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. 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:

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

Ejercicios

Ej. 1. Traduzca los siguientes términos de FD4 a Core FD4 según las reglas de arriba:

a) `let x : Nat = 2 in x + 1`

b) `fun (x:Nat) -> x`

c) `let id (x:Nat) : Nat = x in id 10`

d) `let app5 (f : Nat -> Nat) : Nat = f 5 in app5 (fun (x : Nat) -> x + x)`

e) `fun (x:Nat) (y:Nat) -> ifz x then y else 1`

Ej. 2. Traduzca las siguientes declaraciones de FD4 a Core FD4 según las reglas de arriba:

a) `let rec doble (x:Nat) : Nat = ifz x then 0 else 2 + (doble (x - 1))`

b)
```
let rec ack (m:Nat) (n:Nat) : Nat =
                  ifz m
                  then n + 1
                  else (ifz n
                        then ack (m - 1) 1
                        else ack (m - 1) (ack m (n - 1)))
```

2. Implementación

Como vimos antes, el azúcar sintáctico hace más agradable a Core FD4 al permitirnos escribir funciones con muchos argumentos, `let rec` en vez de un `fix` explícito, etc. Otra forma de lograr esto es agregar estas nuevas funcionalidades al lenguaje, directamente al AST interno (`Tm`), por ejemplo con nodos `Let` y `LetRec`, y cambiando `Fun` para tomar una lista de binders. Pero esto implicaría cambiar *todas* las fases posteriores para trabajar con estos nuevos constructores (typechecking, pretty-printing, compilación, etc). Al implementar estas funcionalidades con azúcar sintáctico, estos cambios solamente deben ser espolvoreados en las primeras etapas del compilador.

Hay dos maneras distintas de implementar el azúcar sintáctico, aunque en ambas (claramente) debe modificarse el parser para reconocer las nuevas construcciones. Como primer alternativa se puede implementar directamente en el parser, agregando una regla para parsear, por ejemplo, `let` $f\ (x : \mathbb{N}) : \mathbb{N} = x + 2$ `in` $f\ 4$ que genere el mismo término que `let` $(f : \mathbb{N}$ `->` $\mathbb{N}) =$ `fun` $(x : \mathbb{N})$ `->` $x + 2$ `in` $f\ 4$. La segunda opción, es definir un tipo de AST "superficial" que contiene nodos que representan el azúcar sintáctico (p. ej. `LetRec`, o declaraciones `type`), y que el parser usa directamente. Luego, hay una etapa de *desugaring* que convierte el AST superficial al AST interno, que no tiene azúcar. Este desacoplamiento puede ser conveniente: al dividir lo que es el parsing del azúcar, las transformaciones quedan más explícitas y el código se torna más mantenible. Por otro lado, tener un AST superficial permite usar el pretty-printer para *formatear* archivos fuente. Podemos tomar un archivo, parsearlo hasta el AST superficial (sin elaborarlo), y pretty-printearlo nuevamente con indentación adecuada. Si usaramos la opción anterior, el parser cambiaría la sintaxis del archivo al remover el azúcar (e.g. los `let rec` desaparecerían), algo que no es deseable para un formateador. A veces, el AST superficial incluye hasta un nódo de paréntesis, para poder conservar `(f g) h`, y otros, como tal. Otras ventajas incluyen que depurar el parser se hace más fácil, y a medida que las transformaciones aumentan es más fácil implementarlas.

En ambos casos, la impresión de términos (ya elaborados) se ve afectada al perder el azúcar. Algunos compiladores intentan imprimir estos términos de mejor manera, por ejemplo manteniendo información sobre cómo se desugareó un término o intentando reconstruir la forma azucarada a partir de la interna. Si se guardan anotaciones en el AST interno para reconstruir la versión azucarada, es importante que

solamente sean usadas para imprimir, y de esta manera estar seguros que el significado del azúcar sólo es dado por el *desugaring*.

2.1. Azúcar en FD4

Implementar el azúcar directamente en el parser es posible, aunque con las desventajas que listamos más arriba. En nuestro curso tomamos la segunda opción, en la cual se define un tipo `STm` que representa a los términos superficiales.

Por lo tanto, para implementar el azúcar sintáctico hay que modificar el tipo `STm` para que contenga nodos apropiados para el azúcar (p. ej. `SLet` con un booleano que indique si es recursivo o no), y cuyos nodos para binders tengan una lista de binders (`[(Name, Ty)]`) en vez de uno (`SLam` y `SLet`) o dos (`SFix`). La etapa de desugaring convertirá un `SLam` con muchas variables a una aplicación sucesiva de nodos `Lam` internos. En el parser, conviene escribir una regla que parsee una lista de binders (`binders :: P [(Name, Ty)]`), que puede reusarse muchas veces. La etapa de elaboración puede ser recursiva, por ejemplo (como en las reglas dadas arriba) al encontrar un `let rec`, podemos transformarlo en un término (superficial) que contiene un `SLam` con muchas variables, y luego elaborarlo recursivamente a funciones anidadas. Luego modificar en el módulo `Elab` la función `elab` para que transforme términos `STerm` en términos *locally nameless* `Term`.

En la sintaxis dada, se requiere que un let rec tenga al menos un argumento para poder elaborarlo a un fix, entre otros requerimientos similares. Hay dos opciones aquí: el parser puede fallar si no hay un argumento, dando un error de sintaxis, o podemos parsear una lista de binders vacía y hacer que la elaboración falle (con gracia, via failPosFD4). La segunda opción tiene la ventaja que permite errores mas descriptivos, pero tiene la desventaja que requiere agregar la posibilidad de que la elaboración fracase.

Es útil agregar un tipo para declaraciones superficiales. A diferencia de las declaraciones `Decl`, pueden almacenar si una declaración es recursiva o no, llevar una lista de argumentos, guardar el tipo declarado, etc.

TAREA 1: Implementar el azúcar sintáctico en el proyecto.

a) Modificar los términos superficiales `STerm` para reflejar la nueva sintaxis superficial y definir declaraciones superficiales.

b) Modificar el parser para que parsee términos superficiales y declaraciones superficiales. Puede hacer test básicos, llamando a `cabal exec ghci -- -isrc src/Parse.hs` y ejecutar el parser desde el entorno interactivo.

c) Modificar el elaborador para que elimine el azúcar sintáctico al producir `Term`, y agregar una función que elabore declaraciones superficiales en `Decl`. Es probable que las funciones de elaboración no puedan ser puras y deban usar la mónada FD4.

Puede hacer test básicos, llamando a `cabal exec ghci -- -isrc src/Parse.hs` y ejecutar el elaborador desde el entorno interactivo.

d) Modificar el pretty-printer para que funcione con los nuevos `STerm`.

e) Modificar el `Main.hs` para trabajar con términos y declaraciones superficiales.

f) (Opcional) agregar un resugaring al pretty-printer.

3. 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.

4. Implementación

Para los tipos la situación es similar a los términos. Conviene definir tipos superficiales, y en la elaboración resolver los sinónimos de tipos a un tipo base. Puede ser conveniente guardar el nombre original del tipo para hacer mas descriptivos los mensajes de error y/o el pretty printing de términos. Para esto último hay que modificar los tipos de Core FD4.

Finalmente, modificar el tipo de las declaraciones superficiales, ya que tenemos dos tipos de declaraciones: declaraciones de sinónimos de tipos y declaraciones de términos.

TAREA 2: Implementar sinónimos de tipo.

- Definir un tipo `STy` para tipos superficiales.
- Modificar el parser para que parsee tipos superficiales.
- Agregar funciones al elaborador que elaboren tipos superficiales en `Ty`.
- Para agregar sinónimos de tipos, agregar al estado global un mapeo de nombres a tipos. Agregar funciones a la `MonadFD4` para poder agregar y buscar en esta nueva componente del estado.
- Si decide recordar los nombres de los sinónimos elegidos por el programador en los tipos `Ty` (para producir mejores mensajes de error / pretty-printing) es necesario modificar la comparación de tipos en el typechecker.

Referencias

[1] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35 – 75, 1991.

[2] Peter J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 01 1964.