
Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Facultad de Ciencias

# *Seguridad de sistemas de tipos vía Verificación de Modelos*

TESIS

QUE PARA OPTAR POR EL GRADO DE:

Maestro en Ciencia e Ingeniería de la Computación

Presenta:

Manuel Soto Romero

Tutores:

Dr. Favio Ezequiel Miranda Perea
Facultad de Ciencias, UNAM

Dr. David Arturo Rosenblueth Laguette
Instituto de Investigaciones en Matemáticas Aplicadas y de Sistemas (IIMAS)

Ciudad Universitaria, Cd. Mx. agosto, 2023

El presente trabajo fue desarrollado como parte de las actividades y productos del proyecto

**“Lógicas no clásicas: Aspectos deductivos de la computación a la filosofía” (PAPIIT IN119920 Octubre 2022 a Marzo de 2023 / PAPIIT IN101723 Abril a Junio de 2023)**

En el Programa de Proyectos de Apoyo a Proyectos de Investigación e Innovación Tecnológica de la Dirección General de Asuntos del Personal Académico de la Universidad Nacional Autónoma de México

El presente trabajo fue desarrollado bajo el apoyo de una beca otorgada por el

**Consejo Nacional de Ciencia y Tecnología (CONACyT)**

Durante el periodo de 2020 a 2022

# Índice general

# Introducción

Los dispositivos de cómputo hoy en día son herramientas que nos acompañan en todo momento y de las cuales dependemos en gran medida. Por mencionar algunas tareas que se llevan a cabo mediante tales dispositivos, tenemos:

- Manejo de cuentas bancarias
- Uso de redes sociales
- Compras en Internet
- Diversos trámites
- Sistemas de pilotaje automático
- Programación de nuevos sistemas
- Implementación de compiladores e intérpretes para lenguajes de programación

Debido a la variedad de problemas que podemos resolver por medio de dichos dispositivos, resulta importante reducir el riesgo de que fallen. Por ejemplo, no quisiéramos que un sistema de compras en línea nos cobrara por un producto que nunca llega, que el sistema de pilotaje automático de un avión fallara en pleno vuelo o que un compilador aceptara programas que contienen errores de sintaxis. Este tipo de características van de la mano con la llamada *calidad* del software.

***Definición 0.1***

> *La calidad de un sistema de cómputo puede definirse como el grado en que éste satisface las necesidades y expectativas para las cuales fue diseñado cuando se usa en las condiciones especificadas.* [10]

Las principales causas de la mala calidad del software son quizá los *defectos* que podemos encontrar en el mismo.

***Definición 0.2***

*Un defecto es el resultado de un error cometido por un desarrollador al generar un producto.* [7]

Los defectos, por más pequeños que sean, como faltas de ortografía o de dedo, pueden ocasionar problemas severos en el software al presentar inconsistencias o respuestas impredecibles. Adicionalmente, el aumento en el volumen de datos que manejan los sistemas hoy en día hace que la calidad del software sea una tarea esencial en cualquier desarrollo. Al no cuidar la calidad se pueden ocasionar pérdidas monetarias, o incluso muertes.

Algunos casos donde el software de mala calidad ha resultado catastrófico son:

- La misión *Fobos 1* tenía por objetivo estudiar Fobos, una de las dos lunas de Marte; sin embargo, poco después de su lanzamiento se perdió el contacto con la sonda debido a que el software de la misma falló. Hoy en día sigue orbitando alrededor del sol como basura espacial. [22]
- La *Mariner 1* fue una misión con el objetivo de sobrevolar, por medio de una sonda, el planeta Venus. La NASA tuvo que activar su sistema de autodestrucción pues un defecto en su software ocasionó un desvío de la sonda hacia el Océano Atlántico. [21]
- El primer vuelo de prueba del cohete Ariana 5, el 4 de junio de 1996, falló. El cohete se destruyó a sí mismo 37 segundos después del despegue por un error de tipos en el software. Una conversión de un valor de punto flotante de 64 bits a un entero de 16 bits causó un error de operando. Ada era el lenguaje en el que estaba escrito. [6]
- *Therac-25* fue una máquina de radioterapia. Estuvo involucrada en al menos seis accidentes entre 1985 y 1987 en los que varios pacientes recibieron sobredosis de radiación, derivando en que fallecieran. Después de realizar una investigación se concluyó que hubo malas prácticas en el análisis de requerimientos y por ende un mal diseño del software. [23]

Las prácticas que usualmente se siguen para comprobar la calidad del software son principalmente la *verificación* y la *validación*:

- *Verificar* un producto de software tiene por objetivo determinar si la implementación del mismo satisface la especificación siempre.
- *Validar* el software consiste en asegurarse de que la implementación hace lo que el usuario espera que haga en casos lo suficientemente representativos.

La validación es una tarea que puede asegurarse con herramientas de *testing* entre las cuales destacan el uso de pruebas unitarias y su automatización, mismas que los departamentos de calidad en distintas empresas de desarrollo de software usan así como el apoyo de técnicas de análisis de requerimientos como el diseño de historias de usuario o casos de uso. La verificación es más complicada.

A lo largo de los años, se ha trabajado en el desarrollo de técnicas y mecanismos que facilitan la verificación del software tales como el desarrollo de marcos de trabajo (*frameworks*), elaboración de patrones de diseño, estilos de programación, lenguajes de propósito específico así como investigación y uso de herramientas matemáticas que permiten especificar y razonar sobre propiedades que debe cumplir el software y que permitan verificar las mismas, tales como demostradores automáticos o asistentes de prueba.

En esta tesis se da la implementación de un verificador de modelos para probar la seguridad de sistemas de tipos de lenguajes de programación por medio de la técnica *Glassbox Model Checking* presentada en [15]. Dicha técnica se enfoca en resolver el problema de la *explosión de estados* mediante reducciones al problema SAT así como el concepto de *poda* sobre espacios de búsqueda que permite reducir los mismos significativamente. La implementación del sistema se da mediante el lenguaje de programación funcional HASKELL.

El trabajo se divide en los siguientes capítulos:

**Capítulo 1 : *Preliminares*** Se da una breve definición y un panorama de los elementos que forman parte del sistema y que son necesarios para lograr la implementación, tales como: verificación de modelos, especificación de un lenguaje de programación mediante su semántica estática y dinámica, seguridad de un sistema de tipos y sobre todo una explicación de la metodología *Glassbox Model Checking*.

**Capítulo 2 : *Implementación puramente funcional*** Se da una breve descripción sobre los módulos que conforman la implementación que en general, pueden resumirse en: arquitectura del sistema, especificación del lenguaje, generación de espacios de búsqueda, poda e integración.

**Capítulo 3: *Casos de estudio*** Se muestra el funcionamiento del sistema con algunos lenguajes de programación de prueba así como sus resultados. Algunos de los lenguajes de programación con los que se realizan las pruebas incluyen errores de diseño intencionales con el fin de ilustrar que el sistema implementado en este trabajo es capaz de reportar lenguajes con sistemas de tipos seguros e inseguros.

**Capítulo 4: *Conclusiones y trabajo futuro*** Se concluyen los resultados finales y se describe algún trabajo a futuro por realizar como resultado de la implementación del sistema de este trabajo entre los que destaca el reconocimiento de lenguajes con sistemas de tipos avanzados que incluyan por ejemplo variables de tipo.

# Capítulo 1

# Preliminares

En este capítulo se presentan tanto conocimientos previos como conceptos sobre la especificación formal de lenguajes de programación, la verificación de modelos y la metodología Glass box que se emplean en el desarrollo de esta tesis. Este material fue tomado principalmente de las referencias [16], [2], [5] y [13].

## 1.1. Especificación formal de un lenguaje de programación

Para definir un lenguaje de programación, usualmente se consideran dos aspectos básicos llamados *sintaxis* y *semántica* [13] que pueden ser formalizados

- para entender o mostrar la correctud de los programas escritos en dicho lenguaje de programación,
- para mostrar la correctud de transformaciones entre programas o
- para poder implementar y verificar compiladores o traductores.

### 1.1.1. Sintaxis

La sintaxis de un lenguaje de programación define una descripción del conjunto de cadenas de símbolos que serán consideradas programas. Para definir la sintaxis se suelen usar dos clases de objetos:

- Las cadenas para definir la sintaxis concreta que representa al programa dado por aquellas personas que vayan a usar nuestro lenguaje de programación.
- Los *árboles de sintaxis abstracta* para modelar la estructura jerárquica de la sintaxis y que la computadora pueda manejar de forma sencilla el programa escrito por los desarrolladores.

En general se tienen dos niveles de sintaxis que se relacionan con su nivel de abstracción y cómo se utilizan en el contexto de los lenguajes de programación.

**Sintaxis concreta**

La sintaxis concreta de un lenguaje de programación se determina usualmente por medio de dos partes:

***Definición 1.1***

*La sintaxis léxica describe la construcción de lexemas (átomos, tokens, símbolos terminales). [1]*

Las palabras reservadas, identificadores de variables, numerales, literales o espacios son ejemplos de la sintaxis léxica. La principal herramienta para la descripción de este tipo de sintaxis son las expresiones regulares.

***Definición 1.2***

*La sintaxis libre de contexto describe la construcción de frases del lenguaje. [1]*

Las expresiones aritméticas y booleanas, estructuras de control, definición de funciones o declaración de variables son ejemplos de la sintaxis libre de contexto pues se pueden representar de manera jerárquica. La principal herramienta para la descripción de este tipo de sintaxis son las gramáticas libres de contexto representadas por lo general en la forma normal extendida de Backus-Naur (EBNF[1]).

***Ejemplo 1.1***

A continuación se presenta la sintaxis concreta para un lenguaje con expresiones aritméticas y booleanas llamado EAB.

```
<expr> ::= 0
         | true
         | false
         | suc(<expr>)
         | pred(<expr>)
         | iszero(<expr>)
         | if <expr> then <expr> else <expr>
```

La gramática dada usa notación EBNF y representa la sintaxis libre de contexto del lenguaje, mientras que

[1]Esta notación es una extensión de la notación BNF que agrega algunas características adicionales para hacer las reglas de producción más concisas y legibles.

los símbolos terminales contenidos en dicha gramática representan la sintaxis léxica.

Algunos ejemplos de expresiones son:

- `suc(0)`
- `suc(pred(suc(0))`
- `suc(false)`

La última expresión nos permite observar que esta gramática nos deja construir expresiones sintácticamente correctas pero que no tienen sentido desde el punto de vista semántico. En general, es común detectar errores de este tipo desde la semántica debido a que se enfoca en el significado y comportamiento de las expresiones, mientras que la sintaxis sólo se encarga de la forma y estructura de las mismas y aunque también juega un papel importante en la detección temprana de errores es más limitada en términos de verificar consistencia y coherencia de tipos a diferencia de la semántica. El nivel de semántica que usaremos para ello es la semántica estática; se habla de esto en breve.

**Sintaxis abstracta**

En principio podríamos procesar directamente las cadenas descritas por la sintaxis concreta para dar un significado a los programas o realizar algún tipo de transformación; sin embargo, es una tarea que puede volverse tediosa a la larga. Para simplificar este tipo de tareas, se usa la sintaxis abstracta, que proporciona una representación usualmente dada por una estructura jerárquica conocida como *árbol de sintaxis abstracta (asa)*. El asa de una expresión captura el orden en que se realizarán las operaciones por medio de una notación prefija. Además es único para cualquier expresión, sin importar su representación concreta.

Un asa es básicamente un árbol cuyos nodos están etiquetados con una etiqueta que indica la operación a realizar. Cada operador tiene un índice asignado que representa el número de argumentos que recibe, y que corresponde con el número de hijos de cualquier nodo etiquetado con él. Para especificar los asa se suelen usar juicios[2] y reglas que definen inductivamente a los árboles; sin embargo, puede ser de gran utilidad dibujar los mismos.

***Ejemplo 1.2***

A continuación se presenta la sintaxis abstracta del lenguaje EAB por medio de juicios y reglas.

[2] Un juicio es una afirmación o enunciado que expresa una proposición sobre la realidad. Los juicios son fundamentales en el razonamiento lógico y se expresan mediante proposiciones, que pueden ser verdaderas o falsas.

$$\frac{}{Zero\,()\ \text{asa}} \qquad \frac{n\ \text{asa}}{Pred\,(n)\ \text{asa}}$$

$$\frac{}{False\,()\ \text{asa}} \qquad \frac{n\ \text{asa}}{IsZero\,(n)\ \text{asa}}$$

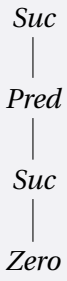
$$\frac{}{True\,()\ \text{asa}}$$

$$\frac{n\ \text{asa}}{Suc\,(n)\ \text{asa}} \qquad \frac{c\ \text{asa} \quad t\ \text{asa} \quad e\ \text{asa}}{If\,(c\,,t\,,e)\ \text{asa}}$$

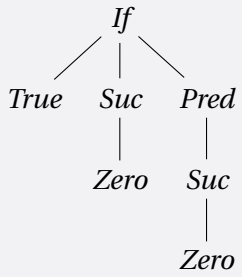
Árboles de sintaxis abstracta de algunas expresiones:

- Sintaxis concreta: `suc(0)`
  Sintaxis abstracta: *Suc(Zero())*
  Representación gráfica:


- Sintaxis concreta: `suc(pred(suc(0))`
  Sintaxis abstracta: *Suc(Pred(Suc(Zero()))*
  Representación gráfica:




- Sintaxis concreta: `if true then suc(0) else pred(suc(0))`
  Sintaxis abstracta: *If(True(), Suc(Zero()), Pred(Suc(0))*
  Representación gráfica:




Una tarea principal de todo traductor de lenguaje es transformar un programa dado en sintaxis concreta a sintaxis abstracta mediante los llamados analizadores léxicos (*lexers)* y los analizadores sintácticos (*parsers*). La especificación de estos programas queda fuera de este trabajo pues únicamente se hace uso de la sintaxis abstracta. Se habla a profundidad de esto más adelante.

En general podemos decir que la sintaxis concreta se refiere a la representación textual o gráfica específica que sigue las reglas y convenciones de un lenguaje de programación, mientras que la sintaxis abstracta se enfoca en la estructura lógica y la semántica de un programa desde un nivel más alto de abstracción. Ambos niveles de sintaxis son importantes en la teoría de lenguajes de programación y en la implementación de compiladores e intérpretes para traducir programas escritos en sintaxis concreta a sintaxis abstracta para su análisis y ejecución.

### 1.1.2. Semántica

La semántica de un lenguaje de programación define el significado de las instrucciones y expresiones del lenguaje. Puede ser descrita de manera informal, por medio de lenguaje natural, en manuales técnicos o bien, de manera formal basada en técnicas matemáticas. En este trabajo usaremos descripciones formales.

Usualmente se consideran dos niveles de semántica.

**Semántica dinámica**

La semántica dinámica determina el valor o evaluación de un programa[3]. Existen tres estilos básicos para definir la semántica dinámica de un lenguaje de programación:

- La **semántica operacional** define el comportamiento de un programa en términos de sus operaciones indicando *cómo* evaluar las expresiones de un lenguaje.
- La **semántica denotativa** asocia las expresiones a objetos matemáticos como pueden ser números, conjuntos o funciones.
- La **semántica axiomática** es un enfoque basado en la lógica para probar que un programa es correcto estableciendo la lógica de condiciones antes y después de la evaluación de un programa.

[3]Consideraremos como programa a una secuencia de expresiones válidas en un lenguaje de programación. En ocasiones se usarán indistintamente los términos programa y expresión.

En este trabajo se opta por usar un estilo operacional para definir la semántica de los lenguajes. Este estilo puede darse siguiendo dos enfoques: de paso grande (natural) o de paso pequeño (estructural) y se define en términos de sistemas de transición que indican cómo transitar de una expresión a otra. Por ejemplo, consideremos la expresión `if(iszero(pred(suc(0))) then true else false`. Siguiendo un enfoque de paso pequeño tendríamos:

```
if iszero(pred(suc(0)) then true else false
→if iszero(0) then true else false
→if true then true else false
→true
```

mientras que usando paso grande sería:

```
if iszero(pred(suc(0)) then true else false
⇓true
```

En ambos casos transitamos de un programa a otro.

***Definición 1.3***

*Un sistema de transición es un modelo abstracto con los siguientes elementos:*

- *Un conjunto* $\Gamma$ *de estados.*
- *Una relación de transición.*
- *Un conjunto I de estados iniciales subconjunto de* $\Gamma$.
- *Un conjunto F de estados finales subconjunto de* $\Gamma$.

Podemos pensar en los sistemas de transición como mecanismos que nos permiten modelar la ejecución de los programas mediante un dispositivo de cómputo abstracto.

***Ejemplo 1.3***

A continuación se presenta un sistema de transición para EAB:

- **Conjunto de estados:** $S = \{a \mid a \text{ asa}\}$, es decir los estados del sistema son las expresiones bien formadas del lenguaje en sintaxis abstracta. Esta definición corresponde con la siguiente regla de inferencia:

$$\frac{a \text{ asa}}{a \text{ estado}}$$

- **Estados iniciales:** $I = \{a \mid a \text{ asa}\}$, en este caso $S = I$, pues podemos partir de cualquier expresión bien formada. Esta definición corresponde con la siguiente regla de inferencia:

$$\frac{a \text{ asa}}{a \text{ inicial}}$$

- **Estados finales:** Se definen como las expresiones que representan a los posibles resultados finales de un proceso de evaluación. Para poder modelarlos definimos una categoría de valores, que pueden ser un subconjunto de expresiones que ya se han terminado de evaluar y no pueden reducirse más, con el juicio $v$ valor. Para el caso de EAB los valores son el cero, las constantes lógicas y el sucesor de un valor. Definimos entonces nuestro conjunto de valores por las reglas:

$$\frac{}{Zero\,() \text{ valor}} \qquad \frac{}{True\,() \text{ valor}} \qquad \frac{}{False\,() \text{ valor}} \qquad \frac{n \text{ valor}}{Suc\,(n) \text{ valor}}$$

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

$$\frac{a \text{ valor}}{a \text{ final}}$$

- **Transiciones:** La definición de las transiciones se define de acuerdo con el enfoque de la semántica operacional (natural o estructural) a partir de los estados previamente definidos.

  La forma de leer las reglas siguientes es de abajo hacia arriba. Por ejemplo la regla de la primera fila de la columna izquierda se puede leer como "El sucesor de un número $n$ se reduce en un paso al sucesor de un número $n'$ si dicho número $n$ se reduce a $n'$.

  *Estructural:*

$$\frac{n \to n'}{Suc\,(n) \to Suc\,(n')} \qquad \frac{}{IsZero\,(Zero\,()) \to True\,()}$$

$$\frac{n \to n'}{Pred\,(n) \to Pred\,(n')} \qquad \frac{}{IsZero\,(Suc\,(n)) \to False\,()}$$

$$\frac{}{Pred\,(Suc\,(n)) \to n} \qquad \frac{c \to c'}{If(c\,,t\,,e) \to If(c'\,,t\,,e)}$$

$$\frac{}{Pred\,(Zero\,()) \to Zero\,()} \qquad \frac{}{If(True\,(),t\,,e) \to t}$$

$$\frac{n \to n'}{IsZero\,(n) \to IsZero\,(n')} \qquad \frac{}{If(False\,(),t\,,e) \to e}$$

  *Natural:*

$$\frac{n \Downarrow n'}{Suc(n) \Downarrow Suc(n')} \qquad \frac{n \Downarrow Suc(n')}{IsZero(n) \Downarrow False()}$$

$$\frac{n \Downarrow Zero()}{Pred(n) \Downarrow Zero()}$$

$$\frac{c \Downarrow True() \quad t \Downarrow t'}{If(c,t,e) \Downarrow t'}$$

$$\frac{n \Downarrow Suc(n') \quad n' \Downarrow n''}{Pred(n) \Downarrow n''}$$

$$\frac{n \Downarrow Zero()}{IsZero(n) \Downarrow True()} \qquad \frac{c \Downarrow False() \quad e \Downarrow e'}{If(c,t,e) \Downarrow e'}$$

Se observa que el estilo natural contiene menos reglas que el estilo estructural al mostrar el resultado final de evaluación en contraste a ir paso a paso evaluando.

Notar además que no hay variables.

### *Ejemplo 1.4*

A continuación se presenta la evaluación de la siguiente expresión usando ambos enfoques:

```
if iszero(pred(pred(suc(suc(0))))) then 0 else suc(0)
```

cuya sintaxis abstracta es:

*If(IsZero(Pred(Pred(Suc(Suc(Zero()))))),Zero(),Suc(Zero()))*

*Enfoque natural:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Zero() \Downarrow Zero()}}{Suc(Zero()) \Downarrow Suc(Zero())}}{Suc(Suc(Zero())) \Downarrow Suc(Suc(Zero()))}}{Pred(Suc(Suc(Zero()))) \Downarrow Zero()}}{Pred(Pred(Suc(Suc(Zero())))) \Downarrow Zero()}}{IsZero(Pred(Pred(Suc(Suc(Zero()))))) \Downarrow True()} \quad \dfrac{}{Zero() \Downarrow Zero()}}{If(IsZero(Pred(Pred(Suc(Suc(Zero()))))),Zero(),Suc(Zero())) \Downarrow Zero()}$$

*Enfoque estructural*

*If(IsZero(Pred(Pred(Suc(Suc(Zero()))))),Zero(),Suc(Zero()))*
*→ If(IsZero(Pred(Suc(Zero()))),Zero(),Suc(Zero()))*
*→ If(IsZero(Zero()),Zero(),Suc(Zero())*
*→ If(True(),Zero(),Suc(Zero())*
*→ Zero()*

La semántica estructural nos dice que el significado de una expresión $e$ es el estado final alcanzado por las reglas de evaluación tomando a $e$ como estado inicial. En contraste, la semántica natural evalúa la expresión devolviendo el resultado final en un solo paso. En este trabajo se usa únicamente el estilo estructural; se habla de esto más adelante.

**Semántica estática**

La semántica estática determina si un programa es correcto mediante criterios sintácticos sensibles al contexto. Incluye todas las propiedades de un programa que pueden verificarse en tiempo de compilación. La definición de estas propiedades, así como qué tanta información obtiene el compilador dependen del lenguaje de programación.

En general este nivel semántico se relaciona con propiedades sobre las reglas de tipificado o alcance de un lenguaje de programación. Para algunos lenguajes, el tipo de una expresión o el alcance de una variable sólo se puede determinar hasta tiempo de ejecución, por lo que no pertenecerían a la semántica estática. Se utiliza también para definir restricciones de tipos sobre las expresiones del lenguaje.

Antes de definir el significado preciso de un programa mediante su semántica operacional es necesario eliminar los programas mal formados con respecto a su *sistema de tipos*.

***Definición 1.4***

*Formalmente un tipo puede definirse como una tupla $T = (V, O)$, donde:*

- *$V$ es el conjunto de valores que pertenece al tipo.*
- *$O$ es el conjunto de operaciones definidas sobre los valores de tipo.*

Como ejemplo de tipo, se tiene al tipo de dato `int` de algunos lenguajes de programación, cuyo conjunto de valores son los números enteros y conjunto de operaciones se tienen operaciones básicas como la suma, resta, división, multiplicación entre muchas otras.

Además de estos conjuntos, los tipos también incluyen restricciones y reglas para el uso correcto de los valores y operaciones dados por su *sistema de tipos*.

### *Definición 1.5*

*Un sistema de tipos es un conjunto de reglas que definen ciertas restricciones en la formación de los programas de un lenguaje. Consiste de:*

- *Un conjunto de tipos: Etiquetas que representan diferentes clases de valores en el lenguaje de programación.*
- *Reglas: Definen restricciones sobre la formación de programas, definen cómo los tipos se asignan a las expresiones y variables en el programa, establecen cómo se verifica la corrección de los tipos y en algunos casos definen cómo inferir automáticamente el tipo de una expresión en función del contexto en que se usa.*

Las frases del lenguaje se clasifican mediante tipos que dictan cómo pueden usarse. Intuitivamente el tipo de una expresión define la forma de su valor. Por ejemplo, la comparación de dos expresiones numéricas con el operador $<$ debe ser un valor booleano mientras que si intentamos comparar una expresión numérica con una booleana con el mismo operador se genera un error de tipos por no estar definido este comportamiento.

Para definir un sistema de tipos necesitamos un conjunto de tipos que se asocian a cada una de las expresiones del lenguaje, mediante una colección de reglas de tipificado. El uso de sistemas de tipos en el diseño de un lenguaje de programación tiene diferentes ventajas como:

- descubrir errores en expresiones tempranamente,
- ofrecer seguridad pues un programa correctamente tipificado no puede funcionar mal,
- documentar un programa de manera más simple y manejable que los comentarios.

### *Ejemplo 1.5*

A continuación se define la semántica estática del lenguaje EAB con el siguiente conjunto de reglas de tipificado para los árboles de sintaxis abstracta de las expresiones del lenguaje mediante un juicio binario entre expresiones $t$ y tipos $\mathsf{T}$ denotado:

$$t : \mathsf{T}$$

que se lee como *la expresión t tiene el tipo* $\mathsf{T}$.

$$\frac{}{False() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{Pred(n) : \mathsf{Nat}}$$

$$\frac{}{True() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{IsZero(n) : \mathsf{Bool}}$$

$$\frac{}{Zero() : \mathsf{Nat}}$$

$$\frac{n : \mathsf{Nat}}{Suc(n) : \mathsf{Nat}} \qquad \frac{c : \mathsf{Bool} \quad t : \tau \quad e : \tau}{If(c, t, e) : \tau}$$

### *Ejemplo 1.6*

A continuación se presenta la derivación del tipo de la siguiente expresión usando las reglas anteriores:

```
if iszero(pred(pred(suc(suc(0))))) then 0 else suc(0)
```

cuya sintaxis abstracta es:

*If(IsZero(Pred(Pred(Suc(Suc(Zero()))))),Zero(),Suc(Zero()))*

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Zero() : \mathsf{Nat}}}{Suc(Zero()) : \mathsf{Nat}}}{Suc(Suc(Zero())) : \mathsf{Nat}}}{Pred(Suc(Suc(Zero()))) : \mathsf{Nat}}}{Pred(Pred(Suc(Suc(Zero())))) : \mathsf{Nat}}}{IsZero(Pred(Pred(Suc(Suc(Zero()))))) : \mathsf{Bool}} \quad \frac{}{Zero() : \mathsf{Nat}} \quad \frac{\dfrac{}{Zero() : \mathsf{Nat}}}{Suc(Zero()) : \mathsf{Nat}}}{If(IsZero(Pred(Pred(Suc(Suc(Zero()))))),Zero(),Suc(Zero())) : \mathsf{Nat}}$$

## 1.2. Seguridad de un sistema de tipos

Una propiedad fundamental de un sistema de tipos es la llamada *seguridad* o *correctud* del sistema la cual dice lo siguiente:

*Los programas correctamente tipificados no pueden funcionar mal*

Un programa funciona mal si su evaluación se bloquea, es decir, termina en una expresión que no es un valor simple y no puede seguir evalúandose.

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

- El **progreso** que indica que un programa correctamente tipificado no se bloquea.
- La **preservación** que dice que si un programa correctamente tipificado se ejecuta entonces la expresión resultante está correctamente tipificada y en muchos casos el tipo de ambos coincide.

A continuación se muestra la demostración de estas propiedades para EAB.

***Proposición (Unicidad del tipo)***

*Para cualquier expresión e de EAB existe a lo más un tipo* $\mathsf{T}$ *tal que se cumple la relación*

$$e : \mathsf{T}$$

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

*Demostración.* Por inducción estructural sobre *e,* usando las reglas de tipificado (más la hipótesis de inducción) para cada caso. □

### 1.2.1. Progreso

***Proposición (Progreso)***

*Si* $e : \mathsf{T}$ *para algún tipo* $\mathsf{T}$*, es decir, se puede derivar el tipo de e, entonces se cumple únicamente una de las siguientes condiciones:*

- *e es un valor*

- *existe una expresión $e'$ tal que $e \to e'$*

*Demostración.* *Por inducción sobre $e : \mathsf{T}$.*

*Casos base:*

*Los casos para True(), False() y Zero() son inmediatos, pues son valores en nuestro sistema de transición.*

*Para otros casos procedemos como sigue:*

*Caso 1:*

*En este caso, $e = Suc(e_1)$ y $e_1 : \mathsf{Nat}$. Por hipótesis de inducción, $e_1$ es un valor o existe algún $e_1'$ tal que $e_1 \to e_1'$. Dado que $e_1 : \mathsf{Nat}$, si $e_1$ es un valor entonces es numérico y por ende $e$ es un valor también. Por otro lado, si $e_1 \to e_1'$ entonces $e \to Suc(e_1')$ por definición de $\to$.*

*Caso 2:*

*En este caso, $e = Pred(e_1)$ y $e_1 : \mathsf{Nat}$. Por hipótesis de inducción, $e_1$ es un valor o existe algún $e_1'$ tal que $e_1 \to e_1'$. Dado que $e_1 : \mathsf{Nat}$, si $e_1$ es un valor entonces es o bien $Zero()$ o bien $Suc(n)$, casos para los cuales existe un caso respectivo dentro de la definición de $\to$. Por otro lado, si $e_1 \to e_1'$ entonces $e \to Pred(e_1')$ por definición de $\to$.*

*Caso 3:*

*En este caso, $e = IsZero(e_1)$ y $e_1 : \mathsf{Nat}$. Por hipótesis de inducción, $e_1$ es un valor o existe algún $e_1'$ tal que $e_1 \to e_1'$. Dado que $e_1 : \mathsf{Nat}$, si $e_1$ es un valor entonces es o bien $Zero()$ de donde $e \to True()$ o bien $Suc(n)$ de donde $e \to False()$ por definición de $\to$. Por otro lado, si $e_1 \to e_1'$ entonces $e \to IsZero(e_1')$ por definición de $\to$.*

*Caso 4:*

*En este caso, $e = If(e_1, e_2, e_3)$ y $e_1 : \mathsf{Bool}$, $e_2 : \tau$, $e_3 : \tau$. Por hipótesis de inducción, $e_1$ es un valor o existe algún $e_1'$ tal que $e_1 \to e_1'$. Dado que $e_1 : \mathsf{Bool}$, si $e_1$ es un valor, entonces es o bien True() de donde $e \to e_2$ o bien False() de donde $e \to e_2$ por definición de $\to$. Por otro lado, si $e_1 \to e_1'$ entonces $e \to If(e_1', e_2, e_3)$ por definición de $\to$.* □

### 1.2.2. Preservación

***Proposición (Preservación de tipos)***

*Si* $e : \mathsf{T}$ *y* $e \to e'$ *entonces* $e' : \mathsf{T}$.

*Demostración. Por inducción sobre* $e : \mathsf{T}$.

*Casos base:*

*Los casos para True* (), *False* () *y Zero* () *se cumplen por vacuidad pues al ser valores, no existe ningún caso donde* $e \to e'$ *para cualquier* $e'$.

*Caso 1:*

*En este caso* $e = Succ(e_1)$, $e : \mathsf{Nat}$ *y* $e_1 : \mathsf{Nat}$. *A partir de la definición de* $\to$ *tenemos que existe una única regla para Succ , misma que podemos usar para reducir* $e \to e'$. *Esta misma regla nos dice que* $e_1 \to e_1'$ *y dado que sabemos que* $e_1 : \mathsf{Nat}$ *podemos aplicar la hipótesis de inducción con lo cual tenemos* $e_1' : \mathsf{Nat}$ *de donde* $Suc(e_1') : \mathsf{Nat}$.

*Caso 2:*

*En este caso* $e = Pred(e_1)$, $e : \mathsf{Nat}$ *y* $e_1 : \mathsf{Nat}$. *A partir de la definición de* $\to$ *tenemos que existen tres reglas para Pred :*

- *El primer caso es cuando* $e_1$ *es Zero () de donde* $e \to Zero()$ *que por nuestras reglas de tipado tiene el tipo* Nat.
- *El segundo caso es cuando* $e_1$ *es Suc (n) de donde* $e \to n$ *y dado que* $Suc(n) : \mathsf{Nat}$, *tenemos que* $n : \mathsf{Nat}$.
- *El tercero es análogo al caso 1 presentado anteriormente con la única regla faltante para Pred , misma que podemos usar para derivar* $e \to e'$. *Esta misma regla nos dice que* $e_1 \to e_1'$ *y dado que sabemos que* $e_1 : \mathsf{Nat}$ *podemos aplicar la hipótesis de inducción con lo cual tenemos* $e_1' : \mathsf{Nat}$ *de donde* $Pred(e_1') : \mathsf{Nat}$.

*Caso 3:*

*En este caso* $e = IsZero(e_1)$, $e : \mathsf{Bool}$ *y* $e_1 : \mathsf{Nat}$. *A partir de la definición de* $\to$ *tenemos que existen tres reglas para IsZero :*

- *El primer caso es cuando* $e_1$ *es Zero () de donde* $e \to True()$ *que por nuestras reglas de tipado tiene el tipo* Bool.
- *El segundo caso es cuando* $e_1$ *es Suc (n) de donde* $e \to False()$ *que por nuestras reglas de tipado tiene el tipo* Bool.

- *El tercero es análogo a los casos 1 y 2 presentados anteriormente con la única regla faltante para IsZero, misma que podemos usar para derivar $e \to e'$. Esta misma regla nos dice que $e_1 \to e_1'$ y dado que sabemos que $e_1 : \mathsf{Nat}$ podemos aplicar la hipótesis de inducción con lo cual tenemos $e_1' : \mathsf{Nat}$ de donde IsZero $(e_1') : \mathsf{Bool}$.*

*Caso 4:*

*En este caso $e = If(e_1, e_2, e_3)$, $e : \tau$, $e_1 : \mathsf{Bool}$, $e_2 : \tau$ y $e_2 : \tau$. A partir de la definición de $\to$ tenemos que existen tres reglas para If:*

- *El primer caso es cuando $e_1$ es True () de donde $e \to e_2$ que tiene el tipo $\tau$.*
- *El primer caso es cuando $e_1$ es False () de donde $e \to e_3$ que tiene el tipo $\tau$.*
- *El tercero es análogo a los casos 1, 2 y 3 presentados anteriormente con la única regla faltante para If, misma que podemos usar para derivar $e \to e'$. Esta misma regla nos dice que $e_1 \to e_1'$ y dado que sabemos que $e_1 : \mathsf{Bool}$ podemos aplicar la hipótesis de inducción con lo cual tenemos $e_1' : \mathsf{Bool}$ de donde $If(e_1', e_2, e_3) : \tau$.*

□

Podemos apreciar de las dos demostraciones anteriores que el proceso es largo, al requerir analizar prácticamente cada uno de los constructores del lenguaje y más aún en muchos casos la demostración es prácticamente idéntica. Si el lenguaje de programación incluyera un conjunto de instrucciones más variado, la demostración se volvería más extensa.

Existen otras formas de verificar la seguridad de un sistema de tipos, algunas de las cuales son:

1. Verificación estática: Se realiza durante el proceso de compilación de los programas, antes de que se ejecuten. El compilador verifica el uso correcto de los tipos en el código fuente, asegurándose de que todas las operaciones sean consistentes con las reglas del sistema de tipos.
2. Verificación dinámica: Se realiza durante el tiempo de ejecución del programa. En este caso, el sistema de tipos no es suficiente para garantizar la seguridad, por lo que se realizan comprobaciones adicionales durante la ejecución para garantizar la coherencia de los tipos.
3. Inferencia de tipos: Algunos lenguajes permiten inferir los tipos de las variables y expresiones en lugar de especificarlos explícitamente. El compilador analiza el código para deducir automáticamente los tipos y asegurarse de que sean coherentes.
4. Análisis de flujo de datos: Esta técnica rastrea cómo fluyen los datos a través del programa y verifica que se utilizan de manera segura según las reglas del sistema de tipos.

Notemos que todas estas formas son por completo a nivel programa, es decir, verifican los programas escritos de un lenguaje y no el lenguaje como tal.

Esto da pie a proponer una automatización del proceso de demostración. Como vimos anteriormente, para probar la seguridad de un sistema de tipos se relacionan las semánticas dinámica estática por medio de preservación y progreso. De la misma forma, la semántica del lenguaje es establecida mediante un sistema de transición que nos indica cómo pasar de un programa otro.

La automatización del proceso de demostración puede realizarse tomando como base este sistema de transición y verificando la propiedad de seguridad entre programas. Esto puede realizarse usando *verificación de modelos*.

## 1.3. Verificación de modelos tradicional

La verificación de modelos es una técnica de verificación formal que consiste en verificar automáticamente propiedades de un sistema o programa mediante el análisis exhaustivo de todos los posibles estados o ejecuciones del modelo del sistema. El objetivo es determinar si ciertas propiedades especificadas se cumplen o no para todas las posibles situaciones del sistema.

***Definición 1.6***

*Dado un sistema o programa S y una propiedad P que deseamos verificar, la verificación de modelos consiste en construir un modelo formal M que representa el comportamiento y estructura del sistema S y posteriormente aplicar algoritmos y técnicas automáticas para verificar si la propiedad P se cumple en todos los estados o ejecuciones del modelo M.*

En la definición anterior, el modelo $M$ puede ser una representación abstracta del sistema, como una gráfica de estados, un sistema de transiciones o una abstracción de alto nivel como el modelo de un protocolo o una especificación formal.

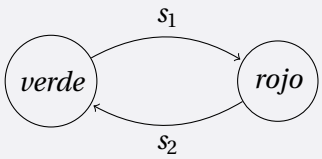
La verificación de propiedades se logra mediante la revisión sistemática de todas las posibles combinaciones de estados y acciones del sistema, o mediante la utilización sistema lógicos.

***Ejemplo 1.7***

Supongamos que queremos verificar una propiedad sencilla que modela un semáforo. El sistema tiene dos estados posibles $E = \{verde, rojo\}$, y sólo puede cambiar de estado si recibe una señal de cambio.

La propiedad $P$ que queremos verificar es que el semáforo no puede estar en el estado *rojo* y *verde* al mismo tiempo.

Para realizar la verificación de modelos, primero construimos un modelo formal del sistema que represente su comportamiento y estructura. En este caso podemos usar un sistema de transiciones que tenga dos estados (*verde* y *rojo* ) y una transición entre ellos cuando se recibe la señal de cambio ($s_1$ para pasar de verde a rojo y $s_2$ para pasar de rojo a verde).




Luego, utilizando herramientas de verificación de modelos, verificamos automáticamente la propiedad en el modelo.

### 1.3.1. Proceso de verificación de modelos

La forma tradicional de hacer verificación de modelos consiste en:

1. Formalizar el sistema computacional sobre el cual se realizará la verificación por medio de una estructura que nos permita manipularlo. Esto suele hacerse por medio de las llamadas *estructuras de Kripke* que consisten en un conjunto de estados, una relación de transición entre ellos y una función de etiquetamiento.

2. La propiedad que el sistema debe cumplir se expresa con una fórmula. Las fórmulas, normalmente escritas en una *lógica temporal* describen propiedades que puede cumplir una estructura de Kripke a lo largo del tiempo.

   Las lógicas temporales más usadas en verificación de modelos son LTL (*Linear Time Logic*) y CTL (*Computational Tree Logic*) que extienden a la lógica proposicional con operadores de tiempo y con cuantificadores de *trayectorias* las cuales son secuencias de estados.

3. La estructura de Kripke y la fórmula que exprese la propiedad deseada, son la entrada del verificador.
4. Cuando el verificador termina, nos dice si la estructura cumple o no con la propiedad especificada.

A grandes rasgos, hacer verificación de modelos es hacer una exploración exhaustiva del conjunto de estados de una estructura de Kripke para corroborar que dicha estructura cumple con una fórmula dada.

Algunas ventajas que trae consigo la verificación de modelos son:

- Sin demostraciones a mano. Todo el proceso se realiza completamente en automático.
- Contrajemplos. Si una estructura de Kripke no cumple una especificación, es posible generar una traza de ejecución y un contraejemplo[4].
- Prevención. No es necesario especificar por completo el sistema que se esta revisando. Esto permite encontrar errores antes de que el sistema esté completamente terminado.

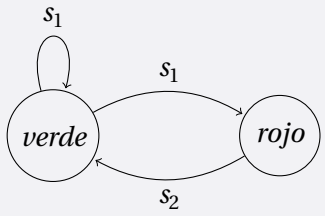
[4]Se entiende como contraejemplo a una instancia donde la propiedad en cuestión no se cumple, lo que demuestra que el modelo no satisface esa propiedad particular.

- Expresividad. Las lógicas temporales permiten expresar con facilidad las propiedades sobre sistemas concurrentes.

***Ejemplo 1.8***

Tomando como base el ejemplo del semáforo, si el verificador encuentra un contraejemplo, significa que la propiedad no se cumple en el sistema y se generará una secuencia de estados que muestra cómo el sistema podría estar en el estado *rojo* y *verde* al mismo tiempo.

Por ejemplo, supongamos que el modelo tiene una transición incorrecta con la misma señal en que se pasa de verde a rojo.




Entonces si el modelo está en el estado verde y recibe la señal de cambio, el sistema cambiará a rojo pero la transición incorrecta también permite que el sistema permanezca en el estado verde, lo que resulta en un contraejemplo que muestra que la propiedad $P$ no se cumple.

Con el contraejemplo podemos identificar el problema en el modelo y corregir la transición incorrecta para garantizar que la propiedad se cumpla en el sistema. Esto nos permita verificar que el semáforo no puede estar en estado rojo y verde al mismo tiempo.

### 1.3.2. Incremento explosivo de estados

La *explosión de estados* es un problema común en la verificación de modelos. Se refiere al rápido crecimiento del número de estados posibles del sistema o programa que se deben analizar durante el proceso de verificación. Surge cuando el sistema tiene muchas variables, estados y decisiones, lo que lleva a un gran número de posibles combinaciones de estados y transiciones. A medida que se verifican propiedades en el modelo, se deben explorar y analizar todos estos posibles estados y transiciones, lo que puede requerir una cantidad masiva de recursos computacionales y tiempo. [4]

La explosión de estados puede hacer que la verificación sea impracticable o muy costosa especialmente para sistemas complejos o de gran tamaño. Incluso sistemas relativamente pequeños pueden dar lugar a modelos con un número de estados tan grande que la verificación se vuelve inviable. [4]

Para abordar este problema, se han desarrollado varias técnicas y estrategias para reducir la explosión de estados como: [4]

1. Técnicas de abstracción: Utilizar técnicas de abstracción para simplificar el modelo y eliminar detalles innecesarios que no afectan la propiedad que se está verificando.
2. Técnicas de poda: Identificar y eliminar partes del modelo que no son relevantes para la propiedad que se está verificando.
3. Técnicas de particionamiento: Dividir el modelo en partes más pequeñas y manejables, lo que facilita la verificación de cada parte por separado.
4. Uso de herramientas y algoritmos eficientes: Utilizar herramientas y algoritmos de verificación que estén optimizados para manejar modelos con grandes cantidades de estados.
5. Enfoque de propiedades críticas: Centrarse en verificar propiedades críticas y de interés clave en lugar de intentar verificar todas las propiedades posibles.
6. Uso de técnicas de verificación simbólica: Utilizar técnicas simbólicas que permitan analizar múltiples estados y transiciones al mismo tiempo, en lugar de explorar uno por uno.

Aunque la explosión de estados sigue siendo un desafío en la verificación de modelos, estas técnicas y enfoques ayudan a reducir el tamaño del modelo a verificar y a permitir que la verificación sea más factible para sistemas grandes y complejos. Sin embargo, es importante tener en cuenta que la verificación de modelos puede seguir siendo un problema desafiante y puede requerir el uso de técnicas avanzadas y potentes.

Podemos usar verificación de modelos para verificar la seguridad de un sistema de tipos, donde el sistema de transición puede ser representado mediante estructura de Kripke y la propiedad de seguridad sería lo que queremos verificar sobre la estructura (el sistema de tipos). En este caso, nuestras reglas de semántica estática y dinámica funcionarían como la lógica. Esta técnica de verificación de modelos llamada *Glass Box* cambia la forma tradicional de hacer verificación de modelos que además trata el problema de explosión de estados.

## 1.4. Verificación de Modelos Glass Box

La verificación de Modelos Glass Box es una técnica de verificación de modelos que trata el problema de explosión de datos sobre problemas específicos donde la especificación de los estados no es del todo posible. En el caso de este trabajo, sería complicado e incluso imposible especificar todos los posibles programas que se pueden escribir en un lenguaje. La técnica de verificación de modelos Glass Box presenta una metodología para realizar verificación de propiedades en este tipo de casos. Su principal aporte es mitigar la explosión de estados. El enfoque en general consiste en los siguientes pasos:

1. Proveer la especificación de lo que se desea verificar: Estructuras, lenguajes, funciones, operaciones, propiedades, etc. Esto dependerá del problema que se esté estudiando.

2. Construir el espacio de búsqueda: Para construir el espacio de búsqueda se deben poder generar todos los posibles estados bajo una especificación dada. Para lograr esto, puede hacerse uso de un solucionador SAT, mismo que deberá alimentarse de una fórmula proposicional generada por la especificación cuyo fin es describir todas las restricciones que deben cumplir los estados que se van a generar.

3. El paso siguiente consiste en verificar si se cumple la propiedad deseada: Para esto se deberá escoger una operación sobre los estados, lo cual emula una transición y posteriormente verificar si cumplen la propiedad deseada.

   Si por alguna razón se encuentra un estado que no cumple la propiedad deseada, se reporta al usuario dando el estado analizado como contraejemplo.

4. Reducir el espacio de búsqueda mediante una poda al espacio de búsqueda. Después de verificar que un estado cumpla con la propiedad deseada, se realizará el proceso de poda quitando todos aquellos estados del espacio de búsqueda que sean similares al mismo.

   Para realizar la poda, ajustamos la fórmula dada al solucionador SAT y continuamos sobre un nuevo espacio sin todos estos estados que cumplen la propiedad.

***Ejemplo 1.9***

Por ejemplo, podemos verificar que las operaciones implementadas para un árbol ordenado son correctas. Los estados en este caso son todos los posibles árboles, mientras que las transiciones entre ellos están dadas por las operaciones de agregar, eliminar, buscar, etc. La propiedad a verificar podría ser que el árbol se mantenga ordenado en todo momento.

Observemos que se tiene un gran número de estados y por el otro ¿cómo los generamos? Estos no pueden ser generados a mano por el usuario, pues hay un vasto número de ellos y nuestro objetivo es que la herramienta sea completamente automática. Este tipo de situaciones no es atacado por los verificadores de modelos tradicionales basados en lógicas temporales.

Con el fin de ejemplificar el proceso propuesto en Glass Box se muestra de forma superficial el proceso de verificación para árboles binarios ordenados.

**(1) Especificación**

La especificación debe contar con todos los elementos que permitan construir cada uno de los estados del espacio de búsqueda. En este caso la construcción de los árboles binarios ordenados. En [1] se muestra una breve especificación usando un subconjunto de JAVA. Veamos una estructura similar, simplificada para fines del ejemplo.

```
public class ArbolBinarioOrdenado<T> {
   private class Nodo {
      ...
   }
```

```
    public <T> get(int i) { ... }

    public void agrega(int i, T elemento) { ... }

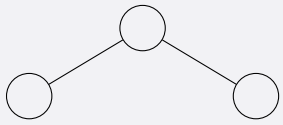
    public boolean esOrdenado() { ... }
}
```

El método `esOrdenado()` es la propiedad que queremos verificar. Básicamente se desea construir un verificador que dada esta especificación de árboles binarios ordenados se asegure de que al agregar un nuevo elemento el árbol siempre va a seguir ordenado. Lo mismo podría aplicar con otras operaciones.

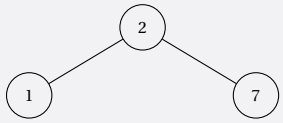
**(2) Construcción del espacio de búsqueda**

Para construir el espacio de búsqueda, acotamos los estados para que sea finito. En este caso verificaremos todos los posibles árboles de una altura dada. Para ello se fija una altura y se construye un esqueleto, que será llenado con valores válidos con respecto a la especificación; en este caso deberá ser llenada de acuerdo con la propiedad *estar ordenado* dada por el método `esOrdenado` de la especificación.

*Esqueleto:*



*Árbol válido*




El verificador construirá a partir de `esOrdenado` una fórmula $\varphi$ que represente la propiedad de estar ordenado y por medio de un solucionador SAT obtendrá todos los estados válidos, que en este caso serán árboles ordenados. Llamaremos a este conjunto de estados $W$.

**(3) Verificar la propiedad deseada y podar**

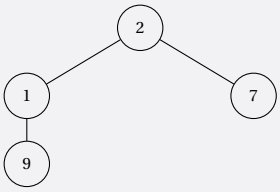
El proceso de verificación consiste en tomar cada posible estado, quitar aquellos que sean similares y continuar hasta que se consuman todos los estados. La forma de proceder es la siguiente:

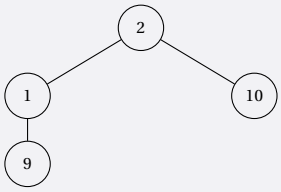
*Sea W el espacio de búsqueda que contiene árboles binarios ordenados.*

*Mientras W no sea vacío:*

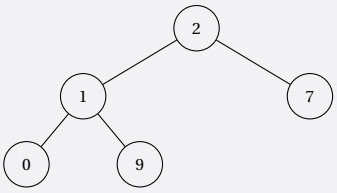
- *Escoger un árbol s de W*
- *Agregar un elemento a s y verificar si el árbol resultante sigue ordenado.*
- *Si el paso anterior falla: Imprimir como contraejemplo el árbol y el elemento que se intentó agregar.*
- *Sea P el conjunto de todos los árboles similares a s.*
- $W := W - P$

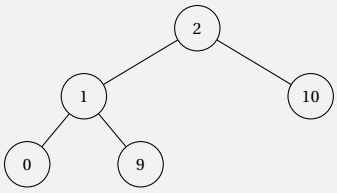
Para comprender el concepto de similitud analicemos los siguientes dos árboles:




Al agregar un 0 como elemento obtendremos:




Si nos damos cuenta, la raíz e hijo izquierdo eran idénticos en ambos árboles al inicio. Después de agregar el 0, el orden se preservó. Esto se va a cumplir para todos aquellos árboles que tengan la misma raíz y mismo hijo izquierdo por lo que no tiene sentido verificarlos todos, con lo cual podemos quitar todos los árboles similares después de verificar que uno cumple la propiedad deseada. A este proceso lo llamamos poda.

Por supuesto, el proceso de detección de árboles similares se realiza sobre otras propiedades adicionales a que la raíz y el hijo izquierdo sean iguales en ambos árboles; sin embargo, para los fines del ejemplo es suficiente.

El objetivo de este trabajo es utilizar la técnica Glass Box para verificar la seguridad de un sistema de tipos. Este proceso se ilustra de manera teórica en [1] sin contar con una implementación pero mencionando el uso de un

subconjunto de JAVA que propone la generación de fórmulas añadiendo una etiqueta `@declarative` que facilite la traducción de la especificación para alimentar al solucionador SAT.

La diferencia de este trabajo con respecto al trabajo original [15] consiste en hacer uso del lenguaje de programación funcional HASKELL que permita tanto facilitar la especificación del lenguaje como la generación de las restricciones dadas al solucionador SAT de forma más sencilla debido al enfoque declarativo del mismo.

## 1.5. Solucionadores SAT

El problema de satisfactibilidad para la lógica proposicional, usualmente denotado como SAT es de suma importancia en la teoría de la complejidad computacional y en general en Ciencias de la Computación. En su forma más común, el problema SAT se define como:

*Dado un conjunto $P = \{p_1, \ldots, p_n\}$ de variables proposicionales y un conjunto $C$ de cláusulas con variables en $P$ ¿Existe una interpretación $\mathscr{I}$ que satisfaga a $C$?*

Este problema tiene aplicaciones tanto teóricas como prácticas. En particular resultó ser el primer problema NP-Completo, lo cual fue probado por Cook en 1971. De hecho antes del Teorema de Cook, el concepto de NP completud ni siquiera existía. A principios de los años 90, los principales problemas consideraban 100 variables, 200 cláusulas. Hoy en día se consideran problemas de incluso un millón de variables y cinco millones de cláusulas. [20]

El área de la lógica conocida como razonamiento automatizado se encarga, entre muchas otras cosas, del desarrollo de algoritmos para resolver el problema SAT sobre todo con respecto a ciertas familias de cláusulas para los que sí existen algoritmos eficientes, como son las llamadas cláusulas de Horn. Hoy en día existen programas sumamente eficientes para resolver el problema SAT, los cuales se conocen como solucionadores SAT[5] y tienen muchas aplicaciones relevantes fuera de la lógica.

### Aplicaciones

Algunas aplicaciones de este tipo de software son: [20]

- Verificación de hardware vía Verificación de Modelos[6]: Muchas de las compañías que desarrollan hardware (como INTEL) usan solucionadores SAT para verificar los diseños de sus chips.
- Verificación de software:
  - Solucionadores SAT basados en SMT[7] son usados para verificar algunos productos de MICROSOFT.
  - Software embebido en autos, aviones, refrigeradores, etc.
  - Paquetes de Unix.

[5] *SAT solvers*

[6] Hablaremos de esto más adelante.

[7] Un solucionador SAT basado en SMT (Satisfiability Modulo Theories) es una herramienta que combina las capacidades de resolución de problemas de satisfacción booleana (SAT) con técnicas de razonamiento en teorías específicas, como aritmética, teoría de conjuntos, teoría de listas, entre otras.

- Pilotos automáticos y calendarización en Inteligencia Artificial: Hoy en día es uno de los mejores enfoques del pilotaje automático.
- Un gran número de problemas numéricos (Tripletas pitagóricas)
- Solución a otros problemas NP-Difíciles (coloración, clique[8], etc.).

### *Ejemplo 1.10*

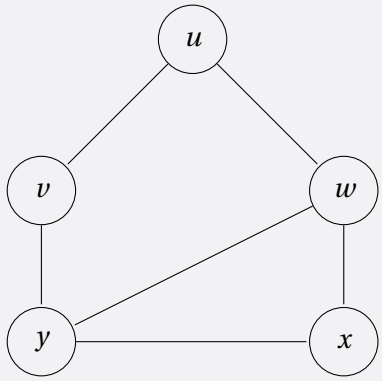
**Problema** (Coloración de gráficas) Dada una gráfica no dirigida $G = (V, A)$, una coloración asigna colores a los vértices, tal que todos los vértices adyacentes tienen colores distintos. Una coloración de a lo más $k$ colores es llamada $k$-coloración. El problema de coloración de gráficas consiste en verificar si existe una $k$-coloración para $G$.

- Codificación: Usa $k \times |V|$ variables lógicas $v_j$ con $v \in V$, $1 \leq j \leq k$, donde $v_j$ es verdadera si el vértice $v$ tiene el color $j$.
- Cláusulas:

  $(v_1 \vee \ldots \vee v_k)$ *Todo vértice tiene un color.*

  $\left(\neg u_1 \vee \neg v_j\right)$ *Los vértices adyacentes tienen diferente color.*

Por ejemplo, en la siguiente gráfica queremos obtener una 3-coloración.




El conjunto de vértices es $V = \{u, v, w, x, y\}$, mientras que se tienen por ejemplo 3 los tres colores rojo (1), verde (2) y azul (3). De esta forma, dado que $k = 3$, necesitamos $k \times |A| = 3 \times 5 = 15$ variables lógicas $u_1, u_2, u_3, \ldots, y_1, y_2, y_3$.

Partiendo de esto definimos las cláusulas correspondientes:

[8]Consiste en encontrar el clique más grande en una gráfica no dirigida. Es decir, se busca un conjunto de vértices donde cada par de vértices está conectado por una arista, y dicho conjunto es máximo en términos de cantidad de vértices.

*Cada vértice tiene un color*

$$(u_1 \vee u_2 \vee u_3)$$
$$\vdots$$
$$\big(y_1 \vee y_2 \vee y_3\big)$$

*Los vértices adyacentes tienen diferentes colores*

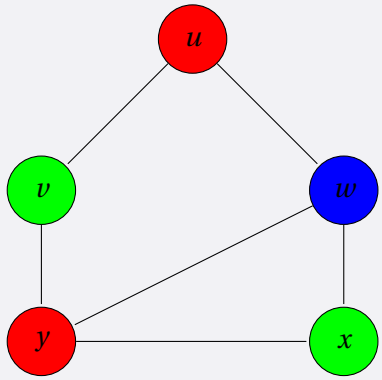
$$(\neg u_1 \vee \neg v_1) \wedge \ldots \wedge (\neg u_3 \vee \neg v_3)$$
$$\vdots$$
$$\big(\neg x_1 \vee \neg y_1\big) \wedge \ldots \wedge \big(\neg x_3 \vee \neg y_3\big)$$

Ahora que tenemos nuestra codificación podemos proceder a encontrar la solución, la cual consiste en encontrar valores de las variables que hagan satisfacibles las fórmulas anteriores.

Una solución a esto es:

$\big\{(u_1,1),(v_2,1),(w_3,1),\big(y_1,1\big),(x_3,1),\ldots\big\}$

El resto de asignaciones $u_i, v_i, w_i, x_i, y_i$ tienen un valor de 0. La cual en efecto es una solución al problema de 3-coloración. Si listamos todas las posibles asignaciones que hagan satisfacible la fórmula encontraremos todas las posibles 3-coloraciones para esta gráfica.




### Solucionadores SAT incrementables

En ocasiones necesitamos resolver una secuencia de instancias SAT que comparten cláusulas entre ellos. ¿Podemos resolver estas secuencias de forma eficiente?

Los solucionadores SAT incrementales añaden y remueven cláusulas constantemente. ¿Cuál es el beneficio? El

solucionador puede recordar cláusulas aprendidas y algunos otros elementos que requieren las heurísticas como podas por ejemplo. Esto hace que sean más veloces.

Un solucionador SAT conocido que usa esta técnica es MINISAT. Es un solucionador SAT minimalista de código abierto que se usa para iniciar en la investigación de proyectos al rededor de SAT. Se publica bajo la licencia del MIT. Algunas de sus características son:

- Es tan pequeño, está tan bien documentado, y posiblemente tan bien diseñado, que lo convierte en un punto de partida ideal para adaptar técnicas basadas en SAT a problemas de dominio específico.
- Ganó varios premios en la competencia SAT de 2005.
- Es un solucionador SAT incremental que además tiene mecanismos para agregar restricciones sin cláusulas. Esto hace que sea una excelente herramienta para integrarse junto con varios sistemas de software.

En este trabajo haremos uso de MINISAT [18] para poder generar todos los posibles programas válidos que requiere la técnica Glassbox para funcionar. Esto permitirá que a partir de una serie de restricciones se pueda generar todos estos programas de manera automática y rápida.

# Capítulo 2

# Implementación puramente funcional

En este capítulo se presentan las principales características de HASKELL que se emplearon para implementar el verificador propuesto en este trabajo así como la integración de todos estos elementos. Este material fue tomado principalmente de las referencias [16], [15], [9] y [18].

## 2.1. HASKELL

El sistema implementado en este trabajo se encuentra desarrollado por completo en el lenguaje de programación HASKELL. De acuerdo con [16] el trabajo previo realizado por Roberson se realizó en un subconjunto del lenguaje de programación JAVA que añade las anotaciones `@declarative` con el fin de poder traducir ciertos métodos en fórmulas proposicionales de forma sencilla. El uso de un lenguaje funcional como HASKELL hace que esto sea más simple pues es declarativo por naturaleza. A continuación se describen algunas bondades que se aprovechan de este lenguaje.

### 2.1.1. Principales características

Algunas de las principales características de HASKELL se listan a continuación:

- **Sin efectos secundarios:** No se puede asignar un valor a una variable y luego cambiarlo.
- **Transparencia referencial:** Si una función es llamada con los mismos parámetros más de una vez, siempre se obtiene el mismo valor.
- **Evaluación perezosa:** No se evalúan expresiones hasta que sea estrictamente necesario.
- **Verificación de tipos estática:** El tipo de una expresión se conoce en tiempo de compilación.
- **Inferencia de tipos:** No es necesario etiquetar explícitamente cada expresión con un tipo. El sistema de tipos lo puede deducir.

### 2.1.2. Listas por comprensión

Una forma de representar listas, que es bastante utilizada en este trabajo, es la notación llamada *por comprensión* donde al igual que en la Teoría de Conjuntos, es posible especificar conjuntos a partir de propiedades u otros conjuntos ya existentes.

***Ejemplo 2.1***

El conjunto

$$\{2x \mid x \in \{0,1,2,3,4,5\}\}$$

denota al conjunto $\{0,2,4,6,8,10\}$, es decir, al conjunto de números $2x$, tal que $x$ es un elemento del conjunto $\{0,1,2,3,4,5\}$.

Es posible representar de la misma manera este conjunto en HASKELL. Se tiene una notación por comprensión similar que se puede usar para construir nuevas listas a partir de otras existentes.

```
Prelude> [2*x | x ← [0..5]]
[0,2,4,6,8,10]
```

Otro ejemplo podría ser:

```
Prelude> [even y | y ← [2..6]]
[True,False,True,False,True]
```

las expresiones `x <- [1..5]` y `y <- [2..6]` en las ejecuciones anteriores, son llamadas *generadores* y se puede tener más de uno dentro de las listas por comprensión, separando éstos por comas. La sintaxis de un generador puede ser alguna de las siguientes:

```
<variable> ← <lista>
(<variable>,<variable>) ← <lista de pares>
(<variable>,<variable>,<variable>) ← <lista de tuplas de tamaño 3>
...
```

También se pueden emplear guardias para filtrar los valores producidos por los generadores. Si la guardia es verdadera, entonces se mantiene el valor en la lista; en caso contrario se descarta.

***Ejemplo 2.2***

A continuación se define una función que obtiene los factores de un número entero *n*. La lista se genera por comprensión, indicando un generador que toma valores de 1 a *n* (`[1..n]`) con la condición de que el módulo de *n* respecto a cada elemento de la lista sea igual a cero (el residuo correspondiente es cero y por lo tanto son factores de *n*). La ejecución de la función muestra los factores del número 10, la lista por comprensión resultante queda como sigue:

```
[x | x ← [1..10], mod 10 x == 0]
```

```
factores :: Int → [Int]
factores n = [x | x ← [1..n], n `mod` x == 0]
```

```
Prelude> factores 10
[1,2,5,10]
```

### 2.1.3. Programación Origami

La *programación origami* es un término utilizado para referirse a un enfoque de programación funcional que se centra en la manipulación de estructuras de datos como listas, utilizando esquemas recursivos. Este enfoque se inspira en la técnica de plegado de papel llamada origami, donde se doblan y manipulan hojas de papel para obtener diferentes formas. En la programación origami sobre listas, se utilizan funciones de orden superior para realizar operaciones sobre listas de manera elegante y concisa. Estas funciones permiten aplicar una operación a cada elemento de la lista, combinando los resultados parciales para obtener un resultado final. A lo largo de este trabajo se hace uso de las funciones de plegado: `foldr` y `foldr1`.

**foldr**

`foldr` captura un esquema recursivo común al definir funciones.

***Ejemplo 2.3***

La siguiente función que suma los elementos de una lista.

```
sumaLst :: [Int] → [Int]
sumaLst [] =  0
sumaLst (x:xs) = x + sumaLst xs
```

contiene un esquema común que consiste en aplicar de manera encadenada una función a todos los elementos de una lista, en este caso la suma pero que puede cambiarse por una multiplicación, división, disyunción, conjunción u otra operación entre dos elementos.

Este esquema requiere cambiar los siguientes elementos por cada operación necesaria:

- El valor v a devolver al llegar a la lista vacía, también llamado *neutro*. Por ejemplo, para la suma se devuelve 0 y para la multiplicación se devuelve 1.
- La función `f` a aplicar.
- La lista a la cual se le realizará el proceso.

Este es el patrón capturado por `foldr`:

```
foldr :: (a → b → b) → b → [a] → [b]
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr v xs)
```

Con lo cual, nuestra función puede ser reescrita de la siguiente manera por medio de `foldr`:

```
sumaLst :: [Int] → [Int]
sumaLst xs = foldr (+) 0 xs
```

**foldr1**

`foldr1` captura prácticamente el mismo esquema recursivo que `foldr` con la diferencia de que no considera un valor para el caso base, sino que toma el último elemento como dicho valor.

***Ejemplo 2.4***

Usando `foldr` la llamada a `sumaLst` con la lista `[1,2,3]` quedaría como:

```
sumaLst [1,2,3]
foldr (+) 0 [1,2,3]
1 + foldr (+) 0 [2,3]
1 + 2 + foldr (+) 0 [3]
1 + 2 + 3 + foldr (+) 0 []
1 + 2 + 3 + 0
6
```

Por otro lado, la especificación de la función `foldr1` es:

```
foldr1 :: (a → b → b) → [a] → [b]
foldr1 _ [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

con lo cual nuestra función puede ser reescrita de la siguiente manera por medio de `foldr`:

```
sumaLst :: [Int] → [Int]
sumaLst xs = foldr1 (+) xs
```

y la ejecución con la lista `[1,2,3]` quedaría como:

```
sumaLst [1,2,3]
foldr1 (+) [1,2,3]
1 + foldr1 (+) [2,3]
1 + 2 + foldr1 (+) [3]
1 + 2 + 3
6
```

Donde se aprecia que no es necesario el parámetro para el caso base que necesita `foldr`.

### 2.1.4. Biblioteca `MiniSAT`

HASKELL cuenta con una biblioteca llamada `MiniSAT` que es un solucionador SAT que permite implementar la generación del espacio de búsqueda del sistema desarrollado en este trabajo. Para ello incluye un lenguaje para representar fórmulas proposicionales mediante los siguientes constructores:

| Constructor | Descripción |
|---|---|
| `Var v` | Una variable |
| `Yes` | La fórmula que siempre es verdadera |
| `No` | La fórmula que siempre es falsa |
| `Not (Formula v)` | Negación |
| `(Formula v) :&&: (Formula v)` | Conjunción |
| `(Formula v) :\|\|: (Formula v)` | Disyunción |
| `(Formula v) :++: (Formula v)` | Disyunción exclusiva |
| `(Formula v) :->: (Formula v)` | Implicación |
| `(Formula v) :<->: (Formula v)` | Si y sólo si |
| `All [Formula v]` | Todas son verdaderas |
| `Some [Formula v]` | Al menos una es verdadera |
| `None [Formula v]` | Ninguna es verdadera |
| `ExactlyOne [Formula v]` | Exactamente una es verdadera |
| `AtMostOne [Formula v]` | A lo más una es verdadera |

e incluye las siguientes funciones:

- ```
  satisfiable :: Formula v → Bool
  ```

  que verifica si una fórmula es satisfacible. Por ejemplo, supongamos la fórmula proposicional $p \vee \neg p$, podemos traducirla y pasarla por MINISAT como sigue:

  ```
  > satisfiable ((Var "p") :||: (Not (Var "p")))
  True
  ```

- ```
  solve :: Formula v → Maybe (Map v Bool)
  ```

  que regresa un valor de las variables que hace satisfactible a la fórmula si es que existe alguno. La salida la da por medio del tipo `Map` que asocia dos valores, en este caso variables con su valor de verdad. Se engloba en un `Maybe` pues en caso de no existir genera un valor `Nothing`. Por ejemplo, para la fórmula proposicional anterior:

  ```
  > solve ((Var "p") :||: (Not (Var "p")))
  Just (fromList [("p",False)])
  ```

- ```
  solve_all :: Formula v → [(Map v Bool)]
  ```
  ```

que regresa todos los estados que hacen satisfacible a la fórmula. La salida en este caso es una lista con todos los mapeos que contienen asignaciones a las variables que hacen verdadera a la fórmula. Por ejemplo, para la fórmula proposicional anterior:

```
> solve_all ((Var "p") :||: (Not (Var "p")))
[fromList [("p",False)],fromList [("p",True)]]
```

Para más información se recomienda consultar la referencia [18].

## 2.2. Arquitectura del sistema

La interfaz del verificador, con la que interactúa el usuario, se puede resumir como muestra la Figura 2.1.

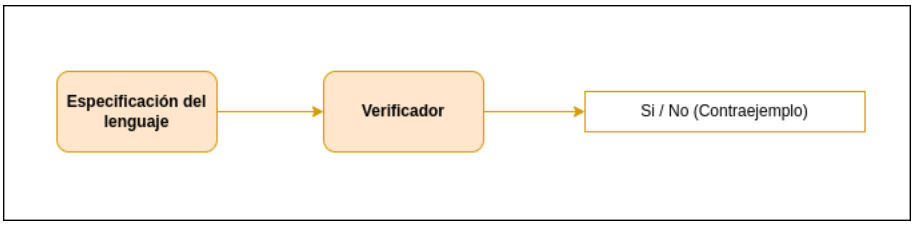



Figura 2.1: Arquitectura global del verificador

Los detalles particulares sobre la implementación se resumen a continuación:

**Especificación del espacio de búsqueda** Lo primero que se debe realizar es que el usuario deberá coloque la especificación del lenguaje que se desea verificar. Los elementos que debe proveer por cada lenguaje a verificar son:

1. Altura del espacio de búsqueda, es decir el tamaño de los programas. La sección 2.3. profundiza en este concepto.
2. Nombre de los constructores (*tokens*) que reconoce el lenguaje de programación así como el número de argumentos que pueden recibir y su tipo.
3. Una función `smdin` que implemente la semántica dinámica de paso pequeño del lenguaje.
4. Una función `smest` que implemente la semántica estática para realizar la verificación de tipos del lenguaje.

**Generación del espacio de búsqueda** El proceso de generación del espacio de búsqueda es el encargado de generar todos los posibles estados (programas) del espacio de búsqueda. Este módulo es quizá el más relevante en todo el desarrollo del trabajo pues se divide en tres submódulos que realizan labores complejas:

1. Generación de las restricciones que deben cumplir todos los programas del espacio de búsqueda. La generación de estas fórmulas debe tomarse de casos comunes entre cualquier lenguaje de programación y las reglas particulares del lenguaje, como el número de argumentos y tipos de los *tokens*.
2. Procesamiento de las restricciones por medio de MiniSAT, mismo que después de ejecutarse genera como salida los estados que hacen satisfacible a la fórmula proporcionada o dicho de otro modo, los programas que cumplen con las restricciones dadas.
3. Traducción de los resultados dados por el solucionador SAT. Partiendo de los estados de cada una de las variables proposicionales, construir el programa correspondiente que será verificado con el apoyo de la semántica dinámica y estática antes proporcionada.

**Verificación del espacio de búsqueda y poda** Dado el espacio de búsqueda generado por el módulo anterior, se procede con la verificación de los estados. Esto se logra por medio de las propiedades de progreso y preservación basadas en la semánticas dinámica y estática proporcionadas por el usuario.

El proceso de verificación dado en [16] que usa esta implementación se muestra en el Algoritmo 2.1.

**Algoritmo 2.1** Verificación del espacio de búsqueda

*Sea W el espacio de búsqueda que contiene programas bien tipados.*
*Mientras W no sea vacío:*

- *Escoger un programa s de W*
- *Ejecutar un paso de evaluación en s.*
- *Si progreso o preservación fallan: Imprimir el programa como contraejemplo.*
- *Sea P el conjunto de todos los programas similares a s.*
- $W := W - P$.

El proceso de detección de programas similares consiste en verificar la raíz del programa y el primer hijo de la raíz, para determinar el comportamiento similar entre programas. Una vez detectados estos elementos de $P$ similares a $s$, se procede a añadir la restricción $\neg P$ al solucionador SAT y volverlo a ejecutar para obtener un espacio de búsqueda reducido, que soluciona el problema de la explosión de estados.

Un diagrama de la interacción entre estos componentes desarrollados en HASKELL se muestra en la Figura 2.2.

- El módulo de `Especificación` es el lugar donde se proveen todos los elementos del lenguaje a verificar.
- El módulo `Formulas` se encarga de la generación de restricciones para el solucionador SAT.
- El módulo `Verificador` se encarga de todo del proceso de verificación del espacio de búsqueda por lo tanto se comunica con todos los módulos.

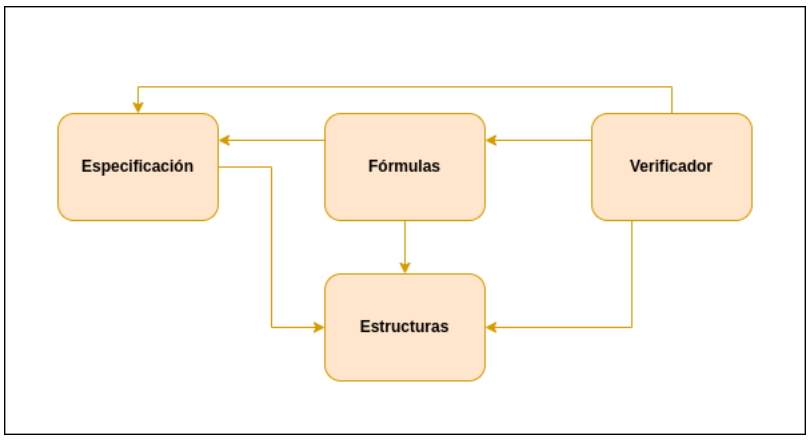



Figura 2.2: Módulos involucrados en la implementación

- Finalmente, el módulo `Estructuras` es el encargado de proporcionar la representación de los programas: Tipos para representar tokens, reglas para representar sistemas de tipos, árboles, etc.

Las siguientes secciones describen a detalle los módulos involucrados y se proveen ejemplos mediante el lenguaje EAB.

## 2.3. Especificación del lenguaje

La especificación del lenguaje debe proveerse en el módulo ESPECIFICACIÓN que debe contener los siguientes 4 elementos:

1. Altura del espacio de búsqueda.

   Esta debe proveerse por medio de la constante `altura`. Esta altura indica el tamaño de los programas. Este parámetro es importante pues no se verifican *todos* los posibles programas que se generan en un lenguaje, sino que los acotamos por tamaño, con respecto a los niveles del árbol de sintaxis abstracta de los mismos.

```
altura :: Int
```

***Ejemplo 2.5***

En nuestro ejemplo, acotamos la altura máximo de los árboles a 2.

```
altura :: Int
altura = 2
```

2. Nombre de los constructores (*tokens*) que reconoce el lenguaje de programación así como el número de argumentos que pueden recibir y su tipo.

   Para especificar los *tokens* del lenguaje debe proveerse la constante `lexemas`. Esta constante hace uso del tipo de dato `Token`. Dada la diversidad de *tokens* que pueden tener los distintos lenguajes de programación, se define el tipo con un constructor `Tk` que representa mediante una cadena el nombre de los *tokens*.

```
lexemas :: [Token]
```

   Para especificar el número de argumentos de los *tokens* se define la función `aridad` que dado un *token* indica el número de argumentos de éstos.

```
aridad :: Token → Int
```

   El tipo de los argumentos del *token* requiere básicamente de la especificación del sistema de tipos. Para ello se provee el tipo `STipos` que permite representar reglas de inferencia en sistemas de tipos. La función `stipos` incluye esta especificación.

```
stipos :: STipos
```

***Ejemplo 2.6***

En nuestro ejemplo, los lexemas a ocupar son:

```
lexemas :: [Token]
lexemas = [Tk "True", Tk "False", Tk "Zero",
           Tk "Suc", Tk "Pred", Tk "IsZero",
           Tk "If"]
```

La aridad de cada token es:

```
aridad :: Token -> Int
aridad (Tk "True")   = 0
aridad (Tk "False")  = 0
aridad (Tk "Zero")   = 0
aridad (Tk "Suc")    = 1
aridad (Tk "Pred")   = 1
aridad (Tk "IsZero") = 1
aridad (Tk "If")     = 3
```

El tipo de los argumentos de cada token es:

```
stipos :: STipos
stipos = [([],(Tk "True", Conc "Bool")),
          ([],(Tk "False", Conc "Bool")),
          ([],(Tk "Zero", Conc "Nat")),
          ([Conc "Nat"], (Tk "Suc", Conc "Nat")),
          ([Conc "Nat"], (Tk "Pred", Conc "Nat")),
          ([Conc "Nat"], (Tk "IsZero", Conc "Bool")),
          ([Conc "Bool",VT, VT], (Tk "If", VT))]
```

Observar que se tienen dos posibles opciones de tipos: (1) `Conc` para especificar tipos concretos o básicos y (2) `VT` para variables de tipo, como el caso de `if` donde el tipo de sus ramas no es concreto sino que puede ser cualquiera.

3. Una función `smdin` que implemente la semántica dinámica de paso pequeño del lenguaje.

   Para especificar la semántica dinámica se debe proporcionar la función `smdin` que dado un árbol de sintaxis abstracta (`Arbol`) compuesto por *tokens* como valores de cada nodo, realice el proceso de evaluación en un paso. La definición del tipo `Arbol` es:

```
data Arbol = Void
           | Mkt Token [Arbol] deriving (Eq,Show)
```

   La función `smdin` se usará durante el proceso de verificación para garantizar el progreso por lo que necesitamos que su resultado nos indique si un programa se bloqueará u obtendrá una nueva expresión (progresará) para lo cual el tipo `Maybe` de `Haskell` sirve a la perfección.

   Notemos además que el resultado final de evaluación debe ser otro árbol, debido a la semántica de paso pequeño. Los valores son árboles también, pues su evaluación produce el mismo resultado.

***Ejemplo 2.7***

En nuestro ejemplo, la semántica dinámica se define como:

```
smdin :: Arbol -> Maybe Arbol
smdin expr@(Mkt (Tk "True") h)  = Just expr
smdin expr@(Mkt (Tk "False") h) = Just expr
smdin expr@(Mkt (Tk "Zero") h)  = Just expr
smdin (Mkt (Tk "Pred") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "Zero") l)
smdin (Mkt (Tk "Pred") ((Mkt (Tk "Suc") (h1:hs)):xs)) = Just h1
smdin (Mkt (Tk "Pred") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Pred") (j:xs))
      Nothing -> Nothing
```

```
smdin (Mkt (Tk "Suc") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Suc") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "IsZero") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "True") [])
smdin (Mkt (Tk "IsZero") ((Mkt (Tk "Suc") _):xs)) =
   Just (Mkt (Tk "False") [])
smdin (Mkt (Tk "IsZero") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "IsZero") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "If") [(Mkt (Tk "True") l),t1,t2]) = Just t1
smdin (Mkt (Tk "If") [(Mkt (Tk "False") l),t1,t2]) = Just t2
smdin (Mkt (Tk "If") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "If") (j:xs))
      Nothing -> Nothing
```

4. Una función `smest` que implemente la semántica estática para realizar la verificación de tipos del lenguaje.

   Para especificar la semántica estática se debe proporcionar la función `smest` que dado un árbol de sintaxis abstracta (`Arbol`) verifica los tipos del mismo.

   Esta función se usará durante el proceso de verificación para garantizar la preservación, por lo que necesitamos que, al igual que la función anterior, ello nos indique si el programa obtiene un tipo (preservará) o un error. Para ello nuevamente usamos el tipo `Maybe`.

```
smest :: Arbol → Maybe Tipo
```

   Notemos además que el resultado final es un tipo definido por `Tipo` que incluye los mismos tipos definidos en la función `stipos`.

***Ejemplo 2.8***

En nuestro ejemplo, la semántica estática se define como:

```
smest :: Arbol -> Maybe Tipo
smest (Mkt (Tk "True") h) = Just (Conc "Bool")
smest (Mkt (Tk "False") h) = Just (Conc "Bool")
smest (Mkt (Tk "Zero") h) = Just (Conc "Nat")
smest (Mkt (Tk "Pred") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
```

```
        Just (Conc "Nat")
    else
        Nothing
smest (Mkt (Tk "Suc") (h1:xs)) =
    if Just (Conc "Nat") == (smest h1) then
        Just (Conc "Nat")
    else
        Nothing
smest (Mkt (Tk "IsZero") (h1:xs)) =
    if Just (Conc "Nat") == (smest h1) then
        Just (Conc "Bool")
    else
        Nothing
smest (Mkt (Tk "If") [b,t1,t2]) =
    let t = (smest t1) in
        (if smest b == Just (Conc "Bool") && (smest t2) == t then
            t
        else
            Nothing)
```

Es importante mencionar que la versión del sistema desarrollado, reconoce lenguajes de programación sencillos, mismos que no incluyen, definición de funciones u operadores recursivos o de iteración. Esto de desarrolla a detalle en el Capítulo de conclusiones.

## 2.4. Generación del espacio de búsqueda

La generación del espacio de búsqueda se da por los siguientes tres pasos:

1. Generación de las restricciones
2. Procesamiento de las restricciones por medio de MiniSAT
3. Traducción de los resultados dados por el solucionador SAT

### 2.4.1. Generación de las restricciones

El proceso de generación de restricciones es quizá el más importante de todo el sistema, pues consiste en construir una fórmula proposicional que represente la noción de *programa correcto* a partir de la especificación dada por el usuario. El diseño de esta fórmula está dividido en seis restricciones que trabajan en conjunto para construir el espacio de búsqueda final.

El objetivo es construir programas por medio de los *tokens* dados por el usuario, respetando la altura de los mismos y las reglas sintácticas como el número de argumentos y su tipo. Por ejemplo, si la altura fuera 2 y el número máximo de argumentos de los programas fuera 3, los programas tendrían la forma dada por la Figura 2.3.

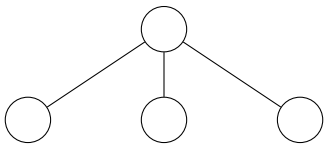


Figura 2.3: Esqueleto de altura 2 y número máximo de argumentos 3.

Llamamos a estos árboles vacíos *esqueleto*. El proceso inicial consiste en construir los mismos. Se requiere construir una fórmula que será proporcionada al solucionador SAT en el siguiente paso, para lo cual necesitamos proposiciones. Nuestras proposiciones serán denotadas por:

$$p_{x,y}$$

que denotan que *el nodo* $x$ *tiene asignado el token* $y$. Además se tienen algunos conjuntos especiales que ayudan en la especificación, tales como:

- *Nodos* que contiene los índices de todos los nodos posibles.

$$Nodos = \{0, 1, \dots\}$$

- *Tokens* que contiene los índices de todos los *tokens* posibles.

$$Tokens = \{0, 1, \dots\}$$

- *Hojas* que contiene los índices de todos los nodos hoja del esqueleto.

$$Hojas = \{x \ : \ x \text{ es el índice de alguna hoja del esqueleto}\}$$

- *Zero* que contiene los índices de todos los *tokens* de aridad cero (terminales) y vacíos.

$$Zero = \{x \ : \ x \text{ es el índice de algún token terminal o vacío}\}$$

- *Internos* que contiene los índices de todos los nodos internos del esqueleto

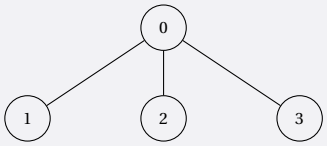
$$Internos = \{x \ : \ x \text{ es el índice de algún nodo interno del esqueleto}\}$$

Se proveen además algunas funciones y constantes auxiliares:

- *vacios*$(x,t)$ función que genera el índice de los *tokens* que deben ser vacíos de acuerdo con un nodo padre $x$ asignado con el *token* $t$.
- *noVacios*$(x,t)$ función que genera el índice de los *tokens* que no deben ser vacíos de acuerdo a un nodo padre $x$ asignado con el *token* $t$.
- *tipoArgs*$(t)$ función que devuelve los tipos que deben tener los argumentos de un *token*.
- *tipoDist*$(t)$ función que devuelve los tipos diferentes a los que deben tener los argumentos de un *token*.
- *tokenTipo*$(t)$ función que devuelve todos los *tokens* de un tipo $t$.
- $v$ constante que representa el índice del *token* vacío.

### *Ejemplo 2.9*

Para nuestro lenguaje, tenemos el siguiente esqueleto con los siguientes índices:




y el siguiente indexado de los tokens:

```
0 -> Tk "True"
1 -> Tk "False"
2 -> Tk "Zero"
3 -> Tk "Suc"
4 -> Tk "Pred"
5 -> Tk "IsZero"
6 -> Tk "If"
7 -> Tk "Vacio"
```

En el código anterior se observa la adición de un nodo adicional llamado “vacío” para representar los casos donde el nodo no requiere token.

de donde:

- *Nodos* $= \{0,1,2,3\}$
- *Tokens* $= \{0,1,2,3,4,5,6,7\}$
- *Hojas* $= \{1,2,3\}$

- $Zero = \{0, 1, 2, 7\}$
- $Internos = \{0\}$

Usaremos estos conjuntos para ejemplificar cada una de las restricciones.

**Restricción 1:** ***Cada nodo tiene asignado a lo más un token*** El objetivo de esta restricción es indicar que en cada uno de los nodos del esqueleto se debe colocar un token con la restricción de que sólo puede haber un token a la vez en cada nodo. La fórmula asociada a esta restricción se muestra a continuación:

$$\bigwedge_{i \in Nodos} \left( ExactlyOne\{p_{i,j} : j \in Tokens\} \right)$$

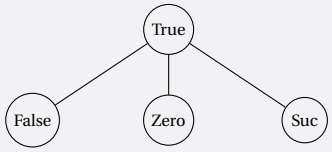
```
restriccion1 :: Formula String
```

### *Ejemplo 2.10*

En nuestro ejemplo, la restricción 1 genera la siguiente salida:

```
Formulas> restriccion1
ExactlyOne [Var "0 0",Var "0 1",Var "0 2",Var "0 3",Var "0 4",
            Var "0 5",Var "0 6",Var "0 7"] :&&:
ExactlyOne [Var "1 0",Var "1 1",Var "1 2",Var "1 3",Var "1 4",
            Var "1 5",Var "1 6",Var "1 7"] :&&:
ExactlyOne [Var "2 0",Var "2 1",Var "2 2",Var "2 3",Var "2 4",
            Var "2 5",Var "2 6",Var "2 7"] :&&:
ExactlyOne [Var "3 0",Var "3 1",Var "3 2",Var "3 3",Var "3 4",
            Var "3 5",Var "3 6",Var "3 7"]
```

Por ejemplo, el siguiente esqueleto es válido de acuerdo a la restricción 1. Notemos que el programa generado por esta restricción no tiene sentido hasta este punto.



**Restricción 2: *Las hojas sólo pueden ser tokens terminales o vacíos*** En un programa, las hojas de un árbol de sintaxis abstracta representan los símbolos terminales del lenguaje. Estos *tokens* terminales en nuestro caso normalmente son los valores como números, constantes booleanas, caracteres, etc. De la misma manera, debido a la forma del esqueleto y a la existencia de *tokens* que no reciben argumentos, existe la posibilidad de contar con programas cuyas hojas sean vacías. La fórmula asociada a esta restricción se muestra a continuación:

$$\bigwedge_{i \in Hojas} \left( ExactlyOne\{p_{i,j} : j \in Zero\} \right)$$

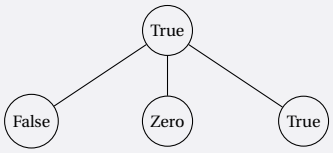
```
restriccion2 :: Formula String
```

### *Ejemplo 2.11*

En nuestro ejemplo, la restricción 2 genera la siguiente salida:

```
Formulas> restriccion2
ExactlyOne [Var "1 0",Var "1 1",Var "1 2",Var "1 7"] :&&:
ExactlyOne [Var "2 0",Var "2 1",Var "2 2",Var "2 7"] :&&:
ExactlyOne [Var "3 0",Var "3 1",Var "3 2",Var "3 7"]
```

Por ejemplo, el siguiente esqueleto es válido de acuerdo con las restricciones 1 y 2. Notemos que el programa generado por esta restricción tampoco tiene sentido hasta este punto.




**Restricción 3: *Cada token se aplica con el número de argumentos correcto*** Esta restricción garantiza que cada *token* se aplique con el número adecuado de argumentos. Por ejemplo si hubiera un constructor de suma, garantizaría que se aplica exactamente a dos expresiones. Si el número máximo de hijos del esqueleto es mayor a la aridad de cada *token*, se rellena con vacíos. Esta restricción sólo se aplicara a los nodos internos. La fórmula asociada a esta restricción se muestra a continuación:

$$\bigwedge_{x \in Internos} \left( \bigwedge_{t \in Tokens} p_{x,t} \rightarrow \left( \bigwedge_{y \in vacios(x,t)} p_{y,v} \right) \wedge \left( \bigwedge_{y \in noVacios(x,t)} \neg p_{y,v} \right) \right)$$

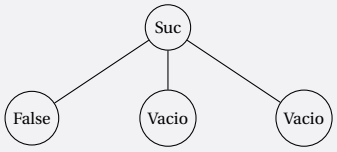
```
restriccion3 :: Formula String
```

### *Ejemplo 2.12*

En nuestro ejemplo, la restricción 3 genera la siguiente salida:

```
Formulas> restriccion3
(Var "0 0" :->: Var "1 7" :&&: Var "2 7" :&&: Var "3 7" :&&:
                Yes :&&: Yes) :&&:
(Var "0 1" :->: Var "1 7" :&&: Var "2 7" :&&: Var "3 7" :&&:
                Yes :&&: Yes) :&&:
(Var "0 2" :->: Var "1 7" :&&: Var "2 7" :&&: Var "3 7" :&&:
                Yes :&&: Yes) :&&:
(Var "0 3" :->: Var "2 7" :&&: Var "3 7" :&&: Yes :&&:
                Not (Var "1 7") :&&: Yes) :&&:
(Var "0 4" :->: Var "2 7" :&&: Var "3 7" :&&: Yes :&&:
                Not (Var "1 7") :&&: Yes) :&&:
(Var "0 5" :->: Var "2 7" :&&: Var "3 7" :&&: Yes :&&:
                Not (Var "1 7") :&&: Yes) :&&:
(Var "0 6" :->: Yes :&&: Not (Var "1 7") :&&: Not (Var "2 7") :&&:
                Not (Var "3 7") :&&: Yes)
```

Por ejemplo, el siguiente esqueleto es válido de acuerdo con las restricciones 1, 2 y 3. Notemos que el programa generado por esta restricción continúa sin tener sentido.




**Restricción 4:** ***Cada token se aplica con los argumentos del tipo correcto*** Una vez que la restricción 3 detecta cuántos hijos no vacíos tiene cada nodo dependiendo del *token* de su padre, la restricción 4 se encarga de buscar *tokens* del tipo adecuado en cada uno de los casos. Por ejemplo, si hubiera un constructor suma, la restricción 3 garantiza que tiene dos hijos no vacíos, mientras que la restricción 4 garantiza que los dos hijos deben ser numéricos,

asignar algo de otro tipo generaría un programa incorrecto. Esta restricción depende en gran medida del sistema de tipos. La fórmula asociada a esta restricción se muestra a continuación:

$$\bigwedge_{i \in Internos} \left( \bigwedge_{j \in Tokens} p_{i,j} \rightarrow \left( \bigwedge_{x \in noVacios(i,j)} AtMostOne\{p_{x,y} : y \in tokenTipo(tipoArgs(j))\} \wedge \bigwedge_{x \in noVacios(i,j)} None\{p_{x,y} : y \in tipoDist(tipoArgs(j))\} \right) \right)$$

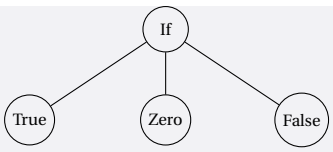
```
restriccion4 :: Formula String
```

### *Ejemplo 2.13*

En nuestro ejemplo, la restricción 4 genera la siguiente salida:

```
Formulas> restriccion4
(Var "0 0" :->: Yes :&&: Yes) :&&:
(Var "0 1" :->: Yes :&&: Yes) :&&:
(Var "0 2" :->: Yes :&&: Yes) :&&:
(Var "0 3" :->:
   AtMostOne [Var "1 2",Var "1 3",Var "1 4",Var "1 6"] :&&: Yes
   :&&: None [Var "1 0",Var "1 1",Var "1 5"] :&&: Yes) :&&:
(Var "0 4" :->:
   AtMostOne [Var "1 2",Var "1 3",Var "1 4",Var "1 6"] :&&: Yes
   :&&: None [Var "1 0",Var "1 1",Var "1 5"] :&&: Yes) :&&:
(Var "0 5" :->:
   AtMostOne [Var "1 2",Var "1 3",Var "1 4",Var "1 6"] :&&: Yes
   :&&: None [Var "1 0",Var "1 1",Var "1 5"] :&&: Yes) :&&:
(Var "0 6" :->:
   AtMostOne [Var "1 0",Var "1 1",Var "1 5",Var "1 6"]
   :&&: AtMostOne [] :&&: AtMostOne [] :&&: Yes :&&:
   None [Var "1 2",Var "1 3",Var "1 4"] :&&: None []
   :&&: None [] :&&: Yes)
```

Por ejemplo, el siguiente esqueleto es válido de acuerdo a las restricciones 1, 2, 3 y 4. Notemos que el programa generado por esta restricción continúa sin tener sentido con respecto a los tipos.




**Restricción 5:** ***Tokens con argumentos que involucran variables de tipo*** Esta restricción se encarga de procesar aquellos *tokens* que involucran variables de tipo, de forma tal que se realicen combinaciones dependiendo de los valores que puede generar cada uno de los argumentos. La fórmula asociada a esta restricción se muestra a continuación:

$$\bigwedge_{x \in Intetrnos} \left( \bigwedge_{y \in Vtipo} p_{x,y} \rightarrow ExactlyOne\left\{ AtMostOne\{p_{i,j} : i \in noVacios(x,y), j \in vTipos(y)\}, \right. \right.$$
$$\left. \left. None\{p_{i,j} : i \in noVacios(x,y), j \in tipoDist(vTipos(j))\} \right\} \right)$$

```
restriccion5 :: Formula String
```

### *Ejemplo 2.14*

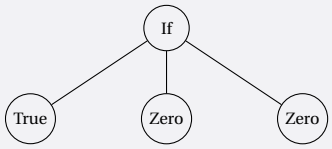
En nuestro ejemplo, la restricción 5 genera la siguiente salida:

```
Formulas> restriccion5
(Var "0 6" :->:
   (AtMostOne [Var "2 0",Var "2 1",Var "2 5",Var "2 6"] :&&:
    AtMostOne [Var "3 0",Var "3 1",Var "3 5",Var "3 6"] :&&:
    Yes :&&:
    None [Var "2 2",Var "2 3",Var "2 4",Var "3 2",Var "3 3",
          Var "3 4"]) :||:
   (AtMostOne [Var "2 2",Var "2 3",Var "2 4",Var "2 6"] :&&:
    AtMostOne [Var "3 2",Var "3 3",Var "3 4",Var "3 6"] :&&:
    Yes :&&:
    None [Var "2 0",Var "2 1",Var "2 5",Var "3 0",Var "3 1",
          Var "3 5"]) :||:
    No) :&&:
```

```
Yes
```

De forma intuitiva, la restricción 5 para este lenguaje garantiza que las ramas *then* y *else* del *if* sean del mismo tipo.

Por ejemplo, el siguiente esqueleto es válido de acuerdo con las restricciones 1, 2, 3 , 4 y 5. Notemos que el programa generado por esta restricción ya es un programa válido.




### 2.4.2. Procesamiento de las restricciones

Dadas las restricciones anteriores, el proceso siguiente consiste en usar la función `solve_all` de la biblioteca `MiniSAT` de HASKELL para obtener todos los programas válidos bajo dichas restricciones.

***Ejemplo 2.15***

La llamada a `solve_all` en nuestro ejemplo se da mediante la función `espacioCompleto` donde `formula` es la conjunción de las 5 restricciones:

```
espacioCompleto :: Formula String -> [Map String Bool]
espacioCompleto formula = solve_all formula
```

y produce una lista con la siguiente estructura:

```
Verificador> espacioBusqueda formula
[fromList [("0 0",False),("0 1",False),("0 2",False),
           ("0 3",False),("0 4",False),("0 5",False),
           ("0 6",True),("0 7",False),
           ("1 0",True),("1 1",False),("1 2",False),
           ("1 3",False),("1 4",False),("1 5",False),
           ("1 6",False),("1 7",False),
           ("2 0",True),("2 1",False),("2 2",False),
           ("2 3",False),("2 4",False),("2 5",False),
           ("2 6",False),("2 7",False),
           ("3 0",True),("3 1",False),("3 2",False),
```

```
            ("3 3",False),("3 4",False),("3 5",False),
            ("3 6",False),("3 7",False)], ...]
```

que es la asignación de estados que indica cómo asignar los tokens a cada nodo del árbol.

### 2.4.3. Traducción de los resultados

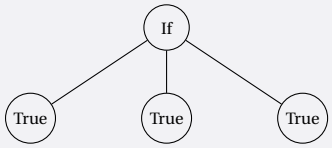
Una vez obtenida la lista de estados que hacen verdadera la fórmula dada por nuestras cinco restricciones, debemos darle forma a cada uno de los estados del espacio de búsqueda, es decir, reconstruir los programas a partir del esqueleto inicial.

***Ejemplo 2.16***

La primera asignación de estados de nuestro espacio de búsqueda es:

```
[("0 0",False),("0 1",False),("0 2",False),("0 3",False),
 ("0 4",False),("0 5",False),("0 6",True),("0 7",False),
 ("1 0",True),("1 1",False),("1 2",False),("1 3",False),
 ("1 4",False),("1 5",False),("1 6",False),("1 7",False),
 ("2 0",True),("2 1",False),("2 2",False),("2 3",False),
 ("2 4",False),("2 5",False),("2 6",False),("2 7",False),
 ("3 0",True),("3 1",False),("3 2",False),("3 3",False),
 ("3 4",False),("3 5",False),("3 6",False),("3 7",False)]
```

que corresponde al siguiente árbol, notando que únicamente llenamos los nodos con aquellas tuplas cuyo valor de verdad sea `True`:




Esta construcción se realiza de forma automática por medio de la función `espacioBusqueda`.

```
Verificador> espacioBusqueda (espacioB formula)
[Mkt If [Mkt True [],Mkt True [],Mkt True []], ...]
```

El constructor `Mkt` permite construir árboles $n$-arios.

## 2.5. Verificación y poda

La verificación y poda del espacio de búsqueda se da por los siguientes pasos:

1. Tomar un estado del espacio de búsqueda y verificar preservación y progreso. Si falla la verificación, imprimir el estado (programa) como contraejemplo.
2. Encontrar los programas similares al programa elegido en el paso anterior y con apoyo del solucionador SAT quitarlos del espacio de búsqueda.
3. Repetir el proceso hasta que no haya más estados en el espacio de búsqueda.

### 2.5.1. Preservación y progreso

La verificación de las propiedades de preservación y progreso viene dada por la especificación de las semánticas estática y dinámica del lenguaje de programación diseñado que, como se explicó con anterioridad, está dada por las funciones `smdin` y `smest`.

Los pasos que se siguen para realizar la verificación completa son:

1. Para verificar la propiedad de progreso es necesario apoyarse de la semántica dinámica al tomar un programa y realizar un paso en la ejecución del mismo, la propiedad se cumple si se obtiene un nuevo programa (estado) y éste no se bloquea. Es importante notar que para el caso de los valores, siempre se obtiene como salida el mismo árbol, por lo que no se considera como estado bloqueado.
2. Para verificar la propiedad de preservación es necesario apoyarse en la semántica estática al tomar el tipo de un programa y después de verificar progreso corroborar que el tipo del programa resultante se mantiene.
3. En caso de que alguna de estas dos propiedades falle, se emite el programa correspondiente como contraejemplo.

```
progreso :: Arbol -> Bool
progreso a =
   case (smdin a) of
        -- Si se obtiene un nuevo árbol, se procede con preservación.
      Just j -> preservacion j a (smest a)
        -- Si no se obtiene un nuevo árbol (error) se emite un
           contraejemplo.
```

```
      Nothing -> error ("Contraejemplo: " ++ show a)

preservacion :: Arbol -> Arbol -> Maybe Tipo -> Bool
preservacion a o (Just t) =
   case (smest a) of
        -- Si se obtiene un tipo y corresponde con el original,
        -- se reporta true
      Just j -> if j == t then
                  True
                else  -- En caso contrario, contraejemplo.
                  error ("Contraejemplo: " ++ show o)
        -- Si no se obtiene un tipo válido, contraejemplo.
      Nothing -> error ("Contraejemplo: " ++ show o)
```

A partir de estas dos propiedades se implementaron dos funciones verificadoras:

- La primera realiza el proceso de verificación Glass Box sin el proceso de poda, es decir, se verifican todos los programas (estados) contenidos en el espacio de búsqueda.
- La segunda realiza el proceso de verificación Glass Box usando el proceso de poda, es decir, no se verifican todos los programas (estados) contenidos en el espacio de búsqueda pues después de la verificación de un programa *s*, se eliminan los programas similares a *s* para evitar repetir la verificación.

### 2.5.2. Verificación sin poda

La verificación sin poda consiste en tomar el espacio de búsqueda (`[Arbol]`) y elegir el primer elemento del mismo para verificar la propiedad de progreso que a su vez llama a la de preservación con dicho programa. Posteriormente, repite el proceso recursivamente hasta consumir todos los programas contenidos en el espacio de búsqueda.

```
verificaSinPoda :: [Arbol] -> Bool
verificaSinPoda [] = True
verificaSinPoda (x:xs) = progreso x && verificaSinPoda xs
```

***Ejemplo 2.17***

Para nuestro lenguaje de prueba EAB la salida del verificador sin poda es la siguiente:

```
*Verificador> verificaSinPoda (espacioBusqueda (espacioB formula))
True
```

La salida `True` indica que el lenguaje de programación dado en la especificación cumple con las propiedades de preservación y progreso y por lo tanto es seguro.

### 2.5.3. Verificación con poda

La verificación con poda consiste en tomar el espacio de búsqueda (`[Arbol]`) y elegir el primer elemento del mismo para verificar la propiedad de progreso que a su vez llama a la de preservación con dicho programa. Si dicho programa cumple con la propiedad de preservación y progreso, entonces se eliminarán (con ayuda del solucionador SAT) todos aquellos programas que se consideren similares, para reducir el espacio de búsqueda y el proceso se repite recursivamente hasta consumir todos los programas contenidos en dicho espacio de búsqueda.

```
verificaPoda :: [Arbol] -> Formula String -> Bool
verificaPoda [] f = True
verificaPoda (x:xs) f =
   progreso x && (verificaPoda (espacioBusqueda (espacioB actualizada))
                               actualizada)
      where actualizada = f :&&: (similares x)
```

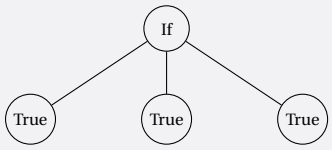
Observar que a diferencia del verificador sin poda, este verificador carga con la fórmula asociada al espacio de búsqueda en todo momento para modificarla durante el proceso de verificación con el fin de realizar la poda.

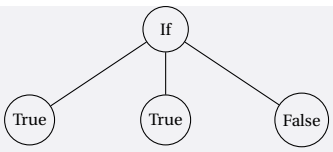
Para los fines de la metodología Glass Box, se consideran dos programas similares cuando:

- tengan la misma raíz y
- el primero de sus argumentos sea el mismo.

***Ejemplo 2.18***

Los siguientes dos programas son similares:






```
similares :: Arbol -> Formula String
similares (Mkt t ((Mkt (Tk "Vacio") _):xs)) = Not (Var ("0 " ++ (s t)))
similares t = (Var ("0 "++f1)) :->: Not (Var ("1 "++f2))
   where f1 = s (nodosIniciales t!!0)
         f2 = s (nodosIniciales t!!1)
```

La detección de estados similares se realiza añadiendo una restricción a la fórmula asociada al espacio de búsqueda. En esta restricción la fórmula plasma que los árboles deben tener una raíz y primer argumento diferentes a los del programa revisado con anterioridad.

### *Ejemplo 2.19*

Para nuestro lenguaje de prueba EAB la salida del verificador con poda es la siguiente:

```
*Verificador> verificador2 (espacioBusqueda (espacioB formula))
   formula
True
```

La salida `True` indica que el lenguaje de programación dado en la especificación cumple con las propiedades de preservación y progreso y por lo tanto es seguro.

El siguiente capítulo ilustra pruebas que se hicieron con distintos lenguajes de programación, su salida y comparaciones en tiempo.

# Capítulo 3

# Casos de estudio

En este capítulo se presentan algunos casos de estudio que se realizaron usando el sistema implementado para verificar la seguridad del sistema de tipos de algunos lenguajes de programación. En todos los casos se muestran los siguientes elementos:

- sintaxis concreta
- semántica dinámica
- semántica estática
- interpretación de los resultados

## 3.1. EAB

El lenguaje de programación *EAB* como vimos en los capítulos anteriores, permite realizar operaciones aritméticas y booleanas. De la misma forma, como se estudió en los capítulos anteriores, este lenguaje cumple con las propiedades de preservación y progreso por lo que cumple con la propiedad de seguridad. En esta sección explicamos los detalles de su especificación así como los resultados obtenidos con el mismo. A continuación su especificación y resultados.

La prueba se realizó usando árboles de altura 2 con el mismo esqueleto que la el ejemplo tratado en el capítulo anterior.

### 3.1.1. Sintaxis concreta

Gramática del lenguaje:

```
<expr> ::= 0
         | true
         | false
         | suc(<expr>)
         | pred(<expr>)
         | iszero(<expr>)
         | if <expr> then <expr> else <expr>
```

Representación de tokens en el sistema implementado:

```
[Tk "True", Tk "False", Tk "Zero", Tk "Suc", Tk "Pred", Tk "IsZero",
 Tk "If"]
```

### 3.1.2. Semántica dinámica

Formalización:

$$\frac{n \to n'}{Suc(n) \to Suc(n')} \qquad \frac{}{IsZero(Zero()) \to True()}$$

$$\frac{n \to n'}{Pred(n) \to Pred(n')} \qquad \frac{}{IsZero(Suc(n)) \to False()}$$

$$\frac{}{Pred(Suc(n)) \to n} \qquad \frac{c \to c'}{If(c,t,e) \to If(c',t,e)}$$

$$\frac{}{Pred(Zero()) \to Zero()} \qquad \frac{}{If(True(),t,e) \to t}$$

$$\frac{n \to n'}{IsZero(n) \to IsZero(n')} \qquad \frac{}{If(False(),t,e) \to e}$$

Representación de las reglas en el sistema implementado:

```
smdin :: Arbol -> Maybe Arbol
smdin expr@(Mkt (Tk "True") h)  = Just expr
smdin expr@(Mkt (Tk "False") h) = Just expr
smdin expr@(Mkt (Tk "Zero") h)  = Just expr
smdin (Mkt (Tk "Pred") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "Zero") l)
```

```
smdin (Mkt (Tk "Pred") ((Mkt (Tk "Suc") (h1:hs)):xs)) = Just h1
smdin (Mkt (Tk "Pred") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Pred") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "Suc") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Suc") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "IsZero") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "True") [])
smdin (Mkt (Tk "IsZero") ((Mkt (Tk "Suc") _):xs)) =
   Just (Mkt (Tk "False") [])
smdin (Mkt (Tk "IsZero") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "IsZero") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "If") [(Mkt (Tk "True") l),t1,t2]) = Just t1
smdin (Mkt (Tk "If") [(Mkt (Tk "False") l),t1,t2]) = Just t2
smdin (Mkt (Tk "If") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "If") (j:xs))
      Nothing -> Nothing
```

### 3.1.3. Semántica estática

Formalización:

$$\frac{}{False() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{Pred(n) : \mathsf{Nat}}$$

$$\frac{}{True() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{IsZero(n) : \mathsf{Bool}}$$

$$\frac{}{Zero() : \mathsf{Nat}}$$

$$\frac{n : \mathsf{Nat}}{Suc(n) : \mathsf{Nat}} \qquad \frac{c : \mathsf{Bool} \quad t : \tau \quad e : \tau}{If(c,t,e) : \tau}$$

Representación de las reglas en el sistema implementado:

```
smest :: Arbol -> Maybe Tipo
smest (Mkt (Tk "True") h) = Just (Conc "Bool")
smest (Mkt (Tk "False") h) = Just (Conc "Bool")
smest (Mkt (Tk "Zero") h) = Just (Conc "Nat")
smest (Mkt (Tk "Pred") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Nat")
   else
      Nothing
smest (Mkt (Tk "Suc") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Nat")
   else
      Nothing
smest (Mkt (Tk "IsZero") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Bool")
   else
      Nothing
smest (Mkt (Tk "If") [b,t1,t2]) =
   let t = (smest t1) in
      (if smest b == Just (Conc "Bool") && (smest t2) == t then
         t
      else
         Nothing)
```

### 3.1.4. Interpretación de resultados

La salida del verificador arroja los siguientes resultados:

```
*Verificador> verificador1 (espacioBusqueda (espacioB formula))
True
```

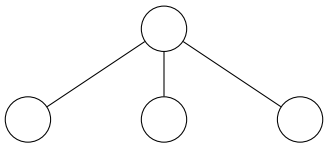
Este resultado indica que, bajo la especificación dada, el sistema de tipos de este lenguaje es seguro, es decir, cumple tanto la propiedad de progreso (ningún programa se bloquea) y preservación (si una expresión tiene un tipo, entonces en un paso, el programa resultante tendrá el mismo tipo).

## 3.2. EAB 2: Un lenguaje sin preservación de tipos

A diferencia del lenguaje EAB original, este lenguaje que incluye los mismos constructores que la versión original del mismo, se fuerza una falla sobre la preservación de tipos. La forma de lograr esto es haciendo que la evaluación de

las expresiones *true* y *false* sean 1 y 0 respectivamente. Esto es algo común en algunos lenguajes de programación, como es el caso de C, por ejemplo.

La prueba se realizó usando árboles de altura 2, con el siguiente esqueleto:



### 3.2.1. Sintaxis concreta

La sintaxis de este lenguaje es idéntica a la del lenguaje EAB original.

```
<expr> ::= 0
         | true
         | false
         | suc(<expr>)
         | pred(<expr>)
         | iszero(<expr>)
         | if <expr> then <expr> else <expr>
```

Representación de tokens en el sistema implementado:

```
[Tk "True", Tk "False", Tk "Zero", Tk "Suc", Tk "Pred", Tk "IsZero",
 Tk "If"]
```

### 3.2.2. Semántica dinámica

Formalización:

Observar las primeras dos reglas de la columna izquierda donde se muestra que los valores booleanos terminan siendo números. De aquí que la propiedad de preservación no se cumpla.

$$\frac{}{True\,() \rightarrow Suc\ (Zero\ ())} \qquad \frac{n \rightarrow n'}{Pred\,(n) \rightarrow Pred\,(n')}$$

$$\frac{}{False\,() \rightarrow Zero\,()}$$

$$\frac{n \rightarrow n'}{Suc\,(n) \rightarrow Suc\,(n')} \qquad \frac{}{Pred\,(Suc\,(n)) \rightarrow n}$$

$$\frac{}{Pred\,(Zero\,())\longrightarrow Zero\,()} \qquad \frac{c\longrightarrow c'}{If(c,t,e)\longrightarrow If(c',t,e)}$$

$$\frac{n\longrightarrow n'}{IsZero\,(n)\longrightarrow IsZero\,(n')}$$

$$\frac{}{If(True\,(),t,e)\longrightarrow t}$$

$$\frac{}{IsZero\,(Zero\,())\longrightarrow True\,()}$$

$$\frac{}{IsZero\,(Suc\,(n))\longrightarrow False\,()} \qquad \frac{}{If(False\,(),t,e)\longrightarrow e}$$

Representación de las reglas en el sistema implementado:

```
smdin :: Arbol -> Maybe Arbol
smdin expr@(Mkt (Tk "True") h)  = Just (Mkt (Tk "Suc") [(Mkt (Tk "Zero")
    [])])
smdin expr@(Mkt (Tk "False") h) = Just (Mkt (Tk "Zero") [])
smdin expr@(Mkt (Tk "Zero") h)  = Just expr
smdin (Mkt (Tk "Pred") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "Zero") l)
smdin (Mkt (Tk "Pred") ((Mkt (Tk "Suc") (h1:hs)):xs)) = Just h1
smdin (Mkt (Tk "Pred") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Pred") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "Suc") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Suc") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "IsZero") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "True") [])
smdin (Mkt (Tk "IsZero") ((Mkt (Tk "Suc") _):xs)) =
   Just (Mkt (Tk "False") [])
smdin (Mkt (Tk "IsZero") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "IsZero") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "If") [(Mkt (Tk "True") l),t1,t2]) = Just t1
smdin (Mkt (Tk "If") [(Mkt (Tk "False") l),t1,t2]) = Just t2
smdin (Mkt (Tk "If") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "If") (j:xs))
      Nothing -> Nothing
```

### 3.2.3. Semántica estática

El sistema de tipos de esta versión de EAB es idéntico al del lenguaje original.

Formalización:

$$\frac{}{False() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{Pred(n) : \mathsf{Nat}}$$

$$\frac{}{True() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{IsZero(n) : \mathsf{Bool}}$$

$$\frac{}{Zero() : \mathsf{Nat}}$$

$$\frac{n : \mathsf{Nat}}{Suc(n) : \mathsf{Nat}} \qquad \frac{c : \mathsf{Bool} \quad t : \tau \quad e : \tau}{If(c,t,e) : \tau}$$

```
smest :: Arbol -> Maybe Tipo
smest (Mkt (Tk "True") h) = Just (Conc "Bool")
smest (Mkt (Tk "False") h) = Just (Conc "Bool")
smest (Mkt (Tk "Zero") h) = Just (Conc "Nat")
smest (Mkt (Tk "Pred") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Nat")
   else
      Nothing
smest (Mkt (Tk "Suc") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Nat")
   else
      Nothing
smest (Mkt (Tk "IsZero") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Bool")
   else
      Nothing
smest (Mkt (Tk "If") [b,t1,t2]) =
   let t = (smest t1) in
      (if smest b == Just (Conc "Bool") && (smest t2) == t then
         t
      else
         Nothing)
```

### 3.2.4. Interpretación de resultados

El verificador genera los siguientes resultados:

```
verificador1 (espacioBusqueda (espacioB formula))
*** Exception:
Contraejemplo: Mkt False [Mkt Vacio [],Mkt Vacio [],Mkt Vacio []]
CallStack (from HasCallStack):
error, called at Verificador.hs:57:56
in main:Verificador
```

Este resultado indica que, bajo la especificación dada, el sistema de tipos de este lenguaje no es seguro. En este caso se da el siguiente programa como contraejemplo:

```
Mkt False [Mkt Vacio [],Mkt Vacio [],Mkt Vacio []]
```

Que corresponde con la expresión `false`.

En este caso el programa no cumple la propiedad de preservación pues el programa inicialmente tiene el tipo Bool mientras que al dar un paso en la ejecución se obtiene como resultado sucesor de cero que tiene como tipo Nat.

## 3.3. EAB 3: Un lenguaje que no cumple progreso

### 3.3.1. Sintaxis concreta

Nuevamente la sintaxis es idéntica a la del lenguaje EAB original.

```
<expr> ::= 0
         | true
         | false
         | suc(<expr>)
         | pred(<expr>)
         | iszero(<expr>)
         | if <expr> then <expr> else <expr>
```

Representación de tokens en el sistema implementado:

```
[Tk "True", Tk "False", Tk "Zero", Tk "Suc", Tk "Pred", Tk "IsZero",
 Tk "If"]
```

### 3.3.2. Semántica dinámica

Formalización

En este caso, para que el lenguaje no cumpla con la propiedad de progreso se omiten algunas de las reglas del lenguaje EAB original. La omisión de estas reglas ocasiona estados bloqueados en el proceso de ejecución.

$$\frac{n \rightarrow n'}{Suc(n) \rightarrow Suc(n')} \qquad \frac{}{IsZero(Suc(n)) \rightarrow False()}$$

$$\frac{n \rightarrow n'}{Pred(n) \rightarrow Pred(n')} \qquad \frac{c \rightarrow c'}{If(c,t,e) \rightarrow If(c',t,e)}$$

$$\frac{}{Pred(Suc(n)) \rightarrow n} \qquad \frac{}{If(True(),t,e) \rightarrow t}$$

$$\frac{}{Pred(Zero()) \rightarrow Zero()} \qquad \frac{}{If(False(),t,e) \rightarrow e}$$

$$\frac{}{IsZero(Zero()) \rightarrow True()}$$

Representación de las reglas en el sistema implementado:

```
smdin :: Arbol -> Maybe Arbol
smdin expr@(Mkt (Tk "True") h)  = Just (Mkt (Tk "Suc") [(Mkt (Tk "Zero")
    [])])
smdin expr@(Mkt (Tk "False") h) = Just (Mkt (Tk "Zero") [])
smdin expr@(Mkt (Tk "Zero") h)  = Just expr
smdin (Mkt (Tk "Pred") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "Zero") l)
smdin (Mkt (Tk "Pred") ((Mkt (Tk "Suc") (h1:hs)):xs)) = Just h1
smdin (Mkt (Tk "Pred") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Pred") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "Suc") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "Suc") (j:xs))
      Nothing -> Nothing
smdin (Mkt (Tk "IsZero") (Mkt (Tk "Zero") l:xs)) =
   Just (Mkt (Tk "True") [])
smdin (Mkt (Tk "IsZero") ((Mkt (Tk "Suc") _):xs)) =
   Just (Mkt (Tk "False") [])
smdin (Mkt (Tk "If") [(Mkt (Tk "True") l),t1,t2]) = Just t1
```

```
smdin (Mkt (Tk "If") [(Mkt (Tk "False") l),t1,t2]) = Just t2
smdin (Mkt (Tk "If") (h1:xs)) =
   case (smdin h1) of
      (Just j) -> Just (Mkt (Tk "If") (j:xs))
      Nothing -> Nothing
```

### 3.3.3. Semántica estática

El sistema de tipos se mantiene idéntico al de EAB.

Formalización:

$$\frac{}{False() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{Pred(n) : \mathsf{Nat}}$$

$$\frac{}{True() : \mathsf{Bool}} \qquad \frac{n : \mathsf{Nat}}{IsZero(n) : \mathsf{Bool}}$$

$$\frac{}{Zero() : \mathsf{Nat}}$$

$$\frac{n : \mathsf{Nat}}{Suc(n) : \mathsf{Nat}} \qquad \frac{c : \mathsf{Bool} \quad t : \tau \quad e : \tau}{If(c,t,e) : \tau}$$

Representación de las reglas en el sistema implementado:

```
smest :: Arbol -> Maybe Tipo
smest (Mkt (Tk "True") h) = Just (Conc "Bool")
smest (Mkt (Tk "False") h) = Just (Conc "Bool")
smest (Mkt (Tk "Zero") h) = Just (Conc "Nat")
smest (Mkt (Tk "Pred") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Nat")
   else
      Nothing
smest (Mkt (Tk "Suc") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Nat")
   else
      Nothing
smest (Mkt (Tk "IsZero") (h1:xs)) =
   if Just (Conc "Nat") == (smest h1) then
      Just (Conc "Bool")
```

```
    else
       Nothing
smest (Mkt (Tk "If") [b,t1,t2]) =
    if (smest b) == Just (Conc "Bool") && (smest t1) == (smest t2) then
       (smest t1)
    else
       Nothing
```

### 3.3.4. Interpretación de resultados

La salida del verificador arroja los siguientes resultados:

```
verificador1 (espacioBusqueda (espacioB formula))
True
```

Este resultado indica que, bajo la especificación dada, el sistema de tipos de este lenguaje es seguro, es decir, cumple tanto la propiedad de progreso (ningún programa se bloquea) y preservación (si una expresión tiene un tipo, entonces en un paso, el programa resultante tendrá el mismo tipo).

Este resultado podría parecer un falso positivo. Por ejemplo ¿qué sucede con el siguiente programa?

```
iszero(pred(zero))
```

El verificador no obtiene este programa como contraejemplo, debido a que el esqueleto formado es únicamente para programas de altura 2, con lo cual el programa antes mencionado no puede construirse. Para poder capturar esta situación procedemos a realizar una prueba con nivel 3.

```
altura :: Int
altura = 3
```

```
verificador1 (espacioBusqueda (espacioB formula))
*** Exception:
Contraejemplo: Mkt IsZero [Mkt Pred [Mkt Zero []]
CallStack (from HasCallStack):
error, called at Verificador.hs:57:56
in main:Verificador
```

En este caso se da el siguiente programa como contraejemplo:

```
Mkt IsZero [Mkt Pred [Mkt Zero []]
```

Que corresponde con la expresión `iszero(pred(0))`.

En este caso el programa no cumple la propiedad de progreso pues el programa no puede ser evaluado usando las reglas especificadas en la semántica dinámica.

Este ejemplo muestra la importancia de la altura en el problema. Una posible mejora al sistema podría ser automatizar la altura hasta encontrar el contraejemplo. Sin embargo, en el caso de lenguajes, cuyo sistema de tipos sea seguro, esta búsqueda podría nunca parar. Una posible propuesta es tomar una altura máxima y dar la opción al usuario de incrementarla, algo similar a los procesos de entrenamiento en redes neuronales.

## 3.4. NumString: Un lenguaje de cadenas y números

### 3.4.1. Sintaxis concreta

NumString es un lenguaje dotado de números y cadenas con operaciones básicas de multiplicación y concatenación. Ejemplos de expresiones del lenguaje son:

- `1729`
- `“foo”`
- `“foo” * 3`
- `3*2`
- `“foo” ++ “bar”`

```
<expr> ::= <num>
         | <string>
         | <expr> * <expr>
         | <string> ++ <string>


<num> ::= 1 | 2 | ...

<string> :: "a" | "hola" | ...
```

Representación de tokens en el sistema implementado:

```
[Tk "Num", Tk "Str", Tk "Mult", Tk "Conc"]
```

### 3.4.2. Semántica dinámica

La semántica de las expresiones de NumString es bastante simple: se pueden concatenar cadenas y multiplicar números. Sin embargo, se tiene un comportamiento especial que consiste en multiplicar cadenas por números. Por ejemplo:

```
"foo"*3
```

La evaluación de la expresión anterior consiste en concatenas 3 veces consigo misma la cadena ‘‘`foo`’’.

```
‘‘foofoofoo’’
```

Esto se describe con las reglas de la sintaxis abstracta *mult* a continuación.

Formalización

$$\frac{}{num(n) \rightarrow num(n)} \qquad \frac{d \rightarrow m}{mult(num(n), d) \rightarrow mult(num(n), m)}$$

$$\frac{}{string(s) \rightarrow string(s)} \qquad \frac{d \rightarrow m}{mult(string(s), d) \rightarrow mult(string(s), m)}$$

$$\frac{s \rightarrow s'}{concat(s, r) \rightarrow concat(s', r)} \qquad \frac{}{mult(string(s), num(n)) \rightarrow concat(string(s), mult(string(s), num(n-1)))}$$

$$\frac{r \rightarrow r'}{concat(string(s), r) \rightarrow concat(string(s), r')} \qquad \frac{}{mult(string(s), num(1)) \rightarrow string(s)}$$

$$\frac{}{concat(string(s), string(r)) \rightarrow string(sr)} \qquad \frac{}{mult(num(n), num(m)) \rightarrow Num(n \times m)}$$

$$\frac{i \rightarrow n}{mult(i, d) \rightarrow mult(n, d)}$$

Representación de las reglas en el sistema implementado:

```
smdin :: Arbol -> Maybe Arbol
smdin expr@(Mkt (Tk "Num") h)  = Just (Mkt (Tk "Num") h)
smdin expr@(Mkt (Tk "Str") h)  = Just (Mkt (Tk "Str") h)
smdin expr@(Mkt (Tk "Conc") [Mkt (Tk "Str") [],r] =
   let r' = smdin s in
      Just (Mkt (Tk "Conc") [Mkt (Tk "Str") [],r']
smdin expr@(Mkt (Tk "Conc") [Mkt (Tk "Str") [],Mkt (Tk "Str") []] =
```

```
      Just (Mkt (Tk "Str") []
smdin expr@(Mkt (Tk "Conc") [s,r])  =
   let s' = smdin s in
      Just (Mkt (Tk "Conc") [s',r]
smdin expr@(Mkt (TK "Mult") [Mkt (Tk "Num") [], d] =
   let m = smdin d in
      Just Just (Mkt (Tk "Mult") [Mkt (Tk "Str") [],m]
smdin expr@(Mkt (TK "Mult") [Mkt (Tk "Str") [], d] =
   let m = smdin d in
      Just Just (Mkt (Tk "Mult") [Mkt (Tk "Num") [],m]
smdin expr@(Mkt (TK "Mult") [Mkt (Tk "Str") [], Mkt (Tk "Num") []] =
      Just (Mkt (Tk "Cont") [Mkt (Tk "Str") [],Mkt (Tk "Num") []]
smdin expr@(Mkt (TK "Mult") [Mkt (Tk "Num") [], Mkt (Tk "Num") []] =
      Just (Mkt (Tk "Num") []
smdin expr@(Mkt (Tk "Mult") [i,d] =
   let n = smdin i in
      Just (Mkt (Tk "Mult") [n,d]
```

### 3.4.3. Semántica estática

El sistema de tipos de este lenguaje es inseguro justamente porque a partir de la semántica dinámica, podemos observar que la operación de multiplicación genera dos posibles valores: números o cadenas. Por otro lado, nuestra semántica estática establece que el tipo de una operación de multiplicación es Nat sin ninguna premisa adicional, con lo cual, no se cumple la propiedad de preservación al multiplicar cadenas, pues se inicia con un tipo Nat y se finaliza con un tipo String.

Formalización

$$\frac{}{num(n):\text{Nat}} \qquad \frac{s:\text{String} \quad r:\text{String}}{concat(s,r):\text{String}}$$

$$\frac{}{string(s):\text{String}} \qquad \frac{}{mult(i,d):\text{Nat}}$$

Representación de las reglas en el sistema implementado:

```
smest :: Arbol -> Maybe Tipo
smest (Mkt (Tk "Num") h) = Just (Conc "Nat")
smest (Mkt (Tk "Str") h) = Just (Conc "String")
smest (Mkt (Tk "Mult") h) = Just (Conc "Nat")
smest (Mkt (Tk "Conc") (h1::h2:xs)) =
   if Just (Conc "String") == (smest h1) &&
      Just (Conc "String") == (smest h2) then
```

```
        Just (Conc "String")
    else
        Nothing
```

### 3.4.4. Interpretación de resultados

La salida del verificador arroja los siguientes resultados:

```
verificador1 (espacioBusqueda (espacioB formula))
*** Exception:
Contraejemplo: Mkt Mult [Mkt Str [],Mkt Num []]
CallStack (from HasCallStack):
error, called at Verificador.hs:57:56
in main:Verificador
```

Este resultado indica que, bajo la especificación dada, el sistema de tipos de este lenguaje no es seguro. En este caso se da el siguiente programa como contraejemplo:

```
Mkt Mult [Mkt Str [],Mkt Num []]
```

Que corresponde con una expresión de la forma ‘‘foo’’ * 3.

En este caso el programa no cumple la propiedad de preservación pues el programa inicialmente tiene el tipo Nat mientras que al dar un paso en la ejecución se obtiene como resultado una concatenación que tiene como tipo String.

Este capítulo muestra algunos casos de estudio para distintos lenguajes así como sus resultados. Aunque las pruebas se realizaron con lenguajes pequeños, es posible tomar fragmentos de lenguajes con más construcciones sintácticas y probar sólo un subconjunto de estos. También es posible aumentar la capacidad de este sistema para analizar lenguajes con tipos función; el capítulo de conclusiones profundiza más en esto.

# Capítulo 4

# Conclusiones y trabajo futuro

## 4.1. Conclusiones

El sistema desarrollado en este trabajo pone a prueba la técnica de verificación de modelos Glass Box con el objetivo de verificar la seguridad de sistemas de tipos reimplementando el sistema original con el apoyo del lenguaje HASKELL con el fin de generar las fórmulas proposicionales entregadas al solucionador SAT de forma directa al ser un lenguaje declarativo.

Las pruebas realizadas se definieron a partir de esqueletos con alturas mínimas como 2 o 3 pues para alturas mayores el proceso de generación de restricciones se vuelve lento generando además espacios de búsqueda de longitudes mayores. En este sentido es importante mencionar que el sistema no prueba todos los posibles programas generados por el lenguaje, principalmente porque el espacio de búsqueda sería infinito aunque sería un problema interesante el tratar de resolver esto, tal y como se menciona en el Capítulo 2 una posible forma de hacer esto es incrementar la altura de forma automática hasta llegar a un contraejemplo. Aún con un espacio de búsqueda finito y con esqueletos de alturas pequeñas, podemos aplicar la *hipótesis de alcance pequeño*[1] que establece que una gran proporción de errores puede ser encontrada haciendo pruebas dentro de un alcance pequeño, la cual se puede apreciar en algunas de las pruebas realizadas.

También es importante mencionar que el sistema desarrollado no sustituye a las pruebas formales que se realizan para verificar seguridad, sino que sirve como una herramienta de apoyo que permite detectar errores tempranos en el diseño. La realización de pruebas formales es difícil en la mayoría de los lenguajes de programación *reales* debido a la cantidad de constructores que estos contienen hoy en día. Incluso el diseño de reglas para especificar sus semánticas estática y dinámica es prácticamente imposible. El uso de herramientas similares al sistema desarrollado en este trabajo pueden emplearse para fragmentos de los lenguajes de programación reales, por ejemplo tomar sólo una parte del núcleo o una biblioteca específica del lenguaje.

Por otro lado, difiere de otros métodos como las pruebas unitarias pues el sistema desarrollado trata de generar todos los posibles casos por medio de fuerza bruta, mientras las pruebas unitarias no necesariamente generarán todos los casos posibles, perdiendo la garantía de que el sistema de tipos cumple con su propiedad de seguridad en todos los

[1] *Small scope hypothesis*

casos. Sin embargo, desde el punto de vista de la variación de las alturas tiene varias similitudes al garantizar sólo la seguridad para programas de un tamaño en específico.

La elección de un lenguaje como HASKELL facilitó el proceso de generación de restricciones gracias a su naturaleza declarativa y funcional, debido principalmente a la traducción de conjuntos de fórmulas proposicionales a listas por comprensión generados con el apoyo de las funciones de plegado `foldr` y `foldr1` que en otros lenguajes de programación podría haberse complicado. Adicionalmente la traducción de las reglas de semántica dinámica y estática se realizó de forma directa pues éstas dos últimas se definen por medio de funciones que pueden manejarse de forma natural como cualquier otro valor en el lenguaje al tratarse como miembros de primera clase, aunque por otro lado un problema que de detectó al realizar este trabajo es que pudieran existir lenguajes con reglas no deterministas, con lo cual no se podrían usar funciones de manera tan sencilla, situación para la cual se planteó la posibilidad de usar un lenguaje lógico con PROLOG.

Las restricciones generadas por el sistema fueron quizá la parte más complicada en el desarrollo de este trabajo debido principalmente a que dependían completamente de la estructura de los esqueletos y de la manipulación del sistema de tipos. Esto puede llegar a complicarse para lenguajes con constructores más elaborados y a sistemas de tipos más potentes que incluyan variables de tipo por ejemplo. Sin embargo, la generación de estas restricciones facilita en gran medida la construcción del espacio de búsqueda pues partiendo de una fórmula genera programas que la cumplan.

El proceso de poda (véase Capítulo 2) proporcionado por la detección de programas similares y generación de fórmulas que eliminen programas del espacio de búsqueda evita en gran medida el problema de explosión de estados presentado en el capítulo 1, los resultados obtenidos muestran una reducción en los tiempos de verificación significativa. Sin embargo, en cuestiones de tiempo, se detectó que la construcción del espacio de búsqueda también debe ser optimizada pues el proceso de transformación de fórmula a árbol se repite en varias ocasiones debido a la reconstrucción de la fórmula asociada a los programas similares.

El proceso de generación de restricciones y de poda del sistema implementado actualmente permite verificar lenguajes de programación con sistemas de tipos básicos, que, como se mencionó con anterioridad, no incluyen variables de tipos ni otros conceptos avanzados sobre tipos. Sin embargo, es posible proporcionar al sistema incluso otros lenguajes que no necesariamente sean de programación, por ejemplo el lenguaje de algún sistema lógico.

## 4.2. Trabajo futuro

Como trabajo a futuro se propone mejorar el proceso de generación de restricciones de forma tal que la traducción de la especificación no se dé mediante código escrito en HASKELL sino de un lenguaje de especificación propio para el sistema que tenga su propio análisis léxico y sintáctico para construir el espacio de búsqueda. Es posible optimizar el proceso de *regeneración* de fórmulas necesario para el proceso de poda con el fin de no volver a construir programas ya existentes en dicho espacio aunque esto debe combinarse de alguna manera con el proceso de generación de restricciones.

Por otro lado al contar con su propio lenguaje de especificación sería adecuado añadir un módulo que tradujera sistemas de tipos que cuenten con variables de tipo para trabajar además con lenguajes que cuenten con polimorfismo. Esto permitirá analizar lenguajes con sistemas de tipos más elaborados donde es más difícil rastrear

errores de seguridad debido principalmente al uso de algoritmos de inferencia de tipos o de verificación previos. Adicionalmente a esto, se debería contar con algún mecanismo en el proceso de verificación que incluya un contexto de tipos para poder lidiar con las variables de tipos.

Un ejemplo de problema que podría apoyarse en el sistema con las adecuaciones antes mencionadas podría ser verificar si una expresión de la lógica de predicados está bien escrita. Si bien la lógica de predicados no cuenta con tipos, la definición de su sintaxis puede realizarse por medio de distintas categorías sintácticas como pueden ser los términos y las fórmulas, mismas que pueden manejarse como tipos. Por ejemplo, verificar si un predicado se está aplicando a la cantidad de términos adecuada.

A continuación se muestra la especificación del lenguaje de la lógica de predicados por medio de esta notación: [19]

$$\frac{}{ar \vdash V : \text{term}}$$

$$\frac{}{ar \vdash C : \text{term}}$$

$$\frac{ar \vdash Ts : terms(n) \qquad ar(F) = n}{ar \vdash F(Ts) : \text{term}}$$

$$\frac{ar \vdash T_1 : \text{term} \qquad ar \vdash T_2 : \text{term}}{ar \vdash let\ V = T_1\ in\ T_2 : \text{term}}$$

$$\frac{ar \vdash F : \text{formula} \qquad ar \vdash T_1 : \text{term} \qquad ar \vdash T_2 : \text{term}}{ar \vdash if\ F\ then\ T_1\ else\ T_2 : \text{term}}$$

$$\frac{ar \vdash T : \text{term}}{ar \vdash T : \text{terms(1)}}$$

$$\frac{ar \vdash T : \text{term} \qquad ar \vdash Ts : \text{terms(n)}}{ar \vdash T,Ts : \text{terms(n+1)}}$$

$$\frac{}{ar \vdash true : \text{formula}}$$

$$\frac{}{ar \vdash false : \text{formula}}$$

$$\frac{ar \vdash Ts : terms(n) \qquad ar(P) = n}{ar \vdash P(Ts) : \text{formula}}$$

$$\frac{ar \vdash T_1 : \text{term} \qquad ar \vdash T_2 : \text{term}}{ar \vdash T_1 = T_2 : \text{formula}}$$

$$\frac{ar \vdash F : \text{formula}}{ar \vdash \neg F : \text{formula}}$$

$$\frac{ar \vdash F_1 : \text{formula} \qquad ar \vdash F_2 : \text{formula}}{ar \vdash F_1 \wedge F_2 : \text{formula}}$$

$$\frac{ar \vdash F_1 : \text{formula} \qquad ar \vdash F_2 : \text{formula}}{ar \vdash F_1 \vee F_2 : \text{formula}}$$

$$\frac{ar \vdash F_1 : \text{formula} \qquad ar \vdash F_2 : \text{formula}}{ar \vdash F_1 \Rightarrow F_2 : \text{formula}}$$

$$\frac{ar \vdash F_1 : \text{formula} \qquad ar \vdash F_2 : \text{formula}}{ar \vdash F_1 \Leftrightarrow F_2 : \text{formula}}$$

$$\frac{ar \vdash F : \text{formula}}{ar \vdash \forall V.F : \text{formula}}$$

$$\frac{ar \vdash F : \text{formula}}{ar \vdash \exists V.F : \text{formula}}$$

$$\frac{ar \vdash T : \text{term} \qquad ar \vdash F : \text{formula}}{ar \vdash let\ V = T\ in\ F : \text{formula}}$$

$$\frac{ar \vdash F : \text{formula} \qquad ar \vdash F_1 : \text{formula} \qquad ar \vdash F_2 : \text{formula}}{ar \vdash if\ F\ then\ F_1\ else\ F_2 : \text{formula}}$$

Algunas observaciones de este lenguaje:

- *ar* representa un contexto que se utiliza como auxiliar para verificar la aridad de los predicados.
- term representa la categoría sintáctica de los términos: variables, constantes y funciones aplicadas a términos de una aridad en específico.
- formula representa la categoría de las fórmulas: conjunciones, disyunciones, implicaciones, equivalencias, fórmulas cuantificadas y una forma de introducir variables mediante *let*.

El verificador implementado en este trabajo podría ayudar a verificar la seguridad del mismo y hacer las adecuaciones necesarias en caso de que el lenguaje no capture la sintaxis del sistema lógico.

Esto se puede generalizar a lenguajes de programación o lenguajes de especificación donde se cuente con etiquetas de clasificación (tipos), donde el verificador implementado en este trabajo puede ser de gran utilidad.

# Bibliografía


[1] Alfred V. Aho y Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. 1 de ene. de 1972. URL: https://doi.acm.org/10.1145/578812.

[2] Christel Baier y Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, abr. de 2008.

[3] Adam Chlipala. *Certified Programming with Dependent Types*. 1 de ene. de 2013. DOI: 10.7551/mitpress/9153.001.0001. URL: https://doi.org/10.7551/mitpress/9153.001.0001.

[4] Edmund M. Clarke Jr y col. *Model Checking, second edition*. MIT Press, 4 de dic. de 2018.

[5] Edmund M. Clarke y Bernd-Holger Schlingloff. *Model checking*. Ene. de 1999. URL: https://dl.acm.org/citation.cfm?id=332656.

[6] Wikipedia contributors. "Ariane 5". En: *Wikipedia* (ago. de 2023). URL: https://en.wikipedia.org/wiki/Ariane_5.

[7] Watts S. Humphrey. *A discipline for software engineering*. Addison-Wesley Professional, 1 de ene. de 1995.

[8] Michael Huth y Mark Ryan. *Logic in computer science*. Ago. de 2004.

[9] Graham Hutton. *Programming in Haskell*. 15 de ene. de 2007. URL: https://assets.cambridge.org/97805218/71723/frontmatter/9780521871723_frontmatter.pdf.

[10] *ISO/IEC 25010:2011*. 1 de ene. de 2015.

[11] Mayur Naik y Jens Palsberg. "A Type System Equivalent to a Model Checker". En: *ACM Transactions on Programming Languages and Systems* 30.5 (ago. de 2008), págs. 1-24. DOI: 10.1145/1387673.1387678. URL: https://doi.org/10.1145/1387673.1387678.

[12] Rob Nederpelt y Herman Geuvers. *Type theory and formal proof. An Introduction*. Cambridge University Press, 6 de nov. de 2014.

[13] Benjamin C. Pierce. *Advanced topics in types and programming languages*. MIT Press, 23 de dic. de 2004.

[14] Zhiqiang Ren y Hongwei Xi. *Combining type-checking with model-checking for system verification*. 2016. DOI: 10.1109/MEMCOD.2016.7797745.

[15] Michael Roberson y col. "Efficient software model checking of soundness of type systems". En: *ACM Sigplan Notices* 43.10 (oct. de 2008), págs. 493-504. DOI: 10.1145/1449955.1449803. URL: https://doi.org/10.1145/1449955.1449803.

[16] Michael E. Roberson. "Glass Box Software Model Checking". Tesis doct. University of Michigan, 2011.

[17] Alan J.A. Robinson y Andrei Voronkov. *Handbook of Automated Reasoning*. Elsevier, 21 de jun. de 2001.

[18] *SAT.MiniSat*. URL: https://hackage.haskell.org/package/minisat-solver-0.1/docs/SAT-MiniSat.html.

[19] Wolfgang Schreiner. *Thinking programs. Logical Modeling and Reasoning About Languages, Data, Computations, and Executions*. Springer Nature, 22 de oct. de 2021.

[20] Carsten Sinz y Tomas Balyo. *Practical SAT Solving: Lecture 1*. URL: https://baldur.iti.kit.edu/sat/files/2019/l01.pdf.

[21] Wikipedia contributors. "Mariner 1". En: *Wikipedia* (18 de jun. de 2023). URL: https://en.wikipedia.org/wiki/Mariner_1.

[22] Wikipedia contributors. "Phobos 1". En: *Wikipedia* (16 de jul. de 2023). URL: https://en.wikipedia.org/wiki/Phobos_1.

[23] Wikipedia contributors. "Therac-25". En: *Wikipedia* (15 de jul. de 2023). URL: https://en.wikipedia.org/wiki/Therac-25.