J is for JavaScript: A direct-style correspondence between Algol-like languages and JavaScript using first-class continuations

Olivier Danvy[1], Chung-chieh Shan[2], and Ian Zerny[1]

[1] Department of Computer Science, Aarhus University
Aabogade 34, DK-8200 Aarhus N, Denmark
({danvy,zerny}@cs.au.dk)

[2] Department of Computer Science, Rutgers University
110 Frelinghuysen Road, Piscataway, NJ 08854, USA
(ccshan@rutgers.edu)

Abstract. It is a time-honored fashion to implement a domain-specific language (DSL) by translation to a general-purpose language. Such an implementation is more portable, but an unidiomatic translation jeopardizes performance because, in practice, language implementations favor the common cases. This tension arises especially when the domain calls for complex control structures. We illustrate this tension by revisiting Landin's original correspondence between Algol and Church's lambda-notation.

We translate domain-specific programs with lexically scoped jumps to JavaScript. Our translation produces the same block structure and binding structure as in the source program, à la Abdali. The target code uses a control operator in direct style, à la Landin. In fact, the control operator used is almost Landin's J—hence our title. Our translation thus complements a continuation-passing translation à la Steele. These two extreme translations require JavaScript implementations to cater either for first-class continuations, as Rhino does, or for proper tail recursion. Less extreme translations should emit more idiomatic control-flow instructions such as `for`, `break`, and `throw`.

The present experiment leads us to conclude that translations should preserve not just the data structures and the block structure of a source program, but also its control structure. We thus identify a new class of use cases for control structures in JavaScript, namely the idiomatic translation of control structures from DSLs.

1 Introduction

It has long been routine to define a programming language by writing an interpreter in or a compiler towards a pre-existing language [27, 28, 32, 39–41]. This tradition began with John McCarthy [34], who argued that it is not a circular argument but a valid pedagogical device to use a pre-existing language as a notation for expressing computation. McCarthy drew an analogy between translations from one programming language to another and Tarski's efforts to define one mathematical logic in terms of another, "which have proved so successful and fruitful" [34, page 7] even though they still draw criticism today [19].

Translations between programming languages have also been used to reason about expressiveness. For example, Böhm and Jacopini used a compiling argument to show that flowcharts and expression languages with recursion have the same expressive power [7], and Fischer invented the CPS transformation to show the equivalence between the deletion strategy and the retention strategy to implement activation records [16]. Finally, this style of formal specification is also useful for building interpreters and compilers in practice. Indeed, it has given rise to industrial-strength software development [6].

The translation approach is alive and well today, whether the defined (source) language is considered to be domain-specific or general-purpose, and whether the defining (target) language is a variant of the lambda calculus, C, or some other language. This stream of successes is especially remarkable given that differences between the defined and defining languages often force the translation to be quite ingenious, albeit not entirely perspicuous.

Pervasive ingenuity: Some translations impose global changes on programs. For example, when the defined language features a side effect that the defining language does not support, a definitional interpreter must encode the effect throughout the translation using, e.g., state-passing or continuation-passing style [41]. These styles have since been crisply factored out into *computational monads* [38, 49].

Homomorphic ingenuity: Some translations manage to relate structural elements between source and target programs. For example, McCarthy translated flowcharts to mutually recursive equations by mapping each program point to a recursive equation and each variable to a parameter in a recursive equation [33]. These equations were tail-recursive, a property that Mazurkiewicz then used for proving properties about flowcharts [31]. Landin translated Algol programs to applicative expressions by mapping block structure to subexpression structure, assignments to state effects, and jumps to control effects [27]. Working homomorphically with a smaller defining language, namely the lambda calculus, Abdali mapped local functions into global, lambda-lifted recursive equations in a storeless fashion [1, 3], but invented continuation-passing style en passant to implement jumps [2].

Seemingly no ingenuity: Even when the translation turns out to be just an identity function, careful design may be required. For example, it is desirable for a partial evaluator to be *Jones optimal*, which means that specializing a self-interpreter to a program yields the same program modulo renaming [21]. Achieving Jones optimality requires much of the partial evaluator (e.g., it must resolve scoping references statically), but also of the interpreter: for one thing, the interpreter must be in direct style; indeed, specializing a continuation-passing self-interpreter with a Jones-optimal partial evaluator yields programs in continuation-passing style. To take another example, even though Pascal can be regarded as a subset of Scheme and hence easily translated to Scheme by (essentially) the identity function, such a translation calls for a Scheme compiler, such as Orbit [25], that is designed so that Pascal-like programs (i.e., programs with only downwards funargs[1] and downwards contargs) would be compiled as efficiently as by a Pascal compiler [26].

Shoehorning: Not all properties of the defined language are naturally supported by the defining language and its run-time system. Perhaps the best-known example is proper tail-recursion and its implementation by trampolining [4, 18, 37, 43, 47]. A close second is first-class continuations [30, 36, 44].

For humans and machines to work with a translation more easily, it is our belief that homomorphic ingenuity and seemingly no ingenuity are preferable over pervasive ingenuity and shoehorning. In short, we believe that a translation should take advantage of the expressive power of the defining language in an *idiomatic* way, if only for implementations of the defining language to execute target programs more efficiently. As shown by the examples above, this goal of idiomaticity often calls for the defined and defining languages to be carefully designed and reconciled. In other words, the principle of idiomaticity guides not just language implementation but also language design, especially today as the proverbial 700 DSLs blossom on new platforms such as browsers running JavaScript.

This article establishes an idiomatic translation between a specific and illustrative pair of languages.

- Our defined language is an Algol-like block-structured language with lexically scoped jumps. It is illustrative not just because most DSLs have block structure and lexical scope, but also because many DSLs feature complex control constructs motivated by their domains such as pattern matching and logical inference.
- Our defining language is JavaScript with first-class continuation objects, as implemented in Rhino [10]. It is illustrative not just because many DSLs are implemented by translation to JavaScript, but also because it lets us exhibit an extremely uniform and frugal translation among more complex alternatives that use a greater variety of control-flow instructions.

Taking advantage of the close correspondence between Rhino's continuation objects and Landin's J operator [14], we thus revive Landin's original correspondence between Algol programs and applicative expressions [27, 28].

[1] 'Funarg' is an abbreviation for 'functional argument' and by analogy, 'contarg' stands for 'continuation argument'. They refer to the ability of passing functions as arguments (downwards funargs) and returning functions as results (upwards funargs). The 'downward' and 'upward' orientation stems from implementing function calls with a current control stack: downwards is toward what has been pushed (and is still there) and upwards is toward what has been popped (and thus may be gone [16]).

2 Continuation objects in JavaScript

This section introduces continuation objects in the JavaScript implementation Rhino, compares them to Landin's J operator, and explains how we use them to express jumps in a block-structured language.

In Rhino, a continuation object can be created by evaluating the expression `new Continuation()`. The resulting object represents the continuation of *the call to* the function that evaluated this expression. That is, invoking a continuation object (as if it were a function) returns from the function that created it. For example, the following program only prints `2`.

```
function foo () {
    var JI = new Continuation();
    JI(2);
    print(1);
}
print(foo());
```

The continuation is undelimited and remains valid throughout the rest of the program's execution. For example, the following program prints `1` followed by an unbounded series of `2`'s.

```
var JI;
function foo () {
    JI = new Continuation();
    return 1;
}
print(foo());
JI(2);
```

2.1 Landin's translation of jumps using J

We name the captured continuations above `JI` because creating a continuation object in Rhino is equivalent to invoking Landin's J operator on the identity function [14]. The J operator is the first control operator to have graced expression-oriented programming languages. Landin invented it specifically to translate Algol jumps to applicative expressions in direct style [27, 29]. If JavaScript featured the J operator, then Landin would have translated the loop

```
      i := 1000000;
loop: i := i - 1;
      if i > 0 then goto loop;
```

to the following JavaScript program.

```
var i;
var loop = J(function () {
    i = i - 1;
    if (i > 0) loop();
});
i = 1000000;
loop();
```

The application of `J` above creates a function `loop` that, when invoked, evaluates the function body `i = i - 1; if (i > 0) loop();` with the continuation of the call to the program. In other words, `loop` denotes a “state appender” [8] and the invocation `loop()` jumps into the function body in such a fashion that the function body directly returns to the caller of the translated program. We thus express a jump to `loop` as `loop()`. This program contains two jumps to `loop`, an implicit fall-through and an explicit `goto`, so the JavaScript code above contains two occurrences of `loop()`.

The expression `new Continuation()` in Rhino is equivalent to the expression `J(function (x) { return x; })` in JavaScript with the J operator. Conversely, Landin and Thielecke [48] noted that J can be expressed in terms of JI as

$$\mathrm{J} = (\lambda c.\, \lambda f.\, \lambda x.\, c(fx))\, \mathrm{JI}.$$

Following Landin’s translation strategy, then, one might define

```
function compose (c,f) { return function (x) { c(f(x)); } }
```

in JavaScript and then translate the loop above as follows.

```
var i;
var loop = compose(new Continuation(), function () {
    i = i - 1;
    if (i > 0) loop();
});
i = 1000000;
loop();
```

(Because we translate each label to a function that takes no argument, the two occurrences of `x` in the definition of `compose` can be omitted.) This translation, like Landin’s, is attractive in that it idiomatically preserves the block and binding structure of the source program: the main program block translates to the main program block, a sequence of commands translates to a sequence of commands, and the variable `i` and the label `loop` translate to the variables `i` and `loop`.

Unfortunately, although this last translation runs in Rhino, it overflows the stack. The reason is that 1000000 calls to `c` pile up on the stack and are not discarded until the function body returns for the first time. Although we can express J in terms of JI, it is unclear how to do so without this space leak.

2.2 Our translation of jumps using JI

In order to translate jumps while preserving the stack-space usage behavior of programs, we modify Landin’s translation slightly: every captured continuation shall accept a thunk and invoke it immediately [15]. In other words, every function call shall expect a thunk to be returned and invoke it immediately.

Because we cannot change the continuation with which Rhino invokes the main program, this modification requires that we wrap up the main program in a function `main`, which returns a thunk that should be invoked immediately. Below is our final translation of the loop, which completes without overflowing the stack in Rhino.

```
var main = function () {
    var JI = new Continuation();
    var i;
    var loop = function () { JI(function () {
        i = i - 1;
        if (i > 0) loop();
    });};
    return function () {
        i = 1000000;
        loop();
    };
};
main()();
```

This use of thunks is reminiscent of trampolining, but our technique using JI does not require the caller of a function to invoke thunks repeatedly until a final result is reached. Rather, the continuation of `main()` accepts a thunk and invokes it exactly once. If another thunk needs to be invoked, such as `function () { i = i - 1; if (i > 0) loop(); }` in this example, the same continuation needs to be invoked again. In other words, the function `main` in our target program returns exactly once more than the number of times a jump occurs in the source program.

2.3 Other uses of J

Outside of its inventor and his student [8], J was first used by Rod Burstall to traverse a search tree in direct style and breadth first, using a queue of first-class continuations [9]. We have re-expressed Burstall's breadth-first tree traversal in Rhino. With its queue of first-class continuations, this program falls out of the range of our translator from pidgin Algol to JavaScript.

3 Source and target languages

The key property of our direct-style translation is that it is homomorphic and thus preserves idioms: declarations translate to declarations, blocks to blocks, commands to commands, function calls to function calls, and so forth. Since the translation is homomorphic, we dispense with it altogether in this presentation and simply consider the restricted language obtained as the image of the translation. We thus present the grammar of JavaScript programs in the image of the translation. Figure 1 displays the productions of special interest to our translation, accounting for the essential features of Algol we wish translated to JavaScript. For completeness we include the full grammar in Appendix A. This full grammar accounts for all of the example programs (see Section 4).

– A program is a sequence of declarations and commands. Such a program is translated to a top-level function in the same way procedures are, as shown by the `<program>` production. This translation is required to support labels and jumps at the top level of a program, as illustrated in Section 2.2.

```
<program>   ::= var main = function () {
                    var JI = new Continuation();
                    <dcl>*
                    return function () { <cmd>* };
                };
                main()();
<dcl-var>   ::= var <ident>;
<dcl-proc>  ::= var <ident> = function ( <formals>? ) {
                    var JI = new Continuation();
                    <dcl>*
                    return function () { <cmd>* };
                };
<dcl-label> ::= var <ident> = function () { JI(function () { <cmd>* }); };
<cmd-goto>  ::= <ident>();
<cmd-yield> ::= (function () { <ident> = new Continuation(); <cmd-goto> })();
<exp-call>  ::= <ident>( <args>? )()
```

Fig. 1. Essential grammar of JavaScript in the image of the translation.

- The `<dcl-var>` production shows that each variable declaration, is translated directly to a variable declaration in JavaScript. All variables are implicitly initialized to $\perp$, i.e., *undefined* in JavaScript.
- The `<dcl-proc>` production shows that a procedure declaration is translated to a JavaScript function that accepts the same formals and returns a thunk containing the procedure commands. In a scope visible to the thunk of commands, nested declarations are defined and the return continuation of the function closure is captured and bound to `JI`. All declarations are mutually recursive so that declarations can refer to each other as well as `JI`.
- A label consists of a name and a sequence of commands. For simplicity we assume that explicit jumps have been inserted in the program wherever execution may flow from one label to another. The `<dcl-label>` production shows that a label definition is translated to a JavaScript function of no arguments, whose body applies the return continuation `JI` to the thunk of label commands. Thus, invoking this function returns the thunk from the currently running procedure.
- The `<cmd-goto>` production shows that goto commands are translated to ordinary function calls with no arguments in JavaScript.
- The `<cmd-yield>` production shows how a *yield* command consists of rebinding the caller's label with the current continuation, followed by a goto command to transfer control. The surrounding function declaration is required to capture the current continuation. The translation must supply both the *from* and *to* labels. The labels are required to be declared in the exact same block. The *yield* command, however, may appear elsewhere.

– The `<exp-call>` production shows that a procedure call is translated to a JavaScript function call with the same arguments, and the result of the application, a thunk of procedure commands, is forced to obtain the actual result of the procedure call.

All remaining productions, found in Appendix A, show the straightforward and idiomatic translation to JavaScript.

For simplicity we consider only characters in the ASCII character set with some restrictions on their use. All identifiers are required to be alpha-numeric; the set of label identifiers must not overlap with that of variable and procedure identifiers; and `JI` is a reserved keyword and must not be used as an identifier. To declare named functions, we use variable binding and function expressions in the form of "`var <ident> = function ...`". This is necessary as function declarations can, in our translation, appear in any block statement, whereas in JavaScript function declarations are only valid at the top-level or directly inside a function body.

4 A representative collection of program samples

The goal of this section is to illustrate with classical examples the direct-style correspondence between pidgin Algol with lexically scoped jumps and JavaScript with continuation objects. We consider in turn backward jumps within the same block (Section 4.1), backward and forward jumps within the same block (Section 4.2), outwards jumps (Section 4.3), and coroutines (Section 4.4). For what it is worth, our embedding passes Knuth's man-or-boy test (Section 4.5).

Each of the following kinds of jumps, except for coroutines, can be translated as a special case using more specialized control structures offered by JavaScript. We discuss such specialized translation schemes further in Section 5.

4.1 Backward jumps

To simulate backward jumps within the same block, our translation simply declares a sequence of lexical variables, each denoting the continuation of a label, and uses one of these variables for each jump.

Backward jumps can be used to repeatedly try a computation until a condition is met, as in a loop. It can also be used to express more involved iteration patterns that recur in domains such as pattern matching and logical inference. These patterns are exemplified by the search phase of Knuth, Morris and Pratt's string matcher [24].

The KMP string matcher searches for the first occurrence of a string in a text. It first preprocesses the string into a failure table, then traverses the text incrementally, searching whether the string is a prefix of the current suffix of the text. In case of character mismatch, the string is shifted farther by a distance determined by the failure table. In their original article, Knuth, Morris and Pratt display a version of the search phase that is 'compiled'—i.e., specialized—with respect to a given string [24, Section 3] [11, Appendix]. Appendix B.1 displays this specialized version in JavaScript.

A similar illustration of backward jumps can be found in Flanagan and Matsumoto's book about the Ruby programming language, where they use first-class continuations to simulate a subset of BASIC [17, Section 5.8.3].

4.2 Backward and forward jumps

To simulate backward and forward jumps within the same block, our translation declares a group of mutually recursive lexical variables, each denoting the continuation of a label, and uses one of these variables for each jump. Appendix B.2 shows this simulation at work with Knuth's modification of Hoare's Quicksort program [23, p. 285].

4.3 Outward jumps

In modern parlance, outward jumps are exceptions. Again, our translation simulates them by declaring a lexical variable denoting the continuation of each label and using one for each jump.

Appendix B.3 displays a program that recursively descends into a binary tree, testing whether it represents a Calder mobile [13]. If one of the subtrees is unbalanced, the computation jumps out.

4.4 Coroutines

Whereas calling a subroutine transfers control to its beginning, calling a coroutine resumes execution at the point where the coroutine last yielded control [12]. Coroutines are useful for modeling domains with concurrent interacting processes. To account for coroutines, we use the `<cmd-yield>` production in Figure 1.

We have implemented a standard example of stream transducers (mapping pairs to triples [46]), Knuth's alternative version of Quicksort using coroutines [23, p. 287], and samefringe, the prototypical example of asynchronous recursive traversals [5, 20, 35]. The samefringe program is displayed in Appendix B.4.

4.5 Knuth's man-or-boy test

Finally, for what it is worth, our translator passes Knuth's man-or-boy test [22], using explicit thunks to implement call by name. Of course, this test exercises not jumps but recursion and non-local references. In that sense, it is more Rhino that passes Knuth's test than our homomorphic translator.

5 Conclusion and perspectives

We have realized, in the modern setting of JavaScript, Landin's visionary correspondence between block-structured programs with jumps and applicative expressions [27]. Our translation is idiomatic in that it maps the blocks and declarations as well as labels and jumps of the source program homomorphically to corresponding structures in the target program. The correspondence is so direct, in fact, that we can regard the image of the translation as a source language, and thus the translation as an identity function. The idiomaticity of the translation is the key to its success in producing programs that run with the same asymptotic time and space behavior as the Algol-like programs in our source language, assuming that Rhino implements downwards contargs in a reasonable way.

The simplicity and efficacy of our translation exemplifies how the principle of idiomaticity is useful for language implementation and language design: the correspondence between Landin's J operator and Rhino's continuation objects guides the implementation and design of our Algol-like language. In particular, we believe that a translation should preserve the control structures of source programs and map them to idiomatic control structures in the target language.

In this regard, our translation is extremely uniform and frugal in that it maps all control structures using continuation objects. Another extreme strategy, which some might consider less idiomatic and more pervasive, is to follow Steele [45] and translate all control structures using continuation-passing style or even trampolining. These translations call for JavaScript implementations that support either first-class continuations or proper tail recursion. Between these two simple extremes, there is an admittedly more complex middle path that relies on a control-flow analysis to detect particular patterns of control and translate them opportunistically to particular control-flow instructions in JavaScript that are more widely implemented well. For example:

– Programs that do not use labels and jumps can be translated without the thunks described in Section 2.2.
– Backward jumps within the same block can be translated to labeled `break` and `continue` statements.
– Backward and forward jumps within the same block can be translated using a loop that dispatches on a current state. (In the case of Quicksort [23], such a translation would reverse the 'boolean variable elimination' optimization that motivated Knuth's use of forward jumps in the first place.)
– Outward jumps can be translated using local exceptions.

In general, each control pattern should be translated idiomatically to, and thus constitutes a new use case for, a corresponding control structure in JavaScript.

Acknowledgments: This article was written while the second author was visiting Aarhus University in the fall of 2008. The authors are grateful to Dave Herman for an e-mail exchange about implementing finite-state machines in JavaScript and to Florian Loitsch and the anonymous reviewers for their comments. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

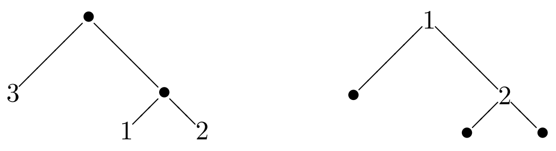
A Grammar of JavaScript in the image of the translation

```
<ascii>       ::= ....
<ident>       ::= [a-zA-Z0-9]+  -- not including JI
<program>     ::= var main = function () {
                      var JI = new Continuation();
                      <dcl>*
                      return function () { <cmd>* };
                  };
                  main()();
<dcl>         ::= <dcl-var>
                | <dcl-proc>
                | <dcl-label>
<cmd>         ::= <cmd-block>
                | <cmd-assign>
                | <cmd-goto>
                | <cmd-yield>
                | <cmd-return>
                | <cmd-if>
                | <cmd-print>
                | <cmd-exp>
<exp>         ::= <exp-var>
                | <exp-int>
                | <exp-bool>
                | <exp-chr>
                | <exp-str>
                | <exp-bottom>
                | <exp-call>
                | <exp-array>
                | <exp-index>
                | <exp-length>
                | <exp-binary>
                | <exp-nest>
<dcl-var>     ::= var <ident>;
<formals>     ::= <ident> | <ident> , <formals>
<dcl-proc>    ::= var <ident> = function ( <formals>? ) {
                      var JI = new Continuation();
                      <dcl>*
                      return function () { <cmd>* };
                  };
<dcl-label>   ::= var <ident> = function () { JI(function () { <cmd>* }); };
<cmd-block>   ::= { <dcl>* <cmd>* }
<cmd-assign>  ::= <lvalue> = <exp>;
<cmd-goto>    ::= <ident>();
<cmd-yield>   ::= (function () { <ident> = new Continuation(); <cmd-goto> })();
<cmd-return>  ::= return <exp>;
<cmd-if>      ::= if ( <exp> ) <cmd>
<cmd-print>   ::= print( <exp> );
<cmd-exp>     ::= <exp>;
<lvalue>      ::= <exp-var> | <exp-index>
```

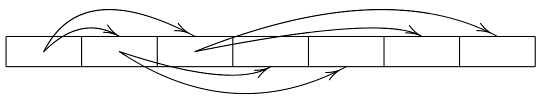
```
<exp-var>    ::= <ident>
<exp-int>    ::= [0-9]+
<exp-bool>   ::= true | false
<exp-chr>    ::= '<ascii>'
<exp-str>    ::= "<ascii>*"
<exp-bottom> ::= undefined
<args>       ::= <exp> | <exp> , <args>
<exp-call>   ::= <ident>( <args>? )()
<array-elm>  ::= <exp> | <exp> , <array-elm>
<exp-array>  ::= [ <array-elm> ]
<exp-index>  ::= <exp>[ <exp> ]
<exp-length> ::= <exp>.length
<bin-op>     ::= == | != | < | > | + | - | *
<exp-binary> ::= <exp> <bin-op> <exp>
<exp-nest>   ::= ( <exp> )
```

B Example JavaScript programs

In the following examples we represent trees in terms of arrays. Depending on the use, a tree can have either weighted internal nodes, as used in Calder mobiles (Appendix B.3), or weighted leaf nodes, as used in Samefringe (Appendix B.4).



In each case we denote the weightless nodes (●) with -1. A tree is then encoded as a linear block of values with the successor functions $\mathit{left}(n) = 2n + 1$ and $\mathit{right}(n) = 2n + 2$ as illustrated below.



B.1 Backward jumps: KMP string search

```
var main = function () {
    var JI = new Continuation();
    var kmp = function (text) {
        var JI = new Continuation();
        var k;
        var n;
        var L0 = function () { JI(function () {
            k = k + 1;
            L1();
        });};
```

```
var L1 = function () { JI(function () {
    if (text[k] != 'a') L0();
    k = k + 1;
    if (k > n) return -1;
    L2();
});};
var L2 = function () { JI(function () {
    if (text[k] != 'b') L1();
    k = k + 1;
    L3();
});};
var L3 = function () { JI(function () {
    if (text[k] != 'c') L1();
    k = k + 1;
    L4();
});};
var L4 = function () { JI(function () {
    if (text[k] != 'a') L0();
    k = k + 1;
    L5();
});};
var L5 = function () { JI(function () {
    if (text[k] != 'b') L1();
    k = k + 1;
    L6();
});};
var L6 = function () { JI(function () {
    if (text[k] != 'c') L1();
    k = k + 1;
    L7();
});};
var L7 = function () { JI(function () {
    if (text[k] != 'a') L0();
    k = k + 1;
    L8();
});};
var L8 = function () { JI(function () {
    if (text[k] != 'c') L5();
    k = k + 1;
    L9();
});};
var L9 = function () { JI(function () {
    if (text[k] != 'a') L0();
    k = k + 1;
    L10();
});};
var L10 = function () { JI(function () {
    if (text[k] != 'b') L1();
    k = k + 1;
    return k - 10;
```

```
        });};
        return function () {
            k = 0;
            n = text.length;
            text = text + "@a";
            L0();
        };
    };
    return function () {
        print(kmp("ababcabcabcacabcacab")());
    };
};
main()();
```

B.2 Backward and forward jumps: Quicksort

```
var main = function () {
    var JI = new Continuation();
    var A;
    var qs = function (m, n) {
        var JI = new Continuation();
        var i;
        var j;
        var v;
        var loop1 = function () { JI(function () {
            if (A[i] > v) {
                A[j] = A[i];
                upf();
            }
            upt();
        });};
        var upt = function () { JI(function () {
            i = i + 1;
            if (i < j) loop1();
            common();
        });};
        var loop2 = function () { JI(function () {
            if (v > A[j]) {
                A[i] = A[j];
                upt();
            }
            upf();
        });};
        var upf = function () { JI(function () {
            j = j - 1;
            if (i < j) loop2();
            common();
        });};
        var common = function () { JI(function () {
            A[j] = v;
            if (n - m > 1) {
                qs(m, j - 1)();
                qs(j + 1, n)();
            }
        });};
        return function () {
            i = m;
            j = n;
            v = A[j];
            loop1();
        };
    };
    return function () {
        A = [5, 2, 1, 4, 6, 3];
        print("Random: " + A);
        qs(0, A.length - 1)();
        print("Sorted: " + A);
    };
};
main()();
```

B.3 Outward jumps: Calder mobiles

```
var main = function () {
    var JI = new Continuation();
    var calder = function (T) {
        var JI = new Continuation();
        var fail = function () { JI(function () {
            return false;
        });};
        var visit = function (i) {
            var JI = new Continuation();
            var n;
            var n1;
            var n2;
            return function () {
                n = T[i];
                if (n == -1) return 0;
                i = i * 2;
                n1 = visit(i + 1)();
                n2 = visit(i + 2)();
                if (n1 == n2) return n + n1 + n2;
                fail();
            };
        };
        return function () {
            visit(0)();
            return true;
        };
    };
    return function () {
        print(calder([2, 5, 1, 2, 2, 4, 4,
                      -1, -1, -1, -1, -1, -1, -1, -1])());
    };
};
main()();
```

B.4 Coroutines: Samefringe

```
var main = function () {
    var JI = new Continuation();
    var t1;
    var t2;
    var traverse = function (t, f) {
        var JI = new Continuation();
        var visit = function (i) {
            var JI = new Continuation();
            var n;
            return function () {
                n = t[i];
                if (n != -1) return f(n)();
                i = 2 * i;
                visit(i + 1)();
                visit(i + 2)();
            };
        };
        return function () {
            visit(0)();
        };
    };
    var samefringe = function (t1, t2) {
        var JI = new Continuation();
        var v1;
        var v2;
        var next1 = function (e) {
            var JI = new Continuation();
            return function () {
                v1 = e;
                (function () { l1 = new Continuation(); l2(); })();
            };
        };
        var next2 = function (e) {
            var JI = new Continuation();
            return function () {
                v2 = e;
                (function () { l2 = new Continuation(); compare(); })();
            };
        };
        var l1 = function () { JI(function () {
            traverse(t1, next1)();
            next1(undefined)();
        });};
        var l2 = function () { JI(function () {
            traverse(t2, next2)();
            next2(undefined)();
        });};
        var compare = function () { JI(function () {
            if (v1 != v2) return false;
            if (v1 == undefined) return true;
            l1();
        });};
        return function () {
            l1();
        };
    };
    return function () {
        t1 = [-1, 1,
              -1, undefined, undefined, 2, 3];
        t2 = [-1,
              -1, 3, 1, 2, undefined, undefined];
        print(samefringe(t1, t2)());
    };
};
main()();
```

References

1. S. Kamal Abdali. A lambda-calculus model of programming languages, part I: Simple constructs. *Computer Languages*, 1(4):287–301, 1976.
2. S. Kamal Abdali. A lambda-calculus model of programming languages, part II: Jumps and procedures. *Computer Languages*, 1(4):303–320, 1976.
3. S. Kamal Abdali and David S. Wise. Standard, storeless semantics for ALGOL-style block structure and call-by-name. In Austin Melton, editor, *Proceedings of the International Conference on Mathematical Foundations of Programming Semantics*, number 239 in Lecture Notes in Computer Science, pages 1–19, Manhattan, Kansas, April 1986. Springer-Verlag.
4. Alan Bawden. Reification without evaluation. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 342–351, Snowbird, Utah, July 1988. ACM Press.
5. Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.
6. Dines Bjørner and Cliff B. Jones. *Formal Specification & Software Development.* Prentice-Hall International, London, UK, 1982.
7. Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.
8. William H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

9. Rod M. Burstall. Writing search algorithms in functional form. In Donald Michie, editor, *Machine Intelligence*, volume 5, pages 373–385. Edinburgh University Press, 1969.
10. John Clements, Ayswarya Sundaram, and David Herman. Implementing continuation marks in JavaScript. In Will Clinger, editor, *Proceedings of the 2008 ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 1–9, Victoria, British Columbia, September 2008.
11. Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.
12. Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.
13. Olivier Danvy. Sur un exemple de Patrick Greussay. Research Report BRICS RS-04-41, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2004.
14. Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, 4(4:12):1–67, November 2008.
15. R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.
16. Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, 1993. Available at `<http://www.brics.dk/~hosc/vol06/03-fischer.html>`. A preliminary version was presented at the ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.
17. David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language.* O'Reilly Media, Inc, Sebastopol, California, 2008.
18. Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, September 1999. ACM Press.
19. Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.
20. Carl Hewitt, Peter Bishop, Richard Steiger, Irene Greif, Brian Smith, Todd Matson, and Roger Hale. Behavioral semantics of nonrecursive control structures. In Bernard Robinet, editor, *Symposium on Programming*, number 19 in Lecture Notes in Computer Science, pages 385–407, Paris, France, April 1974. Springer-Verlag.
21. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, London, UK, 1993. Available online at `<http://www.dina.kvl.dk/~sestoft/pebook/>`.
22. Donald E. Knuth. Man or boy? *ALGOL Bulletin*, 17:7, July 1964.
23. Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, 1974.
24. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
25. David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM Press.
26. David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, February 1988. Research Report 632.

27. Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation, Parts 1 and 2. *Communications of the ACM*, 8:89–101 and 158–165, 1965.
28. Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
29. Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97)*, Technical report BRICS NS-96-13, Aarhus University, pages 1:1–9, Paris, France, January 1997.
30. Florian Loitsch. *Scheme to JavaScript Compilation.* PhD thesis, Université de Nice, Nice, France, March 2009.
31. Antoni W. Mazurkiewicz. Proving algorithms by tail functions. *Information and Control*, 18:220–226, 1971.
32. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.
33. John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Information Processing 1962, Proceedings of IFIP Congress 62*, pages 21–28. North-Holland, August 1962.
34. John McCarthy. A formal description of a subset of ALGOL. In T. B. Steel, editor, *Formal Language Description Languages for Computer Programming*, pages 1–12. North-Holland, 1966.
35. John McCarthy. Another samefringe. *SIGART Newsletter*, 61, February 1977.
36. Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In Ruth E. Davis and John R. Allen, editors, *Conference Record of the 1980 LISP Conference*, pages 154–162, Stanford, California, August 1980.
37. Yasuhiko Minamide. Selective tail call elimination. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003*, number 2694 in Lecture Notes in Computer Science, pages 153–170, San Diego, California, June 2003. Springer-Verlag.
38. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
39. F. Lockwood Morris. *Correctness of Translations of Programming Languages – an Algebraic Approach.* PhD thesis, Computer Science Department, Stanford University, August 1972. Technical report STAN-CS-72-303.
40. F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970.
41. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [42].
42. John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.
43. Michel Schinz and Martin Odersky. Tail call elimination on the Java Virtual Machine. In Nick Benton and Andrew Kennedy, editors, *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, number 59 in Electronic Notes in Theoretical Computer Science, pages 155–168, Firenze, Italy, September 2001. Elsevier Science.
44. Richard M. Stallman. Phantom stacks: If you look too hard, they aren't there. AI Memo 556, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 1980.

45. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.
46. Carolyn L. Talcott. *The Essence of $\mathcal{R}$um: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, August 1985.
47. David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.
48. Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):141–160, 2002.
49. Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.