Lenguajes de Programación, 2022-1
Nota de clase 10: Paso de Parámetros

Karla Ramírez Pulido Francisco Escalona González Manuel Soto Romero

29 de noviembre de 2021
Facultad de Ciencias UNAM

10.1. Introducción

Ahora, analizaremos a fondo la manera en que se asignan los parámetros reales a los parámetros formales en la aplicación de funciones, es decir, veremos que hay distintas maneras en que podemos asociar los identificadores y las expresiones de un lenguaje, y que estos cambios tienen consecuencias importantes en la forma en que son evaluados los programas.

Retomemos en concepto de *estado* que vimos anteriormente: en un lenguaje con estado existe alguna forma de definir *variables,* que son localidades de memoria que almacenan algún valor, y nosotros podemos cambiar dicho valor. Las variables entonces, a diferencia de los identificadores que teníamos al principio, pueden mutar su valor. El mecanismo que usamos para introducir estado en nuestro intérprete fue la adición un nuevo tipo de valor al que llamamos *caja*, y que representa nuestra intensión de que podemos cambiar lo que hay en la caja, sin deshacernos de la misma. Esto es, creamos una diferencia explícita entre identificadores y variables, y para trabajar con las variables debemos hacer uso de una interfaz especial. Sin embargo, internamente una caja es sólo la dirección de memoria donde está almacenado el valor.

En un lenguaje en el que el estado es importante, estar realizando estas operaciones explícitamente resulta tedioso. Daremos entonces una generalización de las cajas que resulta muy útil: dejamos de lado los identificadores y los reemplazamos todos por variables, así mismo, la definición explícita de cajas y las operaciones de *encajar* (crear una nueva caja) y *desencajar* (sacar el valor almacenado en una caja) serán realizadas de forma automática e implícita por el intérprete. Por otro lado, mantenemos la expresión `seqn` para realizar operaciones en secuencia, y cambiamos `setbox` por comodidad. [1]

Por ejemplo, en la siguiente expresión

```
{with {{x 3}}
    {seqn {set x 7}
          x}}
```

la primera linea en el intérprete crea automáticamente una nueva caja, en la que se introduce el valor 3, y luego asigna dicha caja a la variable `x`. El último paso de la evaluación consiste en regresar el valor actualizado de la variable, para lo cual se desencaja automáticamente el valor de `x`. Por supuesto, el resultado de toda la evaluación es 7.

Finalmente, terminamos de extender nuestro lenguaje con tres nuevos constructores que, extrañamente, dan la impresión de que estamos volviendo a usar cajas explícitamente, aunque en realidad lo que estamos haciendo es sutilmente diferente.

[1] Esta forma de manejar el estado se parece, por ejemplo, a la que usan SCHEME o JAVA (en este último caso hay otros cambios, por ejemplo, la secuenciación es implícita también, y la asignación se realiza, para confusión de todos aquellos que practican la lógica, con el símbolo =.

- El constructor `{ref a}` regresa una referencia a la variable `a`. Esta instrucción es similar a nuestro antiguo `box`, excepto por dos cosas: *solamente lo podemos aplicar a variables,* y mientras que `box` crea una caja, `ref` genera una nueva referencia a una caja definida anteriormente.
- Para sacar el contenido de una referencia, usamos la expresión `{deref r}`, donde `r` es obtenida mediante la operación `ref` mencionada antes. Corresponde con el antiguo `unbox`.
- Para alterar el contenido de una referencia, introducimos `setref` que es el análogo a `setbox`, y su estructura sintáctica es similar: `{setref r v}` cambia el valor en la dirección de memoria a la que apunta `r`, por `v`. Otra vez, la referencia `r` debe haber sido obtenida mediante `ref`.

Como ejemplo de todo esto, tenemos

```
{with {{a 10}}
    {with {{x {ref a}}
           {y {ref a}}}
        {seqn {setref y 1729}
              {deref x}}}}
```

que se evalúa a 1729. Si hubiéramos escrito este mismo código usando la sintaxis para cajas, en el viejo lenguaje con el que empezamos esta sección, tendíamos

```
{with {{a 10}}
    {with {{x {box a}}
           {y {box a}}}
       {seqn {setbox y 1729}
             {unbox x}}}}
```

que da como resultado 10, pues el cambio a la caja `y` no afecta el valor contenido en la caja `x`. Esto debe hacer más palpable la diferencia entre las cajas que teníamos anteriormente y las nuevas referencias. Ciertamente, no hemos dado aún una razón para introducir todas estas operaciones de manejo de referencias, pero lo haremos más adelante, cuando usemos las referencias para explicar los nuevos conceptos de paso de parámetros.

Un observador astuto, preguntaría cuál es el valor que debe tener la variable `a` en la primera versión del programa anterior (la que usa referencias), es decir, si al final, en lugar de indagar el valor de `x` regresáramos el valor de `a`, ¿el intérprete debería regresar 10? ¿o 1729?. La respuesta a esta pregunta es que *queremos* que regrese el valor modificado, es decir 1729, ya que este es el objetivo del operador `ref` y es lo que marca la diferencia con las cajas.

En una analogía con la vida diaria podemos pensar a las referencias y variables (cajas) como nuestro nombre y los posibles apodos que tenemos: una misma persona tiene un único nombre y múltiples apodos que hacen *referencia* a la misma persona.

Observación 10.1. Hay que tener cuidado al añadir las referencias al intérprete. ¿Qué pasaría si pedimos la referencia a una referencia? Por ejemplo en la expresión

```
{with {{a 5}}
    {with {{r {ref a}}}
        {with {{p {ref r}}}
            ...}}}
```

tenemos que p es una referencia, pero ¿a qué valor apunta? Pudiera ser que apunte al valor de a que es un número, o bien que apunte al valor de r, *que es a su vez una referencia*. Esta última conducta es quizás menos deseada y un lenguaje de alto nivel debería optar por la primera opción.

Sin embargo, hay que hacer notar que hay lenguajes poderosos que usan la segunda opción, por ejemplo el lenguaje C, en el cuál es común hacer cadenitas de referencias que apuntan a referencias que apuntan a referencias ... y esto viene acompañado generalmente de operadores (aritméticos, relacionales, etcétera) que trabajan sobre las referencias, pues de otro modo no sería muy útil.

En un lenguaje como el nuestro que no provee este tipo de operaciones sobre las referencias es conveniente tratar de evitar las cadenas de referencias, para lo cual hay que añadir una restricción al evaluar `ref`: si la variable contiene una referencia, nos la saltamos y apuntamos directamente al valor al que apunta dicha referencia, y si contiene cualquier otro valor, apuntamos a ese valor.

10.2. Valores y Referencias

Con el lenguaje modificado que definimos en la sección anterior, vamos a explorar ahora sí, los distintos tipos de paso de parámetros. Hasta ahora en todos nuestros intérpretes (incluyendo el descrito en la introducción) hemos estado usando el ***paso por valor***, que consiste en asignar una copia del operando al parámetro formal correspondiente. Estas dos copias son iguales en un principio, pero alterar una no afecta a la otra en absoluto. Considérese la siguiente expresión:

```
{with {{a 3}
       {f {fun {x} {set x 4}}}}
    {seqn {f a}
          a}}
```

La función f, altera el valor al que apunta su parámetro formal x, pero como éste es sólo una copia del valor al que apunta la variable a, el resultado de todo el programa es 3 ya que el cambio a x no afecta a a. Esta es una regla general que se aplica a todos los programas escritos en un lenguaje que usa paso por valor: cuando una función cambia el valor de alguno de sus parámetros formales, este cambio no afecta a los parámetros reales. Puesto de otra manera, un cambio dentro de una función *a cualquiera de sus parámetros* no puede ser visto por la expresión que hizo la llamada a la función.

Regresando a nuestro intérprete, y en particular, al ejemplo anterior, el paso de parámetros por valor es como si a y x fueran dos cajas distintas con el mismo valor, por eso cambiar el valor dentro de una no afecta lo que hay dentro de la otra.

Es importante recalcar que si en cambio escribiéramos

```
{with {{a 3}
       {f {fun {x} {setref x 4}}}}
    {seqn {f {ref a}}
          a}}
```

entonces aunque estamos haciendo paso por valor, el resultado de todo el programa es 4. La razón de esto es que el valor asociado a x *no es* 3 sino una referencia a este número, es decir, lo que el intérprete copia es la referencia[2],

[2]Recordemos que cuando introdujimos las cajas hicimos que éstas fueran también valores que regresaba el intérprete, entonces una referencia explícita de este tipo es un valor y por tanto es copiada como cualquier otro.

pero ambas copias apuntan a la misma dirección de memoria, luego cambiar el valor en esta dirección afecta a ambas por igual.

Sin embargo, a veces es conveniente que la función y la expresión que la llama no estén aisladas de esta manera. Para eso hay otro tipo de paso de parámetros al que llamamos ***paso por referencia***. Con este tipo de paso de parámetros lo que ocurre es que el proceso que vimos en el ejemplo anterior de copiar la referencia en vez del valor, se hace automáticamente en el intérprete, y de forma implícita. Por supuesto si el parámetro real es un valor, no hay referencia que copiar y por lo tanto lo que copiamos es el valor mismo. Para comprender mejor esto, analicemos el siguiente ejemplo:

```
{with {{c 10}                          {with {{c 10}
       {f {fun {a b} {+ a b}}}}                {f {fun {a b} {+ {deref a} b}}}}
   {f c 5}}                               {f {ref c} 5}}
```

Ambas versiones del programa son equivalentes, siempre y cuando la primera versión sea evaluada con paso por referencia, y la segunda con paso por valor. El intérprete de paso por referencia, evaluará la versión de la izquierda exactamente de la misma forma que el intérprete de paso por valor evaluará la versión de la derecha. Aquí se aprecia claramente lo que mencionamos antes: si el parámetro real es una variable, lo que se copia en el parámetro formal es una referencia a la dirección de memoria donde está el valor de la variable (en el caso del ejemplo de la izquierda, `c` es una variable, por lo tanto en `a` se guarda una referencia al valor de `c`). Si en cambio el parámetro real es *cualquier otro valor*, entonces el parámetro formal se convierte en una referencia directa al valor (otra vez, en el ejemplo de la izquierda podemos tratar a `b` como una variable que contiene el valor `5`).

¿Qué pasaría si evaluáramos la versión de la derecha del ejemplo anterior en un lenguaje con paso por referencia? ¿Cómo manejaría el intérprete a los parámetros formales y a los parámetros reales en la aplicación de función `f`? ¿Se podría evaluar el programa? ¿Por qué?

Regresando al primer ejemplo de esta sección pero ahora evaluando en paso por referencia, cambiar el valor de `x` por `4` tendría el efecto de que `a` cambie también su valor, por lo tanto el resultado de todo el programa sería `4`.

Este tipo de paso de parámetros puede ser muy útil, por ejemplo cuando necesitamos hacer una función que regrese más de un valor: la función puede regresar un valor de la manera usual, y el resto asignarlos a algunos de sus parámetros que fueron pasados por referencia. Otro ejemplo es la siguiente función que intercambia el valor de dos variables.

```
{with {{a 3}
       {b 4}
       {swap {fun {x y}
                  {with {{temp x}}
                     {seqn {set x y}
                           {set y temp}}}}}}
    {seqn {swap a b}
          {- a b}}}
```

Con paso por referencia, las variables `a` y `b` intercambian su valor, y el resultado de toda la expresión es `1`. Si en cambio, evaluamos la expresión con paso por valor, el intercambio realizado dentro de la función no puede ser visto fuera de ésta, por lo tanto `a` y `b` mantienen sus valores iniciales y el resultado de todo es `-1`.

¿Qué pasaría si cambiamos la expresión `{swap a b}` en el ejemplo anterior por la nueva `{swap 10 b}` y evaluamos en el intérprete por paso por referencia?

10.3. Observaciones y Variantes

Existe una situación que se presenta frecuentemente cuando se utiliza el paso por referencia. Cosidérese por ejemplo, el siguiente programa.

```
{with {{b 3}
       {f {fun {x y}
               {seqn {set x 4}
                     y}}}}
    {+ {f b b} b}}
```

Si esto es evaluado en el intérprete con paso por referencia, el cambio a `x` afecta a su vez al valor de `b`, pero también afecta al valor de `y` (las tres variables apuntan a la misma dirección de memoria), y entonces el resultado de toda la evaluación es 8. Esta situación es conocida como *renombrado de variables*[3], y en general hace bastante difícil la tarea de entender un programa. De manera natural, al pensar en lo que hace el programa no esperamos que el hecho de asignar un nuevo valor a una variable afecte el valor de otras, luego casi todas las reglas de razonamiento formal que podemos tener sobre el comportamiento de los programas quedan invalidadas.

Ahora vamos a introducir una variante del paso de parámetros por referencia a la que llamaremos ***paso por referencia-regreso***. Con este método, los parámetros reales son pasados a los parámetros formales como en el paso por valor y el cuerpo de la función es evaluado normalmente, es decir que se hace una copia de cada valor y por lo tanto un cambio a cualquier parámetro formal no se refleja fuera de la función. Sin embargo, *cuando la función termina su evaluación*, el valor de las variables que hayan usado como operandos en la aplicación de la función se actualiza con el valor que haya terminado el parámetro formal correspondiente. Para comprender esto, conviene analizar un ejemplo:

```
{with {{b 2}
       {f {fun {x y}
               {seqn {set x 4}
                     y}}}}
    {+ {f b} b}}
```

Con paso por referencia-regreso, el cambio a la variable `x` en el cuerpo de la función no afecta a `b` mientras se evalúa el cuerpo de la misma, por lo tanto la expresión `{f b}` del final se reduce al valor 6. Sin embargo, en el momento en que termina esta evaluación el valor de la `b` se actualiza a `4`, que es el valor final de `x`. Entonces el resultado de todo el programa es 10. El efecto es que cuando se termina de evaluar la función tenemos un comportamiento como en el caso de paso por referencia, pero mientras se evalúa la función se trabaja como el paso por valor.

¿Cuál es el resultado de evaluar el programa anterior usando paso por valor, y cuál es usando paso por referencia?

Si ahora evaluamos el primer programa de esta sección usando paso por referencia-regreso, nos encontramos con un problema: ¿con qué valor termina `b`? En otras palabras, ¿cuál es el resultado de la evaluación, 7 o 6? La respuesta depende de qué parámetro formal escoja el intérprete para cambiar el valor de `b`, si `x` o `y`. Esta elección es una cuestión de diseño y debe ser especificada en un lenguaje que implemente este paso de parámetros.

[3] En inglés, *variable aliasing*.

10.4. Intérpretes perezosos

Para terminar, nos falta estudiar un tipo de paso de parámetros que realmente no tiene mucho que ver con los estudiados anteriormente. De hecho, ya lo hemos visto antes, pero ahora lo juntaremos con los nuevos conceptos. Se trata de cómo se maneja el paso de parámetros en los lenguajes con evaluación perezosa.

Recordemos que introdujimos este concepto porque en algunos casos había argumentos en una función que no utilizábamos en el cuerpo de la misma, y realizar los cálculos necesarios para reducirlos a valores resultaba innecesarios. Entonces, en vez de ligar el valor de la expresión con el identificador, se liga la expresión completa y sólo cuando es imposible continuar sin el valor, evaluamos la expresión.

Para esto, desarrollamos dos políticas distintas: en la primera, cada vez que nos encontramos una variable en una posición donde teneos que reducirla a un valor, evaluamos la expresión completa. A esto se le llama ***paso por nombre***.

En la segunda tratamos de evitar tantas evaluaciones repetidas, y entonces sólo la primera vez evaluamos la expresión, las subsecuentes utilizamos el valor obtenido previamente. Este tipo de paso de parámetros se llama ***paso por necesidad***.

En un lenguaje en el que no hay efectos secundarios ambos métodos generan los mismos resultados, sin embargo, en presencia de efectos secundarios, pueden ser muy distintos. Por ejemplo en

```
{with {{count 0}
       {g {fun {}
               {seqn {set count {+ 1 count}}
                     count}}}}
   {{fun {x} {+ x x}} {g}}}
```

La función g funciona como un contador del número de veces que ha sido llamada. En el intérprete con paso por nombre, cada llamada a la variable x invoca a la función, la primera vez regresa 1, y la segunda regresa 2, por lo tanto todo el programa se reduce a 3. Si en cambio evaluamos esto usando paso por necesidad, sólo la primera aparición de x invoca a g, que regresa 1, y la segunda hace referencia al mismo valor, entonces el programa tiene como resultado 2.

En ausencia de efectos secundarios, un lenguaje con paso de parámetros perezoso hace que sea sencillo entender los programas, basta con sustituir cada aparición de los parámetros formales en una función por la expresión completa a la que están ligadas, y luego evaluar sin tantos identificadores. A esto se le conoce como la *regla de copiado* y refleja bastante bien lo que ocurre si tenemos paso por nombre.

¿Se podría diseñar un lenguaje que utilice más de uno de los cinco tipo de paso de parámetros que estudiamos? Por ejemplo, ¿hay algún lenguaje ya existente que utilice dos o tres de ellos? ¿Qué tal uno que implemente los cinco? ¿Qué tan útil sería un lenguaje así?

Problema 10.1. Una buena manera de comprender los distintos tipos de paso de parámetros que vimos es escribiendo programas que ejemplifiquen las diferencias entre ellos, por ejemplo, que regresen distintos resultados si son evaluados con los distintos métodos. En la tabla siguiente se listan todos los tipos de paso de parámetros que vimos.

Paso por Valor	Paso de parámetros Glotón
Paso por Referencia	Paso por Nombre
Paso por Referencia-Regreso	Paso por Necesidad

Añadimos el paso de parámetros glotón sólo para facilitar el enunciado del problema. Podemos pensar que un lenguaje hace una elección en la primera columna y otra en la segunda, por ejemplo, hay lenguajes glotones con paso de parámetros por valor (como nuestro intérprete hasta antes de este tema). Hay nueve de estas combinaciones, y por lo tanto se pueden hacer 36 programas que regresen distintos resultados para cada pareja de combinaciones. Ya hemos hecho programas que ejemplifican las diferencias entre algunas de estas combinaciones, haz otros para otras parejas de combinaciones.

Se puede hacer este buen ejercicios mental, por ejemplo: crea un programa que regrese 5 valores distintos dependiendo de la combinación que se utilice para evaluarlo, paso glotón por valor, glotón por referencia-regreso, por valor y por nombre, y finalmente por valor y por necesidad.

Para la última prueba, asegúrate de tener mucho tiempo libre que matar, y no la intentes si no has resuelto las anteriores: haz un programa con nuestra sintaxis que genere 36 resultados diferentes, según sea evaluado en cada una de las combinaciones posibles.

10.5. Ejercicios

1. Evalúa las siguientes expresiones usando:

   - Paso de parámetros por valor
   - Paso de parámetros por referencia
   - Paso de parámetros por nombre
   - Paso de parámetros por referencia

   (a)
```
{with {{a 1}
          {b -1}
          {swap {fun {x y}
                     {with {{tmp x}}
                        {seqn {set x y}
                              {set y tmp}}}}}}
       {seqn {swap a b}
             {- a b}}}
```

   (b)
```
{with {{acc 0}
          {f {fun {p} {seqn {set acc {+ acc p}} acc}}}
          {to2 {fun {x} {* x x}}}}
     {to2 {f 10}}}
```

2. Evalúa el siguiente código usando paso de parámetros por referencia-regreso.

```
{with {{x 1}
       {y 2}
       {f {fun {x y z} {seqn {set x {* z 2}
                             {set y {* z y}} y}}}}
   {+ {f x y x} {* x y}}}
```

Referencias

[1] Hoare Charles, *Procedures and parameters: An Axiomatic approach*, Symposium on Semantics of Alrithmic Languages, Páginas 102- 116, 1971.

[2] Crank Erik, Fellisen Matthias, *Parameter-passing and the lambda calculus*, Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Páginas 233-244, 1991.

[3] Fleury, Ann, *Parameter passing: the rules the students construct,* ACM SIGCSE Bulletin, Volumen 23, Páginas 283 - 286, 1991.