CONVERTING TO DATATYPED CPS

Basic Algorithm

- We take any procedure and add a continuation *k* to the parameter list and apply it to the body.
- Example:

```
(define foo
  (lambda (x k) (apply-cont k (+ x 3))))
```

- We then drive in *k* as much as possible.
- There are four rules, we will use.

Four Rules

1. **Simple Expressions:** This includes primitive procedure applications, provided all arguments are simple
2. **Procedure Application with Simple Arguments:** This is a regular procedure application, **not** a primitive procedure application.
3. **Procedure Application with Non-simple Arguments:** It does not matter whether the application is primitive or not.
4. **Special cases (If, Cond)**

Simple Expressions

This includes primitive procedure applications, provided all arguments are simple.

- $C_{simple\text{-}exp}$: If expression *E* is simple, the continuation *k* cannot be driven further through *E,* so we are done processing this *k*, unless we have a lambda-expression
- Examples:

```
(apply-cont k (cons v w))
(apply-cont k 0)
(apply-cont k (lambda (x y) (f x y))) ->
(apply-cont k (lambda (x y k) <drive k further (f x y)>)))
```

- The blue expression may need further processing.

Procedure Application with Simple Arguments

This is a regular procedure application, **not** a primitive procedure application

- $C_{app\text{-}simple\text{-}args}$: If E is a procedure application of the form $(E_1 \ldots E_n)$, where $E_2, \ldots, E_n$ are **simple**, the procedure call is converted by replacing *(apply-cont k E)* with $(E_1 \ldots E_n\ k)$.
- Example:

```
(apply-cont k (g (- n 1)))    ->  (g (- n 1) k)

(apply-cont k (g (lambda (x y) (f x y)))) ->
  (g (lambda (x y k) <drive k further (f x y)>) k))
```

- The blue expression may need further processing.

Procedure Application with Non-simple Arguments

It does not matter whether the application is primitive or not

- C_{app} : If E is a procedure application of the form $(E_1 \ldots E_n)$, where at least one of the E_i is a procedure application A of the form $(A_1 \ldots A_k)$, then replace

  (apply-cont k $(E_1 \ldots (A_1 \ldots A_k) \ldots E_n)$*)* by

  $(A_1 \ldots A_k$ (some-continuation E_2 *…{remove A}* $\ldots E_n\ k)$

  where *some-continuation* is an appropriate continuation that you may have to create.
- If there are any additional procedure applications, they need to be removed and their arguments need to be added to the continuation.
- Example:

```
(apply-cont k (* n m (foo (- n 1) (+ x 42)))) ->
    (foo (- n 1) (+ x 42) (multiply-cont n m k))
```

C_{app} Illustrated – Example 1

1) Grab the offending app

```
(apply-cont k (* n (f (- n 1))))
```

2) Promote the offending app to the level of the expression to be fixed:

```
(f (- n 1))
```

3) Add a continuation because the procedure **f**, once converted to cps needs to have an additional parameter. This continuation should encode the sort of operation we need to complete. In our case, we need to remind us that we need to multiply:

```
(f (- n 1) (multiplication-cont))
```

4) Have this new continuation encapsulate the existing continuation *k*:

```
(f (- n 1) (multiplication-cont k))
```

5) Add all the expressions from the original application except for offending app, i.e. the call to fact. In our case, we move **n**:

```
(f (- n 1) (multiplication-cont n k))
```

C_{app} Illustrated – Example 2

1) Grab one of the offending apps

```
(apply-cont k (* n (f (- n 1)) (g 3)))
```

2) Promote the chosen app to the level of the expression to be fixed:

```
(f (- n 1))
```

3) Add a continuation because the procedure **f**, once converted to cps needs to have an additional parameter. This continuation should encode the sort of operation we need to complete. In our case, we need to remind us that we need to evaluate g and multiply n:

```
(f (- n 1) (g-cont))
```

4) Have this new continuation encapsulate the existing continuation *k*:

```
(f (- n 1) (g-cont k))
```

5) Add all the expressions from the original application except for offending app, i.e. the call to fact. In our case, we move **n**:

```
(f (- n 1) (g-cont n 3 k))
```

Special cases (If, Cond)

- $C_{special}$:Convert **`if`** and **`cond`** expressions as follows, but only if *H* is simple:
- `(apply-cont k (if H` T_1 T_2`)) ->`
  `(if H (apply-cont k` T_1`) (apply-cont k` T_2`))`
- `(apply-cont k (cond [`H_1 T_1`] [`H_2 T_2`]…)) ->`
  `(cond [`H_1 `(apply-cont k` T_1`]`
  `[`H_2 `(apply-cont k` T_2`] …)`
- If *H* is not simple, first convert them with C_{app}
- **`(apply-cont k (if (f x)`** T_1 T_2**`)) ->`**
  `(f x (if-cont` T_1 T_2 **`k))`**
- In this case, many people find it easiest to first convert **`cond-exp`** to nested **`if-exps`** and then convert them.

Flowchart (for processing a procedure call)