Proyecto Final

Lenguajes de Programación

1 Marco Teórico

1.1 Recursión y Combinadores de Punto Fijo

La **recursión** es una técnica fundamental en programación funcional que permite a una función llamarse a sí misma para resolver problemas de manera iterativa o fractal. Esto es especialmente útil en cálculos como el factorial, la generación de series o el recorrido de estructuras de datos [1].

En lenguajes como MiniLisp, la recursión puede implementarse mediante combinadores de punto fijo, que permiten definir funciones recursivas sin asignarles un nombre explícito. Esto es relevante en entornos donde las funciones no tienen nombres asociados, como en la notación lambda pura [2].

Un **combinador de punto fijo** es una función de orden superior que, dada una función f, encuentra un punto fijo tal que $Y(f) = f(Y(f))$. En términos simples, permite que una función se refiera a sí misma durante su ejecución sin requerir un nombre explícito.

Uno de los combinadores más conocidos es el combinador Y, definido como:

$$Y = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

Este combinador puede utilizarse para definir funciones recursivas, como el cálculo del factorial:

$$\text{Factorial} = Y(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1))$$

Aquí, la función recursiva calcula el factorial de n invocando la misma función f con $n - 1$.

En el contexto de MiniLisp, los combinadores de punto fijo permiten extender la capacidad del lenguaje al introducir soporte para recursión. Esto elimina la necesidad de estructuras adicionales o referencias explícitas a funciones, simplificando el diseño y manteniendo el paradigma funcional puro [3].

La implementación de estos combinadores es clave para habilitar constructos avanzados, como el soporte para la definición de funciones anónimas recursivas.

1.2 Combinador de Punto Fijo Y

El **Combinador de Punto Fijo Y**, comúnmente conocido como **Combinador Y**, es una construcción fundamental en el cálculo lambda que permite la definición de funciones recursivas sin necesidad de nombrarlas explícitamente. Este combinador fue introducido por Haskell Curry y es esencial en la teoría de lenguajes de programación funcionales [4].

1.2.1 Definición Formal

El Combinador Y se define en notación lambda de la siguiente manera:

$$Y = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

Esta expresión permite que una función f se aplique a sí misma, facilitando la recursión en entornos donde las funciones no tienen nombres asignados.

1.2.2 Propiedad de Punto Fijo

Una función g tiene un punto fijo si existe un valor x tal que $g(x) = x$. El Combinador Y encuentra dicho punto fijo para una función dada, es decir, $Y(g) = g(Y(g))$. Esto es especialmente útil para definir funciones recursivas en el cálculo lambda, donde no se permite la auto-referencia directa.

1.2.3 Aplicación en Funciones Recursivas

El Combinador Y se utiliza para definir funciones recursivas como el factorial. Por ejemplo, la función factorial se puede expresar utilizando el Combinador Y de la siguiente manera:

$$\text{Factorial} = Y(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1))$$

Aquí, la función anónima toma como argumentos una función f y un número n. Si n es 0, devuelve 1; de lo contrario, multiplica n por la aplicación de f a $n-1$. El Combinador Y permite que f se refiera a sí misma, logrando la recursión necesaria para calcular el factorial.

1.3 Semántica Estática

La **semántica estática** en los lenguajes de programación se refiere al conjunto de reglas y restricciones que determinan la validez de un programa sin necesidad de ejecutarlo. Estas reglas se aplican durante la fase de compilación o análisis estático y aseguran que el código fuente cumpla con ciertas propiedades antes de su ejecución [5].

1.3.1 Aspectos Clave de la Semántica Estática

- **Verificación de Tipos**: Garantiza que las operaciones en el programa se realicen entre tipos de datos compatibles, previniendo errores en tiempo de ejecución. Por ejemplo, evitar la suma de un número entero con una cadena de texto.
- **Alcance de Variables**: Define la región del código donde una variable es accesible, evitando conflictos de nombres y asegurando que las variables se utilicen dentro de su contexto adecuado.
- **Reglas de Visibilidad**: Determina qué partes del programa pueden acceder a ciertas variables o funciones, estableciendo niveles de acceso y encapsulamiento.
- **Constantes y Literales**: Verifica que las constantes se asignen correctamente y que los literales se utilicen de manera coherente con su tipo declarado.
- **Declaraciones y Definiciones**: Asegura que todas las variables y funciones estén declaradas antes de su uso, evitando referencias a elementos inexistentes.

1.3.2 Importancia de la Semántica Estática

La aplicación de la semántica estática es fundamental para detectar errores potenciales en las primeras etapas del desarrollo, mejorando la confiabilidad y seguridad del software. Al identificar inconsistencias y violaciones de las reglas del lenguaje durante la compilación, se reducen los errores en tiempo de ejecución y se facilita el mantenimiento del código.

1.4 Inferencia de Tipos

La **inferencia de tipos** es una característica de ciertos lenguajes de programación que permite al compilador deducir automáticamente el tipo de una expresión sin necesidad de anotaciones explícitas por parte del programador. Este proceso facilita la escritura de código más conciso y legible, manteniendo al mismo tiempo la seguridad y coherencia tipológica del programa [6].

1.4.1 Funcionamiento de la Inferencia de Tipos

Durante la compilación, el sistema de inferencia analiza las expresiones y declaraciones en el código para determinar sus tipos. Por ejemplo, al evaluar la expresión `3 + 4`, el compilador infiere que tanto `3` como `4` son enteros, y por lo tanto, el resultado de la suma también es un entero. Este análisis se basa en las reglas del lenguaje y en el contexto en el que se utilizan las expresiones.

1.4.2 Ventajas de la Inferencia de Tipos

- **Código más conciso**: Al eliminar la necesidad de anotaciones de tipos explícitas, el código se vuelve más limpio y fácil de leer.
- **Detección temprana de errores**: El compilador puede identificar inconsistencias de tipos durante la compilación, reduciendo la posibilidad de errores en tiempo de ejecución.
- **Flexibilidad**: Permite escribir funciones genéricas que pueden operar sobre diferentes tipos sin necesidad de especificar cada uno de ellos explícitamente.

1.4.3 Lenguajes que Implementan Inferencia de Tipos

Varios lenguajes de programación incorporan inferencia de tipos en mayor o menor medida. Algunos ejemplos incluyen:

- **Haskell**: Utiliza un sistema de tipos fuerte y la inferencia de tipos es una característica central, permitiendo escribir código sin anotaciones de tipos explícitas en la mayoría de los casos.
- **ML y sus derivados (OCaml, F#)**: Estos lenguajes funcionales implementan la inferencia de tipos mediante el algoritmo Hindley-Milner, que permite deducir tipos de manera eficiente y segura.
- **Rust**: Aunque es un lenguaje de sistemas, Rust incorpora inferencia de tipos para mejorar la ergonomía del lenguaje sin sacrificar la seguridad.
- **Kotlin**: Ofrece inferencia de tipos en variables locales y en la definición de funciones, reduciendo la necesidad de anotaciones explícitas.

1.4.4 Algoritmo Hindley-Milner

Uno de los algoritmos más conocidos para la inferencia de tipos es el **Hindley-Milner**, desarrollado inicialmente para el lenguaje ML. Este algoritmo permite deducir tipos de manera eficiente y es la base de la inferencia de tipos en varios lenguajes funcionales. Se basa en la unificación de tipos y en la generación de un tipo más general que satisface todas las restricciones impuestas por el código [7].

1.4.5 Limitaciones de la Inferencia de Tipos

Aunque la inferencia de tipos ofrece múltiples ventajas, también presenta ciertas limitaciones:

- **Complejidad en casos avanzados**: En programas con estructuras de tipos complejas o uso intensivo de polimorfismo, el compilador puede requerir anotaciones explícitas para resolver ambigüedades.
- **Mensajes de error menos claros**: En ocasiones, los errores de tipos inferidos pueden generar mensajes menos intuitivos, dificultando la depuración.
- **Rendimiento de compilación**: La inferencia de tipos puede aumentar el tiempo de compilación, especialmente en proyectos grandes o con código altamente genérico.

2 Especificación Formal del Lenguaje

Sintaxis Concreta

⟨*expr*⟩ ::= ⟨*id*⟩
| ⟨*num*⟩
| ⟨*bool*⟩
| ⟨*string*⟩
| (⟨*op*⟩ ⟨*expr*⟩ {⟨*expr*⟩})
| (let ([⟨*id*⟩ ⟨*expr*⟩] {[⟨*id*⟩ ⟨*expr*⟩]} ⟨*expr*⟩)
| (let* ([⟨*id*⟩ ⟨*expr*⟩] {[⟨*id*⟩ ⟨*expr*⟩]} ⟨*expr*⟩)
| (if ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩)
| (cond ({[⟨*expr*⟩ ⟨*expr*⟩]}) (else ⟨*expr*⟩))
| (lambda (⟨*id*⟩ {⟨*id*⟩}) ⟨*expr*⟩)
| (⟨*expr*⟩ ⟨*expr*⟩ {⟨*expr*⟩})
| (letrec (⟨*id*⟩ ⟨*expr*⟩) ⟨*expr*⟩)
| (list {⟨*expr*⟩})

⟨*num*⟩ ::= ... 2.5 | -1 | 0 | 1 | 18.35 ...

⟨*bool*⟩ ::= #t | #f

⟨*string*⟩ ::= "a" | "b" | "Hello␣world!" | ...

⟨*op*⟩ ::= + | - | * | / | add1 | sub1 | sqrt | expt |
< | > | = | not | or | and |
head | tail | length | reverse | concat | map | filter |
sconcat | at | lstostr

⟨*id*⟩ ::= a | b | foo | ...

Sintaxis Abstracta Endulzada

$$Ops = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{add1}, \texttt{sub1}, \texttt{sqrt}, \texttt{expt}, \texttt{<}, \texttt{>}, \texttt{=}, \texttt{not}, \texttt{or}, \texttt{and}, \texttt{head}, \texttt{tail}, \texttt{length}, \texttt{reverse}, \texttt{concat}, \texttt{map}, \texttt{filter} \texttt{sconcat}, \texttt{at}, \texttt{lstostr}\}$$

$$\texttt{Binding} \subseteq \texttt{String} \times \texttt{SASA}$$

$$\frac{i\texttt{:String}}{IdS(i)\texttt{:SASA}} \qquad \frac{b\texttt{:[Binding]} \quad c\texttt{:SASA}}{Let1(b,\, c)\texttt{:SASA}} \qquad \frac{i\texttt{:String} \quad v\texttt{:SASA} \quad c\texttt{:SASA}}{Letrec(i,\, v,\, c)\texttt{:SASA}}$$

$$\frac{n \in \mathbb{Z}}{NumS(n)\texttt{:SASA}} \qquad \frac{c\texttt{:SASA} \quad t\texttt{:SASA} \quad e\texttt{:SASA}}{If(c,\, t,\, e)\texttt{:SASA}}$$

$$\frac{b \in \mathbb{B}}{BooleanS(b)\texttt{:SASA}} \qquad \frac{cs\texttt{:[(SASA, SASA)]} \quad e\texttt{:SASA}}{Cond(cs,\, e)\texttt{:SASA}} \qquad \frac{l\texttt{:[SASA]}}{List(l)\texttt{:SASA}}$$

$$\frac{f \in Ops \quad args\texttt{:[SASA]}}{Op(f,\, args)\texttt{:SASA}} \qquad \frac{p\texttt{:[String]} \quad c\texttt{:SASA}}{Fun(p,\, c)\texttt{:SASA}}$$

$$\frac{b\texttt{:[Binding]} \quad c\texttt{:SASA}}{Let(b,\, c)\texttt{:SASA}} \qquad \frac{f\texttt{:SASA} \quad a\texttt{:[SASA]}}{App(f,\, a)\texttt{:SASA}} \qquad \frac{s\texttt{:String}}{StringS(s)\texttt{:SASA}}$$

Sintaxis Abstracta Desendulzada

$$U = \{\texttt{add1}, \texttt{sub1}, \texttt{sqrt}, \texttt{not}, \texttt{head}, \texttt{lstostr}\}$$

$$B = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{expt}, \texttt{<}, \texttt{>}, \texttt{=}, \texttt{or}, \texttt{and}, \texttt{map}, \texttt{filter}, \texttt{sconcat}, \texttt{at}\}$$

$$U_{ns} = \{\texttt{tail}, \texttt{length}, \texttt{reverse}\}$$

$$B_{ns} = \{\texttt{concat}\}$$

$$\frac{i\texttt{:String}}{Id(i)\texttt{:ASA}} \qquad \frac{f \in B \quad i\texttt{:ASA} \quad d\texttt{:ASA}}{Binop(f,\, i,\, d)\texttt{:ASA}} \qquad \frac{l\texttt{:[ASA]}}{List(l)\texttt{:ASA}}$$

$$\frac{n \in \mathbb{Z}}{Num(n)\texttt{:ASA}} \qquad \frac{c\texttt{:ASA} \quad t\texttt{:ASA} \quad e\texttt{:ASA}}{If(c,\, t,\, e)\texttt{:ASA}} \qquad \frac{s\texttt{:String}}{String(s)\texttt{:ASA}}$$

$$\frac{b \in \mathbb{B}}{Boolean(b)\texttt{:ASA}} \qquad \frac{p\texttt{:String} \quad c\texttt{:ASA}}{Fun(p,\, c)\texttt{:ASA}}$$

$$\frac{f \in U \quad arg\texttt{:ASA}}{Unop(f,\, arg)\texttt{:ASA}} \qquad \frac{f\texttt{:ASA} \quad a\texttt{:ASA}}{App(f,\, a)\texttt{:ASA}}$$

Valores finales

$$\frac{n \in \mathbb{Z}}{NumV(n)\texttt{:Value}} \qquad \frac{p\texttt{:String} \quad c\texttt{:Value} \quad \varepsilon\texttt{:Env}}{\langle p,\, c,\, \varepsilon\rangle\texttt{:Value}} \qquad \frac{l\texttt{:[Value]}}{ListV(l)\texttt{:Value}}$$

$$\frac{b \in \mathbb{B}}{BooleanV(b)\texttt{:Value}} \qquad \frac{e\texttt{:ASA} \quad \varepsilon\texttt{:Env}}{\langle e,\, \varepsilon\rangle\texttt{:Value}} \qquad \frac{s\texttt{:String}}{StringV(s)\texttt{:Value}}$$

Semántica Natural

Identificadores se buscan en el ambiente y se retorna error de variable libre si no son encontrados

$$\frac{}{Id(i), \varepsilon \Rightarrow \varepsilon(i)}$$

Numeros se reducen a si mismos

$$\frac{}{Num(n), \varepsilon \Rightarrow NumV(n)}$$

Booleanos se reducen a si mismos

$$\frac{}{Boolean(b), \varepsilon \Rightarrow BooleanV(b)}$$

Listas se reducen a si mismas

$$\frac{}{List(l), \varepsilon \Rightarrow ListV(l)}$$

Strings se reducen a si mismas

$$\frac{}{String(s), \varepsilon \Rightarrow StringV(s)}$$

Las operaciones unarias y binarias que requieren puntos estrictos (U, B) se interpretan mediante la operación correspondiente

$$\frac{elige(f) = g \qquad arg, \varepsilon \Rightarrow a \qquad strict(a) = v' \qquad g(v') = v''}{Unop(f,\ arg), \varepsilon \Rightarrow v''}$$

$$\frac{elige(f) = g \qquad i, \varepsilon \Rightarrow i' \quad d, \varepsilon \Rightarrow d' \qquad strict(i') = i'' \quad strict(d') = d'' \qquad g(i'', d'') = v}{Binop(f, i, d), \varepsilon \Rightarrow v}$$

Donde *elige* es una función que transforma la operación en sintaxis abstracta a la operación correspondiente en el lenguaje anfitrión

Las operaciones unarias y binarias que **no** requieren puntos estrictos (U_{ns}, B_{ns}) se interpretan mediante la operación correspondiente

$$\frac{elige(f) = g \qquad arg, \varepsilon \Rightarrow a \qquad g(a) = v''}{Unop_{ns}(f,\ arg), \varepsilon \Rightarrow v''}$$

$$\frac{elige(f) = g \qquad i, \varepsilon \Rightarrow i' \quad d, \varepsilon \Rightarrow d' \qquad g(i', d') = v}{Binop_{ns}(f, i, d), \varepsilon \Rightarrow v}$$

Donde *elige* es una función que transforma la operación en sintaxis abstracta a la operación correspondiente en el lenguaje anfitrión

Condicional `if`

$$\frac{c, \varepsilon \Rightarrow c' \qquad strict(c') = Boolean(True) \qquad t, \varepsilon \Rightarrow t'}{If(c, t, e), \varepsilon \Rightarrow t'}$$

$$\frac{c, \varepsilon \Rightarrow c' \qquad strict(c') = Boolean(False) \qquad e, \varepsilon \Rightarrow e'}{If(c, t, e), \varepsilon \Rightarrow e'}$$

Expresiones `lambda`

$$\frac{}{Fun(p, c), \varepsilon \Rightarrow \langle p, c, \varepsilon \rangle}$$

Aplicaciones de función

$$\frac{f, \varepsilon \Rightarrow f' \qquad strict(f') = \langle p, c, \varepsilon' \rangle \qquad c, \varepsilon'[p \leftarrow \langle a, \varepsilon \rangle] \Rightarrow c_v}{App(f, a), \varepsilon \Rightarrow c_v}$$

Notemos que el `letrec` se convertirá en una aplicación de función con el combinador Y

```
(letrec (ft (lambda (n) (if (= n 0) 1 (* n (ft (- n 1)))))) (ft 5)) ⇒
((lambda (ft) (ft 5)) (Y (lambda (ft) (lambda (n) (if (= n 0) 1 (* n (ft (- n 1))))))))
```

Lo cuál es igual a

```
((Y (lambda (ft) (lambda (n) (if (= n 0) 1 (* n (ft (- n 1))))))) 5)
```

$ft = \lambda ft.\lambda n.$ if $n = 0$ then 1 else $n \cdot ft(n-1)$

$(Y\ ft)\ 5$

3 Algoritmo de Inferencia de Tipos

El **Algoritmo de Inferencia de Tipos** es un proceso que permite deducir el tipo de las expresiones en un programa sin necesidad de anotaciones explícitas. Uno de los algoritmos más utilizados es el **Algoritmo W**, asociado al sistema de tipos Hindley-Milner [8].

3.1 Descripción del Algoritmo W

El Algoritmo W funciona recorriendo el árbol sintáctico abstracto del programa y generando un conjunto de ecuaciones de tipos (también conocidas como restricciones de tipos). Luego, resuelve estas ecuaciones mediante unificación, encontrando el tipo más general que satisface todas las restricciones [5].

3.2 Restricciones

Identificadores
Igualar todas las apariciones
$[\![x_1]\!] = [\![x_2]\!] = \cdots = [\![x_n]\!]$
Que en realidad genera
$[\![x_1]\!] = [\![x_2]\!]$
...
$[\![x_1]\!] = [\![x_n]\!]$

Números
$[\![n]\!] =$ `number`

Booleanos
$[\![b]\!] =$ `boolean`

Strings
$[\![s]\!] =$ `string`

Listas
$[\![l]\!] =$ `list`

Operaciones aritméticas
$op = \{$`add1`, `sub1`, `sqrt`, `+`, `-`, `*`, `/`, `expt`$\}$
$[\![$ `(`$op\ n_1\ \ldots\ n_i$`)` $]\!] =$ `number`
$[\![n_1]\!] =$ `number`
...
$[\![n_i]\!] =$ `number`

Comparaciones
$op = \{$`<`, `>`, `=`$\}$
$[\![$ `(`$op\ n_1\ \ldots\ n_i$`)` $]\!] =$ `boolean`
$[\![n_1]\!] =$ `number`
...
$[\![n_i]\!] =$ `number`

Operaciones booleanas
$op = \{$`not`, `or`, `and`$\}$
$[\![$ `(`$op\ b_1\ \ldots\ b_n$`)` $]\!] =$ `boolean`
$[\![b_1]\!] =$ `boolean`
...
$[\![b_n]\!] =$ `boolean`

If
$[\![$ `(if` $c\ t\ e$`)` $]\!] = [\![t]\!]$
$[\![$ `(if` $c\ t\ e$`)` $]\!] = [\![e]\!]$
$[\![c]\!] =$ `boolean`
$[\![t]\!] = [\![e]\!]$

Cond

$[\![\texttt{(cond ([} c_1\ t_1\texttt{] ... [} c_n\ t_n\texttt{]) (else } e\texttt{))}]\!] = [\![t_1]\!]$

...

$[\![\texttt{(cond ([} c_1\ t_1\texttt{] ... [} c_n\ t_n\texttt{]) (else } e\texttt{))}]\!] = [\![t_n]\!]$

$[\![\texttt{(cond ([} c_1\ t_1\texttt{] ... [} c_n\ t_n\texttt{]) (else } e\texttt{))}]\!] = [\![e]\!]$

$[\![c_1]\!] = \texttt{boolean}$

...

$[\![c_n]\!] = \texttt{boolean}$

$[\![t_1]\!] = [\![t_2]\!]$

...

$[\![t_1]\!] = [\![t_n]\!]$

$[\![t_1]\!] = [\![e]\!]$

Let

$[\![\texttt{(let ([} i_1\ v_1\texttt{] ... [} i_n\ v_n\texttt{]) } c\texttt{)}]\!] = [\![c]\!]$

$[\![i_1]\!] = [\![v_1]\!]$

...

$[\![i_n]\!] = [\![v_n]\!]$

Let*

$[\![\texttt{(let* ([} i_1\ v_1\texttt{] ... [} i_n\ v_n\texttt{]) } c\texttt{)}]\!] = [\![c]\!]$

$[\![i_1]\!] = [\![v_1]\!]$

...

$[\![i_n]\!] = [\![v_n]\!]$

Funciones

$[\![\texttt{(lambda (} p_1\ \ldots\ p_n\texttt{) } b\texttt{)}]\!] = [\![p_1]\!] \to \cdots \to [\![p_n]\!] \to [\![b]\!]$

Aplicaciones de función $\texttt{(}f\ \ a_1\ \ldots\ a_n\texttt{)}$

$[\![f]\!] = [\![a_1]\!] \to \cdots \to [\![a_n]\!] \to [\![\texttt{(}f\ \ a_1\ \ldots\ a_n\texttt{)}]\!]$

Concatenación de Strings

$[\![\texttt{(sconcat } s_1\ \ldots\ s_n\texttt{)}]\!] = \texttt{string}$

$[\![s_1]\!] = \texttt{string}$

...

$[\![s_n]\!] = \texttt{string}$

At (Obtener cadena en un índice)

$[\![\texttt{(at } n\ s\texttt{)}]\!] = \texttt{string}$

$[\![n]\!] = \texttt{number}$

$[\![s]\!] = \texttt{string}$

Head

$[\![\texttt{(head } l\texttt{)}]\!] = T_{uuid}$

$[\![l]\!] = \texttt{list}$

Tail

$[\![\texttt{(tail } l\texttt{)}]\!] = \texttt{list}$

$[\![l]\!] = \texttt{list}$

Length

$[\![\texttt{(length } l\texttt{)}]\!] = \texttt{number}$

$[\![l]\!] = \texttt{list}$

Reverse

$[\![\texttt{(reverse } l\texttt{)}]\!] = \texttt{list}$

$[\![l]\!] = \texttt{list}$

Concatenación

$[\![\texttt{(concat } l_1\ l_2\texttt{)}]\!] = \texttt{list}$

$[\![l_1]\!] = \texttt{list}$

$[\![l_2]\!] = \texttt{list}$

Lista a String
$[\![\texttt{(lstostr } l\texttt{)}]\!] = \texttt{string}$
$[\![l]\!] = \texttt{list}$

Filter
$[\![\texttt{(filter } f\ l\texttt{)}]\!] = \texttt{list}$
$[\![f = \texttt{(lambda (}p\texttt{) } b\texttt{)}]\!] = [\![p]\!] \to [\![b]\!]$
$[\![b]\!] = \texttt{boolean}$
$[\![l]\!] = \texttt{list}$

Map
$[\![\texttt{(map } f\ l\texttt{)}]\!] = \texttt{list}$
$[\![f = \texttt{(lambda (}p\texttt{) } b\texttt{)}]\!] = [\![p]\!] \to [\![b]\!]$
$[\![l]\!] = \texttt{list}$

3.3 Detalles del Algoritmo

El algoritmo se basa en los siguientes componentes clave:

- **Variables de Tipo**: Representan tipos desconocidos que se irán determinando a lo largo del proceso.
- **Entorno de Tipos**: Una asociación entre identificadores y sus tipos correspondientes.
- **Unificación**: Un proceso para determinar si dos tipos pueden hacerse iguales mediante la sustitución de variables de tipo.
- **Generalización y Especialización**: La capacidad de crear tipos polimórficos generales o instancias específicas de esos tipos.

3.4 Pasos del Algoritmo

El Algoritmo W se puede describir mediante los siguientes pasos:

1. **Inicialización**: Comienza con un entorno de tipos vacío y sin restricciones.
2. **Análisis de Expresiones**: Para cada expresión en el programa, se realiza:
   (a) **Generación de Variables de Tipo**: Se asignan variables de tipo frescas a las expresiones cuya tipología es aún desconocida.
   (b) **Generación de Restricciones**: Se generan restricciones basadas en cómo interactúan las expresiones, según las reglas del lenguaje.
   (c) **Unificación**: Se aplican las restricciones mediante unificación, encontrando sustituciones que resuelven las variables de tipo.
   (d) **Actualización del Entorno**: Se actualiza el entorno de tipos con la nueva información obtenida.
3. **Generalización**: Al finalizar el análisis de una función o expresión, se generalizan los tipos libres, permitiendo polimorfismo.
4. **Asignación de Tipos**: Se asignan los tipos inferidos a las variables y expresiones correspondientes, completando el proceso.

3.5 Ejemplo de Inferencia de Tipos

Considere la siguiente función en MiniLisp:

```
(lambda (x) (+ x 1))
```

El proceso de inferencia sería:

- Asignar una variable de tipo α a x.
- La expresión $(+\, x\, 1)$ requiere que x sea de tipo numérico, ya que $+$ opera sobre números.

- Se genera la restricción $\alpha = \text{Num}$.
- Tras la unificación, se determina que x es de tipo Num.
- La función completa tiene el tipo $\text{Num} \rightarrow \text{Num}$.

3.6 Implementación en MiniLisp

Para implementar el Algoritmo W en MiniLisp, se deben seguir los siguientes pasos:

1. **Extender el Analizador Sintáctico**: Modificar el parser para que, además de construir el árbol sintáctico abstracto, también recopile información necesaria para la inferencia de tipos.
2. **Crear Estructuras para Tipos y Restricciones**: Definir estructuras de datos que representen variables de tipo, tipos concretos, y las restricciones entre ellos.
3. **Implementar el Motor de Unificación**: Escribir funciones que permitan unificar tipos, resolviendo las restricciones y detectando posibles conflictos o errores de tipos.
4. **Integrar el Algoritmo en el Proceso de Compilación**: Asegurar que antes de la evaluación o generación de código, se realice la inferencia de tipos y se verifiquen las restricciones.
5. **Manejo de Errores**: Diseñar mecanismos para informar al usuario sobre errores de tipos de manera clara y útil.

3.7 Desafíos en la Implementación

Algunos de los desafíos al implementar el Algoritmo W incluyen:

- **Manejo de Polimorfismo**: Asegurar que la generalización y especialización de tipos se realice correctamente, evitando pérdidas de generalidad o errores de tipado.
- **Eficiencia**: Optimizar el proceso de unificación y gestión de restricciones para que sea eficiente incluso en programas grandes.
- **Extensibilidad**: Diseñar el sistema de inferencia de tipos de manera que pueda extenderse para soportar nuevos constructos del lenguaje o sistemas de tipos más avanzados, como tipos algebraicos o de datos.
- **Interfaz con el Usuario**: Proporcionar mensajes de error y advertencias que sean comprensibles para el programador, facilitando la corrección de errores de tipos.

3.8 Referencias Clave

Para una comprensión más profunda del Algoritmo W y su implementación, se recomiendan las siguientes referencias:

- Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 348-375.
- Hindley, J.R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 29-60.
- Pierce, B.C. (2002). *Types and Programming Languages*. MIT Press.

4 Desafíos Encontrados

Durante el desarrollo del proyecto, se presentaron diversos desafíos relacionados con la implementación de la inferencia de tipos y el manejo de la recursión mediante combinadores de punto fijo. Uno de los principales retos fue integrar el Algoritmo W en el intérprete de MiniLisp, asegurando la correcta unificación de tipos y la detección de errores tipológicos.

Otro desafío significativo fue la optimización de la ejecución de funciones recursivas, especialmente en el manejo de la memoria y la prevención de desbordamientos de pila. Esto requirió implementar técnicas como la recursión de cola y analizar la eficiencia de los combinadores utilizados.

5 Trabajo a Futuro

Como continuación de este proyecto, se propone:

- **Extender el sistema de tipos**: Incorporar tipos más avanzados, como tipos algebraicos de datos y tipos dependientes, para aumentar la expresividad del lenguaje.
- **Mejorar la optimización**: Implementar un compilador que genere código optimizado, reduciendo la sobrecarga de los combinadores de punto fijo y mejorando el rendimiento en tiempo de ejecución.
- **Añadir soporte para paralelismo**: Explorar la posibilidad de ejecutar funciones en paralelo, aprovechando la naturaleza pura de las funciones en programación funcional y mejorando la eficiencia en sistemas multicore.
- **Desarrollar una interfaz de usuario**: Crear un entorno de desarrollo integrado (IDE) que facilite la escritura y depuración de código en MiniLisp, incluyendo resaltado de sintaxis y sugerencias de autocompletado.

6 Referencias

[1] Bird, Richard, and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1998.

[2] Barendregt, Henk P. *The Lambda Calculus: Its Syntax and Semantics*. Vol. 103. Elsevier, 1984.

[3] Hutton, Graham. *Programming in Haskell*. Cambridge University Press, 2016.

[4] Curry, Haskell B., and Robert Feys. *Combinatory Logic*. Vol. 1. North-Holland Publishing Company, 1958.

[5] Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.

[6] Milner, Robin. "A theory of type polymorphism in programming." *Journal of Computer and System Sciences* 17.3 (1978): 348-375.

[7] Hindley, J. Roger. "The principal type-scheme of an object in combinatory logic." *Transactions of the American Mathematical Society* 146 (1969): 29-60.

[8] Damas, Luis, and Robin Milner. "Principal type-schemes for functional programs." *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1982.