Funciones currificadas en TypeScript

Edgar Delgado Vega

15 de diciembre de 2024
Compilado el 09 de Mayo de 2025

Resumen

Currificar, en palabras simples y comunes (como diría *Chayanne*), consiste en convertir una función con varios argumentos en una serie de funciones más simples, cada una tomando solo uno de esos argumentos.

En este breve esbozo, exploraremos algunas ideas de manera informal y relajada. Lo he dividido en dos partes: la primera, dedicada a la sintaxis, y la segunda, a los beneficios y una propuesta para que le des una ojeada.

1. La sintaxis de currificación con TS

Si tenemos una función que recibe cuatro parámetros, la descomponemos en cuatro funciones que toman uno de esos parámetros en cada llamada.

- Tú leyendo: Mucho floro, necesito el código.

Comencemos entonces.

1.1. De la declaración al modo Curry

En TypeScript tenemos varias aproximaciones sintácticas y con ciertas minuciocidades que no vamos a sumergirnos demasiado. Empecemos por el conocimiento previo.

```
function sumFourNumbers(x: number, y: number, z: number, w:
    number): number{
  return x + y + z + w;
}
```

Aquí tenemos la sencilla función declarada `sumFourNumbers`, que recibe 4 parámetros y retorna la suma de esos números bien físicos.

Ahora, pongámosle un toque de Curry... ¡es broma! Vamos a currificarla con un tipado no tan estricto en su retorno:

```
function sumFourNumbers(x: number) {
  return function(y: number) {
    return function(z: number) {
      return function(w: number): number {
        return x + y + z + w;
      };
    };
  };
}
```

- Tú leyendo: ¿Pero qué clase de *Hadōken* es esto?

Tenemos una función declarada que devuelve una serie de funciones anónimas anidadas, una dentro de otra. Es importante notar que cada una de estas funciones recibe un único valor y retorna otra función, salvo la última, que devuelve la suma total al final.

1.2. Las flechas dan poder, pero con gran responsabilidad, dice el Tío Ben.

Recuerda que desde ECMAScript 6 hay funciones flecha, las '() =>'. Pues apliquémoslas a nuestro código de arriba (es equivalente, pero recuerda cómo pelotea 'this' en este contexto):

```
function sumFourNumbers(x: number) {
  return (y: number) => {
    return (z: number) => {
      return (w: number): number => {
        return x + y + z + w;
      };
    };
  };
}
```

- Tú leyendo: Ya lo tengo, me costó pero ya estoy en modo activado.

Aún podemos ser más explícitos en el tipado, definiendo el tipo de cada función anidada:

```
function sumFourNumbers(x: number): (y: number) => (z:
    number) => (w: number) => number {
  return function(y: number): (z: number) => (w: number) =>
      number {
    return function(z: number): (w: number) => number {
      return function(w: number): number {
        return x + y + z + w;
      };
    };
  };
}
```

Este es el **real Hadōken**, apúntalo.

Pero, en realidad, no necesitamos ser tan explícitos ni repetitivos. Para eso existe la inferencia de tipos. Basta con definir la firma de la función en la cabecera. Aquí lo ves con las funciones flecha:

```
function sumFourNumbers(x: number): (y: number) => (z:
    number) => (w: number) => number {
  return (y: number) => {
    return (z: number) => {
      return (w: number): number => {
        return x + y + z + w;
      };
    };
  };
}
```

Sin embargo, todo esto resulta aún verboso.

- Tú leyendo: ¿Existe una forma más concisa de hacer todo este juego sin tanta salsa?

1.3. Hacia la belleza lineal

Sí, el tipado está complicado, o mejor dicho, verborreico. Armemos la misma vaina con una **función expresada**, la montamos sobre una constante con el mismo nombre:

```
const sumFourNumbers = (x: number) => {
  return (y: number) => {
    return (z: number) => {
      return (w: number): number => {
        return x + y + z + w;
      };
    };
  };
};
```

¿Más de lo mismo, no? Sí, pero ahora podemos agregar un alias de tipo para modularizar nuestro juego.

```
type SumFourNumbersType = (x: number) => (y: number) => (z:
    number) => (w: number) => number;

const sumFourNumbers: SumFourNumbersType = (x) => {
  return (y) => {
    return (z) => {
      return (w) => {
        return x + y + z + w;
      };
    };
  };
};
```

¿Esto está mejor, cierto? Sí, está más amigable, pero nuestra función es reasignable, para bien o para mal. Además, ¿qué pasó con el *hoisting*, papá? En fin...

- Tú leyendo: Espera, tengo dos preguntas: ¿Se puede ser más tonero chill? ¿Me parece o puedo quitar los `return`? ¡Sí, el alias de tipo ya te está cantando las fijas!

Pues, luego de tanto viaje, hemos llegado a la **belleza en una línea**:

```
type SumFourNumbersType = (x: number) => (y: number) => (z:
    number) => (w: number) => number;

const sumFourNumbers: SumFourNumbersType = (x) => (y) =>
    (z) => (w) => x + y + z + w;
```

2. Bondades y sombras currificadas en TS

En la sección anterior, nos habíamos quedado dando la hora con la belleza de una línea. Ahora es momento de aprender a aprovechar este superpoder que nos ha dejado.

El truco para trabajar con estas funciones es visualizar paréntesis adyacentes, uno tras otro, según los parámetros que recibe la función. De manera general, para cualquier función `f_n` con `n` parámetros, mandamos algo como esto:

$$f_n()_1()_2()_3()\cdots()_n.$$

En TypeScript, el codiguillo termina siendo

```
const totalSum = sumFourNumbers(10)(20)(30)(40);
console.log(totalSum);
```

- Tú leyendo: Es igual a la función tradicional con cuatro parámetros, ¿para qué hemos hecho todo el paseo hasta aquí?

2.1. Compón paso a paso

Aparentemente, como el título de la **salsa** de Tony Vega, hemos hecho todo por las puras. Pero, una de las cosas más chéveres que tienen las funciones currificadas es que te dejan separar tus bebidas para la mañana y para la tarde:

```
const partialSum = sumFourNumbers(100)(100);
const totalSum = partialSum(100)(100);
```

Como ves, hemos separado dos tazas de café (en mg de cafeína) para el desayuno y dos para el lonche. Con las funciones tradicionales, tendríamos que tomarnos las 4 tazas de café a la vena, de un solo sorbo.

Siendo más meticulosos con nuestras bebidas, podríamos recargarnos con un único café y un té muy cargado para ir a descansar por día:

```
const aCoffe = sumFourNumbers(100)
const aTeaForDreams = aCoffe(150)
```

Debes saber que si mandamos aCoffe por consola al *Playground*, nos muestra los números que faltan alimentar a nuestra función y potenciar nuestro entusiasmo.

```
console.log(aCoffe) // (y) => (z) => (w) => x + y + z + w
```

Igual pasa con `aTeaForDreams`.

En dos días completamos las 4 bebidas no espirituosas.

Recibe y bota las funciones que quieras

No se empieza de esta manera un párrafo u oración, pero lo haremos así a continuación. `sumFourNumbers` pone las cosas más versátiles en cuanto a funciones de orden superior. Dado que cada parte es una función por sí misma, puedes mandarla como argumento o retornarla desde otras funciones.

- Tú leyendo: ¿Cómo se come eso?

```
const scaleBy = (func: SumFourNumbersType, times: number)
    => {
  return (x: number) => (y: number) => (z: number) => (w:
      number) => {
    const totalSum = func(x)(y)(z)(w);
    return totalSum * times;
  };
};

const sumWithScaleByTwo = scaleBy(sumFourNumbers, 2);
const finalUltramanTiga = sumWithScaleByTwo(10)(20)(30)(40);
console.log(finalUltramanTiga);
// Playground arroja: 200
```

Hemos aplicado el escalamiento por dos a nuestra función `sumFourNumbers`. Luego, la llamamos como siempre, pasándole los números físicos.

2.2. No X-men unitarios

Por último, otra gran bondad, cuando escribimos los tests en el modo Curry de las funciones, es que cada valor puede validarse por separado. La comprobación de cada paso no interfiere con los demás.

- Tú leyendo: ¿Y los efectos secundarios?

El estado global ni las variables fuera del scope presentan la realidad alterada. También somos más predecibles, estamos declarando con claridad cómo baila cada función.

2.3. ¿Todo es color de La Pantera Rosa?

- Tú leyendo: ¿Me parece o estás exagerando con lo bueno?

De cierta manera, sí, estamos motivándonos hacia la visión Curry. En ese sentido, aunque todo suena de maravilla, hay algunos handicaps al tratar las cosas currificadas.

El primer inconviente que observas es que, como la marea al atardecer, crece la **pila de llamadas**. Las funciones se llaman unas a otras, ¡míralas! Sin embargo, no te preocupe, estamos lejos del *stack overflow*. La profundidad de la pila depende de la **aridad** de `fn`.

El segundo inconveniente está en el ecosistema TypeScript. En la comunidad, la mayoría de las funciones con las que te encontrarás están en su forma tradicional o expresada, no currificada. Muchas bibliotecas y frameworks adoptan por defecto el enfoque convencional. (¡Sí, existe Ramda para el sabor funcional!)

El tercer problemilla, que surge del segundo, es que el código puede parecer raro al principio (especialmente si trabajas en equipo), para aquellos que no están familiarizados con la idea de currificación.

2.4. ¿Queda o no queda? El lado oscuro de la fuerza

- Tú leyendo: ¿Entonces, cuándo aplicar la salsa? Creo que varias cosas se hacen sin Curry.

Para empezar, vamos con lo obvio: si vas a llamar a una función con todos los parámetros de buenas a primeras, entonces será más conveniente la función tradicional, sencilla y ergonómica. Mas, si quieres seleccionar argumentos, quizás podrías plantearte dos cosas:

1. Tipar con parámetros opcionales: `?`,
2. Sobrecargar la firma de la función.

El punto uno es quizás más directo y digerible; la sobrecarga no es como en **C#**.

Por último, hay un caso raro: si tu equipo y el proyecto siguen estilo currificado. ¡Uno nunca sabe!

2.5. Los nombres del expediente X-Curry

Supongamos que el equipo decide refactorizar hacia una versión currificada. ¡Vaya gemita! Cuando tienes varios parámetros en la función, tienes que cuidar el orden.

- Tú leyendo: No quiero marearme con el orden y recordar qué estoy movilizando.

Te propongo una alternativa que no he visto mucho por ahí: **funciones currificadas con parámetros nombrados.**

Vamos a echar mano del *destructuring* para siguiendo la belleza de una línea, tipar ahora la belleza nombrada de una línea:

```
type SumWithDestructuring = ({x}: {x: number}) => ({y}: {y:
    number}) => ({z}: {z: number}) => ({w}: {w: number}) =>
    number;

const sumFourNumbers: SumWithDestructuring = ({ x }) => ({
    y }) => ({ z }) => ({ w }) => x + y + z + w;

const finalSum = sumFourNumbers({ x: 5 })({ y: 3 })({ z: 2
    })({ w: 7 });
```

Igual podrías hacerlo por partes: lo de mañana y lo de la noche del domingo a las 10, cuando dices: «aún me falta comprar mi lámina **Huascarán**». ¿Es un poco más largo? Sí, pero ahora es mucho más claro lo que estás cocinando.

¡Eso es todo!