Assignment: Non-Stop JavaScript

1 Introduction

JavaScript is the programming language of web browsers and many other things. JavaScript does have peculiar features,[1] but at its core, it is a rather ordinary untyped language that has both objects and first-class functions. However, web browsers are a peculiar execution environment.

Consider the following JavaScript program, which runs an infinite loop[2]:

```
var i = 0;
while (true) {
  if (i++ % 100000 === 0) {
    println("The value of i is " + i);
  }
}
```

You can run this program from the command-line and it works as expected: it prints multiples of 100,000 until you quit the program. But, if you run it in the browser, nothing appears. Instead, your browser tab will lock up and, after a while, your browser will offer to kill the program for you.

Here is another program that does not infinite loop:

```
var N = 1000000000;
for (var i = 0; i < N; i++) {
  if (i++ % 100000 === 0) {
    println("The value of i is " + i);
  }
}
```

This program will terminate, but you will not see any output until it does. The browser tab will again lock up until the program terminates. Moreover, if your computer is slow enough, the browser will offer to kill this program too.

The event loop The browser uses an *event loop* to schedule computation. A JavaScript program running in the browser uses *event handlers* to respond to various events (e.g., clicks, key presses, timers, network responses, etc.). When an event occurs, the browser adds it to an internal *event queue*. The event loop is so named because it dequeues an event from this queue, runs the associated event handler to completion, and then loops. In particular, two event handlers never run at the same time (i.e., JavaScript is single-threaded) and the browser does not redraw the screen when an event handler is running.

A typical JavaScript program creates several event handlers and then terminates, which yields control to the browser's event loop. Therefore, when an event occurs, the event loop calls event handlers as described above. The problem with the first program is that it never terminates, thus never yields to the event loop and therefore we do not see any output. The problem with the second program is that it takes several seconds to terminate, which is what makes the browser freeze.

Yielding control to the event loop To get the expected behavior, we need to rewrite both programs to periodically yield control to the event loop. We can do this using the builtin function `setInterval(handler, t)`, which calls `handler()` after `t` milliseconds. For example, fig. 30.1a rewrites the first program to yield control to the event loop in this way. Notice that we had to eliminate the `while` loop, so this program is quite different from the original. Figure 30.1b rewrites the second program in a similar way.

[1] For a funny discussion of JavaScript's peculiar features, see the *Wat* talk by Gary Bernhardt (`https://www.destroyallsoftware.com/talks/wat`). For an academic discussion, read *The Essence of JavaScript* (`https://arxiv.org/abs/1510.00925`).

[2] Figure 30.3 defines the `println` function.

```
var i = 0;

function main() {
  if (i++ % 100000 === 0) {
    println("The value of i is " + i);
  }
  setInterval(main, 0);
}

main();
```

(a) The first program, rewritten to use `setInterval`.

```
var i = 0;
var N = 1000000000;

function main() {
  if (i < N) {
    i++;
    if (i++ % 100000 === 0) {
      println("The value of i is " + i);
    }
    setInterval(main, 0);
  }
}

main();
```

(b) The second program, rewritten to use `setInterval`.

```
function suspend() {
  C(function (k) {
    setInterval(k, 0);
  });
}

var i = 0;
while (true) {
  suspend();
  if (i++ % 100000 === 0) {
    println("The value of i is " + i);
  }
}
```

(c) The first program, with `suspend` inserted.

```
function suspend() {
  C(function (k) {
    setInterval(k, 0);
  });
}

var N = 1000000000;
for (var i = 0; i < N; i++) {
  suspend();
  if (i++ % 100000 === 0) {
    println("The value of i is " + i);
  }
}
```

(d) The second program, with `suspend` inserted.

Figure 30.1: Long running JavaScript programs must yield to the event loop.

2 JavaScript with First-Class Continuations

Imagine that JavaScript supported the *control* operator (which we will write as $\mathcal{C}$). If so, we could use `C` to define a function called `suspend` that saves its continuation, yields control to the event loop, and then schedules an event to restore the saved continuation. We could mechanically insert a call to `suspend` at the top of every function and every loop. Figures 30.1c and 30.1d rewrite our two example programs in this way. The goal of this assignment is to implement `C` for JavaScript and then do this transformation mechanically.

3 The Task

The goal of this assignment is to write a JavaScript-to-JavaScript compiler that consumes a JavaScript program P (that may block the event loop) and produces an equivalent program that periodically yields control to the event loop. You can break this into two steps:

1. Write a compiler that consumes P and produces a nearly identical program P' that simply calls `suspend` at the top of every loop and every function. Since `suspend` uses $\mathcal{C}$, P' is written in JavaScript extended with first-class continuations. (You should assume that the original program P does not use $\mathcal{C}$ itself.)

2. Write a second compiler that compiles P' to an equivalent program P'' that does not use $\mathcal{C}$. Therefore, P'' will run in an ordinary web browser. There are several ways to write this compiler. One approach is to compile P' to continuation passing style, which makes $\mathcal{C}$ trivial to implement.

Minimum Requirements JavaScript is a large, real-world programming language and has far too many features for you to fully support. Instead of prescribing a particular set of features, you should support a reasonable subset of JavaScript features. The *minimum requirement* is that you support first-class functions, `if`-statements, and objects with mutable fields. i.e., you can ignore JavaScript's loops, the `switch` statement, exceptions, getters and setters, etc. Since the assignment is open-ended, you must provide examples of programs (including non-terminating programs) that your compiler supports.

Meta Language You can use any programming language(s) that you want. All you need is a library that can parse JavaScript to an AST and print a modified AST to JavaScript source code. The OCaml code for the course has a JavaScript parser and printer. Here are some alternatives for other languages:

- If you want to use JavaScript or TypeScript as your meta language, use the Babylon parser: `https://github.com/babel/babel/tree/master/packages/babylon`.
- If you want to use Python as your meta language, the SlimIt library looks okay (but I don't have any experience with it): `https://pypi.python.org/pypi/slimit`.
- If you want to use Haskell as your meta language, use the `language-ecmascript` package: `https://hackage.haskell.org/package/language-ecmascript`.

You must include a README that documents how to build and run your program.

Beyond the Minimum Requirements There are several ways to go beyond the minimum requirements:

1. You could support more JavaScript. In order of difficulty, you could:
   - Support loops;
   - Support the `switch` statement;
   - Support `break` and `continue`;
   - Support exceptions; and
   - Support everything else (strongly not recommended).
2. You could insert calls to `suspend` more carefully. The assignment suggests calling `suspend` at the top of every function and loop. However, there are certain common cases where you can safely elide calling `suspend`.
3. Your `suspend` function could be more clever about when it actually suspends. The definition of `suspend` in figs. 30.1c and 30.1d suspends execution every time it is called. Instead, you could suspend execution every N calls to `suspend` for some moderately large N. A better approach may be to dynamically update N by sampling the current time (`Date.now()` in JavaScript). Use some benchmarks to figure out the best approach.
4. Add a "Stop" buton to the web page, which allows the user to terminate a long program whenever they choose.

If you attempt any of these (or anything else), document it in your README.

4 CPS for JavaScript

Transforming JavaScript into continuation-passing style introduces some additional challenges to the implementation details discussed in the lecture notes.

Statement/Expression Distinction Previous assignments have only worked with expression-oriented grammars, where any expression can occur more or less anywhere in the AST. For example, the following snippet is valid syntax in the grammars you have worked with so far:

```
let x = if y then 5 else 6 in
x * (let z = 10 in 2*z)
```

However, the JavaScript AST distinguishes between *statements* and *expressions*. The key difference between these two types of terms is that every *expression* evaluates to some value (or raises an error), whereas *statements* do not evaluate to anything. JavaScript only allows *expressions* to appear in the right hand side of declarations and arithmetic expressions, in loop guards, and `if` conditionals. `if`-statements, loops, and variable declarations are all *statements*, and therefore don't evaluate to a value, and can't appear in these locations in the AST.

Statements and *expressions* cannot be used interchangeably, so care must be taken when implementing CPS. In the lecture notes, the CPS transformation has the signature `cps : expr -> expr`, but this will need to be adjusted for JavaScript. Your CPS implementation should leverage mutually recursive helper functions as is shown in fig. 30.2. Any *expression* can be used in place of a *statment* by wrapping the expression with an *ExpressionStatment* constructor (`ExprStmt` in the ocaml support code), but the opposite—using a *statement* in place of an *expression*—cannot be done the same way.

```
let rec cps (stmts:stmt list) =
  match stmts with
  | [] ->
    FunExpr ([generate_id "k"], BlockStmt [])
  | hd :: tl ->
    let f = cpsStmt hd in
    let k = cps tl in
    FunExpr ([generate_id "k"], BlockStmt [
      ReturnStmt (CallExpr (f, [VarExpr k]))
    ])

  and cpsStmt (s:stmt) =
    match s with
    | ...

  and cpsExpr (e:expr) =
    match e with
    | ...
```

Figure 30.2: Skeleton CPS implementation using mutually recursive `cpsStmt` and `cpsExpr` helpers.

```
const output = document.createElement("div");
document.body.appendChild(output);

function println(str) {
  var elt = document.createElement("div");
  elt.appendChild(document.createTextNode(str));
  output.appendChild(elt);
}
```

Figure 30.3: The `println` function.

Simulating `let` with Functions This distinction presents a challenge when implementing CPS for JavaScript. The CPS transformation introduces variable declarations binding intermediate results of expressions, but this will not directly translate to JavaScript because declarations are *statements*. To work around this issue, we recommend simulating variable declarations (*statements*) with function applications (*expressions*). This will allow CPS to be implemented more closely to the presentation in the notes.

Variable declarations can be rewritten as function applications, where the declared variable becomes the function's formal parameter. For example,

```
let x = f() + g() in
let y = 10 in
x + y
```

can be rewritten as

```
(function (x) {
  return (function (y) {
    return x + y;
    })(10)
  }
})(f() + g())
```

This insight, along with the mutually recursive functions suggested above, will make it easier to maintain consistency in the types of AST nodes generated by CPS.