Capítulo 9

Paradigma funcional

Mitchell 4.4.
Van Roy y Haridi. 2004. Concepts, Techniques, and Models of Computer Programming. MIT Press. Capítulo 3: introducción

9.1. Expresiones imperativas vs. expresiones funcionales

En muchos lenguajes de programación, las construcciones básicas son imperativas, del mismo modo que lo sería una oración imperativa en lenguaje natural, como por ejemplo "*Traeme esa manzana.*". Por ejemplo, esta instrucción de asignación:

```
x: = 5
```

es una orden que el programador le da a la computadora para guardar el valor 5 en un lugar determinado.

Los lenguajes de programación también contienen construcciones declarativas, como la declaración de la función

```
function f(int x) { return x+1; }
```

que describe un hecho, de la misma forma que en lenguaje natural podríamos decir "*La tierra es redonda.*". En este ejemplo, lo que describimos es que "*f es una función cuyo valor de retorno es 1 mayor que su argumento.*".

La distinción entre construcciones imperativas y declarativas se basa en que las imperativas cambian un valor y las declarativas declaran un nuevo valor. Por ejemplo, en el siguiente fragmento de programa:

```
{ int x = 1;
  x = x+1;
  { int y = x+1;
    { int x = y+1;
}}}
```

sólo la segunda línea es una operación imperativa, las otras líneas contienen declaraciones de nuevas variables.

Un punto sutil es que la última línea en el código anterior declara una nueva variable con el mismo nombre que el de una variable declarada anteriormente. La forma más sencilla de entender la diferencia entre declarar una nueva variable y cambiar el valor de una variable ya existente es cambiando el nombre de la variable. Las variables ligadas, que no son libres en una expresión (que están definidas dentro del alcance de la expresión) pueden cambiar de nombre sin cambiar el significado de la expresión. En particular, podemos cambiar el nombre de las variables ligadas en el fragmento de programa anterior de la siguiente manera:

```
{ int x = 1;
  x = x+1;
  { int y = x+1;
    { int z = y+1;
}}}
```

(Si hubiera más ocurrencias de x dentro del bloque interior, también les cambiaríamos el nombre a z.) Después de volver a escribir el programa a esta forma equivalente, podemos ver fácilmente que la declaración de una nueva variable z no cambia el valor de cualquier variable ya existente.

La asignación imperativa puede introducir **efectos secundarios** porque puede destruir el valor anterior de una variable, sustituyéndolo por uno nuevo, de forma que éste no esté disponible más adelante. En programación funcional la asignación imperativa se conoce como asignación o actualización destructiva.

Decimos que una operación computacional es declarativa si, cada vez que se invoca con los mismos argumentos, devuelve los mismos resultados independientemente de cualquier otro estado de computación. Una operación declarativa es:

independiente no depende de ningún estado de la ejecución por fuera de sí misma,

sin estado no tiene estados de ejecución internos que sean recordados entre invocaciones, y

determinística siempre produce los mismos resultados para los mismos argumentos y, por extensión, ejecutar la operación no tendrá efectos secundarios.

Una consecuencia muy valiosa de estas propiedades es la conocida como *transparencia referencial*. Una expresión es transparente referencialmente si se puede sustituir por su valor sin cambiar la semántica del programa. Esto hace que todas las componentes declarativas, incluso las más complejas, se puedan usar como valores en un programa, por ejemplo, como argumentos de función, como resultados de función o como partes de estructuras de datos. Esto aporta una gran flexibilidad a los lenguajes funcionales, ya que se pueden tratar de forma homogénea expresiones de estructura muy variable, como por ejemplo en el siguiente programa:

```
HayAlgunExceso xs n = foldr ( filter (>n) ( map
   convertirSistemaMetrico xs) ) False xs
```

Veamos un ejemplo de dos funciones, una referencialmente transparente y otra referencialmente opaca:

```
globalValue = 0;

integer function rq(integer x)
begin
  globalValue = globalValue + 1;
  return x + globalValue;
end

integer function rt(integer x)
begin
  return x + 1;
end
```

La función *rt* es referencialmente transparente, lo cual significa que `rt(x) = rt(y)` si `x = y`. Sin embargo, no podemos decir lo mismo de `rq` porque usa y modifica una variable global. Por ejemplo, si queremos razonar sobre la siguiente aserción:

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x));
```

Podríamos querer simplificarla de la siguiente forma:

```
integer p = rq(x) + rq(y) * (0);
integer p = rq(x) + 0;
integer p = rq(x);
```

Pero esto no es válido para `rq()` porque cada ocurrencia de `rq(x)` se evalúa a un valor distinto. Recordemos que el valor de retorno de `rq` está basado en una variable global que no se pasa como parámetro de la función y que se modifica en cada llamada a `rq`. Esto implica que no podemos asegurar la veracidad de aserciones como $x - x = 0$.

Por lo tanto, la transparencia referencial nos permite razonar sobre nuestro código de forma que podemos construir programas más robustos, podemos encontrar errores que no podríamos haber encontrado mediante testing y podemos encontrar posibles optimizaciones que podemos trasladar al compilador.

9.2. Propiedades valiosas de los lenguajes funcionales

Los lenguajes funcionales, como Lisp o ML, realizan la mayor parte del cómputo mediante expresiones con declaraciones de funciones. Todos los lenguajes funcionales tienen también construcciones imperativas, pero en esos lenguajes se pueden escribir programas muy completos sin usarlas.

En algunos casos se usa el término "lenguaje funcional" para referirse a lenguajes que no tienen expresiones con efectos secundarios o cualquier otra forma de construcción imperativa. Para evitar ambigüedades entre estos y lenguajes como Lisp o ML, llamaremos a estos últimos "lenguajes funcionales puros". Los lenguajes funcionales puros pueden pasar el siguiente test:

> Dentro del alcance de las declaraciones $x_1, ..., x_n$, todas las ocurrencias de una expresión e que contenga sólo las variables $x_1, ..., x_n$ tendrán el mismo valor.

Como consecuencia de esta propiedad, los lenguajes funcionales puros tienen una propiedad muy útil: si la expresión e ocurre en varios lugares dentro de un alcance específico, entonces la expresión sólo necesita evaluarse una vez. Esto permite que el compilador pueda hacer optimizaciones de cálculo y de espacio en memoria (aunque no necesariamente todos los compiladores de todos los lenguajes funcionales lo van a hacer).

Otra propiedad interesante de los programas declarativos es que son **composicionales**. Un programa declarativo consiste de componentes que pueden ser escritos, comprobados, y probados correctos independientemente de otros componentes y de su propia historia pasada (invocaciones previas).

Otra propiedad interesante es que **razonar** sobre programas declarativos es sencillo. Es más fácil razonar sobre programas escritos en el modelo declarativo que sobre programas escritos en modelos más expresivos. Como los programas declarativos sólo pueden calcular valores, se pueden usar técnicas sencillas de razonamiento algebraico y lógico.

En un programa declarativo, la interacción entre componentes se determina únicamente por las entradas y salidas de cada componente. Considere un programa con un componente declarativo. Este componente puede ser mirado en sí mismo, sin tener que entender el resto del programa. El esfuerzo que se necesita para entender el programa completo es la suma de los esfuerzos que se necesitan para entender el componente declarativo y para entender el resto.

Si existiera una interacción más estrecha entre el componente y el resto del programa no se podrían entender independientemente. Tendrían que entenderse juntos, y el esfuerzo requerido sería mucho más grande. Por ejemplo, podría ser (aproximadamente) proporcional al producto de los esfuerzos requeridos para entender cada parte. En un programa con muchos componentes que interactuán estrechamente, esto explota muy rápido, difcultando o haciendo imposible entenderlo. Un ejemplo de interacción estrecha es un programa concurrente con estado compartido.

Pero para algunos problemas, es necesario trabajar con interacciones estrechas. Sin embargo, podemos adoptar por principio la directiva de tratar de hacer código lo más modular posible, con interacciones estrechas sólo cuando sea necesario y no en cualquier caso. Para soportar este principio, el total de componentes declarativos debería ser el mayor número posible.

Estas dos propiedades son importantes tanto para programar en grande como en pequeño. Sería muy agradable si todos los programas pudieran escribirse en el modelo declarativo. Desafortunadamente, este no es el caso. El modelo declarativo encaja bien con ciertas clases de programas y mal con otras.

9.3. Problemas naturalmente no declarativos

La forma más elegante y natural (compacta e intuitiva) de representar algunos problemas es mediante estado explícito. Es el caso, por ejemplo, de las aplicaciones cliente-servidor y de las aplicaciones que muestran video. En general, todo programa que realice algún tipo de Input-Output lo hará de forma más natural mediante estado explícito. También es el caso en el que estamos tratando de modelar algún tipo de comportamiento (por ejemplo, agentes inteligentes, juegos) en el que representamos a una componente de software como una entidad que tiene *memoria*, porque inherentemente la memoria cambia a lo largo del tiempo.

> **Pregunta 1:**
> *Piense algunos ejemplos de problemas en los que el estado debería estar representado en la solución de software de forma explícita.*

También hay tipos de problemas que se pueden programar de forma mucho más eficiente si se hace de forma imperativa. En esos casos podemos llegar a convertir un problema que es **intratable** con programación declarativa en un problema **tratable**. Este es el caso de un programa que realiza modificaciones incrementales de estructuras de datos grandes, e.g., una simulación que modifica grafos grandes, que en general no puede ser compilado eficientemente. Sin embargo, si el estado está guardado en un acumulador y el programa nunca requiere acceder a un estado ya pasado, entonces el acumulador se puede implementar con asignación destructiva (ver Van Roy y Haridi 6.8.4.).

Por esta razón muchos lenguajes funcionales proveen algún tipo de construcción lingüística para poder expresar instrucciones imperativas. Incluso en lenguajes funcionales puros existen estas construcciones, que en general se conocen como mónadas. Las mónadas son una construcción de un lenguaje que permite crear un alcance aislado del resto del programa. Dentro de ese alcance, se permiten ciertas operaciones con efectos secundarios, por ejemplo, el uso de variables globales (globales al alcance) o asignación destructiva. Está garantizado que estos efectos secundarios no van a afectar a la parte del programa que queda fuera del alcance de la mónada. Las mónadas han sido descritas como un "punto y coma programable", que transportan datos entre unidades funcionales que los van transformando un paso a la vez.

Sabiendo todo esto, ¿podemos considerar que la programación declarativa es eficiente? Existe una distancia muy grande entre el modelo declarativo y la arquitectura de una computadora. La computadora está optimizada para modificar datos en su lugar, mientras que el modelo declarativo nunca modifica los datos sino que siempre crea datos nuevos. Pero sin embargo esto no es un problema tan grave como podría parecer, porque el compilador puede convertir partes importantes de los programas funcionales en instrucciones imperativas.

> **Pregunta 2:**
> *Piense dos ejemplos de operación declarativa y dos ejemplos de operación no declarativa*

Pregunta 3:
Piense por lo menos una operación que no se pueda llevar a cabo sin estado

9.4. Concurrencia declarativa

Una consecuencia muy valiosa de los programas escritos de forma funcional pura, sin expresiones con efectos secundarios, es que se pueden ejecutar de forma concurrente con otros programas con la garantía de que su semántica permanecerá inalterada. De esta forma, paralelizar programas funcionales puros es trivial. En cambio, para paralelizar programas imperativos, es necesario detectar primero las posibles regiones del programa donde puede haber condiciones de carrera, y, si es necesario, establecer condiciones de exclusión mútua.

Backus acuñó el término "*el cuello de botella von Neumann*" (*von Neumann bottleneck*) para referirse al hecho de que cuando ejecutamos un programa imperativo, la computación funciona de a un paso por vez. Como es posible que cada paso en un programa dependa del previo, tenemos que pasar el estado de la memoria (asignación de valores a variables) de la memoria a la CPU cada vez que ejecutamos un paso de la computación. Este canal secuencial entre la CPU y la memoria es el cuello de botella von Neumann.

Sin embargo, aunque paralelizar programas funcionales sea trivial, es difícil aprovechar este potencial en la práctica. Efectivamente, en algunos puntos de un programa tiene sentido introducir paralelismo y en otros no. El paralelismo tiene un pequeño coste (*overhead*) en la creación de nuevos procesos, incluso si son ligeros, y también en el posible cambio de contexto si hay un solo procesador. Por otro lado, si la ejecución de un proceso depende del resultado de otro, no se obtiene ningún beneficio al paralelizarlos. Es necesario entonces hacer algún tipo de análisis previo para que la paralelización de los programas declarativos sea realmente efectiva.

9.5. Ejercicios

9.1. Cuál de estas dos funciones, `rq` o `rt`, es transparente referencialmente?

```
globalValue = 0;

integer function rq(integer x)
begin
  globalValue = globalValue + 1;
  return x + globalValue;
end

integer function rt(integer x)
begin
  return x + 1;
end
```