UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

ANÁLISIS, DISEÑO E IMPLEMENTACIÓN DE UN INTÉRPRETE CON TIPOS EXPLÍCITOS EN RACKET

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA**:**

OMAR ARENAS SILVA

DIRECTORA DE TESIS:

M. EN I. KARLA RAMÍREZ PULIDO

2016

CIUDAD UNIVERSITARIA, CD.MX.

Hoja de Datos del Jurado

1. Datos del alumno.
Arenas
Silva
Omar
56 16 81 00 30
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
304293986

2. Datos del tutor.
M. en I. Karla
Ramírez
Pulido

3. Datos del sinodal 1.
Dr. Favio Ezequiel
Miranda
Perea

4. Datos del sinodal 2.
Dra. Lourdes del Carmen
González
Huesca

5. Datos del sinodal 3.
Dr. Francisco
Hernández
Quiroz

6. Datos del sinodal 4.
M. en C. Araceli Liliana
Reyes
Cabello

7. Datos del trabajo escrito.
Análisis, diseño e implementación de un intérprete con tipos explícitos en Racket.
122 p.
2016

Agradecimientos

A mi madre Silvia Silva Huerta y a mi padre Rubén Arenas por ser mi razón de existir, porque conocerlos ha sido el logro más importante de mi vida. No existen palabras para agradecer la luz que han dado a mi camino. Algún día deseo tener la madurez, energía, humildad y esperanza que ustedes han combinado para ser exitosos. Los amo con toda mi alma.

A mi hermano Rubén Arenas Silva y a mi hermana Ivonne Arenas Silva porque nos amamos, respetamos y apoyamos. Han sido un modelo de vida para mi. Los amo.

A mis amigos por su sinceridad y apoyo, valoro mucho que digamos siempre lo que pensamos y sentimos, aunque no necesariamente coincida con lo que queremos oir. Cada una de esos días que hemos reído y llorado juntos es un día que agradezco estar vivo. Los amo.

Índice general

Introducción

A partir de la década de los años cincuenta, en el área de Ciencias de la Computación los primeros sistemas de tipos fueron usados para hacer comparaciones simples entre números enteros y representaciones de punto flotante. A finales de los años cincuenta y principios de los años sesenta la clasificación se extendió a datos estructurados como listas y arreglos, además de la introducción de funciones de primera clase. En los años setenta aparecieron gran cantidad de conceptos como tipos de datos abstractos, polimorfismo paramétrico, subtipificación, módulos, entre muchos otros, en consecuencia los sistemas de tipos representaron un campo de estudio. Mientras esto sucedió los científicos en computación notaron la conexión entre los sistemas de tipos encontrados en lenguajes de programación con aquellos presentados en el área de lógica matemática, conexión estudiada y desarrollada en el presente [13].

El estudio de los sistemas de tipos ha proporcionado a la evolución de los lenguajes de programación dos corrientes dentro del análisis de tipos, una de ellas consiste en realizar el análisis de tipos en tiempo de compilación, lenguajes pertenecientes a ésta corriente presentan a los tipos como parte de la sintaxis del programa y son conocidos como *lenguajes tipificados explícitamente*. Mientras la otra corriente lleva a cabo el análisis de tipos en tiempo de ejecución, en este caso los tipos no son parte necesariamente de la sintaxis de un programa y los lenguajes pertenecientes a esta categoría son conocidos como *lenguajes tipificados implícitamente* [13].

Las ventajas y desventajas de cada corriente han sido largamente debatidas en propiedades como la eficiencia en la corrección de errores de tipo, así como la expresividad y la detección de la mayor cantidad de errores entre otras. Los escenarios de aplicación a los que se ha dirigido esta discusión presentan dos vertientes, por un lado el mundo de la academia y por otro lado el mundo de la industria del software. Mientras los lenguajes con tipos explícitos como `C`, `C++` y `Java` han dominado el mercado de software desde hace muchos años, lenguajes con tipos implícitos como `Ruby` o `JavaScript` están ganando cada vez más terreno sobre todo en el desarrollo web, mientras `Racket` tiene un gran impacto en la investigación. En este proyecto, se pone en perspectiva la evolución de los lenguajes de programación de forma cronológica, de igual forma se exponen sus principales características destacando su análisis de tipos [1].

Para poner en contexto el desarrollo de este trabajo debemos considerar que la implementación de programas en un lenguaje tipificado explícitamente ayuda al programador a evitar errores antes de la ejecución del código, incluso obliga al programador a proporcionar cierta documentación que facilita el mantenimiento del programa. En pocas palabras, el sistema de tipos de estos lenguajes de programación impone una disciplina en el proceso de programación. Sin embargo, muchos programadores eligen lenguajes tipificados implícitamente para llevar a cabo programas que interactúan con otros sistemas que cambian de forma impredecible (la integración y aplicación de datos) o para el desarrollo de prototipos [3].

Se implementará un intérprete llamado `MinTyRacket`, el cual traduce un lenguaje tipificado explícitamente, implementado en `Racket`, un lenguaje perteneciente al paradigma funcional tipificado implícitamente. `MinTyRacket` será diseñado, implementado y analizado de manera que podrá ser utilizado en algún curso de Lenguajes de Programación a nivel Licenciatura o Maestría.

Se ha elegido el paradigma de programación funcional debido a su creciente popularidad en los últimos años, desde sus inicios en `Lisp` hasta llegar al día de hoy a través de sus lenguajes de propósito múltiple como `Haskell`, `Racket` y `Standard ML`, entre otros. La causa principal de que se usen masivamente es que poseen una facilidad de interacción que no sacrifica elegancia semántica, propiedad que los convierte en componentes de sistemas cada vez mas amplios. La gran expresividad y adaptación en tiempo de ejecución de este paradigma motiva el diseño de este lenguaje de programación.

La teoría requerida para cumplir el propósito de este proyecto nos indica que el corazón de este trabajo es el desarrollo de notaciones y definiciones precisas cuyo fin es poder demostrar algunas propiedades formales acerca de la *sintaxis* y *semántica* de `MinTyRacket`. Se utilizará la técnica usual de definición de una gramática libre de contexto en forma de *forma de Backus-Naur* para denotar los símbolos y cadenas que conforman un programa interpretado por `MinTyRacket`. Se describirán con precisión las transiciones de evaluación, las cuales describirán el comportamiento que una computadora seguirá al ejecutar un programa a través de `MinTyRacket`. Para el análisis matemático se utilizará una representación abstracta que permitirá utilizar el método de inducción para formalizar la demostración de algunas propiedades fundamentales de un lenguaje de programación. Se diseñará un sistema de tipos estático, asociando tipos a los términos del lenguaje definidos antes de la ejecución y basados en tales tipos se definirán las reglas de tipificación. Posteriormente veremos la fundamentación de la propiedad llamada *seguridad* que establece que ciertos errores no suceden durante la ejecución de un programa, por ejemplo, que una constante booleana sea sumada con un entero.

Basados en el diseño formal de nuestro intérprete se hará una implemen-

tación del mismo como un programa hecho en `Racket`, un lenguaje funcional descendiente de `Scheme`, a partir de la cual, se describirán algunas propiedades que permitirán comparar el intérprete de `MinTyRacket` con el intérprete de `Racket`.

Capítulo 1

Historia de los lenguajes de programación

Los primeros lenguajes de programación fueron los lenguajes de máquina diseñados para las primeras computadoras construidas en la década de 1940. Por lo que a principios de la década de 1950, la programación poseía una densidad de información baja, es decir, cada proposición dentro de cualquier lenguaje contenía una cantidad de información mínima, a esta propiedad se le conoce como *lenguaje de bajo nivel* [13].

Por otro lado, programar en lenguaje de máquina requiere de un conocimiento muy profundo de la arquitectura correspondiente, además de un esfuerzo y tiempo considerable, ya que la construcción de un programa depende de la arquitectura donde se va a ejecutar. Ahora bien, si consideramos que una computadora ejecuta sus instrucciones sobre el sistema binario, a través del reconocimiento de patrones, entonces es muy claro que corregir y depurar un programa a partir de lenguaje de máquina, es una tarea muy complicada [19].

Ante este obstáculo se consideró construir funciones cuyos parámetros formales fueran las operaciones representadas como nemotecnias de dos o tres letras, así como diseñar direcciones de memoria en sistema octal[1]. De está manera se conformó una representación simbólica del lenguaje máquina conocida como *lenguaje ensamblador*. Entre sus principales tareas se encuentra la corrección y depuración de programas [13].

Sin embargo, la densidad de información seguía siendo baja, además la programación aún requería de un conocimiento profundo de la arquitectura correspondiente, incluso los programas no podían copiarse en máquinas distintas, por lo que la programación era lenta y escasamente accesible [13].

[1] Octal se refiere a un sistema numérico de base 8 [13].

A partir de estos problemas, los programadores consideraron:

- Generalizar las expresiones aritméticas, por lo que pusieron su atención en los estándares simplificados y estandarizados de la notación matemática.
- Crear procesadores de alto nivel que traduzcan un lenguaje de programación a lenguaje de máquina, ya sea antes del tiempo de ejecución, hoy conocidos como *compiladores* o como un procesador que combine traducción y ejecución, llamado *intérprete*. El objetivo fue prescindir de la comprensión del funcionamiento interno de la máquina.
- Diseñar algoritmos y programas con la capacidad de ejecutarse en varias arquitecturas.

El panorama anterior motivó el desarrollo de `Fortran` (**FOR**mula **TRAN**slator) de 1954 a 1956, donde un grupo que trabajaba para IBM y cuyo líder fue John Backus, planteó facilitar la implementación de cálculos numéricos. Para ello, introdujo la escritura de instrucciones en notación matemática. `Fortran` fue el primer lenguaje de uso popular considerado *lenguaje de alto nivel* que presentó subrutinas, arreglos, formatos para entrada y salida, así como la alternativa de un manejo de memoria manual, a través de instrucciones donde se controlaban las direcciones de memoria que las variables y arreglos ocuparían. Sin embargo, esta propiedad podría significar un reemplazo de datos no intencionados por el programador si no tenía cuidado. Otra propiedad importante es que las subrutinas no eran capaces de llamarse a sí mismas ya que en esa época no existía el concepto de recursión [13].

Eventualmente el panorama buscaba dar una notación concisa y fácil de usar para la descripción de archivos y de las distintas relaciones que guardan los datos entre sí, así como de operaciones sencillas de comparación y desplazamiento de información. Situación que generó el primer diseño de `Cobol` (**CO**mmon **B**ussiness **O**riented **L**anguage) en 1959, hecho por Grace Murray Hooper, cuya sintaxis fue diseñada para lucir como el idioma inglés. Es un lenguaje enfocado a las aplicaciones de negocio, principalmente orientado a archivos y aplicaciones, implementa autodocumentación, posee solamente tres tipos de datos y una estructura de cuatro niveles jerárquicos en sus programas [19].

Posteriormente se deseaba un procesamiento de palabras, denominadas cadenas de caracteres. Fue en el año de 1964 que aparece el lenguaje `Snobol` cuyas funciones predefinidas poseen esta propiedad. Integró en sus funciones elementales la localización y manipulación de patrones particulares. Por otro lado, introdujo la extensibilidad de lenguajes mediante la definición de funciones por el usuario [13].

Mientras tanto el campo de inteligencia artificial demandaba un procesamiento de expresiones simbólicas, por lo que en 1960 surge el lenguaje `Lisp` (**LIS**t **P**rocessor) diseñado por John McCarthy. Su creador lo describió como:

> "*un esquema para representar las funciones parciales recursivas*[2] *de una cierta clase de expresiones simbólicas [13].*"
>
> "*Para entender lo que brindó* `Lisp` *se debe considerar el punto de vista matemático en la programación ya que si se consideran a las funciones como elementos fundamentales de un lenguaje de programación, notamos que existen funciones para los cuáles existen programas que pueden calcular el resultado, llamadas funciones computables.* `Lisp` *se basó en el cálculo lambda*[3] *para representar a la clase de funciones parciales recursivas computables.*" según John Mitchell [13].

Su estructura de datos básica fue la lista, un conjunto básico de operaciones sobre ellas como `cons`, `car`, `cdr`, `eq` y `atom` entre otras. Operadores para definición de funciones y estructuras de control como `cond`, `lambda`, `define`, `quote` y `eval` [13]. `Lisp` aportó funciones capaces de recibir funciones como argumento o regresar funciones como resultado, llamadas *funciones de primera clase*. La ejecución de procesos además de evaluarse a un resultado no modificaba su entorno, fenómeno conocido como *efecto colateral*. También desarrolló un recolector automático de direcciones de memoria que no son utilizadas en la ejecución de un programa, el cual es conocido como *recolector de basura* [13].

Posteriormente se desean implementar las siguientes propiedades:

- Claridad de programas.
- Facilidad de corrección.
- Consistencia.
- Posibilidad de desarrollo a través de refinamientos del problema.

Con base en estas necesidades, la familia `Algol` (**ALGO**rithmic **L**anguage) inició su camino en la programación. Comenzó con `Algol 58` a finales de la década de 1950. Entre 1958 y 1963 donde un comité internacional que incluía a importantes pioneros en computación, tales como John Backus (diseñador de `Fortran`), John McCarthy (diseñador de `Lisp`) y Alan Perlis diseñaron `Algol 60`, cuyo objetivo fue crear un lenguaje de propósito general, lo cual en esa época involucraba un enfoque científico y aplicación numérica. Comparado con `Fortran`, `Algol 60` incluyó la representación de datos, y apoyándose en `Lisp` permitió el usó de funciones que se llaman recursivamente [13]. Entre sus principales características se pueden mencionar las siguientes:

- Sintaxis simple orientada a procedimientos, incluyendo secuencias separadas de cada procedimiento.

[2] Funciones construidas en términos de sí mismas, definidas en algunos argumentos y no definidas en otros, debido a las operaciones básicas o a un cálculo infinito [13].

[3] Sistema matemático enfocado a la definición de funciones, prueba de ecuaciones de expresiones y cálculo de valores de expresiones [13].

- Bloques de código indicados por las instrucciones `begin` y `end`.
- Funciones recursivas.
- Un sistema primitivo de chequeo de tipos en tiempo de compilación.
- Expresiones regulares dentro de los índices de un arreglo.

Sin embargo, `Algol 60` tuvo algunos inconvenientes como:

- Efectos colaterales. La ejecución de procesos además de evaluarse a un resultado puede modificar su entorno. Por ejemplo, modificar el estado de variables globales o imprimir datos en pantalla.
- Un control de flujo con problemas de ejecución, ya que el manejo de memoria en bloques anidados no estaba bien definido[4] [13].

Posteriormente nació `Algol 68` cuyo objetivo fue sanar las deficiencias de su antecesor, sin embargo, al final, este lenguaje resultó ser más complicado, situación que repercutió en su compilación deficiente. Por ejemplo, la combinación de parámetros de procedimiento y valor de retorno, no fue suficientemente comprendida en su tiempo. Otro inconveniente fue la definición de una terminología totalmente nueva, lo que complicó pasar de `Algol 60` a `Algol 68`. Entre sus contribuciones se encuentra su sistema de tipos sistemático y regular. Los tipos eran conocidos como modos. Existían los modos primitivos que incluían `int`, `real`, `char`, `boolean`, `string`, `complex`, `bits`, `bytes`, `semaphore`, `format` (entrada-salida) y `file`. De igual manera contaba con modos compuestos como `array`, `structure`, `procedure`, `set` y `pointer`. Las construcciones de tipos se generan sin restricción [19].

Entre las aportaciones trascendentales de `Algol 68` se encuentra su manejo de memoria a través de pilas en variables locales y almacenamiento dinámico para datos sobrevivientes en toda llamada de función, así como la integración de paso por valor y paso por referencia. Se agregaron los tipos *apuntadores*, propiedades posteriormente heredadas a `C` y `Pascal` [19].

En 1961 surge `APL`(**A P**rogramming **L**anguage) que aportó claridad en la notación de operaciones entre vectores, matrices y árboles extendiendo el uso de los operadores clásicos. Sin embargo, usa caracteres que no pertenecen a códigos estándares para el intercambio de información como **ASCII** o **ISO-8859-1**[13].

Años más tarde el concepto de eventos simultáneos dentro del ámbito naval, provoca que Ole-Johan Dahl y Kristen Nygaard dentro del Centro de Cómputo Noruego, en Oslo, diseñen el lenguaje `Simula 67`, cuya base fue `Algol 60`. El lenguaje `Simula 67` aporta comunicación entre dos eventos simultáneos implementando hilos de ejecución y concurrencia. Más allá de su propósito, `Simula 67`

[4] Se refiere a la ausencia de ambigüedad en su implementación.

se convirtió en el primer lenguaje que introdujo el concepto de clase en la programación, búsqueda dinámica, subtipos y herencia [13].

En los setentas aparece `Pascal`, diseñado por Niklaus Wirth. Lenguaje que hereda de `Algol 68` su estilo de programación por bloques además de la tipificación de sus variables. Aporta extensibilidad en los tipos de variables y simplifica considerablemente su compilación, propiedad que garantizó su mayor disponibilidad. Su aplicación principal se presentó dentro de la enseñanza, además del diseño de sistemas operativos y la creación de aplicaciones para Apple Macintosh [13].

Posteriormente, la idea de diseñar sistemas para computadoras como compiladores, sistemas operativos o editores a partir de un lenguaje de alto nivel, hizo posible la combinación de procesos de alto y bajo nivel. Así se diseñó `C`, en el periodo comprendido por 1969 y 1973, con el objetivo inicial de apoyar la implementación del sistema operativo Unix en los Laboratorios Bell. Su creador es Dennis Ritchie. `C` posee tipos y reglas de chequeo de tipos. Su principal característica es el manejo de locaciones de memoria, arreglos y apuntadores. La tolerancia de los compiladores de `C` a errores de tipo provocó su estandarización por un comité ANSI en 1988. El chequeo de tipos de `C` fue mejorado posteriormente por `C++` [19].

Pocos años después, Xerox PARC desarrolla un proyecto llamado `Smalltalk`, basado en la programación orientada a objetos. Aunque el concepto de programación orientada a objetos proviene del lenguaje `Simula 67` en sus distintas versiones, `Smalltalk` no consiste en una extensión sino en un lenguaje nuevo con terminología y sintaxis propia. Define el concepto de objeto: [13]

- Todo es un objeto, incluso una clase.
- Todas las operaciones son mensajes a los objetos.
- Considera a clases y objetos como conceptos organizados para construir un ambiente de programación.
- Definición de alcances y visibilidad en variables, métodos y clases.
- Expresiones regulares dentro de los índices de un arreglo.

En 1930, Kurt Gödel y Jacques Herbrand estudiaron la noción de computabilidad[5] basada en derivaciones. Trabajo que representó un puente entre la computación y la deducción. Posteriormente Herbrand utilizó el algoritmo de unificación para manipular ecuaciones algebraicas sobre términos. Así en 1965 Alan Robinson muestra la deducción automatizada como un campo de la computación. En este contexto nace Prolog, implementado por Philippe Roussel, en

[5]Partiendo de las funciones computables y no computables, la computabilidad se define como el conjunto de lenguajes de programación capaces de definir funciones computables [13].

principio `Prolog` es un intérprete escrito en `Algol`. Su estandarización se alcanzó cuando David H. Warren propuso en 1983 una máquina abstracta WAM que permitió la portabilidad de `Prolog` [13].

Entre las aportaciones de `Prolog` se puede mencionar la comunicación entre lógica matemática y lógica computacional. Además, los valores son asignados a las variables de manera automática a través de sustituciones llamadas unificaciones más generales. Por último, el control se implementa mediante *retroceso automático* (del inglés *automatic backtracking*) [19].

A mediados de los años 70 en el MIT, inició un experimento en el diseño de lenguajes de programación, desafiando algunos supuestos fundamentales del diseño. Se trataba de un dialecto[6] de `Lisp`, creado por Guy Lewis Steele Jr. y Gerald Jay Sussman. Originalmente llamado `Schemer`, una limitación de 6 caracteres en nombres de archivo provocó su reducción a `Scheme`. Este lenguaje fue diseñado para apoyar una variedad de estilos de programación incluyendo los paradigmas funcional, orientado a objetos e imperativo [18].

Las principales características de `Scheme` son: [19]

- Tipos polimórficos.
- Funciones de primera clase, es decir, funciones que pueden ser pasadas como argumentos y devueltas como resultados de otras funciones.
- Evaluación glotona. Se evalúan las expresiones cuando son ligadas a una variable.
- Extensión sintáctica a través de programas para definir y utilizar nuevos tipos de expresión derivados, conocidos como *macros* [9].
- Manejo de una operación llamada *continuación* (del inglés *continuation*).

> "*Operación capaz de crear una representación del resto del cálculo como un procedimiento de un argumento. Así cuando tal procedimiento proporciona un valor, el resto del cálculo se reiniciará con tal valor.*" según Shriram Krishnamurthi [9]

Desde el laboratorio de Inteligencia Artificial del MIT, `Scheme` trascendió a diversos lugares. Dan Friedman y Mitch Wand lo utilizaron dentro de su grupo de investigación en la Universidad de Indiana, para construir una versión llamada `Scheme84`. La razón principal del uso popular de `Scheme` dentro de la investigación en lenguajes de programación es tan simple y regular que es muy posible estudiar los efectos de adherirle una nueva idea, o incluso de estudiar modelos sintácticos. Posteriormente un grupo de investigadores en la Universidad de Yale, cuyo líder fue Paul Hudak, crearon otro dialecto llamado `T`. Ellos

[6] Lenguaje creado a partir de la extensión o variación de otro lenguaje sin que éste pierda su naturaleza [19].

consideraban a `T` como un proyecto de compilador, con el objetivo de compilar un lenguaje expresivo en lenguaje máquina. En esos momentos se crearon una gran cantidad de mezclas y extensiones de `Scheme` [13].

Una de esas implementaciones surgió en la Universidad Rice en Houston, Texas. Mathias Felleisen, Robby Findler, Matthew Flatt y Shriram Krishnamurthi, buscaban usar `Scheme` para enseñar matemáticas básicas de una manera creativa. Debido al éxito de su objetivo, los cuatro investigadores construyeron un amplio software en `Scheme`, obteniendo versiones de este lenguaje cada vez más pequeñas [13].

Estos lenguajes fueron evolucionando al incorporar sucesivamente estructuras, sistemas de clases, bibliotecas para la construcción de interfaces gráficas de usuario, módulos, continuaciones, entre otras propiedades. Debido a la satisfacción que les generó el proyecto, los investigadores mencionados decidieron llamarle `Racket` a su nuevo lenguaje de programación, cuyas propiedades se verán en el capítulo 4 [13].

En 1987 se diseña `Haskell`, un lenguaje de programación puramente funcional cuyas principales características son: [9]

- El concepto de mónada inmerso en la teoría de categorías[7], tal innovación permite implementar modularidad en los programas.
- Implementó tipos abstractos.
- Capacidad de representar una función que recibe dos argumentos como una función que recibe un argumento y regresa una función de un argumento, propiedad conocida como *curryficación* (del inglés *curryfication*) [7].
- Tipos de datos algebraicos.
- Evaluación perezosa.
- Listas por comprensión.
- Emparejamiento de patrones.
- Concepto de funciones de primera clase.
- Tipos sinónimos.
- Tipos polimórficos.

[7] Teoría inventada a principios de los años cuarenta por Samuel Eilenberg y Sanders Mac Lane, cuyo objetivo es axiomatizar diferentes campos de matemáticas a través de estructuras más abstractas llamadas morfismos [17].

`Java` fue desarrollado por James Gosling en Sun MicroSystems en la década de 1990, su sintaxis se basa en la sintaxis de `C` y `C++`. Omite instrucciones de bajo nivel como apuntadores para el manejo de memoria. Implementa un sistema de tipos nativos, los cuáles no son objetos. Este lenguaje utiliza una Máquina Virtual `JVM` (del inglés *Java Virtual Machine*) que traduce el código compilado a lenguaje máquina. Al igual que `Lisp` contiene un recolector de basura eficiente. `Java` añade a las propiedades de `C` y `C++` el manejo de excepciones, así como la introducción de hilos de ejecución, además permite ejecutar código en otros lenguajes mediante `JNI` (del inglés *Java Native Interface*). Su portabilidad lo ha convertido en un lenguaje muy popular [13].

La Tabla 1.1 presenta gráficamente las propiedades más relevantes de los lenguajes de programación de alto nivel, a partir de los cuales se derivan casi todos los lenguajes de programación existentes en la actualidad.

Lenguaje	Expresiones	Funciones	Excepciones	Módulos	Objetos	Hilos	Tipificado explícitamente	Tipificado implícitamente
Fortran	•	•					•	
Lisp	•	•						•
C	•	•					•	
Algol 60	•	•					•	
Algol 68	•	•				•	•	
Prolog	•	•		•				•
Modula-2	•	•		•			•	
Modula-3	•	•	•	•	•		•	
Scheme	•	•	•	•				•
Racket	•	•	•	•				•
Haskell	•	•	•	•			•	•
Simula	•	•			•	•	•	
Smalltalk	•	•	•		•	•		•
C++	•	•	•	•	•		•	
Java	•	•	•	•	•	•	•	

Figura 1.1: Basada en el libro Concepts in Programming Languages de John Mitchel (página 8) [13].

Durante este capítulo se habló de la historia de los lenguajes de programación con el fin de obtener una perspectiva general sobre las principales necesidades en las que cada paradigma de programación se ha enfocado. Situación que ayudará a entender el impacto de este trabajo de una manera mas detallada y precisa. Incluso las diferencias conceptuales y prácticas de los lenguajes vistos en este capítulo ayudarán a considerar los caminos sobre los que se sitúa esta investigación.

Capítulo 2

Sistema de tipos

Una variable puede tomar cierto rango de valores durante la ejecución de un programa. Dicho rango nos proporciona información sobre su comportamiento. Por ejemplo, una variable x cuyo rango de valores lo conforman los enteros, si se asume que solo recibirá valores enteros, entonces la expresión $x + x$ tendría sentido en cada ejecución del programa, a este concepto se le conoce como *tipo*, sin embargo, su definición no es simple debido a la poca precisión del concepto *valor*. Por ello se incorpora la siguiente definición:

> "*Un tipo es la abstracción de un conjunto de valores*" según Shriram Krishnamurthi [9].

A continuación se presentan los tipos mas importantes en el diseño de un lenguaje de programación: [15]

- *Básico* o *atómico*. Conjunto de valores simples, no estructurados, tales como números, booleanos o caracteres. Por ejemplo, `Integer`, `Boolean`, `String` (como "hello"), `Float` (elementos como 3.1416) [15].
- *Unidad*. Es el tipo interpretado de la manera mas simple. Sirve para indicar solo un valor, el cual no posee contenido alguno. Se denota como `Unit` en los lenguajes funcionales y como `void` en lenguajes como `Java` y `C` [15].
- *Producto*. Es el tipo asociado a las estructuras de datos compuestas, las cuáles son: [15]
  - Pares. Forma mas simple de construir estructuras de datos compuestas, conocidas como pares de valores. Una notación común se da usando llaves, por ejemplo, al par $\{t_1, t_2\}$ posee el tipo $T_1 \times T_2$. Esta notación representa la operación producto cruz usada en el álgebra, en este caso denota conjuntos de valores con tipo T_1 y tipo T_2 .
  - Tuplas. Generalización de productos binarios en productos de aridad n. Por ejemplo, $\{1, 2, \texttt{true}\}$ es una tupla de aridad 3 que contiene dos valores numéricos y un valor booleano. Su tipo es señalado por la tupla $\{\texttt{Integer}, \texttt{Integer}, \texttt{Booleano}\}$.

  - Registros. Generalización de tuplas de aridad n a tuplas etiquetadas. Por ejemplo, $\{x = 5 : \texttt{Float}\}$, representa un registro de aridad 1, donde la etiqueta es x, mientras $\{parte = 5224 : \texttt{Integer}, costo : \texttt{Float}\}$ representa un registro de aridad 2 con las etiquetas *parte* y *costo*.

- *Función*. Tipo asociado a las funciones o procesos definidos dentro de un lenguaje de programación. Por ejemplo, la función *iszero* de `Scheme`, la cual recibe un entero y nos devuelve un booleano `true` si tal entero es el cero o un booleano `false` en caso contrario, posee el tipo $\texttt{Integer} \rightarrow \texttt{Boolean}$ [15].

- *Suma*. Tipo asociado a colecciones heterogéneas de valores, por ejemplo, colecciones donde un nodo en un árbol binario puede ser una hoja o puede ser un nodo interior con dos hijos. Su objetivo es describir un conjunto de valores por dos tipos dados. Por ejemplo, supongamos que estamos usando los siguientes registros (estructura de datos vista en el tipo *Producto*):

$$DirFisica = \{nombre : \texttt{String}, dir : \texttt{String}\}$$
$$DirVirtual = \{id : \texttt{String}, email : \texttt{String}\}$$

  El objetivo es representar tipos de registros de un libro de direcciones. Si se desean manipular ambos tipos de registros uniformemente, se puede introducir el tipo *suma*:

$$Dir = DirFisica + DirVirtual$$

  Donde cada uno de sus elementos puede ser de tipo $DirFisica$ o de tipo $DirVirtual$.

- *Variante*. Generalización de sumas binarias a sumas etiquetadas. También se les conoce como *uniones disjuntas*. Considerando el ejemplo anterior, un manera de asignar tipo variante es:

$$Dir = fisica : DirFisica + virtual : DirVirtual$$

- *Recursivo*. Tipo que permite definir estructuras en términos de sus elementos mas simples. Por ejemplo, si utilizando el lenguaje `Racket` se desea definir la lista de todos los números enteros $List(Integer)$, se debe tomar en cuenta que existen dos opciones, la lista es vacía, la cual en `Racket` se denota por la primitiva *nil* o la lista es un par (estructura de datos vista en el tipo *producto*) compuesta por un número y otra lista. Entonces se puede construir con la operación de `Racket` llamada `cons`, la cual construye una nueva lista tomando como parámetros un entero y una lista. A continuación se denota recursivamente la lista de los números enteros:

$$List(Integer) =< nil : \texttt{Unit}, \texttt{cons} : \{\texttt{Integer}, List(Integer)\} >$$

Donde los símbolos < y > denotan un tipo recursivo, en este caso compuesto por la lista vacía *nil* de tipo `Unit` y la primitiva `cons`, la cual posee el tipo *Producto* denotado por la estructura par formada por los tipos `Integer` y $List(Integer)$.

- *Universal*. Para definirlo se debe introducir una operación conocida como *sustitución de tipos*, la cual, es un mapeo finito de variables que denotan tipos a tipos concretos. Por ejemplo, al escribir $[U \to V]$ se está definiendo la sustitución que asocia U por V.

  Ahora se puede definir un tipo universal de la siguiente forma:

  Un elemento de tipo $\forall X.T$ es un valor que bajo un dominio de tipos S a considerar, posee tipo $[X \to S]T$ para todas las elecciones de S. Su principal aplicación es llamada *polimorfismo universal* [15].

- *Existencial*. De igual forma, se considera la operación *sustitución de tipos* vista en el punto anterior para definir un tipo existencial de la siguiente forma:

  Un elemento de la tupla $\{\exists X, T\}$ es un valor que posee tipo $[X \to S]T$ para algún tipo de S. Su principal aplicación se da en la implementación de tipo *abstracto* [15].

- *Abstracto*. Tipo que define el programador. Se compone por un nombre, una representación concreta de tipos ya existentes y la implementación de operaciones asociadas para manipularlo [15].

Los tipos presentes en un lenguaje de programación permiten definir un *sistema de tipos* cuyo propósito es evitar algunos errores de ejecución:

> "*Un sistema de tipos es una colección de reglas, conocidas formalmente como juicios de tipo, los cuales describen como determinar el tipo de una expresión y un algoritmo para su implementación.*"
>
> según Shriram Krishnamurthi [9].

El proceso de análisis de tipos es llamado *verificación de tipos* (del inglés type checking) y el algoritmo para llevar a cabo dicho análisis es conocido como *verificador de tipos* (del inglés *type checker*) [15].

Un lenguaje es tipificado si posee un sistema de tipos, independientemente de que los tipos aparezcan en el código o no [15].

El sistema de tipos predice el *comportamiento* de un programa infiriendo su información de tipos. Para comprender a lo que se refiere el término *comportamiento*, es importante notar que el chequeo de tipos no asegura la *correctud* de un programa, sólo garantiza que el programa no tendrá cierto tipos de errores, conocidos como errores de tipo en tiempo de ejecución. Por ejemplo, muchos

sistemas de tipos no distinguen si un programa ordena a un conjunto de valores ascendente o descendentemente [10].

2.1. Formalización del sistema de tipos

Con el objetivo de definir propiedades en un sistema de tipos, se debe determinar la variedad de errores de ejecución que existen. El síntoma mas obvio de un error de ejecución es la aparición de una falla en el software que detiene la ejecución del programa al momento en que ocurre, conocidos como *errores cazados* (del inglés *trapped errors*), por ejemplo, divisiones entre cero o el acceso ilegal a direcciones de memoria. Sin embargo, existen errores que no presentan un síntoma inmediato, es decir, que pasan desapercibidos temporalmente llamados *errores no cazados* (del inglés untrapped errors), por ejemplo, acceder a una región de memoria legal pero no adecuada o sobrepasar la longitud de un arreglo. Si en un programa no existen errores no cazados entonces se dice que el programa es *seguro* [15].

Ahora se definirá a un conjunto de errores de ejecución (*errores prohibidos*), el cual está conformado por la totalidad de errores no cazados y algunos errores cazados. Si en un programa no existen errores prohibidos entonces se le denomina *programa bien comportado*. Ahora bien, si todos los programas legales en un lenguaje de programación son bien comportados entonces se dice que este lenguaje es *fuertemente verificado* (del inglés *strongly checked*) [15].

La formalización de un sistema de tipos exige notación precisa además de definiciones. De esta manera, se pueden proveer pruebas formales de aspectos importantes en la definición de un lenguaje de programación. Cuando el sistema de tipos se cumple para todos los programas escritos en un lenguaje, se dice que dicho lenguaje es *correctamente tipificado* (del inglés *type sound*) [15].

A continuación, se señalan los pasos principales para formalizar sistema de tipos:

1. Describir la sintaxis del lenguaje de programación. Se puede definir formalmente usando la **FNB** (*Forma Normal de Backus Naur*). Los tipos son asociados a los términos de un lenguaje de programación. Los términos son los elementos contenidos en su definición, a partir de los cuales se construyen todas las sentencias sintácticas válidas, conocidas como expresiones, en dicho lenguaje [15].

2. Describir las reglas de alcance del lenguaje. Dicho alcance debe ser estático en lenguajes tipificados, es decir, el alcance de cada identificador debe estar bien establecido antes de la ejecución [15].

3. Se definen las reglas de tipificación. Su objetivo es comprobar que un juicio es válido. Se crean las relaciones entre términos y tipos de la forma $X : T$, denotando que el término X es de tipo T [15].

4. Definir un conjunto que guarde la información de las variables no cazadas en un fragmento de código en un momento determinado durante el análisis de tipos. Cada regla de tipificación es formulada respecto a contextos estáticos. La relación se escribe como $\Gamma \vdash X : T$, lo que significa que X es de tipo T en el contexto Γ [15].

5. En este punto se diseñan las reglas de tipificación de la siguiente forma:

$$\frac{\Gamma_1 \vdash \Im_1 \ldots \Gamma_n \vdash \Im_n}{\Gamma \vdash \Im}$$

   Cada regla de tipificación contiene cierta cantidad de *premisas* ($\Gamma_i \vdash \Im_i$) sobre una línea horizontal y un solo juicio de *conclusión* ($\Gamma \vdash \Im$) bajo la línea. Cuando todas las premisas son ciertas, entonces es válida la conclusión; en caso de que haya cero premisas se trata de un hecho. Cuando el contexto es vacío decimos que nuestro juicio es una aseveración [15]. Más adelante, en el subcapítulo Semántica Estática 3.2.1 de `MinTyRacket` se pueden observar ejemplos de reglas de tipificación utilizando la notación descrita.

   Este punto nos permitirá comprobar la validez de un juicio si se utilizan *derivaciones de tipo*. Una derivación de tipo está compuesta por un árbol de juicios con las hojas arriba y la raíz abajo. Cuando la raíz pueda ser derivada mediante el sistema de tipos se puede concluir que el juicio es válido.

6. Definir una relación de valor entre los términos del lenguaje y una colección de resultados. Dicha relación es conocida como *semántica* [15].

Las siguientes son características importantes de los sistemas de tipos [15]:

- Documentación. Los tipos documentan un programa de manera más simple que los comentarios.
- Transparencia. La forma de escribir los tipos debe permitir al programador predecir un error de tipo.
- Abstracción. Un sistema de tipos debe permitir la implementación de tipos abstractos.
- Seguridad. Los programas fuertemente tipificados no pueden funcionar mal.
- Detección de errores. Permiten detectar errores de programación tempranamente.

2.2. Lenguajes tipificados explícitamente

Los *lenguajes tipificados explícitamente* son aquellos en los que la verificación de tipos se realiza en tiempo de compilación. El tipo de sus expresiones forma parte de la sintaxis del lenguaje. Ejemplos de estos lenguajes son `Ada`, `C`, `C++`, `C#`, `Fortran`, `Java`, `ML` y `Scala` entre otros [9].

Un sistema de tipos que pertenece a los lenguajes tipificados explícitamente puede definirse de la siguiente manera:

> "*Un sistema de tipos es un método que monitorea la sintaxis de un programa para probar la ausencia de comportamiento de ciertos programas al clasificar sentencias de acuerdo al tipo de valores que calculan.*"
>
> según Benjamin Pierce [15].

Esta definición presenta a un sistema de tipos que realiza la verificación de tipos a partir de la información de tipos que ha sido señalada por el programador [15].

Una propiedad interesante es que debido a que los sistemas de tipos predicen de manera estática el comportamiento de un programa, necesariamente también rechazan algunos programas semánticamente válidos en tiempo de ejecución:

> "*Ellos pueden probar categóricamente la ausencia de ciertos malos comportamientos de programas, sin embargo, no pueden probar su presencia, por lo que en ocasiones rechazan programas que se comportan adecuadamente en tiempo de ejecución.*"
>
> según Benjamin Pierce [15].

Existen lenguajes tipificados estáticamente que no son seguros, es decir, que permiten errores no cazados en sus programas, a estos lenguajes se les llama *débilmente verificados* (del inglés *weakly checked*), un ejemplo es `C`, debido a su aritmética de punto flotante [15].

2.2.1. ¿Por qué usar lenguajes tipificados explícitamente?

Es necesario comprender las consecuencias que se desprenden al indicar tipos dentro de la sintaxis del código y con base en ello determinar su utilidad.

Supongamos que se presenta el siguiente fragmento de programa escrito en `Java`:

```
1 if (3 > 4) {
2          List<Integer> l = new Integer(4);
```

```
3           l.add(3);
4           }
```

Código 2.1: Líneas con error de tipo [11].

Dentro del Código 2.1, en la línea 1 se establece una condicional, que siempre se evalúa falso, por lo que el cuerpo del condicional contenido en las líneas 2 y 3 nunca se ejecuta, sin embargo, el sistema de tipos de `Java` (lenguaje tipificado explícitamente) no evalúa dichas condiciones, de hecho una vez hecho el análisis de tipos en la condicional, hace un análisis de tipos de las siguientes líneas. En la línea 2 encontrará un error ya que no coinciden los tipos, por un lado se declara una lista de enteros `List<Integer> l`, y en la misma línea se inicializa un dato de tipo entero `new Integer(4)`. El sistema de tipos de `Java` detecta y señala esta disparidad de tipos a través del mensaje `Type mismatch: cannot convert from Integer to List<Integer>`, es decir, el código mostrado en el Código 2.1 no se ejecuta [11].

La situación anterior puede parecer un inconveniente en los lenguajes tipificados explícitamente, pero veamos que sucede si suponemos que un software controla el tránsito aéreo, y tal software contiene el siguiente fragmento de código escrito en `Java`:

```
1 if (interferencia > 400 + tiempo) {
2           List<Integer> l = new Integer(4);
3           l.add(3);
4           }
```

Código 2.2: Líneas con error de tipo [11].

En el Código 2.2, en la línea 1, se establece un condicional, donde aparecen dos variables llamadas $interferencia$ y $tiempo$. La condición puede evaluarse a verdadero si el valor de la variable $interferencia$ es mayor que el valor que se obtiene al sumar 400 al valor contenido por la variable $tiempo$. En este caso, se ejecutan las líneas 2 y 3 que como se vio en el Código 2.1 generan un error de tipo. Es decir, si el sistema de tipos de `Java` permite la ejecución de este fragmento de código el control de tránsito aéreo se interrumpiría, con el inminente riesgo de provocar accidentes [11].

A continuación se mencionan las principales características de los lenguajes tipificados explícitamente: [5]

- Detección temprana de errores de tipo. El análisis se realiza antes de ejecutar el programa.
- Una eficiente documentación. Los tipos proporcionan al programador información útil sobre el comportamiento del programa.
- Economía en compilación. Se puede clasificar en interfaces la información que proporcionan los tipos, lo que permitiría compilar cada interface de manera independiente.

- Reducen el esfuerzo necesario para solucionar errores de tipo. La información que proporcionan los tipos ayuda a detectarlos con mayor facilidad.
- Reduce estilos de codificación que puedan dificultar el mantenimiento del programa. El programador debe ajustarse a un estilo de sintaxis muy rígido.

Los lenguajes tipificados explícitamente son muy populares en la actualidad debido a que presentan una mayor legibilidad en sus programas lo cual facilita su mantenimiento, además, presentan una mayor eficiencia del código que ha sido aceptado por el sistema de tipos correspondiente una vez que se ejecutan [1].

2.3. Lenguajes tipificados implícitamente

Los *lenguajes tipificados implícitamente* son aquellos en los que la verificación de tipos se realiza en tiempo de ejecución. El tipo de sus expresiones no forma parte de la sintaxis del lenguaje. Como ejemplos podemos mencionar a `Groovy`, `JavaScript`, `Lisp`, `Objective-C`, `Perl`, `Python`, `Ruby`, `Scheme`, `Smalltalk`, `Racket`, entre otros [9].

En los lenguajes tipificados implícitamente, durante la traducción de cadenas de caracteres que los programadores leen y escriben conocida como *sintaxis concreta* a la representación interna de los programas conocida como *sintaxis abstracta*, se etiqueta con un tipo X a cada variable que se define en el programa, esta etiqueta debe ser única, es decir, debe ser distinta a las etiquetas definidas anteriormente. Posteriormente, en tiempo de ejecución se lleva a cabo la inferencia de tipos, cuyo propósito es encontrar los valores más generales para dichas variables [15].

2.3.1. ¿Por qué usar lenguajes tipificados implícitamente?

Es necesario comprender la utilidad de lenguajes donde los tipos no son parte de la sintaxis del código y con base en ello determinar las propiedades que generan. Es suficiente determinar a grandes rasgos su uso más común en la actualidad.

Considerando el siguiente programa en `Racket`:

```
1 #lang racket
2
3 (define indeter ( lambda ( f )
4 ( + 3 ( f 5) )))
```

Código 2.3: Líneas con error de tipo indeterminado [9].

En el Código 2.3, línea 3 se define una función de nombre *indeter* que recibe como parámetro una función f, mientras en la línea 4 se describe el cuerpo de *indeter*, donde se suma el número 3 al resultado de aplicar la función recibida f al número 5 [9].

La correctud de este código depende de la función f, si al evaluar la función f en 5 da como resultado un número, ya que el cuerpo de *indeter* presenta una suma que solo tiene sentido si ambos sumandos son un número. Debido a que esta expresión se nos puede presentar en múltiples contextos, no es posible determinar su legalidad sin examinar cada aplicación, las cuales dependen de otras sustituciones, es decir, se debería esperar a la ejecución del programa para determinar si regresa un valor esperado o no [9].

Con el propósito de tener más claridad se exhibe el siguiente código en `Racket`:

```
1 #lang racket
2
3 (define (misterio f)
4 (if (equal? 0 (f 5))
5  5
6 (lambda (x) x)))
```

Código 2.4: Líneas con tipo de resultado indeterminado [9].

En el Código 2.4, en línea 3 se define una función de nombre *misterio* que recibe como parámetro una función f, mientras en las líneas 4, 5 y 6 se describe el cuerpo de *misterio*, donde si al aplicar la función f al número 5 da como resultado el número 0, entonces la función *misterio* nos dará como resultado el número 5, como se indica en la línea 5, en caso contrario, la función *misterio* dará como resultado una función anónima, como se puede observar en la línea 6 [9].

Debido a que no se sabe nada de la función f, es imposible determinar a que tipo de valor se evaluará la función *misterio*, lo que significa que sólo la ejecución de esta función determinará el tipo de valor que *misterio* dará como resultado [9].

Considerando los dos ejemplos expuestos, se puede notar que dentro de escenarios donde se necesita una adaptación y abstracción alta es donde este tipo de códigos resultan muy interesantes. Como ejemplos de estos escenarios se puede mencionar la construcción de aplicaciones que requieren adaptación en tiempo de ejecución, la programación Web y la implementación de prototipos de sistemas entre otros escenarios [14].

A continuación mencionamos las principales características de los lenguajes tipificados implícitamente: [10]

- Código compacto. Omitir información explícita de tipos no forza al usuario a anotar partes excesivas de información ejecutable y no ejecutable de un programa.
- Proveen sistemas de módulos de diverso grado de complejidad, lo que permite mayor facilidad en la creación de grandes sistemas.
- Poseen una gran flexibilidad en tiempo de ejecución, lo que permite adaptarse a escenarios donde los datos de entrada son impredecibles.
- Contienen funciones de primera clase. Funciones que pueden ser pasadas como argumentos y devueltas como resultados de otras funciones.

Con base en estas propiedades, se puede decir que `Racket` es un lenguaje de múltiple aplicación que permite programar desde un alto nivel, necesario para producir sistemas concisos y comprensibles. Además dado que `Racket` tiene excelentes herramientas de abstracción es posible escribir sistemas sustanciales para el proceso de creación de lenguajes [18].

2.4. Subtipos

Si no existieran subtipos las reglas de tipos serían muy rígidas, pues la insistencia del sistema de tipos en que los tipos de los términos pertenecientes al lenguaje correspondan a un solo tipo establecido en su definición formal, obligaría al verificador de tipos a rechazar muchos programas tanto sintáctica como semánticamente válidos. Por ejemplo, supongamos que definimos una función que recibe como parámetro $\{x : \texttt{Integer}\}$, el cual es un tipo registro formado por un elemento x de tipo `integer`, y posteriormente aplicamos dicha función con el argumento denotado por $\{x = 0, y = 1\}$ de tipo $\{x : \texttt{Integer}, y : \texttt{Integer}\}$, debido a que el tipo del argumento es distinto al tipo del parámetro definido para la función y suponiendo que no se ha definido una relación entre ellos, entonces no es tipificable la aplicación de esta función al argumento $\{x = 0, y = 1\}$. Sin embargo, la función solo requiere que el argumento sea una registro que contenga un elemento x sin importar que también contenga otros campos [15].

El objetivo de los subtipos es refinar las reglas de tipificación para rechazar menos programas tanto sintáctica como semánticamente válidos. Intuitivamente podemos decir que S es un subtipo de T, escribiendo $S < T$, si cualquier término de tipo S puede usarse de forma segura en un contexto, donde un término de tipo T es esperado. Esta perspectiva de los subtipos es llama *principio de sustitución segura* [15].

A continuación se presentan dos perspectivas simples de la relación $S < T$:

- Todo valor descrito por S también es descrito por T.
- Los elementos de S son un subconjunto de los elementos de T.

2.4.1. Relación de subtipo

Para conectar esta relación de subtipos con la relación de tipos, se integra una nueva regla de tipos llama *regla de subsunción*:

$$\frac{\Gamma \vdash t : S \quad S < T}{\Gamma \vdash t : T}$$

Esta regla dice que si $S < T$, entonces cada elemento t de S es también un elemento de T.

Una regla fundamental de la relación de subtipos es reflexividad: [15]

$$\frac{}{\Gamma \vdash S : S}$$

Esta regla indica que todo tipo T es un subtipo de sí mismo.

Otra regla fundamental de la relación de subtipos es transitividad: [15]

$$\frac{S < U \quad U < T}{\Gamma \vdash S : T}$$

Esta regla dice que si un tipo S es un subtipo de un tipo U y U es un subtipo de un tipo T, entonces el tipo S es un subtipo del tipo T.

Para formalizar la relación de subtipo se define un conjunto de reglas de tipificación y subtipificación, incluyendo las tres anteriores, cuyo objetivo es derivar sentencias de la forma $S < T$. Considerando de forma independiente a cada forma de tipo (vistas al principio de este capítulo), para cada una de ellas, se define una o mas reglas que formalmente indican cuando es seguro permitir el uso de elementos de un tipo de esta forma en lugar de otra esperada [15].

En este capítulo se han presentado algunos aspectos teóricos tanto de los sistemas de tipos como de la relación de subtipificación. También se han analizado distintas maneras de realizar la inferencia de tipos que ayudará a comprender el diseño y formalización del intérprete `MinTyRacket`, lo cual se verá en el siguiente capítulo.

Capítulo 3

MinTyRacket

En los capítulos anteriores se han introducido los conceptos necesarios para desarrollar este trabajo. En este capítulo se presenta de manera formal un lenguaje de programación llamado `MinTyRacket` en alusión a un lenguaje de programación pequeño, tipificado explícitamente y derivado de `Racket`, el cual se podrá utilizar tentativamente en algún curso de lenguajes de programación a nivel licenciatura o maestría. El objetivo es analizar las propiedades que se presentan al implementar un análisis estático de tipos dentro de un lenguaje funcional como `Racket`.

3.1. Sintaxis

La sintaxis describe la estructura de programas bien formados en dos facetas. La primera de ellas describe lo que el programador escribe como código (un programa). A menudo se utilizan gramáticas libres de contexto para desarrollar esta faceta conocida como *sintaxis concreta*. La segunda faceta modela la estructura jerárquica de la sintaxis a través de árboles de sintaxis abstracta. la cual es conocida como *sintaxis abstracta* [2].

A continuación se describe a un programa legal en `MinTyRacket` como una de las siguientes expresiones:

- Variable.
- Valor literal entero.
- Valor literal flotante.
- Valor literal real.
- Valor literal verdadero.
- Valor literal falso.

- Operación aritmética. Aplicación de un *operador aritmético* a dos operandos, donde un operador aritmético puede ser:
  - Adición.
  - Sustracción.
  - Multiplicación.
  - División.
- Condicional. Considerando el valor de una expresión booleana se determina que expresión evaluar, si la rama descrita por la expresión *then* o la expresión *else*, donde una expresión booleana es una de las siguientes expresiones:
  - Valor literal verdadero.
  - Valor literal falso.
  - Operación relacional. Aplicación de un *operador relacional* a dos operandos, donde un operador relacional puede ser:
    - ◇ Igual que.
    - ◇ Menor que.
    - ◇ Mayor que.
- *Función*. Definición de una función compuesta por un argumento y el cuerpo donde se utilizará dicho argumento para obtener alguna de las siguientes expresiones:
  - Valor literal verdadero.
  - Valor literal falso.
  - Valor literal entero.
  - Valor literal flotante.
  - Valor literal real.
- *Aplicación*. Aplicación de una *función*, que recibe un argumento con el cuál se evaluará su cuerpo.
- *Aplicación de función recursiva*. Aplicación de una *función* que sustituye el cuerpo de la función recibida como argumento en su mismo cuerpo.
- *Lista vacía*. Expresión que denota a una lista sin elementos.
- *Verificador de lista vacía*. Expresión que se evalúa al valor literal verdadero si la lista que recibe como argumento es vacía. En caso contrario se evalúa a valor literal falso.
- *Atómico*. Descomposición de una expresión compleja en partes mas pequeñas. Recibe un argumento que utiliza para la evaluación de su cuerpo.

- *Constructor de lista*. Aplicación de un operador que construye una lista formada por el primer elemento que recibe como argumento y la lista que recibe como segundo argumento.
- *Primer elemento de lista*. Aplicación de un operador que obtiene el primer elemento de una lista que recibe como argumento.
- *Resto de lista*. Aplicación de un operador que devuelve la lista que recibe como argumento sin el primer elemento.
- *Error*. Expresión que denota una forma de terminar un programa a partir de alguna excepción.
- *Término*. Una expresión de `MinTyRacket` se conforma por las expresiones definidas anteriormente incluyendo a las operaciones aritméticas y relacionales.

3.1.1. Sintaxis concreta

Se utilizará la técnica usual de definición de una gramática libre de contexto en **forma de Backus-Naur** o **BNF** (en inglés *Backus Naur Form*) para definir el conjunto de cadenas que conforman un programa en `MinTyRacket`. Esta gramática se compone por un conjunto de producciones, donde cada producción describe la forma de una cierta clase de los elementos del lenguaje como instrucciones, expresiones y rutinas, entre otras. "*Un constructor que aparece del lado izquierdo de al menos una producción dentro de la gramática es conocido como constructor no terminal. Mientras un constructor que no aparece en el lado izquierdo de alguna producción es conocido como constructor terminal. Los símbolos no terminales describirán los constructores sintácticos, en tanto los símbolos terminales representarán los símbolos (cadenas de caracteres) válidos, mientras las reglas de producción dictarán la estructura de la sintaxis a través de reglas recursivas.*" [2]

A continuación se mostrará la sintaxis concreta de `MinTyRacket`:

$$
\begin{aligned}
\mathit{TipoAritm} &::= \texttt{integer} \mid \texttt{float} \mid \texttt{double} \\
\mathit{TipoRel} &::= \texttt{boolean} \\
\mathit{TipoElem} &::= \mathit{TipoAritm} \mid \mathit{TipoRel} \mid \texttt{Max} \mid \texttt{Min} \\
\tau &::= \mathit{TipoElem} \mid \mathit{TipoElem} \rightarrow \mathit{TipoElem} \mid \texttt{List}\ \mathit{TipoElem} \\
\mathit{OpAritm} &::= + \mid - \mid * \mid / \\
\mathit{OpRel} &::= < \mid > \mid == \\
\mathit{Op} &::= \mathit{OpAritm} \mid \mathit{OpRel} \\
\mathit{Var} &::= < \mathit{identificador} >
\end{aligned}
$$

De acuerdo a la definición anterior, un programa en `MinTyRacket` posee tipos como tipo función, booleanos, enteros, flotantes, dobles o tipo lista donde

cada uno de sus elementos posee algún tipo. El tipo `Min` como se verá mas adelante será subtipo de todos los tipos así como el tipo `Max` será el supertipo de todos los tipos, el objetivo es darle flexividad a `MinTyRacket`. Para hacer uso de operaciones primitivas como suma, resta, multiplicación o división se ha añadido la categoría $OpAritm$. Para usar operaciones relacionales se ha agregado la categoría $OpRel$. Por último, para denotar identificadores se ha añadido la categoría Var.

$$
\begin{aligned}
Elem ::= \; & n \mid Var \mid \texttt{true} \mid \texttt{false} \\
& \mid (\texttt{if}\ Elem\ Elem\ Elem\ (\tau)) \\
& \mid (\texttt{lambda}\ (Var\ \tau)\ Elem\ (\tau)) \\
& \mid (Elem\ Elem\ (\tau)) \\
& \mid (\texttt{fix}\ (Var\ Elem)\ (\tau)\ (Var\ Elem)\ (\tau) \\
& \mid (\texttt{let}\ (Var\ \tau\ Elem)\ Elem\ (\tau)) \\
& \mid (\texttt{null}\ (\tau)) \\
& \mid (\texttt{isempty?}\ Elem\ (\tau)) \\
& \mid (\texttt{list}\ Elem\ \ldots\ Elem) \\
& \mid (\texttt{cons}\ Elem\ Elem\ (\tau)) \\
& \mid (\texttt{fst}\ Elem\ (\tau)) \\
& \mid (\texttt{rst}\ Elem\ (\tau)) \\
& \mid (\texttt{error})
\end{aligned}
$$

$$
Exp ::= Elem \mid (Op\ Exp\ Exp\ (\tau))
$$

n puede ser cualquier número entero, flotante y real.

La gramática Exp nos permite establecer la combinación de los posibles programas que conforman `MinTyRacket`. De estas líneas partirá el análisis léxico y sintáctico con el fin de construir el árbol de sintaxis abstracta necesario para el diseño e implementación del sistema de tipos.

Es importante notar que de acuerdo a la gramática Exp no se permiten funciones de primera clase, la causa es que como veremos en el subcapítulo 3.5, introducir subtipos a `MinTyRacket` complica calcular el mínimo supertipo común de dos tipos asociados a funciones en caso de que exista, el cual es un tema extenso fuera del alcance principal de este trabajo.

Cuando se escribe una expresión compleja, con el fin de evitar su repetición y facilitar la lectura de un programa, resulta conveniente dividir la expresión en expresiones más pequeñas y posteriormente darles un nombre. Razón por la cual, se introduce en `MinTyRacket` el enunciado $(\texttt{let}\ (Var\ \tau\ Exp)\ Exp\ (\tau))$.

3.1.2. Sintaxis abstracta

La técnica usual de definición de una gramática libre de contexto en forma de Backus-Naur y sus variantes no representan una base para realizar estudios mas profundos en los lenguajes de programación, debido a que sus especificaciones incluyen detalles irrelevantes como lo son las *palabras clave* (en inglés keywords), así como otras convenciones de sintaxis concreta que añaden complejidad [18]. No obstante, para definir la sintaxis abstracta de `MinTyRacket` se utilizará la forma de Backus-Naur, la cual describirá la estructura jerárquica de los componentes del intérprete en lugar de describir una gramática.

La sintaxis abstracta cubre la necesidad de razonar cada relación definida en sintaxis concreta con el objetivo de establecer algunas propiedades matemáticas. Se establecerán representaciones abstractas para definir la verificación de tipos y la evaluación de programas en `MinTyRacket` ya que proporcionar definiciones y demostraciones utilizando solo cadenas es extremadamente tedioso e inapropiado [2].

A continuación se describe la sintaxis abstracta de `MinTyRacket`.

$$
\begin{aligned}
Abs ::= \ & \texttt{OP}\ Op\ Abs\ Abs \\
| \ & \texttt{IF}\ Abs\ Abs\ Abs \\
| \ & \texttt{LAMBDA VAR} < identificador > \tau\ Abs \\
| \ & \texttt{VAR} < identificador > \\
| \ & \texttt{LET VAR} < identificador > \tau\ Abs\ Abs \\
| \ & \texttt{FIX VAR} < identificador > \tau\ Abs \\
| \ & \texttt{NULL}\ \tau \\
| \ & \texttt{LIST}\ Abs_1\ \ldots\ Abs_n \\
| \ & \texttt{CONST}\ Abs\ Abs \\
| \ & \texttt{ISEMPTY?}\ Abs \\
| \ & \texttt{FST}\ Abs \\
| \ & \texttt{RST}\ Abs \\
| \ & \texttt{APP}\ Abs\ Abs \\
| \ & \texttt{ERROR}
\end{aligned}
$$

$< identificador >$ es cualquier identificador.

3.2. Semántica

La *semántica* nos revela el significado de cadenas de caracteres sintácticamente válidas dentro de un lenguaje, es decir, describe el comportamiento que

una computadora seguirá al ejecutar un programa dentro del lenguaje. Es imprescindible definir de manera formal la semántica de un lenguaje ya que ayuda a evitar errores en el diseño y además es una herramienta de optimización [15].

La semántica debe presentarse de manera dinámica y opcionalmente de forma estática:

- Semántica estática. Se determina si un programa está bien definido mediante criterios sintácticos. Se concluye si hay presencias de variables libres o si son correctos los tipos asignados a los programas.
- Semántica dinámica. Se define el valor o los pasos de evaluación de un programa. Se realiza en tiempo de ejecución.

3.2.1. Semántica estática

En esta vertiente se parte del enfoque sintáctico para determinar si un programa está legalmente construido [15].

A continuación se definirá de manera inductiva la semántica estática del lenguaje `MinTyRacket`, usando juicios de tipo, de la forma $\Gamma \vdash t : \tau$ donde la expresión $\Gamma = \{ \mathrm{x}_1 : \tau_1, x_2 : \tau_2, \ldots , x_n : \tau_n\}$ describe un conjunto finito de variables, llamado *contexto de tipificación*, t es una expresión de sintaxis abstracta y τ es la metavariable que representa a los tipos definidos en el lenguaje de programación [3]. La razón por la que se introduce un contexto de tipificación es que los términos pueden contener abstracciones `LAMBDA` anidadas, por lo que se deben considerar diversas suposiciones.

Primero se define un contexto de tipificación:

$$\begin{aligned} \Gamma ::= {} & \oslash \\ & \mid \Gamma, id : \tau \end{aligned}$$

Esta definición dice que un *contexto de tipificación* Γ es una secuencia de variables y de sus respectivos tipos, la coma indica que la extensión del contexto se realiza por la derecha. El contexto vacío es denotado por $\oslash$, aunque generalmente se omite, por ejemplo $\vdash t : \tau$ denota que el término cerrado t posee tipo τ bajo el conjunto vacío de suposiciones. Para evitar confusión al extender el contexto de tipificación con nombres de variables ya existentes se renombrarán las variables ligadas por las abstracciones `LAMBDA` cuando sea necesario.

Ahora definiremos las reglas de tipificación:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{Regla 1:(T-var)}$$

La Regla 1:(T-var) dice que el tipo asumido por x en Γ es τ.

$$\frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{LAMBDA}\ x\ \tau_1\ t_2 : \tau_1 \to \tau_2} \qquad \text{Regla 2:(T-abs)}$$

La Regla 2:(T-abs) indica que se extiende el contexto de tipificación con el tipo indicado explícitamente en el argumento de la función, si bajo este contexto de tipificación extendido se infiere que t_2 posee tipo τ_2 entonces la expresión `LAMBDA` posee tipo $\tau_1 \to \tau_2$.

$$\frac{\Gamma \vdash t_1 : \tau_{11} \to \tau_{12} \quad \Gamma \vdash t_2 : \tau_{11}}{\Gamma \vdash \texttt{APP}\ t_1\ t_2 : \tau_{12}} \qquad \text{Regla 3:(T-app)}$$

La Regla 3:(T-app) define que si t_1 se evalúa a una función que envía valores de argumentos de tipo τ_{11} a resultados de tipo τ_{12} (bajo la hipótesis de que los valores representados por sus variables libres posean los tipos asumidos en Γ) y si t_2 evalúa a un resultado de tipo τ_{11} entonces si se aplica t_1 a t_2 se obtiene un valor de tipo τ_{12}.

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{LET}\ x\ \tau_1\ t_1\ t_2 : \tau_2} \qquad \text{Regla 4:(T-let)}$$

La Regla 4:(T-let) dice que el tipo de una expresión `LET` puede calcularse a partir del tipo asociado al término ligado de `LET`. Se extiende el contexto de tipificación con una ligadura de tal tipo y sobre el contexto de tipificación extendido se calcula el tipo del cuerpo de `LET`, el cual resulta ser el tipo de toda la expresión `LET`.

$$\frac{\Gamma \vdash t_1 : TipoAritm \quad \Gamma \vdash t_2 : TipoAritm}{\Gamma \vdash \texttt{OP}\ OpAritm\ t_1\ t_2 : TipoAritm} \qquad \text{Regla 5:(T-opAritm)}$$

La Regla 5:(T-opAritm) indica que si se considera el contexto Γ y bajo tal contexto se puede determinar que cada término de la operación aritmética posee tipo $TipoAritm$ entonces el término correspondiente a las operaciones aritméticas tendrá tipo $TipoAritm$.

$$\frac{\Gamma \vdash t_1 : TipoRel \quad \Gamma \vdash t_2 : TipoRel}{\Gamma \vdash \texttt{OP}\ OpRel\ t_1\ t_2 : TipoRel} \qquad \text{Regla 6:(T-opRel1)}$$

La Regla 6:(T-opRel1) dice que si en presencia del contexto Γ se puede determinar que cada término de la operación relacional posee tipo $TipoRel$ entonces el término correspondiente a las operaciones relacionales con operandos de tipo relacional tendrá tipo $TipoRel$.

$$\frac{\Gamma \vdash t_1 : TipoAritm \quad \Gamma \vdash t_2 : TipoAritm}{\Gamma \vdash \texttt{OP}\ OpRel\ t_1\ t_2 : TipoRel} \qquad \text{Regla 7:(T-opRel2)}$$

La Regla 7:(T-opRel2) indica que si se considera el contexto Γ y bajo tal contexto se puede determinar que cada término de la operación relacional posee tipo $TipoAritm$ entonces el término correspondiente a las operaciones relacionales con operandos de tipo aritmético tendrá tipo $TipoRel$.

$$\frac{\Gamma,\ x : \tau \vdash t : \tau}{\Gamma \vdash \texttt{FIX}\ x\ \tau\ t : \tau} \qquad \text{Regla 8:(T-fix)}$$

La Regla 8:(T-fix) dice que se extiende el contexto de tipificación con el tipo indicado explícitamente en el argumento de la función, si bajo este contexto de tipificación extendido se infiere que t posee tipo τ entonces la función recursiva posee tipo τ.

Las características de tipificación que se han desarrollado hasta ahora pueden clasificarse en *tipos base* como booleanos, números y *tipos constructores* como $\rightarrow$ que permiten construir nuevos tipos a partir de otros. A continuación se introduce un tipo constructor llamado `List`. Para cada tipo τ, el tipo `List` τ describe listas de longitud finita cuyos elementos son de tipo `List` τ .

$$\frac{\Gamma \vdash t_1 : \tau\ \ldots\ \Gamma \vdash t_n : \tau}{\Gamma \vdash \texttt{LIST}\ t_1\ \ldots\ t_n : \texttt{List}\ \tau} \qquad \text{Regla 9:(T-list)}$$

La Regla 9:(T-list) define el tipo de una lista verificando que el tipo de sus elementos sea el mismo.

$$\Gamma \vdash \texttt{NULL}\ \tau : \texttt{List}\ \tau \qquad \text{Regla 10:(T-null)}$$

La Regla 10:(T-null) dice que una lista vacía describe una lista de longitud finita en la que sus elementos son de tipo τ.

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \texttt{List}\ \tau}{\Gamma \vdash \texttt{CONST}\ t_1\ t_2 : \texttt{List}\ \tau} \qquad \text{Regla 11:(T-const)}$$

La Regla 11:(T-const) indica el tipo que se obtiene al construir una lista a partir de dos argumentos, si bajo el contexto Γ el tipo del parámetro t_1 es τ y bajo el mismo contexto Γ, el tipo del parámetro t_2 es `List` τ donde τ es el tipo que recibe la expresión `CONST`, entonces el tipo de `CONST` es `List` τ.

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ \tau}{\Gamma \vdash \texttt{ISEMPTY?}\ t_1 : \texttt{boolean}} \qquad \text{Regla 12:(T-isEmpty?)}$$

La Regla 12:(T-isEmpty?) define que para tipificar la expresión `ISEMPTY?` se debe verificar que el tipo del parámetro t_1 es `List` τ , donde τ es el tipo que recibe la expresión `ISEMPTY?`, si es así entonces se evaluará la expresión a un booleano (`boolean`).

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ \tau}{\Gamma \vdash \texttt{FST}\ t_1 : \tau} \qquad \text{Regla 13:(T-fst)}$$

La Regla 13:(T-fst) indica el tipo del primer elemento de una lista, es decir, si el tipo del parámetro t_1 es `List` τ, donde τ es el tipo que aparece en la sintaxis de la expresión `FST`, entonces la expresión completa referente a `FST` tendrá tipo τ.

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ \tau}{\Gamma \vdash \texttt{RST}\ t_1 : \texttt{List}\ \tau} \qquad \text{Regla 14:(T-rst)}$$

La Regla 14:(T-rst) dice el tipo resultante al quitarle el primer elemento a una lista, de modo que si el tipo del parámetro t_1 es `List` τ, donde τ es el tipo aparece en la sintaxis de `RST`, entonces la expresión completa `RST` tendrá tipo `List` τ.

$$\Gamma \vdash \texttt{ERROR} : \texttt{Min} \qquad \text{Regla 15:(T-err)}$$

Debido a que se diseñará subtipificación en `MinTyRacket`, la Regla 15:(T-err) garantiza que la expresión `ERROR` puede ir cambiando de tipo cuando sea necesario. Por esa razón la expresión `ERROR` posee como tipo el subtipo de todos los tipos.

$$\frac{\Gamma \vdash t_1 : \texttt{boolean} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash\ \tau_3 : \tau}{\Gamma \vdash \texttt{IF}\ t_1\ t_2\ t_3 : \tau} \qquad \text{Regla 16:(T-if)}$$

La Regla 16:(T-if) define el tipo de una condicional verificando que el tipo de sus ramas sea el mismo.

$$\frac{\Gamma \vdash t_1 : \texttt{Min} \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{APP}\ t_1\ t_2 : \texttt{Min}} \qquad \text{Regla 17:(T-appMin)}$$

La Regla 17:(T-appMin) determina que si la expresión a la que se le pasa el argumento posee tipo `Min` entonces la aplicación posee tipo `Min`.

3.2.2. Semántica dinámica

Ya establecida la sintaxis de un lenguaje de manera rigurosa, se debe definir formalmente como serán evaluados los términos de dicho lenguaje y determinar que es lo que devuelve la ejecución de un programa , es decir, la semántica del lenguaje. Existen tres enfoques para formalizar la evaluación de un lenguaje de programación:

- Semántica operacional. Se define una *máquina abstracta* para especificar el comportamiento de un lenguaje de programación. Esta máquina toma como entrada a los términos del lenguaje conocidos en este contexto como *estados* y a través de una *función de transición* se completa la definición del comportamiento del lenguaje. El significado de un término t puede considerarse como el estado final que la máquina alcanza al iniciar con t como su estado inicial [15].

- Semántica denotativa. Consiste en encontrar una colección de *dominios semánticos*. Posteriormente encontramos una *función de interpretación* que asigne a cada término elementos de tales dominios. El área que investiga este tipo de semántica es conocida como *teoría de dominios* [15].

- Semántica axiomática. Establece axiomas como la propia definición del lenguaje. El significado de un término consiste sólo en lo que se pueda demostrar acerca de él [15].

En este trabajo se utilizará la semántica operacional debido a su simpleza y flexibilidad para explicar el comportamiento de un programa.

A continuación se usa la técnica usual de definición de una gramática libre de contexto en forma de BNF para definir los estados finales de `MinTyRacket`, donde v es la *metavariable*[1] que los describe.

$$
\begin{aligned}
v ::= \ & \texttt{true} \\
\mid \ & \texttt{false} \\
\mid \ & \texttt{integerV} \\
\mid \ & \texttt{floatV} \\
\mid \ & \texttt{doubleV} \\
\mid \ & \texttt{LIST}\, v_1 \ \ldots \ v_n \\
\mid \ & \texttt{NULL}\, \tau \\
\mid \ & \texttt{LAMBDA}\, x\ \tau\ t
\end{aligned}
$$

Esta definición dice que las constantes booleanas, los números, las abstracciones `LAMBDA` y los términos `NULL` y `LIST` son *valores*, es decir, son estados finales de la semántica operacional de `MinTyRacket`, los cuales permitirán demostrar algunas de sus propiedades fundamentales.

Es importante mencionar que no se ha incluido la expresión `ERROR` en la sintaxis de valores para evitar que se traslapen con las reglas de evaluación de la aplicación, posteriormente se verán las consecuencias de esta situación en las demostraciones de las propiedades de `MinTyRacket`.

[1] Variable que no pertenece a la sintaxis del lenguaje que se define [15].

Para poder definir las reglas de evaluación se va a definir el operador *sustitución*.

Definición 1 *La **sustitución** de un término* `s` *por el término* `n` *dentro de un término t, escrita como* $t[\texttt{n} := \texttt{s}]$ *es definida de la siguiente forma*: [15]

$$(\texttt{VAR x})[\texttt{n} := \texttt{s}] = \begin{cases} \texttt{s} \text{ si } \texttt{x} = \texttt{n} \\ \texttt{x} \text{ en cualquier otro caso} \end{cases}$$

$$(\texttt{IF}\, t_1\, t_2\, t_3)[\texttt{n} := \texttt{s}] = (\texttt{IF}\ t_1[\texttt{n} := \texttt{s}]\ t_2[\texttt{n} := \texttt{s}]\ t_3[\texttt{n} := \texttt{s}])$$

$$(\texttt{LAMBDA x}\,\tau\, t_2)[\texttt{n} := \texttt{s}] = \begin{cases} (\texttt{LAMBDA x}\,\tau\, t_2) \text{ si } \texttt{x} = \texttt{n} \\ (\texttt{LAMBDA x}\,\tau\, t_2[\texttt{n} := \texttt{s}]) \text{ en cualquier otro caso} \end{cases}$$

$$(\texttt{LET x}\,\tau\, t_1\, t_2)[\texttt{n} := \texttt{s}] = \begin{cases} (\texttt{LET x}\,\tau\, t_1\, t_2) \text{ si } \texttt{x} = \texttt{n} \\ (\texttt{LET x}\,\tau\, t_1[\texttt{n} := \texttt{s}]\, t_2[\texttt{n} := \texttt{s}]) \text{ en cualquier otro caso} \end{cases}$$

$$(\texttt{FIX x}\,\tau\, t)[\texttt{n} := \texttt{s}] = \begin{cases} (\texttt{FIX x}\,\tau\, t) \text{ si } \texttt{x} = \texttt{n} \\ (\texttt{FIX x}\,\tau\, t[\texttt{n} := \texttt{s}]) \text{ en cualquier otro caso} \end{cases}$$

$$(\texttt{NULL}\,\tau)[\texttt{n} := \texttt{s}] = (\texttt{NULL}\,\tau)$$

$$(\texttt{LIST}\, t_1\, \ldots\, t_n)[\texttt{n} := \texttt{s}] = (\texttt{LIST}\, t_1[\texttt{n} := \texttt{s}]\, \ldots\, t_n[\texttt{n} := \texttt{s}])$$

$$(\texttt{CONST}\, t_1\, t_2)[\texttt{n} := \texttt{s}] = (\texttt{CONST}\, t_1[\texttt{n} := \texttt{s}]\, t_2[\texttt{n} := \texttt{s}])$$

$$(\texttt{ISEMPTY?}\, t_1)[\texttt{n} := \texttt{s}] = (\texttt{ISEMPTY?}\, t_1[\texttt{n} := \texttt{s}])$$

$$(\texttt{FST}\, t_1)[\texttt{n} := \texttt{s}] = (\texttt{FST}\, t_1[\texttt{n} := \texttt{s}])$$

$$(\texttt{RST}\, t_1)[\texttt{n} := \texttt{s}] = (\texttt{RST}\, t_1[\texttt{n} := \texttt{s}])$$

$$(\texttt{APP}\, t_1\, t_2)[\texttt{n} := \texttt{s}] = (\texttt{APP}\, t_1[\texttt{n} := \texttt{s}]\, t_2[\texttt{n} := \texttt{s}])$$

$$(\texttt{ERROR})[\texttt{n} := \texttt{s}] = (\texttt{ERROR})$$

La operación *sustitución* permitirá a los términos correspondientes de `MinTyRacket` sustituir sus parámetros por valores.

Las funciones de transición se describen a través de las siguientes reglas de evaluación: [15]

$$\frac{t_1 \longrightarrow t_i'}{\texttt{LIST}\ v_1\ \ldots\ v_{i-1}\ t_i\ \ldots\ t_n \longrightarrow \texttt{LIST}\ v_1\ \ldots\ v_{i-1}\ t_i'\ \ldots\ t_n} \qquad \text{Regla 18:(E-list)}$$

La Regla 18:(E-list) dice que para evaluar la expresión `LIST` primero se evalúa cada uno de sus elementos partiendo de izquierda a derecha.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{CONST}\ t_1\ t_2 \longrightarrow \texttt{CONST}\ t_1'\ t_2} \qquad \text{Regla 19:(E-const1)}$$

La Regla 19:(E-const1) dice que para evaluar la expresión `CONST` se debe evaluar primero la expresión t_1.

$$\frac{t_2 \longrightarrow t_2'}{\texttt{CONST}\ v_1\ t_2 \longrightarrow \texttt{CONST}\ v_1\ t_2'}$$ Regla 20:(E-const2)

La Regla 20:(E-const2) indica que después de evaluar la expresión t_1 descrita en la Regla 19:(E-const1), al valor v_1, el paso siguiente es evaluar la expresión t_2 a la expresión t_2'.

$$\frac{}{\texttt{CONST}\ v_1\ \ (\texttt{LIST}\ v_2\ \ldots\ v_n) \longrightarrow \texttt{LIST}\ v_1\ \ldots\ v_n}$$ Regla 21:(E-const3)

La Regla 21:(E-const3) señala que si la expresión `CONST` recibe como argumento un valor v_1 y el valor denotado por la expresión `LIST`, entonces la expresión `CONST` se evaluará a la lista construida por los dos argumentos.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{ISEMPTY?}\ t_1 \longrightarrow \texttt{ISEMPTY?}\ t_1'}$$ Regla 22:(E-isEmpty?1)

La Regla 22:(E-isEmpty?1) define que si la expresión t_1 se evalúa a t_1' entonces la expresión `ISEMPTY?` se evaluará considerando a la expresión t_1' como argumento, es decir, se debe evaluar primero el término t_1.

$$\frac{}{\texttt{ISEMPTY?}\ (\texttt{NULL}\ \tau) \longrightarrow \texttt{true}}$$ Regla 23:(E-isEmpty?2)

La Regla 23:(E-isEmpty?2) postula que si a la expresión `ISEMPTY?` se le pasa como argumento la expresión `NULL` entonces la expresión `ISEMPTY?` se evaluará a `true`.

$$\frac{}{\texttt{ISEMPTY?}\ (\texttt{LIST}\ v_1\ \ldots\ v_n) \longrightarrow \texttt{false}}$$ Regla 24:(E-isEmpty?3)

La Regla 24:(E-isEmpty?3) señala que si la expresión `ISEMPTY?` recibe como argumento la expresión `LIST` entonces la expresión `ISEMPTY?` se evaluará a `false`.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{FST}\ t_1 \longrightarrow \texttt{FST}\ t_1'}$$ Regla 25:(E-fst)

La Regla 25:(E-fst) considera que si la expresión t_1 se evalúa a t_1' entonces la expresión `FST` se evaluará a la expresión `FST` tomando a la expresión t_1' como argumento.

$$\texttt{FST}\ (\texttt{LIST}\ v_1\ \ldots\ v_n) \longrightarrow v_1$$ Regla 26:(E-fstList)

La Regla 26:(E-fstList) indica que la expresión `FST` devuelve el primer elemento de una lista, es decir, se evalúa al valor v_1 si la expresión `FST` recibe como argumento la expresión `LIST`.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{RST}\ t_1 \longrightarrow \texttt{RST}\ t_1'} \qquad \text{Regla 27:(E-rst)}$$

La Regla 27:(E-rst) define que si la expresión t_1 se evalúa a t_1' entonces la expresión `RST` tomará como argumento a la expresión t_1'.

$$\frac{}{\texttt{RST}\ (\texttt{LIST}\ v_1\ v_2\ \ldots\ v_n) \longrightarrow \texttt{LIST}\ v_2\ \ldots\ v_n} \qquad \text{Regla 28:(E-rstList)}$$

La Regla 28:(E-rstList) afirma que la expresión `RST` regresa la lista que queda al eliminar su primer elemento, es decir, se evalúa al valor descrito por la expresión `LIST`, la cual es tomada como argumento de la expresión `RST`.

$$\frac{t_2 \longrightarrow t_2'}{\texttt{OP}\ Op\, v_1\, t_2 \longrightarrow \texttt{OP}\ Op\, v_1\, t_2'} \qquad \text{Regla 29:(E-op1)}$$

La Regla 29:(E-op1) indica que la evaluación se realiza de izquierda a derecha, es decir, que la evaluación de las operaciones primitivas es estricta.

$$\frac{v_1\ v_2}{\texttt{OP}\ \ Op\, v_1\, v_2 \longrightarrow v_1\, Op\, v_2} \qquad \text{Regla 30:(E-op2)}$$

La Regla 30:(E-op2) dice que para evaluar una operación primitiva antes se debe evaluar cada unos de sus operandos hasta conseguir sus valores respectivos para poder aplicar las operaciones elementales, es decir, el lenguaje `MinTyRacket` poseerá como propiedad la evaluación glotona.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{IF}\ \ t_1\ t_2\ t_3 \longrightarrow\ \ \texttt{IF}\ \ t_1'\ t_2\ t_3} \qquad \text{Regla 31:(E-if)}$$

La Regla 31:(E-if) determina una estrategia de evaluación en los valores terminales ya que indica que siempre se debe evaluar la condición del `IF` antes de evaluar $t_2\ \ t_3$.

$$\frac{}{\texttt{IF}\ \ \ \texttt{true}\ t_2\ t_3 \longrightarrow t_2} \qquad \text{Regla 32:(E-ifTrue)}$$

La Regla 32:(E-ifTrue) dice que en caso de que la condición de la expresión `IF` sea la constante `true` entonces se evaluará la expresión t_2.

$$\frac{}{\texttt{IF false}\ t_2\ t_3 \longrightarrow t_3}$$ Regla 33:(E-ifFalse)

La Regla 33:(E-ifFalse) dice que en caso de que la condición de la expresión `IF` sea la constante `false` entonces se evaluará la expresión t_3.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{LET}\ \ x\ \tau\ t_1\ t_2 \longrightarrow \texttt{LET}\ \ x\ \tau\ t_1'\ t_2}$$ Regla 34:(E-let)

La Regla 34:(E-let) dice que si el término t_1 conocido como *término de ligadura* ya que en él se va a sustituir la variable `x` se evalúa a t_1' entonces la expresión `LET` se evaluará a su misma estructura ahora asignando al *término de ligadura* la expresión t_1'.

$$\frac{}{\texttt{LET}\ \ x\ \tau\ v_1\ t_2 \longrightarrow t_2\,[x := v_1]}$$ Regla 35:(E-letV)

La Regla 35:(E-letV) indica que el término de ligadura de `LET` debe ser evaluado antes de que la evaluación del cuerpo del `LET` pueda iniciar, es decir, se debe evaluar el *término ligadura* hasta llegar a un *valor*.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{APP}\ t_1\ t_2 \longrightarrow \texttt{APP}\ t_1'\ t_2}$$ Regla 36:(E-app)

La Regla 36:(E-app) dice que siempre se debe evaluar t_1 antes de evaluar t_2.

$$\frac{t_2 \longrightarrow t_2'}{\texttt{APP}\ v\ t_2 \longrightarrow \texttt{APP}\ v\ t_2'}$$ Regla 37:(E-app2)

La Regla 37:(E-app2) indica que después de evaluar t_1 el siguiente paso es evaluar t_2.

$$\frac{}{\texttt{APP}\ \ (\texttt{LAMBDA}\ x\ \tau\ t_2)\ \ v \longrightarrow t_2\,[x := v]}$$ Regla 38:(E-app3)

La Regla 38:(E-app3) dice que para evaluar toda la expresión `APP` se debe asegurar que t_1 sea una función anónima donde la variable ligada será sustituida por v en la expresión t_2.

$$\texttt{FIX}\ x\ \tau\ t \longrightarrow t\,[x := \texttt{FIX}\ x\ \tau\ t]$$ Regla 39:(E-fix)

La Regla 39:(E-fix) describe la regla de transición de las funciones recursivas donde se sustituye toda la función en su mismo cuerpo.

A continuación se describen las reglas de evaluación acerca del término ERROR.

$$\texttt{APP}\ \ \texttt{ERROR}\ t_2 \longrightarrow \texttt{ERROR} \qquad \text{Regla 40:(E-appError1)}$$

La Regla 40:(E-appError1) dice que si se encuentra el término ERROR mientras se reduce el lado izquierdo de una aplicación a un valor, entonces ERROR es el resultado de nuestra evaluación.

$$\texttt{APP}\ \ v_1\ \ \texttt{ERROR} \longrightarrow \texttt{ERROR} \qquad \text{Regla 41:(E-appError2)}$$

La Regla 41:(E-appError2) dice que después de reducir el lado izquierdo de una aplicación a un valor, en caso de encontrar el término ERROR al tratar de reducir el argumento de tal aplicación, entonces se debe concluir que ERROR es el resultado de la evaluación.

$$\texttt{FST}\ \ \texttt{NULL}\,\tau \longrightarrow \texttt{ERROR} \qquad \text{Regla 42:(E-fstError)}$$

La Regla 42:(E-fstError) dice que si encontramos el término NULL como argumento a la expresión FST se evaluará a ERROR, lo que significa que no se puede obtener el primer elemento de una lista vacía.

$$\texttt{RST}\ \ \texttt{NULL}\,\tau \longrightarrow \texttt{ERROR} \qquad \text{Regla 43:(E-rstError)}$$

La Regla 43:(E-rstError) dice que si encontramos el término NULL como argumento a la expresión RST se evaluará a ERROR, lo que significa que no se puede obtener el resto de lista a partir de una lista que no posee ningún elemento.

$$\texttt{FST}\ \ \texttt{ERROR}\,\tau \longrightarrow \texttt{ERROR} \qquad \text{Regla 44:(E-fstError2)}$$

La Regla 44:(E-fstError2) dice que si el término ERROR es argumento de la expresión FST, entonces se evaluará a ERROR.

$$\texttt{RST}\ \ \texttt{ERROR}\,\tau \longrightarrow \texttt{ERROR} \qquad \text{Regla 45:(E-rstError2)}$$

La Regla 45:(E-rstError2) dice que si el término ERROR es un argumento de la expresión RST, entonces se evaluará a ERROR.

$$\texttt{LIST}\,v_1\ \ldots\ v_{i-1}\,\texttt{ERROR}\ \ldots\ t_n \longrightarrow \texttt{ERROR} \qquad \text{Regla 46:(E-listError)}$$

La Regla 46:(E-listError) dice que la expresión LIST propaga la expresión ERROR.

$$\texttt{CONST}\ \texttt{ERROR}\ t_2 \longrightarrow \texttt{ERROR} \qquad \text{Regla 47:(E-constError1)}$$

La Regla 47:(E-constError1) dice que la expresión `CONST` en caso de que t_1 se evalúe a `ERROR` entonces se propaga la expresión`ERROR`.

$$\texttt{CONST}\ t_1\ \texttt{ERROR} \longrightarrow \texttt{ERROR} \qquad \text{Regla 48:(E-constError2)}$$

La Regla 48:(E-constError2) dice que la expresión `CONST` en caso de que t_2 se evalúe a `ERROR` entonces se propaga la expresión `ERROR`.

$$\texttt{OP}\ \ Op\ \texttt{ERROR}\ t_2 \longrightarrow \texttt{ERROR} \qquad \text{Regla 49:(E-op2Error1)}$$

La Regla 49:(E-op2Error1) dice que para evaluar una operación primitiva si el primer operando se evalúa a `ERROR` entonces se propaga la expresión `ERROR`.

$$\texttt{OP}\ \ Op\ t_1\ \texttt{ERROR} \longrightarrow \texttt{ERROR} \qquad \text{Regla 50:(E-op2Error)}$$

La Regla 50:(E-op2Error) dice que para evaluar una operación primitiva si el segundo operando se evalúa a `ERROR` entonces se propaga la expresión `ERROR`.

$$\texttt{IF}\quad \texttt{ERROR}\ t_2\ t_3 \longrightarrow \texttt{ERROR} \qquad \text{Regla 51:(E-ifError)}$$

La Regla 51:(E-ifError) dice que en caso de que la condición de la expresión `IF` sea la expresión `ERROR` entonces se propagará dicha expresión.

$$\texttt{IF}\quad \texttt{true}\ \texttt{ERROR}\ t_3 \longrightarrow \texttt{ERROR} \qquad \text{Regla 52:(E-ifTrueError)}$$

La Regla 52:(E-ifTrueError) dice que en caso de que la condición de la expresión `IF` sea la constante `true` y la expresión t_2 sea la expresión `ERROR` entonces se propagará dicha expresión.

$$\texttt{IF}\quad \texttt{false}\ t_2\ \texttt{ERROR} \longrightarrow \texttt{ERROR} \qquad \text{Regla 53:(E-ifFalseError)}$$

La Regla 53:(E-ifFalseError) dice que en caso de que la condición de la expresión `IF` sea la constante `false` y la expresión t_3 sea la expresión `ERROR` entonces se propagará dicha expresión.

$$\texttt{FIX}\ x\ \tau\ \texttt{ERROR} \longrightarrow \texttt{ERROR} \qquad \text{Regla 54:(E-fixError)}$$

La Regla 54:(E-fixError) describe la regla de transición de las funciones recursivas cuando el cuerpo es la expresión `ERROR`, de manera que se propaga.

$$\frac{}{\texttt{LET}\ \ x\ \tau\ \texttt{ERROR}\ t_2 \longrightarrow \texttt{ERROR}}$$ Regla 55:(E-letError1)

La Regla 55:(E-letError1) indica la propagación de la expresión `ERROR` cuando el *término de ligadura* de `LET` es la expresión `ERROR`.

$$\texttt{LET}\ \ x\ \tau\ t_1\ \texttt{ERROR} \longrightarrow \texttt{ERROR}$$ Regla 56:(E-letError2)

La Regla 56:(E-letError2) define la propagación de la expresión `ERROR` cuando el *cuerpo* de `LET` es la expresión `ERROR`.

$$\texttt{ISEMPTY?}\ \texttt{ERROR} \longrightarrow \texttt{ERROR}$$ Regla 57:(E-isEmpty?Error)

La Regla 57:(E-isEmpty?Error) postula que si a la expresión `ISEMPTY?` se le pasa como argumento la expresión `ERROR` entonces la expresión `ISEMPTY?` propaga la expresión `ERROR`.

3.3. Subtipificación

Basados en la Sección 2.4 y utilizando la notación $A <: B$ para indicar que A es un subtipo de B, en otras palabras, se indica que cualquier expresión de tipo A puede ser usada en cualquier contexto donde una expresión que posee tipo B es esperada [15]. A continuación se definirán las reglas de subtipificación de `MinTyRacket`:

$$\frac{}{\texttt{Min} <: \tau}$$ Regla 58:(Sub-Min)

La Regla 58:(Sub-Min) indica que el tipo `Min` es subtipo de todo tipo definido en el lenguaje.

$$\frac{}{\tau <: \texttt{Max}}$$ Regla 59:(Sub-Max)

La Regla 59:(Sub-Max) dice que el tipo `Max` es supertipo de todo tipo definido en el lenguaje.

$$\frac{}{\texttt{float} <: \texttt{double}}$$ Regla 60:(Sub-float)

La Regla 60:(Sub-float) determina que es seguro usar el tipo `double` en el lugar que ocupaba el tipo `float`.

$$\frac{}{\texttt{integer} <: \texttt{double}}$$ Regla 61:(Sub-integer1)

La Regla 61:(Sub-integer1) define que que es seguro usar el tipo `double` en el lugar que ocupaba el tipo `integer`.

$$\frac{}{\texttt{integer} <: \texttt{float}}$$ Regla 62:(Sub-integer2)

La Regla 62:(Sub-integer2) indica que es seguro usar el tipo `float` en el lugar que ocupaba el tipo `integer`.

El tipo `integer` posee dos supertipos que contribuyen a implementar transitividad dentro de las reglas de subtipificación, además de generar subderivaciones en la verificación de tipos que reducen el tiempo de interpretación.

A continuación se agregarán reglas referentes al tipo `boolean`, `integer`, `float` y `double` que posteriormente se utilizarán para demostrar que la propiedad reflexiva puede ser derivada de éstas.

$$\frac{}{\texttt{boolean} <: \texttt{boolean}}$$ Regla 63:(Sub-boolean)

La Regla 63:(Sub-boolean) dice que el tipo `boolean` es subtipo de sí mismo.

$$\frac{}{\texttt{integer} <: \texttt{integer}}$$ Regla 64:(Sub-integer3)

La Regla 64:(Sub-integer3) indica que el tipo `integer` es subtipo de sí mismo.

$$\frac{}{\texttt{float} <: \texttt{float}}$$ Regla 65:(Sub-float2)

La Regla 65:(Sub-float2) define que el tipo `float` es subtipo de sí mismo.

$$\frac{}{\texttt{double} <: \texttt{double}}$$ Regla 66:(Sub-double)

La Regla 66:(Sub-double) dice que el tipo `double` es subtipo de sí mismo.

$$\frac{\sigma_1 <: \tau_1}{\texttt{List}\ \sigma_1 <: \texttt{List}\ \tau_1}$$ Regla 67:(Sub-list)

Sean σ_1 τ_1 dos tipos definidos por la categoría $TipoElem$, la Regla 67:(Sub-list) determina que un tipo `List` σ_1 es un subtipo de `List` τ_1 si el tipo σ_1 es un subtipo de τ_1.

$$\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \qquad \text{Regla 68:(Sub-arr)}$$

La Regla 68:(Sub-arr) indica que un tipo de una función $\sigma_1 \to \sigma_2$ es subtipo de $\tau_1 \to \tau_2$ si τ_1 es un subtipo de σ_1 y τ_2 es un supertipo de σ_2.

$$\frac{\Gamma \vdash t : \sigma \quad \sigma <: \tau}{\Gamma \vdash t : \tau} \qquad \text{Regla 69:(Subsunción)}$$

La Regla 69:(Subsunción) dice que si σ es un subtipo de τ, entonces cada elemento t de tipo σ es también un elemento t de tipo τ.

$$\sigma <: \sigma \qquad \text{Regla 70:(Sub-ref)}$$

La Regla 70:(Sub-ref) declara que todo tipo es un subtipo de sí mismo.

$$\frac{\sigma <: \varrho \quad \varrho <: \tau}{\Gamma \vdash \sigma <: \tau} \qquad \text{Regla 71:(Sub-trans)}$$

La Regla 71:(Sub-trans) define que si un tipo σ es un subtipo de un tipo ϱ y ϱ es un subtipo de un tipo τ entonces el tipo σ es un subtipo del tipo τ. Por ejemplo, supongamos que existe una expresión t de tipo `integer`, con base en las reglas de subtipificación de `MinTyRacket` se tiene tanto que `integer` es un subtipo del tipo `float` como que `float` es un subtipo de `double`, entonces la Regla 71:(Sub-trans) determina que el tipo `integer` es subtipo del tipo `double`.

La implementación hasta este punto de `MinTyRacket` no es posible ya que la Regla 69:(Subsunción), la Regla 70:(Sub-ref) y la Regla 71:(Sub-trans) poseen variables en el consecuente que pueden ser aplicadas a cualquier expresión definida en el lenguaje, como consecuencia toda implementación de subtipificación y tipificación basada en estas reglas no sabría cual de las dos reglas usar.

La razón por la cual la Regla 69:(Subsunción) no se puede implementar es que en su conclusión la expresión se especifica con la metavariable t, traslapando las conclusiones de las otras reglas de tipificación. Si se considera la Regla 1:(T-var) se puede notar que solo se puede aplicar a variables, análogamente la Regla 2:(T-abs) solo se aplica a la expresión `LAMBDA`, mientras la Regla 69:(Subsunción) aplica a cualquier expresión, es decir no es *sintácticamente directa* [15].

La Regla 70:(Sub-ref) presenta un problema similar a la Regla 69:(Subsunción), en este caso traslapando conclusiones de reglas de subtipificación [15].

La Regla 71:(Sub-trans) además de presentar un problema similar a la Regla 70:(Sub-ref) y la Regla 69:(Subsunción) usa en sus premisas la metavariable υ, la cual no aparece en la conclusión por lo que en la implementación se debería adivinar que tipo es υ, situación que tiene pocas posibilidades de éxito [15].

3.4. Subtipificación algorítmica

Con el objetivo de solucionar los problemas para la implementación de `MinTyRacket` se introducirán reglas de tipificación y subtipificación conocidas como *relaciones algorítmicas de tipificación* y relaciones *algorítmicas de subtipificación* respectivamente que son sintácticamente directas. Posteriormente se creará un *algoritmo de subtipificación* que trabaja con tipos, dicho algoritmo nos indicará si un tipo es subtipo de otro.

La función del *algoritmo de subtipificación* consiste en decidir si la sentencia $\sigma <: \tau$ es derivable a partir de las reglas de subtipificación definidas. En otras palabras, verifica si (σ,τ) pertenece a la relación de subtipificación, escrito como $\mapsto \sigma <: \tau$, (σ algorítmicamente es un subtipo de τ), lo cual es definido de tal manera que decidir si (σ,τ) pertenece a la relación de subtipificación pueda determinarse solamente a partir de la definición de los tipos.

A continuación se describirán las reglas algorítmicas de subtipificación del lenguaje `MinTyRacket`: [15]

$$\mapsto \sigma <: \texttt{Max} \qquad \text{Regla 72:(SA-max)}$$

La Regla 72:(SA-max) indica a través del algoritmo de subtipificación si `Max` es un supertipo de σ.

$$\frac{\mapsto \tau_1 <: \sigma_1 \quad \mapsto \sigma_2 <: \tau_2}{\mapsto \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \qquad \text{Regla 73:(SA-fun)}$$

La Regla Regla 73:(SA-fun) dice que a través del algoritmo de subtipificación es posible determinar el supertipo de un tipo correspondiente a una función.

$$\mapsto \texttt{Min} <: \tau \qquad \text{Regla 74:(SA-Min)}$$

La Regla 74:(SA-Min) indica a través del algoritmo de subtipificación si `Min` es un subtipo de τ.

$$\mapsto \texttt{float} <: \texttt{float} \qquad \text{Regla 75:(SA-float)}$$

La Regla 75:(SA-float) indica a través del algoritmo de subtipificación que `float` es un subtipo de sí mismo.

$$\mapsto \texttt{float} <: \texttt{double} \qquad \text{Regla 76:(SA-float2)}$$

La Regla 76(SA-float2) dice que `float` es un subtipo del tipo `double`.

$$\mapsto \texttt{integer} <: \texttt{integer} \qquad \text{Regla 77:(SA-integer)}$$

La Regla 77:(SA-integer) define a través del algoritmo de subtipificación que `integer` es un subtipo de sí mismo.

$$\mapsto \texttt{integer} <: \texttt{double} \qquad \text{Regla 78:(SA-integer2)}$$

La Regla 78:(SA-integer2) determina a través del algoritmo de subtipificación que `integer` es subtipo de `double`.

$$\mapsto \texttt{integer} <: \texttt{float} \qquad \text{Regla 79:(SA-integer3)}$$

La Regla 79:(SA-integer3) indica a través del algoritmo de subtipificación que `integer` es subtipo de `float`.

$$\mapsto \texttt{boolean} <: \texttt{boolean} \qquad \text{Regla 80:(SA-boolean)}$$

La Regla 80:(SA-boolean) señala a través del algoritmo de subtipificación que `boolean` es supertipo de sí mismo.

$$\frac{\mapsto \sigma_1 <: \tau_1}{\mapsto \texttt{List}\ \sigma_1 <: \texttt{List}\ \tau_1} \qquad \text{Regla 81:(SA-list)}$$

La Regla 81:(SA-list) define algorítmicamente el supertipo de una expresión `List`.

3.5. Tipificación algorítmica

Ya que se han establecido las reglas algorítmicas de subtipificación correspondientes a la definición declarativa de las reglas de subtipificación haremos lo análogo con la definición declarativa de las reglas de tipificación definiendo las reglas algorítmicas de tipificación de `MinTyRacket`:

$$\frac{x : \tau \in \Gamma}{\Gamma \mapsto x : \tau} \qquad \text{Regla 82:(TA-var)}$$

La Regla 82:(TA-var) dice que el tipo asumido por x en un contexto Γ usando el algoritmo de tipificación es τ.

$$\frac{\Gamma, x : \tau_1 \mapsto t_2 : \tau_2}{\Gamma \mapsto \texttt{LAMBDA}\ \ x\ \tau_1\ t_2 : \tau_1 \to \tau_2} \qquad \text{Regla 83:(TA-abs)}$$

La Regla 83:(TA-abs) dice que se extiende el contexto de tipificación Γ con el tipo indicado explícitamente en el argumento de la función, si bajo este contexto de tipificación extendido $\Gamma, x : \tau_1$ es posible inferir algorítmicamente que t_2 posee tipo τ_2 entonces la expresión `LAMBDA` algorítmicamente posee tipo $\tau_1 \to \tau_2$.

$$\frac{\Gamma \mapsto t_1 : \tau_1 \quad \tau_1 = \tau_{11} \to \tau_{12} \quad \Gamma \mapsto t_2 : \tau_2 \quad \mapsto \ \tau_2 <: \tau_{11}}{\Gamma \mapsto \texttt{APP}\ t_1\ t_2 : \tau_{12}} \qquad \text{Regla 84:(TA-app)}$$

La premisa $\tau_1 = \tau_{11} \to \tau_{12}$ en la Regla 84:(TA-app) representa un orden en la secuencia de operaciones durante la verificación de tipos para la aplicación de funciones. Primero se calcula un tipo τ_1 para t_1, posteriormente se verifica si τ_1 posee la forma $\tau_{11} \to \tau_{12}$, si τ_1 la tiene, entonces se calcula el tipo τ_2 de la expresión t_2. Por último, si τ_2 es subtipo de τ_{11} se determina algorítmicamente que la expresión `APP` posee tipo τ_{12}.

$$\frac{\Gamma \mapsto t_1 : \tau_1 \quad \Gamma, x : \tau_1 \mapsto t_2 : \tau_2}{\Gamma \mapsto \texttt{LET}\ \ x\, \tau_1\ t_1\, t_2 : \tau_2} \qquad \text{Regla 85:(TA-let)}$$

La Regla 85:(TA-let) dice que para verificar el tipo de una expresión `LET`, se extiende el contexto de tipificación Γ con una ligadura de tal tipo y sobre el contexto de tipificación extendido $\Gamma, x : \tau_1$ calculamos el tipo del cuerpo de `LET`, el cual resulta ser el tipo de toda la expresión `LET`.

$$\frac{\Gamma,\ x : \tau \mapsto t : \tau}{\Gamma \mapsto \texttt{FIX}\ x\ \tau\ t\ : \tau} \qquad \text{Regla 86:(TA-fix)}$$

La Regla 86:(TA-fix) dice que para verificar el tipo de la función recursiva se extiende el contexto de tipificación Γ con el tipo indicado explícitamente en el argumento de la función, si bajo este contexto de tipificación extendido $\Gamma, x : \tau$ se infiere que t posee tipo τ entonces la función recursiva posee tipo τ.

$$\frac{\Gamma \mapsto t_1 : \tau\ \ \ldots\ \ \Gamma \mapsto t_n : \tau}{\Gamma \mapsto \texttt{LIST}\ t_1\ \ \ldots\ \ t_n : \tau} \qquad \text{Regla 87:(TA-list)}$$

La Regla 87:(TA-list) define algorítmicamente el tipo de una lista verificando que el tipo de sus elementos sea el mismo.

$$\Gamma \mapsto \texttt{NULL}\ \tau : \texttt{List}\ \ \tau \qquad \text{Regla 88:(TA-null)}$$

La Regla 88:(TA-null) define algorítmicamente que una lista vacía describe una lista de longitud finita en la que sus elementos son de tipo τ

$$\frac{\Gamma \mapsto t_1 : \tau \quad \Gamma \mapsto t_2 : \texttt{List}\ \ \tau}{\Gamma \mapsto \texttt{CONST}\ t_1\ t_2 : \texttt{List}\ \ \tau} \qquad \text{Regla 89:(TA-const)}$$

La Regla 89:(TA-const) indica algorítmicamente que el tipo de la expresión `CONST` se obtiene al construir una lista a partir de dos argumentos, si bajo el contexto Γ el tipo del parámetro t_1 es τ y bajo el mismo contexto Γ, el tipo del parámetro t_2 es `List` τ donde τ es el tipo que recibe la expresión `CONST`, entonces el tipo de `CONST` es `List` τ.

$$\frac{\Gamma \mapsto t_1 : \texttt{List}\ \ \tau}{\Gamma \mapsto \texttt{ISEMPTY?}\ t_1 : \texttt{boolean}} \qquad \text{Regla 90:(TA-isEmpty?)}$$

La Regla 90:(TA-isEmpty?) define que para tipificar algorítmicamente la expresión `ISEMPTY?`, se debe verificar que el tipo del parámetro t_1 es `List` τ , donde τ es el tipo que recibe la expresión `ISEMPTY?`, entonces se evaluará la expresión a un booleano (`boolean`).

$$\frac{\Gamma \mapsto t_1 : \texttt{List } \tau}{\Gamma \mapsto \texttt{FST } t_1 : \tau} \quad \text{Regla 91:(TA-fst)}$$

La Regla 91:(TA-fst) indica algorítmicamente el tipo asociado a la expresión `FST`, es decir, define el tipo del primer elemento de una lista, de manera que si el tipo del parámetro t_1 es `List` τ, donde τ es el tipo que aparece en la sintaxis de la expresión `FST`, entonces la expresión completa referente a `FST` tendrá tipo τ.

$$\frac{\Gamma \mapsto t_1 : \texttt{List } \tau}{\Gamma \mapsto \texttt{RST } t_1 : \texttt{List } \tau} \quad \text{Regla 92:(TA-rst)}$$

La Regla 92:(TA-rst) determina algorítmicamente el tipo que posee la expresión completa `RST`, es decir, determina el tipo resultante al quitarle el primer elemento a una lista, de modo que si el tipo del parámetro t_1 es `List` τ, donde τ es el tipo aparece en la sintaxis de `RST`, entonces la expresión completa `RST` tendrá tipo `List` τ.

$$\Gamma \mapsto \texttt{ERROR} : \texttt{Min} \quad \text{Regla 93:(TA-err)}$$

La Regla 93:(TA-err) determina algorítmicamente que el tipo que posee la expresión `ERROR` es `Min`, el cual es subtipo de todos los tipos definidos en el lenguaje.

Para asignarle tipo a la expresión condicional de `MinTyRacket` se utilizará un tipo conocido como la **conjunción de las posibilidades de flujo condicional** (en inglés *join*), ya que cuando se introduce la subtipificación existen diversas formas de asignar el mismo tipo a las ramas de una expresión condicional. La idea es calcular el *tipo mínimo* tanto de la rama de la expresión `then` como de la expresión `else` para así poder determinar el mínimo supertipo común, conocido como *join* [15].

Definición 2 *Un tipo* `J` *es llamado un* ***join*** *de un par de tipos* σ *y* τ*, denotado como* $\sigma \vee \tau =$ `J` *si* $\sigma <:$ `J`*,* $\tau <$ `J` *y para todo tipo* υ *si* $\sigma <: \upsilon$ *y* $\tau < \upsilon$ *entonces* `J` $< \upsilon$ *[15].*

Basados en la Definición 2 y en la subtipificación, se definirá la siguiente regla algorítmica para la expresión `IF`:

$$\frac{\Gamma \mapsto t_1 : \tau_1 \quad \tau_1 = \texttt{boolean} \quad \Gamma \mapsto t_2 : \tau_2 \quad \Gamma \mapsto \ t_3 : \tau_3 \quad \tau_2 \vee \tau_3 = \tau}{\Gamma \mapsto \texttt{IF } t_1 \ t_2 \ t_3 : \tau} \quad \text{Regla 94:(TA-if)}$$

La Regla 94:(TA-if) permite determinar el tipo de la expresión condicional a través de calcular el mínimo supertipo común de sus ramas. El objetivo de esta regla es rechazar menos programas válidos semánticamente.

Ahora se definirán la siguientes reglas de tipificación algorítmicas para la expresión `OP`:

$$\frac{\Gamma \mapsto t_1 : TipoAritm \quad \Gamma \mapsto t_2 : TipoAritm \qquad \tau_1 \vee \tau_2 = \tau}{\Gamma \mapsto \texttt{OP}\ OpAritm\ t_1\ t_2 : TipoAritm}\ \text{Regla 95:(TA-opAritm)}$$

La Regla 95:(TA-opAritm) señala que si se considera el contexto Γ y bajo tal contexto se puede determinar que cada operando posee tipo $TipoAritm$ y si el mínimo supertipo común de los tipos asociados a los operandos es $TipoAritm$ entonces el término `OP` tendrá tipo $TipoAritm$.

$$\frac{\Gamma \mapsto t_1 : TipoRel \quad \Gamma \mapsto t_2 : TipoRel \qquad \tau_1 \vee \tau_2 = \tau}{\Gamma \mapsto \texttt{OP}\ OpRel\ t_1\ t_2 : TipoRel}\ \text{Regla 96:(TA-opRel1)}$$

La Regla 96:(TA-opRel1) dice que si se supone el contexto Γ y bajo tal contexto se puede concluir que cada operando posee tipo $TipoRel$, además si el mínimo supertipo común de los tipos asociados a los operandos es $TipoRel$ entonces el término `OP` tendrá tipo $TipoRel$.

$$\frac{\Gamma \mapsto t_1 : TipoAritm \quad \Gamma \mapsto t_2 : TipoAritm \qquad \tau_1 \vee \tau_2 = \tau}{\Gamma \mapsto \texttt{OP}\ OpRel\ t_1\ t_2 : TipoRel}\ \text{Regla 97:(TA-opRel2)}$$

La Regla 97:(TA-opRel2) señala que si se considera el contexto Γ y bajo tal contexto se puede determinar que cada operando posee tipo $TipoAritm$ y si el mínimo supertipo común de los tipos asociados a los operandos es $TipoAritm$ entonces el término `OP` tendrá tipo $TipoRel$.

Por último se definirá la regla de tipificación algorítmica para la expresión `Min`:

$$\frac{\Gamma \mapsto t_1 : \tau_1 \qquad \tau_1 = \texttt{Min} \qquad \Gamma \mapsto t_2 : \tau_2}{\Gamma \mapsto t_1\ t_2 : \texttt{Min}}\ \text{Regla 98:(TA-appMin)}$$

La Regla 98:(TA-appMin) determina algorítmicamente si una aplicación posee tipo `Min` para evitar traslapar la conclusión de la regla declarativa correspondiente a la aplicación.

3.6. Seguridad de MinTyRacket

Para demostrar la propiedad de **seguridad** de `MinTyRacket`, debido a que se presenta subtipificación, primero se demostrará la seguridad que aportan las

reglas de tipificación y subtipificación declarativas y las reglas de evaluación de `MinTyRacket`. Posteriormente se demostrará la equivalencia que existe entre las reglas de tipificación declarativas y las reglas de tipificación algorítmicas.

Para demostrar la seguridad que aportan las reglas de tipificación y subtipificación declarativas y las reglas de evaluación de `MinTyRacket` se debe considerar la propiedad de **preservación**, la cual establece que las transiciones de evaluación conservan los tipos, mientras la propiedad de **progreso** asegura que todas las expresiones bien tipificadas o son valores (en tal caso su evaluación termina con éxito) o en caso de que no sean valores seguirán evaluándose, es decir, la evaluación no puede detenerse en un estado para el cuál no exista una transición de evaluación posible. Para demostrar dichas propiedades se deben considerar las siguientes definiciones: [15]

Definición 3 *Un **término** se denomina **cerrado**, si no contiene variables libres.*

Definición 4 *Un **valor** se denomina **cerrado**, si es un término cerrado.*

Posteriormente se debe demostrar el siguiente lema, el cual dice consiste en conjunto de propiedades que indican que cada forma sintáctica bien tipificada presenta subtérminos de formas específicas.

Lema 5 ***(Inversión)***

1. *Si* $\Gamma \vdash$ `x` $: \omega$, *entonces* `x` $: \omega \in \Gamma$.
2. *Si* $\Gamma \vdash$ `IF` $t_1\ t_2\ t_3 : \omega$, *entonces* $\Gamma \vdash t_1 :$ `boolean` *y* $\Gamma \vdash t_2\ , t_3 : \omega$.
3. *Si* $\Gamma \vdash$ `LAMBDA` `x` $\tau\ t : \omega_1$, *entonces* $\omega_1 = \tau \to \omega_2$ *para algún* ω_2 *tal que* Γ , `x` $: \tau \vdash t : \omega_2$.
4. *Si* $\Gamma \vdash$ `APP` $t_1\ t_2 : \omega$, *entonces existe un* τ *tal que* $\Gamma \vdash t_1 : \tau \to \omega$ *y* $\Gamma \vdash t_2 : \tau$.
5. *Si* $\Gamma \vdash$ `FIX` `x` $\tau\ t : \omega$, *entonces* Γ , `x` $: \omega \vdash t : \omega$.
6. *Si* $\Gamma \vdash$ `NULL` $\tau : \omega$, *entonces* $\omega =$ `List` τ.
7. *Si* $\Gamma \vdash$ `CONST` $t_1\ t_2 : \omega$, *entonces* $\omega =$ `List` τ *tal que* $\Gamma \vdash t_1 : \tau$ *y* $\Gamma \vdash t_2 :$ `List` τ.
8. *Si* $\Gamma \vdash$ `LET` `x` $t_1\ t_2 : \omega_2$, *entonces* $\omega_2 = \tau_2$ *para algún* ω_1 *tal que* $\Gamma \vdash t_1 : \omega_1$ *y* Γ , `x` $: \omega_2 \vdash t : \omega_1$.
9. *Si* $\Gamma \vdash$ `ISEMPTY?` $t_1 : \omega$ *entonces* $\omega =$ `boolean`.
10. *Si* $\Gamma \vdash$ `LIST` $t_1\ \ldots\ t_n : \omega$ *entonces* $\omega =$ `List` τ *tal que* $\Gamma \vdash t_1 : \tau\ \ldots\ \Gamma \vdash t_n : \tau$.
11. *Si* $\Gamma \vdash$ `FST` $t_1 : \omega$ *entonces* $\omega = \tau$ *tal que* $\Gamma \vdash t_1 :$ `List` τ.
12. *Si* $\Gamma \vdash$ `RST` $t_1 : \omega$ *entonces* $\omega =$ `List` τ *tal que* $\Gamma \vdash t_1 :$ `List` τ.

13. *Si* $\Gamma \vdash$ `ERROR` $: \omega$ *entonces* $\omega =$ `Min`.
14. *Si* $\Gamma \vdash$ `OP` $OpAritm\ t_1\ t_2 : \omega$ *entonces* $\omega = OpAritm$ *tal que* $\Gamma \vdash t_1 : OpAritm$ $\Gamma \vdash t_2 : OpAritm$.

Demostración.
Cada caso se sigue inmediatamente de la definición de reglas de tipificación de `MinTyRacket`. ■

A continuación se indicarán las formas posibles que puede poseer un valor de acuerdo al tipo que posee.

Lema 6 ***(Formas canónicas)***

1. *Si* `v` *es un valor de tipo* `boolean`*, entonces* `v` *solo puede ser una de dos formas, ó es la expresión* `true` *ó es la expresión* `false`.
2. *Si* `v` *es un valor de tipo* $\tau_1 \to \tau_2$ *entonces* `v` $=$ `LAMBDA x` $\tau_1\ t_2$.
3. *Si* `v` *es un valor de tipo* `List` τ *entonces* `v` *solo puede tomar una de dos formas posibles, ó es* `NULL` *ó es* `LIST` $v_1 \ldots v_n$.
4. *Si* `v` *es un valor de tipo* `integer` *entonces* `v` *es un valor entero definido en el lenguaje.*
5. *Si* `v` *es un valor de tipo* `float` *entonces* `v` *es un valor flotante definido en el lenguaje.*
6. *Si* `v` *es un valor de tipo* `double` *entonces* `v` *es un valor real definido en el lenguaje.*

Demostración.
Caso 1. Si `v` es un valor de tipo `boolean`, entonces de acuerdo al **Lema 4** (***Inversión***) se puede inferir que `v` puede ser la expresión `ISEMPTY?` t_1 y por definición, `v` puede tomar sólo una de las expresiones `true` y `false`. Los casos respectivos a los otros valores no pueden suceder.

Caso 2. Si `v` es un valor de tipo $\tau_1 \to \tau_2$ entonces considerando el **Lema 4** (***Inversión***) se concluye que `v` $=$ `LAMBDA` τ_1 `x` t_2. Similarmente al caso 1, estas son las únicas formas que `v` puede tomar.

Caso 3. Si `v` es un valor de tipo `List` τ, tomando en cuenta el **Lema 4** (***Inversión***) se puede determinar que `v` solo puede tomar una de dos formas posibles, ó es `v` $=$ `NULL` τ ó es `v` $=$ `LIST` $t_1 \ldots t_n$. Análogamente al caso 1, son las únicas formas que `v` puede tomar.

Caso 4. Si `v` es un valor de tipo `integer` entonces por definición `v` es un valor entero en el lenguaje.

Caso 5. Si `v` es un valor de tipo `float` entonces por definición `v` es un valor flotante en el lenguaje.

Caso 6. Si `v` es un valor de tipo `double` entonces por definición `v` es un valor real en el lenguaje. ■

Se utilizará el **Lema** 6 (***Formas canónicas***) y la **Definición** 3 (páginas 43 y 44 respectivamente) para demostrar el **Teorema** 7 (***Progreso***).

Teorema 7 *(**Progreso**). Suponiendo que t sea un término cerrado y bien tipificado, es decir, $t : \tau$ para algún τ, entonces t es un valor, $t =$* `ERROR` *ó existe t' tal que $t \longrightarrow t'$*

Es importante notar que debido a que se agregó el término `ERROR` en el intérprete de `MinTyRacket` se ha agregado al **Teorema 6** (***Progreso***) el caso en que el término sea una salida emergente dentro de un programa interpretado en este lenguaje [15].

Demostración.
Se aplicará inducción estructural sobre las derivaciones de tipos.

Caso (Regla 1:(T-var)) No aplica ya que t es un término cerrado.

Caso (Regla 2:(T-abs)) Se sigue inmediatamente ya que `LAMBDA` es un valor.

Caso (Regla 3:(T-app))
Sean $t =$ `APP` $t_1\ t_2 \qquad \Gamma \vdash t_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash t_2 : \tau_{11} \qquad \tau = \tau_{12}$
Por hipótesis de inducción, t_1 es un valor ó existe una regla de evaluación que puede tomar ó en otro caso es la expresión `ERROR`; situación similar a t_2. Si t_1 puede seguir una regla de evaluación entonces la Regla 36:(E-app) se aplica a t. En caso de que t_1 sea un valor y t_2 pueda seguir una regla de evaluación entonces la Regla 37:(E-app2) se asigna a t. Si tanto t_1 como t_2 son valores, el **Lema 5** (***Formas canónicas***) dice que t_1 tiene la forma `LAMBDA x` $\sigma_{11}\ t_{12}$ por lo que se puede aplicar la Regla 38:(E-app3) a t. Si t_1 es `ERROR` entonces se puede aplicar la Regla 40:(E-appError1) y finalmente si t_2 es `ERROR` entonces se puede aplicar la Regla 41:(E-appError2).

Caso (Regla 4:(T-let))
Sean $t =$ `LET` $x\ \tau\ t_1\ t_2 \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash t_2 : \tau_2$
Por hipótesis de inducción t_1 puede ser un valor ó puede seguir una regla de evaluación ó ser la expresión `ERROR`. En caso de que a t_1 se pueda aplicar una regla de evaluación entonces a t se puede aplicar la Regla 34:(E-let). En caso de que t_1 sea un valor entonces se aplica la Regla 35:(E-letV) a t. Finalmente si t_1 es `ERROR` entonces es posible aplicar la Regla 55:(E-letError1). Para el caso en que t_2 sea `ERROR` entonces se puede aplicar la Regla 56:(E-letError2).

Caso (Regla 5:(T-opAritm))
Sean $t = \texttt{OP}\ OpAritm\ t_1\ t_2$ $\Gamma \vdash t_1 : TipoAritm$ $\Gamma \vdash t_2 : TipoAritm$
Por hipótesis de inducción, sea $1 \leq i \leq 2$ entonces se pueden dar tres casos, ó t_i sea un valor ó t_i puede seguir alguna regla de evaluación ó t_i puede ser la expresión `ERROR`. En caso de que t_i pueda seguir alguna regla de evaluación entonces la Regla 29:(E-op1) se puede aplicar a t. Por otro lado, si t_i es un valor entonces la Regla 30:(E-op2) se puede asignar a t. Finalmente si t_i es la expresión `ERROR` se puede aplicar la Regla 49:(E-opError1) o Regla 50:(E-opError2) según sea el caso.

Caso (Regla 6:(T-opRel1))
Sean $t = \texttt{OP}\ OpRel\ t_1\ t_2$ $\Gamma \vdash t_1 : TipoRel$ $\Gamma \vdash t_2 : TipoRel$
Por hipótesis de inducción, sea $1 \leq i \leq 2$ entonces se pueden dar tres casos, ó t_i sea un valor ó t_i puede seguir alguna regla de evaluación ó t_i puede ser la expresión `ERROR`. En caso de que t_i pueda seguir alguna regla de evaluación entonces la Regla 29:(E-op1) se puede aplicar a t. Por otro lado, si t_i es un valor entonces la Regla 30:(E-op2) se puede asignar a t. Finalmente si t_i es la expresión `ERROR` se puede aplicar la Regla 49:(E-opError1) o Regla 50:(E-opError2) según sea el caso.

Caso (Regla 7:(T-opRel2))
Sean $t = \texttt{OP}\ OpRel\ t_1\ t_2$ $\Gamma \vdash t_1 : TipoAritm$ $\Gamma \vdash t_2 : TipoAritm$
Por hipótesis de inducción, sea $1 \leq i \leq 2$ entonces se pueden dar tres casos, ó t_i sea un valor ó t_i puede seguir alguna regla de evaluación ó t_i puede ser la expresión `ERROR`. En caso de que t_i pueda seguir alguna regla de evaluación entonces la Regla 29:(E-op1) se puede aplicar a t. Por otro lado, si t_i es un valor entonces la Regla 30:(E-op2) se puede asignar a t. Finalmente si t_i es la expresión `ERROR` se puede aplicar la Regla 49:(E-opError1) o Regla 50:(E-opError2) según sea el caso.

Caso (Regla 8:(T-fix))
Sean $t = \texttt{FIX}\ x\ \tau\ t_1$ $\Gamma,\ x : \tau \vdash t_1 : \tau$
Ya sea que t_1 sea un valor ó pueda tomar una regla de evaluación t siempre puede tomar la Regla 39:(E-fix). Si t_1 es la expresión `ERROR` se puede aplicar la Regla 54:(E-fixError).

Caso (Regla 9:(T-list))
Sea $t = \texttt{LIST}\ t_1\ \ldots\ t_n$
Por hipótesis de inducción, sea $1 \leq i \leq$ entonces se pueden dar tres casos, ó t_i es un valor ó t_i puede seguir alguna regla de evaluación ó t_i es la expresión `ERROR`. En caso de que t_i pueda seguir alguna regla de evaluación entonces la Regla 18:(E-list) se puede aplicar a t. Por otro lado, si t_i para todo $1 \leq i \leq$ es un valor, ya que $\texttt{LIST}\ v_1\ \ldots\ v_n$ es un valor de `MinTyRacket` entonces el teorema se cumple. Si la expresión t_i es es la expresión `ERROR` se puede aplicar la Regla 46:(E-listError).

Caso (Regla 10:(T-null))
Sea $t = \texttt{NULL}\ \tau$
Se sigue inmediatamente ya que NULL es un valor.

Caso (Regla 11:(T-const))
Sean $t = \texttt{CONST}\ t_1\ t_2 \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \texttt{List}\ \tau$
Por hipótesis de inducción, ó t_1 es un valor, en otro caso existe una regla de evaluación que puede tomar ó t_1 es la expresión expresión ERROR; situación similar a t_2. Si t_1 puede seguir una regla de evaluación entonces la Regla 19:(E-const1) se aplica a t. En caso de que t_1 sea un valor y t_2 pueda seguir una regla de evaluación entonces la Regla 20:(E-const2) se asigna a t. Finalmente, si tanto t_1 como t_2 son valores, se puede aplicar la Regla 21:(E-const3). Finalmente si t_1 es la expresión error se puede aplicar la Regla 47:(E-constError1), mientras si t_2 es la expresión error se puede aplicar la Regla 48:(E-constError2).

Caso (Regla 12:(T-isEmpty?))
Sean $t = \texttt{ISEMPTY?}\ t_1 \quad \Gamma \vdash t_1 : \texttt{List}\ \tau$
Por hipótesis de inducción t_1 ó es un valor ó puede seguir una regla de evaluación ó ser la expresión ERROR. En caso de que a t_1 se pueda aplicar una regla de evaluación entonces a t se puede aplicar la Regla 22:(E-isEmpty?1). En caso de que t_1 sea un valor entonces el **Lema 5** (***Formas canónicas***) dice que t_1 puede tomar sólo dos formas, una de ellas es $\texttt{NULL}\ \tau$ y la otra es $\texttt{LIST}\ v_1\ \ldots\ v_n$. En caso de que t_1 sea de la forma $\texttt{NULL}\ \tau$ entonces se aplica la Regla 23:(E-isEmpty?2) a t. Finalmente, si t_1 es de la forma $\texttt{LIST}\ v_1\ \ldots\ v_n$ se puede aplicar la Regla 24:(E-isEmpty?3) a t. Finalmente si t_1 es ERROR se puede aplicar la Regla 57:(E-isEmpty?Error).

Caso (Regla 13:(T-fst))
Sean $t = \texttt{FST}\ t_1 \quad \Gamma \vdash t_1 : \texttt{List}\ \tau$
Por hipótesis de inducción t_1 ó es un valor ó puede seguir una regla de evaluación ó ser la expresión ERROR. En caso de que a t_1 se pueda aplicar una regla de evaluación entonces a t se puede aplicar la Regla 25:(E-fst). En caso de que t_1 sea un valor entonces el **Lema 5** (***Formas canónicas***) dice que t_1 puede tomar sólo dos formas, una de ellas es $\texttt{NULL}\ \tau$ y la otra es $\texttt{LIST}\ v_1 \ldots\ v_n$. En caso de que t_1 sea de la forma $\texttt{NULL}\ \tau$ entonces se aplica la Regla 42:(E-fstError) a t. Finalmente, si t_1 es de la forma $\texttt{LIST}\ v_1 \ldots\ v_n$ se puede aplicar la Regla 26:(E-fstList) a t. Por último si t_1 es la expresión ERROR se puede aplicar la regla Regla 44:(E-fstError2).

Caso (Regla 14:(T-rst))
Sean $t = \texttt{RST}\ t_1 \quad \Gamma \vdash t_1 : \texttt{List}\ \tau$
Por hipótesis de inducción t_1 ó es un valor ó puede seguir una regla de evaluación ó ser la expresión ERROR. Si a t_1 se pueda aplicar una regla de evaluación, entonces a t se puede aplicar la Regla 27:(E-rst). En caso de que t_1 sea un valor entonces el **Lema 5** (***Formas canónicas***) dice que t_1 puede tomar sólo dos formas, una de ellas es $\texttt{NULL}\ \tau$ y la otra es $\texttt{LIST}\ v_1 \ldots\ v_n$. En caso de que t_1

sea de la forma `NULL` τ entonces se aplica la Regla 43:(E-rstError) a t. Finalmente, si t_1 es de la forma `LIST` $v_1 \ldots v_n$ se puede aplicar la Regla 28:(E-rstList) a t. Por último si t_1 es la expresión `ERROR` se puede aplicar la Regla 45:(E-rstError2).

Caso (Regla 15:(T-err))
Sea $t =$ `ERROR`
Se sigue inmediatamente por el enunciado del teorema que se desea demostrar.

Caso (Regla 16:(T-if))
Sean $t =$ `IF` $t_1\ t_2\ t_3 \quad \Gamma \vdash t_1 :$ `boolean` $\quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash\ \tau_3 : \tau$
Por hipótesis de inducción se tiene que ó t_1 es un valor ó existe alguna regla de evaluación que puede aplicarse a t_1 ó t_1 es la expresión `ERROR`. En caso de que exista alguna regla de evaluación que puede aplicarse a t_1 entonces se aplica la Regla 31:(E-if) a t. Finalmente, si t_1 es un valor, debido al **Lema 5** (***Formas canónicas***) t_1 puede tomar sólo dos formas. Si la forma que toma t_1 es `true` entonces se aplica la Regla 32:(E-ifTrue) a t. Mientras si la forma de t_1 es `false` entonces se puede asignar la Regla 33:(E-ifFalse) a t. En caso de que se presente la expresión `ERROR` en alguno de los términos se pueden aplicar Regla 51:(E-ifError), Regla 52:(E-ifTrueError) ó Regla 53:(E-ifFalseError) según corresponda.

Caso (Regla 17:(T-appMin))
Sean $t =$ `APP` $t_1\ t_2 \quad \Gamma \vdash t_1 :$ `Min` $\quad \Gamma \vdash t_2 : \tau_2 \quad \tau =$ `Min`
Por hipótesis de inducción, ó t_1 es un valor, en otro caso existe una regla de evaluación que puede tomar; situación similar a t_2. Si t_1 puede seguir una regla de evaluación entonces la Regla 36:(E-app) se aplica a t. En caso de que t_1 sea un valor y t_2 pueda seguir una regla de evaluación entonces la Regla 37:(E-app2) se asigna a t. Considerando el diseño de `MinTyRacket`, existe el caso en que t_1 es el término `ERROR`, por lo que se puede aplicar la Regla 40:(E-appError1) a t. Por otro lado, si el término t_1 es un valor y t_2 es el término `ERROR` entonces se puede asignar la Regla 41:(E-appError2) a t. ■

Antes de demostrar que un término bien tipificado se evaluará de acuerdo a la semántica dinámica y que dicho término al que se evalúa es también bien tipificado, es necesario demostrar que los términos bien tipificados siguen conservando dicha propiedad cuando las variables son sustituidas por términos de tipos apropiados.

Lema 8 *(**Sustitución**). Si* $\Gamma,\ x : \sigma \vdash t : \tau$ *y además* $\Gamma \vdash s : \sigma$, *entonces* $\Gamma \vdash t[\ x := s\ ] : \tau$

Demostración. Utilizando inducción estructural sobre la derivación de tipos.

Caso (Regla 1:(T-var))
Sean $\quad$ `t` $=$ `u`, `u` $: \tau \in (\Gamma\ ,$ `x` $: \sigma)$

SubCaso ($\mathtt{u} = \mathtt{x}$). Se sigue que $\mathtt{u}[\ \mathtt{x} := \mathtt{s}\ ] = \mathtt{s}$. Si se consideran las hipótesis del lema es posible inferir que $\Gamma \vdash \mathtt{s} : \sigma$, por lo que el lema se cumple.

SubCaso ($\mathtt{u} \neq \mathtt{x}$). Se sigue que $\mathtt{u}[\ \mathtt{x} := \mathtt{s}\ ] = \mathtt{u}$, como $\Gamma \vdash \mathtt{u} : \sigma$ entonces el lema se cumple.

Caso (Regla 16:(T-if))
Sean $\mathtt{t} = \mathtt{IF}\ t_1\ t_2\ t_3$
$\Gamma\ ,\ \mathtt{x} : \sigma \vdash t_1 : \mathtt{boolean}$
$\Gamma\ ,\ \mathtt{x} : \sigma \vdash t_2 : \tau$
$\Gamma\ ,\ \mathtt{x} : \sigma \vdash t_3 : \tau$

Usando la hipótesis inductiva sobre t_1, t_2, t_3 se concluye que:
$\Gamma \vdash t_1[\ \mathtt{x} := \mathtt{s}\ ] = \mathtt{boolean}$
$\Gamma \vdash t_2[\ \mathtt{x} := \mathtt{s}\ ] = \tau$
$\Gamma \vdash t_3[\ \mathtt{x} := \mathtt{s}\ ] = \tau$
Usando ahora la (Regla 16:(T-if)) entonces $\Gamma \vdash \mathtt{t} : \tau$, por lo tanto el lema se cumple.

Caso (Regla 2:(T-abs))
Sean $\mathtt{t} = \mathtt{LAMBDA}\ \mathtt{z}\ \tau_2\ \mathtt{t}_1$
$\tau = \tau_2 \to \tau_1$
$\Gamma, \mathtt{x} : \sigma, \mathtt{z} : \tau_2 \vdash \mathtt{t}_1 : \tau_1$

Analizando las derivaciones dadas se tiene que:
$\Gamma, \mathtt{z} : \tau_2, \mathtt{x} : \sigma \vdash t_1 : \tau_1$.
También se puede determinar que:
$\Gamma \vdash \mathtt{s} : \sigma$.
Por lo que se obtiene:
$\Gamma, \mathtt{z} : \tau_2 \vdash \mathtt{s} : \sigma$.
Si se usa la hipótesis de inducción entonces:
$\Gamma, \mathtt{z} : \tau_2 \vdash t_1[\ \mathtt{x} := \mathtt{s}\ ] : \tau_1$.

Aplicando la (Regla 2:(T-abs)) se tiene que:
$\mathtt{LAMBDA}\ \mathtt{z} : \tau_2\ t_1[\ \mathtt{x} := \mathtt{s}\ ] : \tau_2 \to \tau_1$
Por lo tanto si se considera que $\mathtt{t}[\ \mathtt{x} := \mathtt{s}\ ] = \mathtt{LAMBDA}\ \mathtt{z}\ \tau_2\ t_1[\ \mathtt{x} := \mathtt{s}\ ]$ entonces el lema se cumple.

Caso (Regla 3:(T-app))
Sean $\mathtt{t} = \mathtt{APP}\ t_1\ t_2$
$\Gamma, \mathtt{x}{:}\sigma \vdash t_1 : \tau_2 \to \tau_1$
$\Gamma, \mathtt{x}{:}\sigma \vdash t_2 : \tau_2$
$\tau = \tau_1$

Por hipótesis de inducción se obtienen dos hechos:
$\Gamma \vdash t_1[\ \mathtt{x} := \mathtt{s}\ ] : \tau_2 \to \tau_1$

$\Gamma \vdash t_2[\ \texttt{x} := \texttt{s}\ ] : \tau_2$

Usando la (Regla 3:(T-app)):
$\Gamma \vdash t_1[\ \texttt{x} := \texttt{s}\ ]\ t_2[\ \texttt{x} := \texttt{s}\ ] : \tau$ es decir, $(t_1\ t_2)[\ \texttt{x} := \texttt{s}\ ] : \tau$.

Caso (Regla 4:(T-let))
Sean $\texttt{t} = \texttt{LET z}\ \tau_1\ t_1\ t_2$
Γ, $\texttt{x}{:}\sigma \vdash t_1 : \tau_1$
Γ, $\texttt{x}{:}\sigma$, $\texttt{y}{:}\tau_1 \vdash t_2 : \tau_2$
$\tau = \tau_2$

Analizando las derivaciones Γ, $\texttt{x}{:}\sigma \vdash t_1 : \tau_1$ y Γ, $\texttt{x}{:}\sigma$, $\texttt{y}{:}\tau_1 \vdash t_2 : \tau_2$
Γ, $\texttt{y} : \tau_1$, $\texttt{x} : \sigma \vdash t_2 : \tau_2$.
Por lo tanto en la derivación $\Gamma \vdash \texttt{s} : \sigma$ se infiere que:
Γ, $\texttt{y} : \tau_1 \vdash \texttt{s} : \sigma$.
Por hipótesis de inducción y definición de sustitución se tiene que:
Γ, $\texttt{y} : \tau_1 \vdash t_2[\ \texttt{x} := \texttt{s}\ ] : \tau_2$.
$\Gamma \vdash t_1[\ \texttt{x} := \texttt{s}\ ] : \tau_1$.

Si ahora se aplica la (Regla 4:(T-let)):
$\Gamma \vdash \texttt{LET x}\ \tau_1\ t_1[\ \texttt{x} := \texttt{s}\ ]\ t_2[\ \texttt{x} := \texttt{s}\ ] : \tau_2$, es decir, $\texttt{t}[\ \texttt{x} := \texttt{s}\ ]$, lo cual es lo que se deseaba demostrar.

Caso (Regla 5:(T-opAritm))
Sean $\texttt{t} = \texttt{OP}\ OpAritm\ t_1\ t_2$
τ=$TipoAritm$
Γ , $\texttt{x} : \sigma \vdash t_1 : TipoAritm$
Γ , $\texttt{x} : \sigma \vdash t_2 : TipoAritm$

Aplicando la hipótesis de inducción en cada operando de la premisa
Γ , $\texttt{x} : \sigma \vdash t_1 : TipoAritm$
se puede determinar que:
$\Gamma \vdash t_1[\ \texttt{x} := \texttt{s}\ ] : TipoAritm$, así como que $\Gamma \vdash t_2[\ \texttt{x} := \texttt{s}\ ] : TipoAritm$.
Usando ahora la (Regla 5:(T-opAritm)) se concluye que:
$\Gamma \vdash \texttt{t}[\ \texttt{x} := \texttt{s}\ ] : \tau$ que es lo que se deseaba demostrar.

Caso (Regla 6:(T-opRel1))
Sean $\texttt{t} = \texttt{OP}\ OpRel\ t_1\ t_2$
τ=$TipoRel$
Γ , $\texttt{x} : \sigma \vdash t_1 : TipoRel$
Γ , $\texttt{x} : \sigma \vdash t_2 : TipoRel$

Aplicando la hipótesis de inducción en cada operando de la premisa
Γ , $\texttt{x} : \sigma \vdash t_1 : TipoRel$
se puede determinar que:
$\Gamma \vdash t_1[\ \texttt{x} := \texttt{s}\ ] : TipoRel$, así como que $\Gamma \vdash t_2[\ \texttt{x} := \texttt{s}\ ] : TipoRel$.

Usando ahora la (Regla 6:(T-opRel1)) se concluye que:
$\Gamma \vdash \texttt{t}[\ \texttt{x} := \texttt{s}\ ] : \tau$ que es lo que se deseaba demostrar.

Caso (Regla 7:(T-opRel2))
Sean $\texttt{t} = \texttt{OP}\ OpRel\ t_1\ t_2$
τ=$TipoRel$
$\Gamma\ ,\ \texttt{x} : \sigma \vdash t_1 : TipoAritm$
$\Gamma\ ,\ \texttt{x} : \sigma \vdash t_2 : TipoAritm$

Aplicando la hipótesis de inducción en cada operando de la premisa
$\Gamma\ ,\ \texttt{x} : \sigma \vdash t_1 : TipoAritm$
se puede determinar que:
$\Gamma \vdash t_1[\ \texttt{x} := \texttt{s}\ ] : TipoAritm$, así como que $\Gamma \vdash t_2[\ \texttt{x} := \texttt{s}\ ] : TipoAritm$.
Usando ahora la (Regla 7:(T-opRel2)) se concluye que:
$\Gamma \vdash \texttt{t}[\ \texttt{x} := \texttt{s}\ ] : \tau$ que es lo que se deseaba demostrar.

Caso (Regla 8:(T-fix)):
Sean $\texttt{t} = \texttt{FIX}\ \texttt{z}\ \tau_1\ t_1$
$\tau = \tau_1$
$\Gamma, \texttt{x} : \sigma, \texttt{z} : \tau_1 \vdash t_1 : \tau_1$

Analizando las derivaciones dadas se tiene que:
$\Gamma, \texttt{z} : \tau_1, \texttt{x} : \sigma \vdash t_1 : \tau_1$.
También se puede determinar que $\Gamma \vdash \texttt{s} : \sigma$, por lo tanto:
$\Gamma, \texttt{z} : \tau_1 \vdash \texttt{s} : \sigma$.
Usando la hipótesis de inducción:
$\Gamma, \texttt{z}\ \tau_1 \vdash t_1[\ \texttt{x} := \texttt{s}\ ] : \tau_1$.
Aplicando ahora la (Regla 8:(T-fix)) se determina que:
$\texttt{FIX}\ \texttt{z}\ \tau_1\ t_1[\ \texttt{x} := \texttt{s}\ ] : \tau_1$. Es decir,
$\texttt{t}[\ \texttt{x} := \texttt{s}\ ] = \texttt{FIX}\ \texttt{z}\ \tau_1\ t_1[\ \texttt{x} := \texttt{s}\ ]$ por lo que el lema se cumple.

Caso (Regla 9:(T-list))
Sean $\texttt{t} = \texttt{LIST}\ t_1\ \ldots t_n$
τ=$\texttt{List}\ \tau_1$
$\Gamma\ ,\ \texttt{x} : \sigma \vdash t_1 : \tau_1 \ldots \Gamma\ ,\ \texttt{x} : \sigma \vdash t_n : \tau_1$

Aplicando la hipótesis de inducción en cada operando de la premisa:
$\Gamma\ ,\ \texttt{x} : \sigma \vdash t_1 : \tau_1 \ldots \Gamma\ ,\ \texttt{x} : \sigma \vdash t_n : \tau_1$
se puede determinar que:
$\Gamma \vdash t_1[\ \texttt{x} := \texttt{s}\ ] : \tau_1 \ldots \Gamma \vdash t_n[\ \texttt{x} := \texttt{s}\ ] : \tau_1$.
Usando ahora la (Regla 9:(T-list)) se concluye que:
$\Gamma \vdash \texttt{t}[\ \texttt{x} := \texttt{s}\ ] : \tau$ que es lo que se deseaba demostrar.

Caso (Regla 10:(T-null))
Sean $\texttt{t} = \texttt{NULL}\ \tau_1$

τ = `List` τ_1

Considerando que `NULL` τ_1 [`x` := `s`] = `NULL` τ_1, y considerando por reglas de tipificación, $\Gamma \vdash$ `NULL` τ_1: `List` τ_1 se sigue que $\Gamma \vdash$ `t`[`x` := `s`] : τ y por lo tanto el lema se cumple.

Caso (Regla 11:(T-const)):
Sean `t` = `CONST` t_1 t_2
τ = `List` τ_1
Γ , `x` : $\sigma \vdash t_1 : \tau_1$
Γ , `x` : $\sigma \vdash t_2$: `List` τ_1.

Aplicando la hipótesis de inducción en las premisas se concluyen dos cosas:
$\Gamma \vdash t_1$[`x` := `s`] : τ_1
$\Gamma \vdash t_2$[`x` := `s`] : `List` τ_1.
Usando la (Regla 11:(T-const)) se obtiene $\Gamma \vdash$ `t`[`x` := `s`] : τ que es lo que se deseaba demostrar.

Caso (Regla 12:(T-isEmpty?)):
Sean `t` = `ISEMPTY?` t_1
τ = `boolean`
Γ , `x` : $\sigma \vdash t_1$: `List` τ_1.

Aplicando la hipótesis de inducción en las premisas se concluye que:
$\Gamma \vdash t_1$[`x` := `s`] : `List` τ_1.
Por lo tanto se se aplica la (Regla 12:(T-isEmpty?)) entonces:
$\Gamma \vdash$ `t`[`x` := `s`] : τ que es lo que se quería demostrar.

Caso (Regla 13:(T-fst)):
Sean `t` = `FST` t_1
$\tau = \tau_1$
Γ , `x` : $\sigma \vdash t_1$: `List` τ_1.

Aplicando la hipótesis de inducción en las premisas se concluye que:
$\Gamma \vdash t_1$[`x` := `s`] : `List` τ_1.
Aplicando ahora la (Regla 13:(T-fst)) se concluye que:
$\Gamma \vdash$ `t`[`x` := `s`] : τ que es lo que se deseaba demostrar.

Caso (Regla 14:(T-rst)):
Sean `t` = `RST` t_1
τ = `List` τ_1
Γ , `x` : $\sigma \vdash t_1$: `List` τ_1.

Aplicando la hipótesis de inducción en las premisas se determina que:
$\Gamma \vdash t_1$[`x` := `s`] : `List` τ_1.
Usando la (Regla 14:(T-rst)) se concluye que:

$\Gamma \vdash$ `t`[`x` := `s`] : τ que es lo que se quería demostrar.

Caso (Regla 15:(T-err))
Sean `t` = `ERROR`
τ = `Min`
Considerando que `ERROR`[`x` := `s`]= `ERROR`, ya que `ERROR`: `Min` entonces:
$\Gamma \vdash$ `t`[`x` := `s`] : τ.

Caso (Regla 69:(Subsunción)):
Sean `t` = t_1
$\tau = \tau_1$
Γ , `x` : $\sigma \vdash t_1 : \sigma_1$.
$\sigma_1 <: \tau_1$

Aplicando la hipótesis de inducción en las premisas se determina que:
$\Gamma \vdash t_1$[`x` := `s`] : σ_1.
Por lo que si considerando además que una de las premisas es $\sigma_1 <: \tau_1$ se puede usar la (Regla 69:(Subsunción)) y por lo tanto:
$\Gamma \vdash$ `t`[`x` := `s`] : τ que es lo que se quería demostrar. ■

Se utilizará el **Lema** 8 (***Sustitución***) para demostrar que un término bien tipificado, el cual se evalúa de acuerdo a la semántica dinámica seguirá siendo bien tipificado.

Teorema 9 *(**Preservación**). Si* $\Gamma \vdash t : \tau$ *y* $t \longrightarrow t'$ *entonces* $\Gamma \vdash t' : \tau$.

Demostración. Se aplicará inducción estructural sobre las derivaciones de tipos.

Caso (Regla 1:(T-var)) Por vacuidad se cumple ya que t no posee reglas de evaluación.

Caso (Regla 2:(T-abs)) Se sigue inmediatamente ya que `LAMBDA` es un valor.

Caso (Regla 3:(T-app))
Sean $t =$ `APP` $t_1\ t_2$ $\Gamma \vdash t_1 : \tau_{11} \to \tau_{12}$ $\Gamma \vdash t_2 : \tau_{11}$ $\tau = \tau_{12}$

SubCaso (Regla 36:(E-app))
Sean $t_1 \longrightarrow t_1'$ $t' = t_1'\ t_2$
Por hipótesis de inducción en t_1, $\Gamma \vdash t_1' : \tau_{11} \to \tau_{12}$, además por hipótesis $\Gamma \vdash t_2 : \tau_{11}$. Si se aplica a estos hechos la (Regla 3:(T-app)) entonces $\Gamma \vdash t' : \tau_{12}$, lo cual es lo que se deseaba demostrar.

SubCaso (Regla 37:(E-app2))
Sean $t_2 \longrightarrow t_2'$ $t' = v_1\ t_2'$
Por hipótesis de inducción en t_2, $\Gamma \vdash t_2' : \tau_{11}$, además por hipótesis se concluye

que $\Gamma \vdash t_1 : \tau_{11} \to \tau_{12}$. Si se aplica a estos hechos la (Regla 3:(T-app)) entonces $\Gamma \vdash t' : \tau_{12}$, lo cual es lo que se deseaba demostrar.

SubCaso (Regla 38:(E-app3))
Sean $t_1 = \texttt{LAMBDA x } \sigma_{11}\ t_{12}$ $t_2 = v_2$ $t' = t_{12}[\ \texttt{x} := v_2\ ]$
Analizando las premisas se tiene que $\tau_{11} <: \sigma_{11}$ y $\Gamma, \texttt{x} : \sigma_{11} \vdash t_{12} : \tau_{12}$ por lo que usando la Regla 69:(Subsunción) entonces $\Gamma \vdash t_2 : \sigma_{11}$. Por el **Lema 7** (***Sustitución***) se concluye que $\Gamma \vdash t' : \tau_{12}$.

Caso (Regla 4:(T-let))
Sean $t = \texttt{LET}\ x\ \tau_1\ t_1\ t_2$ $\Gamma \vdash t_1 : \tau_1$ $\Gamma, x : \tau_1 \vdash t_2 : \tau_2$ $\tau = \tau_2$

SubCaso (Regla 34:(E-let))
Sean $t_1 \longrightarrow t_1'$ $t' = \texttt{LET}\ x\ \tau_1\ t_1'\ t_2$
Por hipótesis de inducción se tiene que $\Gamma \vdash t_1' : \tau_1$. Considerando las hipótesis descritas en el Caso (Regla 4:(T-let)) y aplicando la (Regla 4:(T-let)) se concluye que $\texttt{LET}\ x\ \tau_1\ t_1'\ t_2$ posee tipo τ_2. Debido a que $t' = \texttt{LET}\ x\ \tau_1\ t_1'\ t_2 : \tau_2$ entonces $t' : \tau_2$, que es lo que se quería demostrar.

SubCaso (Regla 35:(E-letV))
Sean $t_1 = v_1$ $t' = t_2\,[x := v_1]$
Por premisa $t_1 = v_1$ y como $\Gamma \vdash t_1 : \tau_1$ entonces $\Gamma \vdash v_1 : \tau_1$. Por otro lado se tiene que $\Gamma, x : \tau_1 \vdash t_2 : \tau_2$. Utilizando el **Lema 7** (***Sustitución***) se obtiene que $\Gamma \vdash t_2[\text{x} := v_1] : \tau_2$ por lo que el teorema se cumple.

Caso (Regla 5:(T-opAritm))
Sean $t = \texttt{OP}\ OpAritm\ t_1\ t_2$ $\Gamma \vdash t_1 : TipoAritm$
$\Gamma \vdash t_2 : TipoAritm$ $\tau = TipoAritm$

SubCaso (Regla 29:(E-op1))
Sean $t_2 \longrightarrow t_2'$ $t_1 = v_1$
Utilizando la premisa $\Gamma \vdash t_2 : TipoAritm$ y usando la hipótesis de inducción se infiere que $\Gamma \vdash t_2' : TipoAritm$. Considerando además las hipótesis descritas, si aplicamos la (Regla 5:(T-opAritm)) se obtiene que bajo el contexto Γ la expresión denotada por $\texttt{OP}\ op\ v_1\ t_2'$ posee tipo $TipoAritm$ que es lo que se quería demostrar.

Subcaso (Regla 30:(E-op2))
Sean $t_i = v_i$ para $1 \leq \text{i} \leq 2$
Considerando además las hipótesis descritas en caso (Regla 5:(T-opAritm)) se tiene que $\Gamma \vdash t_i : TipoAritm$ con $1 \leq \text{i} \leq 2$. Ya que $t_i = v_i$ para $1 \leq \text{i} \leq 2$ entonces $\Gamma \vdash v_i : TipoAritm$ considerando que $1 \leq \text{i} \leq 2$. Por lo que si se aplica la hipótesis de inducción y la (Regla 5:(T-opAritm)) se concluye que la expresión $\Gamma \vdash v_1\ OpAritm\ v_2 : TipoAritm$, es decir, se cumple el teorema.

Caso (Regla 6:(T-opRel1))

Sean $t = \texttt{OP}\ OpRel\ t_1\ t_2$ $\Gamma \vdash t_1 : TipoRel$
$\Gamma \vdash t_2 : TipoRel$ $\tau = TipoRel$

SubCaso (Regla 29:(E-op1))
Sean $t_2 \longrightarrow t_2'$ $t_1 = v_1$

Utilizando la premisa $\Gamma \vdash t_2 : TipoRel$ y usando la hipótesis de inducción se infiere que $\Gamma \vdash t_2' : TipoRel$. Considerando además las hipótesis descritas, si aplicamos la (Regla 6:(T-opRel1)) se puede concluir que bajo el contexto Γ la expresión denotada por $\texttt{OP}\ opRel\ v_1\ t_2'$ posee tipo $TipoRel$ que es lo que se quería demostrar.

Subcaso (Regla 30:(E-op2))
Sean $t_i = v_i$ para $1 \leq \text{i} \leq 2$
Considerando además las hipótesis descritas en caso (Regla 6:(T-opRel1)) se tiene que $\Gamma \vdash t_i : TipoRel$ con $1 \leq \text{i} \leq 2$. Ya que $t_i = v_i$ para $1 \leq \text{i} \leq 2$ entonces se cumple $\Gamma \vdash v_i : TipoRel$ considerando que $1 \leq \text{i} \leq 2$. Por lo que si se aplica la hipótesis de inducción y la (Regla 6:(T-opRel1)) se concluye que $\Gamma \vdash v_1\ OpRel\ v_2 : OpRel$, es decir, se cumple el teorema.

Caso (Regla 7:(T-opRel2))
Sean $t = \texttt{OP}\ OpRel\ t_1\ t_2$ $\Gamma \vdash t_1 : TipoAritm$
$\Gamma \vdash t_2 : TipoAritm$ $\tau = TipoRel$

SubCaso (Regla 29:(E-op1))
Sean $t_2 \longrightarrow t_2'$ $t_1 = v_1$

Utilizando la premisa $\Gamma \vdash t_2 : TipoAritm$ y usando la hipótesis de inducción se infiere que $\Gamma \vdash t_2' : TipoAritm$. Considerando además las hipótesis descritas, si aplicamos la (Regla 7:(T-opRel2)) se puede concluir que bajo el contexto Γ la expresión denotada por $\texttt{OP}\ opRel\ v_1\ t_2'$ posee tipo $TipoRel$ que es lo que se quería demostrar.

Subcaso (Regla 30:(E-op2))
Sean $t_i = v_i$ para $1 \leq \text{i} \leq 2$
Considerando además las hipótesis descritas en caso (Regla 7:(T-opRel2)) se tiene que $\Gamma \vdash t_i : TipoAritm$ con $1 \leq \text{i} \leq 2$. Ya que $t_i = v_i$ para $1 \leq \text{i} \leq 2$ entonces se cumple $\Gamma \vdash v_i : TipoAritm$ considerando que $1 \leq \text{i} \leq 2$. Por lo que si se aplica la hipótesis de inducción y la (Regla 7:(T-opRel2)) es posible concluir que $\Gamma \vdash v_1\ OpRel\ v_2 : OpRel$, es decir, se cumple el teorema.

Caso (Regla 16:(T-if))
Sean $t = \texttt{IF}\ t_1\ t_2\ t_3$ $t_1 : \texttt{boolean}$ $t_2 : \tau$ $t_3 : \tau$

Subcaso (Regla 32:(E-ifTrue))
Sean $t_1 = \texttt{true}$ $t' = t_2$
Utilizando las hipótesis del Caso (Regla 16:(T-if)) se obtiene $t_2 : \tau$, por lo tanto

$t' : \tau$, por lo que el teorema se cumple.

Subcaso (Regla 33:(E-ifFalse))
Sean $t_1 = \texttt{false}$ $t' = t_3$
Utilizando las hipótesis del Caso (Regla 16:(T-if)) obtenemos que $t_3 : \tau$, entonces $t' : \tau$, por lo que el teorema se cumple.

Subcaso (Regla 31:(E-if))
Sean $t_1 \longrightarrow t_1'$ $t' = \texttt{IF}\ t_1'\ t_2\ t_3$
Considerando las hipótesis del caso (Regla 16:(T-if)) se tiene que $t_1 : \texttt{boolean}$, por hipótesis de inducción $t' : \texttt{boolean}$. Si además se aplican las hipótesis restantes del caso (Regla 16:(T-if)) entonces se puede aplicar la regla de tipificación (Regla 16:(T-if)) y se obtiene que $\texttt{IF}\ t_1'\ t_2\ t_3 : \tau$, es decir, $t' : \tau$, por lo que el teorema se cumple.

Caso (Regla 10:(T-null))
Sean $t = \texttt{NULL}\ \tau$ $\tau = \texttt{List}\ \tau$
Se cumple por vacuidad ya que no existen reglas de evaluación.

Caso (Regla 11:(T-const))
Sean $t = \texttt{CONST}\ t_1\ t_2$ $\Gamma \vdash t_1 : \tau_1$ $\Gamma \vdash t_2 : \texttt{List}\ \tau_1$ $\tau = \texttt{List}\ \tau_1$

Subcaso (Regla 19:(E-const1))
Sean $t_1 \longrightarrow t_1'$ $t' = \texttt{CONST}\ t_1'\ t_2$
Por hipótesis de inducción $\Gamma \vdash t_1' : \tau_1$. Si se aplica la (Regla 11:(T-const)) vemos que $\texttt{CONST}\ t_1'\ t_2 : \texttt{List}\ \tau_1$, es decir, $t' : \texttt{List}\ \tau_1$ por lo que el teorema se cumple.

Subcaso (Regla 20:(E-const2))
Sean $t_2 \longrightarrow t_2'$ $t' = \texttt{CONST}\ v_1\ t_2'$
Por hipótesis de inducción $\Gamma \vdash t_2' : \texttt{List}\ \tau_1$ y $\Gamma \vdash v_1 : \tau_1$ Aplicando ahora la (Regla 11:(T-const)) se tiene que $\texttt{CONST}\ v_1\ t_2' : \texttt{List}\ \tau_1$, es decir, se determina que $t' : \texttt{List}\ \tau_1$, por lo que el teorema se cumple.

Subcaso (Regla 21:(E-const3))
Sean $t_1 \longrightarrow v_1$ $t_2 \longrightarrow \texttt{LIST}\ v_2 \ldots v_n$ $t' = \texttt{LIST}\ v_1 \ldots v_n$
Por hipótesis de inducción $\Gamma \vdash t_2' : \texttt{List}\ \tau_1$ y $\Gamma \vdash v_1 : \tau_1$ Aplicando ahora la (Regla 11:(T-const)) se tiene que $\texttt{LIST}\ v_1 \ldots v_n : \texttt{List}\ \tau_1$, es decir, se determina que $t' : \texttt{List}\ \tau_1$, por lo que el teorema se cumple.

Caso (Regla 12:(T-isEmpty?))
Sean $t = \texttt{ISEMPTY?}\ t_1$ $\Gamma \vdash t_1 : \texttt{List}\ \tau_1$ $\tau = \texttt{boolean}$

Subcaso (Regla 22:(E-isEmpty?1))
Sean $t_1 \longrightarrow t_1'$ $t' = \texttt{ISEMPTY?}\ t_1'$
Como $t_1 \longrightarrow t_1'$, si se aplica la hipótesis de inducción entonces se concluye que $\Gamma \vdash t_1' : \texttt{List}\ \tau_1$

Aplicando a estas conclusiones la regla (Regla 12:(T-isEmpty?)) concluimos que t' : `boolean` por lo que el teorema se cumple.

Subcaso (Regla 23:(E-isEmpty?2))
Sea $t' =$ `true` $t_1 =$ `NULL`

Basados en el sistema de tipos se tiene que `true` : `boolean` por lo que inmediatamente se cumple el teorema.

Subcaso (Regla 24:(E-isEmpty?3))
Sea $t' =$ `false`
Basados en el sistema de tipos se tiene que `false` : `boolean` por lo que inmediatamente se cumple el teorema.

Caso (Regla 8:(T-fix))
Sean $t =$ `FIX` $x\ \tau_1\ t_1 \qquad \Gamma,\ x : \tau_1 \vdash t_1 : \tau_1 \qquad \tau = \tau_1$

(Regla 39:(E-fix))
Sea $t' = t_1\,[x :=$ `FIX` $x\ \tau\ e]$
Aplicando el **Lema 7** (***Sustitución***) es posible concluir el enunciado $t' = t_1\,[x :=$ `FIX` $x\ \tau\ e] : \tau_1$, lo cual es lo que se deseaba demostrar.

Caso (Regla 9:(T-list))
Sean $t =$ `LIST` $t_1\ \ldots\ t_n \qquad \Gamma\ \vdash t_1 : \tau_1 \ldots \Gamma\ \vdash t_n : \tau_1 \qquad \tau =$ `List` τ_1

(Regla 18:(E-list))
Sean $t_i \longrightarrow t_i' \qquad t' =$ `LIST` $v_1\ \ldots\ v_{i-1}\ t_i'\ \ldots\ t_n$
Por hipótesis de inducción en las subderivaciones se tiene que $t_i' : \tau_1$ por lo que si se aplica la Regla 9:(T-list) entonces t' : `List` τ_1, lo cual es lo que se deseaba demostrar.

Caso (Regla 13:(T-fst))
Sean $t =$ `FST` $t_1 \qquad \Gamma \vdash t_1 :$ `List` $\tau_1 \qquad \tau = \tau_1$

Subcaso (Regla 25:(E-fst))
Sean $t_1 \longrightarrow t_1' \qquad t' =$ `FST` t_1'
Por hipótesis de inducción en t_1, se concluye que $\Gamma \vdash t_1'$: `List` τ_1. Por lo que aplicando ahora la Regla 13:(T-fst) se demuestra que $\Gamma \vdash$ `FST` $t_1' : \tau_1$.

Subcaso (Regla 26:(E-fstList))
Sean $t_1 =$ `LIST` $v_1\ \ldots\ v_n \qquad t' = v_1$
Considerando las premisas de este Caso (Regla 13:(T-fst)) se puede determinar que `LIST` $v_1\ \ldots\ v_n = t_1 :$ `List` τ Aplicando el **Lema 4** (***Inversión***) se concluye que $v_1 : \tau_1$.

Caso (Regla 14:(T-rst))
Sean $t = \texttt{RST}\ t_1$ $\Gamma \vdash t_1 : \texttt{List}\ \tau_1$ $\tau = \texttt{List}\ \tau_1$

Subcaso (Regla 27:(E-rst))
Sean $t_1 \longrightarrow t_1'$ $t' = \texttt{RST}\ t_1'$
Por hipótesis de inducción en t_1, se concluye que $\Gamma \vdash t_1' : \texttt{List}\ \tau_1$. Por lo que aplicando ahora la Regla 14:(T-rst) se demuestra que $\Gamma \vdash \texttt{RST}\ t_1' : \texttt{List}\ \tau$.

Subcaso (Regla 28:(E-rstList))
Sean $t_1 = \texttt{LIST}\ v_1\ v_2\ \ldots\ v_n$ $t' = \texttt{LIST}\ v_2\ \ldots\ v_n$
Considerando las premisas de este Caso (Regla 14:(T-rst)) se puede determinar que $\texttt{LIST}\ v_1\ v_2\ \ldots\ v_n : \texttt{List}\ \tau_1$. Aplicando ahora el **Lema 4** (***Inversión***) se concluye que $\texttt{LIST}\ v_2\ \ldots\ v_n : \texttt{List}\ \tau_1$.

Caso (Regla 15:(T-err))
Sea $t = \texttt{ERROR}$
Se cumple por vacuidad ya que no existe regla de evaluación que se pueda aplicar a $t = \texttt{ERROR}$. ■

Con el objetivo justificar la creación de reglas de subtipificación algorítmicas sintácticamente directas, se debe demostrar que éstas reglas y las reglas declarativas son equivalentes. El primer paso es demostrar que para todo tipo definido en `MinTyRacket`, utilizando las reglas de subtipificación se puede concluir que todo tipo es subtipo de sí mismo.

Lema 10 *(**Redundancia de la** Regla 70:(Sub-ref)). La relación $\sigma <: \sigma$ puede derivarse para todo tipo σ sin usar la Regla 70:(Sub-ref) .*

Demostración. Se aplicará inducción sobre la estructura de σ.

Caso (Min) Sea σ=`Min`
Usando la Regla 58:(Sub-Min), sea τ=`Min` entonces el lema se cumple.

Caso (Max) Sea σ=`Max`
Utilizando la Regla 59:(Sub-Max), sea τ=`Max` entonces el lema se cumple.

Caso (boolean) Sea σ=`boolean`
Usando la Regla 63:(Sub-boolean) en lugar de la Regla 70:(Sub-ref) en la derivación el lema se cumple.

Caso (integer) Sea σ=`integer`
Utilizando la Regla 64:(Sub-integer3) en lugar de la Regla 70:(Sub-ref) en la derivación obtenemos el resultado deseado.

Caso (float) Sea σ=`float`
Usando la Regla 65:(Sub-float2) en lugar de la Regla 70:(Sub-ref) en la derivación el lema se ha demostrado.

Caso (double) Sea σ=`double`
Usando la Regla 66:(Sub-double) en lugar de la Regla 70:(Sub-ref) en la derivación el lema se cumple.

Caso (List) Sea σ=`List` σ_1
Por hipótesis de inducción en las subderivaciones se cumple que $\sigma_1 <: \sigma_1$ para todo tipo definido en el lenguaje. Usando la Regla 67:(Sub-list) en la derivación el lema se cumple.

Caso (Flecha) Sea $\sigma=\sigma_1 \to \sigma_2$
Por hipótesis de inducción en las subderivaciones se cumple que $\sigma_1 <: \sigma_1$ y también se cumple que $\sigma_2 <: \sigma_2$ para todo tipo definido en el lenguaje. Usando la Regla 68:(Sub-arr) en la derivación el lema se cumple. ■

El segundo paso para demostrar la equivalencia entre las reglas de tipificación declarativa y las reglas de tipificación algortítmica es demostrar que para cualesquiera dos tipos definidos en `MinTyRacket`, utilizando las reglas de subtipificación se puede concluir si un tipo es un subtipo de otro es una sentencia derivable, no se necesita usar la regla de transitividad.

Lema 11 *(**Redundancia de la** Regla 71:(Sub-trans)). $\sigma <: \tau$ es derivable, entonces puede ser derivada sin usar la Regla 71:(Sub-trans).*

Demostración. Notemos que el **Lema** 10 (***Redundancia de la*** Regla 70:(Sub-ref)) dice que si existe alguna derivación de $\sigma <: \tau$, entonces existe una derivación que no utiliza la Regla 70:(Sub-ref). Se aplicará inducción sobre el tamaño de derivaciones que no utilizan tal regla. La hipótesis de inducción dice que todas las subderivaciones de la regla final de tipificación pueden ser remplazadas por derivaciones que no contengan la Regla 71:(Sub-trans).

Si la regla final en la derivación es cualquier otra cosa que la Regla 71:(Sub-trans), por hipótesis de inducción se tiene que la derivación cumple la propiedad que se quería demostrar.

Suponiendo que la regla final de tipificación es la Regla 71:(Sub-trans)), es decir, existen subderivaciones con conclusiones $\sigma <: \upsilon$ y $\upsilon <: \tau$ para algún tipo υ, se demostrará por casos en el par de reglas finales de tipificación que conforman su subderivación.

Caso (- / Regla 58:(Sub-Min)) Sea υ = `Min`
Si la subderivación izquierda termina con la Regla 58:(Sub-Min), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla

71:(Sub-trans). Analizando las reglas de tipificación se puede ver que la regla final de subtipificación en esta subderivación debe ser la Regla 58:(Sub-Min), por lo tanto el resultado se sigue de tal regla.

Caso (Regla 58:(Sub-Min) / -) Sea $\sigma =$ `Min`
Si la derivación izquierda finaliza con la Regla 58:(Sub-Min), como `Min` $<: \tau$ puede ser derivado usando la Regla 58:(Sub-Min) sin importar que forma tenga τ entonces se cumple el lema.

Caso (- / Regla 59:(Sub-Max)) Sea $\tau =$ `Max`
Si la derivación derecha finaliza con la Regla 59:(Sub-Max), como $\sigma <:$ `Max` puede ser derivado usando la Regla 59:(Sub-Max) sin importar que forma tenga σ entonces se cumple el lema.

Caso (Regla 59:(Sub-Max) / -) Sea $\upsilon =$ `Max`
Si la subderivación derecha termina con la Regla 59:(Sub-Max), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación puede notar que la regla final de subtipificación en esta subderivación debe ser la Regla 59:(Sub-Max), por lo tanto el resultado se sigue de tal regla.

Caso (Regla 60:(Sub-float) / -) Sean $\sigma =$ `float` $\upsilon =$ `double`
Si la subderivación derecha termina con la Regla 66:(Sub-double), por hipótesis de inducción es posible suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se puede resaltar que la regla final de subtipificación en esta subderivación debe ser la Regla 66:(Sub-double), aplicando la Regla 60:(Sub-float) se obtiene el resultado deseado.

Los casos restantes se demuestran similarmente y se pueden consultar en el Apéndice A.

Según la forma que poseen las reglas de subtipificación de `MinTyRacket` se concluye que estas combinaciones de derivaciones son las únicas posibles. ■

Es importante notar que cada regla de tipificación declarativa posee su regla análoga de tipificación algorítmica a excepción de la Regla 71:(Sub-trans) y la Regla 60:(Sub-ref). Si se aplican el **Lema** 10 (***Redundancia de la*** Regla 60:(Sub-ref)) y **Lema** 11 (***Redundancia de la*** Regla 71:(Sub-trans)) además de los casos respectivos a cada regla de tipificación declarativa y su correspondiente regla de tipificación algorítmica se podrá concluir su equivalencia.

Primero se demostrará si un término es tipificable por las reglas de tipificación algorítmicas entonces ese mismo término es también tipificable por las reglas de tipificación declarativas, es decir, no se agrega nada que genere conflicto a las demostraciones ya hechas.

Teorema 12 *(**Correctud**). Si* $\Gamma \mapsto \texttt{t} : \tau$ *entonces* $\Gamma \vdash \texttt{t} : \tau$.

Demostración. Se aplicará inducción sobre las derivaciones de tipos.

Caso (Regla 82:(TA-var))
Sean $\texttt{t} = \texttt{x} \quad \Gamma(\texttt{x}) = \tau$
Por (Regla 1:(T-var)) es posible concluir que $\texttt{t} : \tau$ por lo que el teorema se cumple.

Caso (Regla 83:(TA-abs))
Sean $\texttt{t} = \texttt{LAMB x } \tau_1\ t_2 \quad \Gamma, \texttt{x} : \tau_1 \mapsto t_2 : \tau_2 \quad \tau = \tau_1 \to \tau_2$
Por hipótesis de inducción $\Gamma, \texttt{x} : \tau_1 \vdash t_2 : \tau_2$. Si ahora se aplica la (Regla 2:(T-abs)) entonces $\Gamma \vdash \texttt{t} : \tau_1 \to \tau_2$, es decir, $\texttt{t} : \tau$.

Caso (Regla 84:(TA-app))
Sean $\texttt{t} = t_1\ t_2 \quad \Gamma \mapsto t_1 : \tau_1 \quad \tau_1 = \tau_{11} \to \tau_{12} \quad \Gamma \mapsto t_2 : \tau_2$
$\mapsto \tau_2 <: \tau_{11} \quad \tau = \tau_{12}$
Por la completez del algoritmo de subtipificación se infiere que $\vdash \tau_2 <: \tau_{11}$.
Por hipótesis de inducción se tiene que $\Gamma \vdash t_1 : \tau_1$, $\Gamma \vdash t_2 : \tau_2$. Ya que $\tau_1 = \tau_{11} \to \tau_{12}$ entonces $\Gamma \vdash t_1 : \tau_{11} \to \tau_{12}$. Si se aplica la (Regla 3:(T-app)) finalmente se concluye que $\Gamma \vdash t_1\ t_2 : \sigma_{12}$ por lo que el teorema se cumple.

Caso (Regla 85:(TA-let))
Sean $\texttt{t} = \texttt{LET}\ x\ \tau_1\ t_1\ t_2 \quad \Gamma \mapsto t_1 : \tau_1 \quad \Gamma, x : \tau_1 \mapsto t_2 : \tau_2 \quad \tau = \tau_2$
Por hipótesis de inducción se infiere que $\Gamma \vdash t_1 : \tau_1$ y $\Gamma, \texttt{x} : \tau_1 \vdash t_2 : \tau_2$. Aplicando ahora la Regla 4:(T-let) se concluye que $\texttt{LET}\ x\ \tau_1\ t_1\ t_2 : t_2$.

Caso (Regla 94:(TA-if))
Sean $\texttt{t} = \texttt{IF}\ t_1\ t_2\ t_3 \quad \Gamma \mapsto t_1 : \tau_1 \quad \tau_1 = \texttt{boolean} \quad \Gamma \mapsto t_2 : \tau_2$
$\Gamma \mapsto t_3 : \tau_3 \quad t_2 \vee t_3 = \tau$
Basándonos en las premisas se concluye que $\Gamma \mapsto t_1 : \texttt{boolean}$. Por hipótesis de inducción $\Gamma \vdash t_1 : \texttt{boolean}$. Además se puede concluir que $\Gamma \vdash t_2 : \tau_2$ asi como $\Gamma \vdash t_3 : \tau_3$. Como τ es el join de los tipos τ_1 y τ_2, por definición, tanto $\tau_1 <: \tau$ como $\tau_1 <: \tau$. Usando transitividad se determina tanto que $\Gamma \vdash t_2 : \tau$ como que $\Gamma \vdash t_3 : \tau$ respectivamente. Aplicando ahora la (Regla 16:(T-if)) se infiere que $\Gamma \vdash \texttt{IF}\ t_1\ t_2\ t_3 : \tau$ que es lo que se quería demostrar.

Caso (Regla 95:(TA-opAritm))
Sean $\texttt{t} = \texttt{OP}\ OpAritm\ t_1\ t_2 \quad \Gamma \mapsto t_1 : TipoAritm \quad \Gamma \mapsto t_2 : TipoAritm$
$TipoAritm \vee TipoAritm = \tau$
Por hipótesis de inducción $\Gamma \vdash t_1 : TipoAritm$ así como $\Gamma \vdash t_2 : TipoAritm$. Como τ es el join de los tipos $TipoAritm$ y $TipoAritm$, por definición y transitividad tenemos que $\Gamma \vdash t_1 : TipoAritm$, además es posible probar la sentencia denotada $\Gamma \mapsto t_2 : TipoAritm$. Ahora es posible aplicar la Regla 5:(T-opAritm) concluyendo que $\texttt{OP}$ Op $t_1\ t_2 : \tau$ que es lo que se quería demostrar.

Caso (Regla 96:(TA-opRel1))

Sean $\texttt{t} = \texttt{OP}\ OpRel\ t_1\ t_2 \quad \Gamma \mapsto t_1 : TipoRel \quad \Gamma \mapsto t_2 : TipoRel$ $OpRel \vee OpRel = \tau$
Por hipótesis de inducción $\Gamma \vdash t_1 : TipoRel$ así como $\Gamma \vdash t_2 : TipoRel$. Como τ es el join de los tipos $TipoRel$ y $TipoRel$, por definición y transitividad tenemos que $\Gamma \vdash t_1 : TipoRel$, además podemos probar $\Gamma \mapsto t_2 : TipoRel$. Ahora es posible aplicar la Regla 6:(T-opRel1) concluyendo que $\texttt{OP}\ OpRel\ t_1\ t_2 : \tau$ que es lo que se quería demostrar.

Caso (Regla 97:(TA-opRel2))
Sean $\texttt{t} = \texttt{OP}\ OpRel\ t_1\ t_2 \quad \Gamma \mapsto t_1 : TipoAritm \quad \Gamma \mapsto t_2 : TipoAritm$ $\tau_1 \vee \tau_2 = \tau$
Por hipótesis de inducción $\Gamma \vdash t_1 : TipoAritm$ así como $\Gamma \vdash t_2 : TipoAritm$. Como τ es el join de los tipos τ_1 y τ_2, por definición y transitividad tenemos que $\Gamma \vdash t_1 : TipoAritm$, además podemos probar $\Gamma \mapsto t_2 : TipoAritm$. Ahora es posible aplicar la Regla 7:(T-opRel2) concluyendo que $\texttt{OP}\ OpRel\ t_1\ t_2 : \tau$ que es lo que se quería demostrar.

Caso (Regla 86:(TA-fix))
Sean $\texttt{t} = \texttt{FIX}\ x\ \tau\ t \quad \Gamma, x : \tau_1 \mapsto t : \tau_1 \quad \tau = \tau_1$
Por hipótesis de inducción $\Gamma, x : \tau_1 \vdash t : \tau_1$.
Aplicando ahora la Regla 8:(T-fix) se obtiene el resultado deseado.

Caso (Regla 87:(TA-list))
Sean $\texttt{t} = \texttt{LIST}\ t_1 \ldots t_n \quad \Gamma \mapsto t_1 : \tau \ldots \Gamma \mapsto t_n : \tau$
Por hipótesis de inducción $\Gamma \vdash t_1 : \tau \ldots \Gamma \vdash t_n : \texttt{List}\ \tau$. Aplicando ahora la Regla 9:(T-list) se concluye que $\texttt{LIST}\ t_1 \ldots t_n : \texttt{List}\ \tau$ que es lo que se quería demostrar.

Caso (Regla 88:(TA-null))
Sean $\texttt{t} = \texttt{NULL}\ \tau_1 \quad \tau = \texttt{List}\ \ \tau_1$
Aplicando la Regla 10:(T-null) se cumple el teorema.

Caso (Regla 89:(TA-const))
Sean $\texttt{t} = \texttt{CONST}\ t_1\ t_2 \quad \Gamma \mapsto t_1 : \tau_1$
$\Gamma \mapsto t_2 : \texttt{List}\ \ \tau_1 \quad \tau = \texttt{List}\ \tau_1$
Por hipótesis de inducción se tiene que $\Gamma \vdash t_1 : \tau_1$ y $\Gamma \vdash t_2 : \texttt{List}\ \ \tau_1$. Por lo que si se aplica la Regla 11:(T-const) se obtiene el resultado deseado.

Caso (Regla 90:(TA-isEmpty?))
Sean $\texttt{t} = \texttt{ISEMPTY?}\ t_1 \quad \Gamma \mapsto t_1 : \texttt{List}\ \ \tau_1 \quad \tau = \texttt{boolean}$
Por hipótesis de inducción $\Gamma \vdash t_1 : \texttt{List}\ \ \tau_1$.
Aplicando Regla 12:(T-isEmpty?) se concluye el resultado deseado.

Caso (Regla 91:(TA-fst))
Sean $\texttt{t} = \texttt{FST}\ t_1 \quad \Gamma \mapsto t_1 : \texttt{List}\ \ \tau_1 \quad \tau = \tau_1$
Por hipótesis de inducción $\Gamma \vdash t_1 : \texttt{List}\ \ \tau_1$. Aplicando ahora la Regla 13:(T-fst)

se infiere el resultado deseado.

Caso (Regla 92:(TA-rst))
Sean $\texttt{t} = \texttt{RST}\ t_1 \quad \Gamma \mapsto t_1 : \texttt{List}\ \ \tau_1 \quad \tau = \texttt{List}\ \ \tau_1$
Por hipótesis de inducción $\Gamma \vdash t_1 : \texttt{List}\ \ \tau_1$. Por lo que si se aplica la Regla 14:(T-rst) se demuestra que el teorema es válido.

Caso (Regla 93:(TA-err))
Sean $\texttt{t} = \texttt{ERROR} \quad \tau = \texttt{Min}$
Aplicando la Regla 15:(T-err) se cumple el teorema.

Caso (Regla 98:(TA-appMin))
Sean $\texttt{t} = t_1\ t_2 \quad \Gamma \mapsto t_1 : \tau_1 \quad \tau_1 = \texttt{Min} \quad \Gamma \mapsto t_2 : \tau_2 \quad \tau = \texttt{Min}$
Utilizando las premisas se tiene que $\Gamma \mapsto t_1 : \texttt{Min}$. Por hipótesis de inducción se infiere que $\Gamma \vdash t_1 : \texttt{Min}$ y que $\Gamma \vdash t_2 : \tau_2$ Aplicando ahora la Regla 17:(T-appMin) se concluye que $t_1\ t_2 : \texttt{Min}$, lo cual es lo que se quería demostrar. ■

El segundo y último paso para demostrar la equivalencia entre la relaciones de tipificación y subtipificación declarativas y algorítmicas consiste en demostrar que si un término es tipificable por las reglas de tipificación declarativas entonces ese mismo término es también tipificable por las reglas de tipificación algorítmicas, es decir, la relación de subtipificación algorítmica contiene todo lo derivable por las relación de subtipificación declarativa.

Teorema 13 *(**Completez**). Si $\Gamma \vdash t : \tau$ entonces $\Gamma \mapsto t : \sigma$ para algún $\sigma <: \tau$.*

Demostración. Se aplicará inducción estructural sobre las derivaciones de tipos.

Caso (Regla 1:(T-var)
Sean $\texttt{t} = \texttt{x} \quad \Gamma(\texttt{x}) = \tau$
Por (Regla 82:(TA-var)) el teorema se cumple.

Caso (Regla 2:(T-abs))
Sean $\texttt{t} = \texttt{LAMB x}\ \tau_1\ t_2 \quad \Gamma, \texttt{x} : \tau_1 \vdash t_2 : \tau_2 \quad \tau = \tau_1 \to \tau_2$
Por hipótesis de inducción $\Gamma, \texttt{x} : \tau_1 \mapsto t_2 : \sigma_2$ para algún $\sigma_2 <: \tau_2$. Por la (Regla 83:(TA-abs)) se determina que $\Gamma \mapsto \texttt{t} : \tau_1 \to \sigma_2$. Si ahora se aplica (Regla 68:(Sub-arr)) entonces $\tau_1 \to \sigma_2 <: \tau_1 \to \tau_2$ que es lo que se quería demostrar.

Caso (Regla 3:(T-app))
Sean $\texttt{t} = t_1\ t_2 \quad \Gamma \vdash t_1 : \tau_{11} \to \tau_{12} \quad t_2 : \tau_{11} \quad \tau = \tau_{12}$
Por hipótesis de inducción $\Gamma \mapsto t_1 : \sigma_1$ para algún $\sigma_1 <: \tau_{11} \to \tau_{12}$ así como $\Gamma \mapsto t_2 : \sigma_2$ para algún $\sigma_2 <: \tau_{11}$. Por el **Lema 4** (***Inversión***) σ_1 debe poseer

la forma $\sigma_{11} \to \sigma_{12}$ para algún σ_{11} y σ_{12} con $\tau_{11} <: \sigma_{11}$ y $\sigma_{12} <: \tau_{12}$. Usando transitividad se tiene que $\sigma_2 <: \sigma_{11}$ Si se toma en cuenta el **Teorema ??** (***Correctud y Completez de Subtipos***) entonces $\mapsto \sigma_2 <: \sigma_{11}$. Finalmente aplicando la (Regla 84:(TA-app)) tenemos que $\Gamma \mapsto t_1\ t_2 : \sigma_{12}$.

Caso (Regla 4:(T-let))
Sean $\texttt{t} = \texttt{LET}\ x\ \tau_1\ t_1\ t_2 \quad \Gamma \vdash t_1 : \tau_1 \quad \Gamma, \texttt{x} : \tau_1 \vdash t_2 : \tau_2 \quad \tau = \tau_2$
Por hipótesis de inducción $\Gamma \mapsto t_1 : \sigma_1$ tal que $\tau_1 <: \tau_1$ y $\Gamma, \texttt{x} : \tau_1 \vdash t_2 : \sigma_2$ además de que $\sigma_2 <: \tau_2$. Por subsunción $\Gamma \mapsto t_1 : \tau_1$ y $\Gamma, \texttt{x} : \tau_1 \vdash t_2 : \tau_2$. Si ahora se aplica la Regla 85:(TA-let) entonces $\texttt{LET}\ x\ \tau_1\ t_1\ t_2 : t_2$.

Caso (Regla 69:(Subsunción))
Sean $\texttt{t} : \sigma \quad \sigma <: \tau$
Por hipótesis de inducción $\mapsto \texttt{t} : \sigma_1$ para algún $\sigma_1 <: \sigma$. Utilizando transitividad se concluye que $\sigma_1 <: \tau$.

Caso (Regla 16:(T-if))
Sean $\texttt{t} = \texttt{IF}\ t_1\ t_2\ t_3 \quad \Gamma \vdash t_1 : \texttt{boolean} \quad \Gamma \vdash t_2 : \tau_{23} \quad \Gamma \vdash t_3 : \tau_{23}$
$\tau = \tau_{23}$
Basándonos en las premisas se infiere que $\Gamma \vdash t_1 : \sigma_1$ para algún $\sigma_1 <: \texttt{boolean}$. Analizando las reglas de subtipificación se infiere que `boolean` es el único subtipo posible, entonces tenemos que $\Gamma \vdash t_1 : \texttt{boolean}$. Por hipótesis de inducción se tiene que $\Gamma \mapsto t_1 : \texttt{boolean}$. También por hipótesis de inducción se concluye que $\Gamma \mapsto t_2 : \sigma_2$ para algún tipo $\sigma_2 <: \tau_{23}$ así como que $\Gamma \vdash t_3 : \sigma_2$ para algún $\sigma_2 <: \tau_{23}$. El único caso que importa sucede cuando el tipo join de σ_1 y σ_2 es τ_{23}. Aplicando (Regla 94:(TA-if)) se demuestra el teorema.

Caso (Regla 5:(T-opAritm))
Sean $\texttt{t} = \texttt{OP}\ OpAritm\ t_1\ t_2 \quad \Gamma \vdash t_1 : TipoAritm \quad \Gamma \vdash t_2 : TipoAritm$
$\tau = TipoAritm$
Por hipótesis de inducción se cumple que $\Gamma \mapsto t_1 : \sigma_1$ y $\Gamma \mapsto t_2 : \sigma_2$ tales que $\sigma_1 <: TipoAritm \ldots \sigma_2 <: TipoAritm$. El caso que se necesita ocurre cuando $\sigma_2 \vee \sigma_2 = TipoAritm$. Por lo que aplicando la Regla 95:(TA-opAritm) se concluye el resultado deseado.

Caso (Regla 6:(T-opRel1))
Sean $\texttt{t} = \texttt{OP}\ OpRel\ t_1\ t_2 \quad \Gamma \vdash t_1 : TipoRel \quad \Gamma \vdash t_2 : TipoRel$
$\tau = TipoRel$
Por hipótesis de inducción se cumple que $\Gamma \mapsto t_1 : \sigma_1$ y $\Gamma \mapsto t_2 : \sigma_2$ tales que $\sigma_1 <: TipoRel \ldots \sigma_2 <: TipoRel$. El caso que se necesita ocurre cuando se cumple $\sigma_2 \vee \sigma_2 = TipoRel$. Por lo que aplicando la Regla 96:(TA-opRel1) se concluye el resultado deseado.

Caso (Regla 7:(T-opRel2))
Sean $\texttt{t} = \texttt{OP}\ OpRel\ t_1\ t_2 \quad \Gamma \vdash t_1 : TipoAritm \quad \Gamma \vdash t_2 : TipoAritm$

$\tau = TipoRel$
Por hipótesis de inducción se cumple que $\Gamma \mapsto t_1 : \sigma_1$ y $\Gamma \mapsto t_2 : \sigma_2$ tales que $\sigma_1 <: TipoAritm \ldots \sigma_2 <: TipoAritm$. El caso que se necesita ocurre cuando $\sigma_2 \vee \sigma_2 = TipoAritm$. Por lo que aplicando la Regla 97:(TA-opRel2) se concluye el resultado deseado.

Caso (Regla 8:(T-fix))
Sean $\texttt{t} = \texttt{FIX}\ x\ \tau\ t \qquad \Gamma, x : \tau_1 \vdash t : \tau_1 \qquad \tau = \tau_1$
Por hipótesis de inducción $\Gamma, x : \tau_1 \mapsto t : \sigma_1$ tal que $\sigma_1 <: \tau_1$. Utilizando transitividad $\Gamma, x : \tau_1 \mapsto t : \tau_1$. Aplicando ahora la Regla 86:(TA-fix) se obtiene el resultado deseado.

Caso (Regla 9:(T-list))
Sean $\texttt{t} = \texttt{LIST}\ t_1 \ldots t_n \qquad \Gamma \vdash t_1 : \tau_0 \ldots \Gamma \vdash t_n : \tau_0 \qquad \tau = \texttt{List}\ \tau_0$
Por hipótesis de inducción se cumple que $\Gamma \mapsto t_1 : \sigma_1 \ldots \Gamma \mapsto t_n : \sigma_n$ tales que $\sigma_1 <: \tau_0 \ldots \sigma_n <: \tau_0$. Aplicando la Regla 87:(TA-list) se concluye el resultado deseado.

Caso (Regla 10:(T-null))
Sean $\texttt{t} = \texttt{NULL}\ \tau_1 \qquad \tau = \texttt{List}\ \ \tau_1$
Por Regla 88:(TA-null) se cumple el teorema.

Caso (Regla 11:(T-const))
Sean $\texttt{t} = \texttt{CONST}\ t_1\ t_2 \qquad \Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \texttt{List}\ \ \tau_1 \qquad \tau = \texttt{List}\ \tau_1$
Por hipótesis de inducción se tiene que $\Gamma \mapsto t_1 : \sigma_1$ tal que $\sigma_1 <: \tau_1$ así como que $\Gamma \mapsto t_2 : \texttt{List}\ \ \sigma_1$ tal que $\texttt{List}\ \ \sigma_1 <: \texttt{List}\ \ \tau_1$. Utilizando subsunción y Regla 67:(Sub-List) respectivamente, se puede concluir que $\Gamma \mapsto t_1 : \tau_1$, ademas de que $\Gamma \vdash t_2 : \texttt{List}\ \ \tau_1$ por lo que si se aplica la Regla 89:(TA-const) se obtiene el resultado deseado.

Caso (Regla 12:(T-isEmpty?))
Sean $\texttt{t} = \texttt{ISEMPTY?}\ t_1 \qquad \Gamma \vdash t_1 : \texttt{List}\ \ \tau_1 \qquad \tau = \texttt{boolean}$
Por hipótesis de inducción $\Gamma \mapsto t_1 : \texttt{List}\ \ \sigma_1$ tal que $\texttt{List}\ \ \sigma_1 <: \texttt{List}\ \ \tau_1$
Utilizando subsunción y aplicando la Regla 90:(TA-isEmpty?) se obtiene el resultado deseado.

Caso (Regla 13:(T-fst))
Sean $\texttt{t} = \texttt{FST}\ t_1 \qquad \Gamma \vdash t_1 : \texttt{List}\ \ \tau_1 \qquad \tau = \tau_1$
Por hipótesis de inducción $\Gamma \mapsto t_1 : \texttt{List}\ \ \sigma_1$ tal que $\texttt{List}\ \ \sigma_1 <: \texttt{List}\ \ \tau_1$
Aplicando subsunción y la Regla 91:(TA-fst) se obtiene el resultado deseado.

Caso (Regla 14:(T-rst))
Sean $\texttt{t} = \texttt{RST}\ t_1 \qquad \Gamma \vdash t_1 : \texttt{List}\ \ \tau_1 \qquad \tau = \texttt{List}\ \ \tau_1$
Por hipótesis de inducción $\Gamma \vdash t_1 : \texttt{List}\ \ \sigma_1$ tal que $\texttt{List}\ \ \sigma_1 <: \texttt{List}\ \ \tau_1$
Aplicando subsunción y la Regla 92:(TA-rst) se demuestra que el teorema es

válido.

Caso (Regla 15:(T-err))
Sean $\texttt{t} = \texttt{ERROR} \qquad \tau = \texttt{Min}$
Aplicando la Regla 93:(TA-err) se cumple el teorema.

Caso (Regla 17:(T-appMin))
Sean $\texttt{t} = t_1\ t_2 \qquad \Gamma \vdash t_1 : \texttt{Min} \qquad \Gamma \vdash t_2 : \tau_2 \qquad \tau = \texttt{Min}.$
Por hipótesis de inducción se infiere que $\Gamma \mapsto t_1 : \sigma$ para algún $\sigma <: \texttt{Min}$ y que $\Gamma \mapsto t_2 : \tau_2$. Analizando las reglas de tipificación es posible inferir que el único subtipo de `Min` es él mismo, es decir, $\Gamma \mapsto t_1 : \texttt{Min}$. Aplicando ahora la Regla 98:(TA-appMin) se concluye que $t_1\ t_2 : \texttt{Min}$. ■

Ya que se ha demostrado la equivalencia entre las relaciones de tipificación y subtipificación, ahora `MinTyRacket` posee reglas de tipificación y subtipificación sintácticamente directas, es decir, reglas que pueden constituir un algoritmo que siempre puede saber que hacer en cada derivación de tipos posible, evitando traslapar términos en las conclusiones.

Capítulo 4

Implementación

En este capítulo se presenta una descripción de la implementación del lenguaje `MinTyRacket`, utilizando el lenguaje `Racket`, un lenguaje de programación funcional y con evaluación glotona que soporta funciones de primera clase.

`Racket` es un lenguaje descendiente de `Scheme` que posee las siguientes características [16]:

- Continuaciones. Representaciones del resto de un cálculo como un procedimiento de un argumento, de manera tal que cuando el cálculo se reinicie lo hará con ese valor.
- Funciones de primera clase. Es posible pasar funciones como argumentos en la implementación de funciones.
- Módulos. Unidades de organización de un programa dentro de archivos definidos en directorios relacionados entre sí por referencias.
- Recolección de basura. Esta propiedad se ejecuta de manera automática, adicionalmente posee funciones como `(collect-garbage)` que solicitan y liberan localidades de memoria, así como la función `(current-memory-use)` que nos regresa una estimación de la memoria usada.
- Hilos de ejecución. Diversos flujos de operación pueden ejecutarse simultáneamente.
- Macros. Formas sintácticas que amplían la sintaxis original del lenguaje.
- Emparejamiento de patrones. Reconocimiento de expresiones regulares para calcular patrones.

Para implementar el lenguaje `MinTyRacket` se ha utilizado el ambiente gráfico de desarrollo `DrRacket versión 5.3.6` donde se han desarrollado los módulos `DefinicionesMTR`, `Parser MTR`, `VerificadorTiposMTR`, `EvaluadorMTR` e `interpMTR` correspondientes a las secciones que se describirán a continuación.

En general, en los ejemplos se denota la línea de comandos con el símbolo `>`, mientras el resultado de cada procedimiento se denota con el caracter `->`.

4.1. Análisis sintáctico de MinTyRacket

Para llevar a cabo el análisis sintáctico diseñado en el Capítulo 3 se ha definido el procedimiento `parse` (analizador sintáctico) que implementa la transición de sintaxis concreta a sintaxis abstracta. Dicho procedimiento recibe como parámetro una `sexp` que es una cadena de caracteres, los cuales son tratados como un solo elemento en el lenguaje `Racket`, conocido como `datum`, y que es definido por la forma `'datum` o por su equivalente `(quote datum)` [16]. A continuación se mostrarán algunos ejemplos de la ejecución del procedimiento `parse`.

```
1  > (parse '(fst (list 1.5 2.5 3.5 4.5)(float)))
2  -> (FST
3      (LIST
4       (list
5        (flot 1.5)
6        (flot 2.5)
7        (flot 3.5)
8        (flot 4.5))))
```

Código 4.1: Ejemplo de ejecución del analizador sintáctico con operador de listas.

En el Código 4.1, línea 1 se llama al procedimiento `parse` con una expresión constante como parámetro que recibe tres expresiones sintácticas concretas:

- `fst`. Operación que obtiene el primer elemento de una lista.
- `(list 1.5 2.5 3.5 4.5)`. Representación de las listas en nuestro lenguaje.
- `(float)`. Tipo que el programador asigna a la expresión sintáctica.

Entre las líneas 2 y 8 se muestra el resultado de la llamada al procedimiento `parse`, obteniendo una expresión `FST` que contiene una expresión `LIST` y que a su vez contiene una lista con los elementos denotados en la expresión constante. Tal resultado de la llamada al procedimiento `parse` es la representación abstracta de la expresión `fst`.

```
1  > (parse '(if true (+ 5.6 5) (* 5 5) (float) ))
2  -> (IF
3      (bool #t)
4      (OP '+ (flot 5.6) (num 5))
5      (OP '* (num 5) (num 5)))
```

Código 4.2: Ejemplo de ejecución del analizador sintáctico con la condicional.

En la línea 1 del Código 4.2, se ejecuta el procedimiento `parse` con una expresión constante conformada por cinco expresiones sintácticas concretas:

- `if`. Expresión que denota la operación condicional en nuestro lenguaje.
- `true`. Condición booleana de la operación condicional.
- `(+ 5.6 5)`. Rama a evaluar en caso de que la condicional se evalúa a verdadero.
- `(* 5 5)`. Rama a evaluar en caso de que la condición se evalúe a falso.
- `(float)`. Tipo escrito por el programador para indicar el tipo de la expresión sintáctica.

Entre las líneas 2 y 5 podemos ver el resultado de la llamada al procedimiento `parse` descrito, obteniendo una expresión que conforma la representación abstracta de la expresión `if` que contiene tres elementos, el primer elemento es la condicional `(bool #t)`. En la línea 4 se puede ver el segundo elemento `(OP '+ (flot 5.6) (num 5))` que es la representación abstracta de la rama a evaluar si la condición se evalúa a verdadero. Se puede ver que en este ejemplo la expresión abstracta se conforma por una operación binaria denotada por el elemento `OP`. En la línea 5 se ve el último elemento `(OP '* (num 5) (num 5)))`, el cual es la representación abstracta de la rama a evaluar si la condición se evalúa a falso.

```
1  > (parse '(fix (fac (lambda (n integer)
2     (if (== n 0)  1   ( * n (fac (+ n -1))))) (integer ->
          float))
3     (fac 6)(float)))
4  -> (FIX
5       'fac
6       (LAMBDA
7        '(n)
8        '(integer)
9         (IF
10          (OP '== (VAR 'n) (num 0))
11          (num 1)
12          (OP
13          '*
14         (VAR 'n)
15         (APP (VAR 'fac) (list (OP '+ (VAR 'n) (num -1))))
              ))
16         (APP (VAR 'fac) (list (num 6))))
```

Código 4.3: Ejemplo de ejecución del analizador sintáctico con funciones recursivas.

En el Código 4.3, en la línea 1 se llama al procedimiento `parse` utilizando como parámetro una expresión sintáctica que recibe cuatro expresiones:

- `fix`. Expresión que denota la definición de funciones recursivas.
- `(fac (lambda (n integer) (if (==n 0) 1 (* n (fac(+ n -1)))))) (integer -> float))`. Cuerpo de la función recursiva.
- `(integer -> float)`. Tipo indicado por el programador de la función recursiva descrita como se puede observar en la línea 2.
- `(fac 6)`. Aplicación de la función recursiva como se puede ver en la línea 3.
- `(float)` Tipo escrito por el programador para señalar el tipo de la llamada de la función recursiva.

Entre las líneas 4 y 16 se puede ver la representación abstracta de la expresión `fix` de este ejemplo, la cual es construida al obtener la representación abstracta de cada una de las expresiones que recibe la expresión sintáctica con que se realizó la llamada al procedimiento `parse`, como son `LAMBDA`, `VAR`, `IF`, `OP` y `APP`.

4.2. Verificador de tipos de MinTyRacket

Con base en el Capítulo 3 y con el objetivo de determinar si un programa está legalmente construido en tiempo de compilación, se han implementado definiciones de tipos y procedimientos que calculan las interacciones entre tales definiciones y la representación abstracta de las expresiones pertenecientes al lenguaje `MinTyRacket`. En este capítulo primero se verá tanto la definición de los tipos elementales, como los no elementales, permitidos en dicho lenguaje, mientras en las subsecciones posteriores se describirán las implementaciones de los algoritmos de subtipificación y tipificación diseñados en el capítulo anterior.

```
1  #lang plai
2
3   (define-type tipo_elem
4      [int]
5      [float]
6      [double]
7      [boolean]
8      [Min]
9      [Max]
10    )
```

Código 4.4: Definición de los tipos elementales.

En el Código 4.4, línea 2 se declaran los tipos elementales de `MinTyRacket` como `tipo_elem`, desde la línea 4 hasta la línea 9 se definen sus seis posibles representaciones abstractas, como son entero, flotante, doble, booleano, mínimo y máximo.

```
1  #lang plai
2
3   (define-type tipo
4     [lista (elementos tipo_elem?)]
5     [flecha (antec tipo_elem?) (consec tipo_elem?)]
6    )
```

Código 4.5: Definición de tipos a partir de los tipos elementales. 4.4.

En el Código 4.5, en la línea 3 se declaran los tipos de `MinTyRacket` como `lista` y `flecha` que mas adelante se asociarán a las listas y a las funciones `lambda` respectivamente, mientras que en las líneas 4 y 5 se puede observar su definición utilizando los tipos elementales vistos en el Código 4.4.

4.2.1. Algoritmo de subtipificación

A continuación se describirá la implementación del algoritmo de subtipificación diseñado en el capítulo anterior. Por ello se ha creado el procedimiento `esSubtipo` que determina si un tipo es subtipo de otro tipo. También se ha creado el procedimiento `JOIN` que calcula, en caso de exista, el mínimo supertipo común de dos tipos [15].

```
1  #lang plai
2
3   (define (esSubtipo sigma tau)
4    (cond
5     [(Min? sigma) true]
6     [(Max? tau) true]
7     [(and   (booln? sigma)  (booln? tau)  ) true]
8     [(and   (int? sigma)    (int? tau)    ) true]
9     [(and   (float? sigma)  (float? tau)  ) true]
10    [(and   (float? sigma)  (double? tau) ) true]
11    [(and   (int? sigma)    (double? tau) ) true]
12    [(and   (double? sigma) (double? tau) ) true]
13    [(and   (int? sigma)    (float? tau)  ) true]
14    [(and   (flecha? sigma) (flecha? tau) )
15     (and (esSubtipo(flecha-antec tau) (flecha-antec sigma
          ))
16        (esSubtipo(flecha-consec sigma)(flecha-consec tau)
             ))]
17    [(and (lista? sigma) (lista? tau) )  (esSubtipo(lista-
         elementos sigma) (lista-elementos tau))]
18    [else false]))
```

Código 4.6: Implementación del algoritmo de subtipificación a partir de la definición de tipos 4.5

En el Código 4.6, en la línea 3 se declara el procedimiento `esSubtipo` que recibe dos parámetros, `sigma` y `tau`, la cual implementará el algoritmo de subtipificación definido en el subcapítulo 3.4. A partir de la línea 4 hasta la línea

18 dependiendo de los valores de `sigma` y `tau` se evalúa a verdadero (`true`) si se cumple alguna regla de subtipificación denotada en el subcapítulo 3.4, en caso de que no se cumpla alguna regla de subtipificación entonces el procedimiento `esSubtipo` se evalúa a falso (`false`).

A continuación se expondrán ejemplos de ejecución del procedimiento llamado `esSubtipo`.

```
1  > (esSubtipo (int) (float))
2  ->  #t
```

Código 4.7: Ejemplo de ejecución del algoritmo de subtipificación en tipos elementales.

En el Código 4.7, en la línea 1 se llama al procedimiento `esSubtipo` con dos parámetros como son los tipos `int` y `float`, tipos pertenecientes a la categorización de tipos 4.5. Mientras en la línea 2 se muestra el valor booleano `#t` indicando que de acuerdo al algoritmo de subtipificación el tipo `int` es un subtipo del tipo `float`.

```
1  > (esSubtipo (lista (double)) (lista (float)) )
2  ->  #f
```

Código 4.8: Ejemplo de ejecución del algoritmo de subtipificación con tipos asociados a listas.

En la línea 1 del Código 4.8, se ejecuta el procedimiento `esSubtipo` con los tipos `lista (double)` y `lista (float)` como parámetros, tales tipos pertenecen a la definición de los tipos asociados a las listas en `MinTyRacket`. Mientras en la línea 2 se exhibe el resultado de la llamada, el valor booleano `#f`, señalando que de acuerdo al algoritmo de subtipificación el tipo `lista (double)` no es un subtipo del tipo `lista (float)`.

```
1  > (esSubtipo (flecha (int) (float)) (flecha (int)(double)
       ))
2  ->  #t
```

Código 4.9: Ejemplos de ejecución del algoritmo de subtipificación en tipos asociados a funciones 4.5.

En el Código 4.9, en la línea 1 se llama al procedimiento `esSubtipo` con los tipos `flecha (int) (float)` y `flecha (int)(double)` como parámetros, tipos correspondientes a las funciones de nuestro lenguaje. En la línea 2 se muestra el valor booleano `#t` indicando que el tipo `flecha (int) (float)` es un subtipo del tipo `flecha (int)(double)`.

Para obtener el mínimo supertipo común de dos tipos definidos en el intérprete `MinTyRacket`, también conocido como el `join` de dos tipos, se ha construido el procedimiento `JOIN`:

```
1  #lang plai
2
3 (define (JOIN v_1 v_2)
4   (let([listaSupertiposTotal (obtenerSupertiposComunes v_1
        v_2 TOTALTIPOS supertiposComunes)])
5   (elemMin listaSupertiposTotal
6   (car listaSupertiposTotal))))
```

Código 4.10: Definición del procedimiento que calculará el supertipo mínimo común de dos tipos.

En el Código 4.10, en la línea 3, se define el procedimiento `JOIN`, el cual entre las líneas 4 y 5 utiliza funciones auxiliares como `join` la cual regresa la lista de supertipos comunes entre dos tipos, recibe cuatro parámetros, el primero y segundo son `v_1` y `v_2` respectivamente, el tercer parámetro es `TOTALTIPOS` el cual contiene las combinaciones posibles de acuerdo a los tipos definidos en nuestro lenguaje, por último el cuarto parámetro es `supertiposComunes` que es la lista donde se almacenarán los supertipos comunes. El procedimiento `elemMin` evaluará el tipo mínimo común de ese conjunto de supertipos.

A continuación se describirán los procedimientos necesarios para calcular el mínimo supertipo común de dos tipos, utilizados por el procedimiento `JOIN`.

```
1 (define (obtenerCombinacionesFunc listaTipos1 listaTipos2
      combinacionTipos)
2 (if (null? listaTipos1)
3     combinacionTipos
4     (obtenerCombinacionesFunc (cdr listaTipos1)
          listaTipos2 (cons (flecha (car listaTipos1)
5     (car listaTipos2)) combinacionTipos))))
```

Código 4.11: Definición del procedimiento que obtiene las combinaciones de tipos de funciones.

En el Código 4.11, en la línea 1 se define `obtenerCombinacionesFunc`, el cual es un procedimiento que recibe tres parámetros, el primero es `listaTipos1`, lista que como se puede ver entre las líneas 2 y 5 se recorre para formar las combinaciones de los tipos asociados a funciones utilizando la lista `listaTipos2`, el segundo parámetro. Las combinaciones resultantes se almacenan en la lista `combinacionTipos`.

```
1 (define (obtenerCombinacionesLista listaTipos1
      combinacionTiposLista)
2 (if (null? listaTipos1)
```

```
3      combinacionTiposLista
4      (obtenerCombinacionesLista
5        (cdr listaTipos1) (cons(lista (car listaTipos1))
             combinacionTiposLista))))
```

Código 4.12: Definición del procedimiento que obtiene las combinaciones de tipos de listas.

En el Código 4.12, línea 1 se crea el procedimiento `obtenerCombinacionesLista`, el cual recibe tres parámetros, el primero es `listaTipos1`, lista que como se observa entre las líneas 2 y 5, se recorre para formar las combinaciones de los tipos asociados a listas utilizando la lista `listaTipos2`, el segundo parámetro. Las combinaciones resultantes se almacenan en la lista `combinacionTiposLista`.

```
1 (define TOTALTIPOS (append (append
2  (append (list (int) (Min) (Max) (booln) (double) (float))
3    (obtenerCombinacionesLista listaTypes2
         combinacionTiposLista))
4          (obtenerTotalComb listaTypes2 combinacionTipos))))
```

Código 4.13: Definición del procedimiento que concatena las combinaciones de tipos.

En el Código 4.13, en la línea 1 se crea el procedimiento `TOTALTIPOS`, el cual no recibe parámetros ya que su función es concatenar los tipos elementales definidos en `MinTyRacket`, así como los tipos asociados a listas y funciones calculados en los dos procedimientos anteriores.

```
1 (define (obtenerSupertiposComunes tipo_1 tipo_2 listaTipos
       supertiposComunes)
2 (if (null? listaTipos)  supertiposComunes
3     (cond
4       [(and (esSubtipo tipo_1 (car listaTipos))
5             (esSubtipo tipo_2 (car listaTipos)))
6              (obtenerSupertiposComunes tipo_1 tipo_2
7              (cdr listaTipos)
8                (cons (car listaTipos) supertiposComunes))]
9       [else (obtenerSupertiposComunes tipo_1 tipo_2
10            (cdr listaTipos) supertiposComunes)])))
```

Código 4.14: Definición del procedimiento que obtiene los supertipos de dos tipos.

En el Código 4.14, en la línea 1 se define`obtenerSupertiposComunes`, el cual es un procedimiento recibe cuatro parámetros, el primero y segundo son `tipo_1` y `tipo_2` respectivamente, tipos a partir de los cuales se desean conocer los supertipos comunes. El tercer parámetro es `listaTipos`, la cuál debe contener todas las combinaciones posibles de tipos en `MinTyRacket`. El último parámetro

es `supertiposComunes`, la cuál es una lista donde se almacenarán los supertipos comunes calculados mediante el operador lógico `and`, de forma que entre las líneas 4 y 5 se determina si cada combinación posible de tipos es un supertipo común de los dos tipos `tipo_1` y `tipo_2` ingresados como parámetros.

```
1 (define elemMin
2  (lambda (ls min)
3   (if (null? ls) min
4       (elemMin (cdr ls) (minimo (car ls) min)))))
5
6
7 (define minimo
8  (lambda(ls1 ls2)
9   (if (esSubtipo  ls1  ls2)
10      ls1
11      ls2 )))
```

Código 4.15: Definición del procedimiento que calcula el mínimo supertipo común.

En el Código 4.15, se definen dos procedimientos estrechamente relacionados, entre las líneas 1 y 4 se crea el procedimiento `elemMin`, el cual recibe dos parámetros, el primero es `ls`, lista que debe contener los supertipos comunes de dos tipos, como se puede observar entre las líneas 3 y 4, se recorre la lista `ls` para obtener con la ayuda del procedimiento `minimo`, definido entre las líneas 7 y 11 el mínimo supertipo común de una lista de supertipos comunes entre dos tipos.

A continuación se mostrarán ejemplos del procedimiento `JOIN`:

```
1 > (JOIN (double) (Max))
2 -> (Max)
```

Código 4.16: Ejemplos del procedimiento que evaluará el supertipo mínimo común de tipos elementales.

En el Código 4.16, en la línea 1 se llama al procedimiento `JOIN` con los tipos `double` y `Max` como parámetros, mientras en la línea 2 se muestra el tipo `Max` inferido en la llamada, el cual representa el mínimo supertipo común evaluado por el procedimiento mencionado.

```
1 > (JOIN (flecha (int)(double)) (flecha (double)(Max)))
2 -> (flecha (int) (Max))
```

Código 4.17: Ejemplos del procedimiento que evaluará el supertipo mínimo común de tipos asociados a funciones.

En el Código 4.17, en la línea 1 se ejecuta el procedimiento `JOIN` con los tipos `flecha (int)(double)` y `flecha (double)(Max)` como parámetros. En la línea 2 se muestra el tipo `(flecha (int) (Max)` evaluado por la llamada

descrita, el cual es el mínimo supertipo común.

```
1 > (JOIN (lista (double)) (float))
2 -> (Max)
```

Código 4.18: Ejemplos del procedimiento que evaluará el supertipo mínimo común de tipos asociados a listas.

En el Código 4.18, en la línea 1 se llama al procedimiento `JOIN` con los tipos `lista (double)` y `float` como parámetros. En la línea 2 se muestra el tipo `Max`, el cual es el mínimo supertipo común evaluado por el procedimiento mencionado.

4.2.2. Algoritmo de verificación de tipos

En este subcapítulo se describirá la implementación del algoritmo de verificación de tipos diseñado en el capítulo anterior. Se ha creado el procedimiento `verificador`, el cual es el verificador de tipos y devuelve el tipo correspondiente a la expresión abstracta de la sentencia escrita por el programador. A continuación se describirá la implementación del procedimiento `VERIFICADOR`, el cual se evalúa a verdadero (`true`) si el tipo inferido por el verificador de tipos (`verificador`) es subtipo del tipo indicado por el programador, y en caso contrario el procedimiento `VERIFICADOR` se evalúa a falso (`false`).

```
1  #lang plai
2
3   (define (VERIFICADOR exp envTipo)
4
5   (cond
6   [(and (FIX? (abs exp))   (esSubtipo (verificador (abs
        exp) envTipo) (car (parseTipos exp))))
7    (if
8     (esSubtipo (flecha-consec  (verificador (abs exp)
          envTipo)) (second (parseTipos exp)))
9     true
10    false)]
11  [else
12   (if
13    (esSubtipo (verificador (abs exp) envTipo) (parseTipos
         exp))
14    true
15    false
16    )]))
```

Código 4.19: Definición del procedimiento que determina si el tipo inferido es subtipo del tipo indicado sintácticamente por el usuario.

En el Código 4.19, en la línea 3, se define el procedimiento `VERIFICADOR`, el cual recibe como parámetros una expresión `exp` y un ambiente de tipos `envTipo`

que permitirá almacenar tipos durante la verificación de los mismos, como se puede observar en el siguiente código 4.24. El procedimiento `verificador` evalúa los tipos correspondientes a la expresión abstracta `exp`.

A continuación se muestra la defincón de los ambientes de tipos.

```
1 #lang plai
2
3 (define-type EnvTipos
4   [mtEnvTipos]
5   [anEnvTipos (name symbol?) (value tipo_elem?)
6     (env EnvTipos?)]
```

Código 4.20: Definición de los ambientes de tipos.

En el Código 4.20, se definen los ambientes de tipos de `MinTyRacket`, en la línea 4 se define el ambiente vacío `mtEnvTipos`. En la línea 5 y 6 se define recursivamente un ambiente de tipos que se forma por un identificador `name`, un valor `value` y un ambiente.

El procedimiento `verificador` mencionado en el Código 4.19 implementa las reglas de tipificación definidas en el Subcapítulo 3.5, donde a partir de la representación abstracta de las expresiones de `MinTyRacket` se determinan sus respectivos tipos. A continuación se exhibirán algunos ejemplos de su ejecución.

```
1 > (verificador
2    (IF
3     (bool #t)
4     (OP '+ (flot 5.6) (num 5))
5     (OP '* (num 5) (num 5))) (mtEnvTipos))
6 -> (float)
```

Código 4.21: Ejemplo del verificador de tipos en la expresión condicional.

Entre las líneas 1 y 5 del Código 4.21, se ejecuta el procedimiento `verificador` que recibe dos parámetros:

- Representación abstracta de la expresión `if`, la cual contiene tres elementos:
  - ◇ `(bool #t)`. Condición booleana que determina cual de las ramas de la condicional se evaluará.
  - ◇ `(OP '+ (flot 5.6) (num 5))`. Rama a evaluar en caso de que la condición se evalúe verdadero.
  - ◇ `(OP '* (num 5) (num 5))`. Rama a evaluar en caso de que la condición se evalúe a falso.
- `mtEnvTipos`. Ambiente de tipos.

El procedimiento `verificador` en caso de que se infiera el tipo `bool` en la condición calculará el mínimo supertipo común de las ramas de la expresión condicional descrita en las líneas 4 y 5. En la línea 6 se muestra el tipo `float` como resultado de la llamada al procedimiento `JOIN`, el cual es el mínimo supertipo común calculado.

```
1  > (verificador (APP
2      (LAMBDA
3      '(x)
4      '(double)
5      (OP '+ (VAR 'x) (num 2)))
6      (list (num 1))) (mtEnvTipos))
7  -> (double)
```

Código 4.22: Ejemplo del verificador de tipos en la aplicación de funciones.

En el Código 4.22 entre las líneas 1 y 6, se ejecuta el procedimiento llamado `verificador` que recibe:

- Representación abstracta de la expresión `app`, la cual se conforma con dos elementos:
  - ◇ Representación abstracta de la expresión `lambda`, la cual posee tres componentes:
    - ▷ `'(x)`. Identificador de una variable (línea 3).
    - ▷ `'(double)`. Tipo de la variable definida en el inciso anterior (línea 4).
    - ▷ `(OP '+ (VAR 'x) (num 2))`. Cuerpo del procedimiento definido (línea 5).
  - ◇ `(list (num 1))`. Parámetro con el que se llama al procedimiento descrito en la expresión `LAMBDA` (línea 6).
- `mtEnvTipos`. Ambiente de tipos inicialmente vacío.

El procedimiento `verificador` extenderá el ambiente de tipos con el tipo declarado en la línea 4 y bajo este ambiente de tipos extendido verificará el tipo asociado al cuerpo. En la línea 6 se muestra el tipo `double`, el cual es el tipo del cuerpo inferido por el verificador de tipos, siendo este el resultado de nuestra ejecución del procedimiento `verificador`.

```
1  > (verificador
2      (LAMBDA
3      '(x)
4      '(integer)
5      (OP '+ (VAR 'x) (num 2)))(mtEnvTipos))
```

```
6  -> (flecha (int) (int))
```

Código 4.23: Ejemplo del verificador de tipos en la expresión condicional.

En el Código 4.23 entre las líneas 1 y 6, se ejecuta el procedimiento llamado `verificador` que recibe:

- Representación abstracta de la expresión `lambda`, la cual se conforma con tres elementos:
  - `'(x)`. Identificador de una variable (línea 3).
  - `'(integer)`. Tipo de la variable definida en el inciso anterior (línea 4).
  - `(OP '+ (VAR 'x) (num 2))`. Cuerpo del procedimiento definido (línea 5).
- `mtEnvTipos`. Ambiente de tipos.

El procedimiento `verificador` extenderá el ambiente de tipos con el tipo declarado en la línea 4 y bajo este ambiente de tipos extendido verificará el tipo asociado al cuerpo, el cual es de tipo `flecha` correspondiente a una función en el lenguaje `MinTyRacket`. En la línea 6 se muestra el tipo `(flecha (int) (int))`, el cual es el tipo del cuerpo inferido siendo el resultado de nuestra ejecución del procedimiento `verificador`.

4.3. Algoritmo de Evaluación de MinTyRacket

Con el objetivo de implementar la semántica operacional diseñada en el Subcapítulo 3.2.2, se verá en el siguiente código la definición de los ambientes de valores que sirven para realizar las sustitusiones de identificadores por sus valores en la evaluación de un programa.

```
1  #lang plai
2
3  (define-type Env
4   [mtEnv]
5   [anEnv (name symbol?) (value MinTyRacket_Valores?) (env
        Env?)]
6   [aRecSub (name symbol?)
7           (value isInMinTyRacket_Valores?)
8           (env Env?)])
```

Código 4.24: Definición de los ambientes de valores.

En el Código 4.24, se definen los ambientes de valores de `MinTyRacket`, en la línea 4 se define el ambiente vacío `mtEnv`. En la línea 5 y 6 se define recursivamente un ambiente para funciones que se forma por un identificador `name`, un valor `value` y un ambiente. Análogamente se define un ambiente de valores `aRecSub` para funciones recursivas entre las líneas 6 y 8.

A continuación se mostrará la definición de los valores de `MinTyRacket`, los cuales son los estados finales posibles de las reglas de evaluación que se analizarán posteriormente.

```
1  #lang plai
2
3  (define-type MinTyRacket_Valores
4    [numV (n number?)]
5    [floatV (n flotnum?)]
6    [doubleV (n real?)]
7    [closureV (params (listof symbol?))
8           (body MinTyRacket?)
9           (env Env?)]
10   [booleanV (n boolean?)]
11   [nullV  (ls list?)]
12   [listV (ls cons?)] )
```

Código 4.25: Definición de los estados finales.

En el Código 4.25, entre las líneas 3 y 12 se declaran los estados finales de `MinTyRacket (MinTyRacket_Valores)`, como son `numV` que representa a los números enteros, `floatV`, el cual representa a los flotantes y `doubleV` que representa a los reales. Entre las líneas 7 y 9 se muestra la variante `closureV` asociada a las funciones, conocida como *cerradura* ya que conecta el cuerpo de cada función con las sustitusiones indicadas en su definición. Un `closureV` está formado por una lista de parámetros, indicada por una lista de identificadores `params`, así como por el cuerpo `body` donde se sustituirán tales identificadores por los valores definidos en el ambiente `env`, el cual, es un mapeo de identificadores a valores evaluados en la ejecución de un programa visto en el código anterior. En la línea 10 se exhibe otro estado final denotado por `booleanV` correspondiente a las expresiones booleanas. Mientras en la línea 11 se encuentra `nullV`, estado final asociado a la lista vacía. Por último, en la línea 12 se puede ver la expresión `listV` que representa a las listas.

Una vez definidos los valores del lenguaje `MinTyRacket` se describirá la implementación de las reglas de la semántica dinámica utilizando el procedimiento `eval`, el cual recibe como parámetro la expresión abstracta de los términos permitidos en nuestro lenguaje y un ambiente que contendrá los valores asociados a variables, operadores y aplicaciones de funciones en la evaluación respectiva. A continuación se mostrarán algunos ejemplos del procedimiento `eval`.

```
1    > (eval (RST
2      (LIST
```

```
3       (list
4        (num 1)
5        (num 2)
6        (num 3)
7        (num 4))))(mtEnv))
8    -> (listV (integerV 2) (integerV 3) (integerV 4))
```

Código 4.26: Ejemplos del procedimiento que implementa la semántica dinámica correspondiente a las operaciones con listas.

En el Código 4.26 entre las líneas 1 y 7 se ejecuta el procedimiento `eval` que recibe dos parámetros:

- Representación abstracta de la expresión `rst`, la cual contiene la representación abstracta de la expresión `LIST` formada por:
  - `(list (num 1)(num 2)(num 3)(num 4))`. Lista de representaciones abstractas (entre líneas 3 y 7).
- `mtEnv`. Ambiente de valores.

El procedimiento `eval` regresa la lista que queda al eliminar el primer elemento del componente de la representación abstracta `LIST`, resultado que podemos notar en la línea 8 `(listV (integerV 2) (integerV 3) (integerV 4))`.

```
1    > (eval (ISEMPTY?
2      (LIST
3       (list
4        (num 45)
5        (num 785)
6        (num 3))))(mtEnv))
7    ->  (booleanV #f)
```

Código 4.27: Ejemplos del intérprete que implementa la semántica dinámica correspondiente verificar si una lista está vacía.

En el Código 4.27 entre las líneas 1 y 6 se ejecuta el procedimiento `eval` que recibe como parámetros:

- Representación abstracta de la expresión `ISEMPTY?`, la cual contiene la representación abstracta de la expresión `LIST` cuyo componente es:
  - `(list (num 45)(num 785)(num 3))`. Lista de representaciones abstractas (entre líneas 3 y 6).
- `mtEnv`. Ambiente de valores.

El procedimiento `eval` devolverá verdadero si la lista representada por el primer elemento del componente de la representación abstracta `LIST` es vacía, en caso contrario se evalúa a falso como se puede ver en la línea 7 del Código 4.26.

```
1   > (eval (APP
2       (LAMBDA
3        '(x)
4        '(double)
5        (OP
6         '+
7         (num 5)
8         (OP '+ (VAR 'x) (num 6))))
9         (list (num 1))) (mtEnv))
10  -> (doubleV 12)
```

Código 4.28: Ejemplos del intérprete que implementa la semántica dinámica correspondiente a la aplicación de funciones.

En el Código 4.28 entre las líneas 1 y 9 se ejecuta el procedimiento `eval` que recibe dos parámetros:

- Representación abstracta de la expresión `app`, la cual se conforma con dos elementos:
  - ◇ Representación abstracta de la expresión `lambda`, cuyos componentes son:
    - ▷ `'(x)`. Identificador de una variable (línea 3).
    - ▷ `'(double)` que corresponde al tipo de la variable definida en el inciso anterior (línea 4).
    - ▷ `(OP '+ (num 5)(OP '+ (VAR 'x) (num 6)))`. Cuerpo del procedimiento definido (entre las línea 5 y 8).
  - ◇ `(list (num 1))`. Parámetro con el que se llama al procedimiento descrito en la expresión `LAMBDA` (línea 6).
- `mtEnv`. Ambiente de valores.

El procedimiento `eval` extenderá el ambiente de valores con el valor declarado en la línea 9 y bajo este ambiente de valores extendido verificará el valor asociado al cuerpo. En la línea 10 se muestra el valor `(doubleV 12)`, el cual es el valor del cuerpo inferido por el procedimiento `eval`.

4.4. Intérprete de MinTyRacket

En esta sección se expondrán algunos ejemplos del intérprete de `MinTyRacket`.

```
1  #lang plai
2
3  (define (interpMTR expr)
```

```
4     (if (equal? true (VERIFICADOR expr (mtEnvTipos)))
5      (getValorMTR (eval (parse expr) (mtEnv)) (parseTipos
           expr))
6      (error "Error␣de␣tipo:␣El␣tipo␣declarado␣no␣
           corresponde␣a␣las␣reglas␣de␣tipificación␣de␣
           MinTyRacket")
7      ))
```

Código 4.29: Definición del intérprete.

En Código 4.29, en la línea 3 se define el procedimiento `interpMTR`, el cual recibe como parámetro una expresión sintáctica `expr` (determinada por el programador). Se utiliza el procedimiento `VERIFICADOR` para determinar si los tipos correspondientes a la expresión abstracta obtenida por el procedimiento `parse` son subtipos de los tipos establecidos por el programador, si dicho procedimiento se evalúa a verdadero entonces se llama al procedimiento `eval` que implementará la evaluación del árbol abstracto en `MinTyRacket`. En caso de que el procedimiento `VERIFICADOR` se evalúe a falso, entonces se muestra al usuario el error de incompatibilidad de tipo a través del mensaje:

"`El tipo declarado no corresponde a las reglas de tipificación de MinTyRacket`"

A continuación se mostrarán algunos ejemplos del procedimiento `interpMTR`.

```
1 > (interpMTR '(let (x integer 1) (+ 5 (+ x 6)) (double)))
2 -> (doubleV 12)
```

Código 4.30: Ejemplo del intérprete.

En el Código 4.30, en la línea 1, se llama al procedimiento `interpMTR` con una expresión constante que recibe cuatro expresiones sintácticas concretas:

- `let`. Expresión para dividir cada caso complejo en casos mas simples.
- `(x integer 1)`. Denota la variable acotada, su tipo y valor respectivo.
- `(+ 5 (+ x 6))`. Cuerpo donde se evaluará la variable descrita en el inciso anterior.
- `(double)`. Tipo asociado a toda la expresión escrito por el programador.

Como el tipo correspondiente a la expresión abstracta obtenida por el procedimiento `parse` es subtipo del tipo establecido por el usuario, dicho procedimiento se evalúa a verdadero y entonces es llamado el procedimiento `eval`, por lo que en la línea 2 se muestra el valor `(doubleV 12)` que es el resultado obtenido asociado a la llamada de `interpMTR`, el cual pertenece al conjunto de valores definidos en `MinTyRacket`.

```
1 > (interpMTR '((lambda (x integer) (+ x 2)) 1 (float)))
2 -> (floatV 3)
```

Código 4.31: Ejemplos del intérprete de nuestro lenguaje en la aplicación de funciones

En el Código 4.31, en la línea 1 se ejecuta el procedimiento `interpMTR` utilizando como parámetro una expresión constante que recibe tres expresiones sintácticas concretas:

- `(lambda (x integer) (+ x 2))`. Denota una función anónima.
- `1`. Valor de ligado con que se ejecutará la aplicación de función.
- `(float)`. Tipo indicado por el programador.

Ya que el procedimiento `VERIFICADOR` se evalúa a verdadero en la línea 2 se puede ver el valor `(floatV 3)`, el cual es el resultado de la evaluación asociada a la sentencia con la que se llamó el procedimiento `interpMTR`.

```
1 > (interpMTR '(fix (fac (lambda (n integer)
2       (if (== n 0)  1   ( * n (fac (+ n -1))))) (integer ->
            float))
3       (fac 6)(float)))
4 -> (floatV 720)
```

Código 4.32: Ejemplos de nuestro intérprete en aplicaciones de funciones recursivas.

En el Código 4.32, en la línea 1, se llama al procedimiento `interpMTR` utilizando como parámetro una expresión sintáctica que recibe cuatro expresiones:

- `fix`. Expresión que denota la definición de funciones recursivas.
- `(fac (lambda (n integer) (if(== n 0) 1 (* n(fac(+ n -1)))))(integer -> float))`. Cuerpo de la función recursiva.
- `(fac 6)`. Aplicación de la función recursiva.
- `(float)`. Tipo escrito por el usuario respecto al resultado de la llamada al procedimiento `interpMTR`.

En la línea 2 se muestra la expresión`(floatV 720)`, la cual es el resultado de la evaluación de la expresión sintáctica llevada a cabo por el procedimiento `eval` debido a que el verificador de tipos de `MinTyRacket` (`VERIFICADOR`) lo evaluó a verdadero.

4.5. Análisis de propiedades

A continuación se exhibirán las principales propiedades que presenta el intérprete `MinTyRacket`, un lenguaje perteneciente al paradigma funcional tipificado estáticamente.

4.5.1. Detección temprana de errores

Con el objetivo de permitir la menor cantidad de errores en tiempo de ejecución, por ejemplo, sumar expresiones de tipo booleano, resulta muy plausible detectar tales errores antes de la ejecución del programa. A continuación se muestra un ejemplo de detección de error en tiempo de compilación.

```
1 > (interpMTR '(fix (misterio (lambda (n boolean)
2       (if (== n true)  n  ( * n (misterio n)))) (boolean ->
            boolean)) (misterio false)(boolean)) )
3 -> Error de tipo: se esperaba un número, se ha dado una
     expresión de tipo boolean.
```

Código 4.33: Ejemplo donde se evita sumar un booleano.

En el Código 4.33, en las línea 1 y 2 se llama al procedimiento `interpMTR` utilizando como parámetros:

- `fix`. Expresión que denota la definición de funciones recursivas.
- `(misterio (lambda (n boolean) (if (== n true) n ( * n (misterio n)))) (boolean -> boolean))`. Cuerpo de la función recursiva, el cual recibe una variable de ligadura de tipo `boolean` así como posee la subexpresión `( * n (misterio n))`, donde el operador `*` denota una multiplicación, la cual debe recibir operandos de tipo numérico.
- `(misterio false)`. Aplicación de la función recursiva, la cual recibe la expresión `false` cuyo tipo es booleano.
- `(boolean)`. Tipo escrito por el usuario respecto al resultado de la llamada al procedimiento `interpMTR`.

Debido a que la llamada al procedimiento recursivo `misterio` a través de la expresión `(misterio false)` recibe la expresión `false` cuyo tipo es booleano, entonces el procedimiento `VERIFICADOR` (verificador de tipos de `MinTyRacket`) devuelve en la línea 3 el siguiente mensaje:

"`Error de tipo: se esperaba un número, se ha dado una expresión de tipo boolean`"

Es importante notar que si el procedimiento `VERIFICADOR` (verificador de tipos de `MinTyRacket`) no arrojara errores antes de la ejecución de un programa entonces el procedimiento `misterio` entraría en un ciclo infinito, restricción que además depende del programador detectar y corregir. A continuación se

mostrará un ejemplo de un ciclo infinito implementado en el lenguaje `Racket`, el cual pudo ser evitado por la detección de errores en tiempo de compilación.

```
1  #lang plai
2
3  (define misterio
4   (lambda (n)
5    (if
6     (equal? n true) n
7     ( * n (misterio n)))))
8
9  (display (misterio false))
10
11 ->
```

Código 4.34: Ejemplo donde no se detectan errores en tiempo de compilación.

En el Código 4.34 [10], en las línea 3 y 7 se define el procedimiento `misterio`, la expresión `lambda` recibe como parámetro una variable de ligadura llamada `n`. En la línea 5 se muestra el cuerpo de la función recursiva en las mismas condiciones vistas en el Código 4.33. En la línea 9 se llama al procedimiento `misterio` con parámetro `false`, debido a la forma del cuerpo de la función recursiva, el resultado de este programa se queda atrapado en un ciclo infinito, situación que en cualquier escenario es indeseable.

4.5.2. Mantenimiento del código

Los programas en el lenguaje `MinTyRacket` proporcionan una manera muy efectiva de documentación, situación que los hace más legibles y en consecuencia facilita el mantenimiento del código, ya que las líneas de código pueden crecer indefinidamente pudiendo poseer una abstracción que le impida al programador su manipulación.

Otro beneficio del verificador de tipos de `MinTyRacket` es que permite encontrar errores de tipo en tiempo de compilación, lo cual reduce el esfuerzo del programador para corregir los errores de tipo, acortando las posibilidades de errores provenientes de flujos impredecibles.

A su vez `MinTyRacket` proporciona una documentación del código debido a que tener tipos explícitos aporta, sin haber ejecutado el programa, información útil sobre el comportamiento del programa al programador.

4.5.3. Propiedad de seguridad

En el Capítulo 3 se han demostrado dos propiedades del sistema de tipos diseñado para `MinTyRacket`, la primera de ellas se llama **preservación**, la cual establece que las transiciones de evaluación conservan sus tipos, mientras la segunda propiedad se llama **progreso**, la cual asegura que todas las expresiones

bien tipificadas o son valores, en tal caso su evaluación termina con éxito o en caso de que no sean valores seguirán evaluándose, es decir, la evaluación no puede detenerse en un estado para el cuál no exista una transición de evaluación posible, por lo que dichas propiedades convierten a `MinTyRacket` en un lenguaje confiable.

4.5.4. Expresividad

Existen situaciones donde los programadores pueden utilizar funciones o variables ligadas a cualquier valor de cualquier tipo, especialmente donde el flujo en tiempo de ejecución es impredecible. A continuación se describirán algunos ejemplos del nivel de flexibilidad tanto del lenguaje `MinTyRacket` como de `Racket`.

```
1  > (interpMTR '((lambda (x integer) (if (== x 0) (error)
       (/ 5 x))) 0  (float) ))
2  -> (ERROR)
```

Código 4.35: Ejemplo de notificación de excepción en MinTyRacket.

En el Código 4.35, en la línea 1 se ejecuta el procedimiento `interpMTR` utilizando como parámetro una expresión constante que recibe tres expresiones sintácticas concretas:

- `(lambda (x integer) (if (== x 0)(error) (/ 5 x)))`. Definición de una función.
- `0`. Valor de acotamiento con que se ejecutará la aplicación de la función definida en la expresión anterior.
- `(float)`. Tipo asociado a la expresión constante indicado por el programador.

De acuerdo al cuerpo de la función definida en la expresión (`(if (== x 0) (error) (/ 5 x))`), si el valor ligado a la variable x es igual a `0`, entonces regresa la expresión `(ERROR)`, la cual denota una forma excepcional de terminar programas en `MinTyRacket`, en caso contrario se calcula el valor obtenido al dividir 5 entre el valor ligado a la variable `x`. Es por ello que en la línea 3 se muestra la expresión `(ERROR)` como respuesta a la llamada al procedimiento `interpMTR`.

Tanto el algoritmo de tipificación como el de subtipificación juegan un papel trascendental en el diseño e implementación de la expresión `(error)`, ya que según la regla de tipificación tal expresión posee tipo `Min`, mientras el algoritmo de subtipificación de `MinTyRacket` dice que el tipo `Min` es subtipo de cualquier otro tipo definido en dicho lenguaje, entonces utilizando el cálculo del mínimo supertipo común en expresiones condicionales `if` se obtiene una forma para expresar una evaluación emergente a este tipo de programas.

A continuación se exhibirá una función análoga implementada en el lenguaje `Racket`.

```
1  #lang plai
2
3   ((lambda (x)
4     (if
5      (equal? x 0)
6      (error "La división entre cero no está definida")
7      ( / 5 x ))) 0)
8
9   -> La división entre cero no está definida
```

Código 4.36: Ejemplo de notificación de excepción en Racket.

En el Código 4.36, en las línea 3 y 7 se llama a la aplicación de una función formada por la expresión `lambda` que recibe como parámetro una variable de ligadura llamada `x`. Entre las líneas 4 y 7 vemos el cuerpo de la expresión `lambda` en condiciones idénticas a las analizadas en el Código 4.35. En la línea 9 se muestra la cadena `La división entre cero no está definida`, la cual es el resultado de la aplicación de función definida.

Comparando los dos ejemplos anteriores se puede concluir que la capacidad de definir maneras emergentes de evaluar programas es muy similar entre `Racket` y `MinTyRacket`.

A continuación se mostrarán dos ejemplos acerca de la capacidad de nuestro lenguaje para utilizar funciones y variables ligadas a cualquier valor de cualquier tipo, el primer ejemplo es implementado en `MinTyRacket` mientras el segundo se ha escrito en `Racket` .

```
1
2  > (interpMTR '(fix (satisfacible
3         (lambda (n boolean)
4          (if
5            (== n true)
6             n
7            (satisfacible (n #t))))
8         (boolean -> boolean))
9         (satisfacible (lambda (y boolean)(lambda (z boolean
              ) (== y z))))(boolean)))
10  -> Error de tipo: el tipo declarado no corresponde a las
       reglas de tipificación de MinTyRacket
```

Código 4.37: Ejemplo de tipo indeterminado en MinTyRacket.

En el Código 4.37, en la línea 1 se llama al procedimiento `interpMTR`, el cual recibe como parámetro una expresión sintáctica que recibe cuatro expresiones:

- `fix`. Expresión que denota la definición de funciones recursivas.
- `(satisfacible (lambda (n boolean)(if (== n true) n (satisfacible (n #t))))(boolean -> boolean))`. Cuerpo de la función recursiva (entre líneas 2 y 8).
- `(satisfacible (lambda (y boolean)(lambda (z boolean) (== y z))))`. Aplicación de la función recursiva definida (línea 8).
- `(boolean)`. Tipo escrito por el usuario respecto al resultado de la llamada a la función recursiva `satisfacible`

Debido a que el parámetro con el cual se llama a la función recursiva llamada `satisfacible` no es un valor booleano sino una función booleana denotada por la expresión (`(lambda ((y boolean))(lambda ((z boolean)) (== y z)))`) entonces el verificador de tipos de `MinTyRacket` (`VERIFICADOR`) notifica al programador un error de tipo se puede ver en la línea 9:

"`Error de tipo: el tipo declarado no corresponde a las reglas de tipificación de MinTyRacket`"

A continuación se describirá una función análoga implementada en el lenguaje `Racket`.

```
1   #lang plai
2
3     (define satisfacible
4          (lambda (n)
5                  (if (equal? n true)
6                   n
7                  (satisfacible (n #t)))))
8
9      (display (satisfacible (lambda (y)(lambda (z) (equal?
           y z)))))
10
11  -> #t
```

Código 4.38: Ejemplo de tipo indeterminado en Racket

En el Código 4,31, en las línea 3 y 7 se define el procedimiento recursivo `satisfacible`, la expresión `lambda` recibe como parámetro una variable de ligadura llamada `n`. En la línea 5 se puede ver el cuerpo de la función recursiva en las mismas condiciones analizadas en el Código 4.37. En la línea 9 se llama al procedimiento `satisfacible` con la función booleana `(lambda (y)(lambda (z) (equal? y z)))` como parámetro. Considerando la forma del cuerpo de la función recursiva el intérprete del lenguaje `Racket` evalúa la función booleana utilizando la constante booleana `#t` como parámetro. Debido a que la expresión `( ((lambda (y) (lambda (z) (equal? y z)))#t  )#t)` se evalúa a verdadero entonces en la línea 11 se puede ver que el procedimiento `satisfacible` se evalúa a verdadero (`#t`).

Al introducir subtipos en `MinTyRacket`, se han diseñado e implementado reglas que le permiten a los tipos de los parámetros de procedimientos no necesariamente ser fijos, es decir, se ha alcanzado mayor flexibilidad ya que permiten aceptar una mayor cantidad de programas que se comportan correctamente.

4.6. Comparación general de algunas propiedades

Basados en el análisis de las propiedades de `MinTyRacket` realizado en el Subcapítulo 4.5 se presenta la Tabla 4.1, la cual exhibe las propiedades principales asociadas a la verificación de tipos correspondientes tanto al intérprete de `Racket` como al intérprete `MinTyRacket`.

Intérprete	Detección temprana de errores de tipo	Propiedad de Seguridad	Mayor documentación aportada por código fuente	Mayor rapidez en corrección de errores de tipo	Mayor capacidad de manejo de excepciones	Mayor expresividad	Mejor adaptación en tiempo de ejecución
Racket					•	•	•
MinTyRacket	•	•	•	•			

Figura 4.1: Comparación de algunas propiedades de los dos intérpretes.

En la Tabla 4.1 se puede observar que el intérprete `MinTyRacket` brinda la propiedad de *seguridad*. Asimismo se puede notar que `MinTyRacket` brinda una versión de un lenguaje de programación funcional con evaluación glotona capaz de detectar errores en tiempo de compilación. Incluso aporta documentación al código fuente lo que mejora su mantenimiento. También ofrece una manera más rápida de detectar errores de tipo que el intérprete de `Racket`. No obstante, `Racket` posee una mayor capacidad para manejar excepciones, así como una mayor expresividad, es decir, es posible definir una cantidad mayor de funciones que en `MinTyRacket`. Además, `Racket` se adapta mejor a los posibles datos que se presentan en tiempo de ejecución.

En este capítulo se ha descrito de forma general la implementación del intérprete `MinTyRacket`, en el anexo B se puede ver el código completo de la implementación descrita. Posteriormente se han analizado tanto sus propiedades como las propiedades del intérprete de `Racket` en torno a la verificación de tipos, dicho análisis se compone por ejemplos de códigos análogos evaluados por ambos intérpretes.

Capítulo 5

Conclusiones

Algunos científicos interesados en el área de lenguajes de programación notaron la conexión que existía entre los sistemas de tipos con aquellos expuestos en el área de la lógica matemática, relación que permitió el desarrollo de definiciones formales en torno a éstos. Es así como en el Capítulo 2 se ha presentado una introducción a los sistemas de tipos.

En el Capítulo 3 se ha diseñado un lenguaje llamado `MinTyRacket` usando la técnica usual de definición de una gramática libre de contexto en **forma de Backus-Naur** para establecer la sintaxis concreta y abstracta. También se ha definido inductivamente la semántica estática través de juicios de la forma $\Gamma \vdash t : \tau$. Se ha creado una representación abstracta de las expresiones de `MinTyRacket` que ha permitido utilizar el método de inducción estructural para formalizar la demostración de propiedades fundamentales como *reflexividad*, *transitividad* y *seguridad* [15].

A continuación se muestran los principales elementos por los que `MinTyRacket` ha sido diseñado con el objetivo de implementar su análisis de tipos en tiempo de compilación sin convertirlo en un lenguaje muy rígido.

- Se integran elementos similares a los que forman a `Racket` como la condicional `if`, definición tanto de funciones recursivas `fix` como de funciones anónimas `lambda`, aplicación de funciones , expresiones que permiten dividir expresiones complejas en expresiones más simples `let`, así como representaciones y operaciones asociadas a listas como `rst`, `fst`, `cons`, `isempty?` y `null`.
- Se han añadido tipos booleanos como `true` y `false`, tipos numéricos como `integer`, `double` y `float` así como tipos asociados a funciones y listas.
- Con el objetivo de alcanzar mayor flexibilidad se han diseñado reglas de subtipificación tanto semántica como algorítmicamente equivalentes, estableciendo que nuestro sistema de tipos es computable.

- Se ha diseñado e implementado un algoritmo que permite calcular en caso de que exista, el mínimo supertipo común de dos tipos.
- Se ha involucrado la expresión `error` para formalizar terminaciones que involucren cálculos no razonables denotando al programador que un programa aborta.
- Se ha agregado el tipo `Min` para expresar que algunas operaciones como abortar programas no regresarán valor alguno, así como le indican al verificador implementado en este trabajo de tipos que algunas expresiones pueden ser usadas de forma segura en contextos que esperan cualquier tipo de valor.
- Se ha incluido el tipo `Max` para proporcionar un mayor número de funciones en nuestro lenguaje.

Con base en el análisis de propiedades realizado en el Capítulo 4 se puede concluir que `MinTyRacket` brinda una versión de un lenguaje de programación funcional con evaluación glotona que ofrece los siguientes elementos:

- Capacidad para detectar errores en tiempo de compilación.
- Documentación aportada por el código fuente.
- Facilidad de mantenimiento de código.
- Rapidez en la corrección de errores de tipo.
- Manejo de excepciones para indicar evaluaciones emergentes en un programa.
- Algoritmo de subtipificación que le aporta más expresividad.

Apéndice A

Redundancia de la Regla 71:(Sub-trans)

En este apéndice se incluyen los casos que completan la demostración del **Lema 10** (***Redundancia de la*** Regla 71:(Sub-trans)).

Caso (- / Regla 60:(Sub-float)) Sean $\upsilon = \texttt{float}$ $\tau = \texttt{double}$
Si la subderivación izquierda termina con la Regla 62:(Sub-integer2), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se infiere que la regla final de subtipificación en esta subderivación debe ser la Regla 62:(Sub-integer2), aplicando la Regla 61:(Sub-integer1) se obtiene el resultado deseado.
Por otro lado, si la subderivación izquierda termina con la Regla 65:(Sub-float2), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación vemos que la regla final de subtipificación en esta subderivación debe ser la Regla 65:(Sub-float2), aplicando la Regla 60:(Sub-float) se obtiene el resultado deseado.

Caso (Regla 61:(Sub-integer1) / -) Sean $\sigma = \texttt{integer}$ $\upsilon = \texttt{double}$
Si la subderivación derecha termina con la Regla 66:(Sub-double), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación vemos que la regla final de subtipificación en esta subderivación debe ser la Regla 66:(Sub-double), aplicando la Regla 61:(Sub-integer1) se obtiene el resultado deseado.

Caso (- / Regla 61:(Sub-integer1)) Sean $\upsilon = \texttt{integer}$ $\tau = \texttt{double}$
Si la subderivación izquierda termina con la Regla 64:(Sub-integer3), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se infiere que la regla final de subtipificación en esta subderivación debe ser la Regla 64:(Sub-integer3), aplicando la Regla 61:(Sub-integer1) es posible obtener el resultado deseado.

Caso (Regla 62:(Sub-integer2) / -) Sean $\sigma =$ `integer` $\upsilon =$ `float`
Si la subderivación derecha termina con la Regla 65:(Sub-float2), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación concluimos que la regla final de subtipificación en esta subderivación debe ser la Regla 65:(Sub-float2), aplicando la Regla 62:(Sub-integer2) se obtiene el resultado deseado.

Caso (- / Regla 62:(Sub-integer2)) Sean $\upsilon =$ `integer` $\tau =$ `float`
Si la subderivación izquierda termina con la Regla 64:(Sub-integer3), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se concluye que la regla final de subtipificación en esta subderivación debe ser la Regla 64:(Sub-integer3), aplicando la Regla 62:(Sub-integer2) se obtiene el resultado deseado.

Caso (Regla 63:(Sub-boolean) / -) Sean $\sigma =$ `boolean` $\upsilon =$ `boolean`
Si la subderivación derecha termina con la Regla 63:(Sub-boolean), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se infiere que la regla final de subtipificación en esta subderivación debe ser la Regla 63:(Sub-boolean), aplicando la Regla 63:(Sub-boolean) se obtiene el resultado deseado.

Caso (- / Regla 63:(Sub-boolean)) Sean $\upsilon =$ `boolean` $\tau =$ `boolean`
Si la subderivación izquierda termina con la Regla 63:(Sub-boolean), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación concluimos que la regla final de subtipificación en esta subderivación debe ser la Regla 63:(Sub-boolean), aplicando la Regla 63:(Sub-boolean) se obtiene el resultado deseado.

Caso (Regla 64:(Sub-integer3) / -) Sean $\sigma =$ `integer` $\upsilon =$ `integer`
Si la subderivación derecha termina con la Regla 64:(Sub-integer3), por hipótesis de inducción es posible suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se infiere que la regla final de subtipificación en esta subderivación debe ser la Regla 64:(Sub-integer3), aplicando la Regla 64:(Sub-integer3) se obtiene el resultado deseado.

Caso (- / Regla 64:(Sub-integer3)) Sean $\upsilon =$ `integer` $\tau =$ `integer`
Si la subderivación izquierda termina con la Regla 64:(Sub-integer3), por hipótesis de inducción es posible suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se concluye que la regla final de subtipificación en esta subderivación debe ser la Regla 64:(Sub-integer3), aplicando la Regla 64:(Sub-integer3) se obtiene el resultado deseado.

Caso (Regla 65:(Sub-float2) / -) Sean $\sigma =$ `float` $\upsilon =$ `float`
Si la subderivación derecha termina con la Regla 65:(Sub-float2), por hipótesis de inducción es posible suponer que dicha subderivación no utiliza la Regla

71:(Sub-trans). Analizando las reglas de tipificación se concluye que la regla final de subtipificación en esta subderivación debe ser la Regla 65:(Sub-float2), aplicando la Regla 65:(Sub-float2) se obtiene el resultado deseado.

Caso (- / Regla 65:(Sub-float2)) Sean $\upsilon = \texttt{float}$ $\tau = \texttt{float}$
Si la subderivación izquierda termina con la Regla 65:(Sub-float2), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación se infiere que la regla final de subtipificación en esta subderivación debe ser la Regla 65:(Sub-float2), aplicando la Regla 65:(Sub-float2) se obtiene el resultado deseado.

Caso (Regla 66:(Sub-double) / -) Sean $\sigma = \texttt{double}$ $\upsilon = \texttt{double}$
Si la subderivación derecha termina con la Regla 66:(Sub-double), por hipótesis de inducción es posible suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación concluimos que la regla final de subtipificación en esta subderivación debe ser la Regla 66:(Sub-double), aplicando la Regla 66:(Sub-double) se obtiene el resultado deseado.

Caso (- / Regla 66:(Sub-double)) Sean $\upsilon = \texttt{double}$ $\tau = \texttt{double}$
Si la subderivación izquierda termina con la Regla 66:(Sub-double), por hipótesis de inducción se puede suponer que dicha subderivación no utiliza la Regla 71:(Sub-trans). Analizando las reglas de tipificación tenemos que la regla final de subtipificación en esta subderivación debe ser la Regla 66:(Sub-double), aplicando la Regla 66:(Sub-double) se obtiene el resultado deseado.

Caso (Regla 68:(Sub-arr) / Regla 68:(Sub-arr)) Sean $\sigma = \sigma_1 \to \sigma_2$
$\upsilon = \upsilon_1 \to \upsilon_2$ $\tau = \tau_1 \to \tau_2$ $\upsilon_1 <: \sigma_1$ $\sigma_2 <: \upsilon_2$ $\tau_1 <: \upsilon_1$ $\upsilon_2 <: \tau_2$

Usando la Regla 71:(Sub-trans), se pueden construir derivaciones de la forma $\tau_1 <: \sigma_1$ y $\sigma_2 <: \tau_2$ de las subderivaciones dadas. Como estas subderivaciones serían mas pequeñas que la derivación original, entonces al aplicar la hipótesis de inducción se obtienen derivaciones de $\tau_1 <: \sigma_1$ y $\sigma_2 <: \tau_2$ que no utilizan la Regla 71:(Sub-trans). Combinando dichas subderivaciones con la Regla 68:(Sub-arr) se obtiene una derivación de $\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2$ que no usa la Regla 71:(Sub-trans).

Caso (Regla 67:(Sub-list) / Regla 67:(Sub-list)) Sean $\sigma = \texttt{List}\ \sigma_1$
$\upsilon = \texttt{List}\ \upsilon_1$ $\tau = \texttt{List}\ \tau_1$ $\sigma_1 <: \upsilon_1$ $\upsilon_1 <: \tau_1$

Usando la Regla 71:(Sub-trans), es posible construir derivaciones de $\sigma_1 <: \tau_1$ de las subderivaciones dadas. Como estas subderivaciones serían mas pequeñas que la derivación original, entonces se puede aplicar la hipótesis de inducción para obtener derivaciones de $\sigma_1 <: \tau_1$ que no utilizan la Regla 71:(Sub-trans). Combinando dichas subderivaciones con la Regla 67:(Sub-list) se obtiene una derivación de $\texttt{List}\ \sigma_1 <: \texttt{List}\ \tau_1$ que no usa la Regla 71:(Sub-trans).

Apéndice B

Código Fuente

B.1. DefinicionesMTR.rkt

```
#lang plai



#lang plai


#|Definición de las expresiones|#
(define-type MinTyRacket
  [NULL (tipo tipo_elem?)]
  [FST(ls MinTyRacket?)]
  [ISEMPTY? (ls MinTyRacket?)]
  [CONST(elem MinTyRacket?) (ls MinTyRacket?)]
  [RST (ls MinTyRacket?)]
  [ERROR]
  [OP (op symbol?) (lhs MinTyRacket?) (rhs MinTyRacket?)]
  [VAR (nombre symbol?)]
  [LAMBDA (argus (listof symbol?)) (types (listof symbol?))
      (body MinTyRacket?)]
  [IF (cond MinTyRacket?) (true MinTyRacket?) (else
     MinTyRacket?)]
  [APP (f MinTyRacket?) (args (listof MinTyRacket?))]
  [FIX (varlig symbol?) (nombre MinTyRacket?) (cuerpolig
     MinTyRacket?)]
  [true_]
  [false_]
  [LIST (ls  list?)]
  [bool (n  boolean?)]
  [num (n  integer?)]
  [flot (n  flonum?)]
  [doub (n  real?)])
```

```
29
30
31 #|Definición de la operación aritmética -|#
32 (define (resta c d)
33   ((getSupertipo c d) (- (typeV-n c) (typeV-n d))))
34
35
36 #|Definición de la operación aritmética división|#
37 (define (division m n)
38   (cond
39     ([equal? (typeV-n n) 0] (ERROR))
40     (else (cond
41             ([integer? (/ (typeV-n m) (typeV-n n))]
42             (integerV (/ (typeV-n m) (typeV-n n))))
43             ([real? (/ (typeV-n m) (typeV-n n))]
44             (doubleV (/ (typeV-n m) (typeV-n n))))
45             ))))
46
47
48 #|Definición de la operación booleana menor que|#
49 (define (menorque a b)  (booleanV (< (typeV-n a) (typeV-n b)
       )))
50
51
52 #|Definición de la operación booleana menor que|#
53 (define (mayorque a b)  (booleanV (> (typeV-n a) (typeV-n b)
       )))
54
55
56 #|Definición de la operación aritmética multiplicación|#
57 (define (multiplicacion j k)   ((getSupertipo j k) ( * (
       typeV-n j) (typeV-n k))))
58
59
60 #|Definición de la operación aritmética suma|#
61 (define (suma a b)  ((getSupertipo a b) (+ (typeV-n a) (
       typeV-n b))))
62
63
64 #|Definición de la operación auxiliar de igualdad|#
65 (define (igualdad a b)
66   (booleanV  (equal? (getComponente a) (getComponente b))))
67
68
69 #|Definición de la operación que obtiene los componentes de
       los valores|#
70 (define (getComponente expr)
71      (cond
72        ((MinTyRacket? expr)   (MintyRacket-n expr) )
73        ((MinTyRacket_Valores? expr)       (typeV-n expr) )
```

```
74          (else (append "La␣operación␣no␣puede␣realizarse␣con␣
                tales␣argumentos"))))
75
76
77  #|Definición del conjunto de operaciones binarias|#
78  (define opbinarias (list
79                       (list '+ suma)
80                       (list '< menorque)
81                       (list '> mayorque)
82                       (list '== igualdad)
83                       (list '- resta)
84                       (list '* multiplicacion)
85                       (list '/ division)))
86
87
88  #|Definición de los valores|#
89  (define-type MinTyRacket_Valores
90    [integerV (n integer?)]
91    [floatV (n flonum?)]
92    [doubleV (n real?)]
93    [closureV (params (listof symbol?))
94              (body MinTyRacket?)
95              (env Env?)]
96    [booleanV (n boolean?)]
97    [listV  (ls list?)])
98
99
100 #|Procedimiento que comprueba cuando se trata de un valor|#
101 (define (isInMinTyRacket_Valores? lo)
102   (and (box? lo)
103        (MinTyRacket_Valores? (unbox lo))))
104
105
106 #|Definición del ambiente de valores|#
107 (define-type Env
108   [mtEnv]
109   [anEnv (name symbol?) (value MinTyRacket_Valores?) (env
          Env?)]
110   [aRecSub (name symbol?)
111            (value isInMinTyRacket_Valores?)
112            (env Env?)])
113
114
115 #|Definición de los tipos elementales|#
116 (define-type tipo_elem
117   [int]
118   [float]
119   [double]
120   [booln]
121   [Min]
```

```
122   [Max]
123   )
124
125
126 #|Definición de los tipos|#
127 (define-type tipo
128   [lista (elementos tipo_elem?)]
129   [flecha (antec tipo_elem?) (consec tipo_elem?)])
130
131
132 #|Definición de valores simples|#
133 (define (typeV-n value)
134   (type-case MinTyRacket_Valores value
135     [listV (n) n]
136     [integerV (n) n]
137     [floatV (n) n]
138     [doubleV (n) n]
139     [booleanV (n) n]
140     [consV (n) n]
141     {else (append "El␣tipo␣no␣ha␣sido␣definido")} ))
142
143
144 (define (MintyRacket-n exp)
145   (type-case MinTyRacket exp
146     [num (n) n]
147     [flot (n) n]
148     [doub (n) n]
149     [bool (n) n]
150     {else (append "La␣expresión␣no␣permite␣obtener␣sus␣
          componentes")} ))
151
152
153 #|Definición de tablas hash con los valores asociados|#
154 (define htV (make-hash))
155   (hash-set! htV integerV integerV)
156   (hash-set! htV floatV floatV)
157   (hash-set! htV doubleV doubleV)
158   (hash-set! htV booleanV booleanV)
159   (hash-set! htV integerV-n integerV-n)
160   (hash-set! htV floatV-n floatV-n)
161   (hash-set! htV doubleV-n doubleV-n)
162   (hash-set! htV booleanV-n booleanV-n)
163   (hash-set! htV float floatV)
164   (hash-set! htV double doubleV)
165   (hash-set! htV booln booleanV)
166   (hash-set! htV int integerV)
167   (hash-set! htV list listV)
168   (hash-set! htV #t true)
169   (hash-set! htV #f false)
170
```

```
171
172 (define (exprV expresionV)
173   (cond
174     [(integerV? expresionV)(hash-ref htV integerV)]
175     [(floatV? expresionV)(hash-ref htV floatV)]
176     [(doubleV? expresionV)(hash-ref htV doubleV)]
177     [(booleanV? expresionV)(hash-ref htV booleanV)]))
178
179 (define (exp-V expresionV)
180   (cond
181     [(integerV? expresionV)(hash-ref htV integerV-n)]
182     [(floatV? expresionV)(hash-ref htV floatV-n)]
183     [(doubleV? expresionV)(hash-ref htV doubleV-n)]
184     [(booleanV? expresionV)(hash-ref htV booleanV-n)]
185     ))
186
187 (define htVJ (make-hash))
188 (hash-set! htVJ integerV 0)
189 (hash-set! htVJ floatV 1)
190 (hash-set! htVJ doubleV 2)
191 (hash-set! htVJ booleanV 3)
192
193
194 #|Definición de operaciones que generan la jerarquía de
      valores|#
195 (define (jerarquiaValores a b)
196   (if (< (hash-ref htVJ a) (hash-ref htVJ b)
197       )
198       (hash-ref htV b) (hash-ref htV a)))
199
200
201 (define (gettypeV value)
202   (type-case MinTyRacket_Valores value
203     [integerV (n) (hash-ref htV integerV)]
204     [floatV (n) (hash-ref htV floatV)]
205     [doubleV (n) (hash-ref htV doubleV)]
206     {else (append "El␣tipo␣no␣ha␣sido␣definido")} ))
207
208
209 (define (getSupertipo a b)
210   (jerarquiaValores (gettypeV a)(gettypeV b)))
211
212
213 #|Definición de asocia un valor aritmético a una expresión
      de MinTyRacket|#
214 (define (asociarValorMTR_Tipo value)
215  (type-case tipo_elem value
216     [int () (hash-ref htV int)]
217     [float () (hash-ref htV float)]
218     [double () (hash-ref htV double)]
```

```
219     {else (append "El␣tipo␣no␣ha␣sido␣definido")} ))
220
221
222 #|Definición de asocia un valor a una expresión de
        MinTyRacket|#
223 (define (getValorMTR evalResult tipoProg)
224   (cond
225     [(ERROR? evalResult) evalResult]
226     [(or (float? tipoProg) (double? tipoProg))
227      (integerV? evalResult)
228      ((hash-ref htV (asociarValorMTR_Tipo tipoProg))
229       (+(typeV-n evalResult) 0.0))]
230     [(floatV? evalResult)
231      ((hash-ref htV (asociarValorMTR_Tipo tipoProg))
             evalResult)]
232     [(doubleV? evalResult)
233      ((hash-ref htV (asociarValorMTR_Tipo tipoProg))
             evalResult)]
234     [(booleanV? evalResult)
235      ((hash-ref htV booleanV)evalResult)]
236     [(boolean? evalResult)
237      ((hash-ref htV booleanV)(hash-ref htV evalResult))]
238     [(list? evalResult)
239      (listV (construirList getValorMTR evalResult (
             getTipoProgList tipoProg)))]
240     [(list? tipoProg)(getValorMTR evalResult (second
           tipoProg))]
241     [else evalResult]
242     ))
243
244
245 (define (construirList fn ls getTipoProgListResul)
246   (if (null? ls)
247       '()
248       (cons (fn (car ls) getTipoProgListResul)
249             (construirList fn (cdr ls) getTipoProgListResul)
                  )))
250
251
252 #|Definición de asocia un valor listV a una expresión de
        MinTyRacket|#
253 (define (getTipoProgList tipoProg)
254   (type-case tipo tipoProg
255     [lista (n)
256        (cond
257          [(integer? n) (int)]
258          [(float? n) (float)]
259          [(double? n) (double)]
260          [else "no␣existe␣asociación␣al␣tipo␣indicado␣por␣el
                 ␣programador"]
```

```
261           )
262       ]
263       [else (append "El␣tipo␣no␣ha␣sido␣definido")] ))
```

B.2. ParserMTR.rkt

```
1  #lang plai
2
3
4
5
6  (require "DefinicionesMTR.rkt")
7
8
9
10
11 #|Regresa la lista de identificadores de acotamiento|#
12 (define Arg
13   (lambda (ls)
14     (if (null? ls) '()
15         (cons (car ls) '() ))))
16
17
18
19 #|Regresa la lista de valores de ligado|#
20 (define val
21   (lambda (ls)
22     (if (null? ls) '()
23         (cons (third ls) '()  ))))
24
25
26
27 #|Regresa la lista de tipos|#
28 (define types
29   (lambda (ls)
30     (if (null? ls) '()
31         (cons (second  ls) '() ))))
32
33
34
35 #|Obtiene la representación abstracta de cada elemento de
       una lista|#
36 (define identidadAbstracta
37   (lambda (lista)
38     (if (= (length lista) 0)
39         null
40         ( append  (identidadAbstracta (remove (last lista)
               lista) )    (list (parse(last lista)))))))
```

```
41
42
43
44 #|Función que traduce expresiones escritas en sintaxis
       concreta a sintaxis abstracta|#
45 (define (parse sexp)
46   (cond
47
48     ((boolean? sexp) (bool sexp))
49     ((integer? sexp) (num sexp))
50     ((flonum? sexp) (flot sexp))
51     ((real? sexp) (double sexp))
52     ((equal?  sexp 'true) (parse true))
53     ((equal? sexp 'false) (parse false))
54     ((symbol? sexp) (VAR sexp))
55     ((equal? (first sexp) 'null)  (NULL (parseTiposAuxiliar
           (first(second sexp))) ))
56     ((and (not(equal? sexp 'cons))
57                      (and (symbol? sexp)
58                      (and (not(equal? sexp 'false_)))))
59                      (VAR sexp))
60     ((equal? (first sexp) 'fst) (FST  (parse (second sexp)))
           )
61     ((equal? (first sexp) 'list)  (LIST (identidadAbstracta
           (rest sexp))))
62     ((equal? (first sexp) 'rst) (RST  (parse (second sexp)))
           )
63     ((equal? (first sexp) 'isempty?)
64                              (ISEMPTY?  (parse (second sexp)))
                                    )
65     ((equal? (first sexp) 'cons) (CONST (parse (second sexp)
           ) (parse (third sexp)) ))
66     ((equal?(first sexp) 'error ) (ERROR))
67     ((equal? (first sexp) 'let)
68                            (APP (LAMBDA (Arg (second sexp))
69                             (types (second sexp))
70                             (parse (third sexp)))
71                             (map parse(rest (append '('a) (val
                                    (second sexp)))))))
72     ((equal? (first sexp) 'if)
73                            (IF (parse (second sexp))
74                             (parse (third sexp))
75                             (parse (fourth sexp))))
76     ((equal? (first sexp) 'lambda)
77                            (LAMBDA (Arg (second sexp) )
78                             (types (second sexp))
79                             (parse (third sexp))))
80     ((equal? (first sexp) '+) (OP '+ (parse (cadr sexp))
81                                 (parse (caddr sexp))))
82     ((equal? (first sexp) '==)(OP '== (parse (cadr sexp))
```

```
83                                (parse (caddr sexp))))
84      ((equal? (first sexp) '<) (OP '< (parse (cadr sexp))
85                                (parse (caddr sexp))))
86      ((equal? (first sexp) '>) (OP '> (parse (cadr sexp))  (
            parse (caddr sexp))))
87      ((equal? (first sexp) 'fix) (FIX (first (second sexp))
88                                       (parse (second (second
                                           sexp)))
89                                       (parse (third sexp))))
90      ((equal? (first sexp)'/) (OP '/ (parse (cadr sexp))
91                                       (parse (caddr sexp))))
92      ((equal? (first sexp) '-) (OP '- (parse (cadr sexp))
93                                       (parse (caddr sexp))))
94      ((equal? (first sexp)'*) (OP '* (parse (cadr sexp))
95                                       (parse (caddr sexp))))
96      (else  (APP (parse (first sexp)) (map parse (list (
            second sexp)))))))
97
98
99
100 #|Función que traduce tipos escritos en sintaxis concreta a
        sintaxis abstracta|#
101 (define (parseTipos sexp)
102   (cond
103     ((boolean? sexp) (bool sexp))
104     ((integer? sexp) (num sexp))
105     ((flonum? sexp) (flot sexp))
106     ((real? sexp) (double sexp))
107     ((equal?  sexp 'true) (parseTiposAuxiliar 'true_))
108     ((equal? sexp 'false) (parseTiposAuxiliar 'false_))
109     ((equal? (first sexp) 'null)  (lista (parseTiposAuxiliar
            (second sexp))))
110     ((and (not(equal? sexp 'cons)) (and (symbol? sexp)  (and
             (not(equal? sexp 'false_))))) (VAR sexp))
111     ((equal? (first sexp) 'fst) (parseTiposAuxiliar(third
            sexp)))
112     ((equal? (first sexp) 'list)  (LIST (identidadAbstracta
            (rest sexp))))
113     ((equal? (first sexp) 'rst) (parseTiposAuxiliar (third
            sexp)))
114     ((equal? (first sexp) 'isempty?)
115     (parseTiposAuxiliar (third sexp)))
116     ((equal? (first sexp) 'cons) (parseTiposAuxiliar (fourth
             sexp)))
117     ((equal? (first sexp) 'let)
118      (parseTiposAuxiliar (fourth sexp)))  ;(error "TIPOS␣DE␣
            ARGUMENTOS␣DADOS␣INCORRECTOS")))
119     ((equal? (first sexp) 'if) (parseTiposAuxiliar(fifth
            sexp)))
120     ((equal? (first sexp) 'lambda)
```

```
121      (LAMBDA (Arg (second sexp) )  (types (second sexp))
122                    (parse (third sexp))))
123     ((equal? (first sexp) '+) (parseTiposAuxiliar (fourth
            sexp)))
124     [(equal? (first sexp) 'fix) (cons
125                           (parseTiposAuxiliar (third (second
                                sexp)))
126                           (list (parseTiposAuxiliar (fourth
                                sexp))))]
127     ((equal? (first sexp)'/)(parseTiposAuxiliar (fourth sexp
            )))
128     ((equal? (first sexp) '-) (parseTiposAuxiliar (fourth
            sexp)))
129     ((equal? (first sexp)'*) (parseTiposAuxiliar (fourth
            sexp)))
130     (else (parseTiposAuxiliar(third sexp)))))
131
132
133
134 #|Función que ayuda a transformar los tipos obtenidos en
       sintaxis concreta a sintaxis abstracta|#
135 (define (parseTiposAuxiliar n)
136   (cond
137     [ (equal? n 'boolean) (booln)]
138     [ (and (list? n)  (equal? (first n) 'boolean))  (booln)]
139     [ (equal? n 'float) (float)]
140     [ (equal? n 'integer) (int)]
141     [ (equal? n 'double) (double)]
142     [ (equal? n 'Min) (Min)]
143     [ (equal? n 'Max) (Max)]
144     [ (equal? n 'true) (booln)]
145     [ (equal? n 'false) (booln)]
146     [ (equal? n 'null) 'list_]
147     [ (and (> (length n) 1)
148           (and (list? n) (equal? (second n) '->)))
149                           (flecha (parseTiposAuxiliar (first
                                n))
150                           (parseTiposAuxiliar (third n)))]
151     [ (and (list? n)  (equal? (first n) 'List))
152                   (lista (parseTiposAuxiliar (second n)))]
153     [ (equal? (first n) 'lambda) 'fun]
154     [ (and (list? n)  (member (first n) '(+ - * /)))   (
            parseTiposAuxiliar (fourth n))]
155     [ (and (> (length n) 1)  (nand (equal? (first n) 'float)
             (equal? (second n) '->)))  (float)]
156     [ (and (list? n)  (equal? (first n) 'integer))  (int)]
157     [ (and (list? n)  (equal? (first n) 'float))  (float)]
158     [ (and (list? n)  (equal? (first n) 'double))  (double)]
159     [ (and (list? n)  (equal? (first n) 'Max))  (Max)]
160     [ (and (list? n)  (equal? (first n) 'Min))  (Min)]
```

```
161     [ (and (list? n)  (equal? (first n) 'boolean))  (booln)]
162     [else (error "Tipo␣no␣definido␣en␣MinTyRacket")] ))
163
164
165 #|Función que abstrae una expresion escrita en sintaxis
        concreta a sintaxis abstracta|#
166 (define (abs expr)
167   (parse expr))
```

B.3. VerificadorTiposMTR.rkt

```
1  #lang plai
2
3
4
5
6  (require "DefinicionesMTR.rkt")
7  (require "ParserMTR.rkt")
8  (require "EvaluadorMTR.rkt")
9
10
11
12
13 #|Procedimiento que extiende el ambiente para el verificador
       de tipos|#
14 (define (extender_env params argu_list foo_env arg_env)
15   (cond
16     ((and (empty? params)
17           (empty? argu_list))
18      foo_env)
19     ((or (and (empty? params)
20               (not (empty? argu_list)))
21          (and (not (empty? params))
22               (empty? argu_list)))
23      (error "Lista␣de␣parámetros␣y␣lista␣de␣argumentos␣no␣
            son␣de␣la␣misma␣longitud"))
24     (else
25      (extender_env (cdr params)
26                    (cdr argu_list)
27                    (anEnvTipos (car params)
28                                (parseTiposAuxiliar (car
                                       argu_list))
29                                foo_env)
30                    arg_env))))
31
32
33
34
```

```
35 #|Procedimiento que implementa el algoritmo de tipificación
       |#
36 (define (verificador expr ds)
37
38   (type-case MinTyRacket expr
39     [ERROR () (Min)]
40     [num (n) (int)]
41     [flot (n) (float)]
42     [doub (n) (double)]
43     [bool (n) (booln)]
44     [LIST (ls)
45           (cond
46             [((listof num?) ls) (lista (int))]
47             [((listof flot?) ls) (lista (float))]
48             [((listof doub?) ls) (lista (double))]
49             [((listof bool?) ls) (lista (booln))]
50            )]
51     [ISEMPTY? (ls)
52              (let*[
53                    (tipoT1 (verificador  ls  ds))
54                    ]
55                (type-case tipo tipoT1
56                  [lista (tipoLista) (booln)]
57                  [else "Tipo␣lista␣esperado"]))]
58     [CONST(elem ls)
59           (let*[
60                 (tipoT1 (verificador  elem  ds))
61                 (tipoT2 (verificador  ls  ds))
62                 ]
63             (
64              if (equal? (lista-elementos tipoT2)  tipoT1)
65                 tipoT2
66                 (error "La␣lista␣recibida␣como␣parámetro␣no␣
                       corresponde␣a␣las␣reglas␣de␣tipificación"
                       )
67                 ))]
68     [FST(ls)
69         (let*[
70               (tipoT1 (verificador  ls  ds))
71              ]
72           (lista-elementos tipoT1)
73           )]
74     [RST (ls)
75          (let*[
76                (tipoT1 (verificador  ls  ds))
77                ]
78           (lista (lista-elementos tipoT1) )
79           )]
80     [true_ () (booln)]
81     [NULL (tipo) (lista tipo)]
```

```
82      [false_ () (booln)]
83      [OP (op IZQ DER)
84             (cond
85               [(equal? '== op) (booln)]
86               [(equal? '> op) (booln)]
87               [
88                (let*
89                    ([tipo_OP_IZQ (verificador IZQ ds)]
90                     [tipo_OP_DER (verificador DER ds)])
91                  (cond
92                    [(equal? (ERROR) tipo_OP_DER) (Min)]
93                    [(equal? (booln) tipo_OP_IZQ)
94                     (error "Error␣de␣tipos,␣se␣esperaba␣un␣nú
                           mero,␣se␣ha␣dado␣una␣expresión␣de␣tipo
                           ␣boolean")]
95                    [(equal? (booln) tipo_OP_DER)
96                     (error "Error␣de␣tipos,␣se␣esperaba␣un␣nú
                           mero,␣se␣ha␣dado␣una␣expresión␣de␣tipo
                           ␣boolean")]
97                    [else(JOIN tipo_OP_IZQ tipo_OP_DER)])
98                  )])]
99      [VAR (v) (lookupTipo v ds)]
100     [IF (cond true else)
101          (if (booln? (verificador cond ds))
102              (let*
103                  ([tipo_rama_true (verificador true ds)]
104                   [tipo_rama_else (verificador else ds)]
105                   )
106                (JOIN tipo_rama_true tipo_rama_else))
107              (error "La␣condición␣no␣es␣de␣tipo␣boolean:␣␣"
108                  (eval cond ds)))]
109     [LAMBDA (paramlist types  body)
110               (let* ([env_extendido
111                       (extender_env paramlist types ds ds)]
112                      [tipo_body (verificador body
                           env_extendido)])
113                   (flecha (parseTiposAuxiliar (car types))
                        tipo_body))]
114     [FIX (varlig named cuerpolig)
115       (type-case MinTyRacket named
116         [LAMBDA (paramlist types body)
117               (let* ([env_extendido
118               (extender_env paramlist types ds ds)] [
                    tipo_body (verificador body env_extendido)
                    ])
119               (flecha (parseTiposAuxiliar (car types))
                    tipo_body))]
120          [else (error "La␣función␣no␣pertenece␣al␣lenguaje␣
                MinTyRacket")]
121          )]
```

```
122     [APP (f_expr arglist)
123          (if (VAR? f_expr) (Min)
124
125              (let*
126                  ([tipoT1 (verificador  f_expr  ds)]
127                   [tipoT2 (verificador  (car arglist)  ds)]
128                   )
129                (cond
130                  [(Min? tipoT1) (Min)]
131                  [else (type-case tipo tipoT1
132                       [flecha (antec consec)
133                        (if (esSubtipo tipoT2
134                        (flecha-antec tipoT1))
135                        (flecha-consec tipoT1)
136                        (error "Parámetros␣de␣tipos␣
                               inconsistentes"))]
137                        [else (error "Tipo␣de␣función␣
                               esperado")]
138                          )]))]))
139
140
141
142
143
144 #|Definición del ambiente de tipos|#
145 (define-type EnvTipos
146   [mtEnvTipos]
147   [anEnvTipos (name symbol?) (value tipo_elem?) (env
          EnvTipos?)]
148   [aRecSubTipos (name symbol?)
149                 (value isInMinTyRacket_Valores?)
150                 (env EnvTipos?)])
151
152
153
154
155 #|Procedimiento que busca el tipo recibido en un ambiente de
       tipos|#
156 (define (lookupTipo nombre ds)
157   (type-case EnvTipos ds
158     (mtEnvTipos () (error "no␣existe␣ligadura␣para␣el␣
           identificador:␣" nombre))
159     (anEnvTipos (nombre_lig valor_lig env)
160                 (if (equal? nombre_lig nombre)
161                     valor_lig
162                     (lookupTipo nombre env)))
163     [aRecSubTipos (nombrelig valorlig soc)
164                   (if (symbol=? nombrelig nombre)
165
166                       (unbox valorlig)
```

```
167                         (lookupTipo nombre soc))]))
168
169
170
171
172 #|Procedimiento que implementa las reglas de subtipificación
        |#
173 (define (esSubtipo sigma tau)
174   (cond
175     [(Min? sigma) true]
176     [(Max? tau) true]
177     [(and  (booln? sigma)    (booln? tau) ) true]
178     [(and  (int? sigma)    (int? tau)     ) true]
179     [(and  (float? sigma)    (float? tau)  ) true]
180     [(and  (float? sigma)    (double? tau)  ) true]
181     [(and  (int? sigma)    (double? tau)   ) true]
182     [(and  (double? sigma)    (double? tau)  ) true]
183     [(and  (int? sigma)    (float? tau)  ) true]
184     [(and  (flecha? sigma)    (flecha? tau)  )
185      (and (esSubtipo(flecha-antec tau) (flecha-antec sigma))
           (esSubtipo(flecha-consec sigma) (flecha-consec tau)
          ))]
186     [(and  (lista? sigma)    (lista? tau) )
187      (esSubtipo(lista-elementos sigma) (lista-elementos tau)
          ) ]
188     [else false]))
189
190
191
192 #|Definición de la lista compuesta por todos los tipos en
        MinTyRacket|#
193 (define listaTypes (list (int) (Min) (Max) (booln) (double)
        (float)))
194
195
196
197
198 #|Definición de la lista que almacena los supertipos comunes
         de dos tipos|#
199 (define supertiposComunes '())
200
201
202
203
204 #|Procedimiento que obtiene los supertipos comunes de dos
        tipos|#
205 (define (obtenerSupertiposComunes tipo_1 tipo_2 listaTipos
        supertiposComunes)
206   (if (null? listaTipos)  supertiposComunes
207       (cond
```

```
208         [(and (esSubtipo tipo_1 (car listaTipos))
209               (esSubtipo tipo_2 (car listaTipos)))
210                (obtenerSupertiposComunes tipo_1 tipo_2
211                (cdr listaTipos) (cons (car listaTipos)
                                    supertiposComunes))]
212         [else (obtenerSupertiposComunes tipo_1 tipo_2
213               (cdr listaTipos) supertiposComunes)])))
214
215
216
217 (define listaTypes2 (list (int) (Min) (Max) (booln) (double)
        (float)))
218
219
220
221
222 #|Definición de la lista donde se almacenan todas las
        combinaciones de tipos definidos en MinTyRacket|#
223 (define combinacionTipos '())
224
225
226
227
228 #|Procedimiento que regresa todas las combinaciones de tipos
        definidos en MinTyRacket|#
229 (define (obtenerCombinacionesFunc listaTipos1 listaTipos2
        combinacionTipos)
230   (if (null? listaTipos1)
231       combinacionTipos
232       (obtenerCombinacionesFunc
233        (cdr listaTipos1) listaTipos2
234        (cons (flecha (car listaTipos1)(car listaTipos2))
              combinacionTipos))))
235
236
237 (define combinacionTipos2 '())
238 (define (obtenerTotalComb lista combinacionTipos)
239   (if (null? lista)
240       combinacionTipos
241       (obtenerTotalComb (cdr lista) (append (
            obtenerCombinacionesFunc listaTypes  lista
            combinacionTipos2) combinacionTipos))))
242
243
244
245 #|Procedimiento que obtiene las combinaciones de tipos
        posibles|#
246 (define combinacionTiposLista '())
247 (define (obtenerCombinacionesLista listaTipos1
        combinacionTiposLista)
```

```
248   (if (null? listaTipos1)
249       combinacionTiposLista
250       (obtenerCombinacionesLista (cdr listaTipos1) (cons(
              lista (car listaTipos1)) combinacionTiposLista))))
251
252
253
254 #|Procedimiento que almacena todas las combinaciones de
        tipos posibles|#
255 (define TOTALTIPOS (append (append (append (list (int) (Min)
        (Max) (booln) (double) (float))
256       (obtenerCombinacionesLista listaTypes2
              combinacionTiposLista))
257             (obtenerTotalComb listaTypes2 combinacionTipos)
                  )))
258
259
260
261
262 #|Procedimiento que calcula el mínimo supertipo común de dos
        tipos|#
263 (define (JOIN v_1 v_2)
264   (let([listaSupertiposTotal (obtenerSupertiposComunes v_1
          v_2 TOTALTIPOS supertiposComunes)])
265     (elemMin listaSupertiposTotal(car  listaSupertiposTotal)
          )
266     ))
267
268
269
270
271 #|Procedimiento que calcula el tipo que es subtipo de los
        demás tipos en una lista|#
272 (define elemMin
273   (lambda (ls min)
274
275     (if (null? ls) min
276         (elemMin (cdr ls) (minimo (car ls) min)))))
277
278
279
280
281 #|Procedimiento que regresa el subtipo entre dos tipos|#
282 (define minimo
283   (lambda(ls1 ls2)
284     (if (esSubtipo  ls1  ls2)
285         ls1
286         ls2 )))
287
288
```

```
289
290
291 #|Procedimiento que aplica el algoritmo de tipificación dada
        una expresión escrita en sintaxis abstracta|#
292 (define (VERIFICADOR exp envTipo)
293   (cond
294     [(and (FIX? (abs exp))   (esSubtipo (verificador (abs
          exp) envTipo) (car (parseTipos exp))))
295      (if
296       (esSubtipo (flecha-consec  (verificador (abs exp)
           envTipo)) (second (parseTipos exp)))
297       true
298       false)]
299     [else
300      (if
301       (esSubtipo (verificador (abs exp) envTipo) (parseTipos
            exp))
302       true
303       false
304       )]))
```

B.4. EvaluadorMTR.rkt

```
1  #lang plai
2
3
4
5
6  (require "DefinicionesMTR.rkt")
7
8
9
10
11 #|Procedimiento que regresa los valores respectivos a cada
       elemento de una lista|#
12 (define identidadV
13   (lambda (lista ds)
14     (if (= (length lista) 0)
15         null
16         (append  (identidadV (remove (last lista) lista) ds)
              (list (eval(last lista) ds))))))
17
18
19
20
21 #|Procedimiento que implementa el algoritmo de evaluación|#
22 (define (eval expr ds)
23   (type-case MinTyRacket expr
```

```
24      [CONST (elem ls) (cons (eval elem ds) (eval ls ds))]
25      [ISEMPTY? (ls)
26                (cond [(and (listV? (eval ls ds))
27                            (equal? empty (listV-ls (eval ls
                                  ds)))) true]
28                      [else (empty? (eval ls ds))])]
29      [ERROR () (ERROR)]
30      (FST(LS)
31          (car (eval  LS  ds)))
32      [LIST (ls)  (identidadV ls ds) ]
33      (RST (LS)
34           (cdr (eval  LS  ds)))
35      [true_ () (booleanV true_)]
36      [NULL (tipo) (listV empty)]
37      [false_ () (booleanV false_)]
38      [bool (n)
39            (if  (equal? n #t)  (booleanV true)  (booleanV
                  false)  )]
40      [num (n)
41           (integerV n)]
42      [doub (n)
43            (doubleV n)]
44      [flot (n)
45            (floatV n)]
46      [OP (op IZQ DER)
47             ((cadr (assoc op opbinarias))
48              (eval IZQ ds)
49              (eval DER ds))]
50      [VAR (v) (lookup v ds) ]
51      [IF (cond true else)
52           (if (booleanV? (eval cond ds))
53               (if (equal? (booleanV-n (eval cond ds)) #t)
54                   (eval true ds)
55                   (eval else ds))
56               (error "condicional␣no␣es␣de␣tipo␣booleano:␣␣")
57               (eval cond ds)))]
58      [LAMBDA (paramlist types  body)
59            (closureV paramlist body ds)]
60      [FIX (varlig named cuerpolig)
61        (eval cuerpolig
62                 (ciclico varlig named ds))]
63      (APP (f_expr arglist)
64           (if (closureV? (eval f_expr ds))
65               (eval (closureV-body (eval f_expr ds))
66                       (args_env (closureV-params (eval f_expr
                                ds))
67                                 arglist
68                                 (closureV-env (eval f_expr ds
                                      ))
69                                 ds))
```

```
70              (error "Una␣aplicación␣requiere␣ser␣aplicada␣en
                    ␣una␣función:␣" (eval f_expr ds))
71              ))))
72
73
74
75
76  #|Procedimiento que dada una lista de argumentos, una lista
        de valores, y ambientes, comprueba que cada argumento
        tenga un valor asociado, en caso que así sea extiende el
        ambiente recibido con los nuevos argumentos y valores
        recibidos|#
77  (define (args_env params argu_list foo_env arg_env)
78    (cond
79      ((and (empty? params)
80            (empty? argu_list))
81       foo_env)
82      ((or (and (empty? params)
83                (not (empty? argu_list)))
84           (and (not (empty? params))
85                (empty? argu_list)))
86       (error "lista␣de␣parámetros␣y␣lista␣de␣argumentos␣no␣
              son␣de␣la␣misma␣longitud"))
87      (else
88       (args_env (cdr params)
89                 (cdr argu_list)
90                 (anEnv (car params)
91                        (eval (car argu_list) arg_env)
92                        foo_env)
93                 arg_env))))
94
95
96
97
98  #|Procedimiento que busca el identificador en un ambiente de
        valores
99  y regresa el valor ligado en caso de que exista|#
100 (define (lookup nombre ds)
101   (type-case Env ds
102     (mtEnv () (error "no␣existe␣ligadura␣para␣el␣
            identificador:␣" nombre))
103     (anEnv (nombre_lig valor_lig env)
104            (if (equal? nombre_lig nombre)
105                valor_lig
106                (lookup nombre env)))
107     [aRecSub (nombrelig valorlig soc)
108              (if (symbol=? nombrelig nombre)
109
110                  (unbox valorlig)
111                  (lookup nombre soc))]))
```

```
112
113
114
115
116 #|Implementación de los ambientes cíclicos procedurales|#
117 (define (ciclico liga expr env)
118   (local([define valor (box (numV 2830))]
119          [define nuevo_env (aRecSub liga valor env)]
120          [define expr_valor (eval expr nuevo_env)])
121     (begin
122       (set-box! valor expr_valor)
123       nuevo_env)))
```

B.5. InterpMTR.rkt

```
1    #lang plai
2
3
4
5
6    (require "DefinicionesMTR.rkt")
7    (require "ParserMTR.rkt")
8    (require "VerificadorTiposMTR.rkt")
9    (require "EvaluadorMTR.rkt")
10
11
12
13
14     #|Intérprete de MinTyRacket|#
15     (define (interpMTR expr)
16       (if (equal? true (VERIFICADOR expr (mtEnvTipos)))
17        (getValorMTR (eval (parse expr) (mtEnv)) (parseTipos
             expr))
18        (error "Error␣de␣tipo:␣El␣tipo␣declarado␣no␣
             corresponde␣a␣las␣reglas␣de␣tipificación␣de␣
             MinTyRacket")
19       ))
```

Bibliografía

[1] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G., *Dynamic Typing in a Statically Typed Language*, paper of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, January 2011.

[2] Bertrand, M., *Introduction to the Theory of Programming Languages*, Prentice Hall. 1990.

[3] Cardelli, L., *Type System*, Microsoft research, chapter for te CRC Handbook of Computer Science and Enggineering, second edition. 2002.

[4] Felleisen, M., Barski, C., *Realm of Racket*, No Starch Press. 2013.

[5] Hanenberg, S., Kleinschmager, Robbes R., Tanter, E. , Steflk, *An empirical study on the impact of static typing on software maintainability*, Springer Science and Bussiness Media New York, 2013.

[6] Haskell página oficial, `http://www.haskell.org/haskellwiki/Haskell` 13 de Octubre del 2013, 20:49 hrs.

[7] Hudak, P., Hughes, J., Jones, S.P., Wadler, P., *A history of Haskell: Being lazy with class*, Proceedings - Third ACM SIGPLAN History of Programming Languages Conference, HOPL-III , pp. 12-1-12-55. 2007.

[8] Kozen, D., *Automata and Computability*, Springer New York. 1997.

[9] Krishnamurthi, S., Tucker, D., *Notas del curso de Lenguajes de la Universidad de Brown*, Universidad de Brown. 2007.

[10] Krishnamurthi, S., *Programming Languages: Application and Interpretation*, Universidad de Brown. 2013.

[11] Läuferand, K., Thiruvathukal, G., *The Promises of Typed, Pure and Lazy Lazy Functional Programming*, Chicago University. 2009.

[12] Meijer, E., Drayton, P., *Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages, 2004.

[13] Mitchell, J.C., *Concepts in Programming Languages*, UK: Cambridge University Press, 2003.

[14] Ortin, F., *Type Inference to Optimize a Hybrid Statically and Dynamically Typed Language*, Oxford University Press. 2011.

[15] Pierce, B., *Types and Programming Languages*, MIT Press. 2002.

[16] Racket página oficial, `http://racket-lang.org/` 10 de enero del 2014, 16:20 hrs.

[17] Spivak, D.I., *Category Theory for Scientist*, MIT, 2003.

[18] Springer, G., Friedman, D.P., *Scheme and the Art of Programming*, MIT Press and McGraw-Hill, 1989.

[19] Tucker, A., Noonan R., *Programming Languages, Principles and Paradigms*, McGraw-Hill Higher Education, 2002.