Lenguajes de Programación y sus Paradigmas, 2019-I
Nota de clase 9: Paradigma Imperativo I. Introducción*

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

12 de octubre de 2018

1. Preliminares

Hasta ahora hemos considerado prototipos de lenguajes de programación que incluyen abstracción funcional, tipos básicos y tipos estructurados como records o variantes. Estas características forman el esqueleto de la mayoría de los lenguajes de programación actuales. Nuestros prototipos pertenecen a la clase de lenguajes llamados *puros* porque su modelo de ejecución consiste únicamente en evaluar una expresión hasta hallar su valor. Sin embargo la mayoría de lenguajes de programación reales incluyen diversas características impuras que no pueden ser descritas mediante el marco semántico que hemos desarrollado hasta ahora.

Los modelos de ejecución de lenguajes como ML, C o Java incluyen efectos laterales de control o almacenamiento. Los efectos de control causan una transferencia no local del control del programa, por ejemplo lanzar una excepción, hacer un salto en el programa o el uso de continuaciones. Por otra parte, los efectos de almacenamiento son modificaciones dinámicas a la memoria mutable de un programa, siendo el más importante el concepto de asignación de variables. La mutación del valor de una variable es una característica estándar en la mayoría de lenguajes de programación, aún en lenguajes puros como Haskell existen mecanismos para manejar estados y asignaciones que requieren de conceptos avanzados de programación como las mónadas.

El concepto de estado tiene un lugar relevante en la computación. Si creamos programas que modelen el mundo real, entonces algunos de ellos deben modelar el hecho de que en el mundo real hay eventos que lo modifican. Por ejemplo: un auto consume gasolina mientras está en marcha, de manera que un programa que modele un tanque de gasolina podría usar estados para registrar los cambios en el nivel de combustible. Por otra parte tiene sentido evitar el uso de estados donde sea posible porque dicho uso dificulta el razonamiento formal con programas.

Una vez que un lenguaje tiene entidades mutables o estados, es necesario hacer referencia al programa antes y después de cada mutación, es decir, el razonamiento y análisis formal de un programa depende del estado en que se encuentre. En consecuencia se vuelve mucho más dificil determinar qué hace realmente un programa, pues tal pregunta depende del momento de tiempo en que se formula.

Considérese el siguiente ejemplo:

*Estas notas se basan en los libros de Pierce, Mitchell y Krishnamurti y en material de Ralf Hinze y Eduardo Bonelli.

```
fun f(n:Nat):Nat =>
   flag := not flag;
   if flag then n else 2*n;

begin
 flag := True;
 write ( f 1 );
 write ( f 1 );
 write ( (f 1) + (f 2));
 write ( (f 2) + (f 1));
end
```

En este código figuran elementos de programación que no hemos estudiado hasta ahora, la asignación así como la operación de ejecución secuencial denotada con punto y coma. Ahora bien si nos preguntamos qué hace la parte del bloque de programa anterior podemos darnos cuenta de que escribirá en pantalla en orden los valores $f\,1, f\,1, f\,1 + f\,2, f\,2 + f\,1$ que en este caso corresponden a $2, 1, 4, 5$. De esto se observan dos cosas tal vez inesperadas: f no es una función en el sentido matemático, pues $f\,1$ devuelve dos valores, además la suma deja de cumplir una de sus propiedades más elementales, la conmutatividad, pues $f\,1 + f\,2$ se evalua a 4 y $f\,2 + f\,1$ se evalua a 5. Esto se debe a que la variables `flag` es mutable, es decir, cambia su valor después de cada asignación, por lo que no es posible hablar a secas del valor de una variable sin hacer referencia al estado actual de la memoria. Dado este nuevo obstáculo la semántica de un programa se vuelve más compleja y los programadores deben tener cuidado con los estados y no abusar de su uso. Un uso legítimo de estados es cuando el programa modela una entidad del mundo real que en verdad está cambiando, es decir, se modela una entidad temporal. En contraste muchos usos del concepto de estado, por ejemplo en el contador de un ciclo son superfluos. Un uso que, si no se tiene cuidado, podría ser ilegítimo es cuando se intenta optimizar la evaluación de un programa donde una expresión aparece repetidamente. Por ejemplo, la instrucción

```
z := (2*a*y+b)*(2*a*y+c)
```

requiere de la evaluación de la subexpresión $2 * a * y$ dos veces, por lo que para evitar la doble evaluación podemos sustituir el código por

```
t := 2*a*y;
z := (t+b)*(t+c)
```

En este caso el código es intercambiable y optimiza la evaluación. Sin embargo esto no siempre es posible, por ejemplo el programa

```
y :=  2*a*y+b;
z :=  2*a*y+c;
```

no debe intercambiarse con

```
t :=  2*a*y;
y:= t+b;
z := t+c;
```

puesto que el valor de y ha cambiado en la segunda asignación por lo que el valor de z no será el esperado puesto que t no se volverá a evaluar para actualizar y y evaluar finalmente z.

En conclusión, es importante entender el significado de estado y cómo evaluar programas que usan estados, así como el significado de efectos de control como son las excepciones o continuaciones. Al final de esta parte del curso podremos entender formalmente el significado de programas como el siguiente:

```
type Cuenta = (dep : Nat -> Void, ret : Nat -> Void, baln : Void -> Nat)

exception Insuf Nat

fun creacuenta(n:Nat):Cuenta =>
  let bal = n in
   ( dep = fun (d:Nat) => bal := bal + d,
     ret = fun (r:Nat) =>
        if bal > r then bal := bal - r
          else throw (Insuf r) ,
     baln = fun(_:Void) => bal
   )
  end

let micuenta = creacuenta(100) in
 handle
  micuenta.ret(50);
  micuenta.ret(60);
  micuenta.dep(40);
  micuenta.ret(60);
  write (‘‘Correcto’’)
 with (Insuf r) => writeln(‘‘Fondos insuficientes’’);
                   write(‘‘Retiro inválido:’’);
                   write(show r)
 end
end
```

2. El Paradigma Imperativo

A grandes rasgos podemos definir el paradigma imperativo como:

$$\textit{Paradigma imperativo} = \textit{Paradigma funcional} + \textit{Efectos laterales}$$

Los efectos laterales que veremos son la asignación que es un efecto de almacenamiento modelado mediante *referencias* a la memoria y los mecanismos de manejo de *excepciones* y programación con *continuaciones* que modifican el flujo de control de un programa.

Empezamos describiendo una instrucción sencilla pero tambíen primordial en este paradigma, la ejecución secuencial.

2.1. Ejecución secuencial

Un mecanismo importante en la programación imperativa es la ejecución de instrucciones en secuencia. La notación $e_1; e_2$ indica que se debe ejecutar e_1 y al finalizar proceder con la ejecución de e_2. En los casos de interés la ejecución de e_1 causa un efecto y no devuelve un valor, o lo que es lo mismo el valor que devuelve no tiene interés por lo que se descarta, y se representa con el valor void de tipo unitario Void, este tipo de expresiones se conocen también como *comandos*. La operación de secuencia ; tiene la siguiente semántica:

- Semántica estática:

$$\frac{\Gamma \vdash e_1 : \mathsf{Void} \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash e_1; e_2 : \mathsf{T}}$$

- Semántica dinámica:

$$\frac{e_1 \to e_1'}{e_1; e_2 \to e_1'; e_2} \qquad \frac{}{(); e_2 \to e_2}$$

Extender nuestro prototipo de lenguaje MinHs con la operación de secuencia no es difícil puesto que esta puede introducirse como azúcar sintáctica.

$$e_1; e_2 :=_{def} (\lambda\_ : \mathsf{Void}.e_2)e_1$$

Aquí usamos la variable anónima _ lo cual indica que la variable de la abstracción lambda no figura en el cuerpo e_2. Es decir, se trata de un ligado falso por lo que no importa el nombre de la variable de la abstracción. Este mecanismo es útil en general y lo adoptaremos a partir de este momento.

La operación de secuencia puede generalizarse para ejecutar más de dos comandos como sigue:

$$e_1; e_2; \ldots; e_n =_{def} e_1; \big(e_2; (e_3; \ldots; (e_{n-1}; e_n)\big)$$

donde de acuerdo a la semántica estática, todas las expresiones, excepto tal vez e_n deben ser de tipo Void, es decir, deben ser comandos.

2.2. El enunciado de asignación

Casi todos los lenguajes de programación proporcionan un mecanismo de asignación. En algunos lenguajes como ML los mecanismos para ligar nombres y para la asignación se mantienen separados. Por ejemplo, en ML se puede tener una variable x cuyo valor es 5 (`val x = 5`) o bien una variable y cuyo valor es una *referencia* o apuntador a una celda mutable cuyo contenido actual es 5 (`val y = ref 5`), y tal diferencia es visible por el programador. A la x podemos sumarle, pero no asignarle, otro número. En contraste, podemos asignarle otro valor a la celda referenciada por y, por ejemplo escribiendo $y := 84$, pero no podemos usar y como argumento para una suma. Para poder usar el valor al que y es necesario eliminar la referencia, mediante una operación denotada $!\,y$ la cual recupera el contenido actual de la celda, a saber 5.

En lenguajes imperativos como C y su parentela, JAVA incluido, cada variable se refiere a una celda mutable y la operación de eliminación o recuperación de referencias (!) no se hace explícita. En este caso se habla del valor izquierdo (*l-value*) que representa la dirección de la celda y el valor derecho (*r-value*) que representa el contenido de la celda. Dependiendo del caso particular una variable x se

asocia a alguno de sus dos valores.
Para el estudio formal del concepto de estado conviene mantener esta separación, la cual es deseable también desde la perspectiva del diseño de un lenguaje. El uso explícito de referencias sugiere un estilo casi funcional, lo cual en la práctica hace que los programas sean más fáciles de escribir y mantener.

Terminemos esta nota con un ejemplo de ambos estilos del uso de asignación, En el siguiente programa g es esencialmente una función que multiplica la entrada x por el número de veces que dicha función se ha invocado por lo que el resultado de $g\,2 + g\,3$ debe ser 8.

- Versión implícita

```
let g = let c = 0 in
          λx.c:=c+1; c*x
        end
in
 g 2 + g 3
end
```

- Versión explícita

```
let g = let c = ref 0 in
          λx.c:=!c+1; !c*x
        end
in
 g 2 + g 3
end
```

3. Operaciones con referencias

Para trabajar con referencias tenemos tres operaciones básicas: el alojamiento o referenciación, la recuperación del valor de una celda o derreferencia[1] y la asignación.

- Alojamiento o referenciación. Para crear una referencia se usa el operador ref cuyo efecto es generar una nueva celda y almacenar en ella el valor dado. Por ejemplo ref 5 genera una nueva celda cuyo contenido es 5.
- Recuperación[2] o derreferencia: para obtener el valor actual de una celda se usa el operador de recuperación !; por ejemplo, si r es una expresión que denota a una celda de memoria entonces $!\,r$ causa el efecto de buscar en la memoria dicha celda y devolver su valor actual.
- Asignación: para modificar el valor de una celda se usa la operación de asignación denotada $:=$. Por ejemplo $r := 7$ elimina el valor actual de la celda denotada por r y almacena en su lugar el nuevo valor 7.

[1] Las palabras en inglés son *allocation* y *dereference*. Para no abusar del lenguaje diciendo *alocación* o *derreferencia*, preferimos usar los términos *alojamiento* y *recuperación*.

[2] Kim Bruce en su libro *Foundations of Object-Oriented Languages*, utiliza el nombre *valor* para la operación $!e$. Sin embargo éste podría confundirse con nuestro ya conocido concepto de valores de manera que usaremos siempre el término *recuperación*.

A continuación vamos a discutir una extensión del lenguaje con estos operadores.
La categoría de expresiones se extiende como sigue:

$$e ::= \ldots \mid \mathsf{ref}\, e \mid !\, e \mid e_1 := e_2$$

Antes de pasar a las formalidades veamos algunos ejemplos:

$$\texttt{let}\ x = \mathsf{ref}\, 2\ \texttt{in}\ x := \mathsf{suc}(!x)\,;\, !x\ \texttt{end}$$

La ejecución se realiza así:

- x está asociada a una celda que guarda el valor 2.
- La operación de secuencia primero ejecuta la asignación $x := \mathsf{suc}(!x)$ para lo cual se recupera el valor de la celda x obteniéndose 2, valor al cual se le aplica la función sucesor obteniéndose 3 y finalmente se asigna este nuevo valor a x. Esta asignación es sólo un efecto y no devuelve ningún valor de interés.
- La segunda instrucción de la secuencia es $!x$ que devuelve el valor 3.

Veamos otro ejemplo

$$\texttt{let}\ x = \mathsf{ref}\, 13\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y :=!y + 2\,;\, !y\ \texttt{end})\ \texttt{end}$$

- x se asocia a una celda con valor 13.
- y se asocia a x.
- Al valor de la celda y, dado por $!y$, que es el mismo que el dado por $!x$, se le suman 2 obteniendose el valor final de 15.
- ¿Qué valor se encuentra ahora alojado en x?
- La respuesta correcta es 15. Es decir, la declaración $y = x$ denota exactamente a la misma celda que x. En estas situaciones decimos que x y y son *alias* para la misma celda.

El uso de alias si bien es de gran utilidad, causa que el razonamiento formal acerca de un programa con referencias sea complicado. Como ejemplo considérense los programas

$$e_1 =_{def} x := 1; x :=! y \qquad e_2 =_{def} x :=!y$$

En general ambos programas se comportan de igual manera, con la primera asignación en e_1 irrelevante. Pero hay una excepción si se considera el caso en que x y y sean alias para una misma celda resultando en valores distintos para las expresiones pues e_2 resulta equivalente a $x :=!x$ que claramente no equivale a e_1.

3.1. Recolección de Basura

Ahora que ya hemos descrito las operaciones básicas con referencias podemos preguntarnos ¿por qué no hay una operación para liberar el espacio de una celda mediante el desalojo[3] de su contenido? La respuesta no es sencilla, preferimos pensar en que dicha operación se realiza, al igual que en los lenguajes de programación reales, en tiempo de ejecución.

El sistema de ejecución se encarga del proceso de *recolección de basura*[4] que consiste en recolectar y reutilizar las celdas a las que el programa ya no puede accesar.
Esta elección no es sólo una cuestión de diseño. Es extremadamente difícil conseguir un sistema seguro bajo la presencia de un operador de desalojo. Veamos un ejemplo sencillo para lo cual vamos a suponer la existencia de una función free que recibe una celda de memoria como argumento y desaloja su valor almacenado de tal forma que el siguiente alojamiento de una celda se hace exactamente al mismo sector de memoria. Considérese ahora el siguiente programa:

```
let r = ref 0 in
 let s = r in
   free r;
   let t = ref true in
    t:=false;
    succ (!s)
   end
 end
end
```

La evaluación del programa devolverá el programa bloqueado suc false lo cual viola la seguridad del sistema. Este ejemplo utiliza de manera crucial el mecanismo de alias y puede simularse facilmente en C mediante las funciones `malloc` y `free`.
Dado que nuestro énfasis es en sistemas de tipos seguros el uso del operador free es inaceptable por lo que es adecuado omitirlo, pensando que el recolector de basura se encargará de liberar memoria.

4. Semántica estática provisional

Para definir la semántica estática primero debemos preguntarnos qué tipos devuelven las operaciones recien definidas.

- Alojamiento: esta operación genera una nueva celda con un contenido de cierto tipo. La celda es una nueva clase de entidad por lo que debemos crear un nuevo tipo para élla. Para esto introducimos los tipos referencia, dado un tipo T, el tipo que tiene como elementos a las celdas cuyo contenido es de tipo T será denotado Ref T.
- Recuperación: la operación de recuperación obtiene el valor de una celda, no hay necesidad de un tipo nuevo dado que al crear la celda el tipo de su valor ya existia previamente.
- Asignación: la operación de asignación es una operación impura que causa un efecto, de manera que no tiene sentido tratar de dar un valor no trivial a dicha operación, por lo que su tipo no es de interés y se le asocia el tipo `Void`.

[3] *deallocation*
[4] *garbage collection.*

De estas observaciones surge la siguiente semántica estática:

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{Ref\ T}} \qquad \frac{\Gamma \vdash e : \mathsf{Ref\ T}}{\Gamma \vdash\ !\, e : \mathsf{T}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Ref\ T} \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash e_1 := e_2 : \texttt{Void}}$$

Con estas reglas podemos excluir expresiones como !2 ó 2 := 3.

5. Semántica Dinámica para Referencias

Al intentar formalizar la semántica operacional surgen algunas preguntas:

- ¿Cuáles son los valores para un tipo Ref T?
- ¿Cómo modelar la evaluación de una expresión ref e?
- En general ¿Qué es una referencia?

Tratemos de responder a estas preguntas con una evaluación intuitiva de acuerdo a la semántica operacional del caso funcional.

Considérese nuevamente el siguiente programa p:

```
let g = let c = ref 0 in
         λx.c:=!c+1; !c*x
        end
in
 g 2 + g 3
end
```

La evaluación debería ser como sigue considerando que la expresión $ref\ 0$ ya es un valor:

$$\begin{aligned}
\mathtt{p} \;&=\; (\lambda g.g\,2 + g\,3)\Big((\lambda c.\lambda x.c := !c + 1; !c * x)(ref\ 0)\Big) \\
&\to^\star\; (\lambda g.g\,2 + g\,3)\Big(\lambda x.ref\ 0 := !(ref\ 0) + 1; !(ref\ 0) * x)\Big) \\
&\to^\star\; \Big(\lambda x.ref\ 0 := !(ref\ 0) + 1; !(ref\ 0) * x)\Big)2 + \Big(\lambda x.ref\ 0 := !(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \\
&\to^\star\; \Big(ref\ 0 := !(ref\ 0) + 1; !(ref\ 0) * 2\Big) + \Big(\lambda x.ref\ 0 := !(ref\ 0) + 1; !(ref\ 0) * x)\Big)3
\end{aligned}$$

Hasta aquí la evaluación ha procedido de la manera usual mediante sustitución. El siguiente paso a ejecutar es el enunciado de asignación $ref\ 0 := !(ref\ 0) + 1$ para lo cual necesitamos un modelo de memoria. Podemos decir que para nuestros propósitos una referencia es una abstracción del concepto de memoria, cada celda mutable representa a una porción de la memoria de una máquina la cual se encuentra actualmente en uso.

Intuitivamente la instrucción $ref\ 0$ genera una celda de memoria $(\ell, 0)$, es decir, se genera una celda que guarda al 0, así como un apuntador a dicha celda cuya dirección de memoria es ℓ. Utilizando esta idea la evaluación continuaría de la siguiente manera con un uso explícito de la memoria:

$$\begin{array}{ll} & \Big(ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * 2\Big) + \Big(\lambda x.ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \\ \to^\star & \Big(ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * 2\Big) + \Big(\lambda x.ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \quad \text{con } (\ell, 0) \end{array}$$

La evaluación de $ref\ 0 :=!(ref\ 0) + 1$ debería actualizar la memoria a $(\ell, 1)$ y devolver un valor trivial.

$$\begin{array}{ll} \to^\star & (); !(ref\ 0) * 2 + \Big(\lambda x.ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \quad \text{con } (\ell, 1) \\ \to^\star & !(ref\ 0) * 2 + \Big(\lambda x.ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \quad \text{con } (\ell, 1) \\ \to^\star & 1 * 2 + \Big(\lambda x.ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \quad \text{con } (\ell, 1) \\ \to^\star & 2 + \Big(\lambda x.ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * x)\Big)3 \quad \text{con } (\ell, 1) \\ \to^\star & 2 + \Big(ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * 3)\Big) \quad \text{con } (\ell, 1) \end{array}$$

La evaluación debe continuar ahora identificando nuevamente la expresión $ref\ 0$ con la memoria $(\ell, 1)$ lo cual ahora no es claro puesto que no hay un mecanismo que relacione la expresión con la memoria. Suponiendo que tal mecanismo existe procederiamos de la siguiente forma:

$$\begin{array}{ll} \to^\star & 2 + \Big(ref\ 0 :=!(ref\ 0) + 1; !(ref\ 0) * 3)\Big) \quad \text{con } (\ell, 1) \\ \to^\star & 2 + \Big((); !(ref\ 0) * 3)\Big) \quad \text{con } (\ell, 2) \\ \to^\star & 2 + !(ref\ 0) * 3\Big) \quad \text{con } (\ell, 2) \\ \to^\star & 2 + 2 * 3 \quad \text{con } (\ell, 2) \\ \to^\star & 2 + 6 \quad \text{con } (\ell, 2) \\ \to^\star & 8 \quad \text{con } (\ell, 2) \end{array}$$

En realidad el mecanismo buscado debería existir antes para evaluar el enunciado de asignación $e_1 := e_2$ dado que de acuerdo a la estrategia usual debe evaluarse primero e_1 hasta ser un valor antes de evaluar e_2. En el caso de $ref\ 0 :=!(ref\ 0) + 1$ no podemos proceder con la evaluación en orden de izquierda a derecha sin conocer el mecanismo que asocia $ref\ 0$ con $(\ell, 0)$ para poder hacer la actualización a $(\ell, 1)$.

Esto se resuelve a considerar que las expresiones $ref\ v$ no son valores y deben reducirse a una dirección de memoria ℓ antes de evaluar cualquier expresión que las contenga. De esta manera, en nuestro ejemplo, tenemos $ref\ 0 \to \ell$ con $(\ell, 0)$ y la evaluación entonces procede así:

$$
\begin{array}{lll}
\mathsf{p} & = & (\lambda g.g\,2 + g\,3)\Big((\lambda c.\lambda x.c := !c + 1; !c * x)(ref\ 0)\Big) \\
& \to & (\lambda g.g\,2 + g\,3)\Big((\lambda c.\lambda x.c := !c + 1; !c * x)\ell\Big) \quad \text{con } (\ell, 0) \\
& \to^\star & (\lambda g.g\,2 + g\,3)\Big(\lambda x.\ell := !(ref\,0) + 1; !(\ell) * x)\Big) \quad \text{con } (\ell, 0) \\
& \to^\star & \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)2 + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 0) \\
& \to^\star & \Big(\ell := !(\ell) + 1; !(\ell) * 2\Big) + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 0) \\
& \to^\star & \Big(\ell := 0 + 1; !(\ell) * 2\Big) + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 0) \\
& \to^\star & \Big(\ell := 1; !(\ell) * 2\Big) + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 0) \\
& \to^\star & \Big((); !(\ell) * 2\Big) + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 1) \\
& \to^\star & !(\ell) * 2 + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 1) \\
& \to^\star & 1 * 2 + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 1) \\
& \to^\star & 2 + \Big(\lambda x.\ell := !(\ell) + 1; !(\ell) * x)\Big)3 \quad \text{con } (\ell, 1) \\
& \to^\star & 2 + \Big(\ell := !(\ell) + 1; !(\ell) * 3\Big) \quad \text{con } (\ell, 1) \\
& \to^\star & 2 + \Big(\ell := 1 + 1; !(\ell) * 3\Big) \quad \text{con } (\ell, 1) \\
& \to^\star & 2 + \Big(\ell := 2; !(\ell) * 3\Big) \quad \text{con } (\ell, 1) \\
& \to^\star & 2 + (); !(\ell) * 3 \quad \text{con } (\ell, 2) \\
& \to^\star & 2 + !(\ell) * 3 \quad \text{con } (\ell, 2) \\
& \to^\star & 2 + 2 * 3 \quad \text{con } (\ell, 2) \\
& \to^\star & 2 + 6 \quad \text{con } (\ell, 2) \\
& \to^\star & 8 \quad \text{con } (\ell, 2)
\end{array}
$$