Lenguajes de Programación 2023-1
Nota de clase 10: Continuaciones
Funcional (λ)

Javier Enríquez Mendoza

29 de noviembre de 2022

Las continuaciones proporcionan al programador una forma de controlar el flujo de un programa. Al tener acceso al contenido de la pila de control en un momento específico de la ejecución del programa. El uso de continuaciones da pie a diferentes optimizaciones sobre programas definidos recursivamente siendo CPS (*Continuation Passing Style*) la mas popular.

En esta nota se definirá el concepto de continuaciones y se agregarán los constructores necesarios para utilizarlas dentro de MinHs . Así como la técnica de CPS y algoritmos de traducción de funciones recursivas.

1 ¿Qué es una continuación?

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.

Una continuación puede pensarse como el **resto** de un cómputo. Esta definición puede no tener mucho sentido así que veamos mejor un ejemplo.

Ejemplos (Continuación)**.** Tomemos como ejemplo la siguiente expresión:

$$5 + (6 * 3) - 12$$

Para evaluar esta expresión el cómputo actual es $(6 * 3)$. La continuación en este punto es el resto del cómputo, es decir, toda la expresión menos el cómputo actual es decir

$$5 + \square - 12$$

En donde $\square$ representa un hueco en el cómputo que se llenará con el resultado del cómputo actual. La continuación ira cambiando a lo largo de la ejecución del programa.
Una vez que $(6 * 3)$ termine de evaluarse, el cómputo será

$$5 + 18 - 12$$

Por lo que el cómputo actual se vuelve $(5 + 18)$ y la continuación es:

$$\square - 12$$

Una vez resuelto el cómputo $(5 + 18)$, la expresión se reduce a

$$23 - 12$$

Y el cómputo actual se vuelve $(23 - 12)$ y la continuación es vacía lo que representa que ya no hay cómputos pendientes después de que el cómputo actual termine su ejecución.

Observación. Podemos notar que las continuaciones como fueron definidas en el ejemplo son muy similares al contenido de nuestra pila de control en las máquinas abstractas definidas en la nota anterior.

Otra forma de pensar en las continuaciones es que proveen un mecanismo de hacer referencia al estado de un programa en un punto específico de la ejecución de este.

Ejemplo 1.1 (Continuaciones como viajes en el tiempo)**.** Tomemos el siguiente ejemplo para entender el comportamiento de una continuación:

Pensemos que estamos en casa cocina frente a la nevera, con ganas de una pizza. En este punto tomamos una continuación y la guardamos en el bolsillo. Después tomamos la masa, el queso y el jamón de la nevera, preparamos la pizza y la colocamos en el horno. Ahora invocamos la continuación que guardamos en nuestro bolsillo, viajamos en el tiempo y nos encontramos de nuevo frente a la nevera con ganas de una pizza. Pero afortunadamente esta vez, ya hay una en el horno y todos los ingredientes para prepararla desaparecieron de la nevera.
Entonces la comemos.

Una continuación no guarda un valor, sino que guarda el contexto en un momento especifico para que podamos volver a ese momento pero ya con el resultado que estamos calculando.

1.1 Continuaciones como funciones

En su forma más simple una continuación es simplemente una función que representa al resto de los cómputos de un programa.

Tomemos como ejemplo la siguiente expresión.

$$(2 + 3) * 6$$

Ahora usemos la máquina $\mathcal{H}$ para evaluar esta expresión:

$$
\begin{array}{rcl}
\blacklozenge & \succ & \texttt{prod}(\texttt{suma}(2,3),6) \\
\texttt{prod}(\square,6);\blacklozenge & \succ & \texttt{suma}(2,3) \\
\texttt{suma}(\square,3);\texttt{prod}(\square,6);\blacklozenge & \succ & 2 \\
\texttt{suma}(\square,3);\texttt{prod}(\square,6);\blacklozenge & \prec & 2 \\
\texttt{suma}(2,\square);\texttt{prod}(\square,6);\blacklozenge & \succ & 3 \\
\texttt{suma}(2,\square);\texttt{prod}(\square,6);\blacklozenge & \prec & 3 \\
\texttt{prod}(\square,6);\blacklozenge & \prec & 5 \\
\texttt{prod}(5,\square);\blacklozenge & \succ & 6 \\
\texttt{prod}(5,\square);\blacklozenge & \prec & 6 \\
\blacklozenge & \prec & 30
\end{array}
$$

En cada paso podemos representar la pila de control como una función de la siguiente manera:

1. Cada marco se representa como una función en el punto de espera, por ejemplo $\texttt{suma}(\square,2)$ es la función $\lambda v.v+2$
2. Una secuencia de marcos se representa como la composición de las funciones que los representan.
3. La pila vacía es representada por la función identidad.

Para el ejemplo anterior tenemos las siguientes funciones que representan las pilas de control de la ejecución.

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

2 Agregando continuaciones a MinHs

Consideremos otro ejemplo del uso de continuaciones con la expresión:

$$(2+x)*(3+4)$$

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:

$$\lambda v.v*(3+4)$$

Para modelar continuaciones en MinHs se podrían declarar como funciones usando el constructor `let`, de la siguiente forma:

$$\texttt{let}\, k = \texttt{lam}\, v \Rightarrow v*(3+4)\, \texttt{in}\, k\ (2+x)\, \texttt{end}$$

En lugar de declarar la continuación como una función por separado como en el caso anterior, algunos lenguajes de programación como Haskell proporcionan un nuevo mecanismo de ligado para la continuación, llamado `letcc`. Con lo que el ejemplo anterior queda como:

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

en donde se está ligando la variable k a la continuación de $(2+x)$ que en este caso es $\lambda v.v*(3+4)$

Observación. `letcc` significa *let current continuation*

En el ejemplo anterior, k está ligada a la continuación pero ésta nunca se llama explícitamente, es decir, k no figura nuevamente en la expresión. Recordemos que la continuación es una función, que materializa el contenido de la pila de control en el punto en el que fue declarada mediante `letcc`. Para invocar la continuación se usa el operador `continue` que nos permite llamar a la continuación k con un parámetro.

La pregunta que surge ahora es: ¿Qué uso tiene el invocar una continuación? Veamos un ejemplo en el que las continuaciones nos ayudan en la mejora del rendimiento de un programa.

Ejemplo 2.1 (Ejemplo de uso de continuaciones)**.** Para ejemplificar el uso de continuaciones consideremos el siguiente programa en Haskell que multiplica todos los elementos de una lista de números

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

Si en la lista de entrada para esta función hay alguna aparición del número 0, la función es muy ineficiente, pues calcula todas las multiplicaciones hasta llegar a la lista vacía y al tener un 0 el resultado será 0 sin importar el resto de los elementos. Para mejorar la eficiencia en este caso, se puede reescribir la función como:

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

De esta forma si aparece 0 en la lista, esta función regresa directamente 0, sin embargo aún se tienen que resolver las multiplicaciones que se fueron generando por las llamadas recursivas, para poder obtener finalmente 0 como resultado. Y si el 0 fuera el último elemento de la lista la nueva función es igual de ineficiente que la anterior.
Una solución para esto es el uso de continuaciones de la siguiente forma:

```
mult :: [Int] -> Int
mult xs = letcc k in mult' xs
    where
        mult' []     = 1
        mult' (0:xs) = continue k 0
        mult' (x:xs) = x * (mult xs)
  end
```

en donde k es el resto de los cómputos después de la llamada a `mult'` y su definición con `where`, y el operador `continue` indica que se debe pasar a dicha continuación el valor 0 en lugar de continuar con la evaluación de `mult'`, es decir, la evaluación se detiene de inmediato y regresa 0 como resultado de la función `mult`. De esta forma la continuación k es el punto de escape para regresar el valor 0 a la función principal.

Análogamente en la expresión `1+(mult [3,2,1,0])` la continuación k sería $\lambda v{,}1 + v$ en donde v es el valor que regresa la función `mult` y al momento de ejecutar `continue` k 0 se pasa 0 directo a $\lambda v{,}1 + v$ regresando 1 como resultado.

En resumen, los operadores `letcc` y `continue` se utilizan para el manejo de continuaciones en lenguajes de programación. Así que los agregaremos a MinHs .

2.1 Sintaxis

Definición 2.1 (Sintaxis para continuaciones)**.** Se extiende la sintaxis de MinHs para agregar continuaciones al lenguaje como sigue:

Tipos

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

en donde $\mathsf{Cont}(\mathsf{T})$ es el tipo para continuaciones que esperan un valor de tipo T.

Expresiones en sintaxis concreta

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

Expresiones en sintaxis abstracta

$$a ::= \cdots \mid \texttt{letcc}[\mathsf{T}](k.a) \mid \texttt{continue}(a_1, a_2)$$

Valores del lenguaje

$$v ::= \cdots \mid \mathsf{cont}(\mathcal{P})$$

en donde $\mathcal{P}$ es una pila de control. Y la expresion $\mathsf{cont}(\mathcal{P})$ representa la pila $\mathcal{P}$ materializada como una función. Al igual que el operador `fix` este tipo de expresiones surgen en la evaluación pero no deben estar disponibles para la persona que desarrolla en el lenguaje.

2.2 Semántica Estática

Definición 2.2 (Semántica estática para continuaciones)**.** Para definir la semántica estática debemos permitir un tipado sobre las pilas materializadas. Decimos que una pila $\mathcal{P}$ tiene tipo T si el marco en el tope de ésta espera un valor de tipo T. Por ejemplo, se cumple que:

$$\texttt{if}(\square, e_2, e_3); \mathcal{P} : \mathsf{Bool}$$

Ya que el marco $\texttt{if}(\square, e_2, e_3)$ espera un valor booleano. Con esto en mente definimos las reglas de tipado como sigue:

Pila materializada

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

Declaración de continuaciones

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

Invocación de continuaciones

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

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 nunca se regresará a la evaluación actual el tipo resultante puede ser cualquiera.

2.3 Semántica Dinámica

La semántica operacional para continuaciones no puede modelarse con un sistema de transición puesto que se necesita el uso explícito de la pila de control. Por simplicidad, se utilizará la máquina $\mathcal{H}$ para modelarla, esto con el fin de no tener que lidiar con pilas en donde se encuentren ambientes. Sin embargo quien lee esta nota debe convencerse de que también se puede usar la máquina $\mathcal{J}$ para la definición de la semántica operacional para continuaciones.

Definición 2.3 (Semántica dinámica para continuaciones)**.** Extendemos la semántica dinámica para MinHs definida con la máquina $\mathcal{H}$ para tratar con los nuevos constructores para continuaciones de la siguiente forma:

Marcos

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

Transiciones

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

Evaluar un `letcc` causa la materialización de la pila actual, que se liga a k y se prosigue con la evaluación de e.

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

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

$$\frac{}{\texttt{continue}(\mathsf{cont}(\mathcal{P}_c), \square); \mathcal{P} \prec v_2 \longrightarrow_{\mathcal{H}} \mathcal{P}_c \prec v_2}$$

Para el operador `continue` primero se reducen ambas expresiones a valores, y si regresamos un valor a un `continue` con una pila $\mathcal{P}_c$ se abandona la pila actual y se restaura $\mathcal{P}_c$

Referencias

[1] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.

[2] Ramírez Pulido K., Soto Romero M., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-2

[3] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.

[4] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.

[5] Mitchell J., Foundations for Programming Languages. MIT Press 1996.