Universidad Nacional Autónoma de México

Facultad de Ciencias

Reporte de Actividad Docente

Manual de ejercicios para la materia de Lenguajes de Programación.

Que para obtener el Tíutlo de:

Licenciado en Ciencias de la Computación.

Presenta: Luis Miguel Muñoz Barón

Tutor: Javier Enriquez Mendoza

2023

Nota al lector

El presente trabajo es una compilación de ejercicios teóricos para la materia de lenguajes de programación de la carrera en ciencias de la computación con clave de asignatura 1536, siguiendo el plan de estudios impartido desde el año 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México.

El objetivo principal de este material es que sirva como guía para desarrollar los contenidos visitados en este curso siendo un apoyo didáctico para profesores y alumnos donde cada ejercicio planteado durante el capítulo esté acompañado de la base teórica para poder seguir el desarrollo del mismo y ser entendido en su totalidad.

Cada capítulo cuenta con dos secciones: una sección de teoría autocontenida con ejercicios resueltos y una sección de ejercicios sin respuesta para el lector al final del mismo.

Las demostraciones y desarrollo de los contenidos teóricos serán discutidos de forma laxa dejando a discreción del lector el estudio a profundidad de la información aquí presentada concentrándonos únicamente en la teoría necesaria para poder dar solución a los ejercicios, asimilar los conceptos y definiciones, familiarizarnos con la explicación, planteamiento y respuesta del material aquí presentado.

Para un estudio en profundidad se refiere a las personas interesadas a consultar [1], [2], [5], [12], [22] y [26] de la bibliografía que se encuentra al final del presente manual.

Finalmente espero que las personas que hagan uso de este material encuentren útiles y fructífero los desarrollos y conceptos escritos en las páginas de este texto.

Miguel Barón.

"A language is not just words. It's a culture, a tradition, a unification of a community, a whole history that creates what a community is. It's all embodied in a language"

-Noam Chomsky

Índice general

Capítulo 1

Introducción

En el contexto de la computación, los lenguajes de programación fungen como una interfaz entre programador y procesador para establecer una comunicación sobre lo que queremos que la computadora ejecute por nosotros.

Diferentes paradigmas de lenguajes de programación, estilos, reglas y convenciones se han adoptado en las últimas décadas para ajustarse al mejor planteamiento posible de un problema, mantener un estándar declarativo, un nombramiento homogéneo de variables y métodos, o un estilo idéntico compartido entre los diferentes programas que se agrupan en estas categorías.

Iniciaremos el estudio de este manual planteando los conceptos claves que nos permitirán definir, clasificar y analizar a los lenguajes de programación que existen en el contexto de las ciencias de la computación. Éstos conceptos nos permitirán definir nuestro propio lenguaje de programación desde cero, creando las estructuras y reglas necesarias para cada tipo, variable e instrucción del mismo.

Por último analizaremos sus propiedades y estudiaremos sus características más importantes para poder dar una implementación robusta aplicando las definiciones que iremos presentando en el desarrollo de esta introducción.

Objetivo

Comenzar el estudio formal de los lenguajes de programación brindando un primer acercamiento a la construcción básica de uno, partiendo de las definiciones de conceptos que componen un lenguaje, específicamente: sintaxis, semántica, y pragmática. Así como a la clasificación que agrupa y distingue a los diferentes lenguajes computacionales en la actualidad[1].

Planteamiento

Este capítulo inicia con el estudio de los lenguajes de programación definiendo diferentes gramáticas para generar lenguajes pequeños y familiarizarnos con los elementos que los conforman.

Es de particular interés analizar nuestro primer caso de estudio, el lenguaje `EAL`: Expresiones aritmético-lógicas proporcionando la gramática y esbozando como podemos interpretar las expresiones que pertenecen al mismo mediante la función de evaluación `eval`.

Para cerrar el capítulo se discute brevemente la clasificación de los lenguajes de programación mas importantes de la actualidad basados en las características y paradigma de programación al que pertenecen.

1 Reglas de los lenguajes

La lingüística tiene como propósito definir una serie de reglas que rijan la estructura fundamental que compone un lenguaje. Las reglas que estudian dicha composición pueden ser clasificadas en sintácticas (estructura gramatical), semánticas (significado) y pragmáticas (contexto de uso).

> **Ejercicio 1.1.** Explica que es la sintaxis de un lenguaje.
>
> "En el contexto de las ciencias de la computación, la sintaxis de un lenguaje son las reglas que definen las combinaciones de símbolos que se consideran declaraciones o expresiones correctamente estructuradas. Esto se aplica tanto a los lenguajes de programación, donde el documento representa el código fuente, como a los lenguajes de marcado, donde el documento representa datos. "[a]

[1] Para las personas que poseen una madurez matemática superior a aquella que la que un estudiante de cuarto semestre de la Facultad de Ciencias pudiera tener se recomienda encarecidamente iniciar el estudio de este manual en el capítulo 3: Sintaxis. Los temas recomendados para omitir los primeros dos capítulos del manual son: estructuras discretas, juicios lógicos, estructuras recursivas, reglas de inferencia e inducción estructural.

[a] Definición extraída de [29], [30] y [31]

Ejercicio 1.2. Explica que es la semántica de un lenguaje.

"En el contexto de las ciencias de la computación, la semántica describe los procesos que sigue una computadora al ejecutar un programa en un lenguaje específico. Esto se puede mostrar describiendo la relación entre la entrada y la salida de un programa, o una explicación de cómo se ejecutará el programa en una determinada plataforma, creando así un modelo de cálculo. "[a]

[a] Definición extraída de [20], [21] y [28]

Ejercicio 1.3. Explica que es la pragmática de un lenguaje.

"En lingüística y campos relacionados, la pragmática es el estudio de cómo el contexto contribuye al significado. "[a]

Particularmente en ciencias de la computación la pragmática se refiere a todos aquellos elementos del lenguaje que nos permiten obtener el significado de una expresión bajo un determinado contexto. Un ejemplo de ésto son las bibliotecas que nos permiten importar métodos previamente definidos, así la interpretación de una instrucción de esta biblioteca solo tiene un significado válido bajo el contexto que la importa[b].

[a] Definición extraída de [36] y [37]
[b] Extraído de [1], [5] y [12]

2 Gramáticas y semántica de expresiones

Los lenguajes (no solo los de programación) requieren de un conjunto de reglas que nos permita formar las expresiones que pertenecen a ellos y un método de interpretación que nos diga el significado de las expresiones formadas a partir de dicho conjunto de reglas.

En cursos como autómatas y lenguajes formales[2] se revisan este tipo de estructuras y las reglas que las generan conocidas como sintaxis del lenguaje pueden ser definidas mediante juicios lógicos o gramáticas.

Adicionalmente se necesita una función de evaluación que nos permita obtener el valor asociado a la expresión. A esta rama se le conoce como semántica del lenguaje[3].

Ejercicio 2.1. Define la gramática para las cadenas de la forma $(ab)^*$.

$S ::= \text{ab } S \mid \varepsilon$

Ejercicio 2.2. Define la gramática para números enteros.

$S ::= D \mid D\ S$
$D ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

[2] Conforme al plan de estudios que se imparte desde el 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México con clave de asignatura 1425.
[3] Más adelante en este curso revisaremos a profundidad las reglas semánticas y como evaluar expresiones correctas.

Ejercicio 2.3. Define la gramática para las listas de dígitos.

S ::= [DL] | [ε]
DL ::= D | D , DL
D ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Definición 2.1. Sintaxis de Expresiones aritmético-lógicas: EAL

$$e \quad ::= \quad n \quad | \quad \texttt{true} \quad | \quad \texttt{false} \quad | \quad e+e \quad | \quad e*e \quad | \quad e<e$$

$$| \quad \texttt{iszero}\ e \quad | \quad \texttt{not}\ e$$

En donde $n \in \mathbb{N}$ y las constantes true y false son cadenas del lenguaje (nótese que no son elementos de Bool aún)[a].

[a]Definición formulada de [1], [5]. [12], [26] y [39].

Definición 2.2 (Semántica de EAL)**.** Función de evaluación para el lenguaje de expresiones aritmético-lógicas.

$$[\![\cdot]\!] : \mathsf{EAL} \to \mathbb{N} \cup \text{Bool}$$

Esto quiere decir que la función de evaluación recibe una expresión del lenguaje EAL y regresa un número natural ó una constante booleana y se define de la siguiente forma:

$$\begin{array}{rcl}
[\![n]\!] & = & n \\
[\![\texttt{true}]\!] & = & \texttt{True} \\
[\![\texttt{false}]\!] & = & \texttt{False} \\
[\![e_1 + e_2]\!] & = & [\![e_1]\!] +_{\mathbb{N}} [\![e_2]\!] \\
[\![e_1 * e_2]\!] & = & [\![e_1]\!] \times_{\mathbb{N}} [\![e_2]\!] \\
[\![e_1 < e_2]\!] & = & [\![e_1]\!] <_{\mathbb{N}} [\![e_2]\!] \\
[\![\texttt{iszero}\ e]\!] & = & \mathsf{zero}_{\mathbb{N}}[\![e]\!] \\
[\![\texttt{not}\ e]\!] & = & \neg[\![e]\!]
\end{array}$$

En donde $n \in \mathbb{N}$ es un natural y las constantes `False`, `True` $\in$ Bool.

Los operadores $+_{\mathbb{N}}, \times_{\mathbb{N}}, <_{\mathbb{N}}$ y $\mathsf{zero}_{\mathbb{N}}$ representan las funciones de suma, producto, menor y el *test* de cero definidas para los naturales respectivamente, mientras que $\neg$ es la negación lógica para Bool[a].

[a]Definición extraída de [1], [5]. [12], [26] y [39].

A partir de la definición anterior se puede implementar un intérprete para expresiones del lenguaje EAL mediante una función recursiva `eval` como sigue:

```
eval n          = n
eval true       = True
eval false      = False
eval (e1 + e2)  = eval e1 + eval e2
eval (e1 * e2)  = eval e1 * eval e2
eval (e1 < e2)  = (eval e1) < (eval e2)
eval (not e)    = not (eval e)
```

```
eval (iszero e) = (eval e) == 0
```

Con esta definición ahora podemos evaluar algunas expresiones pertenecientes a nuestro lenguaje EAL.

Ejercicio 2.4. Evalúa la expresión: `not` $3 < 5 + 7$ haciendo uso de la implementación del intérprete anteriormente definido para EAL.

```
eval(not 3 < 5 + 7)  =
not(eval(3 < 5 + 7))  =
not(eval(3) < eval(5 + 7))  =
not(eval(3) <  eval(5) + eval(7)) =
not(3 < 5 + 7)  =
not(3 < 12)  =
not(True)  =
False
```

Ejercicio 2.5. Evalúa la expresión: 3 * `true` > 8 haciendo uso de la implementación del intérprete anteriormente definido para EAL.

```
eval(3 * true > 8)         =
eval(3 * true) > eval(8)   =
eval(3) * eval(true) >  8 =
3 * True > 8               =
error
```

Expresión de EAL mal formada.

Definición 2.3. Sea G una gramática libre de contexto, decimos que que G es una gramática ambigua si existe una cadena para la cual se pueda tener más de una derivación a la izquierda[a].

[a]Definición extraída de [5]. [12], [40] y [41].

Ejercicio 2.6. ¿La gramática definida para EAL es ambigua? Argumenta tu respuesta.
Sí porque podemos derivar de dos formas distintas la cadena: $3 + 4 * 5$

$$\begin{gathered} e \to \\ e + e \to \\ 3 + e \to \\ 3 + e * e \to \\ 3 + 4 * e \to \\ 3 + 4 * 5. \end{gathered}$$

$$\begin{gathered} e \to \\ e * e \to \\ e * 5 \to \\ e + e * 5 \to \\ e + 4 * 5 \to \\ 3 + 4 * 5. \end{gathered}$$

Ejercicio 2.7. ¿Cómo eliminarías la ambigüedad de esta gramática?

Con la introducción de paréntesis para marcar la precedencia de las operaciones. En el ejemplo anterior la ambigüedad queda eliminada en su totalidad:

$$(3 + (4 * 5)) \text{ ó } ((3 + 4) * 5)$$

3 Clasificación de los lenguajes de programación

En computación existen diferentes clasificaciones que nos permiten agrupar lenguajes según sus características principales, paradigma de programación al que pertenecen, tecnología involucrada en el proceso de traducción a lenguaje de máquina, etc. A continuación enlistados algunas de estas categorías:

- Lenguajes compilados, que como el nombre lo indica precisan de un compilador para traducir el código de alto nivel a instrucciones de máquina que el procesador pueda ejecutar. Generalmente suelen ser mas lentos al momento de ejecutarse que el resto de las categorías ya que requieren de un sistema sofisticado de inferencia de tipos, optimizaciones en la traducción de las instrucciones y en ocasiones entornos especiales para la ejecución de los programas como la máquina virtual de Java (*JVM* por sus siglas en inglés)[4].

- Lenguajes interpretados, en donde las instrucciones son mapeadas a su traducción equivalente en el lenguaje de máquina cuando la línea que contiene dicha instrucción se vaya a ejecutar por el procesador[5]. Generalmente son menos robustos a la hora de encontrar errores de tipo, errores en la tipografía, carácteres faltantes, etc. Sin embargo este tipo de lenguajes de programación resultan particularmente útiles por que permiten el modelado y la realización de programas mucho más rápido que los lenguajes compilados[6].

- También se pueden clasificar por paradigma como en el caso de los lenguajes funcionales que se enfocan en la especificación de un problema mediante condiciones en lugar de la solución teniéndose la siguiente definición: "La programación funcional es un paradigma de programación en el que los programas se construyen aplicando y componiendo funciones. Íntimamente relacionado con el paradigma de programación declarativa en el que las definiciones de funciones son árboles de expresiones que asignan valores a otros valores, en lugar de una secuencia de declaraciones imperativas que actualizan el estado de ejecución del programa. "[7]

- La programación imperativa contrasta con la clasificación anterior por su naturaleza secuencial, podemos encapsular sus características más importantes en la siguiente definición: "Un paradigma de programación de software que utiliza declaraciones que cambian el estado de un programa. De la misma manera que el modo imperativo en los lenguajes naturales expresa órdenes, un programa imperativo consta de órdenes que debe ejecutar la computadora. La programación imperativa se centra en describir cómo funciona un programa paso a paso[8] en lugar de descripciones de alto nivel de

[4] Información obtenida de [43], [44], [45], [46]

[5] También es posible interpretar las expresiones de un lenguaje como el valor asociado a dicha expresión una vez que se evalúa. En este curso estudiaremos este proceso en el capítulo 4: Semántica.

[6] Información obtenida de [43], [44], [45], [46]

[7] Definición extraída de [51] y [52]

[8] [[53]

sus resultados esperados. "[9]

El término se utiliza con frecuencia como un contrapunto de la programación declarativa, siendo el primero un conjunto de instrucciones para solucionar un determinado problema mientras que el segundo se centra en la especificación del problema mismo.

- Otra clasificación que es muy socorrida en la computación es la llamada orientada a objetos que modela las entidades de un programa con su identidad, métodos, variables y que brindan caracteristicas deseables como la herencia, el polimorfismo y el encapsulamiento de datos.

  Esta clasificación pertenece al paradigma imperativo dado que se trabaja estrechamente con el estado del programa en cada paso de la ejecución, sus direcciones de memoria y sus sentencias de control[10].

- Por último mencionaremos brevemente la clasificación de los lenguajes lógicos. Los programas que pertenecen a esta categoría se pueden definir de la siguiente forma: "Un programa, base de datos o base de conocimientos en un lenguaje de programación lógica es un conjunto de oraciones en forma lógica que expresan hechos y reglas sobre un dominio específico. "[11]

  Generalmente este tipo de programas se apoyan definiendo predicados y objetos a los cuales estos predicados pueden ser aplicados así como instrucciones y operadores lógicos para dar solución a los problemas planteados en este modelo.

Ejercicio 3.1. Da una implementación del algoritmo `selectionSort` en programación imperativa.

```
void swap(int *xp, int *yp){
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++){
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
```

[9] Extraído de [53], [54], [55] y [56]
[10] Definición extraída de [47], [48], [49]
[11] Definición extraída de [57]

Ejercicio 3.2. Da una implementación del algoritmo `selectionSort` en programación declarativa.

```
ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

4 Ejercicios para el lector

Ejercicio 4.1. El uso del carácter especial ';' al terminar una instrucción de programa en lenguajes de programación como `C` y `Java` ¿a qué tipo de regla corresponde (sintáctica, semántica o pragmática)?

Ejercicio 4.2. Ejemplifica la aplicación de la pragmática aplicada en el contexto de los lenguajes de programación.

Ejercicio 4.3. En el contexto de los lenguajes de programación muchas veces la semántica de una expresión es explicada en lenguaje natural en la documentación del lenguaje.

Explica por qué esto no es una semántica formal.

Ejercicio 4.4. Escribe una grámatica para el alfabeto $\Sigma = \{a, b\}$ y el lenguaje

$$L = \{a^n b^m \ j \ n{>}0, m \geq n\}$$

Ejercicio 4.5. Escribe una gramática para el alfabeto $\Sigma = \{a, b\}$ y el lenguaje

$$L = \text{cadenas palíndromas}$$

Ejercicio 4.6. Escribe una gramática para el alfabeto $\Sigma = \{(,)\}$ y el lenguaje

$$L = \text{cadenas de paréntesis balanceados}$$

Ejercicio 4.7. Evalúa la expresión del lenguaje `EAL` utilizando la función `eval`.

$$7 * 8 < 3 * 9$$

Ejercicio 4.8. Evalúa la expresión del lenguaje `EAL` utilizando la función `eval`.

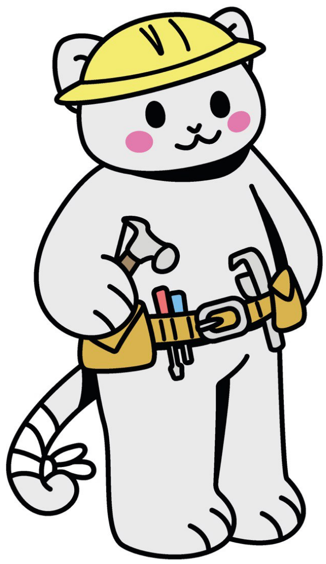
$$\texttt{not(iszero}((7 * 4) + (1 * 0)))$$

Ejercicio 4.9. Escribe el algoritmo de la búsqueda binaria en un arreglo ordenado en el paradigma de programación imperativo.

Ejercicio 4.10. Escribe el algoritmo de la búsqueda binaria en un arreglo ordenado en el paradigma de programación declarativo.

Capítulo 2

Herramientas matemáticas



Para estudiar a los lenguajes de programación es necesario definir una estructura que nos permita capturar su esencia matemática y nos proporcione un mecanismo para demostrar características y propiedades asociadas a estos.

Una manera de definir formalmente un lenguaje de programación es mediante el uso de juicios lógicos[1] para definir propiedades, pertenencia de los elementos de nuestra estructura a un conjunto, relación entre diferentes elementos de una misma estructura, etc[2].

[1]En lógica el equivalente a un juicio lógico son los predicados que denominan una característica que se cumple en una proposición.

[2]Para las personas que poseen una madurez matemática superior a aquella que la que un estudiante de cuarto semestre de la Facultad de Ciencias pudiera tener se recomienda encarecidamente iniciar el estudio de este manual en el capítulo 3: Sintaxis. Los temas recomendados para omitir los primeros dos capítulos del manual son: estructuras discretas, juicios lógicos, estructuras recursivas, reglas de inferencia e inducción estructural.

Objetivo

Estudiar las estructuras matemáticas que utilizaremos para familiarizarnos con el concepto de inducción estructural. En particular centraremos nuestra atención a estructuras recursivas como listas de tipo A y árboles binarios balanceados de tipo A[3] junto con las reglas que nos permiten construir estas estructuras.

Este tipo de mecanismos es particularmente útil para ilustrar las propiedades de los lenguajes que definiremos a lo largo de los capítulos de este manual.

Planteamiento

En este capítulo exploraremos diferentes estructuras empezando por los juicios lógicos que constituyen las entidades fundamentales sobre las cuales iniciaremos el estudio de los lenguajes. También estudiaremos brevemente estructuras recursivas como cadenas y listas para enunciar el principio de inducción que nos permite demostrar propiedades sobre dichas estructuras. Por último estudiaremos un sistema de transición para modelar los estados y reglas de evaluación de la instancia de un juego.

1 Juicios lógicos

Una parte fundamental del razonamiento matemático es la capacidad de determinar si una afirmación es verdadera o falsa basado en la información que poseemos sobre los elementos que son mencionados en dicha afirmación.

Para poder aplicar un juicio sobre algún elemento, objeto o entidad para determinar su veracidad primero debemos definir dichos elementos. A continuación vamos a introducir los conceptos necesarios para poder formalizar esta noción y como aplicarla. Esto será de utilidad para definir características, reglas de construcción y reglas de evaluación en los lenguajes de programación y modelos de computo que discutiremos a lo largo del manual.

Ejercicio 1.1. ¿Qué es una entidad matemática?

"En el lenguaje habitual de las matemáticas, una entidad es cualquier cosa que ha sido (o podría ser) definida formalmente y con la que se pueden realizar razonamientos deductivos y pruebas matemáticas. "[a]

[a]Definición formulada de [58], [59] y [60]

Ejercicio 1.2. ¿Qué es un juicio?

"En lógica matemática, un juicio es una afirmación o enunciación de una característica o propiedad sobre un objecto o elemento de un dominio. "[a]

[a]Definición formulada de [61] y [62].

Ejercicio 1.3. Proporciona un listado de juicios sobre objetos matemáticos (Puedes suponer definidos con anterioridad elementos como $\mathbb{N}$, $\mathbb{Q}$, Bool, *String*, funciones, relaciones de orden, operadores aritméticos, etc).

[3]donde A es un conjunto o una colección de elementos.

”Hola Mundo” $String$	La cadena ”Hola Mundo” es de tipo $String$.
L NP	El lenguaje L es NP.
$a > b$	el número a es más grande que el número b.
A $\mid$ B	El evento A es independiente al evento B.
True Bool	La constante *True* es de tipo Bool.
h $= g \circ f$	la función h es la composición de la función g con la función f.
3.1416... $\mathbb{I}$	El número 3.1416... es irracional
0 $\mathbb{N}$	0 es un Natural.
S(0) impar	El sucesor de 0 es impar.
$\frac{p}{q}$ $\mathbb{Q}$	La fracción $\frac{p}{q}$ es un racional.
...	

2 Reglas de inferencia

Las reglas de inferencia[4] son una representación de la aplicación de un juicio de la forma: ”Si las premisas $p_1, p_2,$ ... p_n son verdaderas, entonces c es verdadero .” Visualmente esto se denota como:

$$\frac{p_1, p_2, \ \dots \ p_n}{c}$$

Podemos observar que las reglas de inferencia están compuestas por dos secciones: una sección superior y una inferior. La sección superior es en donde se agrupan las condiciones necesarias para que la regla pueda ser aplicada (a los elementos enlistados en esta parte se les conoce como premisas o hipótesis), cada una de estas están separadas por una coma (’,’) y todas deben de cumplirse[5].

La sección inferior de una regla de inferencia enlista el resultado de la aplicación de la regla a las premisas. Ocasionalmente la regla tendrá un nombre que ayuda a identificar cuál de ellas es aplicada cuando se trabaja con derivaciones y razonamientos[6]. En esta sección de la regla puede haber uno o más elementos resultados de la aplicación pero no puede ser vacía.

Los axiomas son las reglas de inferencia que carecen de hipótesis dado que siempre serán verdad y no es necesario suponer nada para concluirlas.

Tomemos como ejemplo la definición de las cadenas binarias de tipo A (denominaremos a este tipo de dato como A*) para ilustrar las reglas de inferencia y sus partes anteriormente discutidas.

Definición 2.1. Sea $\mathrm{A} = \{0, 1\}$ un conjunto. Definimos el tipo de dato recursivo A^* (cadenas binarias de tipo A)[a] como sigue:

$$\frac{}{\varepsilon \in \mathrm{A}^*} \ \textit{(eps)} \qquad \frac{a \in \mathrm{A}, \ s \in \mathrm{A}^*}{\langle a, s \rangle \in \mathrm{A}^*} \ \textit{(tup)}$$

La regla *eps* se lee como: ”la cadena vacía es una cadena binaria de tipo A.” Nótese que esta regla es un axioma dado que carece de premisas.

[4] En esta manual utilizaremos el término ”regla de inferencia ” o ”juicio lógico ” de forma indistinta.
[5] El signo de puntuación (’,’) se puede interpretar como el operador $\wedge$ donde cada hipótesis es un argumento de dicho operador.
[6] Definición formulada de [1], [5], [9], [63], [64] y [91].

La regla *tup* se lee como: "si a es de tipo A y s es una cadena binaria de tipo A, entonces la construcción de una tupla con cabeza a y cola s es una cadena binaria de tipo A."

Por ejemplo, la cadena 10101 se representa por la 5-tupla $\langle 1, \langle 0, \langle 1, \langle 0, \langle 1.\epsilon \rangle\rangle\rangle\rangle\rangle$

[a] Definición formulada a partir de [67], [68] y [69]

3 Estructuras recursivas e inducción estructural

Ahora discutamos un tipo de estructuras particulares que se relacionan de forma íntima con los conceptos anteriormente planteados; las estructuras recursivas[7]. Estas se generan a partir de dos casos:

- La estructura está compuesta por el elemento más primitivo que puede formar parte de esta. Muchas veces este elemento constituye la clausula de escape para la recursión inducida por la definición y constituiría un axioma en su representación como regla de inferencia.
- La estructura está compuesta por una llamada a la función constructora con un elemento nuevo y la estructura misma que se desea definir como argumentos.

En el caso del tipo de dato A*, el elemento más primitivo que pertenece a esta estructura es la cadena vacía (denotada como ε) o bien una tupla con cabeza $a \in$ A y cola de tipo A*.

Este principio se aplica tanto para definir la estructura recursiva como para las funciones que se quieran aplicar sobre la misma[8]. Para ilustrar este punto definimos la función para concatenar dos elementos de A* de la siguiente forma:

Definición 3.1. Definimos la concatenación de cadenas A* denotada como: $\bullet :: (\mathrm{A}^* \rightarrow \mathrm{A}^*) \rightarrow \mathrm{A}^*$ mediante la siguiente especificación[a]:

$$\begin{array}{rcr} \text{Caso base:} & \varepsilon \bullet t = t & (\mathit{eps}) \\ \text{Constructor:} & \langle a, s \rangle \bullet t = \langle a, \langle s \bullet t \rangle\rangle & (\mathit{tup}) \end{array}$$

[a] Definición formulada a partir de [67], [68]

La característica mas importante que las estructuras recursivas poseen es el principio de inducción. Esta propiedad nos permite demostrar que una proposición o propiedad es válida para cualquier elemento de la estructura mediante su aplicación de la siguiente forma:

- Se demuestra la validez de la proposición para los elementos primitivos.
- Posteriormente se asume que la propiedad es válida para cualquier instancia.
- Por último se demuestra la propiedad aplicada para los constructores que generan instancias más grandes aplicando la hipótesis del inciso anterior.

[7] Definición formulada a partir de [91], [92] y [93]
[8] Información obtenida de [94] y [95]

Ejercicio 3.1. Utiliza la inducción estructural para demostrar:

$$t \bullet \varepsilon = t$$

Caso Base t $= \varepsilon$
$\varepsilon \bullet \varepsilon = \varepsilon$ (Por la definición de $\bullet$)
$\varepsilon = t.$

Hipótesis Inductiva $t \bullet \varepsilon = t$ para toda $t \in \mathrm{A}^*$

Paso Inductivo por demostrar $\langle a, t\rangle \bullet \varepsilon = \langle a, t\rangle$ para toda $a \in \mathrm{A}$ y $t \in \mathrm{A}^*$
$\langle a, t\rangle \bullet \varepsilon = \langle a, \langle t \bullet \varepsilon\rangle\rangle$ (Por la definición de $\bullet$)
$\langle t \bullet \varepsilon\rangle = t$ (Por la hipótesis inductiva)
$\langle a, t\rangle \bullet \varepsilon = \langle a, \langle t \bullet \varepsilon\rangle\rangle = \langle a, t\rangle$

Ejercicio 3.2. utiliza las reglas (eps) y (tup) para derivar las cadenas:

$$\langle 1, \langle 0, \langle 0, \varepsilon\rangle\rangle\rangle, \qquad \langle 1, \langle 0, \langle 0\langle 1, \varepsilon\rangle\rangle\rangle\rangle \qquad y \qquad \langle 1, \langle 1, \langle 1, \langle 0, \langle 1, \langle 0, \varepsilon\rangle\rangle\rangle\rangle\rangle\rangle$$

$$\dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \qquad \dfrac{\dfrac{}{0 \in \mathrm{A}}\,(ax) \qquad \dfrac{\dfrac{}{0 \in \mathrm{A}}\,(ax) \qquad \dfrac{}{\varepsilon \in \mathrm{A}^*}\,(eps)}{\langle 0, \varepsilon\rangle \in \mathrm{A}^*}\,(tup)}{\langle 0, \langle 0, \varepsilon\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 1, \langle 0, \langle 0, \varepsilon\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)$$

$$\dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \qquad \dfrac{\dfrac{}{0 \in \mathrm{A}}\,(ax) \qquad \dfrac{\dfrac{}{0 \in v}\,(ax) \qquad \dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \qquad \dfrac{}{\varepsilon \in \mathrm{A}}\,(eps)}{\langle 1, \varepsilon\rangle \in \mathrm{A}^*}\,(tup)}{\langle 0\langle 1, \varepsilon\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 0, \langle 0\langle 1, \varepsilon\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 1, \langle 0, \langle 0\langle 1, \varepsilon\rangle\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)$$

$$\dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \quad \dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \quad \dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \quad \dfrac{\dfrac{}{0 \in \mathrm{A}}\,(ax) \quad \dfrac{\dfrac{}{1 \in \mathrm{A}}\,(ax) \quad \dfrac{\dfrac{}{0 \in \mathrm{A}}\,(ax) \quad \dfrac{}{\varepsilon \in \mathrm{A}^*}\,(eps)}{\langle 0, \varepsilon\rangle \in \mathrm{A}^*}\,(tup)}{\langle 1, \langle 0, \varepsilon\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 0, \langle 1, \langle 0, \varepsilon\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 1, \langle 0, \langle 1, \langle 0, \varepsilon\rangle\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 1, \langle 1, \langle 0, \langle 1, \langle 0, \varepsilon\rangle\rangle\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)}{\langle 1, \langle 1, \langle 1, \langle 0, \langle 1, \langle 0, \varepsilon\rangle\rangle\rangle\rangle\rangle\rangle \in \mathrm{A}^*}\,(tup)$$

Definición 3.2. Definimos la función para contar caracteres[a] en una cadena denotada por $n_c :: \mathrm{A}^* \to \mathrm{Int}$ como sigue:

$$n_c(\langle a, t\rangle) = \begin{cases} n_c(\varepsilon) = 0 & \\ 1 + n_c(t) & \text{Sí } a = \mathrm{c} \\ n_c(t) & \text{Sí } a \neq c \end{cases}$$

[a]Definición formulada a partir de [67], [68] y [69]

Ejercicio 3.3. Utiliza la inducción estructural para demostrar

$$n_c(s \bullet t) = n_c(s) + n_c(t)$$

Caso Base $s = \varepsilon$
$n_c(\varepsilon \bullet t) = n_c(t)$ (Por la definición de $\bullet$)

$= 0 + n_c(t) = n_c(\varepsilon) + n_c(t)$ (Por la definición de n_c)
$= n_c(s) + n_c(t)$

Hipótesis Inductiva $n_c(s \bullet t) = n_c(s) + n_c(t)$

Paso Inductivo Por demostrar para $s = \langle a, s' \rangle$
$n_c(\langle a, s' \rangle \bullet t) = n_c(\langle a, s' \bullet t \rangle)$ (Por la definición de $\bullet$)

Caso 1 $a = c$ entonces tenemos
$= n_c(\langle a, s' \bullet t \rangle) = 1 + n_c(s' \bullet t)$ (Por la definición de n_c)
$= 1 + n_c(s') + n_c(t)$ (Por la hipótesis inductiva)
$= (1 + n_c(s')) + n_c(t) = n_c(\langle a, s' \rangle) + n_c(t)$ (Por la definición de n_c)

Caso 2 $a \neq c$ entonces tenemos
$= n_c(\langle a, s' \bullet t \rangle) = n_c(s' \bullet t)$ (Por la definición de n_c)
$= n_c(s') + n_c(t)$ (Por la hipótesis inductiva)
$= n_c(\langle a, s' \rangle) + n_c(t)$ (Por la definición de n_c)

Definición 3.3. Definimos las funciones $rev :: \mathrm{A}^* \to \mathrm{A}^*$ y $len :: \mathrm{A}^* \to \mathrm{Int}$ para obtener la reversa y la longitud[a] de los elementos de A^* respectivamente como:

$$rev(\varepsilon) = \epsilon \qquad rev(\langle a, s \rangle) = rev(s) \bullet \langle a, \varepsilon \rangle$$

$$len(\varepsilon) = \epsilon \qquad len(\langle a, s \rangle) = 1 \ + \ len(s)$$

[a]Definición formulada a partir de [67] y [70]

Ejercicio 3.4. Utiliza inducción estructural para demostrar

$$len(s \bullet t) = len(s) + len(t)$$

Caso Base $s = \varepsilon$
$len(\varepsilon \bullet t) = len(t)$ (Por la definición de $\bullet$)
$= 0 + len(t) = len(\varepsilon) + len(t)$ (Por la definición de len)
$= len(s) + len(t)$

Hipótesis inductiva $len(s \bullet t) = len(s) + len(t)$

Paso Inductivo Por demostrar para $s = \langle a, s' \rangle$
$len(\langle a, s' \rangle \bullet t) = len(\langle a, \langle s' \bullet t \rangle \rangle)$ (Por la definición $\bullet$)
$= 1 + len(s' \bullet t) = 1 + len(s') + len(t)$ (Por la hipótesis inductiva)
$= (1 + len(s')) + len(t) = len(\langle a, s' \rangle) + len(t)$

Ejercicio 3.5. Utiliza inducción estructural para demostrar

$$len(rev(s)) = len(s)$$

Caso Base $s = \varepsilon$
$len(rev(\varepsilon)) = len(\varepsilon)$ (Por la definición de rev)
$len(rev(s)) = len(rev(\varepsilon)) = len(\varepsilon) = len(s)$.

Hipótesis Inductiva $len(rev(s)) = len(s)$

Paso Inductivo $s = \langle a, s\rangle$
$len(rev(\langle a, s\rangle)) = len(rev(s) \bullet \langle a, \varepsilon\rangle)$ (Por la definición de rev)
$len(rev(s) \bullet \langle a, \varepsilon\rangle) = len(rev(s)) + len(\langle a, \varepsilon\rangle)$ (Por la propiedad anterior)
$len(rev(s)) + 1 + len(\langle\varepsilon\rangle) = len(rev(s)) + 1 + 0$
$1 + len(rev(s)) = 1 + len(s)$ (Por la hipótesis inductiva)
$len(\langle a, s\rangle)$ (Por definición de len)

Definición 3.4. Definimos los árboles binarios de tipo A con las siguientes reglas[a]:

1. $\emptyset$ es un árbol de tipo A.
2. Sí t_1 y t_2 son árboles de tipo A y $a \in$A entonces $node(a, t_1, t_2)$ es un árbol binario de tipo A.
3. Son todas.

Decimos que un árbol binario de tipo A y altura k es balanceado sí se cumple alguna de las siguientes dos condiciones:

1. t = $\emptyset$ y $k = 0$
2. t = $node(a, t_1, t_2)$ de altura k donde t_1 y t_2 son árboles binarios balanceados de tipo A y altura k-1.

Definimos la función que cuenta el número de nodos denotada $n_n(t)$ como:

$$\begin{aligned} n_n(\emptyset) &= 0 \\ n_n(node(a, t_1, t_2)) &= 1 + n_n(t_1) + n_n(t_2) \end{aligned}$$

[a] Definición formulada a partir de [67], [68] y [69]

Ejercicio 3.6. Demuestra utilizando inducción estructural que si t es un árbol binario de tipo A con altura k entonces $n_n(t) = 2^k - 1$

Caso Base $t = \emptyset$, $k = 0$

$n_n(\emptyset) = 2^0 - 1 = 1 - 1 = 0.$

Hipótesis Inductiva $n_n(t) = 2^k - 1$ para todo árbol binario de tipo A t donde k es la altura del mismo.

Paso Inductivo Por demostrar para $t = node(a, t_1, t_2)$ donde t_1 y t_2 son árboles binarios balanceados de altura $k - 1$ se cumple que $n_n(node(a, t_1, t_2)) = 2^k - 1$ donde k es la altura asociada al árbol t.

$n_n(node(a, t_1, t_2)) = 1 + n_n(t_1) + n_n(t_2)$ (Por definición de $n_n(t)$)
$= 1 + (2^{k-1} - 1) + (2^{k-1} - 1)$ (Por la hipótesis inductiva)
$= 1 + 2(2^{k-1} - 1) = 2^k - 2 + 1 = 2^k - 1$

4 Sistemas de transición

En el contexto de las ciencias de la computación los sistemas de transición resultan particularmente útiles para obtener cualquier configuración de un programa en cada paso de su ejecución en forma de estados. En particular tendremos dos estados especiales para denotar la configuración inicial y la configuración final. Adicionalmente nos apoyaremos de un conjunto de reglas que nos permiten aplicar instrucciones y operadores para movernos de una configuración a otra[9].

Obtener todas las configuraciones posibles partiendo del estado inicial al estado final nos facilitará hacer el análisis de complejidad en tiempo y en espacio de nuestro programa.

Estructuras similares se han estudiado en cursos como autómatas y lenguajes formales[10] donde adicionalmente se define el alfabeto y se considera la lectura de un carácter para obtener una configuración distinta. En este caso omitiremos estos elementos y nos concentraremos únicamente en los estados y las reglas de transición.

Ejercicio 4.1. Supón que se quiere modelar una partida de pimpón donde hay 2 jugadores A y B (el jugador A siempre empieza primero), y un marcador que cuenta el número de puntos de A y B representado como: jugador en turno, puntuación de A, puntuación de B.

De tal forma que cada estado indica la información del juego hasta ese punto. Por ejemplo si el jugador B anota, el estado se modelaría como (B,X,Y) $\rightarrow$ (B,X,Y+1) ya que el jugador que anota un punto vuelve a tirar la pelota en el siguiente turno.

La partida acaba cuando alguno de los dos logra anotar 10 puntos.

En base a la especificación dada anteriormente contesta lo siguiente:

1. Define formalmente el conjunto de estados (indica estados iniciales con la letra "I", finales con la letra "F" y los intermedios con "Γ"):

$$\mathrm{I} = \{(\mathrm{A}, 0, 0)\}$$

$$\Gamma = \{\mathrm{C}, \mathrm{X}, \mathrm{Y}\} \text{ donde } \mathrm{C} \in \{\mathrm{A}, \mathrm{B}\} \text{ y } 0 \leq \mathrm{X}, \mathrm{Y} \leq 10$$

$$\mathrm{F} = \{\mathrm{C}, \mathrm{X}, \mathrm{Y}\} \text{ donde } \mathrm{C} \in \{\mathrm{A}, \mathrm{B}\} \text{ y } X \text{ o } Y = 10$$

2. Define la función de transición (enlista cuales son los casos y como se comporta la función en cada uno de estos):

   Sí el jugador A puntúa:

$$f((\mathrm{A}, \mathrm{X}, \mathrm{Y})) = (\mathrm{A}, \mathrm{X}+1, \mathrm{Y})$$

   Sí el jugador B puntúa:

$$f((\mathrm{B}, \mathrm{X}, \mathrm{Y})) = (\mathrm{B}, \mathrm{X}, \mathrm{Y}+1)$$

   Sí el jugador A no puntúa:

$$f((\mathrm{A}, \mathrm{X}, \mathrm{Y})) = (\mathrm{B}, \mathrm{X}, \mathrm{Y})$$

[9] Para el enfoque que seguimos en este manual definiremos las reglas de nuestros sistemas de transición como juicios lógicos.

[10] Conforme al plan de estudios que se imparte desde el 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México con clave de asignatura 1425.

Sí el jugador B no puntúa:

$$f((\mathrm{B}, \mathrm{X}, \mathrm{Y})) = (\mathrm{A}, \mathrm{X}, \mathrm{Y})$$

Sí cualquier jugador hace un saque despúes de anotar:

$$f((\mathrm{J}, \mathrm{X}, \mathrm{Y})) = (\mathrm{J}, \mathrm{X}, \mathrm{Y})$$

3. Define con reglas de inferencia la función de transición escrita en el inciso anterior:

$$\frac{(\mathrm{A}, \mathrm{X}, \mathrm{Y})\ \text{Estado}, \quad (\mathrm{A}, \mathrm{X}+1, \mathrm{Y})\ \text{Estado}}{(\mathrm{A}, \mathrm{X}, \mathrm{Y}) \to (\mathrm{A}, \mathrm{X}+1, \mathrm{Y})\ \text{Estado}}\ (\mathrm{A}+) \qquad \frac{(\mathrm{B}, \mathrm{X}, \mathrm{Y})\ \text{Estado}, \quad (\mathrm{B}, \mathrm{X}, \mathrm{Y}+1)\ \text{Estado}}{(\mathrm{B}, \mathrm{X}, \mathrm{Y}) \to (\mathrm{B}, \mathrm{X}, \mathrm{Y}+1)\ \text{Estado}}\ (\mathrm{B}+)$$

$$\frac{(\mathrm{A}, \mathrm{X}, \mathrm{Y})\ \text{Estado}, \quad (\mathrm{B}, \mathrm{X}, \mathrm{Y})\ \text{Estado}}{(\mathrm{A}, \mathrm{X}, \mathrm{Y}) \to (\mathrm{B}, \mathrm{X}, \mathrm{Y})\ \text{Estado}}\ (\mathrm{A}-) \qquad \frac{(\mathrm{B}, \mathrm{X}, \mathrm{Y})\ \text{Estado}, \quad (\mathrm{A}, \mathrm{X}, \mathrm{Y})\ \text{Estado}}{(\mathrm{B}, \mathrm{X}, \mathrm{Y}) \to (\mathrm{A}, \mathrm{X}, \mathrm{Y})\ \text{Estado}}\ (\mathrm{B}-)$$

$$\frac{(\mathrm{J}, \mathrm{X}, \mathrm{Y})\ \text{Estado}}{(\mathrm{J}, \mathrm{X}, \mathrm{Y}) \to (\mathrm{J}, \mathrm{X}, \mathrm{Y})\ \text{Estado}}\ (saque)$$

4. Muestra una ejecución donde gane el jugador B especificando cada estado desde el inicial hasta el final:

$$\begin{gathered}
(\mathrm{A}, 0, 0) \to (\mathrm{A}, 1, 0) \to (\mathrm{A}, 1, 0) \to (\mathrm{B}, 1, 1) \to (\mathrm{B}, 1, 1) \to \\
(\mathrm{B}, 1, 2) \to (\mathrm{B}, 1, 2) \to (\mathrm{B}, 1, 3) \to (\mathrm{B}, 1, 3) \to (\mathrm{B}, 1, 4) \to \\
(\mathrm{B}, 1, 4) \to (\mathrm{B}, 1, 5) \to (\mathrm{B}, 1, 5) \to (\mathrm{A}, 2, 5) \to (\mathrm{A}, 2, 5) \to \\
(\mathrm{B}, 2, 6) \to (\mathrm{B}, 2, 6) \to (\mathrm{B}, 2, 7) \to (\mathrm{B}, 2, 7) \to (\mathrm{A}, 3, 7) \to \\
(\mathrm{A}, 3, 7) \to (\mathrm{A}, 4, 7) \to (\mathrm{A}, 4, 7) \to (\mathrm{B}, 4, 8) \to (\mathrm{B}, 4, 8) \to \\
(\mathrm{A}, 5, 8) \to (\mathrm{A}, 5, 8) \to (\mathrm{B}, 5, 9) \to (\mathrm{B}, 5, 9) \to (\mathrm{B}, 5, 10)
\end{gathered}$$

En donde la secuencia se puede interpretar como:

”El jugador A hizo el saque inicial y el marcador muestra A con 0 y B con 0 puntos ”, después ”El jugador A anotó y el marcador muestra A con 1 punto y B con 0 puntos ”, después ”El jugador A hizo el saque inicial pues anotó en el turno anterior y el marcador muestra A con 1 punto y B con 0 puntos ”, después ”El jugador B anotó y el marcador muestra A con 1 punto y B con 1 punto ”, después ”El jugador B hizo el saque inicial pues anotó en el turno anterior y el marcador muestra A con 1 punto y B con 1 punto ”, después ”El jugador B anotó y el marcador muestra A con 1 punto y B con 2 puntos... ”

5 Ejercicios para el lector

Ejercicio 5.1. Escribe el árbol de derivación para la siguiente cadena utilizando la definición 2.1 para el tipo de dato A*:

$$\langle 1, \langle 0, \varepsilon \rangle \rangle$$

Ejercicio 5.2. Escribe el árbol de derivación para la siguiente cadena utilizando la definición 2.1 para el tipo de dato A*:

$$\langle 1, \langle 1, \langle 1 \langle 0, \varepsilon \rangle \rangle \rangle \rangle$$

Ejercicio 5.3. Escribe el árbol de derivación para la siguiente cadena utilizando la definición 2.1 para el tipo de dato A*:

$$\langle 1, \langle 0, \langle 0, \langle 0, \langle 0, \langle 1, \varepsilon \rangle \rangle \rangle \rangle \rangle \rangle$$

Definición 5.1. Definimos la función $map :: (\mathrm{A} \rightarrow \mathrm{A}) \rightarrow \mathrm{A}^* \rightarrow \mathrm{A}^*$ que recibe una función f y una cadena s de la siguiente forma[a]:

$$\begin{array}{rcl} map\ f\ \varepsilon & = & \varepsilon \\ map\ f\ \langle a, s \rangle & = & \langle f\,\mathrm{a}, map\ f\ s \rangle \end{array}$$

[a]Definición formulada a partir de [67], [68] y [69]

Ejercicio 5.4. Demuestra utilizando la inducción estructural sobre las cadenas binarias de tipo A* que:

$$len(map\ \ f\ \langle a, s \rangle) = len(\langle a, s \rangle)$$

Ejercicio 5.5. Demuestra utilizando la inducción estructural sobre el tipo de dato A* que:

$$map\ f\ \langle a, s \rangle \bullet \langle b, t \rangle = map\ f\ \langle a, s \rangle \bullet map\ f\ \langle b, t \rangle$$

Ejercicio 5.6. Demuestra utilizando la inducción estructural sobre árboles binarios balanceados de tipo A que se cumple la siguiente afirmación:

Sea t un árbol binario balanceado de tipo A y altura h entonces

$$n_n(t) \leq 2^{h+1} - 1$$

Ejercicio 5.7. Demuestra utilizando la inducción estructural sobre árboles binarios balanceados de tipo A que se cumple la siguiente afirmación:

Sea t un árbol binario balanceado de tipo A y altura h entonces

$$n_i(t) = \frac{n_n(t) - 1}{2}$$

donde $n_i(t)$ es el número de nodos internos en un árbol y $n_n(t)$ es el número de total de nodos.

Ejercicio 5.8. De acuerdo a la máquina de transición definida en el ejercicio 3.1 ¿los jugadores A y B pueden empatar?

Justifica tu respuesta

Ejercicio 5.9. De acuerdo con el sistema de transición definido en el ejercicio 3.1 muestra una ejecución en la que el jugador A gane.

Ejercicio 5.10. Suponga que se necesita definir un lenguaje que permita controlar un robot con movimientos y funcionalidades muy simples. El robot se mueve sobre una cuadrícula siguiendo las instrucciones especificadas por el programa.

Al inicio el robot se encuentra en la coordenada (0,0) y viendo hacia el norte. El programa consiste en una secuencia posiblemente vacía de los comandos `move` y `turn` separados por punto y coma (';'), donde cada comando tiene el siguiente funcionamiento[a]:

- `turn` hace que el robot dé un giro de 90 grados en el sentido de las manecillas del reloj.

- `move` provoca que el robot avance una casilla en la dirección hacía la que está viendo.

Donde un ejemplo de un programa válido puede ser el siguiente:

$$\texttt{move}; \texttt{turn}; \texttt{move}; \texttt{turn}; \texttt{turn}; \texttt{turn}; \texttt{move}$$

Al final de la ejecución programa el robot termina en la casilla (2,1). La primera entrada de la coordenada indica la posición vertical mientras que la segunda es la posición horizontal.

Con la especificación discutida anteriormente contesta los siguientes puntos:

1. Determina el conjunto de estados.
2. Identifica los estados iniciales y finales del sistema de transición.
3. Define la función de transición $\to_R$ que indique como se debe transitar entre los estados del sistema.
4. Muestra paso a paso la ejecución del programa:

   $$\texttt{move}; \texttt{turn}; \texttt{move}; \texttt{turn}; \texttt{turn}; \texttt{turn}; \texttt{move}$$

   Utilizando la relación $\to_R$ y partiendo del estado inicial.

[a] ejercicio extraído de [107]

Para finalizar esta sección vamos a revisar dos ejercicios prácticos que implementaremos en `Haskell`. Estos ejercicios combinan las definiciones de las funciones que hemos estudiado hasta este punto para listas y que ejemplifican las propiedades recursivas de estas estructuras en un lenguaje de programación funcional[11].

Ejercicio 5.11. La validación del número de una tarjeta de crédito se hace implementando un algoritmo `checkSum` obteniendo información de los dígitos que la componen. Para eso implementaremos nuestro propio validador de tarjetas de crédito como sigue:

Los dígitos que están una posición impar (empezando por el índice 0 de derecha a izquierda) se deben duplicar.

Posteriormente se suman todos los dígitos de los números que componen la tarjeta (aquellos números que fueron duplicados deben de ser sumados junto con el resto que no se modificaron).

Por último se aplica el módulo 10 al resultado de la suma, si el resultado es diferente de 0 entonces la tarjeta es inválida, en caso contrario es válida.

1. Implementa la función `getDigits :: Int -> [Int]`

   Por ejemplo, la tarjeta

   $$1348\ 1548\ 9998\ 6535$$

   dará como resultado la lista: `[1,3,4,8,1,5,4,8,9,9,9,8,6,5,3,5]`

[11] Los ejercicios 4.11 y 4.12 fueron extraídos de [75].

(asumimos que la entrada son solo los dígitos de la tarjeta SIN separación).

2. Implementa la función `reverseDigits :: [Int] -> [Int]`

   Esta función obtiene la reversa de la lista de los dígitos obtenidos en el paso anterior.

3. Implementa la función `duplicateOddPositionDigits :: [Int] -> [Int]`

   Por ejemplo: la lista `[1,3,4,8,1,5,4,8,9,9,9,8,6,5,3,5]` solo duplicará los números que estén en una posición impar obteniendo como resultado: `[1,6,4,16,1,10,4,16,9,18,9,16,6,10,6,5]`.

4. Implementa la función `toDigits :: Int -> [Int]`

   El objetivo de la implementación de esta función es el de obtener los dígitos de los números mayores o iguales a 10, de tal forma que si tenemos como parámetro de entrada para esta función a la lista `[1,6,4,16,1,10,4,16,9,18,9,16,6,10,6,5]` obtengamos la siguiente como resultado `[1,6,4,1,6,1,0,4,1,6,9,1,8,9,1,6,1,0,6,5]`

5. Implementa la función `sumDigitsInList :: [Int] -> Int`

   Esta función simplemente suma los dígitos contenidos en la lista (puedes utilizar la función sum provista por `GHCi`).

6. Implementa la función `checkSumCreditCard :: Int -> Bool`

   Recuerda que el criterio de validez es que la suma de los dígitos filtrados de la tarjeta módulo 10 sea igual a 0.

Ejercicio 5.12. Las torres de Hanoi son un un rompecabezas clásico cuya solución puede ser escrita de forma recursiva. Los discos de diferentes tamaños se apilan del más grande al más pequeño y el objetivo es moverlos del pivote A al pivote C utilizando un pivote auxiliar B con $n = 4$.

Las reglas son las siguientes:

1. Ningún disco puede estar encima de un disco mas pequeño.
2. Por movimiento solo es válido el desplazamiento de un disco hacía otro pivote.

Se definen los siguientes tipos para implementar la solución al *puzzle* de la siguiente forma:

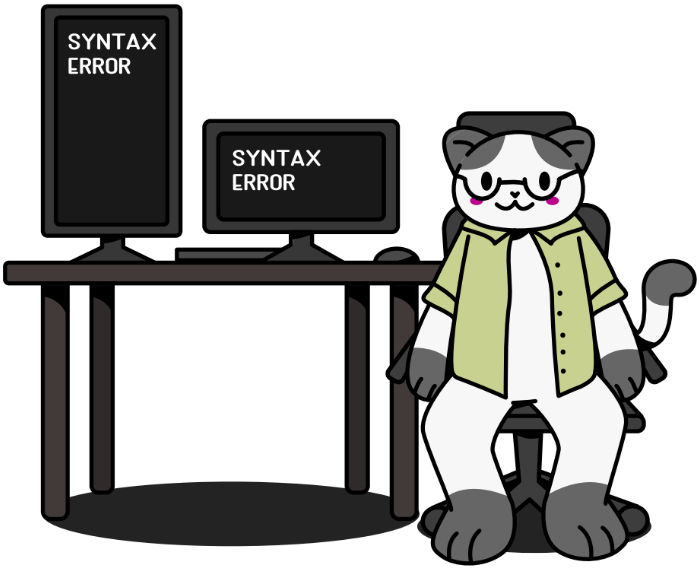
```
type Peg  = String
type move = (Peg, Peg)
```

Implementa la función `Hanoi :: Int -> Peg -> Peg -> Peg -> [move]`
Que dada el número de discos y los pivotes regrese una lista con los movimientos para mover los discos del primer pivote al pivote objetivo, por ejemplo:

```
Hanoi 2 ''A'' ''B'' ''C'' = [(''A'',''C''),(''A'',''B''),(''C'',''B'')]
```

Capítulo 3

Sintaxis



La sintaxis de un lenguaje de programación permite delimitar el conjunto de las cadenas que pertenecen a este y al mismo tiempo constituye una herramienta de razonamiento para demostrar propiedades utilizando el principio de inducción aplicado a los juicios o reglas que lo definen.

De forma similar la sintaxis marca la pauta para la definición de funciones como la función de evaluación para obtener el valor asociado a una expresión correctamente construida de un lenguaje de programación y al mismo tiempo será en esta misma estructura donde nos apoyaremos para definir los procesos de análisis para determinar la validez y el tipo de dichas expresiones.

Para poder comenzar a estudiar los niveles sintácticos de un lenguaje de programación utilizaremos la definición de un nuevo lenguaje el cuál extenderemos con instrucciones a lo largo del desarrollo del manual y lo denominaremos como EAB (expresiones aritmético-booleanas). A partir de la definición de este lenguaje mediante la gramática para generar expresiones y las reglas de inferencia equivalentes abordaremos la construcción y diferencias que existen entre la sintaxis concreta y sintaxis abstracta que se introducen para EAB.

Objetivo

Definir el lenguaje EAB junto con sus diferentes niveles sintácticos, mostrando las principales diferencias que existen entre la sintaxis concreta y abstracta así como estudiar cómo se relacionan proporcionando las reglas para implementar el analizador sintáctico de este mismo.

Planteamiento

Iniciaremos el estudio de este capítulo brindando las reglas de la sintaxis de EAB[1] junto con la definición de los nuevos niveles sintácticos; la sintaxis concreta y la sintaxis abstracta, misma que introduce una nueva estructura conocida como árbol de sintaxis abstracta que captura la jerarquía de operación en las instrucciones que se definen en su representación como cadena válida del lenguaje derivada a partir de las reglas definidas en la sintaxis concreta. En particular estudiaremos el proceso mediante el cual una expresión de la sintaxis concreta se relaciona con un árbol de sintaxis abstracta conocido como análisis sintáctico.

En este nuevo lenguaje vamos a ejemplificar como estos dos niveles sintácticos se relacionan entre sí extendiendo su definición para soportar a los operadores `if` y `let`.
Posteriormente explicaremos la relación que el operador `let` introduce con las variables ligadas, variables libres, el alcance de una variable y estudiaremos los conceptos de sustitución y α-equivalencia.

1 Sintaxis concreta

La sintaxis concreta se relaciona con la representación gráfica de las cadenas del lenguaje y comúnmente es denotada por una gramática libre de contexto. Esta gramática está pensada para el usuario final, es decir una persona.
A esta capa se le denomina de "alto nivel" dado que es un ser humano y no una computadora quien se requiere pueda leer y escribir programas siguiendo las reglas de construcción del lenguaje en cuestión[2]. Vamos a denotar las expresiones del lenguaje con la letra "E".
Definimos entonces el lenguaje EAB[3] con el siguiente conjunto de juicios:

Definición 1.1. Sintaxis concreta en forma de juicios lógicos para cada regla en nuestra gramática para EAB[a]:

$$\frac{}{0 \text{ D}} \quad \frac{}{1 \text{ D}} \quad \dots \quad \frac{}{9 \text{ D}}$$

[1] Este lenguaje es una definición independiente a EAL vista en el capítulo 1: Introducción, dado que iremos extendiendo y agregando nuevas instrucciones durante el desarrollo de los siguientes capítulos.

[2] Definición formulada a partir de [1], [2], [5], [12] y [76]

[3] Iniciaremos definiendo las instrucciones aritméticas primero y posteriormente agregaremos instrucciones lógico-booleanas con el operador `if`.

$$\frac{d\ \mathrm{D}}{d\ \mathrm{N}} \qquad \frac{n\ \mathrm{N} \qquad d\ \mathrm{D}}{nd\ \mathrm{N}}$$

$$\frac{n\ \mathrm{N}}{n\ \mathrm{F}} \qquad \frac{e\ \mathrm{E}}{(e)\ \mathrm{F}}$$

$$\frac{f\ \mathrm{F}}{\mathrm{F\ T}} \qquad \frac{t\ \mathrm{T} \qquad f\ \mathrm{F}}{t * f\ \mathrm{T}}$$

$$\frac{t\ \mathrm{T}}{t\ \mathrm{E}} \qquad \frac{e\ \mathrm{E} \qquad t\ \mathrm{T}}{e + t \quad \mathrm{E}}$$

[a]Esta definición fue acuñada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que nos interesa ejemplificar en este capítulo.

2 Sintaxis abstracta

La sintaxis abstracta por otra parte como su nombre lo indica se relaciona con la abstracción de la estructura del lenguaje y comúnmente es definida mediante reglas de inferencia para construir un árbol sintáctico donde cada nodo corresponde a una instrucción o operador y los hijos a los parámetros que recibe dicha instrucción[4].

Las hojas de los árboles sintácticos corresponden a los tipos primitivos del lenguaje (enteros, booleanos, caracteres, etc.). En este capítulo vamos a denotar a un árbol de sintaxis abstracta como "*ASA*".

Los árboles sintácticos son útiles para definir la jerarquía de las operaciones a realizar durante la evaluación de una expresión bien formada de un lenguaje y eliminar la ambigüedad del mismo dado que un árbol de sintaxis abstracta es único sin importar la representación que una expresión tenga en sintaxis concreta.

Definición 2.1. Sintaxis abstracta de EAB[a]

$$\frac{n \in \mathbb{N}}{num[n]\ \mathrm{ASA}} \qquad \frac{t_1\ ASA \qquad t_2\ ASA}{sum(t_1, t_2)\ ASA} \qquad \frac{t_1\ ASA \qquad t_2\ ASA}{prod(t_1, t_2)\ ASA}$$

En donde el predicado $t_1\ ASA$ se lee como: " t_1 es un árbol de sintaxis abstracta".

Es importante notar que los árboles de sintaxis abstracta aunque similares a la expresión en sintaxis concreta que representan corresponden a una categoría distinta de los niveles sintácticos para EAB.

[a]Esta definición fue acuñada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que nos interesa ejemplificar en este capítulo.

[4]Definición formulada a partir de [5], [12], [77], [78] y [79].

2.1 Analizador sintáctico

Al proceso mediante el cual se obtiene el árbol de sintaxis abstracta de una expresión en sintaxis concreta se le conoce como análisis sintáctico. Es importante notar que no siempre será posible encontrar el árbol de sintaxis abstracta (por ejemplo si la expresión que analizamos no es derivable de las reglas de la sintaxis concreta).

De tal forma que el objetivo ahora será definir formalmente la relación entre ambos niveles donde el siguiente juicio es válido:[5]

$$e\ E \longleftrightarrow a\ ASA$$

Esta relación se interpreta como "a es el árbol de sintaxis abstracta asociado a la expresión e". Definimos entonces las reglas para implementarla en EAB como sigue:

Definición 2.2. Analizador síntactico para expresiones de lenguaje de EAB[a]

$$\frac{e\ \mathsf{N} \quad e \in \mathbb{N}}{e\ \mathsf{F} \longleftrightarrow num[e]\ ASA}\ \textit{(num)} \qquad \frac{e\ \mathsf{E} \longleftrightarrow a\ ASA}{(e)\ \mathsf{F} \longleftrightarrow a\ ASA}\ \textit{(parentE)}$$

$$\frac{e\ \mathsf{F} \longleftrightarrow a\ ASA}{e\ \mathsf{T} \longleftrightarrow a\ ASA}\ \textit{(fact)} \qquad \frac{e_1\ \mathsf{T} \longleftrightarrow a_1\ ASA \quad e_2\ \mathsf{F} \longleftrightarrow a_2\ ASA}{e_1 * e_2\ \mathsf{T} \longleftrightarrow prod(a_1, a_2)\ ASA}\ \textit{(prod)}$$

$$\frac{e\ \mathsf{T} \longleftrightarrow a\ ASA}{e\ \mathsf{E} \longleftrightarrow a\ ASA}\ \textit{(exprE)} \qquad \frac{e_1\ \mathsf{E} \longleftrightarrow a_1\ ASA \quad e_2\ \mathsf{T} \longleftrightarrow a_2\ ASA}{e_1 + e_2\ \mathsf{T} \longleftrightarrow sum(a_1, a_2)\ ASA}\ \textit{(sum)}$$

[a]Definición formulada a partir de [1], [2], [5] y [12]

Ahora queremos extender EAB agregando instrucciones booleanas que permitan tener condicionales en nuestras expresiones de lenguaje. Es decir queremos agregar las constantes booleanas `True` y `False`, el operador `if` ... `then` simple y el operador `if`... `then` ... `else` ... así como parentizado de expresiones para evitar condiciones de "else colgante"[6] y ambigüedad (esta regla puede ser modelada encerrando entre paréntesis la expresión inmediata a evaluar en la instrucción `then`).

Ejercicio 2.1. Define la sintaxis concreta para extender nuestro lenguaje EAB con las instrucciones lógicas previamente discutidas[a]

$$\frac{}{\texttt{True}\ \mathrm{B}} \qquad \frac{}{\texttt{False}\ \mathrm{B}}$$

$$\frac{c\ \mathrm{B}\ t\ \mathrm{LE}\ c\ \mathrm{E}}{\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e\ E} \qquad \frac{c\ \mathrm{B}\ t\ \mathrm{LE}}{\texttt{if}\ c\ \texttt{then}\ t\ \texttt{E}} \qquad \frac{x\ \mathrm{B}}{x\ \mathrm{LE}}$$

$$\frac{e\ \mathrm{E}}{(e)\ \mathrm{LE}} \qquad \frac{e\ \mathrm{LE}}{e\ \mathrm{E}}$$

[a]Definición formulada a partir de [1], [5], [8] y [12]

[5]La definición de esta relación fue formulada a partir de [1], [5] y [12], se refiere a las siguientes publicaciones para un estudio en profundidad sobre los niveles sintácticos de los lenguajes de programación: [76], [77], [78] y [79]

[6]Éste problema es conocido en inglés como "*dangling else*" recurrente en la implementación de *parsers* en el que una cláusula else opcional en una declaración if-then(-else) da como resultado que los condicionales anidados sean ambiguos. [90]

Ejercicio 2.2. Define los juicios que componen la sintaxis abstracta para el operador `If`[a]

$$\frac{\texttt{True}\ \mathrm{B}}{T\ ASA} \quad \frac{\texttt{False}\ \mathrm{B}}{F\ ASA}$$

$$\frac{c\ ASA\ \ t\ ASA\ \ e\ ASA}{\mathit{if}(c\ t\ e)\ ASA} \quad \frac{c\ ASA\ \ t\ \mathrm{ASA}}{\mathit{if}(c\ t)\ ASA}$$

[a]Definición formulada a partir de [1], [5], [8] y [12]

Ejercicio 2.3. Define las reglas del analizador sintáctico para el operador `If`[a].

$$\frac{}{\texttt{True}\ \mathrm{B} \longleftrightarrow \mathrm{T}\ ASA}\ \mathit{(true)} \quad \frac{}{\texttt{False}\ \mathrm{B} \longleftrightarrow \mathrm{F}\ ASA}\ \mathit{(false)}$$

$$\frac{c\ \mathrm{B} \longleftrightarrow c\text{'}\ ASA \quad t\ \mathrm{LE} \longleftrightarrow t'\ \mathrm{ASA} \quad e\ \mathrm{E} \longleftrightarrow \mathrm{e'}\ ASA}{\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e\ \texttt{E} \longleftrightarrow \mathit{if}(c\text{'}\ t\text{'}\ e\text{'})\ ASA}\ (if_e)$$

$$\frac{c\ \mathrm{B} \longleftrightarrow c\text{'}\ ASA \quad t\ \mathrm{LE} \longleftrightarrow t\text{'}\ ASA}{\texttt{if(}c\ \texttt{then}\ t\texttt{)}\ \texttt{E} \longleftrightarrow \mathit{if}(c\text{'}\ t\text{'})\ ASA}\ (if_s)$$

$$\frac{c\ \mathrm{E} \longleftrightarrow e\text{'}\ ASA}{(e)\ \mathrm{LE} \longleftrightarrow e\text{'}\ ASA}\ \mathit{(parentB)} \quad \frac{c\ \mathrm{B} \longleftrightarrow c\text{'}\ ASA}{c\ \mathrm{LE} \longleftrightarrow c\text{'}\ ASA}\ \mathit{(bool)} \quad \frac{e\ \mathrm{LE} \longleftrightarrow e\text{'}\ ASA}{e\ \mathrm{E} \longleftrightarrow e\text{'}\ ASA}\ \mathit{(exprB)}$$

[a]Definición formulada a partir de [1], [5], [8] y [12]

3 El operador let

Ahora necesitamos extender aún mas EAB para incluir expresiones de tipo:

$$\texttt{let}\quad x = e_1\ \texttt{in}\ e_2\ \texttt{end}$$

Donde las apariciones de la variable x serán sustituidas por e_1 en e_2.

Para este propósito se tiene que definir la sintaxis del operador `let` y el concepto de ligado de variables.

Ejercicio 3.1. Proporciona las sintaxis concreta para agregar las instrucciones `let` de tal forma que sea congruente con las especificación mencionada anteriormente[a].

$$\frac{e\ \text{identificador}}{e\ \mathrm{V}}\ \mathit{(var)} \quad \frac{e\ \mathrm{V}}{e\ \mathrm{B}}\ \mathit{(vIn)} \quad \frac{e\ \mathrm{V}}{e\ \mathrm{F}}\ \mathit{(vfac)} \quad \frac{x\ \mathrm{V} \quad e_1\ \mathrm{E} \quad e_2\ \mathrm{E}}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end E}}\ \mathit{(let)}$$

[a]Definición formulada a partir de [1], [5], [12] y [81]

La sintaxis abstracta de orden superior define un tratamiento especial para las variables que se introducen con el operador `let` así como el alcance y ligado de las mismas. Esta sintaxis abstrae estos niveles como parte del metalenguaje con el que se define el árbol de sintaxis abstracta, es decir vamos a delimitar el rango de aplicación de la variable con el árbol que definiremos para el operador mismo.

Extenderemos nuestro lenguaje EAB agregando el árbol de sintaxis abstracta $x.t$ que indica que la variable x está ligada en el árbol t, a este ASA se le llama abstracción[7]. Esto es

[7]Definición acuñada a partir de [1], [2], [5], [12], [80] y [81]

importante dado que al introducir las variables ligadas y su alcance tenemos que delimitar la expresión en la que la sustitución ocurrirá.

Ejercicio 3.2. Proporciona las sintaxis Abstracta para agregar las instrucción `let`[a] considerando el *ASA x.t.*

$$\frac{x \text{ identificador}}{\text{var}[x] \ ASA} \qquad \frac{a_1 \ ASA \quad a_2 \ ASA}{let(a_1,\, x.a_2) \ ASA}$$

[a]Definición formulada a partir de [1], [5], [12] y [81]

Ejercicio 3.3. Define las reglas del analizador sintáctico para agregar las instrucción `let`[a].

$$\frac{x \text{ identificador}}{x \text{ V} \longleftrightarrow \text{var}[x] \ ASA}\ vara \qquad \frac{x \text{ V} \quad e_1 \text{ E} \longleftrightarrow a_1 \ ASA \quad e_2 \text{ E} \longleftrightarrow a_2 \ ASA}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end E} \longleftrightarrow let(a_1 \ x.a_2) \ ASA}\ leta$$

[a]Definición formulada a partir de [1], [5], [12] y [81]

A continuación tenemos tres ejemplos que nos van a ayudar a ilustrar la aplicación de las reglas que hemos definido hasta el momento para construir los *ASA* de EAB partiendo de expresiones derivadas de las reglas de la sintaxis concreta.

Ejercicio 3.4. Obtén la representación en sintaxis abstracta partiendo de la sintaxis concreta de las siguiente expresión:

$\texttt{let } x = 1 \texttt{ in } x + 1 \texttt{ end}$

$$let(num[1],\ x.var[x]+1)\ ASA$$

Ejercicio 3.5. Obtén la representación en sintaxis abstracta partiendo de la sintaxis concreta de las siguiente expresión:

$\texttt{let } y = \texttt{if True then } 1 \texttt{ else } 0 \texttt{ in if } x \texttt{ then False end}$

$$let(if(T,\ num[1],\ num[0]),\ y.if(var[x],\ F))\ ASA$$

Ejercicio 3.6. Obtén la representación en sintaxis abstracta partiendo de la sintaxis concreta de las siguiente expresión:

$\texttt{let } x \texttt{ = False in if } x \texttt{ then } (\texttt{if } \alpha \texttt{ then } (\texttt{if } \alpha \texttt{ then } 4)) \texttt{ else False end}$

$$let(F,\ x.if(var[x],\ if(var[\alpha],\ if(var[\alpha],\ num[4])),\ F))\ ASA$$

3.1 Clasificación de variables en el operador let

La introducción de nuestro nuevo operador `let` acarrea consigo la definición de ligado de una variable, variable ligada[8], variable libre y la definición de la función de sustitución junto con la función para obtener variables libres de una expresión, mismas que serán mostradas a continuación y serán de utilidad para resolver el resto de los ejercicios presentes en

[8]Aún que los nombres son similares ligado de una variable y variable ligada corresponden a dos definiciones distintas.

este capítulo.

Definición 3.1. Ligado de una variable: la instancia de ligado de una variable es aquella que da a esta su valor[a]. Es decir, es la aparición de la variable en donde esta es definida.

De tal forma que si tenemos la siguiente expresión:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$$

Decimos que $x = e_1$ es el ligado de la variable x en la instrucción `let`.

[a]Definición formulada de [2], [5], [12], [80] y [81]

Definición 3.2. Variable ligada: sea $x.t$ un ASA, decimos que x está ligada en t para todas las apariciones de x[a].

De tal forma que si tenemos la siguiente expresión:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$$

Decimos que x es la variable ligada en e_2 dentro de nuestra expresión `let`.

[a]Definición formulada de [2], [5], [12], [80] y [81]

Definición 3.3. Variable libre: una variable está libre en una expresión, si no se encuentra dentro del alcance de una variable de ligado con su nombre. Es decir, una variable es libre si no está ligada[a].

De tal forma que si tenemos la siguiente expresión:

$$\texttt{let}\ x = e_1\ \texttt{in}\ y\ \texttt{end}$$

Decimos que y es una variable libre en nuestra expresión `let`.

[a]Definición formulada de [2], [5], [12], [80] y [81]

Definición 3.4. Función para obtener las variables libres de una expresión: dado el árbol de sintaxis abstracta a definimos el conjunto de variables libres de la expresión[a], denotado $FV(a)$ como sigue:

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(O(a_1, \cdots, a_n)) & = & FV(a_1) \cup \cdots \cup FV(a_n) \\ FV(x.a) & = & FV(a) \backslash \{x\} \end{array}$$

[a]Definición extraída de [2], [5], [12], [80] y [81]

Ejercicio 3.7. Dada la siguiente expresión `let` definida como:

$$
\begin{aligned}
e =_{\text{def}} \texttt{let}\ x = \texttt{let}\ x = (2+4)\ \texttt{in}\ x+1\ \texttt{end}\ \ \texttt{in}\\
\texttt{let}\ y = (x*2)+x\ \texttt{in}\\
(x*y) + \texttt{let}\ z = 7\ \texttt{in}\ y+z\ \texttt{end}\\
\texttt{end}\\
\texttt{end}
\end{aligned}
$$

Por cada subexpresión let en e (inclusive) proporciona el alcance indicado o subrayando las variables ligadas.

$$
\begin{aligned}
e =_{\text{def}} \texttt{let}\ \underline{x} = \texttt{let}\ \underline{x} = (2+4)\ \texttt{in}\ \underline{x}+1\ \texttt{end}\ \ \texttt{in}\\
\texttt{let}\ \underline{y} = (\underline{x}*2)+\underline{x}\ \texttt{in}\\
(\underline{x}*\underline{y}) + \texttt{let}\ \underline{z} = 7\ \texttt{in}\ \underline{y}+\underline{z}\ \texttt{end}\\
\texttt{end}\\
\texttt{end}
\end{aligned}
$$

Ejercicio 3.8. Escribe cada subexpresión let en e (inclusive) indicando la variable de ligado y la expresión `let` para la cual la variable de ligado tiene alcance.

$$
\begin{aligned}
\texttt{let}\ \underline{x} = \texttt{let}\ x = (2+4)\ \texttt{in}\ x+1\ \texttt{end}\ \ \texttt{in}\\
\texttt{let}\ y = (x*2)+x\ \texttt{in}\\
(x*y) + \texttt{let}\ z = 7\ \texttt{in}\ y+z\ \texttt{end}\\
\texttt{end}\\
\texttt{end}
\end{aligned}
$$

La siguiente expresión `let` es la que le da su valor a la variable más externa definida en e la cual se identifica como:

$$\texttt{let}\ \underline{\texttt{x}} = (2+4)\ \texttt{in}\ x+1\ \texttt{end}$$

La expresión `let` que continua está delimitada por los marcadores más externos `in end` en e y se identifica como:

$$
\begin{aligned}
\texttt{let}\ \underline{\texttt{y}} = (x*2)+x\ \texttt{in}\\
(x*y) + \texttt{let}\ z = 7\ \texttt{in}\ y+z\ \texttt{end}\\
\texttt{end}
\end{aligned}
$$

La útltima expresión `let` está delimitada por los marcadores `in end` de la expresión anterior y se identifica como:

$$\texttt{let}\ \underline{\texttt{z}} = 7\ \texttt{in}\ y+z\ \texttt{end}$$

Ejercicio 3.9. La expresión e se puede evaluar a un valor? si sí muestra el procedimiento para la evaluación informal (aún no definimos la manera en la que se va a evaluar la expresiones del lenguaje EAB pero se puede dar una idea de la evaluación partiendo de la descripción hecha en el capítulo).

$$\begin{aligned} e =_{\text{def}} \texttt{let}\ x = \texttt{let}\ x = (2+4)\ \texttt{in}\ x+1\ \texttt{end}\ \ \texttt{in} \\ \texttt{let}\ y = (x*2)+x\ \texttt{in} \\ (x*y) + \texttt{let}\ z = 7\ \texttt{in}\ y+z\ \texttt{end} \\ \texttt{end} \\ \texttt{end} \end{aligned}$$

Primero se resuelve la expresión let que la da valor a la variable x en la expresión e:

$$\texttt{let}\ x = (2+4)\ \texttt{in}\ x+1\ \texttt{end}\ = (2+4)+1 = (6)+1 = 7$$

Se sustituye este valor en el cuerpo de e resultando en la siguiente expresión:

$$\texttt{let}\ y = (7*2)+7\ \texttt{in} \qquad (7*y) + \texttt{let}\ z = 7\ \texttt{in}\ y+z\ \texttt{end}\ \ \texttt{end}$$

Resolviendo la operación para dar variable a la variable y obtenemos el siguiente resultado:

$$y = (7*2)+7 = (14)+7 = 21$$

Sustituyendo el nuevo valor en el cuerpo de la expresión anterior se obtiene:

$$= (7*21) + \texttt{let}\ z = 7\ \texttt{in}\ 21+z\ \texttt{end}$$

Resolviendo las operaciones y la sustitución de la variable z en el cuerpo de la expresión let se obtiene:

$$= 147 + 21 + 7 = 168 + 7 = 175$$

Ejercicio 3.10. En nuestra expresión e hay variables libres, argumenta por qué si o por qué no.

No, todas las variables definidas en la expresión e están asociadas a un operador `let` marcando su alcance en las expresiones anidadas presentes por lo que ninguna variable aparece sin su respectiva asociación.

4 Sustitución y α-equivalencias

Para finalizar este capítulo se introduce el concepto de alfa equivalencia:

Definición 4.1. α-equivalencia: se dice que dos expresiones e_1 y e_2 son α-equivalentes si y sólo si difieren únicamente en el nombre de las variables ligadas[a].

Esta relación será denotada como:

$$e_1 \equiv_\alpha e_2$$

[a]La notación de la relación de α-equivalencia fue extraída de [1], [5] y [12], la definición formulada a partir de [81] y [89]

Esta definición resulta de utilidad para trabajar con expresiones que necesiten ser renombradas a lo más en sus variables ligadas para poder aplicar la sustitución sobre dicha expresión cuando haya un conflicto ocasionado por los identificadores.

A continuación definimos la función de sustitución aplicada a expresiones pertenecientes a nuestro lenguaje EAB.

Definición 4.2. Función de sustitución: se define la función de sustitución[a] sobre los árboles de sintaxis abstracta de EAB:

$$
\begin{array}{lcl}
var[x][var[x] := e] & = & e \\
var[z][var[z] := e] & = & var[z] \\
T\ [var[z] := e] & = & T \\
F\ [var[z] := e] & = & F \\
num[n][var[x] := e] & = & num[n] \\
sum(a_1, a_2)[var[x] := e] & = & sum(a_1[var[x] := e], a_2[var[x] := e]) \\
prod(a_1, a_2)[var[x] := e] & = & prod(a_1[var[x] := e], a_2[var[x] := e]) \\
let(a_1, a_2)[var[x] := e] & = & let(a_1[var[x] := e], a_2[var[x] := e]) \\
if(a_1, a_2)[var[x] := e] & = & if(a_1[var[x] := e], a_2[var[x] := e]) \\
if(a_1, a_2, a_3)[var[x] := e] & = & if(a_1[var[\mathrm{x}] := e], a_2[var[x] := e], a_3[var[x] := e]) \\
(z.a)[var[x] := e] & = & z.(a[var[x] := e]) \text{ Si } x \neq z \text{ y } z \notin FV(e) \\
(z.a)[var[x] := e] & = & \texttt{indefinido} \text{ Si } z \in FV(e)
\end{array}
$$

[a]Esta definición está formulada a partir de [1], [2], [5] y [12] adaptada para las instrucciones que hemos revisado hasta este momento para EAB.

Ejercicio 4.1. Proporciona una expresión α-equivalente a la siguiente expresión:
`let` $x =$ `let` $y =$ `False in if (y then 1 else 0) in` $(x+2)*w$ `end`

`let` $k =$ `let` $j =$ `False in if (j then 1 else 0) in` $(k+2)*w$ `end`

Ejercicio 4.2. Proporciona una expresión α-equivalente a la expresión e definida en el ejercicio 3.7.

$$
\begin{array}{l}
e_\alpha = \texttt{let}\ a = \texttt{let}\ d = (2+4)\ \texttt{in}\ d+1\ \texttt{end}\ \ \texttt{in} \\
\qquad\qquad \texttt{let}\ b = (a*2)+a\ \texttt{in} \\
\qquad\qquad\quad (a*b) + \texttt{let}\ c = 7\ \texttt{in}\ b+c\ \texttt{end} \\
\qquad\qquad \texttt{end} \\
\qquad \texttt{end}
\end{array}
$$

Ejercicio 4.3. Desarrolla la sustitución aplicada a la siguiente expresión de acuerdo con la función de sustitución previamente definida.

$$\texttt{let}\ x = (w+1)*2\ \texttt{in if (True then}\ w\ \texttt{else}\ (2*x))\ \texttt{end}\ [w := 2]$$

$$= let(prod(sum(var[w],\ 1),\ 2),\ if(T,\ var[w],\ prod(2,\ var[x])))\ \ [var[w] := 2]$$

$$= let(prod(sum(var[w],\ 1),\ 2)\ \ [var[w] := 2],\ if(T,\ var[w],\ prod(2,\ var[x]))\ \ [var[w] := 2]\ )$$

$$= let(prod(sum(var[w],\ 1)\ [var[w] := 2],\ 2\ [var[w] := 2]),\ if(T\ [var[w] := 2],\ var[w]\ [var[w] := 2],\ prod(2,\ var[x])[var[w] := 2]\ ))$$

$= let(prod(sum(var[w]\ [var[w] := 2]\ ,\ 1[var[w] := 2]),\ 2),\ if(T,\ 2,\ prod(2\ [var[w] := 2],\ var[x]\ [var[w] := 2])))$

$$= let(prod(sum(2,\ 1),\ 2),\ if(T,\ 2,\ prod(2,\ var[x])))$$

5 Ejercicios para el lector

Ejercicio 5.1. Explica con tus palabras la diferencia entre la sintaxis abstracta y la sintaxis concreta.

Ejercicio 5.2. Explica con tus palabras por que es necesaria la sintaxis abstracta de orden superior que se introduce con el árbol de sintáxis abstracta $x.t$

Ejercicio 5.3. Proporciona el árbol de sintaxis abstracta para la siguiente expresión `let`

$$\texttt{let}\ z = 777\ \texttt{in}\ (w+h)*(z) +\ \texttt{let}\ v = -33\ \texttt{in}\ (z*z)+v\ \texttt{end}\ \ \texttt{end}$$

Ejercicio 5.4. Proporciona el árbol de sintaxis abstracta para la siguiente expresión `if`

$$\texttt{if(True then if(False then}\ 0)\ \texttt{else}\ (2*7)-1)$$

Ejercicio 5.5. La siguiente expresión ¿es una expresión correcta de nuestro lenguaje? Explica por que si o porque no de acuerdo a las reglas de sintaxis concreta definidas.

$$\texttt{if(}\top\ \texttt{then}\ \ \texttt{let}\ x = True\ \texttt{in}\ \ \texttt{if}\ (x\ \texttt{then}\ \bot)\ \texttt{end})$$

Ejercicio 5.6. Dada la siguiente expresión let definida como:

$$\begin{aligned}
e_1 =_{\text{def}}\ & \texttt{let}\ x = \texttt{let}\ x = 2\ \texttt{in}\ \ \texttt{let}\ y = 3\ \texttt{in}\ y + x\ \texttt{end}\ \ \texttt{end}\ \ \texttt{in} \\
& \quad\quad \texttt{let}\ y = (x*2)*19\ \texttt{in} \\
& \quad\quad\quad (x+y) * \texttt{let}\ z =\ \texttt{let}\ v = 8\ \texttt{in}\ z\ \texttt{end}\ \ \texttt{in}\ (y+z)*v\ \texttt{end} \\
& \quad\quad \texttt{end} \\
& \quad \texttt{end}
\end{aligned}$$

Proporciona lo siguiente por cada subexpresión `let` definida en e:

- Las variables ligadas
- Las variables libres
- El alcance de cada variable
- El valor que se obtiene al desarrollar la expresión.

Ejercicio 5.7. Proporciona una expresión α-equivalente a e_1 del ejercicio anterior

Ejercicio 5.8. Desarrolla la siguiente sustitución, en caso de ser inválida explica por qué.

$$\texttt{let}\ z = 3\ \texttt{in}\ (w+z) +\ \texttt{if(True then}\ 45\ \texttt{else}\ y)\ \texttt{end}\ [y := 7]$$

Ejercicio 5.9. Desarrolla la siguiente sustitución paso por paso, en caso de ser inválida explica por qué.

$$\texttt{let } z = 3 \texttt{ in } (w + z) + \texttt{ if (True then } 45 \texttt{ else } y\texttt{) end } [y := z + 7]$$

Ejercicio 5.10. A continuación se define la sintaxis concreta para un lenguaje funcional simple:

$$e ::= \; x \mid n \mid e_1 \; e_2 \mid fun(x) \rightarrow e$$

En donde el primer constructor representa las variables del lenguaje, el segundo números naturales, el tercero aplicación de función y el último la definición de funciones.

De acuerdo a la especificación discutida anteriormente, contesta los siguiente incisos:

- Define las reglas de inferencia para la sintaxis concreta de la gramática anterior.
- Diseña una sintaxis abstracta apropiada para este lenguaje (Primero observa si es necesario alguna especie de ligado como el que define el operado `let` discutido en este capítulo).
- Escribe las reglas para la relación de análisis sintáctico ($\longleftrightarrow$) del lenguaje.
- Diseña un algoritmo de sustitución para este lenguaje.
- La relación de α-equivalencia en este lenguaje se da respecto al operador de definición de función fun. Se dice que dos expresiones son α-equivalentes si solo difieren en el nombre de la variable del parámetro de la función. Por ejemplo:

  $$fun(x) \rightarrow e \equiv_\alpha fun(y) \rightarrow e$$

  Demuestra que $\equiv_\alpha$ es una relación de equivalencia.

Capítulo 4

Semántica

En el capítulo anterior se estudió la composición sintáctica de los lenguajes de programación mediante la definición de juicios lógicos que nos permiten construir expresiones válidas de EAB definiendo así la sintaxis concreta de este lenguaje. También se estudió la estructura jerárquica de bajo nivel con la que se obtiene la representación única de las expresiones válidas de EAB conocida como árboles de sintaxis abstracta, este primer nivel de los lenguajes de programación está enfocado a la representación del lenguaje pero no a su significado.

En el presente capítulo nos ocuparemos de la semántica del lenguaje, que tiene como propósito responder a la pregunta ¿qué significa una expresión del lenguaje? y como podemos interpretar aquellas expresiones correctamente formuladas a partir de la sintaxis.

En el contexto de los lenguajes de programación podemos pensar de forma intuitiva que la respuesta al cómo interpretamos las instrucciones del lenguaje podría ser: ”el comportamiento que tiene la expresión en tiempo de ejecución al ser evaluada.” En particular nos interesa obtener el valor que la expresión retorna al concluir su evaluación (semántica dinámica) y el tipo de esta (semántica estática).[1].

[1]Definición acuñada de [98], [99] y [100]

Muchas veces en la documentación oficial de los lenguajes de programación la semántica está escrita en un manual el cual contiene una descripción de alto nivel acerca de como una determinada instrucción, método o expresión se comporta al momento de ejecutarse. Si bien esto resulta útil al escribir un programa nuestro interés sera definir formalmente la semántica de ejecución mediante juicios y reglas para el lenguaje EAB.

Objetivo

Definir los diferentes niveles semánticos de EAB mediante reglas de inferencia que dictaminen el proceso que se debe seguir al para evaluar o asignar un tipo a una expresión bien formada de este lenguaje.

Planteamiento

Iniciaremos el estudio de este capítulo presentando las reglas para poder asignar un tipo a cada una de las expresiones bien formadas de EAB. Este nivel corresponde a la semántica estática y será de utilidad para fundamentar la seguridad del lenguaje[2] y garantizar que se pueda regresar un valor al final de la evaluación de la expresión a la que se pretende asignar el tipo.

Posteriormente se definirá el procedimiento para evaluar las expresiones bien formadas de EAB. Este mecanismo se conoce como semántica dinámica y se estudiará en dos paradigmas: de paso grande y de paso pequeño. Por último definiremos la función `eval` para EAB.

1 Semántica estática

La semántica estática extrae información del programa en tiempo de compilación, la cantidad y calidad de la información obtenida depende de la implementación de cada compilador[3].

La información que se extrae del programa provisto por el usuario puede ser: el alcance de una variable, el tipo de una expresión, saber si una expresión tiene variables libres, etc. Saber esto es indispensable para poder evaluar las expresiones de EAB (este último punto es importante ya que necesitamos que todas las variables estén ligadas y no haya presencia de variables libres en nuestras expresiones, de lo contrario de la evaluación no se obtendrá valor alguno).

En el lenguaje de expresiones aritméticas con el que hemos estado trabajando, las expresiones de la forma:

$$\texttt{let}\ x = y\ \texttt{in}\ x + x\ \texttt{end}$$

son sintácticamente correctas pero semánticamente incorrectas, pues la variable y está libre en la expresión por lo que no podría evaluarse.

[2]La seguridad de un lenguaje se relaciona con la asignación correcta de tipos y la vamos a estudiar mas adelante en este capítulo.

[3]Definición acuñada de [96] y [97]

Esta es una de las propiedades de las que se encarga la semántica estática, así que daremos un conjunto de reglas para definir una semántica estática encargada de evitar estos errores. Para esto se define el siguiente juicio:

Definición 1.1. Semántica estática para capturar expresiones con variables libres en EAB[a]: Sea Δ un conjunto de variables y a un árbol de sintaxis abstracta, decimos que a no tiene variables libres bajo el conjunto Δ denotado como:

$$\Delta \sim a \ ASA$$

En particular nos interesa que todas las variables ligadas de a estén presentes en Δ. Para construir este conjunto definimos las siguientes reglas que nos permitirán acrecentar Δ cada que encontremos una variable ligada en el árbol $x.t$. Definimos las reglas como sigue:

$$\dfrac{x \in \Delta}{\Delta \sim \ x}\ (\mathit{fvv}) \qquad \dfrac{}{\Delta \sim num[n]}\ (\mathit{fvn}) \qquad \dfrac{\Delta, x \sim e}{\Delta \sim \ x.e}\ (\mathit{fva})$$

$$\dfrac{}{\Delta \sim T \ \text{Bool}}\ (\mathit{fvb}) \qquad \dfrac{\Delta \sim e_1 \quad \cdots \quad \Delta \sim e_n}{\Delta \sim O(e_1, \cdots, e_n)}\ (\mathit{fvo})$$

Para garantizar que una expresión e no tiene variables libres se inicia con el conjunto vacío y se debe probar $\emptyset \sim e$ usando las reglas anteriores.

[a]Definición acuñada de [5] y [12]

Definición 1.2. Expresión cerrada de EAB[a]: Sea a un ASA decimos que a es cerrada si y sólo si

$$FV(a) \ = \ \emptyset$$

Es decir, a es una expresión del lenguaje cerrada si no tiene apariciones de variables libres.

[a]Definición acuñada de [5] y [12]

Ejercicio 1.1. Para la siguiente expresión realiza el análisis estático para encontrar variables libres mediante la derivación aplicando las reglas del juicio $\sim$

$$\texttt{let} \ x = 1 \ \texttt{in} \ (y * x) + z \ \texttt{end}$$

$$\dfrac{\dfrac{}{\emptyset \sim 1}\ (\mathit{fvn}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{Error}{\{x\} \sim y} \quad \dfrac{x \in \{x\}}{\{x\} \sim x}\ (\mathit{fvn})}{\{x\} \sim \mathit{prod(x,y)}}\ (\mathit{fvo}) \qquad \dfrac{Error}{\{x\} \sim z}}{\{x\} \sim \mathit{sum(prod(y,\ x),\ z)}}\ (\mathit{fvo})}{\emptyset \sim \mathit{x.sum(prod(y,\ x),\ z))}}}{\emptyset \sim \mathit{let(1,x.sum(prod(y,x),z))}}\ (\mathit{fva})$$

Ejercicio 1.2. Para la siguiente expresión realiza el análisis estático para encontrar variables libres mediante la derivación aplicando las reglas del juicio $\sim$

$$\texttt{if True then } (1+1)*2 \texttt{ else } 7$$

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{\textit{(fvb)}}
\UnaryInfC{$\emptyset \sim T$}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\emptyset \sim 1$}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\emptyset \sim 1$}
\RightLabel{\textit{(fvo)}}
\BinaryInfC{$\emptyset \sim \mathit{sum(1,\ 1)}$}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\emptyset \sim 2$}
\RightLabel{\textit{(fvo)}}
\BinaryInfC{$\emptyset \sim \mathit{prod(sum(1,\ 1),\ 2)}$}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\emptyset \sim 7$}
\RightLabel{\textit{(fvo)}}
\TrinaryInfC{$\emptyset \sim \mathit{if(T,\ prod(sum(1,\ 1),\ 2),\ 7)}$}
\end{prooftree}
$$

Ejercicio 1.3. Para la siguiente expresión realiza el análisis estático para encontrar variables libres mediante la derivación aplicando las reglas del juicio $\sim$

$$1 + \texttt{if(let } x = \texttt{True in } x \texttt{ end then } 2 \texttt{ else } 1\texttt{)}$$

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\Delta \sim 1$}
\AxiomC{}
\RightLabel{\textit{(fvb)}}
\UnaryInfC{$\Delta \sim T$}
\AxiomC{}
\RightLabel{\textit{(fva)}}
\UnaryInfC{$\{x\} \sim x$}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\Delta \sim x.x$}
\RightLabel{\textit{(fvo)}}
\BinaryInfC{$\Delta \sim \mathit{let(T,x.x)}$}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\Delta \sim 2$}
\AxiomC{}
\RightLabel{\textit{(fvn)}}
\UnaryInfC{$\Delta \sim 1$}
\RightLabel{\textit{(fvo)}}
\TrinaryInfC{$\Delta \sim \mathit{if(let(T,x.x)},2,1)$}
\RightLabel{\textit{(fvo)}}
\BinaryInfC{$\Delta \sim \mathit{sum(1,if(let(T,x.x)},2,1))$}
\end{prooftree}
$$

De los ejercicios anteriores podemos notar que el análisis sintáctico para evaluar expresiones con variables libres fallará en una o más ramas, mientras que los árboles de derivación generados de expresiones cerradas concluirán todas sus ramas con algún axioma (*fvn, fvb o fvv*).

2 Semántica dinámica

La semántica dinámica constituye el siguiente eslabón del proceso de ejecución de un programa. Una vez verificada que la expresión en sintaxis concreta es una expresión válida y no posee variables libres, entonces se puede comenzar a discutir el como se ejecutará dicha expresión.

Existen diferentes paradigmas de ejecución que pueden ser aplicados para evaluar las expresiones de los lenguajes de programación según el interés de estudio (por ejemplo cambios en la memoria, el valor las variables del programa, el valor final al que evalúa una expresión, etc). En este capítulo nos centraremos en la semántica operacional que estudia el proceso de ejecución modelando cada configuración del programa como un estado y las transiciones entre estos.

2.1 Semántica operacional

En esta categoría se hace la distinción entre los dos paradigmas mas importantes para estudiar la semántica operacional de los lenguajes de programación[4]:

- Semántica de paso pequeño: la cuál modela la ejecución de un programa describiendo las transiciones una a una mostrando los cómputos generados de forma individual.
- Semántica de paso grande: que contrasta con la de paso pequeño porque aquí no nos importa que estados sucedieron en la ejecución del programa, únicamente nos interesa el valor que regresa.

[4]Definición acuñada de y [2], [103] y [104]

Definición 2.1. Sistema de transición para la semántica operacional de EAB[a]:

Conjunto de estados $S = \{a \mid a\ asa\}$

Los estados del sistema son las expresiones del lenguaje en sintaxis abstracta. Esta definición corresponde a la regla de inferencia:

$$\frac{a\ asa}{a\ estado}\ \textit{(state)}$$

Estados Iniciales $I = \{a \mid a\ asa,\ FV(a) = \emptyset\}$

Los estados iniciales son todas las expresiones cerradas del lenguaje y corresponde a la regla:

$$\frac{a\ asa \qquad \emptyset \sim a}{a\ inicial}\ \textit{(init)}$$

Estados Finales Son las expresiones que no se pueden reducir mas obtenidas al final del proceso de evaluación.

Definimos una categoría de valores los cuales son un subconjunto de expresiones de EAB, esta nueva categoría se representa con el juicio $v\ valor$. Para el caso de EAB se denota con las reglas:

$$\frac{}{num[n]\ valor}\ \textit{(vnum)} \qquad \frac{}{B\ \mathrm{Bool}\ valor}\ \textit{(vbool)}$$

Entonces se define el conjunto de estados finales $F = \{a \mid a\ valor\}$, correspondiente a la regla:

$$\frac{a\ valor}{a\ final}\ \textit{(fin)}$$

[a]Definición acuñada de [5] y [12] y [105]

Definición 2.2. Estado bloqueado[a]: Un estado s está bloqueado si no existe otro estado s' tal que $s \to s'$ y lo denotamos como $s \not\to$

[a]Definición acuñada de [5] y [12]

2.2 Semántica de paso pequeño

Para esta semántica las transiciones se modelarán paso a paso mediante la función de transición, denotada como:

$$e_1 \to e_2$$

Donde e_1 es llamado "*redex*" y e_2 es llamado "*reducto*". Esta relación se interpreta cómo la transición entre e_1 y e_2 que existe si y solo sí en un paso de evaluación se puede reducir la expresión e_1 a la expresión e_2[5].

Definición 2.3. Función de transición para semántica de paso pequeño[a]:

Para definir las reglas de transición para los cómputos siguiendo el paradigma de semántica de paso pequeño, primero vamos a evaluar el argumento más a la izquierda dentro de las expresiones que denotan un operador hasta obtener un valor. Posteriormente evaluaremos el argumento a la derecha.

[5]Definición acuñada de [5] y [12]

Suma

$$\frac{}{sum(num[n], num[m]) \to num[n +_N m]}\ (sum_N)$$

$$\frac{a_1 \to a_1'}{sum(a_1, a_2) \to sum(a_1', a_2)}\ (sum_L) \qquad \frac{a_2 \to a_2'}{sum(num[n], a_2) \to sum(num[n], a_2')}\ (sum_R)$$

Producto

$$\frac{}{prod(num[n], num[m]) \to num[n \times_N m]}\ (prod_N)$$

$$\frac{a_1 \to a_1'}{prod(a_1, a_2) \to prod(a_1', a_2)}\ (prod_L) \qquad \frac{a_2 \to a_2'}{prod(num[n], a_2) \to prod(num[n], a_2')}\ (prod_R)$$

Expresiones lógicas

$$\frac{}{if(T, e_1, e_2) \to e_1}\ (if_T) \qquad \frac{}{if(F, e_1, e_2) \to e_2}\ (if_F)$$

$$\frac{a_1 \to a_1'}{if(a_1, e_1, e_2) \to if(a_1', e_1, e_2)}\ (if_B)$$

Asignaciones locales

$$\frac{v\ valor}{let(v, x.a_2) \to a_2[x := v]}\ (let_V) \qquad \frac{a_1 \to a_1'}{let(a_1, x.a_2) \to let(a_1', x.a_2)}\ (let_L)$$

La evaluación del operador `if` es una evaluación "perezosa", en donde no se evaluará el cuerpo hasta antes haber determinado el valor de la sentencia de control (`True` ó `False`)

[a] Definición acuñada de [2], [5], [12] y [105]

Ejercicio 2.1. Dada la siguiente expresion de nuestro lenguaje EAB utiliza las reglas de transición de la semántica de paso pequeño para evaluarla.

$$\texttt{let}\ \ k = (3+1)\ \ \texttt{in}\ \ (7 * k) + 1\ \ \texttt{end}$$

$$\begin{aligned}
& let(sum(3,1), k.sum(prod(7,k),1)) \\
\to\ & let(4, k.sum(prod(7,k),1)) \\
\to\ & (sum(prod(7,k),1))[k := 4] \\
\to\ & sum(prod(7,k)[k := 4], 1[k := 4]) \\
\to\ & sum(prod(7[\mathrm{k} := 4], k[k := 4]), 1) \\
\to\ & sum(prod(7,4),1) \\
\to\ & sum(28,1) \\
\to\ & 29
\end{aligned}$$

Ejercicio 2.2. Dada la siguiente expresión de nuestro lenguaje EAB utiliza las reglas de transición de la semántica de paso pequeño para evaluarla.

$$\texttt{if (False then}\ 4 * \texttt{let}\ x = 99 * 99\ \texttt{in}\ x + x\ \texttt{end else}\ 0)$$

$$\begin{array}{ll} & if(F,\ let(prod(99,99),\ x.sum(x,x)),\ 0) \\ \to & 0 \end{array}$$

Del ejercicio anterior podemos notar la ventaja de la evaluación perezosa. Dado que nuestro valor de control es `False` no es necesario evaluar la expresión e_1 y directamente nos saltaremos a evaluar e_2 por la regla if_F que en este caso es el valor 0.

Ejercicio 2.3. Dada la siguiente expresion de nuestro lenguaje EAB utiliza las reglas de transición de la semántica de paso pequeño para evaluarla.

`let` $x =$ `if(True then` 42 `else` 0`) in if(False then` 41 `else` x`)`

$$\begin{array}{ll} & let(if(T,42,0), x.if(F,41,x)) \\ \to & let(42, x.if(F,41,x)) \\ \to & if(F,41,x)[x := 42] \\ \to & if(F[x := 42], 41[x := 42], x[x := 42]) \\ \to & if(F,41,42) \\ \to & 42 \end{array}$$

La relación de transición ($\to$) define tres categorías importantes de aplicación las cuales se conocen como "cerraduras" y pueden ser: reflexiva (un estado puede llegar a si mismo en 0 aplicaciones de pasos), transitiva (si un estado e_1 puede alcanzar un estado e_2 y e_2 puede alcanzar a e_n en un número finito de pasos entonces e_1 puede llegar a e_n en un número finito de pasos) ó positiva (la aplicación de las reglas de transición n veces con n ≥ 1).

A continuación enunciamos las cerraduras, la relación de transición y la iteración en n pasos.

Definición 2.4. Cerradura transitiva y reflexiva: se denota como $t \to^*$ y se define con las siguientes reglas[a]:

$$\frac{}{s \to^* s}\ (refl) \qquad \frac{s_1 \to s_2 \quad s_2 \to^* s_3}{s_1 \to^* s_3}\ (trans)$$

La relación $s_1 \to^* s_2$ modela que es posible llegar desde s_1 hasta s_2 en un número finito de aplicaciones (posiblemente 0) de la relación de transición $\to$.

[a]Definición acuñada de [2], [5] y [12]

Definición 2.5. Cerradura positiva: se denota como $\to^+$ y se define con las siguientes reglas[a]:

$$\frac{s_1 \to s_2}{s_1 \to^+ s_2}\ (one+) \qquad \frac{s_1 \to s_2 \quad s_2 \to^+ s_3}{s_1 \to^+ s_3}\ (trans+)$$

La relación $s_1 \to^+ s_2$ modela que es posible llegar desde s_1 hasta s_2 en un número finito de aplicaciones estrictamente mayor a cero de la relación de transición $\to$. Es decir, se llega de s_1 a s_2 en al menos un paso.

[a]Definición acuñada de [2], [5] y [12]

Definición 2.6. Iteración en n pasos: se denota como $\to^n$ con $n \in N$ y se define con las siguientes reglas[a]:

$$\frac{}{s \to^0 s}\ (iter_z) \qquad \frac{s_1 \to s_2 \quad s_2 \to^n s_3}{s_1 \to^{n+1} s_3}\ (iter_n)$$

La relación $s_1 \to^n s_2$ modela que es posible llegar desde s_1 hasta s_2 en exactamente n pasos de la relación de transición $\to$.

[a]Definición acuñada de [2], [5] y [12]

2.3 Semántica de paso grande

Este paradigma para definir la semántica dinámica encapsula de forma general el proceso de evaluación sin mostrar de forma explícita paso a paso como es que una expresión es evaluada (contrario a la semántica de paso pequeño).

La relación de transición en este caso se denota como $e \Downarrow v$ donde e es una expresión válida de nuestro lenguaje EAB y v es un valor, esta se lee como "la expresión e se evalúa a v".

Definición 2.7. Semántica de paso grande para EAB: Se define la relación de transición $\Downarrow$ mediante las siguientes reglas de inferencia[a]:

Valores

$$\frac{}{num[n] \Downarrow num[n]}\ (bsnum) \qquad \frac{}{B\ \text{Bool} \Downarrow B\ \text{Bool}}\ (bsbool)$$

Suma

$$\frac{e_1 \Downarrow num[n] \quad e_2 \Downarrow num[m]}{sum(e_1, e_2) \Downarrow num[n +_N m]}\ (bssum)$$

Producto

$$\frac{e_1 \Downarrow num[n] \quad e_2 \Downarrow num[m]}{prod(e_1, e_2) \Downarrow num[n \times_N m]}\ (bsprod)$$

Asignaciones locales

$$\frac{e_1 \Downarrow v_1 \quad e_2[x := v_1] \Downarrow v_2}{let(e_1, x.e_2) \Downarrow v_2}\ (bslet)$$

Sentencias de control

$$\frac{e_0 \Downarrow True \quad e_1 \Downarrow v_1}{if(e_0, e_1, e_2) \Downarrow v_1}\ (bsift) \frac{e_0 \Downarrow False \quad e_2 \Downarrow v_2}{if(e_0, e_1, e_2) \Downarrow v_2}\ (bsift)$$

Es importante destacar que los valores de tipo $num[n]$ y B Bool si tienen reglas definidas en esta semántica de paso grande.

[a]Definición formulada a partir de [2], [5] y [12]

Teorema 2.8. Equivalencia entre semántica de paso pequeño y paso grande: Decimos que para cualquier expresión e del lenguaje EAB se cumple:

$$e \to^* v \longleftrightarrow e \Downarrow v$$

Es decir, las semánticas que hemos definido son equivalentes[a].

[a]La demostración se puede hacer por inducción estructural sobre cada instrucción de EAB, esta

queda fuera del alcance del presente manual de carácter práctico pero si se desea consultarla esta está disponible en [106]

Ejercicio 2.4. Dada la siguiente expresión de EAB evalúala utilizando semántica de paso grande. Adicionalmente proporciona la representación en sintaxis abstracta.

Sintaxis concreta:

$$\texttt{let}\ x = \texttt{let}\ y\ \texttt{= False in if(}y\ \texttt{then}\ 0\ \texttt{else}\ 1\texttt{) end in}\ x+1\ \texttt{end}$$

Sintaxis abstracta:

$$let(let(F, y.if(y,0,1)), x.sum(x,1))$$

Evaluación paso grande:

$$\dfrac{\dfrac{\dfrac{}{F \Downarrow F}\,\textit{(bsbool)} \qquad \dfrac{}{if(y,0,1))[y := F] \Downarrow 1}\,\textit{(bsift)}}{let(F,\ y.if(y,0,1)) \Downarrow 1}\,\textit{(bslet)} \qquad \dfrac{}{sum(x,1)[x := 1] \Downarrow 2}\,\textit{(bssum)}}{let(let(F, y.if(y,0,1)), x.sum(x,1)) \Downarrow 2}\,\textit{(bslet)}$$

Ejercicio 2.5. Dada la siguiente expresión de EAB evalúa utilizando la semántica de paso grande, adicionalmente proporciona la representación como árbol de sintaxis abstracta.

Sintaxis concreta:

```
((7 + 4) * 4) + ((8 + 3) * 2)
```

Sintaxis abstracta:

$$sum(prod(sum(7,4),4), prod(sum(8,3),2))$$

Evaluación paso grande:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{7 \Downarrow 7}\,\textit{(bsnum)} \quad \dfrac{}{4 \Downarrow 4}\,\text{bsnum}}{sum(7,4) \Downarrow 11}\,\textit{(bsum)} \quad \dfrac{}{4 \Downarrow 4}\,\text{bsnum}}{prod(sum(7,4),4) \Downarrow 44}\,\textit{(bsprod)} \qquad \dfrac{\dfrac{\dfrac{}{8 \Downarrow 8}\,\textit{(bsnum)} \quad \dfrac{}{3 \Downarrow 3}\,\text{bsnum}}{sum(8,3) \Downarrow 11}\,\textit{(bsum)} \quad \dfrac{}{2 \Downarrow 2}\,\textit{(bsnum)}}{prod(sum(8,3),2)) \Downarrow 22}\,\textit{(bsprod)}}{sum(prod(sum(7,4),4), prod(sum(8,3),2)) \Downarrow 66}\,\textit{(bssum)}$$

Ejercicio 2.6. Dada la siguiente expresión de EAB evalúa utilizando la semántica de paso grande, adicionalmente proporciona la representación en sintaxis abstracta.

Sintaxis concreta:

$$\texttt{if(False then}\ (3*7)+1\ \texttt{else}\ (2*7)+1)$$

Sintaxis abstracta:

$$if(F, sum(prod(3,7),1), sum(prod(2,7),1))$$

Evaluación paso grande:

$$\dfrac{\dfrac{}{F \Downarrow F}\,\textit{(bsbool)} \qquad \dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\,\textit{(bsnum)} \quad \dfrac{}{7 \Downarrow 7}\,\textit{(bsnum)}}{prod(2,7) \Downarrow 14}\,\textit{(bsprod)} \quad \dfrac{}{1 \Downarrow 1}\,\textit{(bsnum)}}{sum(prod(2,7),1)\ ) \Downarrow 15}\,\textit{(bssum)}}{if(F, sum(prod(3,7),1), sum(prod(2,7),1)\ ) \Downarrow 15}\,\textit{(bsiff)}$$

3 La función eval

Concluimos este capítulo con la definición de la función de evaluación para EAB, la cuál nos ayuda a vincular ambos paradigmas tratados para la semántica dinámica mediante la siguiente especificación.

Definición 3.1. Se define la función `eval` en términos de la semántica dinámica del lenguaje como sigue[a]:

$$\texttt{eval}(e) = e_f \text{ si y sólo si } e \to^* e_f \text{ y } e_f \not\to$$

La función de evaluación aplica de forma exhaustiva la semántica dinámica hasta llegar a un estado bloqueado (en el caso de EAB los estados bloqueados son los valores).

[a]Definición formulada a partir de [2], [5] y [12]

La función `eval` será de utilidad para el resto de las secciones que visitaremos a lo largo del curso.

4 Ejercicios para el lector

Ejercicio 4.1. Considera la siguiente sintaxis concreta para un lenguaje proposicional simple `Prop`[a] donde solo se utiliza el conector `AND` ($\wedge$) y el operador `NOT` ($\neg$) definida como:

$$\frac{x\ \texttt{Prop} \quad y\ \texttt{Prop}}{x \wedge y\ \texttt{Prop}} \qquad \frac{x\ \texttt{Prop}}{\neg x\ \texttt{Prop}} \qquad \frac{}{\top\ \texttt{Prop}} \qquad \frac{}{\bot\ \texttt{Prop}}$$

- Proporciona una semántica de paso pequeño para evaluar las expresiones en `Prop`.
- Proporciona una semántica de paso grande para evaluar las expresiones `Prop`.

[a]Ejercicio extraído de [107].

Ejercicio 4.2. Ahora supón que se quieren añadir cuantificadores y variables a nuestro lenguaje `Prop`[a] de la siguiente forma:

$$\frac{x\ \texttt{Prop}}{\exists v,\ x\ \texttt{Prop}} \qquad \frac{x\ \texttt{Prop}}{\forall v,\ x\ \texttt{Prop}} \qquad \frac{v\ \textit{variable}}{v\ \texttt{Prop}}$$

- Proporciona un conjunto de reglas que definan la semántica estática que nos permite decidir cuando una expresión de `Prop` no contiene variables libres bajo un contexto Γ. Denotado de la siguiente forma:
  $\Gamma \vdash e\ Ok$

[a]Ejercicio extraído de [107].

Ejercicio 4.3. Una calculadora de Notación Polaca Reversa (`NPR`)[a] es una calculadora que no requiere de parentizado para evaluar las expresiones que son pasadas como argumento. Esta se apoya de una pila para "empujar" los operandos y los operadores así como de la notación post-fija, es decir el operador se escribe después de los operandos, por ejemplo:

$$7\ +\ 1\ =\ 7\ 1\ +$$

$$7\ -\ (3\ +\ 2)\ =\ 7\ 3\ 2\ +\ -$$

Esta calculadora empuja símbolos a la pila hasta encontrar un operador, en tal caso, los dos símbolos más próximos al tope de la pila son extraídos de la pila y el resultado de la operación es empujado nuevamente.

La sintaxis concreta de la calculadora está definida por las siguientes reglas:

$$\frac{x \in N}{x\ Symbol} \quad \frac{x \in \{+,-,*,/\}}{x\ Symbol} \quad \frac{}{\epsilon\ \texttt{NPR}} \quad \frac{x\ Symbol \qquad xs\ \texttt{NPR}}{x\ xs\ \texttt{NPR}}$$

Esta gramática tiene el problema de poder formar expresiones que no pertenecen necesariamente a `NPR` como:

$$1 + 2$$

$$+ * /$$

- Proporciona un conjunto de reglas para definir la semántica estática que pueda analizar una expresión e y nos diga si es una expresión perteneciente a `NPR` denotando el juicio como: $\vdash e\ Ok$.
- Proporciona un conjunto de reglas para definir la semántica de paso pequeño que evalúen las expresiones de `NPR`.
- Proporciona un conjunto de reglas para definir la semántica de paso grande que evalúen las expresiones de `NPR`.

[a]Ejercicio extraído de [107].

Ejercicio 4.4. Utilizando el analizador sintáctico y las reglas de semántica de paso pequeño y grande definidas en el ejercicio anterior contesta lo siguiente:

Dada la expresión `NPR`

$$e = 7\ 1\ 9\ \ - -$$

Muestra que $\vdash$ e Ok

- Evalúa la expresión utilizando la semántica de paso pequeño.
- Evalúa la expresión utilizando la semántica de paso grande.

Ejercicio 4.5. Dada la siguiente expresión de EAB definida como:

$$\begin{aligned} e = \ & \mathsf{let}\ a = \mathsf{let}\ d = 2\ \mathsf{in}\ d + 1\ \mathsf{end}\ \ \mathsf{in} \\ & \quad \mathsf{let}\ b = (a + 1) + a\ \mathsf{in} \\ & \qquad (a + b) + \mathsf{let}\ c = 1\ \mathsf{in}\ b * c\ \mathsf{end} \\ & \quad \mathsf{end} \\ & \mathsf{end} \end{aligned}$$

- Utilizando el analizador sintáctico definido para `NPR` ($\sim$) decide si esta expresión es cerrada.
- Utilizando la semántica dinámica de paso pequeño definida para `NPR` muestra la evaluación de la expresión hasta obtener un valor.
- Utilizando la semántica dinámica de paso grande definida para `NPR` muestra la evaluación de la expresión hasta obtener un valor.

Capítulo 5

Cálculo Lambda

Cuando pensamos en las bases que sentaron los modelos de cómputo modernos podemos inmediatamente asociarlo con las máquinas de Turing, un sistema de transición que se ha estudiado con anterioridad en materias como autómatas y lenguajes formales[1] siendo el más reconocido cuando se habla de conceptos como: computabilidad, algoritmo o complejidad, términos apenas formalizados en 1936.

En este curso estudiaremos un modelo equivalente de cómputo, es aquí donde Alonzo Church nos introduce en 1928 a su aproximación de un sistema formal basado en el concepto de ”función” con las primitivas de ”abstracción” y ”aplicación” que fungiría como fundamento lógico para sustituir la teoría de conjuntos de Zermelo-Fraenkel y la teoría de tipos de Russell. Este

[1]Conforme al plan de estudios que se imparte desde el 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México con clave de asignatura 1425.

sistema fue demostrado como inconsistente por sus dos alumnos Stephen Kleene y Barkley Russell pero no todo sería desechado, conservando la parte del manejo de funciones particularmente rica, consistente y sorprendentemente equivalente al modelo de Alan Turing, a este modelo se le conoce como Cálculo Lambda[2].

Objetivo

Revisar la composición del Cálculo Lambda mediante el estudio de la sintaxis, semántica operacional y la representación de tipos primitivos en este modelo de cómputo, así como las propiedades inherentes a la semántica del cálculo junto con los sistemas de recursión para la evaluación de λ-expresiones.

Planteamiento

En este capítulo se revisará la sintaxis concreta del Cálculo Lámbda para generar λ-expresiones junto con la semántica operacional del cálculo, basándonos en la operación de sustitución sintáctica ([variable := valor]) conocida como β-reducción.

También se abordará la definición de los booleanos junto con los operadores lógicos, estructuras de datos (tuplas, proyecciones, listas) y los numerales de Church con operadores aritméticos mismos que se espera el lector pueda manipular, razonar y resolver en la sección de ejercicios.

Finalmente se introducirá el sistema de recursión para el Cálculo Lambda con los operadores de punto fijo.

1 Sintaxis del Cálculo Lambda

El cálculo lambda es un modelo simple, su sintaxis comprende sólo tres categorías de términos[3]:

- Variables: los elementos de esta categoría pertenecen a un conjunto infinito de letras minúsculas, generalmente las últimas del alfabeto (y en ocasiones pueden estar numeradas, algunos ejemplos pueden ser: x, y, z, x_1, z_2, etc.) y son las expresiones lambda más simples.

- Abstracciones: esta categoría engloba los términos que definen a las funciones anónimas, compuestas por tres elementos: el primero es la letra griega "λ", el segundo es la variable que estará ligada en el cuerpo de la expresión y por último el cuerpo mismo de la función denotado como e, estas se representan de la forma "$\lambda x.e$".

- Aplicación: esta categoría engloba a las expresiones que representan la aplicación de un argumento a una función "$e_1\, e_2$", donde e_1 es la definición de una función anónima y e_2 es el argumento con el que se evaluará dicha función.

[2] Extraído de [108]
[3] Definición extraída de [108], [109] y [110]

Definición 1.1 (Sintaxis concreta del Cálculo Lambda)**.** Definimos el juicio $e\ \lambda$ que indica que e es una expresión válida en el Cálculo Lambda. A estas expresiones las denominaremos $\lambda-$expresiones[a].

$$\frac{x\ \mathsf{var}}{x\ \lambda}\ var \qquad \frac{x\ \mathsf{var} \quad e\ \lambda}{\lambda x.e\ \lambda}\ abs \qquad \frac{e_1\ \lambda \quad e_2\ \lambda}{e_1\, e_2\ \lambda}\ app$$

[a]Definición extraída de [5], [12], [108], [109]

Ejercicio 1.1. Para las siguientes $\lambda-$expresiones anota a la derecha a que categoría de la sintáxis del Cálculo Lambda corresponde dicho término.

caso 1)	x	variable
caso 2)	y	variable
caso 3)	$\lambda x.x$	función anónima
caso 4)	$\lambda x.y$	función anónima
caso 5)	$(\lambda x.x)v$	aplicación
caso 6)	$(\lambda x.x)v'$	aplicación
caso 7)	$\lambda x.\lambda y.xy$	función anónima
caso 8)	$x(\lambda x.y)$	aplicación
caso 9)	$(\lambda x.x)(\lambda y.y)$	aplicación

En el ejemplo anterior, el tercer caso corresponde a la función identidad que regresa el mismo argumento que recibe. El cuarto caso corresponde a la función constante que siempre regresa el mismo valor sin importar el argumento que recibe. El séptimo caso es la ilustración de como se puede anidar dos funciones anónimas para construir una sola de dos parámetros[4].

2 α-equivalencia en el Cálculo Lambda

En el Cálculo Lambda se tiene un constructor similar al operador `let` que tiene una variable ligada, un alcance asociado a ella y con esto el concepto de α-equivalencia fue introducido para expresiones que difieren a lo más en el nombrado de sus variables ligadas.

Definición 2.1 (α-equivalencia en el Cálculo Lambda)**.** Dos $\lambda-$expresiones e_1 y e_2 son α-equivalentes si y sólo si solo difieren a lo mas en el nombre de las variables ligadas[a]. Esto es denotado como:

$$e_1 \equiv_\alpha e_2$$

Por ejemplo, las expresiones:

$$\lambda x.x \qquad \lambda z.z$$

son α-equivalentes y se representa como:

$$\lambda x.x \equiv_\alpha \lambda z.z$$

[a]Definición extraída de [5] y [12]

[4]A este proceso se le conoce como currificación en honor a Haskell Curry y nos permite anidar tantas funciones como parámetros necesitemos.

3 Semántica operacional del Cálculo Lambda

Cómo se discutió brevemente en la introducción del capítulo, la semántica operacional del Cálculo Lambda está definida por la sustitución sintáctica. A la operación de sustituir los términos que concuerden con la variable del operador en la $\lambda-$expresión se le conoce como β-reducción y se representa con el símbolo $\rightarrow_\beta$.

Definición 3.1 (Semántica operacional del Cálculo Lambda)**.** La semántica operacional está definida por la siguiente regla[a]:

$$(\lambda x.t)\, s \rightarrow_\beta t[x := s] \quad (\beta\text{-reducción})$$

Donde se tienen los siguientes casos según la composición de la $\lambda-$expresión:

- $x[x := r] = r$.
- $y[x := r] = y$ si $x \neq y$.
- $(ts)[x := r] = t[x := r]s[x := r]$.
- $(\lambda y.t)[x := r] = \lambda y.t[x := r]$ donde $y \notin FV(r)$.

Es importante remarcar que la asociatividad de la aplicación de la β-reducción es hacia la izquierda

[a]Definición formulada de [5], [12], [108] y [109]

Ejercicio 3.1. Utiliza la definición de β-reducción para evaluar la siguiente $\lambda-$expresión.

$$e = (\lambda x.z)y$$

$$\begin{aligned}(\lambda x.z)y &\rightarrow_\beta z[x := y]\\ &= z\end{aligned}$$

Ejercicio 3.2. Utiliza la definición de β-reducción para evaluar la siguiente $\lambda-$expresión.

$$e = (\lambda x.x)(\lambda y.yy)z$$

$$\begin{gathered}(\lambda x.x)(\lambda y.yy)z \rightarrow_\beta (x[x := \lambda y.yy])(z)\\ = (\lambda y.yy)(z) \rightarrow_\beta yy[y := z]\\ = y[y := z]y[y := z] = zz\\ = zz\end{gathered}$$

Ejercicio 3.3. Utiliza la definición de β-reducción para evaluar la siguiente $\lambda-$expresión.

$$e = \big(\lambda x.\lambda y.xy\big)(\lambda z.z)(w)$$

$$\begin{gathered}\big(\lambda x.\lambda y.xy\big)(\lambda z.z)(w) \rightarrow_\beta (\lambda y.xy[x := \lambda z.z])(w)\\ = (\lambda y.x[x := \lambda z.z]y[x := \lambda z.z])(w) = (\lambda y.(\lambda z.z)y)(w)\\ \big(\lambda y.(\lambda z.z)y\big)(w) \rightarrow_\beta (\lambda y.z[z := y])(w)\\ = \big(\lambda y.y\big)(w) \rightarrow_\beta y[y := w]\\ = w\end{gathered}$$

4 Definibilidad Lambda

En el Cálculo Lambda las funciones son objetos primitivos de orden superior, esto brinda la flexibilidad de poder definir elementos y ser utilizados como argumentos entre sí para generar nuevas expresiones válidas para el sistema. Si extrapolamos esta idea para definir entidades primitivas nos encontraremos nombrando funciones para construir los elementos que pertenecen a esta categoría.

En el contexto de los lenguajes de programación las entidades primitivas o tipos primitivos nos son proporcionados por la biblioteca estándar del lenguaje (int, char, bool, etc.). En el Cálculo Lambda estos tipos primitivos serán definidos por funciones anónimas a las que convendremos un nombre para facilitar su representación[5].

4.1 Booleanos y operadores lógicos

Las constantes booleanas True y False son entidades opuestas, una es el valor contrario de la otra. Si tratemos de trasladar esta idea a un par de funciones podemos pensar en algo como: la función que toma dos argumentos y regresa el primero es el opuesto de la función que toma dos argumentos y regresa el segundo[6]:

$$\lambda x.\lambda y.x$$

$$\lambda x.\lambda y.y$$

Estas entidades están haciendo exactamente lo contrario que hace la otra, como las funciones son objetos primitivos en el Cálculo Lambda entonces definiremos este par de elementos como nuestro True y nuestro False respectivamente.

De esta forma podemos definir funciones que a su vez, construyan la lógica booleana y nos permitan operar instrucciones de control.

Definición 4.1 (Operadores lógicos para el cálculo Lambda)**.** Definimos las constantes booleanas True y False junto con los operadores NOT, AND e IF con las siguientes funciones[a]:

- True $=_{def}$ $\lambda x.\lambda y.x$
- False $=_{def}$ $\lambda x.\lambda y.y$
- NOT $=_{def}$ $\lambda z.z(\text{False})(\text{True})$
- AND $=_{def}$ $\lambda x.\lambda y.xy(\text{False})$
- If $=_{def}$ $\lambda f.\lambda a.\lambda b.fab$

Es importante mencionar que para estas definiciones estamos abusando de la notación al no escribir la definición de las constantes y usar su nombre directamente. Esto es para ahorrar espacio en los desarrollos y estará permitido su uso en los ejercicios posteriores.

[a] Definición formulada de [108], [109] y [110]

Para el operador NOT la lógica de ejecución es aplicar la función identidad al parámetro de entrada, si este es True entonces se regresará el primer argumento, en este caso la constante

[5] Información consultada de [108] y [109]
[6] Definición formulada de [108], [110] y [111]

False:
$$(\lambda x.x(\mathsf{False})(\mathsf{True}))\mathsf{True} \rightarrow^*_\beta \mathsf{True}(\mathsf{False})(\mathsf{True}) \rightarrow^*_\beta \mathsf{False}$$

Sí al contrario el parámetro de entrada es False este regresará el segundo argumento, en este caso la constante True:

$$(\lambda x.x(\mathsf{False})(\mathsf{True}))\mathsf{False} \rightarrow^*_\beta \mathsf{False}(\mathsf{False})(\mathsf{True}) \rightarrow^*_\beta \mathsf{True}$$

En el caso del operador AND la lógica subyacentes es que si el primer parámetro es un False no importa el segundo parámetro que reciba, siempre regresaremos la constante False:

$$(\lambda x.\lambda y.xy(\mathsf{False}))\ \mathsf{False}\ w \rightarrow^*_\beta \mathsf{False}\ w\ (\mathsf{False}) \rightarrow^*_\beta \mathsf{False}$$

Si recibe como primer parámetro la constante True entonces el resultado será el segundo parámetro:
$$(\lambda x.\lambda y.xy(\mathsf{False}))\ \mathsf{True}\ w \rightarrow^*_\beta \mathsf{True}\ w\ (\mathsf{False}) \rightarrow^*_\beta w$$

Notemos que la única forma de evaluar la instrucción AND como True es si se recibe esta constante dos veces.

Para el Cálculo Lambda es deseable entonces que los operadores anteriormente definidos, aplicando un número finito de veces la regla de $\rightarrow_\beta$ se obtengan los siguientes resultados:

- $\mathsf{If}\ \mathsf{True}\, e_1\, e_2 \rightarrow^*_\beta e_1$
- $\mathsf{If}\ \mathsf{False}\, e_1\, e_2 \rightarrow^*_\beta e_2$
- $\mathsf{NOT}\ \mathsf{True} \rightarrow^*_\beta \mathsf{False}$
- $\mathsf{NOT}\ \mathsf{False} \rightarrow^*_\beta \mathsf{True}$
- $\mathsf{AND}\ \mathsf{False}\, b \rightarrow^*_\beta \mathsf{False}$
- $\mathsf{AND}\ \mathsf{True}\, b \rightarrow^*_\beta b$

5 Aritmética del Cálculo Lambda

En los lenguajes de programación otro tipo de de dato primitivo que es de vital importancia y que se nos es proporcionado por la biblioteca estándar son los números enteros. Estas entidades también tienen una representación en el Cálculo Lambda como funciones.

La construcción de los números naturales se obtiene al aplicar la función sucesor al cero, de tal forma que no tenemos el número dos, tenemos dos veces la aplicación de la función sucesor al cero: *s(s(0))*, en Cálculo Lambda esta idea se mantiene vigente, los números serán la aplicación anidada de una función a la constante cero.

5.1 Numerales de Church

Church introdujo los numerales como la abstracción de dos parámetros ”s” y ”z” en una función anónima, de tal forma que si se desea representar al n-ésimo número este sea formado por la aplicación de s a z, n veces[7]:

- $\bar{0} =_{def} \lambda s.\lambda z.z$
- $\bar{1} =_{def} \lambda s.\lambda z.sz$

[7]Definición formulada de [108], [109] y [111]

- $\bar{2} =_{def} \lambda s.\lambda z.s(sz)$
- $\bar{n} =_{def} \lambda s.\lambda z.\underbrace{s(\ldots(s}_{n\ veces}\,z)\ldots)$

5.2 Funciones aritméticas

Para los numerales de Church podemos difinir operadores aritméticos similares a los que se utilizan cuando trabajamos con los números enteros, estos operadores nos permitirán computar valores y estarán definidos como una función donde los operandos serán los argumentos de entrada.

Función sucesor

Comenzamos el estudio de las funciones aritméticas para el Cálculo Lambda definiendo la función suc para los numerales de Church, esta función recibe un numeral y construye el sucesor añadiendo un símbolo "s" al inicio.

Definición 5.1 (Función sucesor para los numerales de Church). [a]

$$\mathsf{suc} =_{def} \lambda n \lambda z \lambda s.s(n\ z\ s)$$

[a] Definición extraída de [109] y [111]

Ejercicio 5.1. Utiliza la definición de la función suc para evaluar la siguiente λ-expresión

$$\mathsf{suc}\ \bar{2}$$

$$(\lambda n \lambda z \lambda s.s(n\ z\ s))\ \lambda z' \lambda s'.s'(s'z') \to_\beta$$
$$\lambda z \lambda s.s((\lambda z' \lambda s'.s'(s'z'))\ z\ s) \to_\beta$$
$$\lambda z \lambda s.s(\lambda s'.s'(s'z)\ s) \to_\beta$$
$$\lambda s \lambda s.s(s(sz)) \to_\beta$$
$$= \bar{3}$$

Ejercicio 5.2. Utiliza la definición de la función suc para evaluar la siguiente λ-expresión

$$\mathsf{suc}\ \bar{5}$$

$$(\lambda n \lambda z \lambda s.s(n\ z\ s))\ \lambda z' \lambda s'.s'(s'(s'(s'(s'\ z')))) \to_\beta$$
$$\lambda z \lambda s.s((\lambda z' \lambda s'.s'(s'(s'(s'(s'z')))))\ z\ s) \to_\beta$$
$$\lambda z \lambda s.s(\lambda s'.s'(s'(s'(s'(s\ z))))\ s) \to_\beta$$
$$\lambda s \lambda z.s(s(s(s(s(sz)))))$$
$$= \bar{6}$$

IsZero

Este operador está diseñado para regresar True cuando el numeral que recibe como parámetro corresponde al numeral $\bar{0}$. En caso contrario la función regresa False y está definida de la siguiente forma.

Definición 5.2 (Función IsZero para numerales de Church). [a]

$$\mathsf{IsZero} =_{def} \ \lambda m.m \ \mathsf{True} \ (\lambda x.\mathsf{False})$$

[a]Definición extraída de [2], [5] y [12]

Ejercicio 5.3. Utiliza las definición de la función IsZero para evaluar la siguiente λ-expresión

$$\mathsf{IsZero} \ \overline{2}$$

$$(\lambda m.m \ \mathsf{True} \ (\lambda x.\mathsf{False})) \ \lambda z \lambda s.s(s \ z) \rightarrow_\beta$$

$$(\lambda z \lambda s.s(s \ z)) \ \mathsf{True} \ (\lambda x.\mathsf{False}) \rightarrow_\beta$$

$$(\lambda s.s(s \ \mathsf{True})) \ (\lambda x.\mathsf{False}) \rightarrow_\beta$$

$$(\lambda x.\mathsf{false}) \ ((\lambda x.\mathsf{false}) \ \mathsf{true}))) \rightarrow_\beta$$

$$\rightarrow_\beta (\lambda x.\mathsf{False}) \ \mathsf{false} \rightarrow_\beta$$

$$\mathsf{False}$$

Ejercicio 5.4. Utiliza las definición de la función IsZero para evaluar la siguiente λ-expresión

$$\mathsf{IsZero} \ \overline{0}$$

$$(\lambda m.m\mathsf{True} \ (\lambda x.\mathsf{False})) \ \lambda z \lambda s.z \rightarrow_\beta$$

$$(\lambda z \lambda s.z) \ \mathsf{True} \ (\lambda x.\mathsf{False}) \rightarrow_\beta$$

$$(\lambda s.\mathsf{True}) \ \lambda x.\mathsf{False} \rightarrow_\beta$$

$$\mathsf{True}$$

Suma

Esta función se define de forma similar a la función suma para los números naturales donde se reciben dos argumentos m y n y se aplica n veces la función suc al numeral m.

Definición 5.3 (Definición de la suma para los numerales de Church). [a]

$$\mathsf{sum} =_{def} \ \lambda m \lambda n \lambda s \lambda z.n(m \ z \ s) \ s$$

[a]Definición extraída de [109] y [111]

Ejercicio 5.5. Utiliza la definición de la función sum para evaluar la siguiente λ-expresión

$$\mathsf{sum} \ \overline{2} \ \overline{3}$$

$$= (\lambda m \lambda n \lambda z \lambda s.n(m \ z \ s) \ s) \ \lambda z' \lambda s'.s'(s'z') \ \ \lambda z'' \lambda s''.s''(s''(s''z'')) \rightarrow_\beta$$

$$(\lambda n \lambda z \lambda s.n((\lambda z' \lambda s'.s'(s'z')) \ z \ s) \ s) \ \lambda z'' \lambda s''.s''(s''(s''z'')) \rightarrow_\beta$$

$$\lambda z \lambda s.((\lambda z'' \lambda s''.s''(s''(s''z''))) \ ((\lambda z' \lambda s'.s'(s'z')) \ z \ s) \ s) \rightarrow_\beta$$

$$\lambda z \lambda s.(\lambda z'' \lambda s''.s''(s''(s''z'')) \ ((\lambda s'.s'(s'z)) \ s) \ s) \rightarrow_\beta$$

$$\lambda z \lambda s.(\lambda z'' \lambda s''.s''(s''(s''z'')) \ (s(sz)) \ s) \rightarrow_\beta$$

$$\lambda z \lambda s.(\lambda s''.s''(s''(s''(s(sz)))) \ s) \rightarrow_\beta$$

$$\lambda z \lambda s.s(s(s(s(sz)))) = \overline{5}$$

Ejercicio 5.6. Utiliza la definición de la función sum para evaluar la siguiente λ-expresión

$$\mathsf{sum}\ \overline{4}\ \overline{5}$$

$$= (\lambda m \lambda n \lambda z \lambda s.n(m\ z\ s)\ s)\ \lambda z'\lambda s'.s'(s'(s'(s'z')))\ \lambda z''\lambda s''.s''(s''(s''(s''(s''z'')))) \rightarrow_\beta$$
$$(\lambda n \lambda z \lambda s.n((\lambda z'\lambda s'.s'(s'(s'(s'z')))\ z\ s)\ s))\ \lambda z''\lambda s''.s''(s''(s''(s''(s''z'')))) \rightarrow_\beta$$
$$\lambda z \lambda s(\lambda z''\lambda s''.s''(s''(s''(s''(s''z''))))\ (\lambda z'\lambda s'.s'(s'(s'(s'z')))\ z\ s)\ s) \rightarrow_\beta$$
$$\lambda z \lambda s((\lambda z''\lambda s''.s''(s''(s''(s''(s''z'')))))\ (\lambda s'.s'(s'(s'(s'z)))\ s)\ s) \rightarrow_\beta$$
$$\lambda z \lambda s(\lambda z''\lambda s''.s''(s''(s''(s''(s''z''))))\ s(s(s(sz)))\ s) \rightarrow_\beta$$
$$\lambda z \lambda s(\lambda s''.s''(s''(s''(s''(s''(s(s(s(sz)))))))))\ s) \rightarrow_\beta$$
$$\lambda s.\lambda z.s(s(s(s(s(s(s(s(sz)))))))) = \overline{9}$$

Producto

Para el producto de m y n la idea es anidar la operación: $\mathsf{sum}\ \overline{n}\ \overline{n}$, m veces reemplazando las apariciones de de la variable s en el numeral m por $\mathsf{sum}\ \overline{n}$. Esta operación está definida de la siguiente manera:

Definición 5.4 (Definición del producto para los numerales de Church.)**.** [a]

$$\mathsf{prod} =_{def}\ \lambda m.\lambda n.m\ \overline{0}\ (\mathsf{sum}\ n)$$

[a] Definición extraída de [109] y [111]

Ejercicio 5.7. Utiliza la definición de la función prod para evaluar la siguiente λ-expresión

$$\mathsf{prod}\ \overline{2}\ \overline{3}$$

$$\big(\lambda m \lambda n.m\ \overline{0}\ (\mathsf{sum}\ n)\big)\ \lambda z \lambda s.s(sz)\ \overline{3} \rightarrow_\beta$$
$$\big(\lambda n.(\lambda z \lambda s.s(sz))\ \overline{0}\ (\mathsf{sum}\ n)\big)\ \overline{3} \rightarrow_\beta$$
$$(\lambda n.\big(\lambda s.s(s\ \overline{0})\big)\ (\mathsf{sum}\ n))\ \overline{3} \rightarrow_\beta$$
$$\big(\lambda n.(\mathsf{sum}\ n\ (\mathsf{sum}\ n\ \overline{0})\big)\ \overline{3} \rightarrow_\beta$$
$$(\mathsf{sum}\ \overline{3}\ (\mathsf{sum}\ \overline{3}\ \overline{0})) =$$
$$\lambda m \lambda n \lambda s \lambda z.n(m\ z\ s)\ s\ \lambda z'\lambda s'.s'(s'(s'z'))\ (\lambda m'\lambda n'\lambda s''\lambda z''.n'(m'\ z''\ s'')\ s''\ \lambda z'''\lambda s'''.s'''(s'''(s'''z'''))\ \lambda z''''\lambda s''''.z'''')$$

resolveremos por partes la suma dado que la notación crece muy rápido y es poco legible

$$\mathsf{sum}\ \overline{3}\ \overline{0} = \big(\lambda m \lambda n \lambda z \lambda s.n(m\ z\ s)\ s\big)\ \lambda z'\lambda s'.z'\ \overline{3}$$
$$\rightarrow_\beta (\lambda n \lambda z \lambda s.n(\big(\lambda z'\lambda s'.z'\big)\ z\ s)\ s)\ \overline{3}$$
$$\rightarrow_\beta (\lambda n \lambda z \lambda s.n\big(\lambda s'.z\ s\big)\ s)\ \overline{3}$$
$$\rightarrow_\beta \big(\lambda n \lambda z \lambda s.n\ z\ s\big)\ \overline{3}$$
$$\rightarrow_\beta \lambda z \lambda s.(\overline{3}\ z\ s\ ) =\ \lambda z \lambda s.(\big(\lambda z'\lambda s'.s'(s'(s'z'))\big)\ z\ s)$$
$$\rightarrow_\beta \lambda z \lambda s.(\lambda s'.s'\big(s'(s'(z))\big)\ s)$$
$$\rightarrow_\beta \lambda z \lambda s.s(s(sz)) = \overline{3}$$
$$\mathbf{sum}\ \ \overline{3}\ \overline{3} = \big(\lambda m \lambda n \lambda z \lambda s.n(m\ z\ s)\ s\big)\ \lambda z'\lambda s's'(s'(s'.z'))\ \overline{3}$$

$$\to_\beta (\lambda n\lambda z\lambda s.n(\big(\lambda z'\lambda s's'(s'(s'.z'))\big)\ z\ s)\ s)\ \overline{3}$$
$$\to_\beta (\lambda n\lambda z\lambda s.n(\big(\lambda s's'(s'(s'.z))\big)\ s)\ s)\ \overline{3}$$
$$\to_\beta \big(\lambda n\lambda z\lambda s.n\ (s(s(s.z)))\ s\big)\ \overline{3}$$
$$\to_\beta \lambda z\lambda s.((\overline{3})\ (s(s(s.z)))\ s) = \lambda m\lambda n\lambda z\lambda s.(\big(\lambda z'\lambda s'.s'(s'(s'z'))\big)\ (s(s(s.z)))\ s)$$
$$\to_\beta \lambda z\lambda s.(\big(\lambda s'.s'(s'(s'(s(s(s.z))))))\big)\ s)$$
$$\to_\beta \lambda z\lambda s.\lambda s.s(s(s(s(s(s.z))))) = \overline{6}$$

6 Datos estructurados en el Cálculo Lambda

En el Cálculo Lambda es posible definir estructuras para almacenar información junto con las funciones para recuperar los elementos guardados en estas. La estructura más simple que revisaremos serán las tuplas junto con sus proyecciones first y second que ayudan a sentar la base para definir una estructura más compleja; las listas, con sus respectivas funciones para obtener la cabeza y la cola. Juntas proveen una estructura de datos para almacenamiento en este sistema.

6.1 Tuplas

La tupla es la estructura que representa a un par compuesto por un elemento izquierdo y un elemento derecho. En el Cálculo Lambda se define como la función que tiene tiene tres argumentos; el elemento izquierdo (en inglés se le conoce como $first$), el elemento derecho (o *second*) y una función b.

Definición 6.1 (Definición de tuplas para el Cálculo Lambda). [a]
Constructor

$$\mathsf{pair} =_{def} \lambda f\lambda s\lambda b.bfs$$

Proyección del primer elemento

$$\mathsf{fst} =_{def} \lambda p.p\ \mathsf{True}$$

Proyección del segundo elemento

$$\mathsf{snd} =_{def} \lambda p.p\ \mathsf{False}$$

Donde las proyecciones se apoyan de la propiedad de los booleanos de regresar el primer argumento para el caso de True y el segundo en el caso de False.

[a] Definición extraída de [12], [110] y [113]

En general se tienen las siguientes reducciones cuando se aplican las proyecciones a las tuplas:

$$\mathsf{fst\ (pair\ a\ b)} \to_\beta^* \mathsf{a}$$
$$\mathsf{snd\ (pair\ a\ b)} \to_\beta^* \mathsf{b}$$

6.2 Listas

Definición 6.2 (Definición de listas en Cálculo Lambda)**.** Las listas en el Cálculo Lambda se apoyan del constructor de la tupla para ir añadiendo elementos anidando esta operación agregando una unidad por aplicación.

Para esta estructura la lista vacía será representada por una λ-expresión que sea α-equivalente a la constante False.[a]

Lista vacía

$$\mathsf{nil} = \mathsf{False}$$

Constructor para listas

$$\mathsf{cons} = \mathsf{pair}$$

Cabeza de una lista

$$\mathsf{head} = \mathsf{fst}$$

Cola de la lista

$$\mathsf{tail} = \mathsf{snd}$$

[a] Definición extraída de [12], [110] y [113]

7 Propiedades semánticas del Cálculo Lambda

7.1 No terminación

Estudiamos la propiedad de terminación para las expresiones correctamente formadas de EAB, en donde estas se evaluarán a un valor en un número finito de pasos (cuando la evaluación ha llegado a este punto se le conoce como forma normal). No obstante esta propiedad no se cumple para el Cálculo Lambda como se puede ver a continuación:

Ejercicio 7.1. Demuestra o da un contra ejemplo de por qué la propiedad de terminación para el Cálculo Lambda es válida[a].

Consideremos el siguiente par de λ-expresiones:

$$\omega = \lambda x.xx$$

$$\Omega = \omega\omega$$

$$\Omega = \omega\omega = (\lambda x.xx)\omega\omega \to_\beta \omega\omega$$

En general esta expresión cumple que en cada β-reducción se tiene la misma expresión Ω. Por lo tanto una expresión bien formada del Cálculo Lambda no necesariamente tiene una forma normal.

[a] Ejemplo extraído de [120]

7.2 No determinismo

En los lenguajes de programación una propiedad deseable es la propiedad determinista de los programas, es decir, que la evaluación que se haga para una expresión sea siempre la misma. El Cálculo Lambda carece de esta propiedad. Dependiendo del segmento que escogemos para aplicar la β-reducción el siguiente paso de la evaluación puede diferir[8]. Tomemos como ejemplo la siguiente λ-expresión[9]:

$$(\lambda z.(\lambda y.z)a)b$$

Si consideramos la aplicación para la expresión completa, se obtiene la siguiente evaluación:

$$\underline{(\lambda z.(\lambda y.z)a)b} \rightarrow_\beta (\lambda y.b)a$$

Si por el contrario consideramos la aplicación la λ-expresión interna se obtiene:

$$(\lambda z.\underline{(\lambda y.z)a})b \rightarrow_\beta (\lambda z.z)b$$

En ambas evaluaciones el resultado al que se llega al terminar de evaluar la expresión es la variable b sin importar cuál se haya escogido para la evaluación.

Esta característica del Cálculo Lambda no supone una desventaja por el principio de confluencia.

7.3 Confluencia

La propiedad de confluencia nos asegura que dadas dos dos evaluaciones distintas para la misma λ-expresión, estas convergen en un término común en un punto de su evaluación:

Teorema 7.1 (Propiedad de Church-Rosser)**.** [a]

Si $e \rightarrow_\beta^* e_1$ y $e \rightarrow_\beta^* e_2$ entonces existe un término t tal que $e_1 \rightarrow_\beta^* t$ y $e_2 \rightarrow_\beta^* t$.

[a] La demostración queda fuera del alcance de este manual pero se puede consultar en [114]

Corolario 7.2 (Unicidad de formas normales)**.** Para cualquier λ-expresión e si $e \rightarrow_\beta^* e_f$ y $e \rightarrow_\beta^* e'_f$ tal que $\neg\exists\ e_{final}\ y\ e'_{final}$ donde $e_f \rightarrow_\beta e_{final}\ \ y\ \ e'_f \rightarrow_\beta\ e'_{final}$ entonces $e_f = e'_f$ salvo α-equivalencias.

8 Combinadores de punto fijo

El Cálculo Lambda por si mismo no posee un mecanismo iterativo que nos permita formar alguna secuencia de control como los ciclos `for` o `while` como en la mayoría de los lenguajes de programación modernos. Su naturaleza funcional nos hace preguntarnos si podemos emplear un mecanismo mas afín como la recursión para definir funciones que iteren sobre algún parámetro.

Para este fin se precisa de la introducción de los combinadores de punto fijo que capturan la esencia del principio de recursión general en computación:

$$rec\ F \ = \ F\ (rec\ F)$$

De esta forma, si definimos una λ-expresión rec que permita la "auto-aplicación" de si misma a una función que tome como argumento, entonces podemos definir cualquier función

[8] Definición formulada de [119]
[9] Ejemplo extraído de [5] y [12]

recursiva aplicando n veces el mismo procedimiento.

$$rec\ F = F(rec\ F) = F(\ F(rec\ F)) =\ ...\ F(\ F(\ F(\ ...\ )))$$

Definición 8.1 (Combinador de punto fijo)**.** Un λ-expresión cerrada F es un combinador de punto fijo sí y sólo si cumple alguna de las siguientes condiciones[a]:

1. $F\,g \rightarrow^*_\beta g\,(F\,g)$
2. $F\,g \equiv_\beta g\,(F\,g)$ es decir, existe un término t tal que $F\,g \rightarrow^*_\beta t$ y $g\,(F\,g) \rightarrow^*_\beta t$

[a]Definición formulada de [5], [12] y [113]

Definición 8.2 (Combinador Y)**.** Existen diferentes λ-expresiones que cumplen con la definición provista anteriormente. Uno de los combinadores más sencillos y populares es el Combinador Y que se define de la siguiente forma[a]:

$$Y =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

[a]Definición extraída de [113]

Ejercicio 8.1. Demuestra que el combinador Y es un combinador de punto fijo.

$$\begin{array}{ll} & Y\,g \\ & (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\,g \\ \rightarrow_\beta & (\lambda x.g(xx))(\lambda x.g(xx)) \\ \rightarrow_\beta & g((\lambda x.g(xx))(\lambda x.g(xx))) \end{array}$$

Tomando la segunda parte de la igualdad y desarrollando obtenemos

$$\begin{array}{ll} & g\,(Y\,g) \\ & g\,((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\,g) \\ \rightarrow_\beta & g\,((\lambda x.g(xx))(\lambda x.g(xx))) \end{array}$$

de esta forma podemos concluir que $Y\,g \equiv_\beta g\,(Y\,g)$, entonces Y es un combinador de punto fijo.

9 Ejercicios para el lector

Ejercicio 9.1. Supón que se requiere extender la definición 4.1 para booleanos y sus operadores en el Cálculo Lambda implementando el siguiente operador:

$$\mathsf{OR}\ a\ b$$

Donde el operador se comporta de la siguiente forma:

$$OR\ \mathsf{true}\ b \rightarrow^*_\beta\ \mathsf{true}$$

$$OR\ \mathsf{false}\ b \rightarrow^*_\beta\ b$$

Proporciona la λ-expresión que implemente dicha instrucción.

Ejercicio 9.2. Verifica las propiedas de la semántica operacional para cada operador booleano según la definición 4.2:

$$\mathsf{IF}\,\mathsf{true}\,e_1\,e_2 \to_\beta^* e_1$$
$$\mathsf{IF}\,\mathsf{false}\,e_1\,e_2 \to_\beta^* e_2$$
$$\mathsf{NOT}\,\mathsf{true} \to_\beta^* \mathsf{false}$$
$$\mathsf{NOT}\,\mathsf{false} \to_\beta^* \mathsf{true}$$
$$\mathsf{AND}\,\mathsf{false}\,b \to_\beta^* \mathsf{false}$$
$$\mathsf{AND}\,\mathsf{true}\,b \to_\beta^* b$$
$$\mathsf{OR}\,\mathsf{true}\,b \to_\beta^* \mathsf{true}$$
$$\mathsf{OR}\,\mathsf{false}\,b \to_\beta^* b$$

Ejercicio 9.3. Resuelve las siguientes operaciones de los numerales de Church (el abuso de notación para simplificar λ-expresiones está permitido con el fin de tener una representación más compacta cuando sea posible)

$$\mathsf{prod}\ (\mathsf{sum}\ \overline{7}\ \overline{4})\ \overline{2}$$
$$\mathsf{IsZero}(\mathsf{prod}\ \overline{1}\ \overline{0})$$
$$\mathsf{IsZero}(\mathsf{sum}\ \overline{3}\ (\mathsf{prod}\ \overline{5}\ \overline{4}))$$

Ejercicio 9.4. Siguiendo la definición 6.2 de listas en el Cálculo Lambda define la función IsNil que se comporta de la siguiente forma

$$\mathsf{IsNil}\ []\ =\ \mathsf{true}$$
$$\mathsf{IsNil}\ (x : xs)\ =\ \mathsf{false}$$

Ejercicio 9.5. Encuentra la forma normal de las siguientes λ-expresiones, si no es posible explica por qué.

$$(\lambda y.yy)(\lambda z.zz)(\lambda\alpha.\lambda\beta.\alpha\beta\alpha)\ \mathsf{false}\ \overline{0}$$
$$((\lambda z.\lambda y.\lambda u.uzy)(\lambda x.x)(\lambda wv.wv)\ \overline{0}\ \overline{1})\ \mathsf{true}$$

Ejercicio 9.6. Para cada una de las siguientes λ-expresiones demuestra que son combinadores de punto fijo.

$$\text{Turing } V\ =\ UU \text{ en donde } U\ =\ \lambda f.\lambda x.x(ffx)$$
$$\text{Estricto } Z\ =\ \lambda f.(\lambda x.f(\lambda v.xxv))(\lambda x.f(\lambda v.xxv))$$

Ejercicio 9.7. Utilizando cualquiera de los combinadores de punto fijo implementa la función recursiva $\mathsf{factorial}$ para los numerales de Church en el Cálculo Lambda.

$$\mathsf{factorial}\ 1\ =\ 1$$
$$\mathsf{factorial}\ n\ =\ n\ *\ \mathsf{factorial}\ n-1$$

Adicionalmente muestra la ejecución para n = $\overline{3}$

Ejercicio 9.8. Utilizando algún combinador de punto fijo implementa la función recursiva $\mathsf{fibonacci}$ para los numerales de Church en el Cálculo Lambda.

$$\begin{aligned} \mathsf{fibonacci}\ 0 &= 0 \\ \mathsf{fibonacci}\ 1 &= 1 \\ \mathsf{fibonacci}\ n &= \mathsf{fibonacci}\ n-1\ +\ \mathsf{fibonacci}\ n-2 \end{aligned}$$

Adicionalmente muestra la ejecución para n = $\overline{3}$

Ejercicio 9.9. Utilizando el combinador Y, define la función concat para la representación de listas en el Cálculo Lambda.

$$\begin{aligned} \mathsf{concat}\ []\ (y : ys) &= (y : ys) \\ \mathsf{concat}\ (x : xs)\ (y : ys) &= x : \mathsf{concat}(xs\ (y : ys)) \end{aligned}$$

Ejercicio 9.10. Utilizando cualquier combinador de punto fijo resuelve lo siguiente:

Define la función reversa para las listas del Cálculo Lambda.

$$\begin{aligned} \mathsf{rev}\ [] &= [] \\ \mathsf{rev}\ (x : xs) &= \mathsf{rev}\ xs +\!+ [x] \end{aligned}$$

Aplica la función para obtener la reversa de la lista: $\mathsf{pair\ 1\ (pair\ 2\ (pair\ 3\ False))}$

Ejercicio 9.11. Utilizando el combinador Y contesta lo siguiente:

- Da una definición de la función exponencial para los numerales de Church.
- Resuelve la función para $\mathsf{exp}\ \overline{4}\ \overline{2}$.

Ejercicio 9.12. Supón que se requieren extender las estructuras de datos para el Cálculo Lambda con la introducción de árboles binarios, en donde cada nodo puede almacenar un valor (numerales de Church o booleanos). Contesta lo que se te pide a continuación:

- Proporciona la λ-expresión para definir el árbol binario vacío.
- Proporciona la λ-expresión para definir el constructor de un nodo que contiene un elemento, un árbol binario izquierdo y un árbol binario derecho.
- Define la función fold que nos permita iterar el árbol y computar alguna operación sobre sus elementos (deja indicado el operador con el símbolo ”op”).

Ejercicio 9.13. Para cada una de los siguientes λ-expresiones menciona a cuál definición pertenece.

- $\lambda s \lambda z.s(s(s(s(sz)))))$
- $\lambda f.\lambda a.\lambda b.fab$
- $\lambda f \lambda s \lambda b.bfs$
- $\lambda m \lambda n \lambda s \lambda z.n(m\ z\ s)\ s$

Capítulo 6

MinHS

Con la teoría que hemos revisado hasta el momento en nuestro lenguaje EAB cuya definición engloba booleanos e instrucciones lógicas, a los números naturales junto con sus operadores y la implementación del Cálculo Lambda, nos es natural preguntarnos ¿cómo es la implementación de un lenguaje de características similares en una computadora?
Para resolver esta pregunta definiremos un lenguaje que contenga un subconjunto de instrucciones de algún otro lenguaje cuya implementación ya exista para estas.

Por su naturaleza funcional tomaremos como caso de estudio una versión simplificada del lenguaje de programación Haskell. Este posee características importantes para el enfoque de este curso que serán de interés trasladar a EAB.

Hasta el momento se ha trabajado con tipos de datos primitivos, pero no hemos hablado de las reglas de la semántica estática que asignan el tipo a las expresiones. Esto constituye un problema al poder escribir expresiones de EAB sintácticamente correctas pero cuya evaluación no termine en un valor puesto que la ejecución se detendría en algún momento al no hallar una regla de la semántica dinámica que nos permita continuar.

Con esto en mente el objetivo de este capítulo será definir un lenguaje similar que nos permita tener todos los tipos de datos de EAB, el sistema de tipos de Haskell y las características más importante del mismo: evaluación perezosa, pureza funcional y asignación de tipos explícita y estática. A este lenguaje lo llamaremos MinHS al ser una versión acotada de Haskell pero servirá para ilustrar los conceptos que introduciremos en este capítulo.

Objetivo

Proveer la definición del lenguaje funcional MinHS que preserve las características principales del lenguaje de programación Haskell a manera de ilustrar una implementación concreta de los conceptos que se han revisado hasta este momento: sintaxis, semántica, Cálculo Lambda y recursión, haciendo especial énfasis en el sistema de tipos de este lenguaje y sus propiedades.

Planteamiento

El desarrollo del capítulo se plantea de forma similar a como lo hemos hecho con el lenguaje EAB dividiendo su definición en sintaxis concreta, sintaxis abstracta, semántica dinámica y es aquí en donde se introducirá una nueva capa para estudiar la asignación de tipos de las expresiones. Finalmente concluiremos mencionando brevemente las propiedades que este pequeño lenguaje posee.

1 Sintaxis de MinHS

1.1 Sintaxis concreta

Para construir el sistema de tipos de MinHS, se necesita introducir una nueva expresión de tipo al cual las variables pertenecen que será representado por la letra T.

El tipo de las λ-expresiones será representado por la expresión $T_1 \Rightarrow T_2$ en la sintaxis concreta donde T_1 representa el tipo de la variable ligada en el cuerpo de la expresión y T_2 es el tipo que regresa la evaluación de la expresión.

Finalmente se introduce la expresión para funciones recursivas representadas por la expresión

$$\mathsf{letrec}\ var =\ e_1\ \mathsf{in}\ e_2\ \mathsf{end}$$

Definición 1.1 (Sintaxis Concreta de MinHS)**.** A continuación definimos los elementos que componen a las expresiones válidas de MinHS.[a]

El tipo de las expresiones ahora figura como parte de la sintaxis concreta.

Expresiones	e	$::=$	$var \mid n \mid b \mid (e) \mid e_1 \otimes e_2 \mid e_1\ e_2$
		$\mid$	$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$
		$\mid$	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$
		$\mid$	$\mathsf{fun}\ x :: T \to e$
		$\mid$	$\mathsf{recfun}\ f :: (T_1 \Rightarrow T_2)\ x \to e$
Tipos	T	$::=$	$Bool \mid Nat \mid T_1 \Rightarrow T_2$
Variables	var	$::=$	$x \mid y \mid \dots$
Números	n	$::=$	$0 \mid 1 \mid \dots$
Booleanos	b	$::=$	$\mathsf{True} \mid \mathsf{False}$
Operadores Infijos	$\otimes$	$::=$	$+ \mid * \mid - \mid = \mid < \mid >$

[a]Definición extraída de [5], [12], [115] y [116]

Ejercicio 1.1. Escribe la definición de los siguientes instrucciones utilizando la sintaxis concreta de MinHS

- Proporciona la definición de la función sucesor que recibe un número y regresa el sucesor:
$$\mathsf{suc} =_{def} \mathsf{fun}\ x\ ::\ Nat \to x + 1$$
- Proporciona la definición de la función que recibe un número y decide si es cero:
$$\mathsf{IsZero} =_{def} \mathsf{fun}\ x\ ::\ Nat \to\ \mathsf{if}\ (x\ =\ 0)\ \mathsf{then\ True\ else\ False}$$
- Proporciona la definición de una función que recibe un número y regresa el factorial:
$$\mathsf{fact} =_{def} \mathsf{recfun\ fact}\ ::\ (Nat \Rightarrow Nat)\ x \to\ \mathsf{if\ IsZero}(x)\ \mathsf{then}\ 1\ \mathsf{else}\ n\ *\ \mathsf{fact}\ (n-1)$$
- Proporciona la definición de una función que recibe dos números y regresa su producto:
$$\mathsf{prod} =_{def} \mathsf{fun}\ x\ ::\ Nat \to\ \mathsf{fun}\ y\ ::\ Nat \to\ x * y$$

1.2 Sintaxis abstracta

Definición 1.2 (Sintaxis abstracta de MinHS)**.** Una vez definidas las reglas para generar programas en MinHS empleando la sintaxis concreta del lenguaje, podemos definir su representación intermedia como un árbol de sintaxis abstracta[a].

Valores y variables

$$\frac{n \in \mathbb{N}}{num[n]\ asa} \qquad \frac{}{bool[True]\ asa} \qquad \frac{}{bool[False]\ asa}$$

$$\frac{x \in\ Variables}{x\ asa}$$

Operadores

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

Condicional

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

Asignaciones locales

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

Definición de funciones

$$\frac{t\ asa}{fun(T, x.t)\ asa} \qquad \frac{t\ asa}{recfun(T, f.x.t)\ asa}$$

Aplicación de función

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

Operador de punto fijo

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

El operador de punto fijo fix es una implementación interna para evaluar expresiones recursivas. Como no está asociado a ninguna expresión de la sintaxis concreta es imposible que un usuario lo pueda instanciar directamente.

El operador $recfun$ tiene dos variables ligadas en el cuerpo de la definición: el nombre de la función y la variable que recibe como argumento.

[a]Definición extraída de [5], [12], [115] y [116]

2 Semántica de MinHs

2.1 Sistemas de tipos

Los tipos en el contexto de los lenguajes de programación son la descripción abstracta de una colección de valores que nos permite agruparlos y emplearlos de manera similar aún sin saber el contentido específico que estos pudieran tener.

En los lenguajes de programación los sistemas de tipos brindan información adicional sobre la evaluación de las expresiones, qué valores debemos esperar recibir y regresar al concluir la ejecución de nuestro programa para definir restricciones que nos permitan tener seguridad y congruencia en los datos[1].

Este sistema estará definido en la siguiente capa de MinHS de forma similar como fue trabajado con anterioridad con el lenguaje EAB, dicho nivel es la semántica estática que define un conjunto de juicios para brindar la seguridad de evaluación a las expresiones.

2.2 Semántica estática

Haskell es un lenguaje de programación que implementa un sistema de verificación de tipos estático, esto quiere decir que la evaluación de las expresiones solo es posible cuando

[1]definición formulada a partir de [96]

el programa es congruente con las reglas definidas por su sistema de tipos, descartando la evaluación de todas aquellas expresiones que estén bien formadas pero que no respeten las restricciones de este sistema[2].

Definición 2.1 (Semántica estática)**.** La semántica estática nos ayuda a definir criterios para evaluar los programas de MinHS con la información que se pueda inferir acerca de los parámetros que una expresión toma como argumento o el tipo que esta regresa al concluir su evaluación. Para este propósito definimos el siguiente juicio[a]:

$$\Gamma \vdash t : T$$

El cual 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 $\{x_1 : T_1 \ldots x_n : T_n\}$.
Adicionalmente se utiliza la notación $\Gamma, x : T$ para indicar el conjunto $\Gamma \cup \{x : T\}$.

Variables

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

Valores numéricos

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

Valores Booleanos

$$\frac{}{\Gamma \vdash \mathit{bool}[\mathit{False}] : \mathit{Bool}} \qquad \frac{}{\Gamma \vdash \mathit{bool}[\mathit{True}] : \mathit{Bool}}$$

Operadores

$$\frac{\Gamma \vdash t_1 : \mathit{Nat} \quad \Gamma \vdash t_2 : \mathit{Nat}}{\Gamma \vdash \mathit{sum}(t_1, t_2) : \mathit{Nat}} \qquad \frac{\Gamma \vdash t_1 : \mathit{Nat} \quad \Gamma \vdash t_2 : \mathit{Nat}}{\Gamma \vdash \mathit{prod}(t_1, t_2) : \mathit{Nat}}$$

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

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

Condicional

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

Asignaciones Locales

$$\frac{\Gamma \vdash t_v : T \quad \Gamma, x : T \vdash t_b : \mathit{St}}{\Gamma \vdash \mathit{let}(t_v, x.t_b) : \mathit{St}}$$

Funciones

$$\frac{\Gamma, x : T \vdash t : \mathit{St}}{\Gamma \vdash fun(T, x.t) : T \to \mathit{St}} \qquad \frac{\Gamma \vdash f : T \to S, x : T \vdash t : S}{\Gamma \vdash recfun(T \to S, f.x.t) : T \to S}$$

Aplicación de función

$$\frac{\Gamma \vdash t_f : T \to \mathit{St} \quad \Gamma \vdash t_p : T}{\Gamma \vdash \mathit{app}(t_f, t_p) : \mathit{St}}$$

[2]definición formulada a partir de [75]

Operador de punto fijo

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

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

[a]Definición extraída de [5], [12], [115] y [116]

Ejercicio 2.1. Para cada expresión de MinHS enlistada a continuación obtén su representación en sintaxis abstracta y aplica las reglas de la semántica estática para hacer el análisis de tipos de la expresión.

- 1 + (7 - 1)

  Representación en sintaxis abstracta:

  $$sum(num[1], res(num[7], num[1]))$$

  Análisis de tipo aplicando la semántica estática:

  $$\dfrac{\dfrac{}{\vdash num[1] \ : Nat} \qquad \dfrac{\dfrac{}{\vdash num[7] \ : \ Nat} \qquad \dfrac{}{\vdash num[1] \ : \ Nat}}{\vdash res(num[7], num[1]) \ : \ Nat}}{\vdash sum(num[1], res(num[7], num[1])) : Nat}$$

- lam x :: $Nat \to \ x > 1$

  Representación en sintaxis abstracta:

  $$fun(Nat, x.gt(x, num[1]))$$

  Análisis de tipo aplicando la semántica estática:

  $$\dfrac{\dfrac{\dfrac{}{x \ : \ Nat \vdash x \ : \ Nat} \qquad \dfrac{}{x \ : \ Nat \vdash num[1] \ : \ Nat}}{x \ : \ Nat \vdash gt(x, num[1]) : Bool}}{\vdash fun(Nat, x.gt(x, num[1])) \ : \ Nat \to Bool}$$

- let $x \ = \ 3$ in if $x<$ 7 then 7 else $x \ + \ (7 \ - \ x)$ end

  Representación en sintaxis abstracta:

  $$let(num[3], x.if(lt(x, num[7]), num[7], sum(x, res(num[7], x))))$$

  Análisis de tipo aplicando la semántica estática:

  $$\dfrac{\dfrac{}{\vdash num[3] \ : \ Nat} \qquad \dfrac{\dfrac{(A)\cdots}{x : Nat \vdash lt(x, num[7]) : Bool} \qquad \dfrac{}{x : Nat \vdash num[7] : Nat} \qquad \dfrac{(B)\cdots}{x : Nat \vdash sum(x, res(num[7], x)) : Nat}}{x : Nat \vdash if(lt(x, num[7]), num[7], sum(x, res(num[7], x)) \ : \ Nat}}{\vdash let(num[3], x.if(lt(x, num[7]), num[7], sum(x, res(num[7], x)))) \ : \ Nat}$$

  (A) Análisis semántico para lt

  $$\dfrac{\dfrac{}{x : Nat \vdash x : Nat} \qquad \dfrac{}{x : Nat \vdash num[7] : Nat}}{x : Nat \vdash lt(x, num[7]) : Bool}$$

  (B) Análisis semántico para sum

  $$\dfrac{\dfrac{}{x : Nat \vdash x : Nat} \qquad \dfrac{\dfrac{}{x : Nat \vdash num[7] : Nat} \qquad \dfrac{}{x : Nat \vdash x : Nat}}{x : Nat \vdash res(num[7], x) : Nat}}{x : Nat \vdash sum(x, res(num[7], x)) : Nat}$$

2.3 Semántica dinámica

Definición 2.2 (Semántica operacional de paso pequeño)**.** Para concluir con la definición de MinHS enunciaremos las reglas de la semántica operacional.

Es importante observar que la evaluación perezosa característica de Haskell se debe particularmente a los operadores app y let dado que la sustitución se hace sin evaluar la expresión que dará el valor en la variable ligada[a].

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

Condicional

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

$$\frac{a_c \to a'_c}{if(a_c, a_t, a_e) \to if(a'_c, a_t, a_e)}\ \textit{if}$$

Asignaciones locales

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

Aplicación de función

$$\frac{a_f \to a'_f}{app(a_f, a_p) \to app(a'_f, a_p)}\ \textit{appL}$$

$$\frac{}{app(fun(x.a_b), a_p) \to a_b[x := a_p]}\ \textit{app}$$

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

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

Operador de punto fijo

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

Operadores

Para acotar el listado de reglas se omite la representación para los operadores cuya definición es la misma a la que se dió para EAB en el capítulo 4: Semántica. Las reglas para operadores booleanos de comparación $<, >, =$ se definen de manera análoga.

Es importante notar que el operador *fix* puede aplicarse a sí mismo de manera indefinida, reemplazando las apariciones del nombre ligado en el cuerpo por la misma definición *fix*.

Esto supone un problema que revisaremos en la siguiente sección ya que deriva en estados ciclados donde el programa puede aplicar esta regla sin llegar a ningún valor si no se define con cuidado la función que estará en el cuerpo de la expresión.

[a] Definición extraída de [5], [12], [115] y [116]

3 Propiedades de MinHS

3.1 Seguridad del lenguaje

Esta propiedad proporciona un mecanismo para corroborar que derivado de la evaluación de una expresión que cumple las restricciones dictadas por el sistema de tipos del lenguaje, se obtendrá un valor. Vinculando las definiciones para la semánitca estática concerniente al tipo de las expresiones y la semántica dinámica concerniente a la evaluación de las mismas[3].

En el contexto de MinHS nos interesa que a una expresión correctamente construida y correctamente tipificada se le pueda aplicar alguna de las reglas definidas por la semántica dinámica (progreso). Adicionalmente nos interesa que el tipo de las expresiones sea único y que a cada paso en la evaluación este se conserve (preservación). A continuación daremos una definición formal para estas propiedades.

- Progreso: un programa que cumple con las restricciones del sistema de tipos no se bloquea. Este comportamiento está capturado en la definición de progreso enlistada a continuación para las expresiones de MinHS:

  Definición 3.1 (Propiedad de progreso)**.** Si $\Gamma \vdash t : T$ para algún tipo T entonces se cumple una de los siguientes dos casos[a]:
  - t es un valor
  - Existe una expresión t' tal que $t \to t'$

  [a] Definición extraída de [5], [12], [117] y [118]

- Preservación: un programa que cumple con las restricciones del sistema de tipos dará como resultado de la evaluación una expresión correctamente tipificada. Esto se puede observar en las siguientes dos propiedades definidas para las expresiones de MinHS:

  Definición 3.2 (Preservación de tipos)**.** Sí $\Gamma \vdash t : T$ y $t \to t'$ entonces $\Gamma \vdash t' : T$[a].

  [a] Definición extraída de [5], [12] y [118]

  Definición 3.3 (Unicidad de tipo)**.** Para cualesquiera Γ y expresión t de MinHS existe a lo más un tipo T de tal forma que se cumple: $\Gamma \vdash t : T$[a].

  [a] Definición extraída de [5], [12]

[3] Definición formulada de [117] y [118]

3.2 No terminación

MinHS hereda un problema similar a lo que sucedió en el Cálculo Lambda con la introducción de los combinadores de punto fijo para implementar la recursión general. Este problema es la existencia de expresiones del lenguaje sintácticamente y semánticamente correctas que producen un estado conocido estado ciclado.

Ejercicio 3.1. Demuestra o da un contra ejemplo de la propiedad de terminación para MinHS[a]

Considera la función identidad que recibe un parámetro y regresa el mismo representada como $f.f$ sí utilizamos esta expresión como el cuerpo de nuestro operador fix obtenemos la expresión:

$$\mathit{fix}(Nat\ f.f) \rightarrow f[f := fix(Nat\ f.f)] = fix(Nat\ f.f) \rightarrow ...$$

Por lo tanto MinHS no posee la propiedad de terminación.

[a] Ejemplo extraído de [120]

4 Ejercicios para el lector

Ejercicio 4.1. Queremos extender la definición de MinHS para agregar el tipo de dato algebraico tupla representado como pair(x,y), donde ambos elementos no necesariamente tienen que tener el mismo tipo, así como las proyecciones fst y snd.

Con la descripción anterior responde los siguientes incisos:

- Define la sintaxis concreta para las tuplas.
- Define la regla de sintaxis abstracta para las tuplas.
- Define la regla de la semántica estática para verificación de tipos para las tuplas.
- Define la(s) reglas de evaluación para las tuplas.
- ¿Cuál es la diferencia entre una expresión que construye la tupla y la proyección que obtiene el elemento?.

Ejercicio 4.2. Para las siguientes expresiones de MinHS determina el tipo y verifica tu respuesta utilizando las reglas de la semántica estática de este lenguaje.

- pair(3, True)
- snd pair(3, True)
- lam x :: $Nat\ \rightarrow\ x+1$
- recfun *fact* :: $(\mathsf{Nat} \Rightarrow \mathsf{Nat})\ x \rightarrow$ if IsZero(x) then 1 else $n\ *$ *fact* $(n-1)$
- let neg = (lam x : $Bool \rightarrow$ if x then False else True) in neg($3 < 2$) end

Ejercicio 4.3. Queremos definir el operador Case con la siguiente sintaxis[a]:

$$\mathsf{Case}\ x\ \mathsf{of}\ x.g_1 \rightarrow\ c1\ ;\ x.g_2 \rightarrow\ c2\ ;\ ...\ \mathsf{end}$$

Donde Case es la generalización de múltiples expresiones if else para evaluar un argumento de entrada con una condicional o ”guardia” (g_n) para determinar el caso al que este es aplicado (C_n). De tal forma que se toma como resultado de la expresión la primera guardia que se evalúe a verdadero de izquierda a derecha (o de arriba hacia abajo como generalmente se escribe el operador Case).

Adicionalmente queremos definir una expresión que por omisión sea el resultado de la evaluación de Case en caso de que ninguna de las guardias se evalúe a verdadero (en este caso se escribe como el último elemento representado como c_d):

$$\mathsf{Case}\ x\ \mathsf{of}\ x.g_1 \to\ c_1\ ;\ x.g_2 \to\ c_2\ ;\ \ldots\ ;\ c_d\ \mathsf{end}$$

Con la especificación anteriormente dada contesta los siguientes incisos:

- Extiende la sintaxis concreta del lenguaje para el operador Case.
- Extiende la sintaxis abstracta del lenguaje para el operador Case.
- Extiende la semántica estática del lenguaje para el operador Case.
- Extiende la semántica dinámica del lenguaje para el operador Case.

[a] Ejercicio extraído de [121]

Ejercicio 4.4. Dada la siguiente expresión Case:

$$\mathsf{Case}\ n\ \mathsf{of}\ n < 0 \to\ \mathsf{True}\ ; n = 0 \to\ \mathsf{True}\ ;\ \mathsf{False}$$

- Evalúala utilizando las reglas de semántica dinámica definidas en el inciso anterior.
- Sí $\Gamma = \{n : Nat\}$ utiliza las reglas de semántica estática definidas en el inciso anterior para obtener el tipo de la expresión.

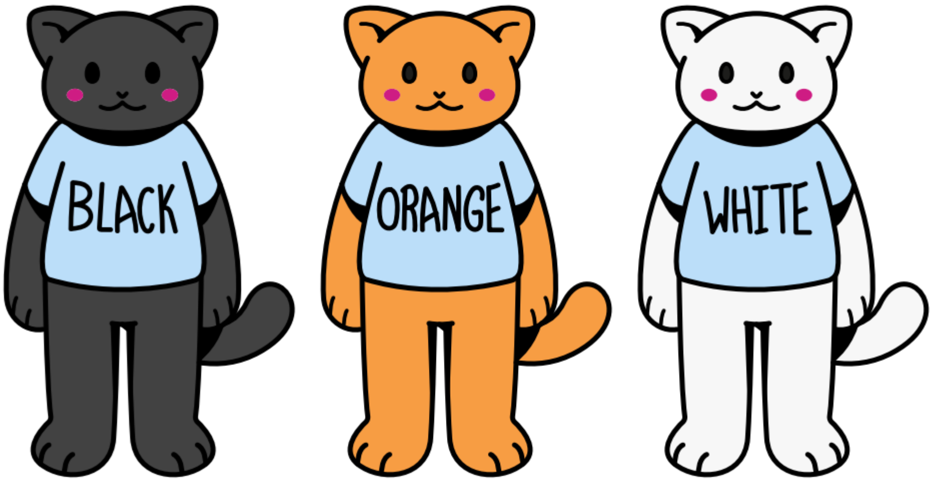
Ejercicio 4.5. Define las siguientes funciones utilizando la sintaxis concreta de MinHS y aplica las reglas de la semántica estática para verificar el tipo de las mismas[a]:

- Define la función exp(n_1, n_2) que eleva el primer argumento a la potencia del segundo.
- Define la función fibonacci(n) que obtiene el n-ésimo elemento de la sucesión de Fibonacci.
- Evalúa la expresión exp(2,2).
- Evalúa la expresión fibonacci(3).

[a] Ejercicio extraído de [121]

Capítulo 7

Inferencia de tipos



Una propiedad importante de Haskell que queremos trasladar a MinHS es el mecanismo de inferencia de tipos. Este mecanismo provee una capa extra que permite trabajar con expresiones aún si estas no poseen un tipo explícito en cada uno de sus parámetros de entrada y valores de retorno.

Anteriormente se definieron mediante la semántica estática las reglas para obtener el tipo de una expresión, no obstante este proceso es dependiente de las anotaciones explícitas de tipo que se encuentran definidas en la sintaxis del lenguaje. Con el mecanismo de inferencia de tipos se pretende desacoplar este conjunto de anotaciones de la sintaxis de MinHS.

Con esto en mente, vamos a formular un sistema de restricciones derivadas de la estructura de la expresión que se desea tipificar, junto con el proceso de estandarización de variables para poder hacer uso de la inferencia de tipos mediante la aplicación del algoritmo para obtener el unificador general μ para expresiones de MinHS.

Objetivo

Definir el sistema de inferencia de tipos para MinHS que nos permita tipificar las expresiones garantizando la seguridad del lenguaje. Para esto, el objetivo principal será proporcionar una definición de las restricciones que cada estructura particular de las expresiones sintácticas de MinHS generan. A estas se les conoce como restricciones de tipo.

Adicionalmente se proporcionará la definición del algoritmo de unificación para aplicar la inferencia de tipos conocido como algoritmo para obtener el unificador general μ.

Planteamiento

En este capítulo revisaremos diferentes procedimientos que se aplican a las expresiones de MinHS para poder garantizar la correcta asignación de un tipo. Estudiaremos la estandarización de variables para obtener α-equivalencias entre las variables ligadas repetidas dentro de una expresión.

Posteriormente vamos a revisar los juicios para generar las restricciones de tipo que cada expresión genera para finalmente definir el algoritmo unificador μ.

1 Estandarización de variables

La semántica estática de MinHS y la de EAB se implementa de forma similar para ir agregando a un contexto (que denotaremos con la letra Γ) aquellas variables definidas en las expresiones junto con el tipo que se espera estas tengan.

En el contexto Γ solo puede existir una sola aparición por variable, es decir, nunca podremos tener $\Gamma = \{x : Nat,\ x : Bool\}$ dado que Γ es un conjunto de variables y tipos, tener la misma variable con dos tipos distintos es una inconsistencia dentro de nuestro programa y constituye un error en el tipificado de la expresión[1].

Para evitar esta situación se define el proceso conocido como estandarización de variables que consiste en brindar una α-equivalencia cuando dos variables ligadas tengan el mismo nombre.

Tomemos como ejemplo la siguiente expresión:

$$\mathsf{let}\ x = \mathsf{True}\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \ \mathsf{let}\ x = 5\ \mathsf{in}\ x < x\ \mathsf{end}\ \ \mathsf{end}$$

En este caso la expresión está bien formada, no hay ninguna regla de la sintaxis concreta que esté mal aplicada para obtener nuestra expresión, la evaluación tampoco comprende un problema dado que la expresión let mas interna se evalúa a verdadero y luego se continúa con la comparación de dos valores booleanos, sin embargo esta expresión nos genera el contexto $\Gamma = \{x : Nat,\ x : Bool\}$ el cual es inconsistente.

El problema puede ser corregido utilizando una α-equivalencia y renombrando las varia-

[1] Definición de semántica estática para MinHS formulada de [5], [12], [115] y [116]

bles x por x_0 y x_1 respectivamente:

$$\textsf{let } x_0 = \textsf{True in if } x_0 \textsf{ then let } x_1 = 5 \textsf{ in } x_1 < x_1 \textsf{ end end}$$

Donde el contexto obtenido será: $\Gamma = \{x_1 : Nat,\ x_0 : Bool\}$

Ejercicio 1.1. Para la siguiente expresión de MinHS expresando el contexto

$$\textsf{let } x = \textsf{False in } x = (\textsf{let } x = 6 \textsf{ in } (x + x + x) > 10 \textsf{ end}) \textsf{ end}$$

$$\Gamma = \{x : Nat,\ x : Bool\}$$

Aplica la estandarización de variables:

$$\textsf{let } x_0 = \textsf{False in } x_0 = (\textsf{let } x_1 = 6 \textsf{ in } (x_1 + x_1 + x_1) > 10 \textsf{ end}) \textsf{ end}$$

$$\Gamma = \{x_1 : Nat, x_0 : Bool\}$$

Ejercicio 1.2. Para la siguiente expresión de MinHS expresando el contexto

$$\textsf{let } x = \textsf{True in } x \textsf{ or } (\textsf{let } x = 5 \textsf{ in } x < 5 \textsf{ end}) \textsf{ end}$$

$$\Gamma = \{x : Bool,\ x : Nat\}$$

Aplica la estandarización de variables:

$$\textsf{let } x_0 = \textsf{True in } x_0 \textsf{ or } (\textsf{let } x_1 = 5 \textsf{ in } x_1 < 5 \textsf{ end}) \textsf{ end}$$

$$\Gamma = \{x_1 : Bool,\ x_0 : Nat\}$$

Ejercicio 1.3. Para la siguiente expresión de MinHS, expresando el contexto

$$\textsf{let } x = 1 \textsf{ in } (\textsf{let } x = \textsf{False in } (\textsf{let } x = fun\ y :: Bool \to y \textsf{ in } x \textsf{ end }) \textsf{ end}) \textsf{ end}$$

$$\Gamma = \{x : Nat, x : Bool,\ x : Bool \to Bool\}$$

Aplica la estandarización de variables:

$$\textsf{let } x_0 = 1 \textsf{ in (let } x_1 = \textsf{False in } (\textsf{let } x_2 = fun\ y :: Bool \to y \textsf{ in } x_2 \textsf{ end}) \textsf{ end}) \textsf{ end}$$

$$\Gamma = \{x_0 : Nat,\ x_1 : Bool,\ x_2 : Bool \to Bool\}$$

Ejercicio 1.4. Propón una semántica estática que resuelva el proceso de estandarización de variables.

Definición 1.1. Definición de la semántica estática para la estandarización de variables[a].

Representaremos la asignación serializada de nuevas variables con una lista de parejas conformada por la variable original seguida de su nueva asignación mediante la aplicación del siguiente juicio:

$$v_n : x_n \ ...\ v_0 : x_0 \ \sim_\alpha \ e\ asa$$

Que se lee como:

"El árbol de sintaxis abstracta e es congruente con la lista $v_n : x_n \ ...\ v_0 : x_0$ bajo α-equivalencia aplicando la(s) sustitución(es) sí existen en la lista".

Variables

$$\frac{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ v_n}{x_n} \qquad \frac{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ y}{y}$$

Valores numéricos

$$\frac{}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ num[n]}$$

Valores Booleanos

$$\frac{}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ \mathit{False}} \qquad \frac{}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ \mathit{True}}$$

Operadores

$$\frac{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_1 \qquad v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_2}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ Op(t_1, t_2)}$$

Condicional

$$\frac{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_c \qquad v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_t \qquad v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_f}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ if(t_c, t_t, t_f)}$$

Asignaciones Locales

$$\frac{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_v \qquad y : x_{n+1},\ v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_b}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ let(t_v, y.t_b)}$$

Funciones

$$\frac{f : f_{n+1} \ldots v_0 : x_0 \ \sim_\alpha \ t}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ fun(T, f.t)} \qquad \frac{f : f_{n+1}, y : x_{n+1} \ldots v_0 : x_0 \ \sim_\alpha \ t}{x : x_n \ldots v_0 : x_0 \ \sim_\alpha \ recfun(T \to S, f.y.t)}$$

Aplicación de función

$$\frac{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_f \qquad v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ t_p}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ app(t_f, t_p)}$$

Operador de punto fijo

$$\frac{y : x_{n+1} \ldots v_0 : x_0 \ \sim_\alpha \ t}{v_n : x_n \ldots v_0 : x_0 \ \sim_\alpha \ fix(T, y.t)}$$

[a]definición formulada de [5] y [12]

2 Generación de restricciones

Para poder tipificar una expresión bien formada de **MinHS** sin utilizar las anotaciones de tipo explícitamente en los argumentos y valores de retorno es necesario definir un proceso que nos ayude a obtener información sobre la estructura que esta posee.

A esta información se le conoce como restricciones y nos ayudan a ligar las condiciones que se deben cumplir para que una expresión esté bien tipificada.

Definición 2.1 (Conjunto de restricciones)**.** Definimos el juicio para generar restricciones partiendo de una expresión válida e representado como[a]:

$$e \mapsto R$$

Que se lee como: ”La expresión e genera el conjunto R de restricciones de tipo”.

[a] Juicio para representar restricciones [123], [124] y [125]

Definición 2.2 (Extensión de tipos para la generación de restricciones)**.** Para definir el algoritmo se extiende la categoría de tipos como sigue[a]:

$$T \ ::= \ X \mid Nat \mid Bool \mid T_1 \ \to T_2 \mid [\![e]\!]$$

En donde X es una variable de tipo. Estas variables nos ayudan en la definición de programas polimorfos, por ejemplo la función identidad:

$$fun\ x \Rightarrow x :: X \to X$$

La variable de tipo X va a tomar su valor hasta que la función sea evaluada, por ejemplo en la aplicación:

$$(fun\ x \Rightarrow x)\ 5 :: Nat \to Nat$$

La expresión $[\![e]\!]$ es una construcción sintáctica para definir el tipo de una expresión del lenguaje y se lee como: ”el tipo de e”.

Es importante aclarar que esta construcción no puede figurar como el tipo final asociado a la expresión y la encontraremos usualmente aplicada de la siguiente forma:

$$[\![e]\!] = E$$

[a] Definición formulada de [5], [12], [125] y [124]

Definición 2.3 (Algoritmo de generación de restricciones)**.** Una restricción es una ecuación de la forma $T_1 = T_2$ en donde T_1 y T_2 son tipos. La ecuación indica que T_1 debe ser igual a T_2 bajo unificación[a].

Variables

$$\frac{}{x_i \mapsto [\![x_i]\!] = X_i}$$

Para el tipo de las variables se usará la misma variable pero en mayúsculas. Todas las apariciones de la variable generarán la misma restricción y como los nombres de variables son únicos no habrá dos variables distintas con el mismo tipo.

Valores numéricos

$$\frac{}{num[n] \mapsto [\![num[n]]\!] = Nat}$$

Valores Booleanos

$$\frac{}{bool[b] \mapsto [\![bool[b]]\!] = Bool}$$

Condicional

$$\frac{c \mapsto R_1 \qquad t \mapsto R_2 \qquad e \mapsto R_3}{if(c,t,e) \mapsto R_1 : R_2 : R_3 : [\![c]\!] = Bool : [\![t]\!] = [\![e]\!] : [\![if(c,t,e)]\!] = [\![e]\!] : [\![if(c,t,e)]\!] = [\![t]\!]}$$

Asignaciones Locales

$$\frac{v \mapsto R_1 \qquad b \mapsto R_2}{let(v, x_i.b) \mapsto R_1 : R_2 : X_i = [\![v]\!] : [\![let(v, x_i.b)]\!] = [\![b]\!]}$$

$$\frac{v \mapsto R_1 \qquad b \mapsto R_2}{recfun(v, f.x_i.b) \mapsto R_1 : R_2 : X_i = [\![v]\!] : [\![recfun(v, f.x_i.b)]\!] = [\![b]\!]}$$

Funciones

$$\frac{t \mapsto R}{fun(x_i.t) \mapsto R : [\![fun(x_i.t)]\!] = X_i \to [\![t]\!]}$$

Aplicación de función

$$\frac{f \mapsto R_1 \qquad p \mapsto R_2}{app(f, p) \mapsto R_1 : R_2 : [\![f]\!] = [\![p]\!] \to [\![app(f, p)]\!]}$$

Operadores

$$\frac{e_1 \mapsto R_1 \qquad e_2 \mapsto R_2}{sum(e_1, e_2) \mapsto R_1 : R_2 : [\![e_1]\!] = Nat : [\![e_2]\!] = Nat : [\![sum(e_1, e_2)]\!] = Nat}$$

$$\frac{e_1 \mapsto R_1 \qquad e_2 \mapsto R_2}{prod(e_1, e_2) \mapsto R_1 : R_2 : [\![e_1]\!] = Nat : [\![e_2]\!] = Nat : [\![prod(e_1, e_2)]\!] = Nat}$$

$$\frac{e_1 \mapsto R_1 \qquad e_2 \mapsto R_2}{sub(e_1, e_2) \mapsto R_1 : R_2 : [\![e_1]\!] = Nat : [\![e_2]\!] = Nat : [\![sub(e_1, e_2)]\!] = Nat}$$

$$\frac{e_1 \mapsto R_1 \qquad e_2 \mapsto R_2}{eq(e_1, e_2) \mapsto R_1 : R_2 : [\![e_1]\!] = Nat : [\![e_2]\!] = Nat : [\![ig(e_1, e_2)]\!] = Bool}$$

$$\frac{e_1 \mapsto R_1 \qquad e_2 \mapsto R_2}{gt(e_1, e_2) \mapsto R_1 : R_2 : [\![e_1]\!] = Nat : [\![e_2]\!] = Nat : [\![gt(e_1, e_2)]\!] = Bool}$$

$$\frac{e_1 \mapsto R_1 \qquad e_2 \mapsto R_2}{lt(e_1, e_2) \mapsto R_1 : R_2 : [\![e_1]\!] = Nat : [\![e_2]\!] = Nat : [\![lt(e_1, e_2)]\!] = Bool}$$

[a] Definición formulada de [5], [12], [125] y [124]

3 Algoritmo de unificación

Una vez obtenida la lista de restricciones asociadas a una expresión e tenemos toda la información que necesitamos acerca de su estructura para comenzar a unificar los tipos aplicando las restricciones hasta encontrar el tipo más general de la expresión, o hasta encontrar un error de tipificado al asignar dos tipos distintos a un mismo elemento.

Para esto construiremos una composición de sustituciones tomando cada una de las restricciones y sustituyendo el tipo al cuál la expresión está ligada en dicha restricción. Al final se obtendrá la lista de sustituciones necesarias para hallar el tipo más general el cuál será la cabeza de la lista, en caso contrario quiere decir que la unificación falló.

Definición 3.1 (Algoritmo de unificación)**.** La entrada del algoritmo es una lista de restricciones y la salida es una composición de sustituciones en caso de que las restricciones se puedan resolver o Fail en caso contrario.

A esta composición de sustituciones la denotamos con la letra μ y se le conoce como: "unificador"[a].

$$
\begin{array}{rcll}
U([\,]) & = & [\,] & \\
U(T = T : R) & = & U(R) & \\
U(X = T : R) & = & U(R[X := T]) \circ [X := T] & \mathit{si}\ X \notin \mathit{var}(T) \\
U(X = T : R) & = & \mathsf{Fail} & \text{si } X \in \mathit{var}(T) \\
U(e = T : R) & = & U(R[e := T]) \circ [e := T] & \\
U(T = X : R) & = & U(X = T : R) & \\
U(T = e : R) & = & U(e = T : R) & \\
U(St_1 \to St_2 = T_1 \to T_2 : R) & = & U(St_1 = T_1 : St_2 = T_2 : R) & \\
U(R) & = & \mathsf{Fail} &
\end{array}
$$

Este algoritmo compone el unificador general convirtiendo la restricción de la cabeza de la lista en una sustitución (cuando sea posible), aplicándola al resto de las restricciones y concatenándola para generar el unificador general.

[a] Definición formulada de [5], [12], [123], [124] y [125]

4 Algoritmo de inferencia de tipos

Definición 4.1 (Algoritmo de Inferencia de tipos)**.** Se define el algoritmo $\mathcal{T}(e)$ de inferencia de tipos que recibe una expresión e de MinHS y regresa el tipo de esta expresión como sigue:[a]:

- Se encuentra la expresión e' con nombres de variables únicas.
- Se encuentra el conjunto de restricciones R tal que $e' \mapsto R$.
- Utilizando la función U se calcula el unificador general μ del conjunto de restricciones R, tal que $U(R) = \mu$.
- Se busca en μ la ecuación $e' := T$.
- T es el tipo general de e, es decir, $\mathcal{T}(e) = T$.

[a] Definición formulada de [5], [12], [123], [124] y [125]

Ejercicio 4.1. Vamos a encontrar el tipo de la expresión:

```
let x = 0 in
    let y = 1 in
        x = y
    end
end
```

Representación en sintaxis abstracta:

$$let(0,\ x.let(1,\ y.eq(x,\ y)))$$

Renombramiento de variables

$$let(0, x_0.let(1, x_1.eq(x_0, x_1)))$$

Generación de restricciones:

- $0 \mapsto [\![0]\!] = Nat$
- $1 \mapsto [\![1]\!] = Nat$
- $x_0 \mapsto [\![x_0]\!] = X_0$
- $x_1 \mapsto [\![x_1]\!] = X_1$
- $eq(x_0,\ x_1) \mapsto \underbrace{[\![x_0]\!] = X_0, [\![x_1]\!] = X_1, [\![x_0]\!] = Nat, [\![x_1]\!] = Nat, [\![eq(x_0, x_1)]\!] = Bool}_{R_1}$
- $let(1,\ x_1.eq(x_0,\ x_1)) \mapsto \underbrace{[\![1]\!] = Nat, R_1, X_1 = [\![1]\!], [\![let(1, x_1.eq(x_0, x_1))]\!] = [\![eq(x_0, x_1)]\!]}_{R_2}$
- $let(0,\ x_0.let(1,\ x_1.eq(x_0,\ x_1))) \mapsto$
  $\underbrace{[\![0]\!] = Nat, R_2, X_0 = [\![0]\!], [\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = [\![let(1, x_1.eq(x_0, x_1))]\!]}_{R_3}$

Lista de restricciones de R:

$$\begin{aligned}
\mathbb{R} \quad = \quad & [\![0]\!] = Nat, \\
& [\![1]\!] = Nat \\
& [\![x_0]\!] = \mathbb{X}_0 \\
& [\![x_1]\!] = \mathbb{X}_1 \\
& [\![x_0]\!] = Nat \\
& [\![eq(x_0, x_1)]\!] = Bool \\
& X_1 = [\![1]\!] \\
& [\![let(1, x_1.eq(x_0, x_1))]\!] = [\![eq(x_0, x_1)]\!] \\
& [\![x_1]\!] = X_1 \\
& X_0 = [\![0]\!] \\
& [\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = [\![let(1, x_1.eq(x_0, x_1))]\!]
\end{aligned}$$

Aplicación del algoritmo para encontrar el unificador general de R:

Restricciones	Unificador μ
$[\![0]\!] = Nat$ $[\![1]\!] = Nat$ $[\![x_0]\!] = X_0$ $[\![x_1]\!] = X_1$ $[\![x_0]\!] = Nat$ $[\![eq(x_0, x_1)]\!] = Bool$ $X_1 = [\![1]\!]$ $[\![let(1, x_1.eq(x_0, x_1))]\!] = [\![eq(x_0, x_1)]\!]$	

$\llbracket x_1 \rrbracket = X_1$ $X_0 = \llbracket 0 \rrbracket$ $\llbracket let(0, x_0.let(1, x_1.eq(x_0, x_1))) \rrbracket = \llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket$	
$\llbracket 1 \rrbracket = Nat$ $\llbracket x_0 \rrbracket = X_0$ $\llbracket x_1 \rrbracket = X_1$ $\llbracket x_0 \rrbracket = Nat$ $\llbracket eq(x_0, x_1) \rrbracket = Bool$ $X_1 = \llbracket 1 \rrbracket$ $\llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket = \llbracket eq(x_0, x_1) \rrbracket$ $\llbracket x_1 \rrbracket = X_1$ $X_0 = Nat$ $\llbracket let(0, x_0.let(1, x_1.eq(x_0, x_1))) \rrbracket = \llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket$	$\llbracket 0 \rrbracket := Nat$
$\llbracket x_0 \rrbracket = X_0$ $\llbracket x_1 \rrbracket = X_1$ $\llbracket x_0 \rrbracket = Nat$ $\llbracket eq(x_0, x_1) \rrbracket = Bool$ $X_1 = Nat$ $\llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket = \llbracket eq(x_0, x_1) \rrbracket$ $\llbracket x_1 \rrbracket = X_1$ $X_0 = Nat$ $\llbracket let(0, x_0.let(1, x_1.eq(x_0, x_1))) \rrbracket = \llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket$	$\llbracket 0 \rrbracket := Nat$ $\llbracket 1 \rrbracket := Nat$
$\llbracket x_1 \rrbracket = X_1$ $X_0 = Nat$ $\llbracket eq(x_0, x_1) \rrbracket = Bool$ $X_1 = Nat$ $\llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket = \llbracket eq(x_0, x_1) \rrbracket$ $\llbracket x_1 \rrbracket = X_1$ $X_0 = Nat$ $\llbracket let(0, x_0.let(1, x_1.eq(x_0, x_1))) \rrbracket = \llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket$	$\llbracket 0 \rrbracket := Nat$ $\llbracket 1 \rrbracket := Nat$ $\llbracket x_0 \rrbracket := X_0$
$X_0 = Nat$ $\llbracket eq(x_0, x_1) \rrbracket = Bool$ $X_1 = Nat$ $\llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket = \llbracket eq(x_0, x_1) \rrbracket$ $X_1 = X_1$ $X_0 = Nat$ $\llbracket let(0, x_0.let(1, x_1.eq(x_0, x_1))) \rrbracket = \llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket$	$\llbracket 0 \rrbracket := Nat$ $\llbracket 1 \rrbracket := Nat$ $\llbracket x_0 \rrbracket := X_0$ $\llbracket x_1 \rrbracket := X_1$
$\llbracket eq(x_0, x_1) \rrbracket = Bool$ $X_1 = Nat$ $\llbracket let(1, x_1.eq(x_0, x_1)) \rrbracket = \llbracket eq(x_0, x_1) \rrbracket$ $X_1 = X_1$	$\llbracket 0 \rrbracket := Nat$ $\llbracket 1 \rrbracket := Nat$ $\llbracket x_0 \rrbracket := X_0$ $\llbracket x_1 \rrbracket := X_1$

$Nat = Nat$ $[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = [\![let(1, x_1.eq(x_0, x_1))]\!]$	$X_0 := Nat$
$X_1 = Nat$ $[\![let(1, x_1.eq(x_0, x_1))]\!] = Bool$ $X_1 = X_1$ $Nat = Nat$ $[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = [\![let(1, x_1.eq(x_0, x_1))]\!]$	$[\![0]\!] := Nat$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $[\![x_1]\!] := X_1$ $[\![x_0]\!] := Nat$ $[\![eq(x_0, x_1)]\!] = Bool$
$[\![let(1, x_1.eq(x_0, x_1))]\!] = Bool$ $Nat = Nat$ $Nat = Nat$ $[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = [\![let(1, x_1.eq(x_0, x_1))]\!]$	$[\![0]\!] := Nat$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $[\![x_1]\!] := X_1$ $[\![x_0]\!] := Nat$ $[\![eq(x_0, x_1)]\!] := Bool$ $X_1 := Nat$
$Nat = Nat$ $Nat = Nat$ $[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = Bool$	$[\![0]\!] := Nat$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $[\![x_1]\!] := X_1$ $[\![x_0]\!] := Nat$ $[\![eq(x_0, x_1)]\!] := Bool$ $X_1 := Nat$ $[\![let(1, x_1.eq(x_0, x_1))]\!] := Bool$
$Nat = Nat$ $[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = Bool$	$[\![0]\!] := Nat$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $[\![x_1]\!] := X_1$ $[\![x_0]\!] := Nat$ $[\![eq(x_0, x_1)]\!] := Bool$ $X_1 := Nat$ $[\![let(1, x_1.eq(x_0, x_1))]\!] := Bool$
$[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] = Bool$	$[\![0]\!] := Nat$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $[\![x_1]\!] := X_1$ $[\![eq(x_0, x_1)]\!] := Bool$ $X_1 := Nat$ $[\![let(1, x_1.eq(x_0, x_1))]\!] := Bool$
	$[\![0]\!] := Nat$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$

	$[\![x_1]\!] := X_1$ $[\![eq(x_0, x_1)]\!] := Bool$ $X_1 := Nat$ $[\![let(1, x_1.eq(x_0, x_1))]\!] := Bool$ $[\![let(0, x_0.let(1, x_1.eq(x_0, x_1)))]\!] := Bool$

Del proceso anterior se puede concluir que el tipo de la expresión es *Bool*

Ejercicio 4.2. Vamos a encontrar el tipo de la expresión:

```
recfun fib n ->
    if (n < 2)
        then 1
    else fib (n-1) + fib (n-2)
```

Representación en sintáxis abstracta

$$app(recfun(fib.n.if(lt(n,2),1,sum(app(fib,(sub(n,1))),app(fib,(sub(n,2)))))),4)$$

Renombramiento de variables

$$app(recfun(x_0.x_1.if(lt(x_1,2),1,sum(app(x_0,(sub(x_1,1))),app(x_0,(sub(x_1,2)))))),4)$$

Generación de restricciones

- $1 \mapsto [\![1]\!] = Nat$
- $2 \mapsto [\![2]\!] = Nat$
- $x_0 \to [\![x_0]\!] = X_0$
- $x_1 \mapsto [\![x_1]\!] = X_1$
- $sub(x_1, 1) \mapsto \underbrace{[\![x_1]\!] = X_1, [\![1]\!] = Nat, [\![x_1]\!] = Nat, [\![sub(x_1, 1)]\!] = Nat}_{R_1}$
- $sub(x_1, 2) \mapsto \underbrace{[\![x_1]\!] = X_1, [\![2]\!] = Nat, [\![x_1]\!] = Nat, [\![sub(x_1, 2)]\!] = Nat}_{R_2}$
- $app(x_0, sub(x_1, 1)) \mapsto \underbrace{[\![x_0]\!] = X_0, R_1, [\![x_0]\!] = [\![sub(x_1, 1)]\!] \mapsto [\![app(x_0, sub(x_1, 1))]\!]}_{R_3}$
- $app(x_0, sub(x_1, 2)) \mapsto \underbrace{[\![x_0]\!] = X_0, R_2, [\![x_0]\!] = [\![sub(x_1, 2)]\!] \mapsto [\![app(x_0, sub(x_1, 2))]\!]}_{R_4}$
- $sum(app(x_0, sub(x_1, 1)), app(x_0, sub(x_1, 2))) \mapsto$
  $\underbrace{R_3, R_4.[\![app(x_0, sub(x_1, 1))]\!] = Nat, [\![app(x_0, sub(x_1, 2))]\!] = Nat,}_{R_5}$
  $\underbrace{[\![sum(app(x_0, sub(x_1, 1))]\!] = Nat}_{R_5}$

- $lt(x_1, 2) \mapsto \underbrace{[\![x_1]\!] = X_1, [\![2]\!] = Nat, [\![x_1]\!] = Nat, [\![lt(x_1, 2)]\!] = Bool}_{R_6}$
- $if(lt(x_1, 2), 1, sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))) \mapsto$
  $\underbrace{R_6, [\![1]\!] = Nat, R_5, [\![1]\!] = [\![sum(app(x_0, (sub(x_1, 1))),}_{R_7}$
  $\underbrace{app(x_0, (sub(x_1, 2))))]\!], [\![if(...)]\!] = [\![1]\!], [\![if(...)]\!] = [\![sum(...)]\!]}_{R_7}$
- $recfun(x_0,\ x_1.if(lt(x_1, 2), 1, sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2)))))) \mapsto$

  $\underbrace{R_7, X_1 = [\![x_0]\!], [\![recfun(...)]\!] = [\![if(...)]\!]}_{R_8}$

Como resultado se obtiene la lista de restricciones R:

$$
\begin{aligned}
R \quad = \quad & [\![x_1]\!] = X_1 \\
& [\![2]\!] = Nat \\
& [\![lt(x_1, 2)]\!] = Bool \\
& [\![1]\!] = Nat \\
& [\![x_0]\!] = X_0 \\
& [\![x_1]\!] = X_1 \\
& [\![x_1]\!] = Nat \\
& [\![sub(x_1, 1)]\!] = Nat \\
& [\![x_0]\!] = [\![sub(x_1, 1)]\!] \mapsto [\![app(x_0, sub(x_1, 1))]\!] \\
& [\![2]\!] = Nat \\
& [\![sub(x_1, 2)]\!] = Nat \\
& [\![x_0]\!] = [\![sub(x_1, 2)]\!] \mapsto [\![app(x_0, sub(x_1, 2))]\!] \\
& [\![app(x_0, sub(x_1, 1))]\!] = Nat \\
& [\![app(x_0, sub(x_1, 2))]\!] = Nat \\
& [\![sum(app(x_0, sub(x_1, 1))]\!] = Nat \\
& [\![1]\!] = [\![sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]\!] \\
& [\![if(...)]\!] = [\![1]\!] \\
& [\![if(...)]\!] = [\![sum(...)]\!] \\
& [\![recfun(...)]\!] = [\![if(...)]\!]
\end{aligned}
$$

Aplicación del algoritmo para encontrar el unificador general de R:

Restricciones	Unificador μ
$[\![x_1]\!] = X_1$ $[\![2]\!] = Nat$ $[\![lt(x_1, 2)]\!] = Bool$ $[\![1]\!] = Nat$ $[\![x_0]\!] = X_0$ $[\![x_1]\!] = X_1$ $[\![x_1]\!] = Nat$ $[\![sub(x_1, 1)]\!] = Nat$	

$\llbracket x_0 \rrbracket = \llbracket sub(x_1,1) \rrbracket \mapsto \llbracket app(x_0, sub(x_1,1)) \rrbracket$ $\llbracket 2 \rrbracket = Nat$ $\llbracket sub(x_1,2) \rrbracket = Nat$ $\llbracket x_0 \rrbracket = \llbracket sub(x_1,2) \rrbracket \mapsto \llbracket app(x_0, sub(x_1,2)) \rrbracket$ $\llbracket app(x_0, sub(x_1,1)) \rrbracket = Nat$ $\llbracket app(x_0, sub(x_1,2)) \rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1,1)) \rrbracket = Nat$ $\llbracket 1 \rrbracket = \llbracket sum(app(x_0,(sub(x_1,1))), app(x_0,(sub(x_1,2)))) \rrbracket$ $\llbracket if(...) \rrbracket = \llbracket 1 \rrbracket$ $\llbracket if(...) \rrbracket = \llbracket sum(...) \rrbracket$ $\llbracket recfun(...) \rrbracket = \llbracket if(...) \rrbracket$	
$\llbracket 2 \rrbracket = Nat$ $\llbracket lt(x_1,2) \rrbracket = Bool$ $\llbracket 1 \rrbracket = Nat$ $\llbracket x_0 \rrbracket = X_0$ $X_1 = X_1$ $X_1 = Nat$ $\llbracket sub(x_1,1) \rrbracket = Nat$ $\llbracket x_0 \rrbracket = \llbracket sub(x_1,1) \rrbracket \mapsto \llbracket app(x_0, sub(x_1,1)) \rrbracket$ $\llbracket 2 \rrbracket = Nat$ $\llbracket sub(x_1,2) \rrbracket = Nat$ $\llbracket x_0 \rrbracket = \llbracket sub(x_1,2) \rrbracket \mapsto \llbracket app(x_0, sub(x_1,2)) \rrbracket$ $\llbracket app(x_0, sub(x_1,1)) \rrbracket = Nat$ $\llbracket app(x_0, sub(x_1,2)) \rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1,1)) \rrbracket = Nat$ $\llbracket 1 \rrbracket = \llbracket sum(app(x_0,(sub(x_1,1))), app(x_0,(sub(x_1,2)))) \rrbracket$ $\llbracket if(...) \rrbracket = \llbracket 1 \rrbracket$ $\llbracket if(...) \rrbracket = \llbracket sum(...) \rrbracket$ $\llbracket recfun(...) \rrbracket = \llbracket if(...) \rrbracket$	$\llbracket x_1 \rrbracket := X_1$
$\llbracket lt(x_1,2) \rrbracket = Bool$ $\llbracket 1 \rrbracket = Nat$ $\llbracket x_0 \rrbracket = X_0$ $X_1 = \mathbb{X}_1$ $X_1 = Nat$ $\llbracket sub(x_1,1) \rrbracket = Nat$ $\llbracket x_0 \rrbracket = \llbracket sub(x_1,1) \rrbracket \mapsto \llbracket app(x_0, sub(x_1,1)) \rrbracket$ $Nat = Nat$ $\llbracket sub(x_1,2) \rrbracket = Nat$ $\llbracket x_0 \rrbracket = \llbracket sub(x_1,2) \rrbracket \mapsto \llbracket app(x_0, sub(x_1,2)) \rrbracket$ $\llbracket app(x_0, sub(x_1,1)) \rrbracket = Nat$ $\llbracket app(x_0, sub(x_1,2)) \rrbracket = Nat$	$\llbracket x_1 \rrbracket := X_1$ $\llbracket 2 \rrbracket := Nat$

$\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket 1\rrbracket = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = \llbracket 1\rrbracket$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	
$\llbracket 1\rrbracket = Nat$ $\llbracket x_0\rrbracket = X_0$ $X_1 = X_1$ $X_1 = Nat$ $\llbracket sub(x_1, 1)\rrbracket = Nat$ $\llbracket x_0\rrbracket = \llbracket sub(x_1, 1)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $Nat = Nat$ $\llbracket sub(x_1, 2)\rrbracket = Nat$ $\llbracket x_0\rrbracket = \llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket 1\rrbracket = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = \llbracket 1\rrbracket$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$
$\llbracket x_0\rrbracket = X_0$ $X_1 = X_1$ $X_1 = Nat$ $\llbracket sub(x_1, 1)\rrbracket = Nat$ $\llbracket x_0\rrbracket = \llbracket sub(x_1, 1)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $Nat = Nat$ $\llbracket sub(x_1, 2)\rrbracket = Nat$ $\llbracket x_0\rrbracket = \llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket = Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$
$X_1 = X_1$ $X_1 = Nat$ $\llbracket sub(x_1, 1)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 1)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $Nat = Nat$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$

$\llbracket sub(x_1, 2)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket = Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	
$X_1 = Nat$ $\llbracket sub(x_1, 1)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 1)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $Nat = Nat$ $\llbracket sub(x_1, 2)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket = Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$
$\llbracket sub(x_1, 1)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 1)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $Nat = Nat$ $\llbracket sub(x_1, 2)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket = Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$ $X_1 := Nat$
$X_0 = Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $Nat = Nat$ $\llbracket sub(x_1, 2)\rrbracket = Nat$ $X_0 = \llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket := Nat$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$ $X_1 := Nat$ $\llbracket sub(x_1, 1)\rrbracket := Nat$

$Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	
$Nat = Nat$ $\llbracket sub(x_1, 2)\rrbracket = Nat$ $Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket =$ $\llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket := Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $\llbracket sub(x_1, 1)\rrbracket := Nat$
$\llbracket sub(x_1, 2)\rrbracket = Nat$ $Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket =$ $\llbracket sub(x_1, 2)\rrbracket \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket := Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $\llbracket sub(x_1, 1)\rrbracket := Nat$
$Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat \mapsto \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket := Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$ $\llbracket if(...)\rrbracket = \llbracket sum(...)\rrbracket$ $\llbracket recfun(...)\rrbracket = \llbracket if(...)\rrbracket$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$ $\llbracket sub(x_1, 1)\rrbracket := Nat$ $\llbracket sub(x_1, 2)\rrbracket := Nat$
$Nat = Nat$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = \llbracket app(x_0, sub(x_1, 2))\rrbracket$ $\llbracket app(x_0, sub(x_1, 1))\rrbracket = Nat$ $\llbracket app(x_0, sub(x_1, 2))\rrbracket = Nat$ $\llbracket sum(app(x_0, sub(x_1, 1))\rrbracket := Nat$ $Nat = \llbracket sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))\rrbracket$ $\llbracket if(...)\rrbracket = Nat$	$\llbracket x_1\rrbracket := X_1$ $\llbracket 2\rrbracket := Nat$ $\llbracket lt(x_1, 2)\rrbracket := Bool$ $\llbracket 1\rrbracket := Nat$ $\llbracket x_0\rrbracket := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto \llbracket app(x_0, sub(x_1, 1))\rrbracket$

$[\![if(...)]\!] = [\![sum(...)]\!]$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![sub(x_1, 1)]\!] := Nat$ $[\![sub(x_1, 2)]\!] := Nat$
$[\![app(x_0, sub(x_1, 1))]\!] = [\![app(x_0, sub(x_1, 2))]\!]$ $[\![app(x_0, sub(x_1, 1))]\!] = Nat$ $[\![app(x_0, sub(x_1, 2))]\!] = Nat$ $[\![sum(app(x_0, sub(x_1, 1))]\!] := Nat$ $Nat = [\![sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]\!]$ $[\![if(...)]\!] = Nat$ $[\![if(...)]\!] = [\![sum(...)]\!]$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1, 2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0, sub(x_1, 1))]\!]$ $[\![sub(x_1, 1)]\!] := Nat$ $[\![sub(x_1, 2)]\!] := Nat$
$[\![app(x_0, sub(x_1, 2))]\!] = Nat$ $[\![app(x_0, sub(x_1, 2))]\!] = Nat$ $[\![sum(app(x_0, sub(x_1, 1))]\!] := Nat$ $Nat = [\![sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]\!]$ $[\![if(...)]\!] = Nat$ $[\![if(...)]\!] = [\![sum(...)]\!]$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1, 2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0, sub(x_1, 1))]\!]$ $[\![sub(x_1, 1)]\!] := Nat$ $[\![sub(x_1, 2)]\!] := Nat$ $[\![app(x_0, sub(x_1, 1))]\!] :=$ $[\![app(x_0, sub(x_1, 2))]\!]$
$[\![app(x_0, sub(x_1, 2))]\!] = Nat$ $[\![sum(app(x_0, sub(x_1, 1))]\!] := Nat$ $Nat = [\![sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]\!]$ $[\![if(...)]\!] = Nat$ $[\![if(...)]\!] = [\![sum(...)]\!]$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1, 2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0, sub(x_1, 1))]\!]$ $[\![sub(x_1, 1)]\!] := Nat$ $[\![sub(x_1, 2)]\!] := Nat$ $[\![app(x_0, sub(x_1, 1))]\!] :=$ $[\![app(x_0, sub(x_1, 2))]\!]$
$[\![sum(app(x_0, sub(x_1, 1))]\!] = Nat$ $Nat = [\![sum(app(x_0, (sub(x_1, 1))), app(x_0, (sub(x_1, 2))))]\!]$ $[\![if(...)]\!] = Nat$ $[\![if(...)]\!] = [\![sum(...)]\!]$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1, 2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0, sub(x_1, 1))]\!]$ $[\![sub(x_1, 1)]\!] := Nat$

	$[\![sub(x_1,2)]\!] := Nat$ $[\![app(x_0,sub(x_1,1))]\!] :=$ $[\![app(x_0,sub(x_1,2))]\!]$ $[\![app(x_0,sub(x_1,2))]\!] := Nat$
$Nat = [\![sum(app(x_0,(sub(x_1,1))),app(x_0,(sub(x_1,2))))]\!]$ $[\![if(...)]\!] = Nat$ $[\![if(...)]\!] = [\![sum(...)]\!]$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1,2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0,sub(x_1,1))]\!]$ $[\![sub(x_1,1)]\!] := Nat$ $[\![sub(x_1,2)]\!] := Nat$ $[\![app(x_0,sub(x_1,1))]\!] :=$ $[\![app(x_0,sub(x_1,2))]\!]$ $[\![app(x_0,sub(x_1,2))]\!] := Nat$ $[\![sum(app(x_0,sub(x_1,1))]\!] := Nat$
$[\![if(...)]\!] = Nat$ $[\![if(...)]\!] = Nat$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1,2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0,sub(x_1,1))]\!]$ $[\![sub(x_1,1)]\!] := Nat$ $[\![sub(x_1,2)]\!] := Nat$ $[\![app(x_0,sub(x_1,1))]\!] :=$ $[\![app(x_0,sub(x_1,2))]\!]$ $[\![app(x_0,sub(x_1,2))]\!] := Nat$ $[\![sum(app(x_0,sub(x_1,1))]\!] := Nat$ $[\![sum(...)]\!] := Nat$
$[\![if(...)]\!] = Nat$ $[\![recfun(...)]\!] = [\![if(...)]\!]$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1,2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0,sub(x_1,1))]\!]$ $[\![sub(x_1,1)]\!] := Nat$ $[\![sub(x_1,2)]\!] := Nat$ $[\![app(x_0,sub(x_1,1))]\!] :=$

	$[\![app(x_0, sub(x_1, 2))]\!]$ $[\![app(x_0, sub(x_1, 2))]\!] := Nat$ $[\![sum(app(x_0, sub(x_1, 1))]\!] := Nat$ $[\![sum(...)]\!] := Nat$
$[\![recfun(...)]\!] = Nat$	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1, 2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0, sub(x_1, 1))]\!]$ $[\![sub(x_1, 1)]\!] := Nat$ $[\![sub(x_1, 2)]\!] := Nat$ $[\![app(x_0, sub(x_1, 1))]\!] :=$ $[\![app(x_0, sub(x_1, 2))]\!]$ $[\![app(x_0, sub(x_1, 2))]\!] := Nat$ $[\![sum(app(x_0, sub(x_1, 1))]\!] := Nat$ $[\![sum(...)]\!] := Nat$ $[\![if(...)]\!] := Nat$
	$[\![x_1]\!] := X_1$ $[\![2]\!] := Nat$ $[\![lt(x_1, 2)]\!] := Bool$ $[\![1]\!] := Nat$ $[\![x_0]\!] := X_0$ $X_1 := Nat$ $X_0 := Nat \mapsto [\![app(x_0, sub(x_1, 1))]\!]$ $[\![sub(x_1, 1)]\!] := Nat$ $[\![sub(x_1, 2)]\!] := Nat$ $[\![app(x_0, sub(x_1, 1))]\!] :=$ $[\![app(x_0, sub(x_1, 2))]\!]$ $[\![app(x_0, sub(x_1, 2))]\!] := Nat$ $[\![sum(app(x_0, sub(x_1, 1))]\!] := Nat$ $[\![sum(...)]\!] := Nat$ $[\![if(...)]\!] := Nat$ $[\![recfun(...)]\!] := Nat$

Del proceso anterior se puede concluir que el tipo de la expresión es Nat

5 Ejercicios para el lector

Ejercicio 5.1. Para la siguiente expresión de MinHS contesta los incisos enumerados a continuación:

$$\mathsf{let}\ z = 0\ \mathsf{in}\ z < z\ \mathsf{end}$$

- obtén la representación en sintaxis abstracta.
- obtén una expresión α-equivalente
- obtén la lista de variables con su tipo.

Ejercicio 5.2. Para la siguiente expresión de MinHS contesta los incisos enumerados a continuación:

$$\mathsf{fun}\ x :: Nat\ \rightarrow x\ \text{*}\ (x\ \text{-}\ 1)\ /\ (x\ \text{*}\ x)$$

- obtén la representación en sintaxis abstracta.
- obtén una expresión α-equivalente
- obtén la lista de variables con su tipo.

Ejercicio 5.3. Para la siguiente expresión de MinHS contesta los incisos enumerados a continuación:

$$\mathsf{let}\ x = \mathsf{True}\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{False}\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{let}\ y = 1\ \mathsf{in}\ y\ \mathsf{end}\ \mathsf{else}\ \mathsf{let}\ y = 2\ \mathsf{in}\ y\ \mathsf{end}\ \mathsf{end}\ \mathsf{end}$$

- obtén la representación en sintaxis abstracta.
- obtén una expresión α-equivalente
- obtén la lista de variables con su tipo.

Ejercicio 5.4. Dada la siguiente expresión de MinHS, obtén el tipo más general del unificador μ utilizando el algoritmo de inferencia[a]

$$\mathsf{fun}\ x :: Bool \rightarrow (\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\ \mathsf{True}$$

[a] Ejercicio extraído de [12]

Ejercicio 5.5. Dada la siguiente expresión de MinHS, obtén el tipo más general del unificador μ utilizando el algoritmo de inferencia[a]

$$\mathsf{fun}\ x\ ::\ Nat\ \rightarrow\ \mathsf{app}(x, x)$$

[a] Ejercicio extraído de [12]

Ejercicio 5.6. Dada la siguiente expresión de MinHS, obtén el tipo más general del unificador μ utilizando el algoritmo de inferencia.

$$\mathsf{let}\ x = 0\ \mathsf{in}\ \mathsf{fun}\ y :: Nat \to y + x\ \mathsf{end}$$

Ejercicio 5.7. Dada la siguiente expresión de MinHS, obtén el tipo más general del unificador μ utilizando el algoritmo de inferencia.

```
(recfun factorial n →
    if (n < 0)
        then 1
    else factorial (n-1)*n)9
```

Ejercicio 5.8. Contesta las siguientes preguntas.

- ¿Cuál es la complejidad de cada fase del algoritmo para encontrar el unificador general μ?
- ¿Cuál es la complejidad total del algoritmo para encontrar el unificador general μ?

Capítulo 8

Máquinas abstractas

Una máquina abstracta define una colección de estados y reglas de transición que nos permiten obtener información acerca de la ejecución de un programa por cada cómputo que este realiza. Este tipo de mecanismos de evaluación ya han sido revisados en cursos previos, específicamente en autómatas y lenguajes formales[1], donde se estudian sistemas de transición como autómatas y las máquinas de Turing.

En este capítulo introduciremos un sistema de cómputo mezclando las ideas de los sistemas de transición para definir una máquina de ejecución para MinHS, conocidas como las máquinas $\mathcal{H}$ y $\mathcal{J}$.

[1]Conforme al plan de estudios que se imparte desde el 2013 en la Facultad de Ciencias de la Universidad Nacional Autónoma de México con clave de asignatura 1425.

Objetivo

En este capítulo exploraremos la definición de las máquinas $\mathcal{H}$ y $\mathcal{J}$ implementando la colección de estados, cómputos y los axiomas de transición que nos permitan visualizar el proceso de evaluación de un expresión cerrada de MinHS, ejemplos de programas válidos serán revisados para ilustrar los cómputos.

Planteamiento

El capítulo está estructurado para definir los estados, marcos y reglas de transición de las máquinas $\mathcal{H}$ y $\mathcal{J}$ seguido de la ejecución detallada paso a paso de programas correctamente formados de MinHS para ilustrar la ejecución de ambas máquinas.

De forma similar, la pila de ejecución será introducida para controlar aquellos cómputos que se quedan pendientes de evaluación, hasta obtener un valor que pueda ser regresado como argumento a la última llamada contenida en el tope de la misma, y progresar en la ejecución del programa. Posteriormente extenderemos la definición de la máquina $\mathcal{H}$ para abstraer el manejo de variables definiendo así la máquina $\mathcal{J}$.

Finalmente se estudian los conceptos de alcance estático y dinámico que cada una de estas implementa junto con el concepto de cerradura.

1 La máquina $\mathcal{H}$

Para la implementación de este máquina de estados es necesaria la introducción de marcos. Los marcos representan un programa con marcadores de los cómputos pendientes que necesitamos para continuar con la evaluación de un determinado estado en la expresión de MinHS que queremos evaluar. Adicionalmente se presenta la pila de ejecución para llevar un control de dichos cómputos (en inglés se conocen como $frames$ pero en esta nota les llamaremos ”marcos”[2]).

Esta implementación resulta particularmente cómoda para seguir la ejecución de un programa de una manera simple y legible. Sin embargo, posteriores optimizaciones serán propuestas con la introducción de la máquina $\mathcal{J}$

1.1 Marcos y la pila de control

Los marcos (o $frames$ en inglés) son estructuras sintácticas que denotan cómputos pendientes en la evaluación de un programa. Por ejemplo, para el caso de los operadores, si ambos operandos no han sido evaluados exhaustivamente hasta ser reducidos a un valor, la ejecución del operador no puede continuar. Es aquí cuando los marcos resultan útiles para indicar que no se continuará con la evaluación del programa hasta no resolver los cómputos pendientes en dicho marco.

Adicionalmente tendremos una pila en la que almacenaremos todos los marcos pendientes para conservar el orden en el que se fueron encontrando en el programa. Cada marco será empujado al tope de la pila para evaluar las subexpresiones asociados a el.

[2]Este término es una traducción directa de [122]

Una vez concluida la evaluación de las subexpresiones de éste computo, el valor obtenido es empujado al siguiente marco en el tope de la pila y detenido hasta que todas las subexpresiones hayan sido terminadas de evaluar, ó se regresa como valor final si la pila está vacía (según sea el caso aplicable por axioma).

Definición 1.1 (Marcos)**.** Los marcos son marcadores estructurales que registran los cómputos pendientes de una expresión. En nuestra definición, el símbolo $\square$ indica el lugar en donde se está llevando acabo la evaluación actual[a].

Operadores primitivos

$$\frac{}{O(\square, e_2)\ marco} \qquad \frac{}{O(v_1, \square)\ marco}$$

Condicional

$$\frac{}{if(\square, e_2, e_3)\ marco}$$

Aplicación de función

$$\frac{}{app(e_1, \square)\ marco}$$

Asignaciones locales

$$\frac{}{let(\square, x.e_1)\ marco}$$

Notemos que los operadores fix, fun y recfun no tienen una definición con un marco. Esto es porque estas expresiones solo pueden ser aplicadas con la expresión app que define el marcador de cómputo pendiente para el argumento de la función.

[a]Definición formulada de [5], [8], [12], y [122]

Definición 1.2 (Pila de control)**.** Una pila de control está formada a partir de marcos, y se define recursivamente como sigue[a]:

$$\frac{}{\diamond\ pila}\ \text{vacia} \qquad \frac{\text{m}\ marco \quad \text{p}\ pila}{\text{m;p}\ pila}\ \text{top}$$

[a]Definición formulada de [5], [8] [12], y [122]

1.2 Estados

Los estados para la máquina $\mathcal{H}$ estarán agrupados en dos categorías; de evaluación y de retorno. Los estados de evaluación exhaustarán una expresión hasta obtener un valor y los de retorno empujarán dicho valor a la pila de control (los valores pueden ser retornados a un marco o regresados como valores finales).

Los estados iniciales comenzarán la evaluación de una expresión con una pila vacía y los estados finales retornarán un valor a una pila vacía.

Definición 1.3 (Estados de la máquina $\mathcal{H}$)**.** Los estados están compuestos de una pila de control P y una expresión e cerrada y son de alguna de las siguientes formas[a]:

- **Estados de evaluación**: Se evalúa e siendo P la pila de control y lo denotamos como $P \succ e$
- **Estados de retorno**: Devuelve el valor v a la pila de control P, que denotamos como $P \prec v$

En donde se distinguen dos tipos de estados en particular:

- **Estados iniciales**: comienzan la evaluación con la pila vacía denotados como $\diamond \succ e$.
- **Estados finales**: regresan un valor a la pila vacía y se denota $\diamond \prec v$

[a]Definición formulada de [5], [8] y [12]

1.3 Transiciones

La función de transición vincula la pila de ejecución con los marcos, definiendo las transiciones posibles entre cada uno de los estados.

Sí el cómputo pendiente de un marco ha terminado su evaluación, el valor regresa al marcador de lugar y se continúa con el resto de los cómputos. Sí todos los cómputos han sido resueltos entonces se evalúa lo operación con los valores retornados al marco.

Definición 1.4 (Transiciones para la máquina $\mathcal{H}$)**.** Las transiciones se definen por medio de la relación $\to_h$ y son de la forma[a] :

$$P \succ e \to_h P' \succ e'$$

Valores Los valores del lenguaje son números, booleanos y funciones. La evaluación de un valor simplemente lo regresa como resultado a la pila, pues un valor ya finalizó su proceso de evaluación.

$$\frac{}{P \succ v \to_h P \prec v}$$

Operaciones Definimos el proceso para evaluar cualquier operador de la siguiente forma: Para evaluar $O(e_1, e_2)$ agregamos el marco $O(\Box, e_2)$ a la pila y evaluamos e_1.

$$\frac{}{P \succ O(e_1, e_2) \to_h O(\Box, e_2); P \succ e_1}$$

Si tenemos en el tope de la pila el marco $O(\Box, e_2)$ y se regresa como resultado un valor v, entonces, evaluamos e_2 y sustituimos el tope de la pila por el marco $O(v_1, \Box)$.

$$\frac{}{O(\Box, e_2); P \prec v_1 \to_h O(v_1, \Box); P \succ e_2}$$

Si se devuelve un valor a la pila que tiene como tope el marco $O(v_1, \Box)$ entonces podemos devolver al resto de la pila el resultado de la operación de ambos valores.

$$\overline{O(v_1, \Box); P \prec v_2 \rightarrow_h P \prec v_1\ O\ v_2}$$

Condicional Para evaluar la expresión $if(e_1, e_2, e_3)$ agregamos el marco $if(\Box, e_2, e_3)$ al tope de la pila y evaluamos e_1.

$$\overline{P \succ if(e_1, e_2, e_3) \rightarrow_h if(\Box, e_2, e_3); P \succ e_1}$$

Si se regresa True a la pila con el marco $if(\Box, e_2, e_3)$ en el tope, entonces evaluamos e_2 con el resto de la pila.

$$\overline{if(\Box, e_2, e_3); P \prec \mathsf{True} \rightarrow_h P \succ e_2}$$

Si se regresa False a la pila con el marco $if(\Box, e_2, e_3)$ en el tope, entonces evaluamos e_3 con el resto de la pila.

$$\overline{if(\Box, e_2, e_3); P \prec \mathsf{False} \rightarrow_h P \succ e_3}$$

Asignaciones locales Si se quiere evaluar la expresión $let(v, x.e)$ con la pila P entonces se evalúa e en donde se sustituyen las apariciones de x por v.

$$\overline{P \succ let(e_1, x.e_2) \rightarrow_h let(\Box, x.e_2); P \succ e_1}$$

$$\overline{let(\Box, x.e); P \prec v \rightarrow_h P \succ e[x := v]}$$

Aplicación de función Para evaluar una aplicación $app(e_1, e_2)$ en una pila P se agrega el marco $app(\Box, e_2)$ como tope y se evalúa e_1. Si se regresa un valor $fun(x.e_1)$ a la pila con tope $app(\Box, e_2)$ entonces se quita el tope y se evalúa e_1 sustituyendo x por e_2.

$$\overline{app(\Box, e_2); P \prec fun(x.e_1) \rightarrow_h app(fun(x.e_1), \Box); P \succ e_2}$$

$$\overline{app(fun(x.e_1), \Box) : P \prec v_1 \rightarrow_h P \succ e_1[x := v_1]}$$

Si se regresa un valor $recfun(f.x.e_1)$ a la pila con tope $app(\Box, e_2)$ entonces se quita el tope y se evalúa e_1 sustituyendo f por su punto fijo y x por e_2.

$$\overline{app(\Box, e_2); P \prec recfun(f.x.e_1) \rightarrow_h app(recfun(f.x.e_1), \Box) : P \succ e_2}$$

$$\overline{app(recfun(f.x.e_1), \Box) : P \prec v_1 \rightarrow_h P \succ [f := fix(f.x.e_1), x := v_1]}$$

El operador de punto fijo Para evaluar la expresión $fix(f.e)$ en la pila P se evalúa e sustituyendo f por $fix(f.e)$.

$$\overline{P \succ fix(f.e) \rightarrow_h P \succ e[f := fix(f.e)]}$$

[a]Definición formulada de [5], [8] y [12]

Ejercicio 1.1 (Ejecución máquina $\mathcal{H}$ con *marcos*)**.** Para ver el funcionamiento de la máquina $\mathcal{H}$ vamos a evaluar la siguiente expresión:

```
let x = False in
    let y = 4 in
       if ( x then y + 1 else y - 1)
    end
end
```

En sintaxis abstracta:

$$let(\mathsf{False}, x.let(4, y.if(x, sum(y,1), sub(y,1)))))$$

La evaluamos en la máquina $\mathcal{J}$.

$$\begin{array}{rcll}
\diamond & \succ & let(\mathsf{False}, x.let(4, y.if(x, sum(y,1), sub(y,1)))) & \to_h \\
let(\square, x.let(4, y.if(x, sum(y,1), sub(y,1)))) : \diamond & \succ & \mathsf{False} & \to_h \\
let(\square, x.let(4, y.if(x, sum(y,1), sub(y,1)))) : \diamond & \prec & \mathsf{False} & \to_h \\
\diamond & \succ & let(4, y.if(x, sum(y,1), sub(y,1)))[x := \mathsf{False}] & \to_h \\
\diamond & \succ & let(4, y.if(\mathsf{False}, sum(y,1), sub(y,1))) & \to_h \\
let(\square, y.if(\mathsf{False}, sum(y,1), sub(y,1))) : \diamond & \succ & 4 & \to_h \\
let(\square, y.if(\mathsf{False}, sum(y,1), sub(y,1))) : \diamond & \prec & 4 & \to_h \\
\diamond & \succ & if(\mathsf{False}, sum(y,1), sub(y,1))[y := 4] & \to_h \\
\diamond & \succ & if(\mathsf{False}, sum(4,1), sub(4,1)) & \to_h \\
if(\square, sum(4,1), sub(4,1)) : \diamond & \succ & \mathsf{False} & \to_h \\
if(\square, sum(4,1), sub(4,1)) : \diamond & \prec & \mathsf{False} & \to_h \\
\diamond & \succ & sub(4,1) & \to_h \\
sub(\square, 1) : \diamond & \succ & 4 & \to_h \\
sub(\square, 1) : \diamond & \prec & 4 & \to_h \\
sub(4, \square) : \diamond & \succ & 1 & \to_h \\
sub(4, \square) : \diamond & \prec & 1 & \to_h \\
\diamond & \succ & 4 - 1 & \to_h \\
\diamond & \prec & 3 &
\end{array}$$

Ejercicio 1.2 (Ejecución máquina $\mathcal{H}$ con *marcos*)**.** Para ver el funcionamiento de la máquina $\mathcal{H}$ vamos a evaluar la siguiente expresión:

```
let x = fun z -> z + 1 in
    let y = 4 in
       x y
    end
end
```

En sintaxis abstracta:

$$let(fun(z.sum(z,1)), x.let(4, y.app(x,y)))$$

La evaluamos en la máquina $\mathcal{J}$.

$$
\begin{array}{rclr}
 \diamond & \succ & let(fun(z.sum(z,1)), x.let(4, y.app(x,y))) & \rightarrow_h \\
let(\square, x.let(4, y.app(x,y))) : \diamond & \prec & fun(z.sum(z,1)) & \rightarrow_h \\
 \diamond & \succ & let(4, y.app(x,y))[x := fun(z.sum(z,1))] & \rightarrow_h \\
 \diamond & \succ & let(4, y.app(fun(z.sum(z,1)), y)) & \rightarrow_h \\
let(\square, y.app(fun(z.sum(z,1)), y)) : \diamond & \succ & 4 & \rightarrow_h \\
let(\square, y.app(fun(z.sum(z,1)), y)) : \diamond & \prec & 4 & \rightarrow_h \\
 \diamond & \succ & app(fun(z.sum(z,1)), y)[y := 4] & \rightarrow_h \\
 \diamond & \succ & app(fun(z.sum(z,1)), 4) & \rightarrow_h \\
app(\square, 4) : \diamond & \succ & fun(z.sum(z,1)) & \rightarrow_h \\
app(\square, 4) : \diamond & \prec & fun(z.sum(z,1)) & \rightarrow_h \\
app(fun(z.sum(z,1)), \square) : \diamond & \succ & 4 & \rightarrow_h \\
app(fun(z.sum(z,1)), \square) : \diamond & \prec & 4 & \rightarrow_h \\
 \diamond & \succ & sum(z,1)[z := 4] & \rightarrow_h \\
 \diamond & \succ & sum(4,1) & \rightarrow_h \\
sum(\square, 1) : \diamond & \succ & 4 & \rightarrow_h \\
sum(\square, 1) : \diamond & \prec & 4 & \rightarrow_h \\
sum(4, square) : \diamond & \succ & 1 & \rightarrow_h \\
sum(4, square) : \diamond & \prec & 1 & \rightarrow_h \\
 \diamond & \succ & 4+1 & \rightarrow_h \\
 \diamond & \prec & 5 &
\end{array}
$$

2 La máquina $\mathcal{J}$

La máquina $\mathcal{J}$ es una extensión de la máquina $\mathcal{H}$ con la adición de un *cache* para almacenar variables y sus valores al que denominamos entorno. Este entorno será el encargado de regresar el valor asociado a un atributo en las operaciones de sustitución (let, lam, fix, fun, app). Esto con el objetivo de tener una implementación eficiente para la sustitución, operación que hasta el momento posee una complejidad lineal sobre el tamaño de la expresión que queremos evaluar.

Con pequeños ajustes podemos redefinir este nuevo modelo partiendo de las reglas definidas para $\mathcal{H}$ los cuales discutiremos a continuación.

2.1 Marcos

Definición 2.1 (Marcos)**.** En la máquina $\mathcal{J}$ se usa el mismo conjunto de marcos que los presentados para la máquina $\mathcal{H}$[a].

[a] Definición formulada de [5], [8] y [12]

2.2 Ambientes

Definición 2.2 (Ambientes)**.** Un ambiente es una estructura que almacena asignaciones de variables con su valor y la definimos recursivamente como[a]:

$$\frac{}{\bullet\ env}\ \text{vacio} \qquad\qquad \frac{x \quad var \qquad v \quad valor \qquad \bullet\ env}{x \leftarrow v\ ;\bullet\ env}\ \text{asig}$$

La forma de acceder a los elementos del ambiente es mediante el nombre de la variable, entonces si el ambiente $\mathbb{E}$ tiene la asignación $x \leftarrow v$. Podemos acceder al valor de x como $\mathbb{E}[x]$ y el resultado es v.

En caso de tener mas de una asignación sobre el mismo nombre de variable $\mathbb{E}[x]$ nos regresa la primera aparición de x en el ambiente. Por ejemplo en el ambiente $\mathbb{E} =_{def} x \leftarrow v_1; x \leftarrow v_2; \mathbb{E}$ la operación $\mathbb{E}[x]$ nos arroja como resultado v_1.

[a]Definición formulada de [5], [8] y [12]

2.3 Estados

Definición 2.3 (Estados de la máquina $\mathcal{J}$)**.** Los estados ahora son una relación ternaria de una pila de control P un ambiente $\mathbb{E}$ y una expresión e, denotados como[a]:

- Estados de evaluación: $P \mid \mathbb{E} \succ e$
- Estados de retorno: $P \mid \mathbb{E} \prec e$

Un estado inicial es de la forma:

$$\diamond \mid \bullet \succ e$$

Mientras que los estados finales son de la forma:

$$\diamond \mid \mathbb{E} \prec v$$

Notemos que en los estados finales no importa que está guardado en el ambiente solo importa que la pila de control esté vacía.

[a]Definición formulada de [5], [8] y [12]

2.4 Transiciones

Definición 2.4. Transiciones de variables en la máquina $\mathcal{J}$[a]:

Variables En la máquina $\mathcal{H}$ una variable representa un estado bloqueado, pues no hay forma de continuar la evaluación del programa si no se aplica una operación de sustitución sobre la misma.

En la máquina $\mathcal{J}$ no está definida la operación de sustitución. Tenemos que evaluar las variables buscándolas en el ambiente, lo que definimos con la siguiente regla:

$$\frac{}{P \mid \mathbb{E} \succ x \rightarrow_j P \mid \mathbb{E} \prec \mathbb{E}[x]}$$

[a]Definición formulada de [5], [8] y [12]

Queremos utilizar los ambientes para introducir el concepto de alcance de una variable[3]. De manera informal el alcance de una variable es la porción del programa donde la definición de dicha variable es válida, por ejemplo en la expresión:

$$let(e_1, x.e_2)$$

El alcance de x solo estará definido para el cuerpo de la expresión e_2, una vez que se termine de evaluar la expresión *let* queremos eliminar el ambiente donde la variable x tiene una asignación para evitar repetir la misma en futuras evaluaciones pues el alcance donde es válida ya se ha terminado.

Para esto debemos extender la pila de los marcos para que pueda almacenar ambientes y deshacerse de ellos.

Definición 2.5. Pila de control para $\mathcal{J}$[a]

$$\frac{}{P\ pila}\ \textsf{vacia} \qquad \frac{m\ marco \quad P\ pila}{m; P\ pila}\ \textsf{top} \qquad \frac{\mathbb{E}\ env \quad P\ pila}{\mathbb{E}\ ; P\ pila}\ \textsf{top}$$

De esta forma cuando evaluemos una expresión que genera un ambiente distinto (llamémosle ambiente B), se guarda el ambiente actual en la pila de control (llamémosle ambiente A) y se definen las nuevas asignaciones dentro del ambiente B.

Una vez que termine la ejecución y se regrese un valor a la pila, recuperamos el ambiente A y continuamos con la ejecución del programa.

[a]Definición formulada de [5], [8] y [12]

Finalmente enlistaremos las transiciones para la máquina $\mathcal{J}$ que difieren con la máquina $\mathcal{H}$. El resto de transiciones que no aparecen en esta definición se heredan directamente de esta última máquina.

Estas reglas permiten manipular los entornos para guardar, definir y sustituir variables durante la ejecución del programa. Una vez conlcuida los entornos serán descartados hasta liberar la pila de ejecución para retornar el valor final.

[3]Exploraremos de manera formal este concepto mas adelante en este capítulo.

Definición 2.6. Transiciones con alcance en la máquina $\mathcal{J}$[a].

Asignaciones locales

$$\frac{}{P \mid \mathbb{E} \succ let(e_1, x.e_2) \to_j let(\square, x.e_2); P \mid \mathbb{E} \succ e_1}$$

$$\frac{}{let(\square, x.e_2)\ ;\ P \mid \mathbb{E} \prec v \to_j \bullet\ ;\ P \mid x \leftarrow v\ ;\ \mathbb{E} \succ e_2}$$

Aplicación de función

$$\frac{}{app(\square, e_2); P \mid \mathbb{E} \prec v_1 \to_j app(v_1, \square); P \mid \mathbb{E} \succ e_2}$$

$$\frac{}{app(fun(x.e_1), \square)\ ;\ P \mid \mathbb{E} \prec v_2 \to_j \mathbb{E}\ ;\ P \mid x \leftarrow v_2; \mathbb{E} \succ e_1}$$

$$\frac{}{app(recfun(f.x.e_1), \square); P|E \prec v_2 \to_j \mathbb{E}; P|f \leftarrow fix(f.x.e_1); x \leftarrow v_2; \mathbb{E} \succ e_1}$$

Liberación del ambiente

$$\frac{}{\mathbb{E}; P \mid \mathbb{E}_1 \prec v \to_j P \mid \mathbb{E} \prec v}$$

Esta regla se agrega para liberar el ambiente de la pila de control.

[a] Definición formulada de [5], [8] y [12]

Ejercicio 2.1. Ejecución de la máquina $\mathcal{J}$ Tomemos como ejemplo la siguiente expresión de MinHS

```
let f = fun y => x + y + z in
    let y = 4 in
        let z = 10 in
            f 1
        end
    end
end
```

La cual nos da la siguiente representación en sintaxis abstracta:

$$let(fun(y.sum(x, sum(y, z)), f.let(4, y.let(10, z.app(f, 1))))$$

La evaluación en la máquina $\mathcal{J}$ es de la siguiente forma:

$$
\begin{array}{rccl}
\diamond \mid \bullet & \succ & let(fun(y...)) & \to_j \\
let(\square, f.let(...)) : \diamond \mid \bullet & \succ & fun(y.sum(x, sum(y,z))) & \to_j \\
let(\square, f.let(...)) : \diamond \mid \bullet & \prec & fun(y.sum(x, sum(y,z))) & \to_j \\
\diamond \mid f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & let(4, y.let(10, z.app(f,1))) & \to_j \\
let(\square, y.let(10, z.app(f,1))) : \diamond \mid f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & 4 & \to_j \\
let(\square, y.let(10, z.app(f,1))) : \diamond \mid \underline{f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet}_{E_1} & \prec & 4 & \to_j \\
E_1 : \diamond \mid y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & let(10, z.app(f,1)) & \to_j \\
let(\square, z, app(f,1)) : E_1 : \diamond \mid y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & 10 & \to_j \\
let(\square, z, app(f,1)) : E_1 : \diamond \mid \underline{y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet}_{E_2} & \prec & 10 & \to_j \\
E_2 : E_1 : \diamond \mid z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & app(f,1) & \to_j \\
app(\square, 1) : E_2 : E_1 : \diamond \mid z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & f & \to_j \\
app(\square, 1) : E_2 : E_1 : \diamond \mid z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & fun(x.sum(x, sum(y,z))) & \to_j \\
app(\square, 1) : E_2 : E_1 : \diamond \mid z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & fun(x.sum(x, sum(y,z))) & \to_j \\
app(fun(y.sum(x, sum(y,z))), \square) : E_2 : E_1 : \diamond \mid z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & 1 & \to_j \\
app(fun(y.sum(x, sum(y,z))), \square) : E_2 : E_1 : \diamond \mid \underline{z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet}_{E_3} & \prec & 1 & \to_j \\
E_3 : E_2 : E_1 : \diamond \mid y \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(y.sum(x, sum(y,z))) : \bullet & \prec & sum(x, sum(y,z)) & \to_j \\
sum(\square, sum(y,z)) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & x & \to_j \\
sum(\square, sum(y,z)) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & 1 & \to_j \\
sum(1, \square) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & sum(y,z) & \to_j \\
sum(\square, z) : sum(1, \square) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & y & \to_j \\
sum(\square, z) : sum(1, \square) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & 4 & \to_j \\
sum(4, \square) : sum(1, \square) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \succ & z & \to_j \\
sum(4, \square) : sum(1, \square) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & 10 & \to_j \\
sum(1, \square) : E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & 4+10 & \to_j \\
E_3 : E_2 : E_1 : \diamond \mid x \leftarrow 1 : z \leftarrow 10 : y \leftarrow 4 : f \leftarrow fun(x.sum(x, sum(y,z))) : \bullet & \prec & 14+1 & \to_j \\
E_2 : E_1 : \diamond \mid E_3 : \bullet & \prec & 14+1 & \to_j \\
E_1 : \diamond \mid E_2 : \bullet & \prec & 15 & \to_j \\
\diamond \mid E_1 : \bullet & \prec & 15 & \to_j
\end{array}
$$

2.5 Alcance

El alcance de una variable es la porción del programa en la cual esta está definida. Dependiendo de la implentación del lenguaje este alcance puede ser estático o dinámico.

Definición 2.7 (Alcance estático)**.** Para los programas que implementan alcance estático, las variables solo estarán definidas en un área delimitada por bloques de indentación, marcadores de lugar como el operador `let`, `in`, `end`, cuerpo de una función etc[a].

[a] Definición extraída de [2] y [107]

Definición 2.8 (Alcance dinámico)**.** En un lenguaje que implementa el alcance dinámico, el alcance de una variable es todo el programa y se toma la última asignación hecha a la misma cuando se requiera evaluar[a].

[a] Definición extraída de [2] y [107]

2.6 Cerraduras

Los alcances de las máquinas $\mathcal{H}$ y $\mathcal{J}$ suponen un problema dado que los resultados obtenidos al evaluar la misma expresión son distintos para las expresiones de tipo función.

Esto se debe de corregir para la definición de la máquina $\mathcal{J}$ mediante la introducción de cerraduras, que encapsulan el contexto de evaluación y la expresión para la que dicho contexto será aplicado cuando se evalúe.

Las cerraduras nos ayudan a imitar la definición de un alcance estático en la implementación de $\mathcal{J}$, para esto deberemos modificar las reglas de evaluación.

Definición 2.9 (Cerraduras)**.** Una cerradura es una pareja de una expresión de función de MinHS y un ambiente que se denota[a]:

$$\ll \mathbb{E}, f \gg$$

Cuya interpretación es que el ambiente adecuado para evaluar la función f es $\mathbb{E}$.

De esta forma se respeta el ambiente en el que se define una función para usar el mismo en su ejecución y tener una evaluación con alcance estático.

[a] Definición formulada de [5], [8] y [12]

2.7 Transiciones con cerradura para $\mathcal{J}$

Una vez hecho el análisis de la diferencia de alcances en ambas máquinas, necesitamos homogeneizar la evaluación para que el ambiente no introduzca efectos secundarios y el resultado de la evaluación de un programa en ambas máquinas difiera.

Solo las reglas que modifican el ambiente serán candidatas para ser ajustadas en la definición de las transiciones para la máquina $\mathcal{J}$, en particular app, fun, recfun y fix.

Definición 2.10 (Transición para funciones)**.** En lugar de regresar las funciones como una expresión, se guardará como una pareja de la expresión y el ambiente en el que fue definido[a].

$$\frac{}{P \mid \mathbb{E} \succ fun(x.e) \to_j P \mid \mathbb{E} \prec\ll \mathbb{E}, x.e \gg}$$

$$\frac{}{app(\square, e_2); P \mid \mathbb{E} \prec\ll \mathbb{E}_f, x.e_1 \gg\to_j app(\ll \mathbb{E}_f, x.e_1 \gg, \square); P \mid \mathbb{E} \succ e_2}$$

$$\frac{}{app(\ll \mathbb{E}_f, x.e_1 \gg, \square); P \mid \mathbb{E} \prec v \to_j \mathbb{E}; P \mid x \leftarrow v; \mathbb{E}_f \succ e_1}$$

$$\frac{}{P \mid \mathbb{E} \succ recfun(f.x.e) \to_j P \mid \mathbb{E} \prec fix(f. \ll \mathbb{E}, x.e \gg)}$$

$$\frac{}{app(\square, e_2); P|\mathbb{E} \prec fix(f. \ll \mathbb{E}_f, x.e_1 \gg) \to_j app(fix(f. \ll \mathbb{E}_f, x.e_1 \gg), \square); P \mid \mathbb{E} \succ e_2}$$

$$\frac{}{app(fix(f. \ll \mathbb{E}_f, x.e_1 \gg), \square); P \mid \mathbb{E} \prec v \to_j \mathbb{E}; P \mid x \leftarrow v; f \leftarrow fix(f. \ll \mathbb{E}_f, x.e_1 \gg); \mathbb{E}_f \succ e_1}$$

[a] Definición formulada de [5], [8] y [12]

Para finalizar el capítulo presentamos un ejemplo de la evaluación en la máquiina $\mathcal{J}$ con las nuevas reglas para mostrar como ahora se pueden guardar ambientes y sustituirlos según se necesite durante el proceso de evaluación.

Esto nos da la flexibilidad de capturar un ambiente para una porción específica del programa sin perder el resto de las asignaciones contenidas en el.

Una vez concluida la evaluación de la instrucción para la cual un ambiente fue guardado, se restaura el ambiente general para continuar con el resto del programa.

Ejercicio 2.2.

```
let y = 4 in
    let z = 10 in
        let f = fun x => x + y + z in
            f 1
        end
    end
end
```

La cual nos da la siguiente representación en sintaxis abstracta:

$$let(4, y.let(10, z.let(fun(x.sum(x, sum(y, z)), f.app(f, 1)))))$$

La evaluación en la máquina $\mathcal{J}$ es de la siguiente forma:

$$
\begin{array}{rcll}
\diamond \mid \bullet & \succ & let(4, y.let(10, z.let(fun(x.sum(x, sum(y, z)), f.app(f, 1)))) & \rightarrow_j \\
let(\square, y.let...) : \diamond \mid \bullet & \succ & 4 & \rightarrow_j \\
let(\square, y.let...) : \diamond \mid \bullet & \prec & 4 & \rightarrow_j \\
\bullet : \diamond \mid y \leftarrow 4 : \bullet & \succ & let(10, z.let(fun(x.sum(x, sum(y, z)), f.app(f, 1))) & \rightarrow_j \\
let(\square, z.let...) : \bullet : \diamond \mid y \leftarrow 4 : \bullet & \succ & 10 & \rightarrow_j \\
let(\square, z.let...) : \bullet : \diamond \mid \underline{y \leftarrow 4 : \bullet}_{\ E_1} & \prec & 10 & \rightarrow_j \\
E_1 : \bullet : \diamond \mid z \leftarrow 10 : E_1 & \succ & let(fun(x.sum(x, sum(y, z)), f.app(f, 1)) & \rightarrow_j \\
let(\square, f.app...) : E_1 : \bullet : \diamond \mid z \leftarrow 10 : E_1 & \succ & fun(x.sum(x, sum(y, z))) & \rightarrow_j \\
let(\square, f.app...) : E_1 : \bullet : \diamond \mid \underline{z \leftarrow 10 : E_1}_{\ E_2} & \prec & \ll E_2, fun(x.sum(x, sum(y, z))) \gg & \rightarrow_j \\
E_2 : E_1 : \bullet : \diamond \mid f \leftarrow \ll E_2, fun(x.sum(x, sum(y, z))) \gg : E_2 & \prec & app(f, 1) & \rightarrow_j \\
app(\square, 1) : E_2 : E_1 : \bullet : \diamond \mid f \leftarrow \ll E_2, fun(x.sum(x, sum(y, z))) \gg : E_2 & \succ & f & \rightarrow_j \\
app(\square, 1) : E_2 : E_1 : \bullet : \diamond \mid f \leftarrow \ll E_2, fun(x.sum(x, sum(y, z))) \gg : E_2 & \succ & \ll E_2, fun(x.sum(x, sum(y, z))) \gg & \rightarrow_j \\
app(\square, 1) : E_2 : E_1 : \bullet : \diamond \mid f \leftarrow \ll E_2, fun(x.sum(x, sum(y, z))) \gg : E_2 & \prec & \ll E_2, fun(x.sum(x, sum(y, z))) \gg & \rightarrow_j \\
app(\ll E_2, fun(y.sum(x, sum(y, z))) \gg, \square) : E_2 : E_1 : \bullet : \diamond \mid f \leftarrow \ll E_2, fun(x.sum(x, sum(y, z))) \gg : E_2 & \succ & 1 & \rightarrow_j \\
app(\ll E_2, fun(y.sum(x, sum(y, z))) \gg, \square) : E_2 : E_1 : \bullet : \diamond \mid \underline{f \leftarrow \ll E_2, fun(x.sum(x, sum(y, z))) \gg : E_2}_{\ E_3} & \prec & 1 & \rightarrow_j \\
E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \succ & sum(x, sum(y, z)) & \rightarrow_j \\
sum(\square, sum(y, z)) : E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \succ & x & \rightarrow_j \\
sum(\square, sum(y, z)) : E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \prec & 1 & \rightarrow_j \\
sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond \mid \ : x \leftarrow 1 E_2 & \succ & sum(y, z) & \rightarrow_j \\
sum(\square, z) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \succ & y & \rightarrow_j \\
sum(\square, z) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \prec & 4 & \rightarrow_j \\
sum(4, \square) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \succ & z & \rightarrow_j \\
sum(4, \square) : sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond \mid x \leftarrow 1 : E_2 & \prec & 10 & \rightarrow_j \\
sum(1, \square) : E_3 : E_2 : E_1 : \bullet : \diamond \mid \ : x \leftarrow 1 : E_2 & \prec & 10 + 4 & \rightarrow_j \\
E_3 : E_2 : E_1 : \bullet : \diamond \mid \ : x \leftarrow 1 : E_2 & \prec & 14 + 1 & \rightarrow_j \\
E_2 : E_1 : \bullet : \diamond \mid E_3 & \prec & 15 & \rightarrow_j \\
E_1 : \bullet : \diamond \mid E_2 & \prec & 15 & \rightarrow_j \\
\bullet : \diamond \mid E_1 & \prec & 15 & \rightarrow_j \\
\diamond \mid \bullet & \prec & 15 &
\end{array}
$$

Como podemos observar el resultado obtenido ahora es el mismo en ambas máquinas, evaluando a 15.

De esta forma concluimos el estudio de la implementación de las máquinas abstractas para MinHS. A continuación se encuentran los ejercicios de comprensión para el lector.

3 Ejercicios para el lector

Ejercicio 3.1. Dada la siguiente expresión de MinHS contesta lo siguiente:

```
let x = False in
    let y = fun z => leq(z,0) in
        let x = True in
            if x then y 0 else False
        end
    end
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{H}$.
- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{J}$.
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina $\mathcal{J}$ con cerraduras
- El resultado obtenido es el mismo?.

Ejercicio 3.2. Dada la siguiente expresión de MinHS contesta lo siguiente:

```
let recfun pow x n =
    if  n = 0 then 1 else x * pow x  n-1 in
        pow 2 4
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{H}$.
- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{J}$.
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina $\mathcal{J}$ con cerraduras.
- El resultado obtenido es el mismo?.

Ejercicio 3.3. Dada la siguiente expresión de MinHS contesta lo siguiente:

```
let recfun fib n =
    if n < 1 then 1 else n  +  fib n - 1 in
        fib 5
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{H}$.

- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{J}$.
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina $\mathcal{J}$ con cerraduras.
- El resultado obtenido es el mismo?.

Ejercicio 3.4. Dada la siguiente expresión de MinHS contesta lo siguiente:

```
let recfun fac n =
    if n < 1 then 1 else n * fac n - 1 in
        fac 4
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{H}$.
- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{J}$.
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina $\mathcal{J}$ con cerraduras.
- El resultado obtenido es el mismo?.

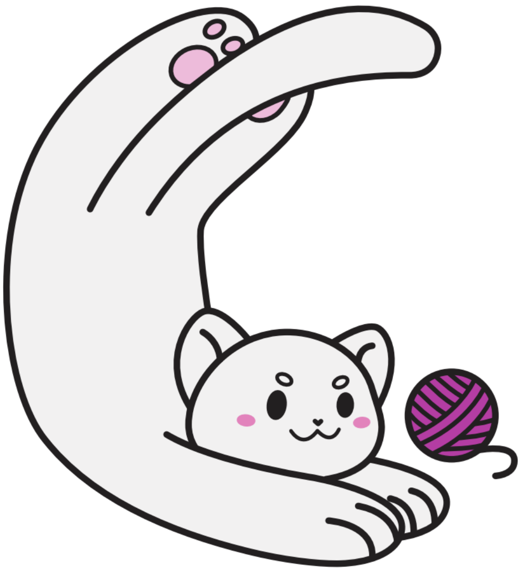
Ejercicio 3.5. Dada la siguiente expresión de MinHS contesta lo siguiente:

```
let x = True in
    let x = False in
        if x then 1 else 0
    end
end
```

- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{H}$.
- Evalúa la expresión de acuerdo a la definición de la máquina $\mathcal{J}$.
- Los resultados obtenidos son los mismos o son diferentes?.
- Evalúa la expresión utilizando la definición extendida para la máquina $\mathcal{J}$ con cerraduras.
- El resultado obtenido es el mismo?.

Capítulo 9

TinyC



Hasta este punto en el manual hemos discutido la implementación de diferentes modelos de cómputo como MinHs, que nos fue de utilidad para ilustrar el paradigma funcional. La realidad es que muchos de los lenguajes que utilizamos actualmente no se implementan de esta forma.

Es aquí cuando haremos un cambio abrupto en nuestro caso de estudio para poner nuestra atención en el paradigma procedimental cuyo primer ejemplo sera una implementación de un lenguaje de programación basado en C conocido como TinyC.

Este tipo de lenguajes requieren del manejo de estados los cuales son afectados por el contexto del programa, los cambios en la memoria y las instrucciones que se ejecutan en un determinado momento. Similar a un autómata o a una máquina de estados.

Objetivo

En este capítulo comenzamos el estudio del paradigma procedimental, para esto nuestra interés principal será el de definir el primer caso de estudio conocido como TinyC. Proporcionando su sintaxis, reglas de transición y manejo de memoria.

Planteamiento

Se revisarán los componentes básicos de este lenguaje, a saber la semántica operacional mediante la implementación de la máquina $\mathcal{C}$.

Proporcionaremos una definición completa incluyendo sus marcos, estados y transiciones. Adicionalmente se discutirá la semántica estática de TinyC para asignar un tipo a los programas de este lenguaje.

0.1 Sintaxis

En TinyC tendremos las declaraciones de variables y funciones que se pueden identificar por nombre, las expresiones aritméticas y booleanas para construir la lógica de los programas y por último las sentencias encargadas de la ejecución y control del estado del programa.

Definición 0.1. Sintaxis Concreta de TinyCl[a].

De esta definición es importante notar la omisión entera de apuntadores dado que el tema escapa del enfoque de este capítulo quedándonos únicamente con un subconjunto de expresiones del lenguaje de programación C.

$$
\begin{array}{rcl}
\mathsf{progam} & ::= & \mathsf{global\text{-}decs}\;\; \mathsf{stmt} \\
\mathsf{global\text{-}decs} & ::= & \varepsilon \mid \mathsf{global\text{-}dec}\quad \mathsf{global\text{-}decs} \\
\mathsf{global\text{-}dec} & ::= & \mathsf{fun\text{-}dec} \mid \mathsf{var\text{-}dec} \\
\mathsf{var\text{-}decs} & ::= & \varepsilon \mid \mathsf{var\text{-}dec}\;\; \mathsf{var\text{-}decs} \\
\mathsf{var\text{-}dec} & ::= & \mathsf{type}\;\; \mathsf{ident} = \mathsf{expr}\ ; \\
\mathsf{fun\text{-}dec} & ::= & \mathsf{type}\;\; \mathsf{ident}\ (\mathsf{arguments})\;\; \mathsf{stmt}\ ; \\
\mathsf{stmt} & ::= & \mathsf{expr}\ ; \mid \mathsf{if}\;\; \mathsf{expr}\;\; \mathtt{then}\;\; \mathsf{stmt}\ \mathtt{else}\;\; \mathsf{stmt}\ ; \mid \mathsf{if}\;\; \mathsf{expr}\;\; \mathtt{then}\;\; \mathsf{stmt}; \\
 & & \mid \mathsf{return}\;\; \mathsf{expr}\ ; \mid \{\mathsf{var\text{-}decs}\;\; \mathsf{stmts}\} \mid \mathsf{while}\ (\mathsf{expr})\;\; \mathsf{stmt} \\
\mathsf{stmts} & ::= & \varepsilon \mid \mathsf{stmt}\quad \mathsf{stmts} \\
\mathsf{expr} & ::= & \mathsf{num} \mid \mathsf{ident} \mid \mathsf{asig} \mid \mathsf{expr} + \mathsf{expr} \mid \mathsf{expr}\ -\ \mathsf{expr} \\
 & & \mid \mathsf{expr} > \mathsf{expr} \mid \mathsf{expr}\ <\ \mathsf{expr} \mid \mathsf{ident}\ (\mathsf{exprs}) \\
\mathsf{asig} & ::= & \mathsf{ident} = \mathsf{expr} \\
\mathsf{exprs} & ::= & \mathsf{expr} \mid \mathsf{expr}, \mathsf{exprs} \\
\mathsf{arguments} & ::= & \varepsilon\ |\mathsf{type}\;\; \mathsf{ident}, \mathsf{arguments} \\
\mathsf{type} & ::= & \mathit{Int} \mid \mathit{Bool}
\end{array}
$$

[a]Definición formulada de [126], y [127]

Es importante remarcar el impacto que la introducción de las variables en este paradigma tiene en el estado de un programa, dado que el valor que contienen puede ser modificando alterando así el estado mismo del programa en cualquier punto. Esto no ocurría con la introducción de las variables en la sintaxis de orden superior para MinHs.

Ejercicio 0.1. Escribe un programa en TinyC que implemente una función que nos permita calcular el enésimo número de la sucesión de Fibonacci.

```
fibonacci(Int n){

    Int i = ;
    Int pre = 1;
    Int post = 1;
    Int fib;

    while(i < n){
        fib = aux2 + aux1;
        pre = post;
        post = fib;
    }
    return fib;
};

Int result = fibonacci(9);
```

1 La Máquina $\mathcal{C}$

Para la semántica operacional de TinyC vamos a definir una máquina abstracta siguiendo la misma línea del capítulo anterior, donde contaremos con una pila para guardar los marcos de ejecución y cómputos pendientes y un contexto para las variables y los valores asignados a éstas.

1.1 Marcos

Para la sección de expresiones vamos a tener un mapeo directo con los marcos para los operadores de la máquina $\mathcal{J}$ dado que la categoría *expr* de TinyC también regresa un valor al ser evaluadas.

Una diferencia importante a notar es la manera en la que las funciones son declaradas en MinHs. Aquí la declaración se hace mediante funciones anónimas que pueden ir en cualquier parte del programa, mientras que en TinyC y en C deben ser declaradas antes de ser llamadas. Adicionalmente estas declaraciones de funciones pueden ser múltiples mientras que en TinyC solo pueden declarase un valor a la vez.

Definición 1.1. Marcos de TinyC[a].

Se definen los siguientes marcos que usaremos en la pila de control de la máquina $\mathcal{C}$.

Estos marcos corresponden a la evaluación de nuevas instrucciones como return que evalúa el contenido hasta obtener un valor, secu que nos ayuda a escribir una secuencia de instrucciones y call que es el equivalente a app de MinHs.

Declaraciones

$$\overline{vardec(T, x, \square)\ marco}$$

Asignaciones

$$\overline{asig(x, \square)\ marco}$$

Secuencia

$$\overline{secu(\square, e_2)\ marco}$$

Condicionales

$$\overline{if(\square, e_2, e_3)\ marco} \qquad \overline{if(\square, e_2)\ marco}$$

Return

$$\overline{return(\square)\ marco}$$

Llamada a función

$$\overline{call(f, \square, e_2, \ldots, e_n)\ marco} \quad \ldots \quad \overline{call(f, v_1, v_2, \ldots, \square)\ marco}$$

[a]Definición formulada de [5], [8] y [12]

1.2 Estados

Esta categoría introduce una diferencia sustancial con las máquinas abstractas hasta ahora estudiadas. Tendremos un elemento para la pila de cómputos pendientes y dos categorías para el contexto de evaluación de las variables, el primero local representando el alcance de las variables declaradas en el cuerpo de una función y uno global para aquellas declaraciones cuyo contexto sea todo el programa.

Definición 1.2. Estados de la máquina $\mathcal{C}$[a].

Los estados de la máquina $\mathcal{C}$ son de la siguiente forma:

$$P \mid L \mid G \succ e \qquad\qquad P \mid L \mid G \prec e$$

En donde P es una pila de control, L y G son ambientes de variables y e es una expresión del lenguaje. Esta relación nos permite almacenar marcos de cómputos pendientes en P, ambientes previos en L y evaluar la expresión e con el ambiente G.

[a]Definición formulada de [5], y [8]

1.3 Ambientes

Los ambientes serán una lista de asignaciones de la forma:

$$\text{variable} \rightarrow \text{valor}$$

A continuación vamos a definir las funciones para hacer referencia al valor de una variable por nombre (búsqueda), y la función para modificar el contenido de una variable (actualización).

Definición 1.3. Referencia de variable en ambiente para la máquina $\mathcal{C}$[a].

$$\frac{}{\bullet[x] = \; fail} \qquad \frac{}{x \leftarrow v; \mathbb{E}[x] = v} \qquad \frac{}{y \leftarrow v; \mathbb{E}[x] = \mathbb{E}[x]}$$

Esta función simplemente itera sobre la lista de variables contenidas en el ambiente hasta encontrar un *match*. Sí la búsqueda llega hasta la pila vacía sin encontrar nada regresamos un $fail$.

[a]Definición formulada de [5], y [8]

Definición 1.4. Modificación del ambiente para una variable en la máquina $\mathcal{C}$[a].

$$\frac{}{\bullet[x \rightarrowtail v] = fail} \qquad \frac{}{x \leftarrow u; \mathbb{E}[x \rightarrowtail v] = x \leftarrow v; \mathbb{E}}$$

$$\frac{}{y \leftarrow u; \mathbb{E}[x \rightarrowtail v] = \mathbb{E}[x \rightarrowtail v]}$$

Esta función itera sobre la lista hasta hacer un *match* y actualiza el valor contenido en esta variable. Sí la búsqueda llega hasta la pila vacía sin encontrar nada regresamos un $fail$.

[a]Definición formulada de [5], y [8]

1.4 Transiciones

Definición 1.5. Transiciones de la máquina $\mathcal{C}$ [a]

Las transiciones de la máquina $\mathcal{C}$ se definen en términos de los programas que se están evaluando. Como vimos en secciones anteriores en el caso de TinyC un programa no tiene un resultado final, es decir no se reduce a un valor. Por lo que se agrega un programa especifico que indica el final de la ejecución, este programa es el programa vacío y se denota como:

$$\perp$$

y define el final del proceso de evaluación de una sentencia, por lo que los estados finales de la máquina $\mathcal{C}$ son los que tienen la forma siguiente:

$$\diamond \mid L \mid G \prec \perp$$

Con lo que se definen las transiciones con las siguientes reglas:

Secuencia

$$\frac{}{P \mid L \mid G \ \succ secu(e_1, e_2) \rightarrow_C secu(\square, e_2); P \mid L \mid G \ \succ e_1}$$

$$\frac{}{secu(\square, e_2); P \mid L \mid G \ \prec \bot \rightarrow_C \ P \mid L \mid G \ \succ e_2}$$

Declaraciones

$$\frac{}{P \mid L \mid G \ \succ \ \ vardec(T, x, e) \rightarrow_C vardec(T, x, \square); \ P \mid L \mid G \ \succ e}$$

$$\frac{G[x] = fail}{vardec(T, x, \square); P \mid L \mid G \ \prec v \rightarrow_C \ P \mid L \mid x \leftarrow v; G \prec \bot}$$

$$\frac{}{P|L|G \succ fundec(T, f, x_1 : T_1. \cdots .x_n : T_n.e) \rightarrow_C P|L|f \leftarrow x_1. \cdots .x_n.e; G \prec \bot}$$

Asignación

$$\frac{}{P \mid L \mid G \succ asig(x, e) \rightarrow_C \ asig(x, \square); P| \ L \mid G \ \succ e}$$

$$\frac{G[x \rightarrowtail v] = G'}{asig(x, \square); P \mid L \mid G \ \prec v \rightarrow_C P \mid L \mid G' \prec \bot}$$

$$\frac{G[x \rightarrowtail v] = fail \qquad L[x \rightarrowtail v] = L'}{asig(x, \square); P \mid L \mid G \ \prec v \rightarrow_C P \mid L' \mid G \prec \bot}$$

Condicionales

$$\frac{}{P \mid L \mid G \ \succ if(e_1, e_2, e_3) \rightarrow_C if(\square, e_2, e_3); P \mid L \mid G \ \succ e_1}$$

$$\frac{}{if(\square, e_2, e_3); P \mid L \mid G \ \prec true \rightarrow_C P \mid L \mid G \ \succ e_2}$$

$$\frac{}{if(\square, e_2, e_3); P \mid L \mid G \ \prec false \rightarrow_C P \mid L \mid G \ \succ e_3}$$

$$\frac{}{P \mid L \mid G \ \succ if(e_1, e_2) \rightarrow_C if(\square, e_2); P \mid L \mid G \ \succ e_1}$$

$$\frac{}{if(\square, e_2); P \mid L \mid G \ \prec true \rightarrow_C P \mid L \mid G \ \succ e_2}$$

$$\frac{}{if(\square, e_2); P \mid L \mid G \ \prec false \rightarrow_C P \mid L \mid G \ \prec \bot}$$

Variables

$$\frac{G[x] = v}{P \mid L \mid G \succ x \to_C P \mid L \mid G \prec v}$$

$$\frac{G[x] = fail \qquad L[x] = v}{P \mid L \mid G \succ x \to_C P \mid L \mid G \ \prec v}$$

While

$$\frac{}{P \mid L \mid G \ \succ while(e_1, e_2) \to_C P \mid L \mid G \ \succ if(e_1, secu(e_2, while(e_1, e_2)))}$$

Esta regla traduce la evaluación de un $while$ a un if con un solo caso. Modela la siguiente regla equivalencia entre programas:

$$while(e_1)\{e_2\} \equiv if\,(e_1)\,\{\,e_2\,;\,while\,(e_1)\{\,e_2\,\}\}$$

Observe como la expresión $while$ sigue apareciendo en el lado derecho. De esta forma se mantiene el ciclo tantas veces como se cumpla la condición.

Llamada a función

$$\frac{}{P \mid L \mid G \ \succ call(f, e_1, \cdots, e_n) \to_C call(\square, e_1, \cdots, e_n); P \mid L \mid G \ \succ f}$$

$$\frac{}{call(\square, e_1, \cdots, e_n); P \mid L \mid G \ \prec v \to_C call(v, \square, e_2, \cdots, e_n); P \mid L \mid G \ \succ e_1}$$

$$\vdots$$

$$\frac{}{call(x_1. \cdots x_n.e, v_1, \cdots, \square); P \mid L \mid G \ \prec v_n \to_C P \mid G \bigstar L \mid x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; \bullet \succ e}$$

Para la evaluación de una llamada a función, primero es necesario evaluar cada uno de los parámetros con los que se llama.

Una vez que todos son valores, entonces se ejecuta el cuerpo de la función usando un ambiente vacío como ambiente principal y agregando a el los parámetros de la llamada.

Guardamos el ambiente principal anterior con un símbolo especial ($\bigstar$) que sirve como separado para saber en donde termina uno y comienza el otro.

Return

$$\frac{}{P \mid L \mid G \ \succ return(e) \to_C return(\square);\ P \mid L \mid G \ \succ e}$$

$$\frac{}{return(\square); P \mid G \bigstar L_1 \mid L_2 \prec v \to_C P \mid L_1 \mid G \ \prec v}$$

El constructor *return* indica el final de la ejecución de una llamada a función.

Cuando termina la ejecución de una llamada a función. Se restaura el ambiente global y nos deshacemos del local pues solo era necesario dentro del cuerpo de la función.

[a]Definición formulada de [5], y [8]

Ejercicio 1.1 (Ejecución de la máquina C)**.** Dado el siguiente programa de TinyC Evalúa el programa para obeter el resultado de acuerdo a las reglas de transición y estados definidos en la sección anterior para la máquina $\mathcal{C}$:

```
fibonacci(Int n){

    Int i = ;
    Int pre = 1;
    Int post = 1;
    Int fib = 0;

    while(i < n){
        fib = pre + post;
        pre = post;
        post = fib;
        i++;
    }
    return fib;
};

Int result = fibonacci(3);
```

Para facilitar la representación y la evaluación de la máquina $\mathcal{C}$ vamos a seccionar el programa para tener fragmenos de este. Definimos entonces:

$$A = secu(vardec(Int, i, 1), secu(vardec(Int, pre, 1), secu(vardec(Int, post, 1), vardec(Int, fib, 0))))$$

$$B = secu(assig(fib, sum(pre, post)), secu(asig(pre, post), secu(asig(i, sum(i{+}1), asig(post, fib)))$$

$$W = while(lt(i, n), B))$$

$$F = fundec(Int, fibonacci, n\ :\ Int.secu(A, secu(W, return(fib))$$

$$P = secu(F, vardec(Int, result, call(fibonacci, 3)))$$

En la escritura de los ejercicios también se optó por el renombramiento de los contextos cuando se necesita almacenar, modificar o agregar alguna definición de variable o función. El contexto puede ser renombrado utilizando la notación

$$\underline{v_1 \to n_1, ..., v_n \to n_k}_{L_n}$$

$$
\begin{array}{rccr}
\diamond \mid \bullet \mid \bullet & \succ & secu(F, vardec(Int, result, call(fibonacci, 3))) & \rightarrow_{\mathcal{C}} \\
secu(\square, vardec(Int, result, call(fibonacci, 3))) : \diamond \mid \bullet \mid \bullet & \succ & fundec(Int, fibonacci, n : Int.secu(A, secu(W, return(fib))) & \rightarrow_{\mathcal{C}} \\
secu(\square, vardec(Int, result, call(fibonacci, 3))) : \diamond \mid \bullet \mid fibonacci \leftarrow n.secu(A, secu(W...))\bullet & \prec & \perp & \rightarrow_{\mathcal{C}} \\
\diamond \mid \bullet \mid fibonacci \leftarrow n.secu(A, secu(W...))\bullet & \succ & vardec(Int, result, call(fibonacci, 3)) & \rightarrow_{\mathcal{C}} \\
vardec(Int, result, \square) : \diamond \mid \bullet \mid fibonacci \leftarrow n.secu(A, secu(W...))\bullet & \succ & call(fibonacci, 3) & \rightarrow_{\mathcal{C}} \\
call(\square, 3) : vardec(Int, result, \square) : \diamond \mid \bullet \mid fibonacci \leftarrow n.secu(A, secu(W...)) : \bullet & \succ & fibonacci & \rightarrow_{\mathcal{C}} \\
call(\square, 3) : vardec(Int, result, \square) : \diamond \mid \bullet \mid fibonacci \leftarrow n.secu(A, secu(W...)) : \bullet & \prec & n.secu(A, secu(W...)) & \rightarrow_{\mathcal{C}} \\
call(n.secu(A, secu(W...)), \square) : vardec(Int, result, \square) : \diamond \mid \bullet \mid fibonacci \leftarrow n.secu(A, secu(W...)) : \bullet & \succ & 3 & \rightarrow_{\mathcal{C}} \\
call(n.secu(A, secu(W...)), \square) : vardec(Int, result, \square) : \diamond \mid \bullet \mid \underline{fibonacci \leftarrow n.secu(A, secu(W...)) : \bullet}_{L_1} & \prec & 3 & \rightarrow_{\mathcal{C}} \\
vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid n \leftarrow 3 : \bullet & \succ & secu(A, secu(W...)) & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid n \leftarrow 3 : \bullet & \succ & secu(vardec(Int, i, 1), ...) & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(vardec(Int, pre...))) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid n \leftarrow 3 : \bullet & \succ & vardec(Int, i, 1) & \rightarrow_{\mathcal{C}} \\
vardec(Int, i, \square) : secu(\square, secu(vardec(Int, pre...))) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid n \leftarrow 3 : \bullet & \succ & 1 & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(vardec(Int, pre...))) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid i \leftarrow 1 : n \leftarrow 3 : \bullet & \prec & \perp & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & secu(vardec(Int, pre...)) & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(vardec(Int, post...))) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & vardec(Int, pre, 1) & \rightarrow_{\mathcal{C}} \\
vardec(Int, pre, \square) : secu(\square, secu(vardec(Int, post...))) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & 1 & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(vardec(Int, post...))) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet & \prec & \perp & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & secu(vardec(Int, post...))) & \rightarrow_{\mathcal{C}} \\
secu(\square, vardec(Int, fib...)) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & vardec(Int, post, 1) & \rightarrow_{\mathcal{C}} \\
vardec(Int, post, \square) : secu(\square, vardec(Int, fib...)) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & 1 & \rightarrow_{\mathcal{C}} \\
secu(\square, vardec(Int, fib...)) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet & \prec & \perp & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 :\succ & \succ & vardec(Int, fib, 0) & \rightarrow_{\mathcal{C}} \\
vardec(Int, fib, \square) : secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet & \succ & 0 & \rightarrow_{\mathcal{C}} \\
secu(\square, secu(W...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid \underline{fib \leftarrow 0 : post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet}_{L_2} & \prec & \perp & \rightarrow_{\mathcal{C}} \\
vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid L_2 & \succ & secu(while(lt(i, n), B), return(fib)) & \rightarrow_{\mathcal{C}} \\
secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid L_2 & \succ & while(lt(i, n), B) & \rightarrow_{\mathcal{C}} \\
secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid L_2 & \succ & if(lt(i, n), secu(B, while(...))) & \rightarrow_{\mathcal{C}} \\
if(\square, secu(...)) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid L_2 & \succ & lt(i, n) & \rightarrow_{\mathcal{C}} \\
if(\square, secu(...)) : vardec(Int, result, \square) : \diamond \mid L_1 \bigstar \bullet \mid L_2 & \succ & lt(i, n) & \rightarrow_{\mathcal{C}}
\end{array}
$$

$lt(\square, n) : if(\square, secu(...)) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	i	$\to_C$
$lt(\square, n) : if(\square, secu(...)) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\prec$	1	$\to_C$
$lt(1, \square) : if(\square, secu(...)) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	n	$\to_C$
$lt(1, \square) : if(\square, secu(...)) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\prec$	3	$\to_C$
$if(\square, secu(...)) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\prec$	$true$	$\to_C$
$secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	$secu(B, while(...)))$	$\to_C$
$secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	$secu(asig(fib, sum(pre, post)...))$	$\to_C$
$secu(\square, secu(asig(pre, post), ...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	$asig(fib, sum(pre, post))$	$\to_C$
$asig(fib, \square) : secu(\square, secu(asig(pre, post), ...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	$sum(pre, post)$	$\to_C$
$sum(\square, post) : asig(fib, \square) : secu(\square, secu(asig(pre, post), ...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	pre	$\to_C$
$sum(\square, post) : asig(fib, \square) : secu(\square, secu(asig(pre, post), ...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\prec$	1	$\to_C$
$sum(1, \square) : asig(fib, \square) : secu(\square, secu(asig(pre, post), ...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\succ$	$post$	$\to_C$
$sum(1, \square) : asig(fib, \square) : secu(\square, secu(asig(pre, post), ...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\prec$	1	$\to_C$
$asig(fib, \square) : secu(\square, secu(asig(pre, post), ...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_2$	$\prec$	2	$\to_C$
$secu(\square, secu(asig(pre, post), ...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 2 : post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet}_{L_3}$	$\prec$	$\bot$	$\to_C$
$secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	$secu(asig(pre, post), ...)))$	$\to_C$
$secu(\square, secu(asig(i...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid fib \leftarrow 2 : post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 1 : n \leftarrow 3 : \bullet$	$\succ$	$asig(pre, post)$	$\to_C$
$assig(pre, \square) : secu(\square, secu(asig(i...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	$post$	$\to_C$
$assig(pre, \square) : secu(\square, secu(asig(i...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\prec$	1	$\to_C$
$secu(\square, secu(asig(i...))) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\prec$	$\bot$	$\to_C$
$secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	$secu(assig(i, sum(i, 1)), asig(post, fib))$	$\to_C$
$secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	$assig(i, sum(i, 1))$	$\to_C$
$assig(i, \square) : secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	$sum(i, 1)$	$\to_C$
$sum(\square, 1) : assig(i, \square) : secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	i	$\to_C$
$sum(\square, 1) : assig(i, \square) : secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\prec$	1	$\to_C$
$sum(1, \square) : assig(i, \square) : secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\succ$	1	$\to_C$
$sum(1, \square) : assig(i, \square) : secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\prec$	1	$\to_C$
$assig(i, \square) : secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_3$	$\prec$	2	$\to_C$
$secu(\square, asig(post, fib)) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 2 : post \leftarrow 1 : pre \leftarrow 1 : i \leftarrow 2 : n \leftarrow 3 : \bullet}_{L_4}$	$\prec$	$\bot$	$\to_C$
$secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_4$	$\succ$	$asig(post, fib)$	$\to_C$
$asig(post, \square) : secu(\square, while(...))) : secu(\square, return(fib)) : vardec(Int, result, \square) : \diamond \mid L_1\bigstar\bullet \mid L_4$	$\succ$	fib	$\to_C$

$asig(post,\Box) : secu(\Box, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_4$	$\prec$	2	$\to_{\mathcal{C}}$
$secu(\Box, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 2 : post \leftarrow 2 : pre \leftarrow 1 : i \leftarrow 2 : n \leftarrow 3 : \bullet}_{L_5}$	$\prec$	$\bot$	
$secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$while(lt(i,n), B)$	$\to_{\mathcal{C}}$
$secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$if(lt(i,n), secu(B, while(...)))$	$\to_{\mathcal{C}}$
$if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$lt(i,n)$	$\to_{\mathcal{C}}$
$lt(\Box, n) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	i	$\to_{\mathcal{C}}$
$lt(\Box, n) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	2	$\to_{\mathcal{C}}$
$lt(2, \Box) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	n	$\to_{\mathcal{C}}$
$lt(2, \Box) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	3	$\to_{\mathcal{C}}$
$if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	$True$	$\to_{\mathcal{C}}$
$secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$secu(B, while(...)))$	$\to_{\mathcal{C}}$
$secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$secu(assig(fib, sum(pre,post), secu(...))$	$\to_{\mathcal{C}}$
$secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$assig(fib, sum(pre,post))$	$\to_{\mathcal{C}}$
$assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$sum(pre,post)$	$\to_{\mathcal{C}}$
$sum(\Box,post) : assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	pre	$\to_{\mathcal{C}}$
$sum(\Box,post) : assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	1	$\to_{\mathcal{C}}$
$sum(\Box,post) : assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	1	$\to_{\mathcal{C}}$
$sum(\Box,post) : assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	1	$\to_{\mathcal{C}}$
$sum(1,post) : assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\succ$	$post$	$\to_{\mathcal{C}}$
$sum(1,post) : assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	2	$\to_{\mathcal{C}}$
$assig(fib,\Box) : secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_5$	$\prec$	3	$\to_{\mathcal{C}}$
$secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 3 : post \leftarrow 2 : pre \leftarrow 1 : i \leftarrow 2 : n \leftarrow 3 : \bullet}_{L_6}$	$\prec$	$\bot$	$\to_{\mathcal{C}}$
$secu(\Box, secu(asig(pre,post), ...)) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_6$	$\prec$	$\bot$	$\to_{\mathcal{C}}$
$secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_6$	$\succ$	$secu(asig(pre,post), secu(asig(i, sum(i+1), ...)))$	$\to_{\mathcal{C}}$
$secu(\Box, secu(asig(i, sum(i+1), ...))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_6$	$\succ$	$asig(pre,post)$	$\to_{\mathcal{C}}$
$asig(pre,\Box) : secu(\Box, secu(asig(i, sum(i+1), ...))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_6$	$\succ$	$post$	$\to_{\mathcal{C}}$
$asig(pre,\Box) : secu(\Box, secu(asig(i, sum(i+1), ...))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_6$	$\prec$	2	$\to_{\mathcal{C}}$
$secu(\Box, secu(asig(i, sum(i+1), ...))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 3 : post \leftarrow 2 : pre \leftarrow 2 : i \leftarrow 2 : n \leftarrow 3 : \bullet}_{L_7}$	$\prec$	$\bot$	$\to_{\mathcal{C}}$
$secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7$	$\succ$	$secu(asig(i, sum(i,1), asig(post, fib)))$	$\to_{\mathcal{C}}$
$secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7$	$\succ$	$asig(i, sum(i,1))$	$\to_{\mathcal{C}}$

$$
\begin{array}{rclc}
asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \succ & sum(i,1) & \rightarrow_{\mathcal{C}} \\
asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \succ & sum(i,1) & \rightarrow_{\mathcal{C}} \\
asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \succ & sum(i,1) & \rightarrow_{\mathcal{C}} \\
sum(\Box,1) : asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \succ & 2 & \rightarrow_{\mathcal{C}} \\
sum(i,1) : asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \prec & 2 & \rightarrow_{\mathcal{C}} \\
sum(2,\Box) : asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \prec & 1 & \rightarrow_{\mathcal{C}} \\
asig(i,\Box) : secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_7 & \prec & 3 & \rightarrow_{\mathcal{C}} \\
secu(\Box, asig(post, fib))) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 3 : post \leftarrow 2 : pre \leftarrow 2 : i \leftarrow 3 : n \leftarrow 3 : \bullet}_{L_8} & \prec & \top & \rightarrow_{\mathcal{C}} \\
secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_8 & \succ & asig(post, fib) & \rightarrow_{\mathcal{C}} \\
asig(post,\Box) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_8 & \succ & fib & \rightarrow_{\mathcal{C}} \\
asig(post,\Box) : secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_8 & \prec & 3 & \rightarrow_{\mathcal{C}} \\
secu(\Box, while(...)) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid \underline{fib \leftarrow 3 : post \leftarrow 3 : pre \leftarrow 2 : i \leftarrow 3 : n \leftarrow 3 : \bullet}_{L_9} & \prec & \bot & \\
secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \succ & while(lt(i,n), B) & \rightarrow_{\mathcal{C}} \\
secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \succ & if(lt(i,n), secu(B, while(...))) & \rightarrow_{\mathcal{C}} \\
if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \succ & lt(i,n) & \rightarrow_{\mathcal{C}} \\
lt(\Box, n) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \succ & i & \rightarrow_{\mathcal{C}} \\
lt(\Box, n) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \prec & 3 & \rightarrow_{\mathcal{C}} \\
lt(i, \Box) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \succ & n & \rightarrow_{\mathcal{C}} \\
lt(i, \Box) : if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \prec & 3 & \rightarrow_{\mathcal{C}} \\
if(\Box, secu(B, while(...))) : secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \prec & false & \rightarrow_{\mathcal{C}} \\
secu(\Box, return(fib)) : vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \prec & \bot & \rightarrow_{\mathcal{C}} \\
vardec(Int, result, \Box) : \diamond \mid L_1\bigstar\bullet \mid L_9 & \succ & return(fib) & \rightarrow_{\mathcal{C}} \\
return(\Box) : vardec(Int, result, \Box) : \diamond \mid \bullet \mid \mathrm{L}_1 & \succ & fib & \rightarrow_{\mathcal{C}} \\
return(\Box) : vardec(Int, result, \Box) : \diamond \mid \bullet \mid \mathrm{L}_1 & \prec & 3 & \rightarrow_{\mathcal{C}} \\
vardec(Int, result, \Box) : \diamond \mid \bullet \mid L_1 & \prec & 3 & \rightarrow_{\mathcal{C}} \\
\diamond \mid \bullet \mid result \leftarrow 3 : L_1 & \prec & \bot &
\end{array}
$$

Observe que no regresamos el resultado al final de la evaluación si no que lo almacenamos en una de las variables que queda impresa en el contexto final.
Nuestra atención está no en el resultado que se obtiene en el último paso ($\bot$) mas bien el estado de la memoria.

2 Ejercicios para el lector

Ejercicio 2.1. Utilizando la sintaxis definida para TinyC proporciona la definición de un programa que dados dos enteros n y m revise que n es múltiplo de m.

Ejercicio 2.2. Utilizando la definición de la máquina C evalúa el ejercicio anterior con los valores 3 y 9.

Ejercicio 2.3. Utilizando la sintaxis definida para TinyC proporciona la definición de un programa que dado un número n revise su paridad.

Ejercicio 2.4. Utilizando la definición de la máquina C evalúa el ejercicio anterior con el valor 4

Ejercicio 2.5. Utilizando la sintaxis definida para TinyC proporciona la definición de un programa que dados dos números n y m nos regrese el resultado de n^m.

Ejercicio 2.6. Utilizando la definición de la máquina C evalúa el ejercicio anterior con los valores 3 y 2.

Ejercicio 2.7. Utilizando la sintaxis definida para TinyC proporciona la definición del algoritmo $Merge\ Sort$.

Ejercicio 2.8. Utilizando la definición de la máquina C evalúa el ejercicio anterior con el arreglo [7,5,1,3]

Ejercicio 2.9. Utilizando la sintaxis definida para TinyC proporciona la definición del algoritmo $Binary\ Search$.

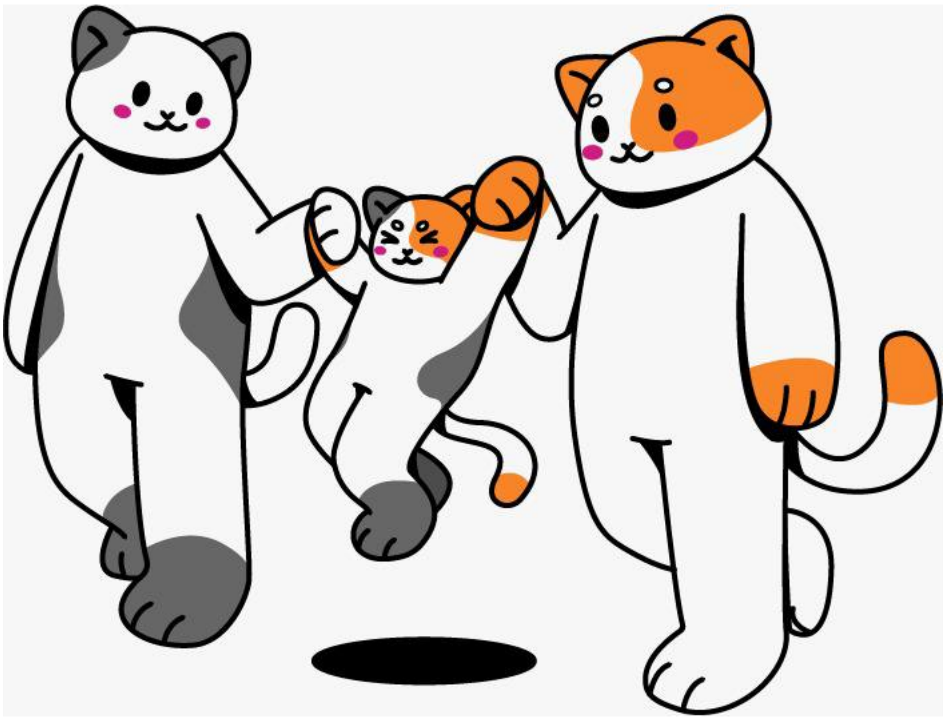
Ejercicio 2.10. Utilizando la definición de la máquina C evalúa el ejercicio anterior con el arreglo [8,4,1,2,7] y búsca el número 2.

Ejercicio 2.11. Utilizando la sintaxis definida para TinyC proporciona la definición de un programa que calcule el n-ésimo término de la sucesión de Cataluña.

Ejercicio 2.12. Utilizando la definición de la máquina C evalúa el ejercicio anterior con el número 7

Capítulo 10

Herencia y subtipos



En este capítulo vamos a revisar el último paradigma de programación que estudiaremos en este manual, la orientación a objetos y sus características más importantes; la herencia y los subtipos.

La introducción del paradigma de la orientación a objetos presenta una ventaja al momento de reutilizar código que ya hemos definido anteriormente y que durante los últimos años supuso una revolución en la forma en la que se escriben y desarrollan los programas que corren en nuestros teléfonos y computadoras personales.

Lenguajes como Java, Python, Ruby, Golang, Swift, C# entre muchos otros implementan la herencia, en donde métodos, constructores y variables de la clase padre puede ser reutilizadas por otra clase hija.

Más aún, en este tipo de lenguajes la flexibilidad no solo se extiende a hacer uso de código previamente definido y heredado si no que se puede extender y sobrecargar sobre lo ya existente, proporcionando una mecanismo modular para desarrollar software y evitar la redefinición de clases, métodos y variables.

Objetivo

El objetivo de este capítulo es proporcionar una definición de los conceptos principales de la orientación objetos y su aplicación a la herencia y el subtipificado en el paradigma procedimental para las diferentes categorías definidas en la sintaxis de TinyC.

Planteamiento

En este capítulo se presentan las definiciones principales del paradigma de la orientación a objetos como clase, objeto y subtipo, así como las reglas que nos permiten inferir el tipo de una expresión.

Finalmente se concluye el capítulo introduciendo el concepto de *casting* (en español conversión) junto con los juicios que nos permiten transformar el tipo de una expresión.

1 Orientación a objetos

La orientación a objetos tiene como fundamento la abstracción de los elementos mas importantes de las entidades que se desean modelar en la computadora, su identidad, su comportamiento, relaciones, etc. Para este propósito se utiliza una abstracción conocida como clase.

Definición 1.1. Definición para clases[a].

Una clase es una abstracción de una entidad, elemento u objeto que define sus características más importantes como el nombre, los atributos, variables y métodos que describen su comportamiento.

[a]Definición formulada de [128] y [133]

Definición 1.2. Definición de objeto[a].

Un objeto es una instancia de una clase.

[a]Definición formulada de [128] y [133]

Las características que se comparten entre los lenguajes que implementan este paradigma incluyen[1] :

[1]Definición formulada de [133]

- Herencia: permite la definición de una clase original dejando que esta sea extendida o sobrecargada por las clases que heredan de ella, constituyendo un mecanismo para reutilizar código previamente escrito y adaptarlo al contexto particular donde se desean instanciar los objetos.

- Encapsulamiento de datos: esta característica comprende la seguridad de la información contenida en las variables y métodos de una clase. Permitiendo que solo el objeto mismo sea el encargado de alterar la información contenida en el, proporcionando seguridad y consistencia.

- Polimorfismo: esta característica permite que un objeto que ha heredado de una clase pueda ser identificado por cualquiera de los tipos que posea la cadena de herencia, pudiendo este ser su mismo tipo, o cualquiera de los tipos de la clase o clases padres.

- Representaciones múltiples: esta característica permite que dos objetos de una misma clase puedan presentar comportamiento distinto pues éste puede cambiar según el estado que posea cada uno.

- Recursión abierta: un objeto es capaz de invocarse a si mismo dentro de la definición de los métodos de su propia clase, utilizando las palabras reservadas como `self` o `this`.

2 Subtipos

Los sistemas de tipos que hemos estudiado hasta este momento son restrictivos en el sentido de que muchas operaciones que parecerían intuitivas al desarrollar un programa no serían aceptadas por dicho sistema. Particularmente operaciones como la suma (+) entre los tipos *Int* y *Float* por ejemplo.

Necesitamos "relajar" este sistema para tener mas flexibilidad sobre este tipo de casos. En particular necesitamos construir la relación de subtipificado; A $\leq$ B que simboliza:"A es subtipo de B".

Definición 2.1. Relación de subtipificado[a]

Reflexividad

$$\frac{}{A \leq A}$$

Transitividad

$$\frac{A \leq B \quad B \leq C}{A \leq C}$$

Subsunción

$$\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B}$$

Esta propiedad expresa que si una expresión tiene tipo A y A es subtipo de B entonces puede usarse en cualquier contexto en donde sea necesaria una expresión de tipo B.

[a] Definición formulada de [130], [131] y [132]

Ejercicio 2.1. Para este ejemplo consideremos la expresión:

$$5{,}5 + 2$$

Donde podemos observar que $5{,}5 : Float$ y $2 : Int$, suponemos entonces que los tipos $Float$ e Int están relacionados bajo la relación de subtipificado como

$$Int \ \leq \ Float$$

por lo que la derivación de tipos de la expresión anterior queda como sigue:

$$\dfrac{\dfrac{}{\varnothing \vdash 5{,}5 : Float} \qquad \dfrac{\dfrac{}{\varnothing \vdash 3 : Int} \quad Int \leq Float}{\varnothing \vdash 3 : Float}}{\varnothing \vdash 5{,}5 + 3 : Float}$$

Gracias a la propiedad de subsunción podemos reemplazar Int por $Float$ en la derivación de los tipos conservando la propiedad de ser una expresión válida.

La relación de subtipificado puede ser aplicada a partir de dos interpretaciones diferentes[2]:

- Interpretación por subconjuntos: sí A $\leq$ B entonces toda expresión de tipo A también es una expresión de tipo B pues A está contenido en B.

- Interpretación por coerción: entendemos coerción como la acción de ejercer presión sobre un objeto para forzar su conducta. Esta acción aplicada al contexto de software nos permite modelar el siguiente comportamiento: si s es de tipo A y A $\leq$ B entonces s se puede convertir de forma única a una expresión de tipo B, es decir forzamos el cambio de tipo mediante una conversión explícita.

  En el ejemplo anterior teníamos $2 : Int$ el cuál fue coercido a $2.0 : Float$.

2.1 Subtipificado de tipos primitivos

Las reglas para definir la relación de subtipificado entre los tipos primitivos proporcionados en la definición de los lenguajes de programación, constituyen los axiomas de dicho sistema y con ellos se pude derivar el resto de las reglas utilizando las propiedades de subsunción y transitividad.

Para ilustrar este punto se agrega el tipo $Float$ junto con los axiomas:

Definición 2.2. Relación de subtipificado para Nat, Int y $Float$[a].

$$\dfrac{}{Nat \leq Int} \qquad\qquad \dfrac{}{Int \leq Float}$$

[a]Definición formulada de [130], [131] y [132]

[2]Definición formulada de [5], [12]

2.2 Subtipificado de funciones

Definamos el siguiente tipo de para una función f como $f : Int \to Int$, entonces se tiene que $f\,n : Int$ cuando recibe cualquier argumento de tipo $n : Int$, así $f\,n : Float$ por la regla de subsunción que revisamos en la sección anterior, por consiguiente se tiene que:

$$Int \to Int \leq Int \to Float$$

es una derivación de tipo válida. Es decir la relación de subtipificado original se preserva en el codominio de la función. En tal caso se dice que esta posición es covariante.

Por otro lado si definimos el tipo de nuestra función como $f : Float \to Int$. Tenemos que como todas las expresiones $e : Int$ son también de tipo $Float$ podemos utilizar la relación de subtipificado en sentido opuesto para restringir el dominio (de forma inversa a como se usó en el caso anterior), así podemos introducir elementos de tipo Int en el dominio de f. Por lo que se cumple la siguiente relación de subtipificado:

$$Float \to Int \leq Int \to Int$$

Es decir, la relación de subtipos original $Int \leq Float$ usada en la primera derivación se invierte para este caso, decimos entonces que este argumento es contravariante.

Finalmente, usando la propiedad de transitividad tenemos la siguiente relación de subtipificado

$$Float \to Int \leq Int \to Int \leq Int \to Float$$

De esta forma podemos definir la regla de subtipificado para el caso general de los tipos función como sigue.

Definición 2.3. Definición para subtipificación de funciones[a]:

$$\frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

[a]Definición formulada de [5], [12], [131] y [132]

2.3 Subtipificado para suma y producto

Para esta aplicación la relación de subtipificado se comporta como contravariante, es decir la dirección de la relación se preserva como es de esperar, de tal forma que se introducen las siguientes reglas al sistema de tipos.

Definición 2.4. Reglas para subtipificar expresiones aritméticas[a].

$$\frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 + S_2 \leq T_1 + T_2} \qquad\qquad \frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2}$$

[a]Definición formulada de [5], [12], [131] y [132]

2.4 Subtipificado para registros

Los registros como hemos estudiado en secciones anteriores son la generalización de una tupla permitiendo extender el concepto para n elementos. Asociados a éstos existen las proyecciones para recuperar el n-ésimo elemento contenido en estas.

Veremos brevemente las características asociadas al tipo de esta estructura una vez se introduce la relación de subtipificado.

Definición 2.5. Definición de la relación de subtipificado aplicado a los tipos registros[a].

Amplitud: dados dos registros A y B donde A posee una cantidad menor de elementos que B, decimos que el tipo del registro A es subtipo del registro B representado de la siguiente forma:

$$\frac{}{(l_1 : T_1, \ldots, l_{n+k} : T_{n+k}) \leq (l_1 : T_1, \ldots, l_n : T_n)}$$

Profundidad: esta propiedad es la aplicación de la contravarianza de la relación de subtipificado, aplicandose a cada uno de los campos contenidos en el registro:

$$\frac{T_1 \leq S_1 \quad \cdots \quad T_n \leq S_n}{(l_1 : T_1, \ldots, l_n : T_n) \leq (l_1 : S_1, \ldots, l_n : S_n)}$$

Permutación: El orden de los campos de un valor de tipo registro no importa.

$$\frac{(s_1 : S_1, \ldots, s_n : S_n) \text{ permutación de } (l_1 : T_1, \ldots, l_n : T_n)}{(s_1 : S_1, \ldots, s_n : S_n) \leq (l_1 : T_1, \ldots, l_n : T_n)}$$

[a] Definición formulada de [5], [12], [131] y [132]

2.5 Elementos máximos

En nuestra implementación para el sistema de tipos es importante contar con un elemento máximo, en donde elemento de tipo distinto al elemento máximo es subtipo de este. Aquí introducimos Top que cumple exactamente con esta característica representada de la siguiente forma:

Definición 2.6. Definición del elemento máximo para el sistema de subtipificado [a].

$$\frac{}{T \leq Top}$$

[a] Definición formulada de [5], [12]

La idea detrás de Top es que a este tipo pertenecen todos aquellos programas que están correctamente tipificado. En Java este tipo corresponde al tipo $Object$.

La idea de la existencia de un tipo que es subtipo de todos los demás también existe y se conoce como *Bot*, este tipo es un tipo inhabitado, ninguna expresión debe de existir dentro del mismo. Este tipo ayuda a expresar aquellas expresiones que no deben de regresar ningún valor similar a la forma en la que se utiliza el tipo *void* en Java.

La descripción anteriormente dada para *Bot* se representa de la siguiente forma:

Definición 2.7. Definición del elemento máximo para el sistema de subtipificado [a].

$$\frac{}{Bot \leq T}$$

[a]Definición formulada de [5], [12]

3 Conversión de tipos

En lenguajes de programación como Java hemos hecho uso de este mecanismo para cambiar el tipo de un dato o variable por otro.

Similar al ejemplo que tenemos para pasar de $2 : Int$ a $2.0 : Float$. La relación de subtipificado nos provee la regla de subsunción sin embargo definiremos el mecanismo para ambos sentidos de la relación conocidos como conversión ascendiente (*Upcasting*) y conversión descendiente (*Downcasting*).

3.1 Conversión ascendiente *(Upcasting)*

En este tipo de casting (también conocido como ”casting hacia arriba”) el objetivo es que al tipificar un término, a este se le atribuye un supertipo esperado.

De esta forma para los subtipos donde $T \leq R$ entonces usando subsunción podemos concluir que $\Gamma \vdash \langle R \rangle\, e : R$, en donde $\langle R \rangle$ es la notación para el *casting* explícito, representado como:

Definición 3.1. Definición para Conversión ascendiente[a]:

$$\frac{\Gamma \vdash e : T \qquad T \leq R}{\Gamma \vdash \langle R \rangle\, e : R}$$

[a]Definición formulada de [5], [12], [131] y [132]

3.2 Conversión descendiente *(Downcasting)*

En este tipo de casting (también conocido como ”casting hacía abajo”) la asignación de tipos es arbitraria, la relación de subtipificado no necesariamente se tiene que cumplir.

Ésto supone una amenaza a la consistencia de tipos en el programa que se escribe con esta instrucción, el lenguaje está forzado a aceptar el *downcasting* en tiempo de compilación pero en tiempo de ejecución un futuro conjunto de verificaciones deben de ser ejecutados para corroborar que el tipo asignado no supone un error. Esta regla se representa de la siguiente manera:

Definición 3.2. Definición para conversión descendiente[a]:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \langle R \rangle\, e : R}$$

[a]Definición formulada de [5], [12], [131] y [132]

4 Ejercicios para el lector

Ejercicio 4.1. Dada la siguiente expresión utiliza las reglas de tipificado para suma y producto, deriva el tipo de la misma.

$$(4{,}0 \ * \ 1) \ - \ 3{,}0 \ : \ \mathit{Float}$$

Ejercicio 4.2. Dada la siguiente expresión utiliza las reglas de tipificado para suma y producto, deriva el tipo de la misma.

$$(3 \ * \ 3{,}0) \ + \ (4 \ * \ 4) \ : \ \mathit{Float}$$

Ejercicio 4.3. Dada la siguiente expresión utiliza las reglas de tipificado para el operador *if*, deriva el tipo de la misma.

$$\mathsf{if}\ (7 \ * \ 0{,}0) < 0 \ \mathsf{then}\ 1 \ \mathsf{else}\ 0 : \ \mathit{Float}$$

Ejercicio 4.4. Dada la siguiente expresión utiliza las reglas de tipificado para el operador *while*, deriva el tipo de la misma.

$$\mathit{Int}\ i \ = \ 0; \ \mathit{while}(i < 10)\{i + 1;\} \ \mathsf{return}\ i; \ : \ \mathit{Int}$$

Ejercicio 4.5. Dada la siguiente expresión utiliza las reglas de tipificado para el operador *fun*, deriva el tipo de la misma.

$$fun(x : \mathit{Int} \rightarrow \ x + 1 \ ) : \mathit{Int} \rightarrow \mathit{Int}$$

Ejercicio 4.6. Utiliza la regla de *Upcasting* para obtener el tipo de la siguiente expresión basado en las reglas para tipos primitivos.

$$fun(x : \mathit{Int} \rightarrow if(x \leq 0 \ then \ (x - 1{,}0) \ else \ x)) : \mathit{Int} \rightarrow \mathit{Int}$$

Ejercicio 4.7. Utiliza las reglas de elementos máximos para obtener la derivación del tipificado para la siguiente expresión.

$$fun(x : \mathit{Int} \rightarrow if(x \leq 0 \ then \ (x - 1{,}0) \ else \ x)) : \mathit{Bot}$$

Ejercicio 4.8. Utiliza las reglas de elementos máximos para obtener la derivación del tipificado para la siguiente expresión.

$$fun(x : \mathit{Int} \rightarrow if(x \leq 0 \ then \ (x - 1{,}0) \ else \ x)) : \mathit{Top}$$

Capítulo 11

Java Peso Pluma

Bibliografía

[1] Ramírez Pulido K., Soto Romero M., Enríquez Mendoza J., *Nota de Clase del curso de Lenguajes de Programación*, Facultad de Ciencias, Universidad Nacional Autónoma de México, Ciudad de México, 2022.

[2] Brooks A., *Modern Programming Languages: A Practical Introduction* (2nd Edition). Franklin, Beedle & Associates, cop. Sherwood, Oregon, 2011.

[3] Deepika P., *Selection Sort* (2021), [Fecha de consulta: 14/11/2022]. Geeks for geeks. Disponible en https://www.geeksforgeeks.org/selection-sort/

[4] Lipovaca M., *Learn You a Haskell for Great Good!: A Beginner's Guide.* (digital publication). San Francisco, California. 2011.

[5] Miranda Perea F., González Huesca L., *Nota de Clase del curso de Lenguajes de Programación,* Facultad de Ciencias, Universidad Nacional Autónoma de México, Ciudad de México, 2021.

[6] 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, 2020.

[7] Nielson F., *Semantics with Applications: An Appetizer*, Springer Publishing, 2007.

[8] Harper R., *Practical Foundations for Programming Languages.* Working draft, Carnegie Mellon University Press, San Francisco, California, 2010. Disponible en https://moss.cs.iit.edu/cs440/readings/harper.pdf

[9] Mitchell J., *Foundations for Programming Languages,* Massachusetts Institute of Technology Press, Cambridge, Massachusetts, 1996.

[10] Krishnamurthi S., *Programming Languages Application and Interpretation,* Brown University press, Providence, Rhode Island, 2007.

[11] Spector-Zabusky A., *How would the Lambda Calculus add numbers?* 2021 [fecha de consulta: 16/4/2023.]. Stack Overflow. Disponible en https://stackoverflow.com/questions/29756732/how-would-the-lambda-calculus-add-numbers

[12] Enríquez Mendoza J., *Lenguajes de Programación Nota de clase.* Facultad de Ciencias, Universidad Nacional Autónoma de México, Ciudad de México, 2022.

[13] Keller. G., O'Connor-Davis. L., *Concepts of Programming Languages: Data types in Explicitly Typed Lenguages,* Department of Information and Computing Sciences, Utrecht University, The Netherlands, 2022.

[14] Karavirta V., Shaffer C., *Formal Languages Spring Chapter 1 Introduction Grammar Exercises* (online tool), [fecha de consulta: 7/10/2022]. Disponible en https://opendsa-server.cs.vt.edu/OpenDSA/Books/PIFLAS21/html/IntroGrammarEx.html

[15] JerrettDavis, *Binary Search, a haskell approach* (2022), [fecha de consulta: 29/11/2022]. Disponible en https://programming-idioms.org/idiom/124/binary-search-for-a-value-in-sorted-array/2120/haskell

[16] kirankumarambati, *Binary Search Data Structure and Algorithm Tutorials* (2023), [fecha de consulta: 29/11/2022]. Geeks for geeks. Disponible en https://www.geeksforgeeks.org/binary-search/

[17] Dahiya A., Wasserman Z., *CIS 194: Introduction to Haskell, Homework 1* (2013), [fecha de consulta: 4/11/2022]. University of Pennsylvania, Filadelfia, Pensilvania. Disponible en https://www.seas.upenn.edu/ cis1940/spring13/hw/01-intro.pdf

[18] Yorgey B., *Typeclassopedia* (2011), [fecha de consulta: 24/11/2022]. Wiki Haskell. Disponible en https://wiki.haskell.org/Typeclassopedia

[19] King J. K., *Haskell List Problem Set* (2018), [fecha de consulta: 10/12/22]. Github. Disponible en https://github.com/JD95/haskell-problem-sets/blob/master/Lists/Problems.hs

[20] Goguen J. A., *Semantics of computation. Category Theory Applied to Computation and Control. Lecture Notes in Computer Science.* Vol. 25. Springer, 1975.

[21] Floyd, R. W., *Assigning Meanings to Programs. In Schwartz, J.T. (ed.). Mathematical Aspects of Computer Science. Proceedings of Symposium on Applied Mathematics.* Vol. 19. American Mathematical Society, 1967.

[22] Winskel G. *The formal semantics of programming languages: an introduction*, Massachussetts Institute of Technology Press, Cambridge, Massachussetts, 1993.

[23] Schmidt, D. A., *Denotational Semantics: A Methodology for Language Development.* William C. Brown Publishers, 1986.

[24] Plotkin, G. D., *A structural approach to operational semantics* (Technical Report DAI-MI FN-19), Computer Science Department, Aarhus University, Denmark, 1981.

[25] Deransart, P., Jourdan M., Lorho B., *Attribute Grammars: Definitions, Systems and Bibliography* (Lecture Notes in Computer Science 323), Springer-Verlag, Berlin Heidelberg, 1988.

[26] Krishnamurthi S., *Programming Languages: Application and Interpretation* (2nd ed.), Brown University Press, Providence, Rhode Island, 2012.

[27] Slonneger K., Kurtz B. L., *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Publishing Co., United States, 1995.

[28] Colaboradores de Wikipedia, *Semantics (computer science)*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 18/10/2023]. Disponible en https://en.wikipedia.org/wiki/Semantics_(computer_science)

[29] Colaboradores de Wikipedia, *Syntax (programming languages)*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 18/10/2023]. Disponible en https://en.wikipedia.org/wiki/Syntax_(programming_languages)

[30] Friedman D. P., Mitchell W., Haynes C. T., *Essentials of Programming Languages* (1st ed.), The Massachusetts Insitute of Technology Press, Cambridge, Massachusetts, 1992.

[31] Smith D., *Designing Maintainable Software.* Springer Science & Business Media, United States, 1999.

[32] Aho A. V., Lam M. S., Seth R., Ullman J. D., *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley Publishing Co., United States, 2017

[33] Louden K. C., *Compiler Construction: Principles and Practice.* Brooks-Cole Publishers. United States, 1997.

[34] Sipser M., *Introduction to the Theory of Computation.* PWS Publishing Co., United States, 1997.

[35] Colaboradores de Wikipedia. *Pragmatics*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 19 10 2023]. Disponible en https://en.wikipedia.org/wiki/Pragmatics

[36] Coppock E., Champollion L., *Invitation to Formal Semantics (manuscript draft)*, eClass NKUA digital plataform, 2019.

[37] Mey J. L., *Pragmatics: An Introduction* (2nd ed.). Oxford-Blackwell Publishing, United Kingdom, 2001.

[38] Winter Y., *Flexibility principles in Boolean semantics.* Massachusetts Institute of Technology Press, Cambridge, Massachusetts, 2001.

[39] Felleisen M., Findler R. B., Flatt M., Krishnamurthi S., *How to Design Programs* (1st ed.), Massachusetts Institute of Technology Press, Cambridge, Massachusetts, 2003.

[40] Colaboradores de Wikipedia. *Ambiguous grammar*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 19/10/2023]. Disponible en https://en.wikipedia.org/wiki/Ambiguous_grammar

[41] Levelt W., *An Introduction to the Theory of Formal Languages and Automata.* John Benjamins Publishing. United States, 2008.

[42] Scott E., *SPPF-Style Parsing From Earley Recognizers* (Electronic Notes in Theoretical Computer Science). Elsevier B.V. Royal Holloway, University of London Egham, Surrey, United Kingdom, 2008.

[43] Colaboradores de Wikipedia. *Compiled Language.* Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 19 10 2023]. Disponible en https://en.wikipedia.org/wiki/Compiled_language

[44] Colaboradores de Wikipedia. *Interpreter (computing)*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 19 10 2023]. Disponible en https://en.wikipedia.org/wiki/Interpreter_(computing)

[45] Terence P., Luber J., *The Difference Between Compilers and Interpreters*, Wayback Machine Archive, United States, 2014.

[46] Colaboradores de IONOS Digital Guide, *Compilers vs. interpreters: explanation and differences*, IONOS Digital Guide (2023), [fecha de consulta: 26/11/2023]. Disponible en: https://www.ionos.com/digitalguide/websites/web-development/compilers-vs-interpreters

[47] Colaboradores de Wikipedia. *Object-oriented programming.* Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 19 10 2023]. Disponible en https://en.wikipedia.org/wiki/Object-oriented_programming

[48] Martin A., Cardelli L., *A Theory of Objects*, Springer Verlag. United States, 1998.

[49] Armstrong D. J., *The Quarks of Object-Oriented Development* (Communications of the ACM), Research Gate digital archive, 2006. Disponible en: https://www.researchgate.net/publication/220425366_The_quarks_of_object-oriented_development

[50] Colaboradores de Wikipedia. *Programación por procedimientos*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 19 10 2023]. Disponible en https://es.wikipedia.org/wiki/Programacion_por_procedimientos

[51] Colaboradores de Wikipedia. *Functional programming.* Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 23 10 2023]. Disponible en https://en.wikipedia.org/wiki/Functional_programming

[52] Hudak P., *Conception, evolution, and application of functional programming languages.* ACM Computing Surveys. Yale University, Department of Computer Science, New Haven, Connecticut, 1989.

[53] Jain A., *Javascript Promises: Is There a Better Approach?* Medium (2023), [fecha de consulta: 29/11/2023]. Disponible en: https://medium.datadriveninvestor.com/javascript-promises-is-there-a-better-approach

[54] Colaboradores de Wikipedia, *Imperative programming.* Wikipedia, La enciclopedia libre (2023). [fecha de consulta: 23/10/2023]. Disponible en https://en.wikipedia.org/wiki/Imperative_programming

[55] Colaboradores de IONOS Digital Guide, *Imperative programming: Overview of the oldest programming paradigm.* IONOS Digital Guide (2021), [fecha de consulta: 21/4/2022]. Disponible en: https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/

[56] Eckel B., *Thinking in Java*, Pearson Education Publishers. United States, 2006.

[57] Colaboradores de Wikipedia. *Logic programming*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 24/10/2023]. Disponible en https://en.wikipedia.org/wiki/Logic_programming

[58] Colaboradores de Wikipedia. *Mathematical object*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 25/10/2023]. Disponible en https://en.wikipedia.org/wiki/Mathematical_object

[59] Azzouni, J., *Metaphysical Myths, Mathematical Practice*, Cambridge University Press, United States, 1994.

[60] Burgess J., Rosen G., *A Subject with No Object*, Oxford University Press, United Kingdom, 1997.

[61] Colaboradores de Wikipedia. *Judgment (mathematical logic)*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 25/10/2023]. Disponible en https://en.wikipedia.org/wiki/Judgment_(mathematical_logic)

[62] Martin-Löf P., *On the meanings of the logical constants and the justifications of the logical laws.* Nordic Journal of Philosophical Logic, Department of Mathematics, University of Stockholm, Sweden, 1996.

[63] Colaboradores de Wikipedia. *Rule of inference*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 25/10/2023]. Disponible en https://en.wikipedia.org/wiki/Rule_of_inference

[64] Boolos G., Burgess J., Jeffrey R. C., *Computability and logic*, Cambridge University Press, United States, 2007.

[65] John C. R., *Theories of Programming Languages*, Cambridge University Press, United States, 2009.

[66] Bergmann M., *An introduction to many-valued and fuzzy logic: semantics, algebras, and derivation systems*, Cambridge University Press, United States, 2008.

[67] Miranda F., Viso E. G., *Matemáticas Discretas*, Facultad de Ciencias, Universidad Nacional Autónoma de México, Ciudad de México, 2016.

[68] Dossey J. A., Otto D. A., Spence L. E., Eynden C. V., *Discrete Mathematics* (5-th edition), Pearson-Addison-Wesley Publishing Co., Boston, United States, 2006.

[69] Gersting J. L., *Mathematical Structures for Computer Science* (3rd edition), Computer Science Press, W.H. Freeman and Company, United States, 1993.

[70] Grassman W. K., Tremblay J., *Logic and Discrete Mathematics, A computer Science Perspective*, Prentice-Hall Inc., United States, 1996.

[71] Gries D., Schneider F. B. *A Logical Approach to Discrete Mathematics*, Springer-Verlag, United States, 1994.

[72] Grossman J. W., *Discrete Mathematics, An introduction to concepts, methods and applications*, Macmillan Publishing Company, United States, 1990.

[73] Koshy T., *Discrete Mathematics with Applications*, Elsevier Academic Press, 2004.

[74] Rossen K. H., *Discrete Mathematics and its Applications* (6-th edition), McGraw Hill, 2006.

[75] Jessica S., *Introduction to Haskell* (2013), [fecha de consulta: 4/11/2022]. University of Pennsylvania, Filadelfia, Pennsylvania. Disponible en https://www.seas.upenn.edu/ cis1940

[76] Krahn, H., Rumpe B., Völke S., *Model Driven Engineering Languages and Systems.* Technische Universität Braunschweig, Braunschweig, Germany. 2007.

[77] Chomsky N. *Aspects of the Theory of Syntax*, Massachusetts Institute of Technology Press, Cambridge, Massachusetts, 2014.

[78] Colaboradores de Wikipedia, *Parse tree*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 30/10/2023]. Disponible en https://en.wikipedia.org/wiki/Parse_tree

[79] Colaboradores de Wikipedia, *Abstract syntax*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 30/10/2023]. Disponible en https://en.wikipedia.org/wiki/Abstract_syntax

[80] Colaboradores de Wikipedia, *Scope*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 31/10/2023]. Disponible en https://en.wikipedia.org/wiki/Scope_(computer_science)

[81] Colaboradores de Wikipedia, *Let expression*, Wikipedia, La enciclopedia libre (2023). [fecha de consulta: 31 10 2023]. Disponible en https://en.wikipedia.org/wiki/Let_expression

[82] Colaboradores de Wikipedia, *Lamabda Calculus*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 31 10 2023]. Disponible en https://en.wikipedia.org/wiki/Lambda_calculus

[83] Turing A., *Computability and λ-Definability*, The Journal of Symbolic Logic, United Kingdom, 1937.

[84] Thierry C., Zalta E. N.,*Type Theory*, The Stanford Encyclopedia of Philosophy, Department of Philosophy, Stanford University, United States, 2013.

[85] Mitchell J. C., *Concepts in Programming Languages*, Cambridge University Press. Cambridge, Massachusetts, United States, 2003.

[86] Pierce B. C., *Basic Category Theory for Computer Scientists*, The MIT Press, Cambridge, Massachusetts, 1991.

[87] Church A., *A set of postulates for the foundation of logic*, (Annals of Mathematics Archive), Mathematics Department, Princeton University Press, Princeton, Nueva Jersey, 1932.

[88] Selinger P., *Lecture Notes on the Lambda Calculus* (vol. 0804), Department of Mathematics and Statistics, University of Ottawa Press, Ottawa, Canada, 2018.

[89] Turbak F., Gifford D., *Design concepts in programming languages*, The MIT press, Cambridge, Massachusetts, 2008.

[90] Abrahams, P. W., *A final solution to the Dangling else of ALGOL 60 and related languages*, Communications of the ACM, Volume 9, Issue 9, 1986.

[91] Colaboradores de Wikipedia. *Mathematical induction*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 14 11 2023]. Disponible en https://en.wikipedia.org/wiki/Mathematical_induction

[92] DeVos M., *Mathematical Induction*, Simon Fraser University Press, British Columbia, Canada, 2023.

[93] Diaz G., *Mathematical Induction* (Wayback Machine Archive), Harvard University Press, Cambridge, Massachusetts, 2023.

[94] Colaboradores de Wikipedia. *Recursion*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 14 11 2023]. Disponible en https://en.wikipedia.org/wiki/Recursion

[95] Causey R. L., *Logic, sets, and recursion* (2nd ed.), Sudbury, Mass: Jones and Bartlett Publishers, New England, 2006.

[96] user207421., *Static Semantics meaning?* Stack Overflow (2016), [fecha de consulta: 14/11/2023], Disponible en: https://stackoverflow.com/questions/40430578/static-semantics-meaning

[97] Remer F., *Compiler Construction course*, University of California Press, Santz Cruz, 1979.

[98] Colaboradores de Wikipedia. *Dynamic syntax.* Wikipedia, La enciclopedia libre, 2023 [fecha de consulta: 28/11/2023]. Disponible en https://en.wikipedia.org/wiki/Dynamic_syntax

[99] Cann R., Kempson R., Lutz M., *The dynamics of language: an introduction*, Elsevier, Amsterdam, 2005.

[100] Colaboradores de Wikipedia. *Operational semantics*, Wikipedia, La enciclopedia libre (2023), [fecha de consulta: 28/11/2023]. Disponible en https://en.wikipedia.org/wiki/Operational_semantics

[101] Gilles K., *Natural Semantics*, Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, London, 1987.

[102] Myers A., *IMP: Big-Step and Small-Step Semantics*, Cornell University Online Repository, Ithaca, NY, Estados Unidos. 2007. [fecha de consulta 30/4/2024]. Disponible en https://www.cs.cornell.edu/courses/cs6110/2009sp/lectures/lec05-fa07.pdf

[103] Beckam M., *Big Step Semantics*, University of Illinois Online Repository, Champaign, IL, Estados Unidos. 2020. [fecha de consulta 30/4/2024]. Disponible en https://courses.engr.illinois.edu/cs421/sp2020/slides/04.1.2-big-step-semantics.pdf

[104] Pierce B., et al. *Programming Language Foundations*, The MIT Press, Cambridge, Massachusetts, Estados Unidos. 2021. [fecha de consulta 30/4/2024]. Disponible en https://softwarefoundations.cis.upenn.edu/plf-current/Smallstep.html

[105] Aldrich J., *Lecture Notes: Small-Step Operational Semantics*, Carnegie Mellon University Online Repositor, Pittsburgh, PA, Estados Unidos. 2022. [fecha de consulta 30/4/2024]. Disponible en https://www.cs.cmu.edu/ aldrich/courses/17-363/notes/lecture06-small-step.pdf

[106] Chong S., *Large-step semantics, continued*, Harvard Online Repository, Cambridge, Massachusetts, 2016. [fecha de consulta 30/4/2024]. Disponible en https://groups.seas.harvard.edu/courses/cs152/2016sp/lectures/lec04-largestep.pdf

[107] Keller. G. et al., *Concepts of Programming Language Design*, Utrecht University Online Repository, Utrecht, Países Bajos, 2022. [fecha de consulta 14/5/2024]. Disponible en https://github.com/jaeem006/Lenguajes/blob/main/Gabrielle_exc/Semantics_exercises_sol.pdf

[108] Korbmacher J. et al., *The Lambda Calculus*, Stanford Encyclopedia of Philosophy, California, Estados Unidos, 2023. [fecha de consulta 21/5/2024]. Disponible en https://plato.stanford.edu/entries/lambda-calculus/

[109] Rojas R., *A Tutorial Introduction to the Lambda Calculus*, Dallas University Online Repository, Texas, Estados Unidos, 2021. [fecha de consulta 21/5/2024]. Disponible en https://personal.utdallas.edu/ gupta/courses/apl/lambda.pdf

[110] Bonacci B., *Lambda Calculus Boolean logic*, Bruno Bonacci Blog, 2007. [fecha de consulta 21/5/2024]. Disponible en https://blog.brunobonacci.com/2017/10/08/lambda-calculus-and-boolean-logic/

[111] Cartwright R., *Comp 311 - Review 2*, Rice University Online Repository, Texas, Estados Unidos, 2010. [fecha de consulta 21/5/2024]. Disponible en https://www.cs.rice.edu/ javaplt/311/Readings/supplemental.pdf

[112] Chiang D., *Data structures in the lambda calculus*, Notre Dame Online Repository, Indiana, Estados Unidos, 2010. [fecha de consulta 21/5/2024]. Disponible en https://www3.nd.edu/ dchiang/teaching/pl/2019/church.html

[113] Sampson A., *Recursion and Fixed-Point Combinators*, Cornell University Online Repository, Nueva York, Estados Unidos, 2017. [fecha de consulta 21/5/2024]. Disponible en https://www.cs.cornell.edu/courses/cs6110/2017sp/lectures/lec05.pdf

[114] Richards G., *Proof of the Church-Rosser Theorem*, Waterloo University Online Repository, Ontario, Canadá, 2022. [fecha de consulta 4 6 2024]. Disponible en https://student.cs.uwaterloo.ca/ cs442/W22/extras/c-r-thm-proof.pdf

[115] Pohjola J., *COMP3161/9164 23T3 Assignment 1*, UNSW Online Repository, Sydney, Australia, 2023. [fecha de consulta 12/6/2024]. Disponible en https://www.cse.unsw.edu.au/ cs3161/23T3/Assignment %201/Spec.pdf

[116] Pohjola J., *Functional Programming Languages: MinHs*, UNSW Online Repository, Sydney, Australia, 2023. [fecha de consulta 12/6/2024]. Disponible en https://www.cse.unsw.edu.au/ cs3161/23T3/Week %2004/Tuesday/Slides.pdf

[117] Pfenning F., *Foundations of Programming Languages: Lectures Notes on Progress*, Carnegie-Mellon University Online Repository, Pittsburgh, Pensilvania, Estados Unidos, 2004. [fecha de consulta 18/6/2024]. Disponible en https://www.cs.cmu.edu/ fp/courses/15312-f04/handouts/07-progress.pdf

[118] Pfenning F., *Foundations of Programming Languages: Lectures Notes on Type Safety*, Carnegie-Mellon University Online Repository, Pittsburgh, Pensilvania, Estados Unidos, 2004. [fecha de consulta 18/6/2024]. Disponible en https://www.cs.cmu.edu/ fp/courses/15312-f04/handouts/06-safety.pdf

[119] Biernacka M. et al., *Non-Deterministic Abstract Machines.* CONCUR 2022 - 33rd International Conferenceff on Concurrency Theory, Varsovia, Polonia, 2022. [fecha de consulta 18/6/2024]. Disponible en https://inria.hal.science/hal-03772712/document

[120] Erwan A., *Self-application in Church's untyped lambda calculus*, 2012. Stack Overflow. [fecha de consulta 18/6/2024]. Disponible en https://math.stackexchange.com/questions/1316377/self-application-in-churchs-untyped-lambda-calculus

[121] Enriquez J. et al., *Notas para Lenguajes de Programación 2023-1: Boletín de ejercicios 4*, Facultad de Ciencias. Universidad Nacional Autónoma de México, Ciudad de México, 2023.

[122] Watt D., *Programming Language Desing Concepts*, John Wiley & Sons, Ltd. University of Glasgow, Glasgow, Scottland. 2004.

[123] Kozen D., *CS3110 Notes on Data Structures and Functional Programming: lecture 26: Type Inference and Unification*, Conrell University Online Repository. Cornell University, New York, United States, 2011.

[124] Ribeiro R. et al., *A Mechanized Textbook Proof of a Type Unification Algorithm*, Universidade Federal de Ouro Preto Online Repository, Universidade Federal de Ouro Preto, Minas Gerais, Brazil. 2015.

[125] Amaro M. et al., *A Mechanized Textbook Proof of a Type Unification Algorithm*, Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745 Vol. 27, Num. 3 (2020) 13-24.

[126] Feeley M., *Compiler for the Tiny-C language*, 2002. Université de Montreal Online Repository. [fecha de consulta 29/10/2024]. Disponible en https://www.iro.umontreal.ca/felipe/IFT2030-Automne2002/Complements/tinyc.c

[127] Slávik M., *TinyC Optimizing Compiler*, Faculty of Information Technology CTU. Prague. 2023.

[128] Igarashi A. Pierce B. Wadler P., *Featherweight Java: A Minimal Core Calculus for Java and GJ*, University of Tokyo, University of Pennsylvania. 2002.

[129] Weirich S. Huang L., *A Design for Type-Directed Programming in Java*, University of Pennsylvania. Estados Unidos. 2002.

[130] Colaboradores de Wikipedia. *Subtyping*, Wikipedia, La enciclopedia libre (2024), [fecha de consulta: 15/11/2024]. Disponible en https://en.wikipedia.org/wiki/Subtyping

[131] Meyers A., *Introduction to Compilers: Subtype Polymorphism*, Cornell University Online Repository. Nueva York. Estados Unidos. 2022.

[132] Silva A., *Advanced Programming Languages: Subtyping*, Cornell University Online Repository. Nueva York. Estados Unidos. 2023.

[133] Colaboradores de Wikipedia. *Object-oriented programming*, Wikipedia, La enciclopedia libre (2024), [fecha de consulta: 15/11/2024]. Disponible en https://en.wikipedia.org/wiki/Object-oriented_programming