Lenguajes de Programación, 2019-1
Nota de clase 13: Paradigma Imperativo IV / Continuaciones*

Favio E. Miranda Perea Lourdes del Carmen González Huesca
Facultad de Ciencias UNAM

12 de noviembre de 2018

1. Introducción

Las continuaciones constituyen una técnica de programación basada en funciones de orden superior, la cual proporciona una manera simple y natural de modificar el flujo de una evaluación en lenguajes funcionales. La habilidad de tomar la continuación actual de un programa y considerarla como un ente material (un valor) proporciona una base para definir numerosos conceptos como excepciones, saltos y corrutinas. La idea básica detrás de las continuaciones es la de considerar a la pila de control de un programa como un valor el cual puede devolverse o pasarse como argumento a otra función. Tal abstracción se conoce como *materialización*[1] de la pila y permite entre otras cosas reemplazar la pila de control actual por otra, efectuándose de esta forma una transferencia no local del control del programa. Por esta razón las continuaciones a veces se comparan con la dañina instrucción `goto` de los lenguajes imperativos. Sin embargo una instrucción `goto` permite saltar a una línea arbitraria del programa lo cual puede causar que se continúe con la evaluación en un estado sin sentido, por ejemplo, las variables en cierta línea podrían no estar asignadas a un valor inicial. En contraste, una continuación permite seguir con una evaluación en un estado que ya se habia utilizado. Si bien esto no es completamente seguro, se evita que las continuaciones sean tan dañinas como las etiquetas `goto`.

Por otro lado también son más poderosas pues una instrucción `goto` sólo puede transferir el control a una expresión léxicamente cercana, debido a como funcionan los compiladores. En cambio una continuación puede representar un cómputo anterior arbitrario y el programador puede causar que dicho cómputo "reviva" en cualquier momento.

Las aplicaciones de esta técnica van desde simples optimizaciones hasta elegantes implementaciones de compiladores e intérpretes mediante la técnica de programación por paso de continuaciones *CPS*[2] iniciada con la implementación del compilador Rabbit para Scheme a mediados de los años setenta por Steele y Sussman.

*Estas notas se basan en los libros de Harper y Mitchell y en material de Gerwin Klein y Frank Pfenning.
[1] En inglés *stack reification*.
[2] *Continuation-Passing-Style*

En conclusión las continuaciones son útiles porque representan el estado de control de un cómputo en un instante de tiempo dado. Usando continuaciones podemos materializar el estado actual del control de un programa, almacenarlo en una estructura de datos y volver a él más tarde. Esto es precisamente lo que se necesita para implementar *hilos*[3], que son programas ejecutados de forma concurrente. El calendarizador de hilos debe ser capaz de guardar un programa para ejecutarlo más tarde, tal vez después de que un evento pendiente se complete o de que otro hilo desocupe la memoria o deje de usar el procesador.

2. Ejemplos

En su forma más simple una continuación es simplemente una función que representa al resto de un programa. Como se ha discutido, a nivel de máquina, una llamada a una función corresponde a la siguiente secuencia de acciones:

1. Guardar suficiente información para poder continuar con la ejecución cuando la función regrese. Esta información usualmente incluye la dirección de la instrucción actual y los contenidos de algunos registros y se almacena en un marco de la pila de control.
2. Saltar al código de la función pasando los argumentos.
3. Restaurar el contexto guardado pasándole el valor calculado a la función.

La información adicional necesaria para seguir con la ejecución del programa después de que la llamada a la función regresa un valor es lo que llamamos *continuación.*

Como ejemplo considérese la siguiente función

$$\mathsf{fun}\ f(x,y) \Rightarrow 2x + 3y + 1/x + 2/y$$

Si se usa una estrategia de evaluación de izquierda a derecha podemos observar lo siguiente:

- Si detenemos la evaluación justo antes de la primera división habremos calculado $2x + 3y$
- Cuando la división $1/x$ se haya completado de evaluar, el proceso de evaluación *continuará* con una suma y otra división.
- De esta manera la continuación de $1/x$ es el resto del cómputo después de $1/x$, es decir, el cómputo pendiente.

Las observaciones anteriores pueden hacerse explícitas como sigue:

```
fun f(x,y) => let antes = 2x + 3y;
                  fun cont(div) => antes + div + 2/y
              in
                cont(1/x)
              end
```

[3] *Threads.*

Ahora podemos preguntarnos qué sucede cuando $x = 0$. Digamos que se tiene una respuesta alternativa dada por $antes/5$. Esta respuesta puede devolverse usando usando excepciones o bien podemos definir una función de división que reciba como argumento adicional a la continuación de la división original y la aplique sólo en el caso en que el divisor x no sea 0, con lo cual obtenemos el siguiente programa

```
fun div(num,den,cont,valt) =>
   if den > 0.001 then
     cont(num/den)
   else
     valt


fun f(x,y) => let antes = 2x+3y;
                  fun cont(coc) =>  antes + coc + 2/y
              in
                  div(1,x,cont,antes/5)
```

Observamos que en este caso la continuación se pasa como un parámetro y la división maneja errores pudiendo terminar de dos formas, con la continuación o con el valor alterno.
Podemos generalizar este ejemplo aun más considerando no solo un valor alterno sino una continuación alterna:

```
fun div(num,den,cont_nor,cont_err) =>
   if den > 0.001 then
     cont_nor(num/den)
   else
     cont_err(...)


fun f(x,y) => let antes = 2x+3y;
                  fun contn(coc)=> antes + coc + 2/y;
                  fun conte(...) => ....
              in
                  div(1,x,contn,conte)
```

De estos ejemplos se ve la utilidad de manejar explícitamente los cómputos pendientes en la evaluación de un programa. Veamos ahora cómo representar estos cómputos mediante funciones que representan la pila de control de la máquina $\mathcal{K}$.

Considérese la evaluación de $(2 + 3) * 6$ en la máquina $\mathcal{K}$:

$$
\begin{array}{rcl}
\Box & \succ & (2+3)*6 \\
\texttt{prod}(-,6);\Box & \succ & 2+3 \\
\texttt{suma}(-,3);\texttt{prod}(-,6);\Box & \succ & 2 \\
\texttt{suma}(-,3);\texttt{prod}(-,6);\Box & \prec & 2 \\
\texttt{suma}(2,-);\texttt{prod}(-,6);\Box & \succ & 3 \\
\texttt{suma}(2,-);\texttt{prod}(-,6);\Box & \prec & 3 \\
\texttt{prod}(-,6);\Box & \prec & 5 \\
\texttt{prod}(5,-);\Box & \succ & 6 \\
\texttt{prod}(5,-);\Box & \prec & 6 \\
\Box & \prec & 30
\end{array}
$$

En cada paso la pila de control puede representarse como una función de la siguiente manera:

1. Cada marco se representa como una función del punto de espera, por ejemplo el marco $\texttt{prod}(-,6)$ corresponde a la función $\lambda v.v*6$.

2. La secuencia de marcos se representa mediante la composición de las funciones correspondientes. Si $f_{\mathcal{P}}$ es la función que representa a la pila $\mathcal{P}$ y f_m representa al marco m entonces la función para la pila $m;\mathcal{P}$ es
$$f = \lambda v.f_{\mathcal{P}}(f_m v)$$

3. La pila vacía se representa como la función identidad $\lambda v.v$ puesto que el estado $\Box \prec v$ indica que el valor v se devolverá sin quedar cómputos pendientes por lo que v es el resultado final del proceso.

Para el ejemplo anterior las representaciones funcionales de las pilas en cada paso son:

$$
\begin{array}{rcl}
\Box & & k_{top} = \lambda v.v \\
\texttt{prod}(-,6);\Box & & k_1 = \lambda v.k_{top}(v*6) \\
\texttt{suma}(-,3);\texttt{prod}(-,6);\Box & & k_2 = \lambda v.k_1(v+3) \\
\texttt{suma}(2,-);\texttt{prod}(-,6);\Box & & k_3 = \lambda v.k_1(2+v) \\
\texttt{prod}(-,6);\Box & & k_1 \\
\texttt{prod}(5,-);\Box & & k_4 = \lambda v.k_{top}(5*v)
\end{array}
$$

Por convención las pilas se representa con la letra k y en particular la pila vacía con k_{top}.
Consideremos ahora una expresión aritmética más complicada y hagamos explícitas las conti-

nuaciones en cada punto ya sin hacer referencia a la pila de control:

Expresión	Continuación
$(6 * 5 + (7 - 3))/2$	$k_{top} = \lambda v.v$
$6 * 5 + (7 - 3)$	$k_1 = \lambda v.k_{top}(v/2)$
$6 * 5$	$k_2 = \lambda v.k_1(v + (7 - 3))$
6	$k_3 = \lambda v.k_2(v * 5)$
5	$k_4 = \lambda v.k_2(6 * v)$
$30 \star$	k_2
$7 - 3$	$k_5 = \lambda v.k_1(30 + v)$
7	$k_6 = \lambda v.k_5(v - 3)$
3	$k_7 = \lambda v.k_5(7 - v)$
$4 \star$	k_5
$34 \star$	k_1
2	$k_8 = \lambda v.k_{top}(34/v)$
$17 \star$	k_{top}

Los pasos anotados con $\star$ denotan a valores intermedios devueltos por eso las continuaciones correspondientes no son nuevas. Por lo general estos pasos pueden omitirse.

Consideremos ahora otro ejemplo esta vez usando declaraciones fun:

$$\mathsf{fun}\ f(x) \Rightarrow (2 + x) * (3 + 4)$$

una vez evaluada la expresión $2 + x$ el resto del programa consiste en multiplicar el valor devuelto por $(3 + 4)$, es decir la continuación es la función

$$\mathsf{fun}\ k(v) \Rightarrow v * (3 + 4)$$

En lugar de declarar la continuación como una función por separado como en los casos anteriores, algunos lenguajes proporcionan un nuevo mecanismo de ligado para la continuación, denotado letcc. En el caso del ejemplo anterior la declaración

$$(\mathsf{letcc}\ k\ \mathsf{in}\ 2 + x\ \mathsf{end}) * (3 + 4)$$

liga la variable k a la continuación de la expresión $2+x$ que en este caso es $\lambda v.v*(3+4)$. De manera mas coloquial, k es un nombre para la continuación de $(2 + x)$ en la expresión $(2 + x) * (3 + 4)$.

Análogamente, la expresión

$$\mathsf{letcc}\ k\ \mathsf{in}\ 1 + 2\ \mathsf{end}$$

liga a k con la continuación de $1 + 2$ en $1 + 2$. Dicha continuación es vacía, despues de devolver el valor $1 + 2$ no hay nada más que hacer por lo que k está ligada a la función identidad $\lambda v.v$ que recibe un valor y lo deja intacto.

Otros ejemplos son:

- En $\mathsf{fst}(\mathsf{letcc}\ k\ \mathsf{in}\ \langle \mathsf{true}, \mathsf{false} \rangle\ \mathsf{end})$, la continuación k está ligada a $\lambda x.\, \mathsf{fst}\, x$
- En $2x + 3y + (\mathsf{letcc}\ k\ \mathsf{in}\ 1/x\ \mathsf{end}) + 2/y$ la continuación k está ligada a $\lambda z.2x + 3y + z + 2/y$

En los ejemplos anteriores, k está ligada a la continuación pero esta nunca se llama explícitamente, es decir, la k no figura nuevamente en la expresión, entonces ¿qué utilidad tiene nombrar a la continuación?

Considérese el siguiente programa en HASKELL para multiplicar una lista de números.

```
mult [] = 1
mult (x:xs) = x*(mult xs)
```

Este programa es ineficiente si en la lista dada figura un cero, en cuyo caso lo más conveniente sería detener todo cómputo posterior al momento en el que apareció el cero y devolver cero como resultado final. Por ejemplo mediante la siguiente adecuación:

```
mult [] = 1
mult (0:xs) = 0
mult (x:xs) = x*(mult xs)
```

Si la lista dada tiene un cero esta versión es mejor, pero es más lenta si no lo hay. Además devuelve un cero tantas veces como elementos antes del primer cero habían en la lista original, por lo que realmente no es óptimo.
Una solución[4] elegante está en el uso de continuaciones, considérese la siguiente versión.

```
mult xs =
 letcc k in
   let
    mult' [] = 1
    mult' (0:ys) = continue k 0
    mult' (y:ys) = y * (mult' ys)
   in
    mult' xs
   end
 end
```

De acuerdo a lo ya explicado la continuación ligada a k es el resto del programa después del `let` sin embargo la expresión `continue k 0` indica que se debe pasar a dicha continuación el valor 0. Es decir, en lugar de seguir evaluando el cuerpo de la declaración `let` nos detenemos de inmediato pasando el valor 0 a la continuación k que en este caso es la identidad y devolviendo el mismo cero como resultado de `mult` de manera que con k hemos ligado un punto de escape para el retorno de la función principal.
Análogamente en `1 + (mult [1,2,0,3])` la continuación ligada a k es $\lambda v.1 + v$ siendo v el valor retornado por la evaluación de `mult`, el cual debido a la instrucción `continue` es 0 y la evaluación devuelve 1 de inmediato. Se observa que la variable ligada por la continuación, en este caso k funciona como una especie de etiqueta *goto* permitiendo una transferencia no-local del control del programa.

Veamos otros ejemplos del uso de los operadores `letcc` y `continue`.

[4]Por supuesto que podrían usarse excepciones.

- En la expresión letcc k in (continue k $(1+2)$) end $*$ $(3+4)$ la continuación es $\lambda v.v * (3+4)$ y la instrucción continue envia el valor de $1+2$ a dicha continuación de manera que el resultado final es 21. En este caso la expresión tiene el mismo significado que la expresión letcc k in $1+2$ end $*$ $(3+4)$.

- En la expresión (letcc k in $1+$ (continue k 2) end) $*$ $(3+4)$ la continuación es nuevamente $\lambda v.v * (3+4)$. Sin embargo el valor final es 14, la operación 1+ ya no se lleva a cabo debido a que el operador continue causa que se escape de ella.

- En la expresión $(1+$letcc k in $2+$continue k 3 end$)*(3+4)$ la continuación es $\lambda v.(1+v)*(3+4)$ pero el resultado es 28 ya que el operador continue escapa de la operación 2+.

- En la expresión fst letcc k in $\langle e,$ continue k $\langle$false$, e'x\rangle\rangle$ end la continuación es $\lambda v.$ fst v y el resultado es false

- En la expresión snd letcc k in $\langle$continue k $\langle e,$ true$\rangle, e'\rangle$ end la continuación es $\lambda v.$ snd v y el valor final es true.

El operador continue también se denota con throw. Nosotros seguiremos usando continue para continuaciones y throw exclusivamente para excepciones.
El lector que ha tenido contacto con las continuaciones probablemente lo tuvo por medio de SCHEME donde el operador correspondiente a letcc se llama `call-with-current-continuation` o `call/cc`, el cual se comporta exactamente igual que letcc sólo que requiere que su único argumento sea una función.

Pasemos ahora a estudiar la sintaxis y semántica formal de las continuaciones.

3. Sintaxis

La extensión se define como sigue:

- La categoría de tipos es:

$$\mathsf{T} ::= \ldots \mid \mathsf{Cont}(\mathsf{T})$$

  aquí Cont(T) es el tipo de continuaciones que esperan un valor de tipo T.

- Las expresiones en sintaxis concreta son:

$$e ::= \ldots \mid \text{letcc } k \text{ in } e \text{ end} \mid \texttt{continue } e_1\ e_2$$

- Las expresiones en sintaxis abstracta son:

$$e ::= \ldots \mid \texttt{letcc}[\mathsf{T}](k.e) \mid \texttt{continue}(e_1, e_2)$$

- La categoria de valores se extiende mediante

$$v ::= \ldots \mid \texttt{cont}(\mathcal{P})$$

donde $\mathcal{P}$ es una pila de control.

la expresión $\texttt{cont}(\mathcal{P})$ representa a una pila materializada, este tipo de expresiones surgen durante la evaluación pero no deben estar disponibles al programador por eso no figuran en la categoria de expresiones.

4. Semántica

La semántica de muchos constructores de control, como excepciones y corrutinas, puede expresarse en términos de pilas de control materializadas, es decir, representando una pila de control como un valor. Esto puede lograrse permitiendo el paso de una pila como valor a un programa la cual puede restaurarse como pila en un punto posterior del programa aún cuando el control haya regresado a un punto posterior a la materialización. Las pilas de control materializadas son lo que hemos llamado continuaciones o continuaciones de primer orden, lo cual indica que son valores con un tiempo de vida indefinido que pueden pasarse y devolverse al gusto en cualquier cómputo. Las continuaciones no expiran nunca de manera que permiten un viaje ilimitado en el tiempo, es decir, podemos regresar a un punto anterior y luego volver a un punto en el futuro del programa. Esto se puede implementar mediante el uso de estructuras de datos persistentes[5], es decir, estructuras tales que una nueva operación sobre ellas genera una nueva versión de la misma sin eliminar la versión vieja.
Por ejemplo las listas en ML son persistentes en el sentido de que si agregamos un nuevo elemento al inicio no destruimos la lista original sino que generamos una nueva lista con un elemento adicional, manteniendo la posibilidad de usar la lista anterior para otros propósitos. Esto es lo que hace Unix internamente para intercambiar el control entre procesos pero puede hacerse al nivel del lenguaje y no del sistema operativo.

En contraste las estructuras de datos en lenguajes imperativos convencionales son efímeras, por ejemplo, la inserción en una lista ligada causa que la lista mute perdiéndose la versión original; agregar algo a una pila modifica el apuntador y escribe en la memoria subyacente, eliminar de una pila escribe el apuntador. En lenguajes funcionales las pilas pueden implementarse de manera que la operación de agregar genera una nueva pila dejando a la anterior intacta.

Para poder implementar las continuaciones de manera segura debemos usar una representación persistente de la pila de control, por ejemplo como una lista ligada de marcos tal que las operaciones de agregar y eliminar generen multiples copias de la pila. Por supuesto para reciclar copias que ya no se usarán más debemos depender de un buen recolector de basura.

4.1. Semántica estática

Para definir la semántica estática debemos permitir un tipado sobre las pilas materializadas similar al utilizado para la seguridad de la máquina $\mathcal{K}$. Decimos que una pila $\mathcal{P}$ tiene tipo T si el marco m en el tope está esperando un valor de tipo T, lo cual denotaremos $\mathcal{P} : \mathsf{T}$. Por ejemplo $(\texttt{suma}(-, e_2); \mathcal{P}) : \mathsf{Nat}$ pues el marco tope $\texttt{suma}(-, e_2)$ está esperando un valor natural.

Las reglas de tipado son:

[5]Una estructura de datos persistente no debe confundirse con una estructura de datos comprometida con un almacenamiento persistente.

- Si $\mathcal{P}$ es una pila de control que espera un valor de tipo T entonces $\texttt{cont}(\mathcal{P})$ es una continuación materializada del tipo $\mathsf{Cont}(\mathsf{T})$, es decir, $\texttt{cont}(\mathcal{P})$ recibe el tipo de continuaciones que esperan valores de tipo T.

$$\frac{\mathcal{P} : \mathsf{T} \quad \mathcal{P}\ \mathsf{pila}}{\Gamma \vdash \texttt{cont}(\mathcal{P}) : \mathsf{Cont}(\mathsf{T})}$$

- Una declaración de continuación `letcc` se comporta estáticamente como sigue:

$$\frac{\Gamma, k : \mathsf{Cont}(\mathsf{T}) \vdash e : \mathsf{T}}{\Gamma \vdash \texttt{letcc}[\mathsf{T}](k.e) : \mathsf{T}}$$

  Si e es de tipo T y k representa a una continuación que espera valores de tipo T entonces k se liga a la continuación de e.

- La expresión $\texttt{continue}(e_1, e_2)$ causa que se suspenda el cómputo actual enviando el valor de e_2 a la pila materializada e_1 y dado que jamás se regresará a la evaluación actual el tipo resultante puede ser cualquiera.

$$\frac{\Gamma \vdash e_1 : \mathsf{Cont}(\mathsf{T}) \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash \texttt{continue}(e_1, e_2) : \mathsf{S}}$$

  Por ejemplo la siguiente expresión se evalua a 3 y la semántica estática debe aceptarla:

$$1 + \mathsf{letcc}\ k\ \mathsf{in}\ (\mathsf{if}\ (\mathsf{continue}\ k\ 2)\ \mathsf{then}\ 3\ \mathsf{else}\ 4)\ \mathsf{end}$$

  de manera que $\mathsf{continue}\ k\ 2$ debe tomar el tipo Bool.
  Más aún, el operador `continue` podría aparecer en distintas partes del mismo código, por ejemplo en la expresión

```
1 + letcc k in
     head ( if b then [(continue k 3) + 1]
             else
              5:(continue k 4) )
    end
```

  k se invoca en dos partes, en la primera se espera un número por lo que $\mathsf{continue}\ k\ 3$ debe ser de tipo Nat, mientras que en la segunda, se espera una lista de números a la que se agregará el 5 por lo que $\mathsf{continue}\ k\ 4$ debe ser de tipo $[\mathsf{Nat}]$.

4.1.1. La negación lógica

Respecto a la correspondencia entre tipos de un lenguaje de programación y conectivos lógicos conocida como correspondencia de Curry-Howard brevemente mencionamos que una posible correspondencia para la negación $\neg\mathsf{T}$ es el tipo $\mathsf{Cont}(\mathsf{T})$. Recordemos que una de las posibles reglas de eliminación de la negación intuicionista es:

$$\frac{\Gamma \vdash \neg\mathsf{T} \quad \Gamma \vdash \mathsf{T}}{\Gamma \vdash \mathsf{S}}$$

la cual corresponde a la regla de tipado del operador `continue`. Esta interpretación de la negación es uno de los tópicos actuales de investigación en lógica computacional.

4.2. Semántica Dinámica

Las continuaciones no pueden modelarse con una semántica operacional estructural puesto que necesitamos un uso explícito de la pila, de manera que lo natural es extender la máquina $\mathcal{K}$. Necesitamos dos nuevas clases de marco:

$$\frac{}{\texttt{continue}(-, e_2)\ \mathsf{marco}} \qquad \frac{}{\texttt{continue}(v_1, -)\ \mathsf{marco}}$$

Las nuevas transiciones son:

- La evaluación de $\texttt{letcc}[\mathsf{T}](k.e)$ bajo la pila $\mathcal{P}$ causa la materialización de la pila en $\texttt{cont}(\mathcal{P})$ la cual se liga a k y se prosigue evaluando e.

$$\frac{}{\mathcal{P} \succ \texttt{letcc}[\mathsf{T}](k.e) \longrightarrow_{\mathcal{K}} \mathcal{P} \succ e[k := \texttt{cont}(\mathcal{P})]}$$

- Una expresión $\texttt{continue}(e_1, e_2)$ pasa el control a la pila materializada e_2 enviándole el valor e_1. Para esto primero se evalua e_1:

$$\frac{}{\mathcal{P} \succ \texttt{continue}(e_1, e_2) \longrightarrow_{\mathcal{K}} \texttt{continue}(-, e_2)\ ; \mathcal{P} \succ e_1}$$

- Una vez devuelto el primer argumento se procede a evaluar el segundo:

$$\frac{}{\texttt{continue}(-, e_2)\ ; \mathcal{P} \prec v_1 \longrightarrow_{\mathcal{K}} \texttt{continue}(v_1, -)\ ; \mathcal{P} \succ e_2}$$

  Obsérvese que en este caso v_1 es necesariamente de la forma $\texttt{cont}(\mathcal{P}')$.

- Devolver un valor v_2 a la pila con tope $\texttt{continue}(\texttt{cont}(\mathcal{P}'), -)$ causa que la pila actual de control $\mathcal{P}$ se abandone y la evaluación continue devolviendo v_2 a la pila $\mathcal{P}'$.

$$\frac{}{\texttt{continue}(\texttt{cont}(\mathcal{P}'), -)\ ; \mathcal{P} \prec v_2 \longrightarrow_{\mathcal{K}} \mathcal{P}' \prec v_2}$$

El sistema con continuaciones cumple las propiedades de seguridad, los detalles se omiten.

Veamos un ejemplo de ejecución en la máquina $\mathcal{K}$ para continuaciones. Según lo visto, la expresión

$$\mathsf{letcc}\ k\ \mathsf{in}\ 1 + \mathsf{continue}\ k\ 2\ \mathsf{end}$$

se evalua a 2, veamos la ejecución en $\mathcal{K}$:

$$
\begin{array}{lll}
 & \Box \succ \texttt{letcc}[\mathsf{Nat}](k.1 + \texttt{continue}(k, 2)) \\
\longrightarrow_{\mathcal{K}} & \Box \succ (1 + \texttt{continue}(k, 2))[k := \texttt{cont}(\Box)] \\
\equiv & \Box \succ 1 + \texttt{continue}(\texttt{cont}(\Box), 2) \\
\longrightarrow_{\mathcal{K}} & \texttt{suma}(-, \texttt{continue}(\texttt{cont}(\Box), 2));\ \Box \succ 1 \\
\longrightarrow_{\mathcal{K}} & \texttt{suma}(-, \texttt{continue}(\texttt{cont}(\Box), 2));\ \Box \prec 1 \\
\longrightarrow_{\mathcal{K}} & \texttt{suma}(1, -);\ \Box \succ \texttt{continue}(\texttt{cont}(\Box), 2) \\
\longrightarrow_{\mathcal{K}} & \texttt{continue}(-, 2);\ \texttt{suma}(1, -);\ \Box \succ \texttt{cont}(\Box) \\
\longrightarrow_{\mathcal{K}} & \texttt{continue}(-, 2);\ \texttt{suma}(1, -);\ \Box \prec \texttt{cont}(\Box) \\
\longrightarrow_{\mathcal{K}} & \texttt{continue}(\texttt{cont}(\Box), -);\ \texttt{suma}(1, -);\ \Box \succ 2 \\
\longrightarrow_{\mathcal{K}} & \texttt{continue}(\texttt{cont}(\Box), -);\ \texttt{suma}(1, -);\ \Box \prec 2 \\
\longrightarrow_{\mathcal{K}} & \Box \prec 2
\end{array}
$$

de manera que efectivamente la expresión se evalua a 2.

5. Programación CPS

El uso de operadores `letcc` y `continue` puede evitarse manteniendo una copia de la pila de control materializada, la cual se representa con una función. De esta forma el control se pasa a la pila mediante una aplicación y la pila se materializa mediante una abstracción lambda. Esto requiere un estilo de programación sistemático y global llamado estilo de paso-de-continuaciones CPS (*Continuation-passing style*). A continuación damos los objetivos e ideas principales detras de esta transformación.

- La transformación CPS es relevante no tanto como un método de programación sino como un método de transformación de programas que permite una optimización en el proceso de compilación.
- Su objetivo es transformar cualquier programa (funcional en nuestra discusión) en un programa que tenga un comportamiento iterativo con respecto al control. La idea del método es colectar y pasar, como argumento adicional a cada función, toda la información de control y datos necesaria para que la ejecución del programa continue después de retornar desde una llamada a dicha función.

- En vez de evaluar una función para después devolver su valor al contexto de evaluación (el marco de control en el tope de la pila) se pasa el contexto de llamada a la función, de modo que esta "sepa" que hacer con su valor. Este argumento adicional es lo que se llama "continuación".
- La transformación CPS puede verse como un proceso de pre-compilación que convierte a los programas a una forma que admite una compilación eficiente. Este proceso es usado por muchos lenguajes importantes como SCHEME o ML.
- La transformación por si sola no asegura que haya un beneficio real en el proceso de compilación puesto que los argumentos adicionales pasados a cada función pueden crecer demasiado creando un cuello de botella. Por esta razón muchos compiladores utilizan adicionalmente optimizaciones de las abstracciones lambda de forma agresiva.
- He aquí un bosquejo de lo que sucede en un proceso de compilación basado en continuaciones:
  - Se realizan el análisis léxico y sintáctico, así como la verificación de tipos
  - Se transforma el programa al cálculo lambda.
  - Se convierte al estilo CPS.
  - Se optimiza este código.
  - Se elimina el mayor número de variables libres.
  - Se eliminan los alcances anidados.
  - Se vuelve a eliminar el mayor número de variables libres.
  - Se genera el programa en lenguaje de bajo nivel (ensamblador).
  - Se genera el código máquina a partir del programa en ensamblador.
- La investigación actual en continuaciones se motiva con aplicaciones en Internet, por ejemplo en sistemas cliente-servidor donde el servidor mantiene una continuación para cada cliente, con el objetivo de continuar con el servicio.

Presentamos ahora una breve introducción al método mediante ejemplos.

- Identidad:

```
id x = x

cpsid x k = k x
```

- Función suma:

```
suma n 0 = n
suma n (s m) = s (suma n m)

cpssuma n 0 k = k n
cpssuma n (s m) k  = cpssuma n m (\ v -> k (s v))
```

Ahora discutimos, de manera más informal y haciendo una analogía con la pila de control, el ejemplo del programa para la multiplicación de listas de números.

```
mult [] = 1
mult (x:xs) = x*(mult xs)
```

Este programa puede transformarse al estilo CPS como sigue:

```
fun cpsmult(ys:[Nat],k:Cont(Nat)):Nat =>
             cpsmult [] k = k 1
             cpsmult (x:xs) k = cpsmult xs (fun (y:Nat) => k (x*y) )
```

Se observa que k es un parámetro adicional que representa a la continuación, en el caso base se pasa el 1 a la continuación k y en el caso recursivo se llama a la misma función con una continuación nueva obtenida al aplicar la continuación anterior al resultado de multiplicar la cabeza de la lista por el valor que se recibirá.

Finalmente el programa original para multiplicar listas en su versión CPS es:

```
mult xs = cpsmult xs (fun (y:Nat) => y)
```

Se observa que la continuación es la función identidad $\lambda y.y$ pues no hay nada que hacer después de calcular `mult xs`.

Con el estilo de programación CPS no necesitamos de los constructores `letcc` y `continue` a costa de tener que definir una continuación adicional para causar un comportamiento distinto, por ejemplo para el manejo de errores. En el caso del programa para multiplicar listas la versión CPS con manejo del cero es:

```
fun cpsmult(ys:[Nat],k0:Cont(Nat),k:Cont(Nat)): Nat =>
             cpsmult [] k0 k = k 1
             cpsmult (0:xs) k0 k = k0 0
             cpsmult (0:xs) k0 k = cpsmult xs k0 (fun (y:Nat) => k (x*y) )
```

Finalmente la versión del programa para multiplicar listas que utiliza `letcc` y `continue` , en su versión CPS es:

```
mult xs = cpsmult xs (fun (y:Nat) => y) (fun (y:Nat) => y)
```

Veamos ahora el ejemplo de la función `map` en listas:

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

La versión CPS es la siguiente:

```
cpsmap cpsf [] k = k []

cpsmap cpsf (x:xs) k =  cpsf x (\ v -> cpsmap cpsf xs (\ vs -> k(v:vs)))
```

Se observa que esta función es la transformación cps no sólo de map sino que también considera que la función f original fue transformada a su forma cps, llamada cpsf. Esta función puede obtenerse observando que la transformación cps de map dejando a f en su forma común es:

```
cpsm f [] k = k []

cpsm f (x:xs) k =  cpsm f xs (\ v -> k (f x :v))
```

Pero considerando a la definición original `map f (x:xs) = f x : (map f xs)` nos percatamos que al calcular `f x` mediante la versión cps de `f`, ésta es la última función a aplicarse puesto que las funciones en forma cps NUNCA regresan al punto de invocación, de forma que es la continuación pasada a cpsf la que debe encargarse de invocar a cpsmap, quedando la ecuación final, a saber

```
cpsmap cpsf (x:xs) k =  cpsf x (\ v -> cpsmap cpsf xs (\ vs -> k(v:vs)))
```

Seguimos con un ejemplo de una función prefijo que toma un predicado y una lista y devuelve el prefijo inicial de la lista que cumple el predicado. Usamos código de HASKELL y en particular el tipo `Maybe` para devolver `Nothing` cuando el prefijo no exista.

```
prefaux p [ ] k = k Nothing
prefaux p (y:ys) k = if (p y)
                        then prefaux p ys (\ vs -> k (y:vs))
                        else k []

pref p xs = prefaux p xs (\ z -> Just z)
```

`pref` es esencialmente la función `takeWhile` de Haskell.

5.1. La transformación CPS

A continuación describimos la transformación CPS para el cálculo lambda sin tipos λ^U. Este proceso convierte un término lambda usual en una expresión del llamado cálculo lambda CPS, denotado λ^{CPS} y definido como sigue:

$$e ::= v_0\, v_1\, \ldots\, v_n \qquad\qquad v ::= x \mid \lambda x_1 \ldots x_n.e$$

Es decir, las expresiones son aplicaciones de valores y los valores son variables o abstracciones lambda de múltiples variables.

La semántica operacional es:

$$(\lambda x_1 \ldots x_n.e)\, v_1 \ldots v_n \to e[x_1, \ldots, x_n := v_1, \ldots, v_n]$$

La traducción del cálculo lambda al cálculo lambda CPS se realiza usando la función $[\![\cdot]\!] : \lambda^U \to \lambda^{\mathsf{CPS}}$ que recibe un término lambda e y devuelve un término CPS $[\![e]\!]$, el cual es una función que toma una

continuación como argumento. Esta función se conoce como transformación de Fischer (o Fischer-Reynolds o Fischer-Reynolds-Plotkin) y su definición es:

$$\begin{aligned} [\![x]\!] &= \lambda k.k\, x \\ [\![\lambda x.e]\!] &= \lambda k.k\, (\lambda k' \lambda x.[\![e]\!]k') \\ [\![e_1\, e_2]\!] &= \lambda k.[\![e_1]\!](\lambda w.[\![e_2]\!](\lambda v.w\, k\, v)) \end{aligned}$$

A continuación damos una idea intuitiva de las definiciones anteriores:

- Variables: se consideran como valores de manera que deben pasarse a la continuación representada con k, obteniendose $\lambda k.k\, x$.
- Abstracciones: como las abstracciones son valores nuevamente deben pasarse a la continuación k, por lo que una primera transformación de $\lambda x.e$ debe ser $\lambda k.k(\lambda x.[\![e]\!])$, sin embargo en el estilo CPS toda función debe tomar a su continuación como un argumento adicional, en este caso $[\![e]\!]$ ya es una función que espera una continuación, así que basta hacer explícito este hecho, digamos con k' por lo que $\lambda x.[\![e]\!]$ debe transformarse en $\lambda k' \lambda x.[\![e]\!]k'$
- Aplicaciones: las versiones CPS de e_1 y e_2 se combinan como sigue para obtener la versión CPS de la aplicación e_1:
  - Primero se evalua e_1 obteniéndose un valor w,
  - después se evalua e_2 obteniéndose un valor v,
  - en este punto se pasa el valor v al valor w para obtener el valor de $e_1\, e_2$. La expresión $[\![e_2]\!]wk$ indica el paso del valor w de e_1 y de la continuación k de $e_1\, e_2$ a $[\![e_2]\!]$. Esto es para cualquier valor w de e_1, por lo que w se liga obteniéndose $\lambda w.[\![e_2]\!]wk$,
  - esta expresión es precisamente la continuación de e_1 en la aplicación, lo cual se expresa con $\lambda k.[\![e_1]\!](\lambda w.[\![e_2]\!]wk)$.
  - Finalmente se hace explícito el paso del valor v de e_2 a wk, con lo que se obtiene la definición dada arriba: $\lambda k.[\![e_1]\!](\lambda w.[\![e_2]\!](\lambda v.wkv))$

La idea de transformar programas al estilo de paso por continuación apareció a mediados de los años sesenta del siglo pasado. La transformación fue parte del folklore en ciencias de la computación por unos años hasta que Fischer y Reynolds dieron una definición formal en 1972. Fischer estudió dos estrategias de implementación para λ^U: una estrategia de retención basada en montículos (heaps) en donde todos los ligados de variables se retienen hasta que ya no se necesiten, y una estrategia de eliminación en donde cada ligado de variable se destruye cuando el control sale del bloque donde dicho ligado fue creado, concluyendo que: no hay una pérdida real de poder computacional al restringirnos a la implementación de la estrategia de eliminación, puesto que cualquier programa puede traducirse en un programa equivalente que funciona correctamente bajo tal implementación.

Por otra parte Reynolds investigó acerca de intérpretes para lenguajes de orden superior. Entre sus metas estaba el deseo de liberar a la definición de un lenguaje de la técnica de paso de parámetros. Él desarrolló un método constructivo pero informal para transformar cualquier intérprete en uno que se vuelva indiferente al método de paso de parámetros, ya sea por nombre o por valor.

Su transformación es esencialmente la transformación de Fischer. Las ideas de Reynolds fueron demostradas formalmente por Plotkin en 1975 en su famoso artículo *Call-by-name, call-by-value, and the λ-calculus*, aparecido en el primer número de la revista *Theoretical Computer Science*. Dicho artículo se ha convertido en uno de los cinco trabajos más relevantes de todos los tiempos en el área de investigación en lenguajes de programación.[6]

A grandes rasgos los resultados más relevantes de este artículo son:

- Simulación : $e \rightarrow^{\star}_{\lambda^U} v$ si y sólo si $[\![e]\!]k \rightarrow^{\star}_{\lambda^{CPS}} [\![v]\!]k$, con k una variable tal que $k \notin FV(e)$
- Indiferencia $eval_v([\![e]\!](\lambda x.x)) = eval_n([\![e]\!](\lambda x.x))$

donde $eval_v, eval_n$ son las funciones de evaluación de las estrategias de llamada-por-valor y llamada-por-nombre respectivamente.

[6] Ver `http://www.cis.upenn.edu/~bcpierce/courses/670Fall04/GreatWorksInPL.shtml`