Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-422 — Compiladores

Apunte

Azúcar Sintáctico

El *azúcar sintáctico* es un término acuñado por Peter J. Landin [2] para designar a la *sintaxis redundante* que se le agrega a un lenguaje de programación para que sea más fácil la escritura y la lectura de los programas.

Se establece entonces una distinción entre construcciones esenciales del lenguaje, y aquellas que pueden ser explicadas mediante una traducción a las construcciones esenciales, sin agregar expresividad. Pero, ¿a qué nos referimos con expresividad? ¿Son todos los lenguajes Turing-completos igual de expresivos? Felleisen introduce una noción formal más útil de expresividad [1]: informalmente, un lenguaje es más expresivo que otro si nos permite observar más distinciones desde dentro del lenguaje. Una consecuencia de esta idea es que dos lenguajes son igual de expresivos si se puede traducir de uno a otro mediante cambios *locales*. Por lo tanto llamamos azúcar sintáctico a la sintaxis que se puede eliminar mediante cambios locales.

1. Azucarando FD4

Para hacer más agradable la programación extendemos FD4 con azúcar sintáctico. Llamamos entonces FD4 al lenguaje azucarado, y Core FD4, o lenguaje núcleo, al lenguaje sin azúcar sintáctico (que es el lenguaje definido en el apunte "El lenguaje FD4").

1.1. Términos

Vamos a permitir hacer definiciones locales con `let` sin usar paréntesis. Para ellos definimos

$$\texttt{let}\ x : \tau = t\ \texttt{in}\ t' \stackrel{\text{def}}{=} \texttt{let}\ (x : \tau) = t\ \texttt{in}\ t'$$

Para escribir las funciones en una forma más familiar definimos:

$$\texttt{let}\ f\ (x : \tau) : \tau' = t\ \texttt{in}\ t' \stackrel{\text{def}}{=} \texttt{let}\ f : \tau \texttt{->} \tau' = \mathsf{fun}\ (x : \tau) \texttt{->} t\ \texttt{in}\ t'$$

Además permitimos definir funciones de varios argumentos:

$$\mathsf{fun}\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n) \texttt{->} t \stackrel{\text{def}}{=} \mathsf{fun}\ (x_1 : \tau_1) \texttt{->} \ldots \texttt{->} \mathsf{fun}\ (x_n : \tau_n) \texttt{->} t$$

y funciones recursivas de varios argumentos

$$\begin{array}{ll} & \mathsf{fix}\ (f : \tau \texttt{->} \tau_1 \texttt{->} \ldots \texttt{->} \tau_n)(x : \tau)(x_1 : \tau_1)\ \ldots\ (x_n : \tau_n) \texttt{->} t \\ \stackrel{\text{def}}{=} & \mathsf{fix}\ (f : \tau \texttt{->} \tau_1 \texttt{->} \ldots \texttt{->} \tau_n)(x : \tau) \texttt{->} \mathsf{fun}\ (x_1 : \tau_1) \ldots (x_n : \tau_n) \texttt{->} t \end{array}$$

También extendemos las funciones definidas con let a varios argumentos

$$\begin{array}{ll} & \texttt{let}\ f\ (x_1 : \tau_1)\ \ldots\ (x_n : \tau_n) : \tau = t\ \texttt{in}\ t' \\ \stackrel{\text{def}}{=} & \texttt{let}\ f : \tau_1 \texttt{->} \ldots \texttt{->} \tau_n \texttt{->} \tau = \mathsf{fun}\ (x_1 : \tau_1) \ldots (x_n : \tau_n) \texttt{->} t\ \texttt{in}\ t' \end{array}$$

Para facilitar la definición de funciones recursivas definimos:

$$\texttt{let rec}\ f\ (x:\tau):\tau' = t\ \texttt{in}\ t' \stackrel{\text{def}}{=} \texttt{let}\ f:\tau\texttt{->}\tau' = (\mathsf{fix}\ (f:\tau\texttt{->}\tau')\ (x:\tau)\texttt{->}t)\ \texttt{in}\ t'$$

Usamos `let rec` para definir funciones, así que también lo extendemos a varios argumentos.

$$\begin{array}{ll} & \texttt{let rec}\ f\ (x_1:\tau_1)\ldots(x_n:\tau_n):\tau = t\ \texttt{in}\ t' \\ \stackrel{\text{def}}{=} & \texttt{let rec}\ f\ (x_1:\tau_1):\tau_2\texttt{->}\ldots\texttt{->}\tau_n\texttt{->}\tau = \mathsf{fun}\ (x_2:\tau_2)\ldots(x_n:\tau_n)\texttt{->}t\ \texttt{in}\ t' \end{array}$$

1.2. Declaraciones

Introducimos azúcar sintáctico para declaraciones en forma análoga a los casos de términos `let` y `let rec`.

$$\texttt{let}\ f\ (x:\tau):\tau' = t \stackrel{\text{def}}{=} \texttt{let}\ f:\tau\texttt{->}\tau' = \mathsf{fun}\ (x:\tau)\texttt{->}t$$

$$\texttt{let}\ f\ (x_1:\tau_1)\ \ldots\ (x_n:\tau_n):\tau = t \stackrel{\text{def}}{=} \texttt{let}\ f:\tau_1\texttt{->}\ldots\texttt{->}\tau_n\texttt{->}\tau = \mathsf{fun}\ (x_1:\tau_1)\ldots(x_n:\tau_n)\texttt{->}t$$

$$\texttt{let rec}\ f\ (x:\tau):\tau' = t \stackrel{\text{def}}{=} \texttt{let}\ f:\tau\texttt{->}\tau' = \mathsf{fix}\ (f:\tau\texttt{->}\tau')\ (x:\tau)\texttt{->}t$$

$$\texttt{let rec}\ f\ (x_1:\tau_1)\ldots(x_n:\tau_n):\tau = t \stackrel{\text{def}}{=} \texttt{let rec}\ f\ (x_1:\tau_1):\tau_2\texttt{->}\ldots\texttt{->}\tau_n\texttt{->}\tau = \mathsf{fun}\ (x_2:\tau_2)\ldots(x_n:\tau_n)\texttt{->}t$$

Por otro lado, por uniformidad y para soporte de futuros cambios, no permitiremos las declaraciones sin tipo (al estilo de $\texttt{let}\ x = t$), requiriendo declaraciones:

$$\texttt{let}\ x:\tau = t$$

1.3. Operadores

El programa `let f (g : Nat -> Nat) : Nat = t in g (print "x=")` no es un término de Core FD4, ya que en Core FD4 el operador unario `print "x="` debe estar aplicado a un término. La solución es η-expandir el término de la siguiente manera:

```
let f (g : Nat -> Nat) : Nat = t in f (fun (x : Nat) -> print "x=" x)
```

Para permitir el operador unario sin aplicar, realizaremos la η-expansión en las ocurrencias no aplicadas del operador `print`. Notar que sólo queremos hacer la η-expansión en las ocurrencias no aplicadas para no introducir una aplicación innecesaria.

1.4. Sinónimos de Tipos

El azúcar sintáctico también se puede aplicar a los tipos. Introducimos en nuestro lenguaje la posibilidad de declarar sinónimos de tipos para todo tipo τ ya definido:

$$\texttt{type}\ n\ =\ \tau$$

El significado de esta declaración es que, de aquí en adelante, n debe interpretarse como τ. La declaración en sí no se convierte en una declaración de Core FD4.

1.5. Opcional: multibinders

En cada construcción del lenguaje que puede tener varios binders (`fun`, `let` y `let rec`, tanto en términos como declaraciones), podemos abreviar el caso cuando muchos binders consecutivos comparten un tipo de la siguiente forma:

$$\begin{array}{ll} & \texttt{let}\ f\ \vec{b_0}\ (x_0\ x_1\cdots x_n:\tau) \qquad\qquad \vec{b_1}:\tau' = t \\ \stackrel{\text{def}}{=} & \texttt{let}\ f\ \vec{b_0}\ (x_0:\tau)\ (x_1:\tau)\ \cdots\ (x_n:\tau) \quad \vec{b_1}:\tau' = t \end{array}$$

Donde $\vec{b_0}$ y $\vec{b_1}$ representan otros binders. Así, podemos definir la funcion *suma* vista anteriormente como:

$$\begin{array}{l} \texttt{let rec}\ \mathit{suma}\ (m\ n:\mathbb{N}):\mathbb{N} = \\ \quad \mathsf{ifz}\ n\ \mathsf{then}\ m\ \mathsf{else}\ 1 + \mathit{suma}\ m\ (n-1) \end{array}$$