

Principles of Programming Languages

From Recursion to Iteration Through CPS

The Problem We're Solving Today

We observed in a previous lecture that with our *L5* interpreter because it is executed in a JavaScript interpreter which does NOT implement Tail Call Optimization, we run out of stack space even when we execute a tail-recursive function in *L5*.

The Problem We're Solving Today

In this section, we introduce a general method which models recursion explicitly - so that we can properly implement tail-recursion as iteration even when our meta-language does not support tail call optimization. In other words, our implementer explains how to turn tail recursion into iteration.

The Problem We're Solving Today

We first illustrate the general approach on a concrete example (a simple tail recursive function written in TypeScript, which we turn into an equivalent iterative program in JavaScript through gradual semantic transformations).

The Problem We're Solving Today

We then implement the same method on the *L5* interpreter and obtain an iterative version of the interpreter. We finally demonstrate that when we execute the iterative interpreter on a tail-recursive *L5* program, this yields an iterative execution process which does not consume control memory.

The Problem We're Solving Today

The end-result is an interpreter which executes tail-recursive programs as iteration even though the meta-language does not implement tail recursion optimization.

From Recursion to Tail Recursion Using the CPS Transformation

Consider the following recursive function in TypeScript:

```
const sum = (n: number): number =>  
  (n === 0) ? 0 : n + sum(n - 1);
```

This function is recursive (because the recursive call does not appear in tail position). We know that if we execute it on a large value for n , it will create a stack overflow error.

From Recursion to Tail Recursion Using the CPS Transformation

We could instead rewrite this function as a tail recursive function:

```
const sumIter = (n: number, acc: number): number =>
  (n === 0) ? acc : sumIter(n - 1, n + acc);
```

This version is tail-recursive. On an interpreter that would perform Tail Call Optimization (TCO), this function would be executed in an iterative manner. But JavaScript does not perform TCO, and therefore, this function will also cause a stack overflow when it is executed on a large value of n .

From Recursion to Tail Recursion Using the CPS Transformation

We could, of course, write the same function directly in an iterative manner:

```
const sumLoop = (n: number): number => {  
  let sum = 0;  
  for (let i = 0; i <= n; i++) {  
    sum += i;  
  }  
  return sum;  
}
```

What we want to obtain instead is a systematic transformation process which will turn any recursive function into a semantically equivalent function which can be executed iteratively – that is, without consuming control memory.

We demonstrate this transformation as a series of smaller transformations - starting with the CPS transformation, which is the key semantic transformation involved in this overall compilation of the recursive program into an equivalent iterative program.

Note that when we perform the CPS transformation, we do not transform `sum` into `sumIter`. `sumIter` implements a different algorithm than `sum`. This is the result of the CPS transformation applied on the TypeScript code of `sum`:

From Recursion to Tail Recursion Using the CPS Transformation

```
type Cont = (x: number) => number;
```

```
const sumCPS = (n: number, cont: Cont): number =>  
  n === 0 ? cont(0) :  
  sumCPS(n - 1, (sn1) => cont(n + sn1));
```

```
const sumCPS1 = (n: number): number =>  
  sumCPS(n, (x) => x);
```

From Recursion to Tail Recursion Using the CPS Transformation

The resulting program is tail recursive (which is always the case for a CPS program).

But because JavaScript does not apply TCO, when we invoke this program, we still obtain a stack overflow:

```
sumCPS1(10000);  
// RangeError: Maximum call stack size exceeded
```

The reason we still suffer from control memory consumption is that even for tail calls, the JavaScript interpreter internally allocates a stack frame.

That is, in the tail call:

```
sumCPS(5, cont); // ->  
    sumCPS(4, (sn1) => cont(5 + sn1))
```

we still allocate a call frame on the stack.

In fact, in JavaScript, we **always** allocate a call frame whenever we invoke a function - regardless of the position of the function application. If we want to avoid allocating call stacks, we must avoid calling functions. How can we achieve this for the cases of continuations?

We will achieve this objective in two steps:

- We will transform procedural continuations into concrete data structures which implement the same interface as continuations.
- We will transform the invocation of continuations into an iterative process without parameter passing and without return address.

Up to this point, we described continuations as closures which receive as parameter the result of the current function and pass this parameter to the continuation of the code.

These closures encapsulate any local variable that does not depend on the result. When a procedure is transformed in CPS-form, the structure of the resulting function is:

From Procedural Continuations to Concrete Continuations

- A procedure which receives a single parameter
- All computation which does not invoke a user-defined procedure (any sequence of primitive applications) is performed at the beginning of the computation.
- The result of this computation is then passed to the continuation.

In this context, when we invoke a CPS function, we **construct** its continuation as the last parameter of the CPS function. That is, each time we invoke a CPS function with a continuation, we construct a new continuation instance with the specific parameters this continuation receives.

Within the body of the continuations, we invoke the continuation passed as a parameter with different parameters.

We, accordingly, can distinguish two contexts in which continuations participate:

- Places where a new continuation is constructed with its specific parameters
- Places where a continuation is invoked

From Procedural Continuations to Concrete Continuations

For example, in the `sumCPS` example:

```
type Cont = (x: number) => number;

const sumCPS = (n: number, cont: Cont): number =>
  (n === 0) ? cont(0) : // The cont parameter is invoked

  // A new continuation is constructed with n and cont
  // in its memory. In the body of this continuation,
  // the cont continuation is invoked.
  sumCPS(n - 1, (sn1) => cont(n + sn1));

const sumCPS1 = (n: number): number =>
  // The default identity continuation is
  // constructed (it has no memory)
  sumCPS(n, (x) => x);
```

In the first step of the transformation from CPS to iteration, we transform procedural continuations into concrete data structures. This transformation is systematic:

- We identify all the types of continuations which appear in the program. In our trivial `sum` example, there are only two distinct continuation types, the one constructed inside `sumCPS` and the one constructed in the driver function `sumCPS1`.

- We give them distinct names: in our case, **Cont1** and **IdCont**.

- For each type of continuation, we identify which variables are closed in the corresponding closure: those are the variables which must be remembered in the activation frame of the corresponding closure. In our case, for **Cont1** the variables captured in the closure are **cont** and **n**; for **IdCont**, no variables are captured.

- We define a **disjoint type union system** to represent these continuations. In our case, we define the interfaces **Cont1** and **IdCont** - each one with the captured variables as members of the interface, a discriminative tag for each interface, and a parent type union to represent all possible continuations in a polymorphic manner.

- Finally, we define a polymorphic function **applyCont** which dispatches on the type of the **cont** and executes the body of the continuation according to its type when it is invoked.

- In all places in the code where a continuation is constructed, we invoke the appropriate continuation constructor with the relevant parameters, and in the places where a continuation is invoked, we explicitly call **applyCont** on the concrete **cont** parameter.

With this transformation, we obtain the following code for the **sumCPS** example:

Concrete Continuations vs. Procedural Continuations

```
type CCont = IdCont | Cont1;

interface Cont1 {tag: "Cont1"; n: number; cont: CCont};
const makeCont1 = (n: number, cont: CCont): Cont1 =>
  ({tag: "Cont1", n: n, cont: cont});
const isCont1 = (x: any): x is Cont1 => x.tag === "Cont1";

interface IdCont {tag: "IdCont"};
const makeIdCont = (): IdCont =>
  ({tag: "IdCont"});
const isIdCont = (x: any): x is IdCont => x.tag === "IdCont";

const applyCont = (cont: CCont, val: number): number =>
  isIdCont(cont) ? val :
  isCont1(cont) ? applyCont(cont.cont, cont.n + val) :
  -1;

const sumCPSC = (n: number, cont: CCont): number =>
  (n === 0) ? applyCont(cont, 0) :
  sumCPSC(n - 1, makeCont1(n, cont));
```

Concrete Continuations vs. Procedural Continuations

```
const sumCPS2 = (n: number): number =>  
  sumCPSC(n, makeIdCont());
```

This implementation of the CPS program with concrete continuations is still not iterative: we invoke JavaScript functions each time we construct a continuation and each time we invoke one. These are places where we will still consume stack memory.

We use functions in the `sumCPS2` program for 3 reasons:

- To pass the appropriate parameters to the continuation constructors
- To pass the appropriate parameters to the `applyCont` polymorphic method
- To know where to return in the code execution once the procedures complete so that we can continue processing

Observe, however, that because we are operating on code in CPS form, the function calls are all in tail position. This means that we do **not** need to remember where to return to, we can simply go to the next call and never return.

We observe that the continuation constructors all have the same structure – they simply initialize the fields of a record with the parameters. Hence, we are ensured that invoking a continuation constructor will consume a single stack frame and will not chain to further function calls.

In contrast, `applyCont` often yields recursive chains of function calls.

In this step, we remove the need to use a stack frame to pass parameters to the functions involved in the CPS program.

Instead, we define a finite set of **registers** - that is, a set of variables of the appropriate types which are defined over the scope of the whole program (all the functions involved in the CPS transformation). There is one register for each parameter of all the continuation constructors and the **applyCont** function.

Instead of invoking a function by passing parameters - we initialize the corresponding registers and then call a function of zero parameters. In our example, this transformation yields the following program:

Registerization of Concrete Continuations

- We define the registers **nREG** and **contREG** for the parameters of the **Cont1** constructor
- We define the register **valREG** for the parameter to the **applyCont** function
- We initialize the registers in the driver function **sumREG1**
- We transform each function invocation with parameters into a sequence of assignments to the registers and then invocation of the function without parameters.

The result of this transformation can be found
[here](#).

Why Registers Can Be Safely Overridden

The set of registers we defined replace activation frames in a stack. Consider why in the current state of our transformation, it is safe to use a single flat set of registers instead of stacked activation frames.

Why Registers Can Be Safely Overridden

The reason is that in CPS form, when we enter a user-defined function, we read the parameters (in our case from the registers), execute a set of primitive computation steps, and then move on to the tail call.

Why Registers Can Be Safely Overridden

In particular:

- After a tail call, we do not return to the function from which we invoked the tail call. This means that we do not need to read values from the activation frame anymore after the tail call is entered.
- It is, therefore, safe to override the current register values with other values that will be consumed in the tail call.

Realize that the registerization transformation
is safe **only on code in CPS form.**

Observe the code in the `sumREG1` program. We still invoke unbounded functions (that is, functions which can invoke more user defined functions in an unbounded chain of calls):

- `applyContReg()`
- `sumCPSReg()`

These functions have zero-argument by construction, but their invocation still consumes stack frames. In the last step of the transformation, called **pipelining**, we now transform these calls into an explicit loop.

In the register version of the program, we still invoke zero-parameter functions, all in tail positions. We want to avoid calling functions altogether to avoid consuming activation frames on the stack.

The solution we introduce is to convert tail calls with no parameters into the equivalent of a “computed GOTO” instruction in a very simple “abstract machine”. An abstract machine has a set of pre-defined instructions and a set of registers.

In our case, these instructions are the different types of continuations we can compute in the process of the function execution. We add a single instruction to indicate that we have reached the end of the computation and that the Virtual Machine can shutdown and return the value of a specific register.

To encode the name of the instruction of the next command to be executed, we introduce an additional register - traditionally called the **Program Counter** (or PC). The type of this register is an enumeration of the possible continuation types. We call this type the **InstructionSet** of the virtual machine.

Finally, we define a **VM()** procedure which implements the logic of the virtual machine:

- Fetch: the instruction to be executed and the corresponding registers
- Decode: dispatch the name of the instruction to the appropriate sequence of primitive calls
- Execute: execute the sequence of primitive calls and update the registers to prepare for the next instruction

All procedures in the implementation are defined in the scope of the registers. To invoke a new procedure, we end each call by setting **pcREG** to the name of the procedure we want to execute next. Setting **pcREG** in tail position is equivalent to invoking a procedure in tail position.

This transformation of calls of procedures in tail position to an explicit loop and dispatch over the PC register yields the following version of the program.

The resulting program is an iterative implementation of the original program.

Because the original program was tail-recursive, the resulting program is also iterative.

The Registerized VM is Iterative

We confirm this through empirical tests:

```
sum(10000);      // Stack overflow
sumCPS1(10000);  // Stack overflow
sumCPS2(10000);  // Stack overflow
sumREG1(10000);  // Stack overflow
sumREG2(10000);  // 100010000 / 2 = 50005000
```

This was quite a lot of work to turn a single tail recursive procedure into an iterative program. The benefit of this method is that, by relying on the properties of the CPS transformation, we can transform **any program** into a corresponding iterative program.

In particular, we can transform the code of the L5 interpreter into an iterative interpreter which does not consume JavaScript stack.

Naturally, this program will still consume some memory when it executes a recursive program. But this memory will be managed explicitly, in the form of a concrete continuation data structure. It will not rely implicitly on the stack of the meta-language (JavaScript).

As a consequence, if we execute an *L5* tail-recursive program, this iterative interpreter will consume bounded control memory - as expected of a Scheme interpreter.

We illustrate this transformation in multiple steps:

- *L5* is the original recursive interpreter. It is recursive because the various evaluation methods are by definition recursive over the inductive structure of the input AST.

Transforming the L5 Interpreter into an Iterative Interpreter

- *L6* is the CPS transformation of *L5*. We simply apply the rules of the CPS transformation over the code of *L5* in TypeScript. In this CPS transformation, we only include procedures which can lead to unbounded execution. Any user-defined procedure which has bounded execution is left unchanged (for example, **applyPrimitive** and all the procedures manipulating the AST are left unchanged)

Transforming the L5 Interpreter into an Iterative Interpreter

- *L7a* is the transformation of *L6* with explicit continuation ADT but still using closures for the ADT implementation
- *L7b* turns continuations into concrete data structures
- *L7c* introduces the iterative version of the interpreter with registers and explicit dispatch according to **pcREG**.

Transforming the L5 Interpreter into an Iterative Interpreter

To validate the transformation, we confirm that the evaluation of the following *L7* program completes without stack overflow:

```
const evalP = (x: string): Result<Value> =>
  bind(parseL5(x), evalProgram);

expect(evalP(`
  (L5 (define sumCPS
    (lambda (n cont)
      (if (= n 0)
          (cont 0)
          (sumCPS (- n 1)
                  (lambda (sn1) (cont (+ n sn1)))))))
    (sumCPS 1000 (lambda (x) x)))`))
.to.deep.equal(makeOk(500500));
```