Lenguajes de Programación 2023-1
Nota de clase 6: MinHs
Funcional (λ)

Javier Enríquez Mendoza

31 de octubre de 2022

En esta nota de clase se presenta un lenguaje de programación funcional basado en Haskell, este lenguaje es MinHs . Este lenguaje sirve como base teórica para la formalización y estudio de los lenguajes de programación funcionales al implementar en él conceptos primordiales del lenguaje Haskell, como lo son: evaluación perezosa, tipado estático, tipado explícito, pureza funcional, entre otros.

1 Anotaciones de tipos

Consideremos la siguiente expresión:

```
if 5 < True then 2 else 6
```

Esta expresión puede ser sintacticamente correcta, sin embargo la evaluación de ésta no va a regresar un valor como resultado, esto se debe a que la subexpresión `5 < Truet` es incorrecta, ya que el operador $<$ se define para números y no para booleanos, sin embargo, en la expresión se usa con el valor `True`.

En realidad, la expresión anterior tiene un error ya que estamos intentando aplicar una operación a una expresión sobre la cual no puede aplicarse, es decir, el tipo de los parámetros de la operación $<$ no corresponden con los esperados. Esto se trata de un error semántico, por lo que es necesario definir una semántica que capture estos errores.

Para corregir esto se busca definir un sistema de tipos, con el que podremos capturar los errores de tipos de las expresiones del lenguaje. El proceso de encontrar errores de tipos puede ser un proceso complicado, ya que en algunos casos es difícil saber si una expresión está bien tipada o no. Para facilitarlo se puede definir un lenguaje en donde su sintaxis incluya anotaciones de tipos, es decir, que algunos constructores del lenguaje indiquen los tipos con los que trabajan.

Ya con estas anotaciones, podemos encontrar los errores de tipos de forma meramente sintáctica, esto significa que podemos definir el proceso de verificación de tipos de una expresión como una semántica estática para el lenguaje.

En MinHs se buscará tener un mínimo de anotaciones de tipos, esto es, que sólo se agreguen anotaciones en aquellos constructores en donde sea estrictamente necesario para verificar la correctud de su tipado, mientras que para los constructores en donde no sea necesaria la anotación, ésta será omitida.

2 Sintaxis Concreta

A continuación se presenta la sintaxis concreta del lenguaje utilizando una gramática libre de contexto.

Definición 2.1 (Sintaxis Concreta de MinHs)**.** La sintaxis concreta para MinHs cuenta con anotaciones explicitas de tipos y se define con la siguiente gramática libre del contexto.

Expresiones	e	$::=$	$var \mid n \mid b \mid (e) \mid e_1 \otimes e_2 \mid e_1\, e_2$
		$\mid$	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
		$\mid$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$
		$\mid$	$\texttt{lam}\ x :: \mathsf{T} \Rightarrow e$
		$\mid$	$\texttt{recfun}\ f :: (\mathsf{T}_1 \to \mathsf{T}_2)\ x \Rightarrow e$
Tipos	T	$::=$	$\mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{T}_1 \to \mathsf{T}_2$
Números	n	$::=$	$0 \mid 1 \mid \ldots$
Booleanos	b	$::=$	$\texttt{true} \mid \texttt{false}$
Operadores Infijos	$\otimes$	$::=$	$+ \mid * \mid - \mid = \mid < \mid > \mid \geq \mid \leq$

Observación. Es importante notar que la gramática anterior es ambigua, sin embargo por simplicidad se trabajará con ésta, sabiendo que la forma de quitar la ambigüedad es haciendo explicita la precedencia y asociatividad de los operadores de forma similar a como se hizo con el lenguaje EA .

En esta definición del lenguaje la representación de funciones es algo peculiar, siendo mas cercana a una representación interna usada por un compilador, al separar las funciones sin recursión de aquellas que son recursivas.

Observación. En MinHs tenemos dos tipos básicos Nat y Bool y un constructor binario de tipos $\to$, que denota el tipo de una función. El constructor de tipos función asocia a la derecha. Por ejemplo el tipo:

$$\mathsf{Nat} \to \mathsf{Bool} \to \mathsf{Nat}$$

es el mismo que

$$\mathsf{Nat} \to (\mathsf{Bool} \to \mathsf{Nat})$$

Y representa a una función que recibe como parámetro una expresión de tipo Nat y regresa una función $\mathsf{Bool} \to \mathsf{Nat}$.
De la misma forma que en el lenguaje de programación Haskell el tipo:

$$\mathsf{Nat} \to \mathsf{Bool} \to \mathsf{Nat}$$

se puede pensar como una función que recibe un primer parámetro de tipo Nat y un segundo parámetro de tipo Bool y regresa un valor de tipo Nat .

Ejemplos. A continuación se presentan algunos ejemplos de expresiones válidas del lenguaje MinHs .

- Operaciones aritméticas:

```
5 + 6
```

- El condicional `if`

```
if 5 > x then true else 8
```

- La función identidad sobre los naturales

```
let id (Nat -> Nat) = (fun x :: Nat => x) in id 5 end
```

- La función factorial se define como:

```
recfun fact :: (Nat -> Nat) n =>
    if (n == 0)
        then 1
        else n * fact (n-1)
```

- Una función de dos parámetros se puede definir como:

```
fun x :: Nat => fun y :: Bool => x + y
```

es decir, mediante currificación.

Es importante notar en los ejemplos anteriores como las anotaciones de tipo por si solas no son suficiente para evitar los errores de tipos, es por eso que es necesario un proceso de verificación de tipos que se define en secciones siguientes.

3 Sintaxis Abstracta

En esta sección se presenta la sintaxis abstracta de orden superior del lenguaje MinHs .

Definición 3.1 (Sintaxis abstracta de MinHs)**.** La sintaxis abstracta se define con las siguientes reglas:

Constantes

$$\frac{n \in \mathbb{N}}{\texttt{num}[n]\ \mathsf{asa}} \qquad \frac{}{\texttt{bool}[\texttt{true}]\ \mathsf{asa}} \qquad \frac{}{\texttt{bool}[\texttt{false}]\ \mathsf{asa}}$$

Operadores

$$\frac{t_1\ \mathsf{asa} \quad \cdots \quad t_n\ \mathsf{asa}}{o(t_1, \ldots, t_n)\ \mathsf{asa}}$$

Condicional

$$\frac{t_1\ \mathsf{asa} \quad t_2\ \mathsf{asa} \quad t_3\ \mathsf{asa}}{\texttt{if}(t_1, t_2, t_3)\ \mathsf{asa}}$$

Asignaciones locales

$$\frac{t_1\ \mathsf{asa} \quad t_2\ \mathsf{asa}}{\texttt{let}(t_1, x.t_2)\ \mathsf{asa}}$$

Definición de funciones

$$\frac{t \text{ asa}}{\texttt{lam}(\mathsf{T}, x.t) \text{ asa}} \qquad \frac{t \text{ asa}}{\texttt{recfun}(\mathsf{T}, f.x.t) \text{ asa}}$$

Aplicación de función

$$\frac{t_1 \text{ asa} \quad t_2 \text{ asa}}{\texttt{app}(t_1, t_2) \text{ asa}}$$

Operador de punto fijo

$$\frac{t \text{ asa}}{\texttt{fix}(\mathsf{T}, f.t) \text{ asa}}$$

El operador de punto fijo `fix` es el único árbol de sintaxis abstracta que no corresponde a ninguna expresión de la sintaxis concreta, esto quiere decir que el usuario del lenguaje no puede definir expresiones `fix`. Esto se debe a que este operador es de uso interno para la evaluación de expresiones, como se ve mas adelante en esta nota.

Observación. Notemos que en el caso de las funciones recursivas con el operador `recfun` la variable definida se liga tanto en el cuerpo como en la expresión que le da valor a la variable. Esto es ya que permite las llamadas recursivas.

4 Sistema de tipos

Definición 4.1 (Tipo)**.** Un tipo es una descripción abstracta de una colección de valores particulares.

Un sistema de tipos es un conjunto de reglas que definen ciertas restricciones en la formación de programas. Las frases del lenguaje se clasifican mediante tipos que dictan cómo pueden usarse. Intuitivamente el tipo de una expresión define la forma de su valor, por ejemplo, la comparación de dos expresiones numéricas con el operador $<$ debe ser un valor booleano.

De esta forma si intentamos comparar una expresión numérica con una expresión booleana, como en el ejemplo de la sección anterior, debería generar un error de tipos.

Para poder definir un sistema de tipos sobre un lenguaje, necesitamos un conjunto de tipos los cuales vamos a asociar a cada una de las expresiones del lenguaje mediante una colección de reglas de tipado.

El uso de sistemas de tipos en el diseño de un lenguaje de programación tiene diferentes ventajas, como pueden ser:

- Permite descubrir errores en expresiones tempranamente.
- Ofrece seguridad, un programa correctamente tipado no puede funcionar mal.
- Los tipos documentan un programa de manera mas simple y manejable que los comentarios.
- Los lenguajes tipados pueden implementarse de manera más clara y eficiente.

5 Semántica estática

Como habiamos visto con anterioridad la semántica estática determina qué expresiones del lenguaje están bien formadas de acuerdo a ciertos criterios sensibles al contexto como la resolución del alcance, al requerir que cada variable sea declarada antes de usarse (como la definida en la nota 4 mediante la relación $\Delta \sim e$).

Por lo general la semántica estática consiste de dos fases:

- La resolución del alcance de variables.
- La verificación de correctud estática de un programa mediante el sistema de tipos.

Antes de definir el significado preciso de un programa, mediante su semántica operacional, es necesario eliminar los programas mal formados, en el caso de MinHs que es un lenguaje es fuertemente tipado su semántica dinámica sólo está definida si el programa en cuestión está bien formado respecto a su sistema de tipos.

Definición 5.1 (Semántica estática)**.** Se define la semántica estática del lenguaje MinHs con el siguiente conjunto de reglas de tipado para los árboles de sintaxis abstracta de las expresiones del lenguaje mediante un juicio ternario entre expresiones t, tipos T y contextos de declaraciones de variables tipadas Γ denotado:

$$\Gamma \vdash t : \mathsf{T}$$

que se lee como *la expresión t tiene tipo T bajo el contexto Γ*. En donde Γ es un conjunto de asignaciones de tipos a variables de la forma $\{x_1 : \mathsf{T}_1 \dots x_n : \mathsf{T}_n\}$, en donde cada variable aparece una única vez en Γ, es decir, a cada variable se le asigna un sólo tipo en el contexto.[a]

Variables

$$\frac{}{\Gamma, x : \mathsf{T} \vdash x : \mathsf{T}}$$

Valores numéricos

$$\frac{}{\Gamma \vdash \mathtt{num}[n] : \mathsf{Nat}}$$

Valores Booleanos

$$\frac{}{\Gamma \vdash \mathtt{bool}[\mathtt{true}] : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathtt{bool}[\mathtt{true}] : \mathsf{Bool}}$$

Operadores

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{suma}(t_1, t_2) : \mathsf{Nat}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{prod}(t_1, t_2) : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{sub}(t_1, t_2) : \mathsf{Nat}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{eq}(t_1, t_2) : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{gt}(t_1, t_2) : \mathsf{Bool}} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{lt}(t_1, t_2) : \mathsf{Bool}}$$

Condicional

$$\frac{\Gamma \vdash t_c : \mathsf{Bool} \quad \Gamma \vdash t_t : \mathsf{T} \quad \Gamma \vdash t_e : \mathsf{T}}{\Gamma \vdash \mathtt{if}(t_c, t_t, t_e) : \mathsf{T}}$$

Asignaciones Locales

$$\frac{\Gamma \vdash t_v : \mathsf{T} \quad \Gamma, x : \mathsf{T} \vdash t_b : \mathsf{S}}{\Gamma \vdash \mathtt{let}(t_v, x.t_b) : \mathsf{S}}$$

Funciones

$$\frac{\Gamma, x : \mathsf{T} \vdash t : \mathsf{S}}{\Gamma \vdash \mathtt{lam}(\mathsf{T}, x.t) : \mathsf{T} \to \mathsf{S}} \qquad \frac{\Gamma, f : \mathsf{T} \to \mathsf{S}, x : \mathsf{T} \vdash t : \mathsf{S}}{\Gamma \vdash \mathtt{recfun}(\mathsf{T} \to \mathsf{S}, f.x.t) : \mathsf{T} \to \mathsf{S}}$$

Aplicación de función

$$\frac{\Gamma \vdash t_f : \mathsf{T} \to \mathsf{S} \quad \Gamma \vdash t_p \mathsf{T}}{\Gamma \vdash \mathtt{app}(t_f, t_p) : \mathsf{S}}$$

Operador de punto fijo

$$\frac{\Gamma, x : \mathsf{T} \vdash t : \mathsf{T}}{\Gamma \vdash \mathtt{fix}(\mathsf{T}, x.t) : \mathsf{T}}$$

Obsérvese que en el caso de `fix` se está asumiendo el mismo tipo que se debe concluir.

[a] Se utiliza la notación $\Gamma, x : \mathsf{T}$ para indicar el conjunto $\Gamma \cup \{x : \mathsf{T}\}$

Observación. En el conjunto de reglas anteriores, para las reglas que agregan variables al contexto (`let`, `letrec`, `lam` y `fix`) se tiene que verificar que conserven la estructura de Γ en donde no puede haber dos presencias de la misma variable, esto se soluciona con el uso de α-equivalencias pues en todos estos constructores las variables definidas están ligadas por lo que se puede cambiar su nombre. Mas adelante en el curso veremos estrategias automatizadas que son utilizadas por los compiladores para resolver esto.

Ejemplo 5.1 (Derivación de tipos.)**.** Para ejemplificar el uso de la semántica estática del lenguaje MinHs se hará la verificación de tipos de la siguiente expresión.

$$\mathtt{let}\, x : \mathsf{Nat} = 5\, \mathtt{in}\, \mathtt{if}\, (x = 0)\, \mathtt{then}\, x * 2\, \mathtt{else}\, x\, \mathtt{end}$$

Que se ve en sintaxis abstracta como sigue:

$$\mathtt{let}(\mathsf{Nat}, 5, x.\mathtt{if}(\mathtt{eq}(x, 0), \mathtt{prod}(x, 2), x))$$

$$\dfrac{\dfrac{}{\vdash 5 : \mathsf{Nat}} \qquad \dfrac{\dfrac{\dfrac{\vdots}{x : \mathsf{Nat} \vdash \mathtt{eq}(x, 0) : \mathsf{Bool}} \quad \dfrac{\vdots}{x : \mathsf{Nat} \vdash \mathtt{prod}(x, 2) : \mathsf{Nat}} \quad \dfrac{}{x : \mathsf{Nat} \vdash x : \mathsf{Nat}}}{x : \mathsf{Nat} \vdash \mathtt{if}(\mathtt{eq}(x, 0), \mathtt{prod}(x, 2), x) : \mathsf{Nat}}}{\vdash \mathtt{let}(\mathsf{Nat}, 5, x.\mathtt{if}(\mathtt{eq}(x, 0), \mathtt{prod}(x, 2), x)) : \mathsf{Nat}}$$

La derivación para la condición del `if`se ve como sigue:

$$\frac{\dfrac{}{x : \mathsf{Nat} \vdash x : \mathsf{Nat}} \qquad \dfrac{}{x : \mathsf{Nat} \vdash 0 : \mathsf{Nat}}}{x : \mathsf{Nat} \vdash \mathtt{eq}(x, 0) : \mathsf{Bool}}$$

Mientras que la derivación para el caso `then` se muestra a continuación.

$$\frac{\dfrac{}{x : \mathsf{Nat} \vdash x : \mathsf{Nat}} \quad \dfrac{}{x : \mathsf{Nat} \vdash 2 : \mathsf{Nat}}}{x : \mathsf{Nat} \vdash \mathtt{prod}(x, 2) : \mathsf{Nat}}$$

6 Semántica Dinámica

Como mencionamos anteriormente MinHs es un lenguaje basado en Haskell, de hecho su nombre significa mini Haskell, es por esto que este lenguaje sigue muchas de convenciones utilizadas en Haskell.

Quizá una de las caracteristicas mas representativas de Haskell sea el uso de una estrategia de evaluación perezosa, por esta razón, la semántica operacional definida para MinHs se define de esta misma forma.

Definición 6.1 (Semántica Operacional de paso pequeño perezosa)**.** Se define la semántica dinámica de MinHs mediante el siguiente sistema de transición:

- Conjunto de estados $S = \{a \mid a \ \mathsf{asa}\}$
- Estados Iniciales $I = \{a \mid a \ \mathsf{asa}, \ \varnothing \sim a\}$
- Estados Finales $F = \{\mathtt{num}[n], \mathtt{bool}[\mathtt{true}], \mathtt{bool}[\mathtt{false}], \mathtt{lam}(x.t)\}$
- Transiciones, dadas por las siguientes reglas:

  Condicional

$$\frac{}{\mathtt{if}(\mathtt{bool}[\mathtt{true}], a_t, a_e) \to a_t}\ \mathsf{ifT} \qquad \frac{}{\mathtt{if}(\mathtt{bool}[\mathtt{false}], a_t, a_e) \to a_e}\ \mathsf{ifF}$$

$$\frac{a_c \to a_c'}{\mathtt{if}(a_c, a_t, a_e) \to \mathtt{if}(a_c', a_t, a_e)}\ \mathsf{if}$$

  Asignaciones locales

$$\frac{}{\mathtt{let}(a_1, x.a_2) \to a_2[x := a_1]}\ \mathsf{let}$$

  Aplicación de función

$$\frac{a_f \to a_f'}{\mathtt{app}(a_f, a_p) \to \mathtt{app}(a_f', a_p)}\ \mathsf{app}$$

$$\frac{}{\mathtt{app}(\mathtt{lam}(\mathsf{T}, x.a_b), a_p) \to a_b[x := a_p]}\ \mathsf{appL}$$

$$\frac{}{\mathtt{app}(\mathtt{recfun}(\mathsf{T}, f.x.a_b), a_p) \to a_b[f := \mathtt{fix}(\mathsf{T}, f.x.a_b), x := a_p]}\ \mathsf{appR}$$

$$\frac{}{\mathtt{app}(\mathtt{fix}(\mathsf{T}, f.x.a_b), a_p) \to a_b[f := \mathtt{fix}(\mathsf{T}, f.x.a_b), x := a_p]}\ \mathsf{appF}$$

  Operador de punto fijo

$$\frac{}{\mathtt{fix}(\mathsf{T}, f.a) \to a[f := \mathtt{fix}(\mathsf{T}, f.a)]}\ \mathsf{fix}$$

  El resto de las reglas de transición se heredan directamente del lenguaje EA o se definen de forma análoga a las dadas para los operadores de suma y producto dentro de ese lenguaje.

Observación. La pereza de la semántica operacional se puede ver en las reglas de evaluación de asignaciones locales y aplicación de función en donde los valores se sustituyen directamente en el cuerpo sin evaluarlos previamente.

7 Propiedades del lenguaje

En esta sección únicamente se estudiará la propiedad de terminación de MinHs , el resto de las propiedades serán estudiadas en la nota siguiente después de agregar anotaciones de tipos al lenguaje.

7.1 No terminación de MinHs

Podemos observar que en la semántica operacional definida para el operador `fix` se sustituye su argumento por la misma expresión que se desea evaluar, lo que modela una recursión general.

$$\texttt{fix}(f.a) \to a[f := \texttt{fix}(f.a)]$$

A esta operación se le conoce como el desdoblamiento de la definición recursiva.

Esta definición puede generar programas con ciclos infinitos, lo que hace que se pierda la propiedad de terminación para el lenguaje MinHs .

Ejemplo 7.1 (Ciclo infinito en MinHs)**.** Para la expresión $\texttt{fix}(x.x)$ se tiene el siguiente proceso de evaluación:

$$\texttt{fix}(x.x) \to x[x := \texttt{fix}(x.x)] = \texttt{fix}(x.x) \to \texttt{fix}(x.x) \to \texttt{fix}(x.x) \dots$$

7.2 Seguridad del lenguaje

Una propiedad fundamental de los sistemas de tipos es la llamada seguridad o correctud del sistema la cual dice lo siguiente : los programas correctamente tipados no pueden funcionar mal.

Como vimos al principio de la nota un programa es erróneo si su evaluación se bloquea, es decir termina en una expresión que no es un valor simple y no puede seguir evaluándose.

La seguridad del sistema de tipos relaciona a la semántica estática con la semántica dinámica y se prueba generalmente en dos partes:

- **Progreso** Un programa correctamente tipado no se bloquea.
- **Preservación**: Si un programa correctamente tipado se ejecuta entonces la expresión resultante está correctamente tipada y en muchos casos el tipo de ambos coincide.

Veamos estas propiedades para el sistema de tipos de MinHs que acabamos de definir.

Proposición 7.1 (Unicidad del tipado)**.** Para cualesquiera contexto Γ y expresión t de MinHs ,

existe a lo mas un tipo T tal que se cumple la relación:

$$\Gamma \vdash t : \mathsf{T}$$

Es decir, cada expresión del lenguaje tiene unicamente un tipo correspondiente.

Demostración. La demostración es por inducción sobre t. □

Proposición 7.2 (Preservación de tipos)**.** Si $\Gamma \vdash t : \mathsf{T}$ y $t \to t'$ entonces $\Gamma \vdash t' : \mathsf{T}$

Demostración. La demostración es por inducción sobre $\Gamma \vdash t : \mathsf{T}$. □

Proposición 7.3 (Progreso)**.** Si $\vdash t : \mathsf{T}$ para algun tipo T, es decir se puede derivar el tipo de e desde el contexto vacío, entonces se cumple unicamente una de las siguientes condiciones:

- t es un valor.
- Existe una expresión t' tal que $t \to t'$

Demostración. La demostración es por inducción sobre $\vdash t : \mathsf{T}$ □

Referencias

[1] Keller G., O'Connor-Davis L., Class Notes from the course Concepts of programming language design, Department of Information and Computing Sciences, Utrecht University, The Netherlands, Fall 2020.

[2] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.

[3] Ramírez Pulido K., Soto Romero M., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-2

[4] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.

[5] Mitchell J., Foundations for Programming Languages. MIT Press 1996.

[6] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.