInt8Array([1, 2, 3])</td>
</tr>
<tr>
<td>'symbol</td>
<td>Core.Symbol.make(&quot;symbol&quot;)</td>
</tr>
<tr>
<td>#:keyword</td>
<td>Core.Keyword.make(&quot;keyword&quot;)</td>
</tr>
</tbody>
</table>

Figure 4: Translation of basic data types.

JavaScript however, does not support multiple return values. Thus, multiple return values are stored in an object bound to a temporary variable, and then extracted and bound to individual variables. The following is the conversion of the above Racket code:

```javascript
var let_result1 = values(1, 2, 3);
var a1 = let_result1.getAt(0);
var b2 = let_result1.getAt(1);
var c3 = let_result1.getAt(2);
M0.displayln(list(a1, b2, c3));
```

**Lambda** AST and IL lambdas differ in two ways. First, all IL lambdas are basically plain lambdas, i.e. there are no case-lambda’s in IL. Secondly, IL lambdas can either take a fixed number of arguments or any number of arguments, bound to the `arguments` variable. Hence, a RacketScript `(lambda (a b . c) c)` must be converted to an IL lambda where each argument is retrieved from the `arguments` binding. The following is the JavaScript translation of the lambda example. The `a` and `b` arguments are directly retrieved, while `c` is obtained by slicing `arguments` from the third position and converting to a list:
Case Lambda `case-lambda` is translated to an IL lambda which consumes a variable number of arguments. The body then dispatches to different code depending on the length of `arguments`:

```
(case-lambda
  [(a b) (+ a b)]
  [(a b c) (* a b c)]
  [(a b c . d)
    (+ a b c (foldl + 0 d))])
```

Translation of the `case-lambda` example above is shown in figure 5. Each `case-lambda` clause is compiled independently like a regular lambda, and then wrapped inside conditionals to drive the dispatch logic. A closer inspection, however reveals some improvements. Specifically, for the first two clauses, we can replace the use of `apply` with direct function application. This simple change also removes the need to convert `arguments` to a Racket list at the entry of the dispatch function. Additionally, we can lift each clause outside the dispatch function to avoid their re-evaluation, and make the generated code faster and arguably more readable.

Conditionals `if` is an expression in AST, but it is a statement in IL. Therefore, the result of this translation produces a list of statements and the name of the variable which binds the value of result.

```
(define result
  (if (let ([x 0])
        (positive? x))
    "yes"
    "no"))
```

Function application Each argument expression is translated individually and is assigned a fresh variable binding. There is another possible translation, that applies the value expression produced by the translation of arguments directly. The following is an example showing both translation strategies:
(function() {
    var args1 = $rjs_core.Pair.listFromArray(
        $rjs_core.argumentsToArray(arguments));

    if (M0.equal_p(M0.length(args1), 2)) {
        return M0.apply(function(a1, b2) {
            return a1 + b2;
        }, args1);
    } else {
        if (M0.equal_p(M0.length(args1), 3)) {
            return M0.apply(function(a3, b4, c5) {
                return a3 * b4 * c5;
            }, args1);
        } else {
            if (M0._gt__eq_(M0.length(args1), 1)) {
                return M0.apply(function() {
                    var a6 = arguments[0];
                    var b7 = arguments[1];
                    var c8 = arguments[2];
                    var d9 = $rjs_core.Pair.listFromArray(
                        $rjs_core.argumentsSlice(
                            $rjs_core.argumentsToArray(arguments), 3));
                    return a6 + b7 + c8 + M0.foldl(M0._plus_, 0, d9);
                }, args1);
            } else {
                return M0.error("case-lambda: invalid case");
            }
        }
    }
});

Figure 5: Translation of case-lambda
The problem with the second approach is due to side-effects, specifically, if \( \text{fnb} \) has side-effects then the translation could be wrong because it changes the order of execution. The first translation, evaluates the first argument completely before evaluating the second one, following JavaScript’s specification. The second translation partially evaluates both arguments by calling \( \text{fnb} \), and then call \( \text{fnc} \) on the partial results.

7 Modules

Modules help organize code, and both Racket and JavaScript support modular programming, though to a different degree. This section briefly summarizes the module system of both languages and then introduces the RacketScript implementation. Strictly speaking, Racket has a more expressive module system than JavaScript due to features such as submodules and phases. Hence, a one-to-one transformation is not possible.

7.1 Modules in Racket

Racket’s rich module system enables an effective form of code organization. A Racket module typically resides in its own file; however, programmers may also declare several submodules inside a Racket source file using the \( \text{module} \) form. In addition, a programmer can extend an existing module using \( \text{module}+ \).

Module names inherit the directory structure of its containing file. To find a module, Racket looks at relative paths or at \textit{collection paths}.

Bindings are exported and imported using \( \text{%provide} \) and \( \text{%require} \) forms. Using various sub-forms for \( \text{%provide} \) and \( \text{%require} \), programmers can specify the set of bindings that they want to export or import. For example, \( \text{all-defined} \) exports all bindings in a module, \( \text{all-from} \) exports all bindings imported from specific module, and \( \text{all-defined-except} \) and \( \text{all-expect} \) removes given bindings from the exported set. In addition, identifier names could be changed using \( \text{rename} \) or prefixed using \( \text{prefix-all-defined} \) and \( \text{prefix-all-defined-except} \). Finally, macro expansion happens at various phases and \( \text{for-meta} \) and \( \text{for-syntax} \) allows shifting of binding phase levels during import and export. Most of the above operations can be done when importing modules as well.

Figure 6 illustrate the usage of require and provide in Racket. Both \( \text{require} \) and \( \text{provide} \) are macros which expand to the core forms \( \text{%require} \) and \( \text{%provide} \), respectively. The example imports all bindings from “foobar.rkt”
and "posn.rkt", but imports only sin, cos and tan from "math.rkt". The pro-
provide form exports Origin from the current module, Bar from foobar.rkt, and
sin and cos from "math.rkt", all with prefix Foo:. In other words, the module
exports the identifiers Origin, Foo:Bar, Foo:sin, and Foo:cos:

#lang racket/base

(require "foobar.rkt" ;; imports Bar
 "posn.rkt" ;; imports posn
 (only-in "math.rkt" sin cos tan))

(provide Origin
 (prefix-out Foo:
   (prefix-out Foo:
     (combine-out
      (all-from-out "foobar.rkt")
      (except-out "math.rkt" tan)))))

(define Origin (posn 0 0))

Figure 6: Use of require and provide forms in Racket

7.2 Modules in JavaScript

Prior to ES6, JavaScript did not have linguistic support for modules and in-
stead relied on libraries to emulate them. These libraries use closures and
immediately-invoked functions to achieve module-like behavior. The follow-
ing example uses a function to create a namespace invisible to the outside
world. Bindings are exported by extending the object named exports passed
to this function:

(function (exports) {
    var add = (a, b) => a + b; // Module private
    var add1 = (v) => add(v, 1); // Exported

    exports.add1 = add1
})(global);

ES6 introduced a native module system to JavaScript. The ES6 module sys-
tem can be expressed as syntactic sugar on top of ES5, and hence be compiled
down to ES5. There is exactly one file per ES6 module, and there is no concept
of submodules or extending existing modules. When importing a particular
binding, the imported binding is introduced to the current namespace. Unlike
Racket, each binding must be specifically exported or imported, except for one
exception. Programmers may use the `*` import form, which creates one binding which is an object representing the complete module. Exported bindings can be accessed by object referencing.

```javascript
// Imports add1 and sub1 from module named "math",
// and introduces them to current namespace.
import {add1, sub1} from "math";
console.log(add1(1));
console.log(sub1(1));

// Importing complete module. In this case
// bindings have to accessed through the module
// object
import * as M from "math";
console.log(M.add1(1));
console.log(M.sub1(1));
```

### 7.3 Modules in RacketScript

RacketScript compiles a subset of Racket module features to ES6 modules. This transformation is not one-to-one, because ES6 modules are less expressive than Racket modules. Since we compile fully expanded programs, we consider runtime bindings and ignore all other phases.

**Exports** The RacketScript compiler computes the set of bindings that is exported from a module and expands that to individual ES6 exports. When expanding, the compiler tracks all `#%provide` and translates them to corresponding ES6 exports. Re-exports using `all-from` are ignored to keep the size of generated code small but this does not affect runtime behaviour (see the next section). Macro expansion can also introduce bindings that are not exported by defining module, or imported at the user module. Such bindings, which are in scope at a macro definition site but not at the use site, are typically not provided but RacketScript must compute them and add them to the set of exports. Thus, RacketScript compiles modules in topological order, storing all the identifiers and their sources used during expansion phase. Therefore, by the time the compiler starts to compile a module, it knows all the bindings that need to be exported.

**Imports** There are two ways we can compile `require`:

1. Parse each require spec and statically resolve all imported identifiers in a module. This means that we have to import each identifier individually.
2. Using the `identifier-binding` function, figure out where the binding was originally defined. Then import that module as an object, utilizing the `*` import form, and then refer to each binding via that module object.
RacketScript uses the second approach, because it has two advantages over the first one:

- **Smaller generated code**: The first approach will add import statements for each binding regardless of whether it is used or not. The second approach does not import individual bindings, resulting in less code. Similarly, since we refer to the module where the binding is originally defined, we can ignore re-exports.

- **Avoid name collisions**: As mentioned earlier, macros may use bindings that are in scope at the macro definition site but not at the use site. RacketScript must explicitly import these bindings at the use site, creating potential conflicts if the another binding with the same name is already in scope. Referring through module object automatically solves the problem.

### 7.4 Non-supported module features

RacketScript currently do no support submodules. Specifically, RacketScript does not allow `module` or `module*` as top-level module form. Additionally, RacketScript does not support `dynamic-require` and `lazy-require` [11, 12]. Among subforms of `#%require` and `#%provide`, RacketScript does not implement the `struct` and `submod`.

### 8 Runtime Library

While some basic Racket data types are directly mapped to JavaScript, some kinds of data in Racket do not have an immediate counterpart in JavaScript. Examples are numbers, symbols, keywords, pairs, vectors, and hashes. Thus, RacketScript relies on a JavaScript runtime library containing additional functions and classes to help with compilation of these kinds of data. While this runtime library increases the size of compiled programs, it is essential in order to support key features of Racket such as hashes and structs. Nonetheless, we strive to limit its size whenever possible and thus it remains reasonable (approx. 120KB\(^{10}\)) especially in comparison to compiled Whalesong programs (approx. 580KB).

For reference, the primitive data types in JavaScript are strings, booleans, floating point numbers, objects (arrays are also objects in JavaScript), null, and undefined. Racket, on the other hand, supports strings, booleans, a large numeric tower that ranges from fixnums to complex numbers, characters, byte strings, symbols, keywords, pairs, hashes, vectors, boxes, and void.

RacketScript utilizes ES6 classes to implement Racket data that does not have an immediate JavaScript counterpart. Classes for these kinds of data implement methods that returns their string representation, checks their type,

---

\(^{10}\) We calculate code size after compiling our ES6 runtime to ES5 using Babel. Original code size is approximately 85KB.

\(^{11}\) `#%kernel` library is not included.
checks for equality, and computes hashes. Thus, RacketScript programmers may use widely used Racket operations such as equal?, eqv?, eq? and display.

Several data types are directly wrapped in a simple class, without any processing before or after construction:

- **Pairs** are implemented as a class which keeps references to the head and tail components. We could have used a JavaScript array instead, but using a class makes type checking easier since JavaScript's `instanceof` operator can be used directly.
- **Boxes** are implemented as a class with a field that points to its value.
- **Symbols** and **keywords** are again implemented as classes wrapping the actual value. They are interned and thus all three Racket equality functions (eq?, eqv? and equal?) behave as they would in Racket.
- JavaScript functions can not return multiple values. Instead multiple values are wrapped in a `Value` class provided by runtime.

Unlike the above classes, the implementation of **hashes** require more than just trivial wrapper classes. JavaScript provides two kinds of key/value stores, viz. objects and maps. However, they can not be used as Racket hashes directly because Racket hashes support a wide variety of keys and key comparisons. Objects can not be used, because their keys can only be strings or symbols. ES6 introduced the `Map` class whose keys can be any JavaScript primitive, including functions and objects, but in this case the problem is key equality. Racket hashes uses three kinds of equality including value based equality, while on the other hand `Map` just supports reference based equality. Lastly, both objects and maps are mutable data structures, which means it will be inefficient to use them directly. RacketScript uses a hash array mapped trie (HAMT) [1] implementation. This increases the size of our runtime but faithfully implements immutable Racket hashes with custom key equality and hash functions. Specifically, an immutable hash simply updates the internal reference with the updated HAMT object.

Our runtime provides two different **vector** implementations, depending on whether they are mutable or immutable. For mutable vectors, we implement a class that keeps a JavaScript array internally. Accessing and mutating vector elements corresponds to JavaScript array indexing and assignment, respectively. Immutable vectors reuse the `hasheq` implementation where keys are all integers.

RacketScript mostly supports Racket **structures**. Specifically, it supports super types, guards, auto fields, mutable fields and structure properties, though the runtime does not currently include any built-in structure properties. Structures can be opaque or transparent but do not currently respect inspectors. The implementation of structs is divided into three parts. The first is struct types, i.e., `struct-type-descriptor`, a first class value that records all type information for a struct. This includes, the fields, structure properties, guards, super struct

12 https://github.com/mattbierner/hamt
types, etc. The second part is the `Struct` class which constructs the actual struct object of a given struct type. Lastly, we implement a class to represent structure type properties, which can be attached to any structure types. We do not currently support the `proc-spec` (same as `prop:procedure` property) parameter and thus structure objects cannot be used as functions. We plan to add such support in the future, because this feature can be easily implemented if we add a function field called `call` to the structure class, and replace all function application of form `func(e0, e1, ...)` with `func.call(this, e0, e1 ...)`. Since functions are objects in JavaScript, and already have a `call` method in their prototype, everything else may remain the same.

9 Properly implementing (some) tail-calls

As discussed earlier, loops are expressed in terms of recursion in Racket. Thus the improper implementation of tail-calls in JavaScript makes it impractical to compile even basic Racket programs to JavaScript, as one may easily run out of stack space. For special case of self-recursive functions, we can simply convert them to a JavaScript's primitive loop. This compromise seems to work reasonably well for the programs we have implemented (see section 12).

Consider a function that computes the sum of \( n \) numbers. Here the function `sum` calls itself at tail position.

\[
\begin{align*}
(\text{define} & \ (\text{sum} \ n \ a) \\
& (\text{if} \ (\text{zero?} \ n) \ a \\
& \quad (\text{sum} \ (\text{sub1} \ n) \ (+ \ a \ n)))
\end{align*}
\]

A direct translation to JavaScript might produce the following output:

```javascript
var sum = function(n1, a2) {
    if (M0.zero_p(n1)) {
        var if_res1 = a2;
    } else {
        var if_res1 = sum(M0.sub1(n1), a2 + n1);
    }
    return if_res1;
};
```

Since the self-call is at tail position, the result is returned to the original caller. By observation we can see that if we (1) update the input arguments, i.e., \( n1 \) and \( a2 \), with the arguments we are passing to self-tail-call (2) restart from the beginning of body the function, we can get same the execution, but without consuming any stack space. Following these steps, RacketScript guarantees the following code:
var sum = function(_n13 , _a24 ) {
  lambda_start2 : while ( true ) {
    let n1 = _n13 ;
    let a2 = _a24 ;
    if (M0. zero_p(n1 )) {
      return a2 ;
    } else {
      _n13 = M0. sub1(n1 );
      _a24 = a2 + n1 ;
      continue lambda_start2 ;
    }
  }
};

The second step, i.e., restarting, is implemented by wrapping the body in a loop, and then executing continue statement after updating the arguments.

In order to preserve the expected behavior of recursive calls, our RacketScript compiler must perform an additional transformation called Return Lifting prior to loop translation. To explain return lifting, we first define previous executable statements, i.e., a set of statements whose successor statement is always the current statement. Hence, if just before the current statement, we had an if-else block, the previous executable statements would be the last statements of the blocks. Return lifting, replaces one of the previous executable statement with return when:

1. the expression returned is simply an identifier, and
2. the identifier was initialized or reassigned in previous executable statement.

Figure 7 is an example demonstrating return lifting. On the left, we see that var res = ... are the previous executable statements of return res, and they satisfy the conditions mentioned above. RacketScript often produces code like that on the left, e.g., a conditional branch makes a self-call, only to return the result in next statement i.e. var res = .... Clearly, compilation has moved self-calls from tail position to non-tail position\(^{13}\), and thus the code would not trigger our loop translation. Return lifting moves these self-calls back to tail positions, as seen on the right, enabling the proper loop translations.

Additionally, observe that in our generated loops, we use let statements instead of var statements to update the arguments. Since the var binding has a lifetime of the entire function, even though these statement are inside a loop, all variables are same instances, resulting in all closures produced in this loop pointing to the last final value of n1 and a2. Since let’s lifetime spans only the local block, i.e., lexical scope, each closure produced will point to different instances of these bindings. Examples in figure 8 illustrates this problem.

\(^{13}\) This step also corrects programmer’s oversight.
function sum (n, acc) {
  if (n === 0) {
    var res = acc;
  } else {
    var res = sum(n -1 , n+ acc);
  }
  return res;
}

function sum (n, acc) {
  if (n === 0) {
    return acc;
  } else {
    return sum(n -1 , n+ acc);
  }
}

Figure 7: Return Lifting

// Use var statement
function useVar () {
  var result = [];
  for (var i = 0; i < 5; ++i) {
    result.push(function() {
      console.log(i);
    });
  }
  return result;
}

for (let g of useLet()) {
g();
}

// Output:
// => 0, 1, 2, 3, 4
Figure 8: Use of var and let bindings inside loop.
10 Continuation Marks

Racket’s continuation marks [5] allow programmers to annotate continuation frames with a set of key/value pairs and later retrieve them. This simple feature can then be used to implement other higher-level language feature such as exceptions, parameters, and a stepping debugger [5]. Concretely, a language with continuation marks requires two primitive operations:

- `(with-continuation-mark key-expr values-expr result-expr)` This operator installs a mark on the most recent frame of the call-stack. This operator is abbreviated as `w-c-m`.
- `(current-continuation-marks)` Returns marks installed on each frame, from innermost to outermost, of the current call-stack. This operator is abbreviated as `c-c-m`. In Racket, this operator returns an opaque value, which is used with primitive functions such as `continuation-mark->list` to extract values associated with particular keys. For consistency, we follow the Racket model.

JavaScript does not allow access to stack frames and therefore RacketScript cannot compile continuation marks directly. Instead, RacketScript’s runtime library emulates continuation marks. This suffices to support some common Racket features such as parameters and exceptions, though we are still working on adding some common Racket parameters to RacketScript’s runtime library. We use exceptions and a separate stack to emulate continuation marks in JavaScript:

- A global **mark stack** stores continuation marks. Each element of the stack, called a frame, stores a key/value pair in a `hasheq`. A `w-c-m` function adds or updates stack frames that are visible only during evaluation of its body, and then restores the stack to its prior state.
- **Exception handling** is used to ensure that the mark stack is restored to its original state after executing body. Raising exception breaks the normal flow of execution and jumps to the exception handler. Therefore, if an exception is raised by the body of `w-c-m` and is un-handled at the site of `w-c-m`, code that restores mark stack does not get executed. We therefore handle any exceptions raised by the body of `w-c-m` so that mark stack is always restored.

---

14 [http://docs.racket-lang.org/reference/parameters.html](http://docs.racket-lang.org/reference/parameters.html)
15 Clements et al. [6] modified Rhino JavaScript engine to implement continuation marks. However, RacketScript targets JavaScript engines deployed to end-users, and therefore cannot follow the same approach.
16 JavaScript supports exceptions natively, but Racket’s exception handling is implemented using continuation marks and prompts. Specifically, explicit Racket exception handlers are expanded away and thus have no immediate translation to JavaScript.
The following example, maps a key to a value as a continuation mark, and later retrieve all marks attached to that key:

```
(define (main)
  (with-continuation-mark 'key 'value
    (continuation-mark-set->list
      (current-continuation-marks)
      'key)))

;; => (list 'value)
```

Since the value returned by `current-continuation-marks` is an opaque value, `continuation-mark-set->list` is used to extract the marks attached with any given key.

RackScript compiles the above example as follows. The `try` block (1) saves the old frames retrieved using `Marks.getFrames()`, (2) creates a new frame in stack using `Marks.enterFrame()`, (3) attaches the key to the value as a continuation mark in the latest frame using `Marks.setMark()`, and lastly (4) executes the body. The `finally` block simply restores the old stack:

```javascript
var main = function () {
  var _old_frames = $rjs_core.Marks.getFrames();
  var _new_frames;
  try {
    _new_frames = $rjs_core.Marks.enterFrame();
    $rjs_core.Marks.setMark($rjs_core.Symbol.make("key"),
    $rjs_core.Symbol.make("value"));
    var _wcm_result = M0.continuation_mark_set__gt_list(
      M0.current_continuation_marks(),
      $rjs_core.Symbol.make("key"));
  } finally {
    $rjs_core.Marks.updateFrame(_old_frames, _new_frames);
  }
  return _wcm_result;
};
```

Clements [5] briefly mentions this approach, describing it as infeasible since it breaks tail-calls. Since RackScript does not provide proper tail-call implementation, we consider the approach acceptable under these circumstances. This difference in tail call handling, however, may lead to differing continuation mark behavior because the stack at a given program state may contain different frames.

If a `w-c-m` is nested inside the body of another `w-c-m`, and the inner `w-c-m` is at tail position of the body of outer `w-c-m`, there is no need to create a new frame, as the new mark should overwrite the previous one. The RackScript compiler calculates the position of nested `w-c-m`'s with respect to the outer `w-c-m` within the enclosing lambda, and creates a frame only if necessary:
In the example above, the inner w-c-m is at the tail position of outer w-c-m. RacketScript detects this when compiling and does not create another frame again. Therefore, it uses the previous frame created by the first w-c-m, and produces \( \text{list 'inner-value} \) as result. However, as shown in figure 9, if the inner w-c-m were not at tail position, a new frame would be created, and result would be \( \text{list 'inner-value 'outer-value} \).

\[
\text{(with-continuation-mark 'key 'outer-value}
\text{(if #t}
\text{(if #t}
\text{(with-continuation-mark 'key 'inner-value}
\text{(continuation-mark-set->list}
\text{(current-continuation-marks)}
'key))
#f)
#f))

;;=> (list 'inner-value)
\]

Figure 9: Continuation mark at non-tail position.

As mentioned before, RacketScript does not behave like Racket for tail-calls. For example, the example in figure 10 produces \( \text{list 1} \) when executed in Racket, while \( \text{list 1 2 3 4 5} \) when executed under RacketScript. Since JavaScript does not have proper tail-calls, our approach captures the right context of execution.

10.1 Application: Dynamic Binding parameterize

Parameters [11, 12] introduces a form of dynamic binding in Racket, controlled using the parameterize form. Like let, parameterize binds a parameter to a
(define (do-n n)       ;; => Outputs
  (if (zero? n)
    (continuation-mark-set->list
      (current-continuation-marks)
      'res)
    (with-continuation-mark 'res n
      (do-n (sub1 n)))))
  ;; => RacketScript
  ;; => (list '1 '2 '3 '4 '5)

(do-n 5)

Figure 10: Tail-call inside \textit{w-c-m}

particular value. Unlike \textit{let} the parameter maintains that value throughout the entire evaluation of the body expression, not just in the lexical context. It is only shadowed by other \textit{parameterize} forms. The code fragment in figure 11 demonstrates this behaviour. Observe how the “outer” in func-b does not always display the value in lexical scope (#f).

\textbf{Implementation} Racket implements parameters using continuation marks. Specifically, Racket keeps a table of all parameters and this table is attached via a unique key as continuation mark. Each \textit{parameterize} form expands to \texttt{with-continuation-mark} form which extends the current parameterization. The following is the expanded version of \texttt{(parameterize ([fruit "apple"])) (apple)):

\begin{verbatim}
(with-continuation-mark
  parameterization-key
  (extend-parameterization
    (continuation-mark-set-first '#f parameterization-key)
    fruit
    "apple")
  (displayln (fruit)))
\end{verbatim}

The \texttt{extend-parameterization} function returns a current parameterization with the given parameter updated with the new value. \texttt{continuation-mark-set-first} simply extracts the most recent continuation mark value attached with the given key. RacketScript uses immutable hash tables [1] to keep the parameter-value pairs, and hence \texttt{extend-parameterization} is a simple set operation.

The value returned by \texttt{make-parameter} is also a function. When applied to a single argument \texttt{e0}, it updates the current value of parameter to \texttt{e0}. When applied without argument, this function simply returns current value of parameter.

In RacketScript \texttt{make-parameter} is a primitive function, and its implementation, as shown in figure 12, can be broken down in two steps:
(define param (make-parameter #f))

(define (func-b)
  (displayln (list "funa-b:outer" (param)))
  (parameterize ([param "apples"])
    (displayln (list "func-b:inner" (param)))))

(define (func-a)
  (displayln (list "func-a: outer" (param)))
  (parameterize ([param "oranges"])
    (func-b))
  (displayln (list "func-a: finished" (param))))

(func-a)
;; Output
;;=> (func-a: outer #f)
;;=> (func-a: outer oranges)
;;=> (func-b:outer apples)
;;=> (func-a: finished #f)

(func-b)
;; Output
;;=> (funa-b:outer #f)
;;=> (func-b:inner apples)

Figure 11: Simple use of parameters.
function makeParameter(initValue) {
    let param = function (maybeSetVal) {
        // Get current value box from parameterization. The function
        // ‘param’ that we result is the key.
        let pv = getCurrentParameterization().get(param, false) ||
            __top.get(param, false);
        // Create entry in __top if its a mutation.
        if (!pv && maybeSetVal !== undefined) {
            pv = Box.make(initValue);
            __top.set(param, pv);
        }
        // Get/Set
        if (maybeSetVal === undefined) {
            return (pv && pv.get()) || initValue;
        } else {
            pv.set(maybeSetVal);
        }
    }
    return param;
}

Figure 12: Implementation of `make-parameter`

1. **Retrieve the value of the parameter**: Parameter values are kept in boxes. We
   first look into current parameterization stored as a continuation marks. If
   the parameter was never updated inside a `parameterize` form, the current
   parameterization will not hold its value box. Value box for such parameters
   are kept in a global top-level parameterization table. In the code fragment
   above, it is named `__top`.

2. **Mutate or Get value**: If the parameter function is applied to no arguments,
   simply return the value of box returned by last step, otherwise update the
   value of the same box.

   Figure 13 illustrates the steps discussed above. There are three parameters
   P1, P2 and P3 in the figure. Each row parameterizes one of these parameters
   and the arrows points to the boxes that store their value after parameterization
   is updated. Since we only create new box for parameters that are updated in
   `parameterize` form, any direct mutation will persist throughout its context.

10.2 Asynchronous Callbacks

JavaScript APIs and programs are mostly asynchronous. This raises the ques-

tion of how callbacks interact with continuation marks. If the runtime sus-
pends the main execution to execute another callback, our implementation of
Figure 13: Parameterization table linkage
continuation marks would sometimes produce the wrong values. Fortunately
JavaScript guarantees that execution cannot be suspended. Hence, callbacks
are executed only after the current execution finishes, guaranteeing that the
continuation mark stack is empty.

In some situations, a programmer may want a callback to inherit all the
parameters at the point when it is invoked. This behavior is similar to Racket,
where a new thread inherits the parameterization of its creator thread. The
parameter values are therefore stored in thread cells to avoid any race condi-
tions. It is non-trivial to achieve the same guarantees in JavaScript, because it
is impossible to know which functions will be called asynchronously.

Therefore RacketScript requires programmers to annotate these callbacks
with the parameters that should be available during execution of the callback.
This annotation is a function that wraps the callback with code to copy current
parameterization. We create new boxes for each parameter value to avoid race
conditions between callbacks.

11 Foreign Function Interface

RacketScript provides a Foreign Function Interface (FFI) that allows RacketScript
programs to:

- import JavaScript modules.
- create and update JavaScript objects/arrays.
- access elements of an array or object.
- convert RacketScript data types to JavaScript data types.

All these operations could be implemented as primitive functions in the
runtime, but that would often result in unnecessary function calls and conver-
sions from Racket data types to JavaScript data types. Instead, we make them
special forms so that the RacketScript compiler can identify them and use the
JavaScript syntax. For example, to create an array the former approach will use
a FFI function with signature $\text{List<Any> \rightarrow Array<Any>}$ to convert a Racket list
to JavaScript array, while the latter approach would directly use the JavaScript
syntax to create the same array.

11.1 Low-level API

All the aforementioned FFI operations are implemented by adding a special
case to the fully-expanded program syntax. We consider any function applica-
tion with $\#\%js-ffi$ as the first argument as an FFI operation. These forms must
match the grammar in figure 14:

Since we expand Racket programs to fully-expanded form, the result is
always a $\#\%plain-app$ with $\#\%js-ffi$ as its first argument. The RacketScript
compiler looks for these $\#\%js-ffi$ bindings to identify foreign calls. The second
argument is a symbol to tell the compiler what kind of FFI form to use:
Figure 14: Fully-expanded RacketScript FFI

- `var`: Get the value bound to given identifier from JavaScript environment. The identifier must be a valid JavaScript identifier.
- `ref`: Use JavaScript dot notation to get value mapped to given key from an object. We chain the operation if more than one identifier is given. The identifier must be a valid JavaScript identifier.
- `index`: Use JavaScript `"[]"` notation to get value mapped to a given key/index from an object/array. The key could be any arbitrary expression, but must evaluate to number/string. Like `ref`, the operation is chained for more than one key expression.
- `assign`: Use JavaScript's assignment operator.
- `new`: Use JavaScript's `new` operator to create a new instance of an object using the given constructor expression.
- `array`: Use JavaScript array syntax to construct a new array initialized with given list of items.
- `object`: Use JavaScript object syntax to construct a new object initialized with the given list of keys/value pairs. The arguments to this form take a list of keys followed by their corresponding values.

11.2 High-level API

The low-level FFI is clearly not ideal for writing readable programs. We therefore provide some high-level forms which is basically a syntactic sugar over low-level API using Racket macros.

Figure 15 shows how to create JavaScript objects and arrays using `$/obj` and `$/array` forms respectively.

*Method Chaining* is a common pattern used in JavaScript. Method chaining demands that each method returns an object, thus allowing further method calls. jQuery a JavaScript library for browsers, uses this pattern. In the following example, we use jQuery to find all `div` elements in page, change some CSS properties and register a callback for mouse click:
(define p1
  ($/obj [name "John"]
    [city "Boston"]
    [age 21]))

var p1 = {
  'name': "John",
  'city': "Boston",
  'age': 21
};

(define p2
  ($/obj [name "Jack"]
    [city "New York"]
    [age 22]))

var p2 = {
  'name': "Jack",
  'city': "New York",
  'age': 22
};

(define p3
  ($/obj [name "Jill"]
    [city "Cambridge"]
    [age 23]))

var p3 = {
  'name': "Jill",
  'city': "Cambridge",
  'age': 23
};

(define people
  [$/array p1 p2 p3])

var people = [p1, p2, p3];

Figure 15: FFI forms to create JavaScript data types.

jQuery("div").css("margin" 0)
  .css("padding" 0)
  .click(function () { alert("Hello World!") });

An exact translation of this in Racket would be hard to read. However, thanks to macros, it is easy to define a new syntax form for method chaining. We implement $> form so that the above becomes the following:

($> (jQuery "body")
  (css "margin" 0)
  (css "padding" 0)
  (click (lambda ()
    (#js=.alert "Hello World"))))

RacketScript also implements a shorthand for dot notation as a Racket reader extension. Identifiers prefixed with #js/.#js* expand to the use of the dot operator. For example, #js.console.log would be expanded to ($ console 'log). #js* works similarly, but the first identifier is translated as is and is therefore not required to be present in Racket’s namespace. Following is an example using a NodeJS library "Express" to build a simple HTTP server:
12 Evaluation

To evaluate our compiler, we implemented several thousand lines of RacketScript programs. Figure 18 presents a summary of our experiments. We compared our compiler’s output with the Whalesong compiler when appropriate. We started by writing a small web application (figure 17) in RacketScript, both client and server, to easily experiment with RacketScript programs.

Next, we re-implemented the big-bang and image library [10] in RacketScript, adapted some existing games, and observed the number of frames rendered per second (frames per second or FPS) and percentage of time spent idle.

Following is a brief description of the games we ported:

---

We do not use micro-benchmarks, because source-to-source compilation always performs better than a semantics-preserving compiler [4, 24]
<table>
<thead>
<tr>
<th>Program</th>
<th>BI</th>
<th>LOC</th>
<th>Source Size</th>
<th>Compiled Size</th>
<th>Minified</th>
<th>TPS</th>
<th>FPS (CR)</th>
<th>FPS (FF)</th>
<th>Idle Time (FF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello-world</td>
<td>No</td>
<td>2</td>
<td>N/A</td>
<td>44B</td>
<td>46B</td>
<td>46B</td>
<td>CR</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>2htdp/image</td>
<td>No</td>
<td>661</td>
<td>N/A</td>
<td>24KB</td>
<td>N/A</td>
<td>N/A</td>
<td>CR</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>2htdp/universe</td>
<td>No</td>
<td>349</td>
<td>N/A</td>
<td>12KB</td>
<td>N/A</td>
<td>N/A</td>
<td>CR</td>
<td>FF</td>
<td></td>
</tr>
<tr>
<td>Happy Birds</td>
<td>Yes</td>
<td>560</td>
<td>+87/3</td>
<td>12KB</td>
<td>20KB</td>
<td>20KB</td>
<td>2.0MB</td>
<td>2.0MB</td>
<td></td>
</tr>
<tr>
<td>Tetris</td>
<td>No</td>
<td>619</td>
<td>+71/60</td>
<td>20KB</td>
<td>1MB</td>
<td>1MB</td>
<td>31KB</td>
<td>1.4MB</td>
<td>15.9%</td>
</tr>
<tr>
<td>Worms</td>
<td>No</td>
<td>585</td>
<td>+3/3</td>
<td>24KB</td>
<td>24KB</td>
<td>24KB</td>
<td>3.8MB</td>
<td>2.8MB</td>
<td>15%</td>
</tr>
<tr>
<td>JezzBall</td>
<td>No</td>
<td>1460</td>
<td>+3/3</td>
<td>52KB</td>
<td>32KB</td>
<td>32KB</td>
<td>2.3MB</td>
<td>1.6MB</td>
<td>3.21%</td>
</tr>
<tr>
<td>NESArchery</td>
<td>Yes</td>
<td>2091</td>
<td>+47/3</td>
<td>160KB</td>
<td>1MB</td>
<td>1MB</td>
<td>1.5MB</td>
<td>2.9MB</td>
<td></td>
</tr>
</tbody>
</table>

* WS: Whalesong
* RS: RacketScript
* FPS: Frames rendered per second
* BI: Bitmap images (The RacketScript port will embed these images in source)
* FPS: Ticks per second set by on-tick parameter
* TPS: Google Chrome 55
* CR: Mozilla Firefox 50

Figure 17: The RacketScript Playground

Figure 18: Summary of our experiments
- **Flappy Birds** is a side-scrolling game where the player steers a bird through obstacles. The game uses a simple physics and is not computationally expensive. We ported Jens Axel Søgaard’s implementation.

- **2048** is a puzzle game, where the player is presented with numbered blocks and these blocks can together slide towards any given direction. Initially all blocks are numbered 2 or 4, and the objective is to combine them to form higher-valued blocks. We use Daniel Prager’s implementation which uses several higher-order functions and loops.

- **Tetris** We use Nguyễn et al. [17]’s implementation with contracts removed.

- **Worms** also known as **Snake** is an arcade game, where player controls a dot, with the goal of growing it by feeding it randomly appearing food items, and not moving into itself. This program is widely used in classrooms, in conjunction with the big-bang platform, to introduce concepts from the How To Design Programs curriculum. The implementation involves lightweight list manipulations.

- **JezzBall** starts with a room with bouncing balls, and the player can create horizontal or vertical walls to make smaller rooms. The goal is to contain balls in these smaller rooms, and avoid collision of balls with the growing walls. Since each ball recomputes its bounding box every tick, the game gets computationally expensive as the number of walls and balls increases.

- **NESArchery** is a darts like game, where the the player shoots arrows towards moving target with the goal of hitting as close to center as possible. The game employs some simple physics by creating random winds. We use Jake Lawrence’s implementation of this game.

Porting Racket programs to RacketScript generally requires some manual effort, and programmers should follow the steps below:

- Implement any unsupported primitive/library function, or replace them with something equivalent. For example, Racket sequences are currently not supported, and hence the entire `for` loop family must be replaced. These loops can be replaced with explicit recursion, or macros that do not use sequences.

- Look for arithmetic operations and make sure that they don’t break code in JavaScript environment. Specifically, look for code that compares exact numbers after division, or expect big numbers.

- Keep JavaScript’s call-stack limit in mind. Self-recursion at tail positions should be favored over normal self-recursion, especially for functions that handle huge amount of data. In general, an optimized Racket program should already be doing that.

- For big-bang programs, rename `2htdp/*` modules with our `racketscript/htdp/*`. Image loading functions such as `bitmap/file` or `bitmap/url` should be replaced with our `bitmap/data`. The `bitmap/data` function expects an image encoded using Data URI scheme (typically a base64 encoding), and hence allow embedding of images in source. In general, programmers have to ensure that all external assets are completely fetched before the game loop starts.
We were able to port six out of seven games by just following the last two steps. Figure 18 describes the porting effort in terms of number of added and deleted lines. The worst case, the 2048 game, required changing roughly 10% of the program (out 600 lines). This different was largely due to unsupported primitives such as for loops (total 30 lines) and some string functions which we do not support. We replaced loops with other higher-order function or recursion. Also, the size of some files increased after porting due to embedding bitmap images directly in the program source.

For the evaluation of performance, we tested our examples on a machine with Intel Core i7-5500U CPU @ 2.40GHz and 8GB RAM running Ubuntu 16.10 (64bit, Linux 4.8), Google Chrome (version 55) and Firefox (version 50). We manually tested each program, approximately three times each, and calculated arithmetic means for the final result. For Google Chrome, we used an FPS meter available in Chrome Tools, and periodically noted down FPS. However, this approach implies our average doesn’t take FPS fluctuations in account. Firefox in contrast, calculated average FPS throughout the execution, and hence a better instrumentation. For simple games (e.g. Flappy Birds, Tetris, 2048, NES Archery), we did not observe much difference. For computationally intensive games, however, we saw a significant FPS drop in Whalesong programs. In our implementation of JezzBall, as we increased the number of balls and walls as shown in figure 19, the amount of computation increased, resulting in major FPS drop. For the scenario shown on figure 19, Whalesong program ran at 15fps, while RacketScript program remained consistent at 60fps. We also calculated the percentage of time spent idle by these games. A higher idle-time for same fps indicates that the program is able to finish work for current tick before a program with lower idle-time. Figure 18 shows that RacketScript programs are more often idle than the Whalesong programs, while at the same time deliver similar or higher FPS. Although, Google Chrome shows us idle-time, it does not show us the proportion of total time spent idle, whereas the number was readily available in Firefox.

In terms of code-size RacketScript produces far smaller output than Whalesong. A "Hello World" Racket program compiled using Whalesong is 1.1MB but only 190KB in RacketScript. The JezzBall example is about 2.3MB when compiled with Whalesong, while 340KB when compiled with RacketScript. We compared the size of JavaScript files after feeding them to a JavaScript minifier [3]. The minifier program removed whitespaces and comments, mangled the names, and applied dead code elimination. In the table (figure 18), apart from the smaller absolute size, it is also evident that the tool was able to apply optimizations more successfully on RacketScript programs as compared to Whalesong programs.

Whalesong’s core runtime library (not including the #%kernel module) alone costs around 580KB, while RacketScript’s runtime is approximately 120KB (85KB in ES6) when compiled using Babel. However, the list of features supported by Whalesong and RacketScript differ, which makes the comparison

---

18 Firefox and Google Chrome have a FPS cap of 60fps.
inaccurate. Specifically, jQuery and Whalesong’s number library are responsible for 176KB in the total code size while RacketScript just utilizes ES6 runtime libraries (usually in ballpark of 120KB) and JavaScript numbers only. RacketScript supports a larger subset of Racket including hashes, structs, and continuation marks, which inflates its runtime. Overall, despite the imprecise comparisons, a smaller filesize is highly valued in practice no matter how it is achieved.

From our experience with RacketScript and Whalesong, we noted that RacketScript’s source-to-source compilation is more scalable not only in terms of performance, but also in terms of developer productivity (see section 3.1). Like other source-to-source [9] compilers, RacketScript programs can also benefit from third-party tools [13], to further optimize code in terms of size and performance. We already showed how UglifyJS [3] produced better results for RacketScript. For profiling, Chrome/Firefox’s built-in tools produce far more useful reports for RacketScript programs than for Whalesong programs, which helps us improve RacketScript quickly.

13 Related and Future Work

With the rise of the web, almost all popular languages target JavaScript. Many JavaScript compilers have to consider tradeoffs in terms of memory, performance, and semantics. Statically typed language have some advantage over dynamically typed language in terms of preserving semantics and offering more optimizations. For example, BuckleScript [26] rounds numbers which it knows to be an integer.
There are several compilers that target the JavaScript environment from Scheme. Whalesong [24] used trampoline based calling along with a manual stack to support delimited continuations and proper tail calls. Another notable example is Hop by Loitsch and Serrano [16], a source-to-source compiler for a Scheme dialect. They convert tail-calls to loops if possible, and otherwise use a fairly efficient trampoline if they can detect potential cycles in the call chain.

Within the Racket ecosystem, Søgaard [21] recently developed a Racket-to-JavaScript compiler, Urlang, that shares many high-level similarities with RacketScript and serves as a source of inspiration for some of parts of RacketScript's design. For example, both are lightweight source-to-source compilers that do not preserve semantics. A closer inspection, reveals some differences in the two approaches, both philosophically and implementation-wise.

Philosophically, RacketScript tries to generate JavaScript that is as "native" as possible, while still supporting a wide variety of high-level Racket features. This means eschewing some optimizations and possibly altering the behavior of some Racket programs, e.g., with numerical calculations, while relying mainly on JavaScript's JIT compilers to perform optimizations. For example, Urlang utilizes a lambda lifting pass, commonly found in most functional compilers, and hence its JavaScript lambdas require a closure record as first argument. This could lead to compatibility problems, however, if these functions are passed to a foreign function. In contrast, RacketScript favors compiling lambdas as-is, for better readability, and relies on JavaScript VM to optimize closures. RacketScript also translates data types differently: for types where RacketScript uses object constructors, Urlang uses an array literal with a tag and the value itself. Although there is no clear feedback from the JavaScript community over literals vs constructor performance, there is some indication that the performance difference is negligible in practice, which is good news for RacketScript's more idiomatic approach.

In terms of implementation, RacketScript supports a larger subset of Racket. Specifically, Urlang does not support hashes, tail-call optimization, modules, continuation marks, structure guards and structure properties. Thus Urlang programmers must program more frequently at the JavaScript level.

13.1 Future Work

We have presented the RacketScript language and compiler, which supports enough features of Racket to write many useful programs. There is plenty of room for improvement. In terms of the runtime, RacketScript has to support more primitives to be able to run more Racket programs. Some are trivial, while others (e.g. regular expressions) are not. Although, we tried to reuse some standard Racket libraries with success, we are still behind. Contracts and classes are two other major features which are currently unsupported. In terms of the compiler, RacketScript can benefit vastly from in-lining and dead-code elimination. For example, if we try to compile nested-loops, the inner loop would be wrapped inside a lambda, which is immediately-invoked during each iteration of outer loop. Such instances can clearly be inlined to
get much better performance. BuckleScript [26] removes name mangling by keeping track of variables names in the current scope, which makes generated code smaller and more readable. To make RacketScript self-hosted in future, Racket’s macro expander will have to be rewritten in Racket instead of C. Finally, one may take a different direction by compiling Racket bytecode to JavaScript as done by Vouillon and Balat [22] for js_of_ocaml.

References


