Normal-order direct-style beta-evaluator with syntax-rules, and the repeated applications of call/cc

- Repeated applications of `call/cc`, formally
- Normal-order direct-style beta-normalizer as syntax-rules
- Use (2) to prove (1)
- A few less common examples

The full title of the talk, in the expanded form, is _this_. We will talk about less frequently mentioned applications of call/cc and hygienic Scheme macros. For example, we will use macros as a sort of a proof assistant, to help in tedious lambda-calculations. As you can see, we will be talking about undelimited and delimited continuations, continuation-passing style transforms – with a detour on hygienic macros and beta-normalization. It should come as no surprise that all these topics are directly influenced by Dan Friedman – and, in fact, developed straight in response to his challenges. Perhaps he didn't know about that though.

The call/cc challenge

```
((call/cc call/cc) (call/cc call/cc))
```

```
(call/cc call/cc) ==> ??
(call/cc (call/cc call/cc)) ==> ??
(call/cc ... (call/cc call/cc)) ==> ??
(call/cc ... (call/cc id)) ==> ??
```

```
(define (p x) (if (eq? x p) '(p p) `(p ,x)))
((call/cc (call/cc call/cc)) p)  ===> (p p)
```

It all started about two years ago, upon the meditation on this famous incantation. It was disclosed by Shriram on USENET. What is the application of call/cc to itself? And one more time? And a few more times? What if we have the identity function?

To be sure, on one hand, these questions seem trivial. We can easily find the answer if we just apply these things to a suitable function, like that. It'll tell us what is being applied to what. So we easily find out that all these expressions are just self-applications. A more interesting question is this: are there any side-conditions on p? Can we prove it? As we know, the field of first-class continuations is sufficiently interesting to reasonably doubt one's intuition and rely on formal proof instead. BTW, there was one surprise, to be mentioned later.

This is an instance of a type-directed partial evaluation, btw.

The call/cc theorem

Theorem 1. Expressions

- `((call/cc ... (call/cc call/cc)) p)`
- `((call/cc ... (call/cc (call/cc id))) p)`
- `((lambda (x) (x x)) p)`

where `p` is a value,

all yield β-equivalent terms after CBV CPS.

Catchy phrase 1.
self-application is the fixpoint of call/cc

Here (call/cc ...) signify zero or more applications of call/cc, and id is the identity function (lambda (x) x).

Proving the theorem: CPS

```
(define-syntax CPS
  (syntax-rules (lambda call/cc p)
    ((CPS (?e1 ?e2))
      (lambda (k) ((CPS ?e1) (lambda (f)
         ((CPS ?e2) (lambda (a) ((f a) k)))))))
    ((CPS (lambda (x) ?e))
      (lambda (k) (k (lambda (x) (CPS ?e)))))
    ((CPS call/cc)
      (lambda (k0)
        (k0 (lambda (p) (lambda (k)
              ((p (lambda (a)
                    (lambda (k1) (k a)))) k)))))
    ((CPS p)
      (lambda (k) (k pv)))
    ((CPS ?x)
      (lambda (k) (k ?x)))))
```

We have mentioned CPS in the formulation of the theorem, so the best way to prove it would be via the CPS transform. CPS is a source-to-source translation. Therefore, it behooves us to use Scheme's syntax-transformers.

This macro is straightforward: it is the standard CPS transformation written in Latin rather than in Greek.

Here we see the CPS for application, abstraction, call/cc, our skolem constant for a value, and just other value.

Need a more perspicuous CPS

```
> (expand '(CPS (lambda (x) (x x))))
(lambda (#:k)
  (#:k (lambda (#:x)
         (lambda (#:k)
           ((lambda (#:k) (#:k #:x))
            (lambda (#:f)
              ((lambda (#:k) (#:k #:x))
               (lambda (#:a) ((#:f #:a) #:k)))))))))
```

We will be using Petite Chez Scheme, which conveniently provides a form `(expand e)` to macro-expand an expression `e`. We see a problem however: the expansion isn't entirely easy to look at: it has a bit too many lambdas. We know how to deal with that: we just have to reduce them. We need a beta normalizer. So, we take a detour on lambda-calculators.

Hilsdale-Friedman λ-calculator

```
(define-syntax beta1
  (syntax-rules (lambda)
    ((beta1 (lambda (Formal) Body) SK FK)
     (beta1 Body (beta1-lambda-k SK Formal) FK))
    ((beta1 ((lambda (Formal) Body) Arg) SK FK)
     (lc-subst Arg Formal Body SK))
    ((beta1 (Op Arg) SK FK)
     (beta1 Op (beta1-op-k SK Arg)
         (beta1 Arg (beta1-arg-k SK Op) FK)))
    ((beta1 X SK FK) (apply-syn-cont FK))))

(define-syntax beta*
  (syntax-rules ()
    ((beta* Exp K)
      (beta1 Exp (beta*-k K)
        (apply-syn-cont K Exp)))))

(define-syntax beta*-k
  (syntax-rules ()
    ((beta*-k K NewExp) (beta* NewExp K))))
```

We are looking for a beta-normalizer as a source-to-source transformer – that is, again as a Scheme macro. At the Scheme workshop 2000 in Montreal Dan Friedman and Erik Hilsdale presented a paper on a *systematic* writing of hygienic Scheme macros. The key turns out to be CPS. After that paper, and an incidental discovery of macro-lambda, a lot of CPS macros have been written. A lot here applies both to the number of macros of this sort as well to each macro of that sort.

One of the examples in their paper was a normal order beta-normalizer. Here's a *short* excerpt. The code for macros such as beta1-lambda-k and lc-subst occupies a two-column page in the paper.

As you can see, the normal-order normalizer attempts to reduce a term – and if successful, reduces the result, etc. This algorithm can be expressed by a phrase 'cook until done'. I should say that this method did not appeal to me, probably because I dislike cooking. So, I chose a different algorithm – which is that of bottom-up parsing, with shift and reset, I mean, reduce.

Direct-style λ-calculator

```
(define-syntax NORM
  (syntax-rules (lambda)
    ((NORM t) (NORM t () ()))
    ((NORM (lambda (x) e) env ())
     (let-syntax ((ren (syntax-rules ()
       ((ren ?x ?e ?env)
         (lambda (x) (NORM ?e ((?x () x) . ?env) ()))))))
       (ren x e env)))
    ((NORM (lambda (x) b) env ((enve e) . stack))
      (NORM b ((x enve e) . env) stack))
    ((NORM (e1 e2) env stack)
      (NORM e1 env ((env e2) . stack)))
    ((NORM x () ()) x)
    ((NORM x () ((enve e) ...))
      (x (NORM e enve ()) ...))
    ((NORM x env stack)
      (let-syntax
        ((find
           (syntax-rules (x)
             ((find ?x ((x ?envs ?es) . _) ?stack)
              (NORM ?es ?envs ?stack))
             ((find ?x (_ . ?env) ?stack)
              (NORM ?x ?env ?stack)))))
        (find x env stack)))
    ))
```

This calculator is written in a *direct* style. There are no success and failure continuations as we saw on previous slide. It is also a *single*, stand-alone macro. It does no alpha-renaming directly and no substitutions directly. Everything is integrated with normalization, and everything is delayed until the latest possible moment. As you can see, the only recursion here is via `NORM` itself. The auxiliary macros are not directly recursive and exit back to `NORM`.

This is my second attempt. The first had a substitution semantics: when processing a redex, the normalizer will traverse and rebuild the term being substituted into. However, that rebuilding was shallow: we stop at the first application. Rather than traverse both sub-terms of an application, we make a promise to traverse further later. We make such a promise by building beta-redexes on both sub-terms. Indeed, a beta-redex is a reification of a substitution. Alas, that first normalizer was not lazy enough – it would build large intermediate terms and could quickly run out of memory. Also, the first attempt was bigger: the substitution part was as big as this whole normalizer.

Sample normalization

$$(\lambda x.x\,x)(\lambda x.x)\ env_0\ stack_0$$

$$\mathsf{let}\, s_1 = (\lambda x.x)e_0 : s_0\, \mathsf{in}\ \ \lambda x.x\,x\ \ e_0\ \ s_1$$

$$\mathsf{let}\, e_1 = x.(\lambda x.x)e_0 : e_0\, \mathsf{in}\ \ xx\ \ e_1\ \ s_0$$

$$\mathsf{let}\, s_2 = xe_1 : s_0\, \mathsf{in}\ \ x\ \ e_1\ \ s_2$$

$$\lambda x.x\ \ e_0\ \ s_2$$

$$\mathsf{let}\, e_2 = x.xe_1 : e_0\, \mathsf{in}\ \ x\ \ e_2\ \ s_0$$

$$x\ \ e_1\ \ s_0$$

$$\lambda x.x\ \ e_0\ \ s_0$$

if `e0` and `s0` are empty:

$$\lambda x_9.\ \mathsf{let}\, e_3 = x.x_9()\, \mathsf{in}\ \ x\ \ e_3\ \ ()$$

To see how that macro works, let's normalize this sample term given the initial environment `env0` and the term stack `stack0`. We will abbreviate those just to `e0` and `s0`. As I said, the algorithm is almost the same as that of bottom-up parsing. The term stack stores terms to be applied to the current term.

At every time, we maintain the following invariant:

$$head\ (\underbrace{term\ ...}_{stack}) \equiv head\ term\ ...$$

In addition to the stack, we have the environment. Our normalizer implements the calculus of explicit substitutions, and the environment is the record of those. It is actually a stack, to handle shadowing of bound variables.

So, our initial term is an application. We shift its argument onto the term stack – along with the environment in effect, `e0`. The current term now is an abstraction, the term stack is not empty – and so we have a redex. We remove the top of the term stack – along with its saved environment – and push it into the environment stack, associated with the bound variable `x`. And that is all we do for a redex. No extra term traversals, no alpha-renaming. The current term now is abstraction's body, and we process it as usual. We will deal with the substitution only when we are pressed. The current term is now an application, of `x` to `x`, and we handle it as before: shift the argument into the term stack,

along with the current environment `e1`. The current term now is a bare variable – and now we are pressed for the substitution. We reduce that variable against the environment stack: we shift the associated term, along with its environment, out of that stack into the current position. And continue. The re-parsing is the only difference between our normalizer and a bottom-up parser.

Now, we get the redex again: we reduce from the term stack and shift onto the environment stack. The current term becomes just `x`, we reduce from the environment and get our answer. As you can see we never traverse a term just for the sake of substitutions. Substitution becomes a part of overall normalization. We don't do an explicit alpha-conversion either. Only when we encounter a lambda on an empty stack – we dive underneath and have to change the bound variable into something else. Again, we do not rush into the renaming: we merely create a record on the environment to do the replacement when we are pressed into it, that is, when we come across the bound variable.

Direct-style λ-calculator

```
(define-syntax NORM
  (syntax-rules (lambda)
    ((NORM t) (NORM t () ()))
    ((NORM (lambda (x) e) env ())
     (let-syntax ((ren (syntax-rules ()
       ((ren ?x ?e ?env)
         (lambda (x) (NORM ?e ((?x () x) . ?env) ()))))))
       (ren x e env)))
    ((NORM (lambda (x) b) env ((enve e) . stack))
      (NORM b ((x enve e) . env) stack))
    ((NORM (e1 e2) env stack)
      (NORM e1 env ((env e2) . stack)))
    ((NORM x () ()) x)
    ((NORM x () ((enve e) ...))
      (x (NORM e enve ()) ...))
    ((NORM x env stack)
      (let-syntax
        ((find
           (syntax-rules (x)
             ((find ?x ((x ?envs ?es) . _) ?stack)
              (NORM ?es ?envs ?stack))
             ((find ?x (_ . ?env) ?stack)
              (NORM ?x ?env ?stack)))))
        (find x env stack)))
    ))
```

Here's our macro again. This short incantation is to get the macro-expander to give us a fresh name with the same stem. When we view the result of the macro-expansion – and we will – it's nice to see variable names close to the ones that existed in the original term.

So, with two stacks, we have two shifts and two reductions. Applications shift onto the term stack. Redexes reduce the term stack and shift onto the environment stack. Variables reduce from the env stack.

Again, this single, stand-alone macro is all there is to lambda-calculus, really. As we all know, it is indeed very simple on the surface.

CPS with normalization

```
> (expand '(CPS (lambda (x) (x x))))
(lambda (#:k)
  (#:k (lambda (#:x)
         (lambda (#:k)
           ((lambda (#:k) (#:k #:x))
            (lambda (#:f)
              ((lambda (#:k) (#:k #:x))
               (lambda (#:a) ((#:f #:a) #:k))))))))

> (expand `(NORM
            ,(expand '(CPS (lambda (x) (x x))))))
(lambda (#:k)
  (#:k (lambda (#:x)
     (lambda (#:k) (#:x #:x #:k)))))
```

Now that we've got the normalizer, we can apply it to the result of the CPS transform. Here's one way of doing it. As you can see, the result is indeed more comprehensible.

However, this nested use of `expand` isn't nice, let alone cumbersome to type. As I said, I am lazy. We can integrate `NORM` into the CPS – performing a deforestation.

CPS with normalization, truly

```
(define-syntax CPS
  (syntax-rules (lambda call/cc p)
    ((CPS (?e1 ?e2) . args)
      (NORM (lambda (k) ((CPS ?e1)
          (lambda (f) ((CPS ?e2) (lambda (a)
             ((f a) k)))))) . args))
    ((CPS (lambda (x) ?e) . args)
      (NORM (lambda (k)
         (k (lambda (x) (CPS ?e)))) . args))
    ((CPS call/cc . args)
      (NORM (lambda (k0)
        (k0 (lambda (p) (lambda (k)
              ((p (lambda (a)
                    (lambda (k1) (k a)))) k)))))
                          . args))
    ((CPS p . args)
      (NORM (lambda (k) (k pv)) . args))
    ((CPS ?x . args)
      (NORM (lambda (k) (k ?x)) . args))))


(define-syntax NORM
  (syntax-rules (lambda CPS)
    ((NORM (CPS e) env stack) (CPS e env stack))
    ...
```

That is quite straightforward. We add to the CPS transformer the opaque normalization state – args – to carry around. We make the transformer do the normalization after the transformation. We also add one more rule to the normalizer, to handle delayed CPS. CPS and normalization are truly intertwined.

Incidentally, we could easily mark which lambda's come from the source expression and which come from the CPS itself – so the normalizer will handle the administrative lambdas only.

OTH, for the theorem we have to make a few further reductions. So, here we make no distinction between administrative and serious lambdas and try to reduce what we can. We use this as an assistant anyway: if it takes too long, there is always Control-C.

Proving the theorem 1/3

Lemma 1. CPS transform of $(\lambda x.x\,x)p$ is $\lambda k.pv\,pv\,k$

Proof:

```
> (expand '(CPS ((lambda (x) (x x)) p)))
(lambda (#:k) (pv pv #:k))
```

Lemma 2. CPS of $call_{cc}\,call_{cc}$ is $\lambda k.k\,(\lambda a.\lambda\,k_1\,.\,k\;\;a)$

Proof:

```
> (expand '(CPS (call/cc call/cc)))
(lambda (#:k)
  (#:k (lambda (#:a) (lambda (#:k1) (#:k #:a)))))
```

We are all set for proving our theorem. We start with the following lemmas.

Proving the theorem 2/3

Lemma 3. CPS transform of $call_{cc}\,(call_{cc}\,call_{cc}\,)$ is the same as that of $call_{cc}\,call_{cc}$

Proof:

```
> (expand '(CPS (call/cc (call/cc call/cc))))
(lambda (#:k)
  (#:k (lambda (#:a) (lambda (#:k1) (#:k #:a)))))
```

Lemma 4. CPS transform of $call_{cc}\,(call_{cc}\,id)$ is the same as that of $call_{cc}\,call_{cc}$

Proof:

```
> (expand '(CPS (call/cc
                  (call/cc (lambda (u) u)))))
(lambda (#:k)
  (#:k (lambda (#:a) (lambda (#:k1) (#:k #:a)))))
```

Proving the theorem 3/3

Lemma 5. CPS transform of $(call_{cc}\, call_{cc}\,)p$ is the same as that of $(\lambda x.x\, x)p$

Proof:

```
> (expand '(CPS ((call/cc call/cc) p)))
(lambda (#:k) (pv pv #:k))
```

The call/cc theorem

Theorem 1. Expressions

- `((call/cc ... (call/cc call/cc)) p)`
- `((call/cc ... (call/cc (call/cc id))) p)`
- `((lambda (x) (x x)) p)`

where `p` is a value,

are observationally equivalent in CBV.

Follows by Plotkin simulation theorem

$\Phi\,(eval_v M) = eval_v((\Psi\, M)(\lambda x.x))$

As our lemmas, and their trivial inductive extension, showed, all these three expressions have identical CPS transforms. So, our theorem follows from Plotkin simulation theorem.

Catchy Conclusions

- *self-application is the fixpoint of call/cc*
- *macro-expander is a proof assistant*
- *direct normal syntax-rule lambda-calculator*

`pobox.com/~oleg/ftp/Scheme/callcc-fixpoint.txt`

These catchy phrases conclude the talk.

But no computer science paper is complete without working out a factorial or Fibonacci. Let us indeed consider a factorial. As we saw, we can get self-application via call/cc. And self-application is at the heart of the Y combinator. So we get the tantalizing opportunity to write the factorial function like this.

Factorial via call/cc

```
(begin
  (define fact
    ((lambda (f)
       ((lambda (u) (u (lambda (x)
                 (lambda (n) ((f (u x)) n)))))
        (call/cc (call/cc (call/cc
           (call/cc (call/cc (lambda (x) x))))))
      (lambda (f) (lambda (n)
           (if (<= n 0) 1 (* n (f (- n 1)))))))))
  (display (map fact '(5 6 7)))
  (newline))
```

Note, that there is no overt recursion nor iteration nor self-application.

Note the top-level `BEGIN`. Bonus question: why it is there? Does it matter?

It doesn't matter for a compiler, but does if the code is 'load'ed or typed in in an interpreter. Separate top-level statements in most interpreters (REPL) – tried: Bigloo, Scheme48, Petite Chez, Gambit, Gauche – have an implicit prompt around them – which cuts the continuation in call/cc. NB: the cupto paper strongly argues that continuations should have a limited extent – act within one 'prompt'. Here, the technique is based on the fact that the continuations should be truly *unlimited*!

The case of delimited continuations

```
(define (H a b)
  (lambda (on-h) (lambda (on-hv)
      ((on-h a) b))))

(define (HV v)
  (lambda (on-h) (lambda (on-hv)
      (on-hv v))))

(define-syntax greset
  (syntax-rules ()
    ((greset hr e) (hr (reset (HV e))))))

(define-syntax gshift
  (syntax-rules ()
    ((gshift hs f e)
      (shift f* (H (lambda (x) (hs (f* x)))
                   (lambda (f) e))))))
```

As we know, shift and reset **macro**-express everything, from `prompt` and `control` all the way to `shift0` and even `call/cc` if we consider its prompt to be in the infinite future. That macro-expression is a simple one: this pair, depending on the choice of simple functions `hr` and `hs`, gives us everything from `shift` and `reset` to `prompt` and `control` to less and less limited continuations.

The case of delimited continuations

```
(define (hr-stop v)
  ((v
    ; on-h
    (lambda (f) (lambda (x)
        (greset hr-stop (x f)))))
    ; on-hv
    (lambda (v) v)))
(define hs-stop hr-stop)

(define (hr-prop v)
  ((v (lambda (f) (lambda (x)
        (x f))))
    (lambda (v) v)))

(define (hs-prop v)
  ((v
    (lambda (f)
      (lambda (x)
        (shift g
          (H (lambda (y) (hs-prop (g (f y)))) x)))))
    (lambda (v) v)))

(define-syntax prompt
  (syntax-rules ()
    ((prompt e) (greset hr-stop e))))
(define-syntax control
  (syntax-rules ()
    ((control f e) (gshift hs-prop f e))))
```

There are actually two choices for each of the functions `hr` and `hs`. Here's the sample expression for `prompt` and `control`.

Other delimited continuation operators – from shift/reset to prompt0/control0 and cupto – are obtained by varying `hr` and `hs` functions here from these two choices.

As you can see, shift and reset indeed *macro*-express prompt and control. That's all there is to delimited continuations.

Given these simple definitions, we can express the factorial in the following particular simple form, and try it out.

Shift control to factorial

```
(prompt
 (let ()
   (define (call/cc p)
     (shift* (lambda (f)
        (f (p (lambda (x)
          (shift* (lambda (g) (f x))))))))))

   (let ()
    (define fact
      ((lambda (u)
         (u (lambda (x)
              (lambda (n)
                (if (<= n 0) 1
                      (* n ((u x) (- n 1))))))))
       (call/cc (call/cc (call/cc
        (call/cc (call/cc (shift* control*))))))))

      (display (map fact '(5 6 7)))
      (newline))))
```

Here `shift*` and `control*` are procedural versions of shift and control, just like call/cc is. I could have swapped shift and control here.

```
(define (control* p) (control f (p f)))
```

Justification for `(shift* control*)` etc. – straight from the operational – context reduction – semantics of delimited control. First of all, `(shift* control*)` is `(shift f (control* f))` is `(shift f (control g (f g)))`. Let `X`,`Y` in the following be either shift or control, and `reset?` be either `reset` for the shift alternative, or nothing for the control alternative.

```
(reset (C [(X f (Y g (f g)))]))
==> (reset (Y g (f g)) with f x = (reset? C[x]))
==> (reset (f g) with f x = (reset? C[x]) and
                        g x = (reset? [x]))
==> (reset (f g) with f x = (reset? C[x]) and g = id)
==> (reset (reset? C[id]))
==> (reset C[id])
```

as if `(X f (Y g (f g)))` were just `id`.
It is also easy to see that if `X` is shift, then `Y` can even be `shift0` or `control0` – the end result is the same.