Universidad Nacional Autónoma de México

Facultad de Ciencias

El lenguaje Java Peso Pluma, la esencia del paradigma orientado a objetos

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
EDUARDO MENDOZA AGUILAR

DIRECTOR DE TESIS:
FAVIO EZEQUIEL MIRANDA PEREA

2012

Con amor para mi padre, 9 años te bastaron...

Con amor para mi madre, la mejor mamá del mundo, esa que lo puede todo y más, sin ti esto no sería posible.

A mi "che pimo", gracias por tu máquina de escribir, mira adónde nos llevó...

Índice general

Capítulo 1

Introducción

Este trabajo consiste en el desarrollo del intérprete Java Peso Pluma. Dicho intérprete es un prototipo que contiene las características esenciales de su paradigma, por tanto es muy útil para iniciar el estudio a fondo de un lenguaje de programación. Se decidió usar Java Peso Pluma para este desarrollo por su ligereza y gran documentación, así como sus similitudes con el lenguaje Java, el cual se encuentra muy en boga actualmente. El desarrollo se hará sobre el lenguaje Haskell, uno de los más adecuados debido a sus características funcionales las cuales permite un desarrollo ágil y con una (relativamente) pequeña cantidad de código.

1.1. Historia y usos comunes del paradigma orientado a objetos

Para algunas personas escuchar las palabras *paradigma* y *orientado a objetos* en una misma frase podría resultar más que confuso, para los alumnos de la carrera de Ciencias de la Computación quizá sea algo bastante común, pues lo han escuchado en diversas ocasiones desde el inicio de su carrera. Sin embargo, muchas veces no entendemos completamente el significado de los términos que usamos día a día. Antes de introducirnos en la historia hay que mencionar de una forma más completa el significado de estos conceptos. Si nos damos a la tarea de buscar, podremos encontrar una gran cantidad de definiciones para estas palabras, por ejemplo:

Paradigmas: Realizaciones científicas universalmente reconocidas que, durante cierto tiempo, proporcionan modelos de problemas y soluciones a una comunidad científica.[7]

Programación Orientada a Objetos: Contestar a la pregunta ¿Qué es la programación orientada a objetos? no es sencillo, y depende del punto de vista particular de quién responda, sin embargo existen ciertas características impor-

tantes del paradigma que deberían estar incluidas en cualquier respuesta a esta pregunta: representaciones múltiples, encapsulado, subtipo (herencia de interfaces), herencia (de implementaciones) y recursión abierta.[8] Más adelante se retomarán estos conceptos y se darán sus respectivas definiciones, dando así respuesta a dicha pregunta.

Un programador tiene diversas formas de dar instrucciones a una computadora. A lo largo de la historia se han creado distintos lenguajes de programación, algunos diseñados para hacer programas más rápidos y eficientes, otros para ser amigables con el programador; sin embargo, algo ha quedado claro después de tantos años de desarrollo, los lenguajes más exitosos han sido aquellos que fueron creados para un propósito específico, esto no significa que cada lenguaje sea bueno únicamente para un propósito. Ahora bien, ¿Cómo se define el término lenguaje de programación?; De acuerdo con John Mitchell: [9] "*Los lenguajes de programación son el medio de expresión en el arte de la programación de computadoras.*"

Ya hemos mencionado que se han creado muchos lenguajes a lo largo de la historia de la computación, pero ¿cómo clasificarlos? ¿qué los hace diferentes? Bien, lo que se hace es juntar en una categoría a aquellos que usan componentes similares, como podrían ser funciones, objetos, predicados, etc, para la comunicación con la computadora. Es decir, aquellos que varían en el paradigma.

Los paradigmas reciben distintos nombres dependiendo del tipo de componentes a los que hacen mención, por ejemplo algunos de los más comunes son:

- Paradigma Imperativo o por Procedimientos. Lenguajes cuyas instrucciones son procedimientos (C, Basic, Pascal).
- Paradigma Funcional. Lenguajes cuyas instrucciones son funciones (Scheme, ML, Haskell)
- Paradigma Lógico. Lenguajes cuyas instrucciones son hechos y predicados (Prolog)
- Paradigma Orientado a Objetos. Lenguajes cuyas instrucciones son Clases y Objetos (Smalltalk, Java)

Es importante mencionar que existen otros lenguajes que combinan dos o más paradigmas llamados *lenguajes multiparadigma*, como es el caso de C++, el cual combina el paradigma imperativo con el paradigma orientado a objetos.

El estudio de esta tesis estará enfocado al lenguaje Java Peso Pluma, el cual es un prototipo de lenguaje de programación con las características primitivas del paradigma orientado a objetos.

1.1.1. Breve Historia

En los últimos 50 años se han creado cientos de lenguajes de programación. Aproximadamente 50 de ellos contienen conceptos nuevos, mejoras útiles o innovaciones. Pocos lenguajes llegan a ser usados en desarrollos de software. Seis de estos lenguajes han trascendido en el mundo de la programación: Lisp, ML, C, C++, Smalltalk y Java; juntos contienen los elementos más importantes de los lenguajes de alto nivel desde que este concepto dio un salto a la programación, en la Conferencia de Lenguajes de Programación llevada a cabo alrededor de 1960.[1]

La historia de los lenguajes de programación modernos da inicio a finales de la década de 1950, cuando se desarrollaron Algol, Cobol, Fortran y Lisp. En aquella época se creó una gran cantidad de lenguajes, buscando con ello simplificar el proceso de escribir secuencias de instrucciones para las computadoras, las cuales eran muy primitivas en comparación con una computadora moderna. De esta década los dos lenguajes más importantes fueron Fortran y Cobol.

IBM desarrolló Fortran de 1954 a 1956 con un equipo de programadores liderados por John Backus. La principal innovación aportada por este lenguaje fue permitir escribir instrucciones en notación matemática, por ejemplo:

$$j \;\; is \;\; i + 2 * j.$$

Anteriormente, esto hubiera supuesto una secuencia de instrucciones que no permitían al programador pensar de manera natural en el cálculo matemático *per se*. Fortran también tenía subrutinas, arreglos, formato para entrada y salida, así como instrucciones que permitían al programador controlar los lugares que las variables y los arreglos ocuparían en memoria. Sin embargo, Fortran aún contaba con muchas limitaciones, pues los valores almacenados en memoria eran fácilmente reemplazados por los programadores si éstos no eran cuidadosos. Además las subrutinas no eran capaces de llamarse a sí mismas, pues aún no existía el concepto de recursión.

El primer diseño de Cobol fue hecho por el pionero en computación Grace Murray Hopper, y la sintaxis de Cobol fue diseñada para verse como el inglés común; es un lenguaje diseñado para las aplicaciones en negocios. Tanto Fortran como Cobol tuvieron gran auge en la programación, actualmente muchos de los programas en uso fueron programados en alguna versión de estos dos lenguajes.

En la década de 1960 se desarrollaron Algol y Lisp. Estos lenguajes contaban con un manejador de memoria y funciones o procedimientos recursivos. Lisp proporcionaba funciones de orden superior y recolector de basura. Así mismo, Algol tenía un sistema de tipos mejorado y estructuras de datos. En la década de 1970 surgieron métodos para la organización de datos, como los *records*, tipos de datos abstractos y estructuras parecidas a los objetos.

Los conceptos de la programación orientada a objetos tienen su origen en Simula 67, un lenguaje diseñado para hacer simulaciones de naves, creado por

[1] Esta sección está basada en el libro *Concepts in Programming Languages* de John Mitchell (capítulo: *1.3 Programming language history*)[9]

Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo.

La programación orientada a objetos tomó posición como el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las interfaces gráficas de usuario, para las cuales la programación orientada a objetos está particularmente bien adaptada.

Las características de la orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, entre otros. La incorporación de estas características a los lenguajes que no fueron diseñados inicialmente trajeron a menudo problemas tanto de compatibilidad como de capacidad de mantenimiento del código.

El lenguaje Java se volvió uno de los lenguajes representantes del paradigma orientado a objetos, en gran parte por la aparición de Internet, y a la implementación de su máquina virtual en la mayoría de navegadores. Fue desarrollado por **Sun Microsystems** a principio de la década de 1990.

1.2. Conceptos fundamentales

Los conceptos más importantes referentes a los lenguajes de programación quizá son aquellos que nos dan una idea clara de las diferencias y posibles usos entre los lenguajes relacionados con nuestro tema en cuestión. De manera gráfica veamos las propiedades más representativas de algunos lenguajes vinculados con nuestro estudio.[2]

Lenguaje	Expresiones	Funciones	Almacenamiento por Montículo	Excepciones	Módulos	Objetos	Hilos
Lisp	●	●	●				
C	●	●	●				
Algol 60	●	●					
Algol 68	●	●	●				●
Pascal	●	●	●				
Modula-2	●	●	●		●		
Modula-3	●	●	●	●	●	●	
ML	●	●	●	●	●		
Simula	●	●	●			●	●
Smalltalk	●	●	●	●		●	●
C++	●	●	●	●	●	●	
Objective C	●	●	●			●	
Java	●	●	●	●	●	●	●

Cuadro 1.1: Tabla del libro *Concepts in Programming Languages* de John Mitchell (página 8) [9]

[2]Esta sección está basada en el libro *Concepts in Programming Languages* de John Mitchell (páginas 7-9)[9]

La tabla representa a unos pocos lenguajes así como sólo algunas propiedades. El tema de los lenguajes y sus propiedades es en realidad muy extenso, pero de manera gráfica se pretende mostrar las diferencias más importantes que estos lenguajes tienen entre sí. Podemos notar como muy pocos lenguajes cuentan con el manejo de excepciones o la creación de hilos. Sin embargo, el lenguaje JAVA, que es nuestro caso de estudio, cuenta con todas la propiedades, pues a lo largo del desarrollo de los lenguajes orientados a objetos éste se ha colocarlo como uno de los lenguajes más completos. Pero son estas propiedades; ser un lenguaje sólido y la gran aceptación de su paradigma en las aplicaciones actuales, lo que lo hacen objetivo para ser estudiado fundamentalmente desde sus características más primitivas.

1.2.1. Java y sus propiedades

Hablemos ahora del lenguaje JAVA, como ya vimos en los apartados anteriores, se trata de un lenguaje que usa el paradigma orientado a objetos, pero aún no mencionamos qué lo hace diferente a otros lenguajes que comparten dicho paradigma.

JAVA fue desarrollado por **Sun Microsystems** en la década de 1990, su sintaxis se basa en la de C y C++, pero omite muchas herramientas de bajo nivel, como son la manipulación directa de apuntadores y memoria, además de que su sistema de objetos es más simple. **Sun Microsystems** liberó la mayoría de sus tecnologías JAVA bajo la licencia GNU GPL (*Licencia Pública General de GNU*), lo que hace a JAVA software libre, excepto por las bibliotecas necesarias para ejecutar programas.

Este lenguaje trabaja con el apoyo de una Máquina Virtual (JVM *Java Virtual Machine*), la cual se encarga de pasar el código compilado a *bytecode* o lenguaje de máquina. Además la JVM tiene un sistema de memoria avanzado, disminuye el tiempo de espera para el recolector de basura y provee de portabilidad al lenguaje.

En sentido estricto se podría decir que JAVA no es completamente un lenguaje orientado a objetos, puesto que mantiene un sistema de tipos nativos los cuales no son objetos los cuales son: **int, short, long, float, double, char**, entre otros. Sin embargo, sigue siendo uno de los lenguajes más atractivos para los programadores, sus herramientas permiten hacer desarrollos en distintos ámbitos, pues tiene un excelente manejo de excepciones, soporta hilos de ejecución, permite ejecutar código en otros lenguajes mediante JNI (*Java Native Interface*) y finalmente su portabilidad hace que sus características sean lo suficientemente atractivas para hacerlo uno de los lenguajes más usados en la actualidad.

1.3. Características fundamentales del P.O.O.

A continuación veremos las características fundamentales del paradigma orientado a objetos, para entender mejor su funcionamiento.[3]

- **Múltiples Representaciones**. Quizá la propiedad básica de un lenguaje orientado a objetos es que cuando se invoca al método de un objeto, el objeto en sí determina qué código se ejecutará. Dos objetos que responden al mismo conjunto de operaciones (es decir, pertenecen a la misma *interfaz*) pueden usar representaciones completamente diferentes, donde cada uno cuenta con una implementación de la operación que funcionará con su representación particular. Estas implementaciones se llaman *métodos de los objetos*, e invocar una operación en un objeto se conoce como *invocación a un método*.

- **Encapsulamiento**. La representación interna de un objeto generalmente se oculta de todo aquello ajeno a la definición del objeto, ya que sólo sus métodos pueden acceder y modificar sus atributos, lo cual significa que las modificaciones internas del objeto serán muy pocas, facilitando así el mantenimiento y la legibilidad de sistemas muy grandes.

- **Subtipado**. El tipo de un objeto —lo que se define como su interfaz— es el conjunto de nombres y tipos de sus operaciones. La representación interna del objeto no aparece en su tipo, lo que no afecta al conjunto de operaciones que podemos hacer directamente con él.
  La interfaz de un objeto se relaciona de manera natural con la relación de subtipado. Si un objeto satisface la interfaz **I** y el mismo objeto satisface la interfaz **J** la cual cuenta con menos operaciones que **I**, un contexto que espera recibir un objeto **J**, puede sin ningún problema recibir un objeto de **I**, pues dicho objeto **I** cuenta con las operaciones definidas en el objeto **J**.

  La habilidad de ignorar partes de la interfaz de un objeto nos permite escribir un pequeño fragmento de código que manipula una variedad de objetos (no necesariamente del mismo tipo) de manera uniforme, demandando sólo un cierto conjunto de operaciones en común.

- **Herencia**. A los objetos que comparten comportamiento y partes de una misma interfaz, podríamos querer implementarles dicho comportamiento sólo una vez. La mayoría de los lenguajes orientados a objetos logran este reuso del comportamiento mediante *clases-plantilla* en las que se realiza la implementación de los objetos y un mecanismo llamado *subclase*, que permite crear nuevas clases derivadas de las ya existentes, agregando implementaciones de nuevos métodos y de ser necesario, sobrescribiendo implementaciones de métodos ya existentes.

[3] Esta sección está basada en las notas de clase *Lenguajes de Programación y sus Paradigmas 2010-2* Nota de clase 16: *Paradigma Orientado a Objetos II* de Favio E. Miranda [8]

- **Recursión Abierta**. La mayoría de los lenguajes que usan clases y objetos permiten que un método, desde su cuerpo, pueda llamar a otro que pertenece al mismo objeto. Esto se hace mediante una variable especial llamada *this*.

Durante este capítulo se habló de la historia y características del lenguaje JAVA a grandes rasgos, en los siguientes capítulos se analizará el lenguaje de manera más formal y específica en lo referente a sus componentes, comenzando así a estudiar los aspectos teóricos detrás del lenguaje.

Capítulo 2

Sistema de tipos y subtipos

Los tipos en un lenguaje de programación sirven de herramienta para la elaboración de programas de computadora, su finalidad es restringir los posibles valores que tomarán los datos dentro de un programa. A continuación se explicaremos la forma en que se incluye esta propiedad en los lenguajes y las ventajas que proporcionan al programador.

2.1. Sistema de tipos

El sistema de tipos juega un rol muy importante en los lenguajes de programación, su función es prevenir *errores de ejecución*, los cuales se producen al ejecutar algún programa. Dichos errores motivan el estudio formal del sistema de tipos. Pero, todo ello requiere una aclaración más precisa, pues definir qué es un *error de ejecución* —aún cuando ya se haya aclarado con definiciones previas— no es trivial. Cuando el sistema de tipos se cumple para todos los programas en un lenguaje, decimos que ese lenguaje es *correctamente tipado,*[1] sin embargo, es importante evitar pedir a los lenguajes una *correctez*[2] excesiva, por ello el estudio de los sistemas de tipos se ha vuelto una disciplina formal.

La formalización del sistema de tipos requiere de un desarrollo con notación precisa y definiciones, así como pruebas formales y detalladas con el fin de que éstas sean fiables.

Cuando se desarrolla apropiadamente, el sistema de tipos provee de herramientas conceptuales que permiten juzgar aspectos importantes de las definiciones de un lenguaje. En general, las definiciones informales del sistema de tipos de un lenguaje provocan fallas en la especificación de la estructura de tipos y de este modo el lenguaje podría permitir implementaciones ambiguas. De tal modo que diferentes compiladores del mismo lenguaje cuenten con pequeñas diferencias en el sistema de tipos. Más aún, muchas definiciones de lenguajes

[1] Traducción de: type sound.
[2] Traducción de: soundness.

se han encontrado *incorrectas,*[3] lo que permite la existencia de programas que incurren en errores aún después de que han sido analizados y aceptados por el *verificador de tipos.*[4]

2.1.1. Errores de ejecución

El síntoma más obvio de un error de ejecución es la aparición de un fallo inesperado en el software, como podría ser una instrucción ilegal o un acceso ilegal a la memoria.

Sin embargo, existen más tipos de errores como: datos corruptos, los cuales no presentan un síntoma inmediato. Por ejemplo la división entre cero y referencias a datos nulos, los cuales usualmente no son prevenidos por el sistema de tipos. Finalmente, existen lenguajes que carecen de un sistema de tipos, y aún así, los fallos de software no ocurren. En la siguiente sección se hablará más a fondo de estos lenguajes.

2.1.2. Lenguajes tipados y no tipados

Una variable puede tomar cierto rango de valores durante la ejecución de un programa. Dicho rango es impuesto a la variable por lo que se conoce como *tipo* de la variable. Por ejemplo, una variable x de tipo *Boolean*, asumimos que solo recibirá valores booleanos en cada ejecución del programa. Si x es de tipo booleano entonces la expresión $not(x)$ tiene sentido en cada ejecución del programa. Los lenguajes donde las variables pueden ser tipadas se conocen como *lenguajes explícitamente tipados*.

Los lenguajes que no restringen el rango de valores para las variables se conocen como *lenguajes no tipados explícitamente*. Estos no tienen tipos explícitos o, de manera equivalente, pueden tener un tipo universal para todos los valores posibles de las variables. En estos lenguajes las operaciones pueden ser aplicadas con argumentos inapropiados, el resultado podría ser arreglado de manera arbitraria dando un valor, un fallo, una excepción o un efecto no especificado. El extremo de estos lenguajes es el λ-cálculo puro, el cual es no tipado explícitamente y estos errores no ocurren nunca, ya que los operadores hacen aplicaciones a funciones y los valores son funciones, por tanto no hay fallos.

Un sistema de tipos es un componente de los lenguajes tipados que provee de tipos a las variables del lenguaje y, en general, a todas las expresiones en un programa. El sistema de tipos es usado para determinar cuándo un programa se comporta adecuadamente. Sólo aquellos programas que cumplen con el sistema de tipos se pueden considerar programas válidos del lenguaje tipado, de otra forma el programa deberá ser descartado antes de la ejecución.

Un lenguaje es tipado en virtud de la existencia de un sistema de tipos para él, aparezcan o no tipos en la sintaxis de los programas. Se dice que un lenguaje tipado es *explícitamente tipado* si los tipos son parte de la sintaxis, e *implícitamente tipado* en caso contrario. No se han establecido lenguajes puramente im-

[3] Traducción de: unsound.
[4] Traducción de: typechecker.

plícitamente tipados, pero lenguajes como ML o HASKELL, soportan grandes cantidades de código omitiendo la información de tipos. El sistema de tipos de estos lenguajes asigna un tipo a los fragmentos de código automáticamente.[5]

2.1.3. Errores de ejecución y seguridad

Se puede distinguir entre dos tipos de errores en tiempo de ejecución: aquellos que provocan que el programa se detenga por completo al momento que ocurren (*errores cazados*[6]) y aquellos que pueden pasar desapercibidos durante un tiempo, dando como resultado errores a largo plazo en nuestro programa (*errores no cazados*[7]).

Ejemplos de errores no cazados son: acceder a una región de memoria legal pero inadecuada, saltar a una región de memoria incorrecta o sobrepasar la longitud de un arreglo. Ejemplos de errores cazados son: divisiones entre cero o tratar de acceder a una región de memoria ilegal lo que provocaría una interrupción de la ejecución (en la mayoría de las arquitecturas).

Se dice que una fragmento de código es seguro si no existen en él errores no cazados, y si todos los fragmentos del programa son seguros entonces se dice que el programa es seguro. Los lenguajes no tipados usan análisis en tiempo de ejecución para hacer al lenguaje seguro, mientras que los lenguajes tipados hacen análisis estáticos o en tiempo de compilación. Los lenguajes tipados también pueden hacer análisis en tiempo de ejecución en combinación con los estáticos.

Aunque la seguridad es una propiedad crucial de los lenguajes de programación, es muy raro que un lenguaje tipado haga demasiados esfuerzos por eliminar todos los errores no cazados.

2.1.4. Errores de ejecución y programas bien comportados

Para cualquier lenguaje dado, definiremos a un conjunto de posibles errores de ejecución como *errores prohibidos*. Este conjunto de errores debe incluir todos los errores no cazados y además incluir algunos errores cazados. Un fragmento de código se considera bien comportado, si en él no ocurren errores prohibidos. En caso contrario se dice que el fragmento de código es mal comportado. En particular un fragmento bien comportado es seguro. Un lenguaje en el que todos los programas (legales) son bien comportados se llama *fuertemente verificado*.[8]

Así, las condiciones necesarias para que un lenguaje sea fuertemente verificado son:

- No ocurren errores no cazados (garantía de seguridad).
- No ocurren errores cazados que también sean errores prohibidos.

[5] A esta propiedad se le llama: inferencia de tipos.
[6] Traducción de: trapped errors.
[7] Traducción de: untrapped errors.
[8] Traduccion de: Strongly Checked.

- Pueden ocurrir errores cazados; es responsabilidad del programador evitarlos.

Los lenguajes tipados pueden asegurar ser bien comportados (incluyendo seguridad) de manera estática, es decir, en tiempo de compilación. Estos lenguajes hacen análisis estático; el proceso de análisis es llamado *verificación de tipos*[9] y el algoritmo para hacer dicho análisis se conoce como *verificador de tipos.*[10] Un programa que pasa el verificador de tipos se dice que es bien tipado, se dice mal tipado en caso contrario y por tanto mal comportado. Ejemplos de lenguajes satisfactoriamente analizados son ML, JAVA y PASCAL.

Los lenguajes no tipados pueden garantizar ser bien comportados (incluyendo seguridad) de una forma diferente, generando análisis de tipos en tiempo de ejecución. A esto se le conoce como análisis dinámico. Un ejemplo puede ser el lenguaje LISP.

Algunos lenguajes combinan ambas técnicas mediante el uso de herramientas adicionales, como por ejemplo: SIMULA67 usando INSPECT, MODULA-3 usando TYPECASE o JAVA con instanceof.

2.1.5. Fallos en la seguridad

Dada nuestra definición, un programa bien comportado es un programa seguro. La seguridad es la propiedad primordial de que un lenguaje se comporta adecuadamente. Lo más importante del sistema de tipos es asegurar que el lenguaje no tenga errores no cazados en ninguno de sus programas. Sin embargo, la mayoría de los sistemas de tipos están diseñados para asegurar de manera más general el buen comportamiento del lenguaje.

En realidad, existen lenguajes con análisis estático que no son seguros. Esto es, los errores prohibidos no incluyen todos los errores no cazados. A estos lenguajes se les conoce como *débilmente verificados.*[11] Uno de ellos es el lenguaje C, cuyos problemas en seguridad se ven al usar su aritmética de punto flotante y *casting*. La mayoría de los lenguajes no tipados son seguros, dado que los programadores encontrarían frustrante tener que hacer dos revisiones, una en tiempo de compilación y otra en tiempo de ejecución para asegurar que sus programas funcionen.

SEGURIDAD →

	Tipado	**No tipado**
Seguro	ML,Java	Lisp
Inseguro	C	Assembler

Cuadro 2.1: Tabla basada en el libro *Types and Programming Languages* de Benjamin Pierce(página 6) [10]

[9] Traducción de: typechecking.
[10] Traducción de: typechecker.
[11] Traducción de: weakly checked.

No todos los lenguajes deben ser seguros. Lenguajes como C no son seguros, sin embargo, esto en realidad es deliberado, ya que el análisis en tiempo de ejecución necesario para hacerlo seguro es costoso.

2.2. Cómo formalizar el sistema de tipos

El sistema de tipos es la herramienta que permite decidir si un programa está bien tipado. La que en sí misma es una aproximación de bien comportado (incluyendo seguridad) pero, ¿cómo podemos asegurar que un programa bien tipado será bien comportado? Una vez formalizado el sistema de tipos procederemos a probar el teorema de *correctez de tipos*[12], el cual asegura que un programa bien tipado es bien comportado. Si el teorema de correctez de tipos es comprobado entonces el sistema de tipos es correcto, que todos los programas de un lenguaje son bien comportados y el sistema de tipos es correcto significan lo mismo.[13]

Formalizar el sistema de tipos y probar el teorema de correctez implica formalizar el lenguaje en sí. Y el primer paso para formalizar un lenguaje de programación es describir su sintaxis. Esto implica describir la sintaxis de sus tipos y términos.

El siguiente paso es describir las reglas de alcance del lenguaje. Dicho alcance debe ser estático en lenguajes tipados, es decir, el alcance de cada identificador deben estar bien establecidos antes de la ejecución. El alcance puede ser definido formalmente mediante la declaración del conjunto de variables libres en un fragmento del programa. La sustitución de variables libres por tipos o términos ahora puede ser especificada.

Cuando lo anterior haya sido determinado, se procede a definir las reglas de tipado en el lenguaje. Esto es, crear las relaciones entre términos y tipos de la forma $M{:}A$, donde los términos M se asocian con los tipos A. Algunos lenguajes también requieren de la relación de subtipado, donde $A \leq B$ relaciona los tipos A y B, haciendo de A un subtipo de B. Y obteniendo una relación de igualdad de tipos, formando tipos equivalentes: $A = B$.

La colección de reglas de tipado generan el sistema de tipos y un lenguaje que cuenta con sistema de tipos es un lenguaje tipado.

En este punto no podemos continuar sin definir un elemento fundamental en la formalización del lenguaje que no se refleja en la sintaxis. Lo que haremos será definir un conjunto que guarde la información de las variables no cazadas en un fragmento de código en un momento determinado durante el análisis de tipos. Las reglas de tipado son siempre formuladas con respecto a contextos estáticos. Por ejemplo, la relación $M{:}A$ es asociada al contexto estático Γ que contiene información de las variables no cazadas de M y A. La relación se escribe como $\Gamma \vdash M{:}A$, lo que significa que M es de tipo A en el contexto Γ.

El último paso para formalizar un lenguaje consiste en definir su semántica como una relación *de valor* entre los términos y una colección de resulta-

[12]Traducción de: type soundness.

[13]Esta sección está basada en la Nota de clase 4 y 15, 2010-2 de Favio E. Miranda. [8]

dos. Dicha relación depende fuertemente del estilo de semántica que se use. En cualquier caso, la semántica y el sistema de tipos se interconectan: el tipo de un término y el tipo de su valor es el mismo; ésta es la esencia del teorema de correctez.

El sistema de tipos especifica las reglas de tipado de un lenguaje de programación independientemente del algoritmo en particular que se use para el análisis de tipos. Esto es análogo a describir la sintaxis de un lenguaje como una gramática formal, independientemente del algoritmo de análisis sintáctico. Es conveniente separar ambas cosas, pues el sistema de tipos pertenece a la definición del lenguaje y el algoritmo verificador de tipos pertenece al compilador. Inclusive, diferentes compiladores pueden usar diferentes algoritmos para el mismo sistema de tipos. Esto hace posible la existencia de sistemas de tipos que sólo admiten algoritmos de análisis de tipos irrealizables o ningún algoritmo en absoluto; sin embargo, en general se busca que el sistema de tipos admita algoritmos eficientes.

2.2.1. Juicios

La descripción del sistema de tipos comienza con la descripción formal de una colección de enunciados a los que se les llama *juicios*, un juicio se ve de la siguiente forma:

$$\Gamma \vdash \Im$$

donde $\Im$ es una afirmación; las variables libres de $\Im$ están declaradas en Γ, decimos que $\Im$ se deriva de Γ. Γ es un contexto de tipos estático; es una lista ordenada de variables y sus tipos, de la forma $\varnothing, x_1{:}A_1, ..., x_n{:}A_n$. El contexto vacío se denota por $\varnothing$, y la colección de variables $x_1, ..., x_n$ declaradas en Γ, son indicadas por $dom(\Gamma)$, el dominio de Γ. La forma de la afirmación $\Im$ varía de juicio a juicio, pero todas las variables no cazadas de $\Im$ deben estar declaradas en Γ.

El juicio más importante para nuestro propósito es el juicio de tipado, el cual afirma que un término M es de tipo A con respecto al contexto de tipos estático para las variables libres de M, la forma de este juicio es:

$\Gamma \vdash M{:}A$ leído como: M es de tipo A en el contexto Γ

Los juicios pueden ser válidos ($\Gamma \vdash$*true:Bool*) o no válidos ($\Gamma \vdash$*true:Nat*). Existen muchas formas de expresar los juicios válidos y no válidos; sin embargo, expresando los juicios válidos únicamente, se tiene una formalización más estilizada, además, al tener solo juicios válidos se tiene mayor modularidad y hace a nuestra formalización más entendible y fácil de leer.

2.2.2. Reglas de tipado

Las reglas de tipado nos aseguran si un juicio es válido, esto se hace mediante otros juicios que ya se comprobaron válidos. Por ejemplo, presentamos la forma general de una regla de tipado:

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

Cada regla de tipado se presenta con cierto número de *premisas* ($\Gamma_i \vdash \Im_i$) sobre una línea horizontal, y con un solo juicio de *conclusión* ($\Gamma \vdash \Im$) bajo la línea. Cuando todas las premisas son ciertas, entonces es válida la conclusión; puede haber cero premisas, en cuyo caso se trata de un hecho.

La regla fundamental es la regla cuyo contexto es vacío:

$$\frac{}{\varnothing \vdash \Im}$$

donde no hay variables libres, ni premisas que cumplir; se dice entonces que, nuestro juicio es una aseveración.

2.2.3. Derivación de tipos

Una derivación en un sistema de tipos dado, es un árbol de juicios con las hojas arriba y la raíz abajo, donde cada juicio se obtiene de otros inmediatamente arriba de él, y éstos a su vez de otros juicios, hasta llegar a las hojas, las cuales son reglas del sistema de tipos. El sistema de tipos tiene como parte de su tarea el poder decidir, cuándo una derivación está o no bien construida.

Un *juicio válido* es aquel que desde la raíz puede ser derivado mediante el sistema de tipos dado. Es decir, cuando la conclusión puede ser obtenida mediante una aplicación correcta de las reglas de tipado.

Por ejemplo, si tenemos las siguientes reglas:

$$\frac{}{\Gamma \vdash \mathtt{n} : \mathtt{Nat}} \quad (\textbf{val n})$$

donde n=1,2,3,...

$$\frac{\Gamma \vdash \mathtt{M} : \mathtt{Nat} \quad \Gamma \vdash \mathtt{N} : \mathtt{Nat}}{\Gamma \vdash \mathtt{M} + \mathtt{N} : \mathtt{Nat}} \quad (\textbf{val} +)$$

podemos usar la derivación para decir si es válido el juicio: $\varnothing \vdash \{1+2\} : Nat$

$$\frac{\overline{\varnothing \vdash \mathtt{1:Nat}}\ \mathtt{de\ (val\ n)} \qquad \overline{\varnothing \vdash \mathtt{2:Nat}}\ \mathtt{de\ (val\ n)}}{\varnothing \vdash \{\mathtt{1} + \mathtt{2}\} : \mathtt{Nat}}\ \mathtt{de\ (val\ +)}$$

2.2.4. Correctez de tipos

Al introducirnos de lleno en las reglas de tipado, debemos tener en mente que el sistema de tipos es un mecanismo muy sensible y es más que una colección de reglas arbitrarias.

Un lenguaje bien tipado, refiriéndonos a la semántica, es un programa bien comportado, es decir, comprobar el teorema de correctez para asegurar la consistencia del sistema de tipos. Aquí es donde el sistema de tipos se une

a la semántica. Para una semántica denotacional[14] nosotros esperamos que si $\varnothing \vdash M : A$ es válido, entonces $[[M]] \in [[A]]$ se cumple. (el valor de M pertenece al conjunto de valores denotados por el tipo A) Y en semántica operacional esperamos que para $\varnothing \vdash M : A$, si M se reduce a M', entonces $\varnothing \vdash M' : A$. En ambos casos el teorema de correctez de tipos asegura que los programas bien tipados no tienen errores de ejecución.

2.3. Subtipado

La idea de subtipos es fundamental en el diseño de lenguajes de programación. La subtipificación permite escribir programas más concisos y de lectura más fácil, al eliminar la necesidad de hacer conversiones explícitas de tipos. También puede ser usada para expresar propiedades adicionales de programas y es un concepto absolutamente fundamental en la programación orientada a objetos, ya que las nociones de subclase y herencia se encuentran fuertemente ligadas a la noción de subtipo.

Las relaciones específicas de subtipo que se incorporen a un lenguaje en particular dependen de muchos factores. Además de las propiedades teóricas que buscamos satisfacer, tenemos que tomar en cuenta los aspectos pragmáticos de la verificación de tipos y de la semántica operacional.

En general debería permitirse cierta flexibilidad en la tipificación si un tipo es mejor que el que se necesita, en el sentido de que un elemento de aquel puede usarse siempre de manera segura cuando se espera un argumento del tipo necesario.

Esta intuición se formaliza introduciendo la relación de subtipos $\leq$, la cual permite usar valores de un tipo en lugar de otro. El principio básico de esta relación es la propiedad de sustitución:

Si $\mathtt{S}$ es subtipo de $\mathtt{T}$, es decir, $\mathtt{S} \leq \mathtt{T}$, entonces cualquier expresión de tipo $\mathtt{S}$ puede emplearse sin causar un error de tipos en cualquier contexto que requiera una expresión de tipo $\mathtt{T}$. Las propiedades más importantes de la relación de subtipo son:

- Debe ser reflexiva y transitiva.

$$\frac{}{\mathtt{S} \leq \mathtt{S}} \qquad \frac{\mathtt{R} \leq \mathtt{S} \quad \mathtt{S} \leq \mathtt{T}}{\mathtt{R} \leq \mathtt{T}}$$

- Una regla de subsunción[15] la cual afirma que si $\mathtt{S} \leq \mathtt{T}$ entonces una expresión de tipo $\mathtt{S}$ puede usarse en cualquier contexto que necesite una expresión de tipo $\mathtt{T}$.

[14] La semántica denotacional o denotativa consiste en hallar una colección de dominios semánticos para después definir una función de interpretación que envia términos a elementos de dichos dominios.

[15] Según el diccionario de la lengua española[11] la palabra subsumir significa *Considerar algo como parte de un conjunto más amplio o como caso particular sometido a un principio o norma general.*

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S} \quad \mathtt{S} \leq \mathtt{T}}{\Gamma \vdash \mathtt{e} : \mathtt{T}}$$

Obsérvese que la regla de subsunción no está dirigida por la sintaxis; la expansión o promoción de la expresión `e` a un nuevo tipo `T` ocurre de manera implícita, es decir la expresión `e` no se modifica al modificar su tipo de `S` a `T`. Además la expansión implica una pérdida de información, es decir, el tipado `e : T` es mas general que `e : S`.

Este principio se formaliza mediante dos estilos de manejo de subtipos:

- **Interpretación mediante subconjuntos:**

  Si $\mathtt{S} \leq \mathtt{T}$ entonces toda expresión de tipo `S` es también una expresión de tipo `T`.

- **Interpretación mediante coerción**[16]

  Si $\mathtt{S} \leq \mathtt{T}$ entonces cualquier expresión de tipo `S` puede convertirse en forma única en una expresión de tipo `T`. Es decir, hay una coerción (forzamiento) del tipo `S` al tipo `T`. Esta coerción puede generarse en tiempo de ejecución aunque esto no es necesario. [8]

De este modo el sistema de tipos ha sido formalizado, continuaremos con la definición de Java Peso Pluma.

[16] Del inglés *coerción*. Según el diccionario de la lengua española la palabra coerción significa *presión ejercida sobre alguien para forzar su voluntad o su conducta.*

Capítulo 3

Java Peso Pluma

En los capítulos anteriores hicimos una revisión de los conceptos necesarios para profundizar en las características de los lenguajes de programación orientados a objetos, lo anterior, con el fin de dar las herramientas necesarias para una mayor comprensión de Java Peso Pluma. Ahora estudiaremos dicho lenguaje y veremos sus propiedades y características. [1]

3.1. Introducción

Java Peso Pluma (En inglés *Featherweight Java*) es un intérprete que modela las características fundamentales del sistema de tipos de Java.

La idea fundamental es buscar un lenguaje compacto en vez de uno completo, de esta forma sólo necesitamos cinco formas para las expresiones: creación de objetos, invocación a métodos, acceso a atributos, *casting* (conversión de tipos explícita) y variables. La idea de crear Java Peso Pluma(en adelante JPP) fue propuesta por Igarashi, Pierce y Wadler en 1999, su diseño es ligeramente mas grande que el Cálculo Lambda de Church, se creó bajo el postulado *todo en Java Peso Pluma es un objeto.*

Ahora bien, no existe un modelo perfecto puesto que deben abstraerse ciertos detalles, el paradigma orientado a objetos es un modelo y como tal debe seleccionar sus características:

- Código fuente vs. Código en bytes.
- Incluyentes y grandes vs. Específicos y pequeños.
- Sistemas de tipos vs. Características en ejecución
- Extensiones.

[1] Este capítulo está basado en el capítulo *19 Case Study: Featherwaight Java* del libro *Types And Programming Laguages* de Benjamin C. Pierce[10]

En nuestro caso el propósito es modelar características fundamentales de la programación orientada a objetos y su sistema de tipos. Es por ello que JPP excluye diversas características de JAVA como: clases internas, carga de clases, reflexión, concurrencia, hilos, excepciones, ciclos, interfaces, sobrecarga, y asignación (referencias).

El objetivo para el diseño de JPP fue hacer el análisis de tipos tan simple como fuera posible, capturando la esencia de seguridad de tipos de JAVA. Cualquier característica que hiciera más complicado el análisis de tipos sin proveer algo significativo en el modelo sería omitida.

El punto clave de la simplificación de JPP fue eliminar la asignación. Se asume que, los atributos se inicializan en el constructor y no cambian su valor. Este aspecto restringe JPP a un fragmento "funcional" de JAVA. Este fragmento es computacionalmente completo (es fácil identificar al λ-cálculo en él), y es lo suficientemente grande para definir programas útiles. Por ejemplo, muchos de los programas en el libro A LITTLE JAVA, A FEW PATTERNS [1] usan el estilo puramente funcional.

3.2. Vista general

En JPP, un programa se compone de una colección de definiciones de clases y una expresión a evaluar, dicha expresión corresponde al cuerpo del método *main* de JAVA. Éstas son algunas definiciones típicas de clases en JPP:

```
class A extends Object { A() {super(); }}

class B extends Object { B() {super(); }}

class Par extends Object {
    Object fst;
    Object snd;

    // Constructor:
    Par(Object fst, Object snd){
       super();
       this.fst = fst;
       this.snd =snd;
    }

    // Definición de método:
    Par setfst(Object newfst){
       return new Par(newfst, this.snd);
    }
}
```

En adelante se ocuparán algunas convenciones para asegurar la regularidad sintáctica, es decir, siempre incluiremos la superclase, aún cuando ésta sea

`Object`, siempre escribiremos el constructor aún cuando sea trivial como es el caso de las clases A y B, y en el caso de acceso a atributos o invocación a métodos siempre nombraremos al receptor (como `this.snd`), aún cuando el receptor sea `this`.

Los constructores siempre tendrán la misma forma estilizada: esto es, un parámetro por cada atributo pero obligatoriamente con un nombre diferente del atributo; se llama al constructor `super` para inicializar los atributos de la superclase; y entonces los atributos serán inicializados con respecto al atributo correspondiente.

En este caso, la superclase de las tres clases de ejemplo es `Object` la cual no tiene atributos, por tanto, la invocación a super no recibe argumentos.

El constructor es la única sección de código donde aparece `super` o `=` en los programas de JPP. Dado que JPP carece de operaciones con efecto secundario[2], el cuerpo de los métodos iniciará siempre con la palabra `return` seguido de una expresión, como en el cuerpo del método `setfst`.

Como se dijo al inicio de este capítulo, contamos con cinco formas de expresiones de JPP. `new A(), new B()` y `new Par(...)` sirven para la creación de objetos.

En el cuerpo de `setfst`, la expresión `this.snd` sirve para realizar el acceso a un atributo, donde las ocurrencias `newfst` y `this` son variables.

Ahora, si tenemos las expresiones:

```
new Par(new A(),new B()).setfst(new B())                          (1)

((Par) new Par(new Par(new A(), new B()), new A()).fst).snd       (2)
```

Entonces `(...).setfst(...)` de (1) es una invocación a un método y `(Par) new Par(...)` de (2) es una conversión explícita de tipos (*cast*). Esto es necesario ya que el acceso al atributo `fst` se evalúa al tipo de `fst`, el cual es a su vez un `Object`, y el acceso al atributo `snd` sólo está definido para los objetos de `Par`. En tiempo de compilación las reglas del analizador de tipos determinarán si es válido tomar el tipo de `snd` como `Par` (en este caso es correcto).

3.3. Sistema de tipos nominales

En el lenguaje Java, siempre se escribe el tipo al que se está haciendo referencia, por ejemplo, si tenemos un método que devuelve un tipo *String* y a su vez los parámetros del método son de tipo *int y double* respectivamente, tendremos algo como lo siguiente:

```
String miMetodo(int parametro1,double parametro2){...}
```

Claramente todos los tipos son nombrados. Lo que hace que la verificación de

[2]Como lo es la operación de asignación

tipos sea eficiente y rápida, puesto que el verificador se queda sólo con los nombres, de este modo lo único que le resta por hacer al verificador es asegurarse que los tipos de los parámetros recibidos por una llamada al método sean *(int, double)* y que el cuerpo del método devuelva un *String*.

Bien, este sistema de tipos recibe el nombre de **tipos nominales** y es el que usaremos con JAVA PESO PLUMA, por lo que, siempre escribiremos el tipo en los métodos y atributos de una clase, donde dichos tipos serán a su vez nombres de clases. Un claro ejemplo de otro sistema de tipos es el sistema de tipos estructural, el cual es usado por el lenguaje HASKELL. Las diferencias de estos dos sistemas de tipos se pueden distinguir a grandes rasgos de la siguiente forma:

- Sistema de tipos estructural:
  - Lo importante acerca de un tipo es su estructura.
  - Los nombres son simples abreviaturas.
  - El subtipado a su vez es estructural.
  - Son más simples.
  - Fáciles de extender.
  - Al considerar tipos recursivos se pierde la simplicidad.
- Sistemas de tipos nominales:
  - Los tipos siempre se nombran.
  - El verificador de tipos manipula la mayor parte del tiempo únicamente los nombres y no las estructuras, lo cual causa que la verificación sea fácil y eficiente.
  - El programador declara explícitamente la relación de subtipos, para ser verificados posteriormente por el compilador.
  - Los nombres de tipos son útiles en tiempo de ejecución, por ejemplo para realizar conversiones explícitas de tipos (*casting*), pruebas de tipos, reflexión, etc.

A continuación presentamos la sintaxis formal de JPP, cabe mencionar que la omisión de efectos imperativos de asignación causa un efecto lateral ventajoso: las únicas maneras en las que dos objetos difieren son mediante las clases de las cuales provienen o bien mediante los parámetros que son pasados a sus constructores en el momento de creación. Esta información está disponible en el operador de generación de objetos `new` de manera que podemos identificar el objeto creado con la expresión `new`, lo cual indica que los valores del sistema deben ser objetos creados mediante la palabra `new`.

3.4. Definiciones

Ahora, formalicemos la definición de JPP, la primera tarea será la definición de la sintaxis, de esta manera indicaremos las reglas con las que contará nuestro lenguaje para construir Clases y Objetos.

3.4.1. Sintaxis

La sintaxis de JPP se define como sigue:

SINTAXIS:

CL	::=	class C extends C $\{\vec{C}\ \vec{f};\ K\ \vec{M}\}$		*Declaración de clase*
K	::=	$C(\vec{C}\ \vec{f})\{\text{super}(\vec{f});\ \text{this}.\vec{f} = \vec{f};\}$		*Declaración de constructor*
M	::=	$C\ m(\vec{C}\ \vec{x})\{\text{return e};\}$		*Declaración de método*
e	::=	x	**(variable)**	*Expresiones*
		e.f	**(acceso a atributo)**	
		$e.m(\vec{e})$	**(invocación a método)**	
		new $C(\vec{e})$	**(creación de objeto)**	
		(C) e	**(cast)**	
v	::=	new $C(\vec{v})$	**(creación de objeto a partir de valores)**	*Valores*

En la definición de la sintaxis las meta-variables A, B, C, D y E son usadas para referirse a los nombres de clases; f y g hacen referencia a atributos de clase; m hace referencia al nombre del método; x hace referencia al nombre del parámetro; e es expresión; u y v son valores.

La declaración "class C extends D $\{\vec{C}\ \vec{f};\ K\ \vec{M}\}$" define una nueva clase con nombre C y superclase D. La nueva clase tiene atributos $\vec{f}$[3] cuyos tipos son $\vec{C}$, un constructor K y una sucesión de métodos $\vec{M}$. Las variables declaradas en C serán agregadas a las ya declaradas en D y en sus superclases y deben tener nombres distintos a éstas. Los métodos, por otro lado, pueden tener los nombres de los definidos en D, sobrescribiendo el método o incluso dándole una nueva funcionalidad.

La declaración del constructor

$$\text{"}C(\vec{D}\ \vec{g},\ \vec{C}\ \vec{f})\{\text{super}(\vec{g});\ \text{this}.\vec{f} = \vec{f};\}\text{"}$$

[3] Escribimos "$\vec{f}$" para simplificar la escritura de "$f_1, f_2, ..., f_n$" (igualmente se hace para: $\vec{C}$, $\vec{x}$, $\vec{e}$, etc.) y escribimos "$\vec{M}$" para simplificar "$M_1M_2...M_n$" (sin comas). Abreviamos operaciones en pares de manera similar, escribiendo "$\vec{C}\ \vec{f}$" para "$C_1\ f_1,\ C_2\ f_2,\ ...,\ C_n\ f_n$". "$\vec{C}\ \vec{f};$" para la declaración "$C_1\ f_1;\ C_2\ f_2;\ ...;\ C_n\ f_n;$" y
"$\text{this}.\vec{f} = \vec{f};$" para "$\text{this}.f_1 = f_1;\ \text{this}.f_2 = f_2;\ ...;\ \text{this}.f_n = f_n;$". Todo esto para las secuencias de declaración de atributos, nombres de parámetros y declaración de métodos, donde, asumimos que ninguna contiene nombres repetidos.

muestra cómo inicializar los atributos de una instancia de `C`, su forma es completamente determinada por la instanciación de las variables declaradas en `C` y su superclase: ésta debe tomar exactamente tantos parámetros como variables estén declaradas, así como las necesarias para instanciar a su superclase `D`.

El cuerpo del constructor debe consistir de una llamada a su superclase, instanciándola con sus respectivos parámetros $\vec{\texttt{g}}$, seguido de la inicialización de sus atributos $\vec{\texttt{f}}$.

La declaración del método

$$\text{“}\texttt{D m}(\vec{\texttt{C}}\ \vec{\texttt{x}})\{\texttt{return e;}\}\text{”}$$

introduce un método llamado `m` con resultado de tipo `D`, cuyos parámetros $\vec{\texttt{x}}$ son de tipos $\vec{\texttt{C}}$. El cuerpo del método es simplemente la expresión `return e`. Las variables $\vec{\texttt{x}}$ así como la variable especial `this` están ligadas en `e`.

Una tabla de clases TC[4] mapea los nombres de clase `C`, las definiciones de clase DC[5], lo anterior es necesario para asegurar una estructura correcta en nuestro código JPP y acceder eficientemente a las definiciones según se vayan necesitando. Un programa es una pareja $(\texttt{TC}, \texttt{e})$ de una tabla de clases y una expresión, para aligerar la notación asumiremos que la tabla TC siempre estará bien formada.

Cada clase tiene una superclase, dicha relación se declarará con la palabra `extends`, con excepción de la clase padre de todas las clases (la clase `Object`) cuya definición no cumple con esta regla.

3.4.2. Subtipado

La definición formal de subtipado se muestra a continuación:[6]

SUBTIPADO:

$$\frac{}{\texttt{C} \leq \texttt{C}}\ \textit{(Reflexividad)}$$

$$\frac{\texttt{C} \leq \texttt{D} \quad \texttt{D} \leq \texttt{E}}{\texttt{C} \leq \texttt{E}}\ \textit{(Transitividad)}$$

$$\frac{\texttt{TC(C)} = \texttt{class C extends D } \{...\}}{\texttt{C} \leq \texttt{D}}\ \textit{(Definición de clase)}$$

El subtipado es reflexivo ya que una clase es subclase de sí misma, es transitivo porque una clase `C` subclase de una clase `D` también es subclase de la clase

[4] Tabla de clases
[5] Definiciones de clases
[6] Las definiciones de éste capítulo están basadas en el libro *Types and Programming Languages* de Benjamin Pierce(páginas 255,256) [10]

padre de D, es decir, C es subclase de E, y D es subclase de E.

Dada una tabla de clases, asumiremos que ésta cumple con las siguientes cuatro reglas:

1. TC(C) = class C ... para cada clase C, tal que, C ∈ Dom(TC)
2. Object ∉ Dom(TC)
3. Para cada nombre de clase C (excepto Object) que figure en cualquier parte de la TC, tenemos que, C ∈ Dom(TC)
4. No hay ciclos en la relación de *subtipado* en la TC, lo que significa que la relación es antisimétrica

3.4.3. Definiciones auxiliares

Para definir las reglas de tipado y evaluación, necesitamos dar las siguientes definiciones auxiliares:[7]

Reglas auxiliares:

Atributos De Clase

$$\boxed{\texttt{atributos(C)} = \vec{\texttt{C}}\ \vec{\texttt{f}}}$$

$$\texttt{atributos(Object)} = \varnothing$$

$$\frac{\texttt{TC(C)} = \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{atributos(D)} = \vec{\texttt{D}}\ \vec{\texttt{g}}}{\texttt{atributos(C)} = \vec{\texttt{D}}\ \vec{\texttt{g}},\ \vec{\texttt{C}}\ \vec{\texttt{f}}}$$

La función atributos(C) regresa los atributos de la clase C, los cuales son la secuencia $\vec{\texttt{D}}\ \vec{\texttt{g}},\ \vec{\texttt{C}}\ \vec{\texttt{f}}$ que empareja a cada atributo con su respectivo tipo, para todos los atributos declarados en C y en sus superclases. Los atributos se ordenan colocando primero los atributos de la clase padre y después los de la clase pasada como parámetro a la función.

El Tipo Del Método

$$\boxed{\texttt{tipoMetodo(m, C)} = \vec{\texttt{C}} \to \texttt{C}}$$

$$\frac{\texttt{TC(C)} = \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \in \vec{\texttt{M}}}{\texttt{tipoMetodo(m, C)} = \vec{\texttt{B}} \to \texttt{B}}$$

[7] Las definiciones de éste capítulo están basadas en el libro *Types and Programming Languages* de Benjamin Pierce(página 257) [10]

$$\frac{\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \notin \vec{\texttt{M}}}{\texttt{tipoMetodo(m, C)} \;=\; \texttt{tipoMetodo(m, D)}}$$

La función `tipoMetodo(m, C)` devuelve el tipo del método `m` en la clase `C`, el cual es la secuencia $\vec{\texttt{B}}$ de tipos de sus argumentos seguida por el tipo del resultado `B`. Si el método en la clase `C` no fue definido se buscará su definición en la clase padre.

El Cuerpo Del Método

$$\boxed{\texttt{cuerpoMetodo(m, C)} \;=\; (\vec{\texttt{x}}, \texttt{e})}$$

$$\frac{\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \in \vec{\texttt{M}}}{\texttt{cuerpoMetodo(m, C)} \;=\; (\vec{\texttt{x}}, \texttt{e})}$$

$$\frac{\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \notin \vec{\texttt{M}}}{\texttt{cuerpoMetodo(m, C)} \;=\; \texttt{cuerpoMetodo(m, D)}}$$

De manera similar, la función `cuerpoMetodo(m, C)` regresa el cuerpo del método `m` en la clase `C`, el cual es una pareja compuesta en su primer elemento por la secuencia $\vec{\texttt{x}}$ de nombres de parámetros y por la expresión a evaluar `e` en su segundo elemento. Si el método en la clase `C` no fue definido se buscará su definición en la clase padre.

3.4.4. Reglas de tipado

A continuación veremos las reglas formales de tipado para expresiones, declaración de métodos y declaración de clases en JPP.[8]

Un contexto Γ es un mapeo finito de variables a tipos, escrito $\texttt{x} : \texttt{C}$.

Las expresiones se tipan mediante la relación $\Gamma \vdash \texttt{e} : \texttt{C}$ y se lee: "en el contexto Γ la expresión `e` tiene tipo `C`" y existe una regla para cada expresión, excepto para la conversión explícita de tipos, para la cual hay tres reglas que explicaremos mas adelante. Las reglas de tipado para constructores e invocación a métodos buscan que cada argumento tenga un tipo que sea subtipo del usado en las declaraciones, esta es la propiedad llamada **herencia**.

Abreviaremos las secuencias de tipos de la siguiente forma: $\Gamma \vdash \vec{\texttt{e}} : \vec{\texttt{C}}$ para $\Gamma \vdash \texttt{e}_1 : \texttt{C}_1, ..., \Gamma \vdash \texttt{e}_\texttt{n} : \texttt{C}_\texttt{n}$ y $\vec{\texttt{C}} \leq \vec{\texttt{D}}$ para $\texttt{C}_1 \leq \texttt{D}_1, ..., \texttt{C}_\texttt{n} \leq \texttt{D}_\texttt{n}$.

Las definiciones dependen de una tabla de clases particular que está implícita. En las reglas de evaluación escribiremos $\vdash_\texttt{T}$ para hacerla explícita.

[8] Las definiciones de éste capítulo están basadas en el libro *Types and Programming Languages* de Benjamin Pierce(página 259) [10]

Tipado de expresiones:

$$\boxed{\Gamma \vdash \mathtt{e} : \mathtt{C}}$$

- **Variables:** el tipo debe ser declarado en un contexto. Esto es, cuando sea declarado en una clase, la variable será tipada. Dentro de esa clase la variable mantendrá ese tipo, pero sólo en esa clase, es decir, sólo en ese contexto. El símbolo `x` representa el nombre de una variable, la cual podría ser `this`.

$$\frac{}{\Gamma, \mathtt{x} : \mathtt{C} \vdash \mathtt{x} : \mathtt{C}} \quad (\text{T-VAR})$$

- **Acceso a atributos:** el tipo de acceso a un atributo es aquel que le fue impuesto al atributo en la definición de la clase de la que es tipo la expresión.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathtt{atributos}(\mathtt{C}) = \vec{\mathtt{C}}\ \vec{\mathtt{f}}}{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C_i}} \quad (\text{T-ATR})$$

- **Invocación de métodos:** el tipo de una invocación de método es el tipo de su resultado (el tipo de la expresión después de la palabra *return*), sin embargo, es necesario verificar que los tipos de cada uno de sus parámetros sea subtipo del impuesto en la definición del método.

$$\frac{\Gamma \vdash \mathtt{e_0} : \mathtt{C_0} \qquad \Gamma \vdash \vec{\mathtt{e}} : \vec{\mathtt{C}} \qquad \mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C_0}) = \vec{\mathtt{D}} \to \mathtt{C} \qquad \vec{\mathtt{C}} \leq \vec{\mathtt{D}}}{\Gamma \vdash \mathtt{e_0.m}(\vec{\mathtt{e}}) : \mathtt{C}}$$
$$(\text{T-INVMET})$$

- **Creación de objetos:** el tipo de un objeto es el de la clase a la que pertenece, sin embargo, es necesario verificar que los tipos de cada uno de sus parámetros sea subtipo del impuesto en la definición del constructor.

$$\frac{\mathtt{atributos}(\mathtt{C}) = \vec{\mathtt{D}}\ \vec{\mathtt{f}} \qquad \Gamma \vdash \vec{\mathtt{e}} : \vec{\mathtt{C}} \qquad \vec{\mathtt{C}} \leq \vec{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\vec{\mathtt{e}}) : \mathtt{C}} \quad (\text{T-NEW})$$

- **Conversión de tipos explícita hacia arriba:** cambia el tipo de una expresión por una superclase de la clase original.

$$\frac{\Gamma \vdash \mathtt{e_0} : \mathtt{D} \qquad \mathtt{D} \leq \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e_0} : \mathtt{C}} \quad (\text{T-CAST-AR})$$

- **Conversión de tipos explícita hacia abajo:** cambia el tipo de una expresión por una subclase de la clase original.

$$\frac{\Gamma \vdash \mathtt{e_0} : \mathtt{D} \qquad \mathtt{C} \leq \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e_0} : \mathtt{C}} \quad (\text{T-CAST-AB})$$

- **Conversión de tipos explícita estúpida**[9]**:** cambia el tipo de una expresión, aún cuando no hay relación en los tipos. Esto en JAVA se considera un caso mal tipado, sin embargo, es necesario que permitamos estos casos si queremos probar el teorema de correctez con una semántica operacional estructural o de paso pequeño. Pues podría ocurrir un caso en el que una expresión que contiene un conversión de tipos explícita estúpida pueda ser reducida ((A)(Object)new B() $\rightarrow_{\mathtt{T}}$ (A)new B()). Indicamos este caso especial mediante una advertencia estúpida hipotética; el tipado de JPP corresponde al tipado legal de JAVA sólo si no es necesaria esta regla.

$$\frac{\Gamma \vdash \mathtt{e_0} : \mathtt{D} \qquad \mathtt{C} \not\leq \mathtt{D} \qquad \mathtt{D} \not\leq \mathtt{C} \quad \textit{advertencia estupida}}{\Gamma \vdash (\mathtt{C})\mathtt{e_0} : \mathtt{C}} \quad \text{(T-CAST-EST)}$$

TIPADO DE MÉTODOS:

$$\boxed{\mathtt{M\ OK\ en\ C}}$$

- Asignar un tipo en el caso de declaración del método tiene la forma `M OK en C` y se lee: "La declaración del método `M` está bien formada si esta figura en la clase `C`". Se hace usando el tipado de expresiones en el cuerpo del método, teniendo en el contexto variables libres como parámetros del método junto con los tipos impuestos en la definición y la variable `this` con tipo `C`, además de que se verifica el tipado del método en la superclase.

$$\frac{\begin{array}{c}\mathtt{TC(C)} \;=\; \mathtt{class\ C\ extends\ D\{...\}} \\ \mathtt{tipoMetodo(m, D)} = \vec{\mathtt{C}} \rightarrow \mathtt{C_0} \\ \vec{\mathtt{x}} : \vec{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e_0} : \mathtt{C_0'} \\ \mathtt{C_0'} \leq \mathtt{C_0}\end{array}}{\mathtt{C_0\ m}(\vec{\mathtt{C}}\ \vec{\mathtt{x}})\{\mathtt{return\ e_0;}\}\ \mathtt{OK\ en\ C}}$$

En las premisas se está considerando la posibilidad de que el método `m` se haya redefinido cuidando que se respete el tipo dado en la superclase, tal vez mediante subtipado. Es decir, el cuerpo del método en la subclase (en este caso $\mathtt{e_0}$) debe regresar un subtipo, en este caso $\mathtt{C_0'}$ del tipo del resultado del mismo método en la superclase. También se observa que los tipos de los parámetros del método deben ser exactamente los mismos que los de la superclase.
De importancia es observar que si `D = Object` entonces el método no se está redefiniendo puesto que `Object` no tiene métodos. En tal caso la premisa acerca de `tipoMetodo` debe ignorarse, siendo así aceptadas cualesquiera $\mathtt{C_0}$ y $\vec{\mathtt{C}}$.

TIPADO DE CLASES:

$$\boxed{\mathtt{C\ OK}}$$

[9]Este es el nombre oficial de la regla, *stupid cast*, ver el libro [10]

- Realizar el tipado para el caso de la declaración de clase que tiene la forma `C OK` y se lee: "La declaración de la clase `C` está bien formada".

  Aseguramos que la aplicación de `super` por parte del constructor es correcta con los tipos definidos en la superclase y de que la inicialización de los atributos se encuentra correctamente tipada. Además, los métodos de la clase cumplen con `M OK en C`.

$$\frac{\texttt{K} \;=\; \texttt{C}(\vec{\texttt{D}}\ \vec{\texttt{g}}, \vec{\texttt{C}}\ \vec{\texttt{f}})\{\texttt{super}(\vec{\texttt{g}});\ \texttt{this}.\vec{\texttt{f}} = \vec{\texttt{f}};\} \qquad \texttt{atributos}(\texttt{D}) \;=\; \vec{\texttt{D}}\ \vec{\texttt{g}} \qquad \vec{\texttt{M}}\ \texttt{OK en C}}{\texttt{class C extends D}\{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\}\ \texttt{OK}}$$

3.4.5. Semántica Operacional

Existen tres estilos básicos para definir la semántica dinámica de un lenguaje de programación: semántica denotacional, semántica axiomática y semántica operacional. Para nuestro lenguaje usaremos este último estilo.[10]

En la semántica operacional el significado de una instrucción de programa se especifica mediante el cómputo que induce en una máquina. En particular resulta de interés cómo se produce el resultado o efecto de un cómputo. El comportamiento del programa se modela definiendo una *máquina abstracta* en el sentido de que sus estados o código de máquina son entidades abstractas como las mismas expresiones del lenguaje. Para lenguajes simples un estado es una expresión y el comportamiento de la máquina se define mediante una relación o sistema de transición que devuelve el siguiente estado, el cual se obtiene al desarrollar una simplificación o evaluación o bien declarando que la máquina se ha detenido. El significado de la expresión `e` es entonces el estado final alcanzado por la máquina al iniciar su funcionamiento tomando a `e` como estado inicial. Esto se conoce como semántica estructural o de paso pequeño.

A continuación formalizaremos la evaluación en nuestro lenguaje usando esta semántica.

3.4.6. Reglas de evaluación

[11]

Para definir las reglas de evaluación usaremos semántica operacional estructural, es decir, crearemos reglas dinámicas las cuales serán usadas en tiempo de ejecución para pasar de un estado dado a otro sin romper con la estructura de nuestras definiciones. La aplicación de estas reglas llega a su fin cuando se obtiene un valor. Recordemos que un valor en JPP está dado por una expresión que crea un nuevo objeto: `new` $\texttt{C}(\vec{\texttt{v}})$ donde $\vec{\texttt{v}}$ indica que todos los parámetros

[10] Esta sección está basada en la Nota de clase 4, 2010-2 de Favio E. Miranda. [8]

[11] Las definiciones de éste capítulo están basadas en el libro *Types and Programming Languages* de Benjamin Pierce(página 258) [10]

del nuevo objeto ya son valores. Las reglas se definen para tres casos: acceso a atributos, invocación de métodos y conversión explícita de tipos.

Definimos la relación $\mathtt{e} \rightarrow_{\mathtt{T}} \mathtt{e}'$ cuyo significado es: La expresión $\mathtt{e}$ se evalúa a la expresión $\mathtt{e}'$ en un paso, según la tabla $\mathtt{T}$.

EVALUACIÓN:

$$\boxed{\mathtt{e} \rightarrow_{\mathtt{T}} \mathtt{e}'}$$

- **Acceso a atributos:** puesto que los parámetros del constructor corresponden exactamente con los atributos de clase, una instancia de la clase contiene toda la información necesaria para determinar los valores de esta instancia. De manera que una instancia es esencialmente una colección etiquetada de atributos.

$$\frac{\mathtt{atributos}(\mathtt{C}) = \vec{\mathtt{C}}\ \vec{\mathtt{f}}}{\mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}}).\mathtt{f_i} \rightarrow_{\mathtt{T}} \mathtt{v_i}} \quad \text{(E-ATRNEW)}$$

  La selección de atributos recupera el valor correspondiente de la superclase o de la subclase. [12]

- **Invocación de métodos:** cada vez que un método recibe como parámetros valores se cuenta con la información suficiente para sustituir las apariciones de los nombres de variables por los valores recibidos, así como la aparición de la variable `this` por la instancia misma.

$$\frac{\mathtt{cuerpoMetodo}(\mathtt{m}, \mathtt{C}) = (\vec{\mathtt{x}}, \mathtt{e_0})}{\mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}}).\mathtt{m}(\vec{\mathtt{u}}) \rightarrow_{\mathtt{T}} \mathtt{e_0}[\vec{\mathtt{x}} := \vec{\mathtt{u}}, \mathtt{this} := \mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}})]} \quad \text{(E-INVNEW)}$$

- **Conversión de tipos explícita:** el comportamiento en tiempo de ejecución de la operación de conversión de tipos es en realidad un prueba para asegurar que el tipo actual es una subclase del tipo declarado en la conversión. En ese caso ésta se quita y el resultado es el objeto mismo. Lo cual corresponde exactamente a la semántica de JAVA: en tiempo de ejecución la operación de conversión de tipos explícita no cambia un objeto en ninguna forma— simplemente se permite y continúa, o bien falla y se manda una excepción. En el caso de JPP, la evaluación se detendría al no haber ningún caso para continuar.

$$\frac{\mathtt{C} \leq \mathtt{D}}{(\mathtt{D})(\mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}})) \rightarrow_{\mathtt{T}} \mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}})} \quad \text{(E-CASTNEW)}$$

  Obsérvese que ésta es una regla de evaluación condicional que causa dificultades en la implementación del sistema, pues delega la verificación de subclase al tiempo de ejecución.

[12] Esto puede hacerse explícito cambiando la regla anterior, sin embargo no es necesario.

- **Orden de evaluación:** en caso de no contar aún con un valor, se pasa al siguiente estado de la expresión. Esto se hace de izquierda a derecha, según las siguientes reglas.

$$\frac{\mathtt{e}_0 \to_{\mathtt{T}} \mathtt{e}'_0}{\mathtt{e}_0.\mathtt{f} \to_{\mathtt{T}} \mathtt{e}'_0.\mathtt{f}} \quad \text{(E-ACCATR)}$$

$$\frac{\mathtt{e}_0 \to_{\mathtt{T}} \mathtt{e}'_0}{\mathtt{e}_0.\mathtt{m}(\vec{\mathtt{e}}) \to_{\mathtt{T}} \mathtt{e}'_0.\mathtt{m}(\vec{\mathtt{e}})} \quad \text{(E-INVMET)}$$

$$\frac{\mathtt{e}_\mathtt{i} \to_{\mathtt{T}} \mathtt{e}'_\mathtt{i}}{\mathtt{v}_0.\mathtt{m}(\vec{\mathtt{v}}, \mathtt{e}_\mathtt{i}, \vec{\mathtt{e}}) \to_{\mathtt{T}} \mathtt{v}_0.\mathtt{m}(\vec{\mathtt{v}}, \mathtt{e}'_\mathtt{i}, \vec{\mathtt{e}})} \quad \text{(E-INVMET-ARGS)}$$

$$\frac{\mathtt{e}_\mathtt{i} \to_{\mathtt{T}} \mathtt{e}'_\mathtt{i}}{\mathtt{new\ C}(\vec{\mathtt{v}}, \mathtt{e}_\mathtt{i}, \vec{\mathtt{e}}) \to_{\mathtt{T}} \mathtt{new\ C}(\vec{\mathtt{v}}, \mathtt{e}'_\mathtt{i}, \vec{\mathtt{e}})} \quad \text{(E-NEW-ARGS)}$$

$$\frac{\mathtt{e}_0 \to_{\mathtt{T}} \mathtt{e}'_0}{(\mathtt{C})\mathtt{e}_0 \to_{\mathtt{T}} (\mathtt{C})\mathtt{e}'_0} \quad \text{(E-CAST)}$$

Estas reglas muestran los instantes en tiempo de ejecución, avanzando un estado a la vez.

3.5. Propiedades Fundamentales

Es de esperar que los teoremas de preservación y progreso requieran de restricciones adicionales respecto a sus análogos en otros sistemas. Por ejemplo ahora debemos considerar la herencia así como el uso de una conversión de tipos explícita no válida.

En general la seguridad por tipos expresa la coherencia entre la semántica estática y la semántica dinámica. La semántica estática predice que el valor de una expresión tendrá cierta forma de tal modo que la semántica de dicha expresión se encuentra bien definida. En consecuencia, la evaluación no puede bloquearse en un estado no final para el cual no haya transición posible, lo cual corresponde en una implementación a la ausencia de errores por instrucción ilegal en tiempo de ejecución. Esto se prueba mostrando que cada paso de transición preserva el tipado y mostrando que los estados donde es posible asignar un tipo están bien definidos.

La seguridad consta de dos aspectos, el progreso de la evaluación y la preservación de tipos:

- Preservación: relaciona la evaluación y el tipado, si $\Gamma \vdash \mathtt{e} : \mathtt{T}$ y $\mathtt{e} \to \mathtt{e}'$ entonces $\Gamma \vdash \mathtt{e}' : \mathtt{T}$, es decir si un programa tipado correctamente toma un paso de evaluación entonces el resultado es un programa correctamente tipado (y generalmente con el mismo tipo).
- Progreso: relaciona el tipado y las formas normales, si $\Gamma \vdash \mathtt{e} : \mathtt{T}$ entonces $\mathtt{e}$ es un valor o $\mathtt{e} \to \mathtt{e}'$ para algún $\mathtt{e}'$, es decir, en un programa correctamente tipado la evaluación progresa hasta terminar.

Ahora procederemos a probar que JPP es seguro probando que preserva los tipos y termina. Para ello probaremos primero los siguientes lemas:

Lema 1. *Si* $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{D}) = \vec{\mathtt{C}} \to \mathtt{C}_0$*, entonces* $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}) = \vec{\mathtt{C}} \to \mathtt{C}_0$ *para toda* $\mathtt{C} \leq \mathtt{D}$.

Prueba. Inducción sobre $\mathtt{C} \leq \mathtt{D}$.

Caso base: Reflexividad. Tenemos $\mathtt{C} = \mathtt{D}$, por lo tanto la afirmación del lema es obvia.

Paso inductivo:

1. **Transitividad**. $\mathtt{C} \leq \mathtt{D}$ se obtuvo de $\mathtt{C} \leq \mathtt{E}$ y $\mathtt{E} \leq \mathtt{D}$.
   Sea $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{D}) = \vec{\mathtt{C}} \to \mathtt{C}_0$

   Por demostrar $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}) = \vec{\mathtt{C}} \to \mathtt{C}_0$

   Por hipótesis de inducción, como $\mathtt{E} \leq \mathtt{D}$ entonces $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{E}) = \vec{\mathtt{C}} \to \mathtt{C}_0$, por lo tanto, nuevamente por hipótesis de inducción con $\mathtt{C} \leq \mathtt{E}$, se tiene que $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}) = \vec{\mathtt{C}} \to \mathtt{C}_0$.

2. **Definición de clase**. $\mathtt{C} \leq \mathtt{D}$ se obtuvo de
   $\mathtt{TC}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}$

   Por demostrar $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}) = \vec{\mathtt{C}} \to \mathtt{C}_0$.

- **caso 2.1** $\mathtt{m}$ está definido en $\mathtt{C}$ digamos $\mathtt{m}$ como $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}) = \vec{\mathtt{B}} \to \mathtt{B}_0$ como se cumple $\mathtt{m\ OK\ en\ C}$ y sabemos que $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{D}) = \vec{\mathtt{C}} \to \mathtt{C}_0$ entonces $\vec{\mathtt{B}} = \vec{\mathtt{C}}$ y $\mathtt{C}_0 = \mathtt{B}_0$.
- **caso 2.2** $\mathtt{m}$ no está definido en $\mathtt{C}$ en tal caso
  $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}) = \mathtt{tipoMetodo}(\mathtt{m}, \mathtt{D}) = \vec{\mathtt{C}} \to \mathtt{C}_0$

□

De este modo demostramos que cualquier par de clases en nuestro lenguaje que se encuentren relacionadas mediante la herencia, cumplen con preservar su tipo en la definición de un método.

Lema 2 (Preservación de tipos bajo sustitución)**.** *Si* $\Gamma, \vec{x} : \vec{B} \vdash e : D$ *y* $\Gamma \vdash \vec{s} : \vec{A}$ *donde* $\vec{A} \leq \vec{B}$*, entonces,* $\Gamma \vdash e[\vec{x} := \vec{s}] : C$ *para alguna* $C \leq D$*.*

Prueba. Inducción sobre $\Gamma, \vec{x} : \vec{B} \vdash e : D$

Caso (T-VAR): $e = x \quad x : D \in \Gamma$.

Si $x \notin \vec{x}$, entonces el resultado es trivial puesto que $x[\vec{x} := \vec{s}] = x$. Por otro lado, si $x = x_i$ y $D = B_i$, entonces, se tiene que $x[\vec{x} := \vec{s}] = s_i$, dejando que $C = A_i$ termina el caso.

Caso (T-ATR): Tenemos $e = e_0.f_i \quad$ con $\quad \Gamma, \vec{x} : \vec{B} \vdash e_0 : D_0$, $\texttt{atributos}(D_0) = \vec{C}\ \vec{f} \quad$ y $\quad D = C_i$.

Por hipótesis de inducción, existe alguna C_0 tal que $\Gamma \vdash e_0[\vec{x} := \vec{s}] : C_0$ y $C_0 \leq D_0$. Es fácil probar que $\texttt{atributos}(C_0) = (\texttt{atributos}(D_0), \vec{D}\ \vec{g})$ para algunas $\vec{D}\ \vec{g}$. Por tanto, por T-ATR, $\Gamma \vdash (e_0[\vec{x} := \vec{s}]).f_i : C_i$, y terminamos dado que $C_i \leq C_i$

Caso (T-INVMET): Tenemos $e = e_0.m(\vec{e}) \quad$ con $\quad \Gamma, \vec{x} : \vec{B} \vdash e_0 : D_0$, $\texttt{tipoMetodo}(m, D_0) = \vec{E} \to D \quad$, $\Gamma, \vec{x} : \vec{B} \vdash \vec{e} : \vec{D} \quad$ y $\quad \vec{D} \leq \vec{E}$.

Por hipótesis de inducción, existen algunas C_0 y $\vec{C}$ tales que: $\Gamma \vdash e_0[\vec{x} := \vec{s}] : C_0$ con $C_0 \leq D_0$ y $\Gamma \vdash \vec{e}[\vec{x} := \vec{s}] : \vec{C}$ con $\vec{C} \leq \vec{D}$.
Por el lema 1, $\texttt{tipoMetodo}(m.C_0) = \vec{E} \to D$. Más aún, $\vec{C} \leq \vec{E}$ por transitividad de $\leq$. Por lo tanto, por (T-INVMET), $\Gamma \vdash e_0[\vec{x} := \vec{s}].m(\vec{e}[\vec{x} := \vec{s}]) : D$.

Caso (T-NEW): Tenemos $e = \texttt{new}\ D(\vec{e}) \quad$ con $\quad \texttt{atributos}(D) = \vec{D}\ \vec{f}$, $\Gamma, \vec{x} : \vec{B} \vdash \vec{e} : \vec{C} \quad$ y $\quad \vec{C} \leq \vec{D}$.

Por hipótesis de inducción, $\Gamma \vdash \vec{e}[\vec{x} := \vec{s}] : \vec{E}$ para algunas $\vec{E} \leq \vec{C}$. Tenemos que $\vec{E} \leq \vec{D}$ por transitividad de $\leq$. Por lo tanto, por (T-NEW), $\Gamma \vdash \texttt{new}\ D(\vec{e}[\vec{x} := \vec{s}]) : D$.

Caso (T-CAST-AR): Tenemos $e = (D)e_0 \quad$ con $\quad \Gamma, \vec{x} : \vec{B} \vdash e_0 : C \quad$ y $\quad C \leq D$

Por hipótesis de inducción, existe alguna E tal que $\Gamma \vdash e_0[\vec{x} := \vec{s}] : E$ y $E \leq C$. Tenemos $E \leq D$ por transitividad de $\leq$. Por lo tanto, por (T-CAST-AR)
$\Gamma \vdash (D)(e_0[\vec{x} := \vec{s}]) : D$.

Caso (T-CAST-AB): Tenemos $e = (D)e_0 \quad$ con $\quad \Gamma, \vec{x} : \vec{B} \vdash e_0 : C, \quad D \leq C \quad$ y $\quad D \neq C$

Por hipótesis de inducción, existe alguna $\mathtt{E}$ tal que $\Gamma \vdash \mathtt{e_0}[\vec{\mathtt{x}} := \vec{\mathtt{s}}] : \mathtt{E}$ y $\mathtt{E} \leq \mathtt{C}$. Si $\mathtt{E} \leq \mathtt{D}$ o $\mathtt{D} \leq \mathtt{E}$, entonces $\Gamma \vdash (\mathtt{D})(\mathtt{e_0}[\vec{\mathtt{x}} := \vec{\mathtt{s}}]) : \mathtt{D}$ por (T-CAST-AR) o (T-CAST-AB) respectivamente.

Por otro lado, si ambos $\mathtt{D} \not\leq \mathtt{E}$ y $\mathtt{E} \not\leq \mathtt{D}$, entonces $\Gamma \vdash (\mathtt{D})(\mathtt{e_0}[\vec{\mathtt{x}} := \vec{\mathtt{s}}]) : \mathtt{D}$ (con advertencia estúpida) por (T-CAST-EST).

Caso (T-CAST-EST): Tenemos $\mathtt{e} = (\mathtt{D})\mathtt{e_0}$ con $\Gamma, \vec{\mathtt{x}} : \vec{\mathtt{B}} \vdash \mathtt{e_0} : \mathtt{C}$, $\mathtt{D} \not\leq \mathtt{C}$ y $\mathtt{C} \not\leq \mathtt{D}$

Por hipótesis de inducción, existe alguna $\mathtt{E}$ tal que $\Gamma \vdash \mathtt{e_0}[\vec{\mathtt{x}} := \vec{\mathtt{s}}] : \mathtt{E}$ y $\mathtt{E} \leq \mathtt{C}$. Esto significa que $\mathtt{E} \not\leq \mathtt{D}$ y $\mathtt{D} \not\leq \mathtt{E}$. (Si tuviéramos $\mathtt{D} \leq \mathtt{E}$ entonces también $\mathtt{D} \not\leq \mathtt{C}$ lo cual no es cierto, por lo tanto $\mathtt{D} \not\leq \mathtt{E}$. Si tuviéramos $\mathtt{E} \leq \mathtt{D}$ entonces $\mathtt{C} \leq \mathtt{D}$ o $\mathtt{D} \leq \mathtt{C}$ lo cual no es cierto, por lo tanto $\mathtt{E} \not\leq \mathtt{D}$. Así, $\Gamma \vdash (\mathtt{D})(\mathtt{e_0}[\vec{\mathtt{x}} := \vec{\mathtt{s}}]) : \mathtt{D}$ (con advertencia estúpida) por (T-CAST-EST).

□

Sustituir una variable dentro de una expresión (según la demostración de nuestro lema) no modifica el tipo de la expresión según el contexto y si lo hace, lo hace usando un subtipo del que originalmente tenía nuestra expresión. De este modo se preservan los tipos en la sustitución.

Lema 3 (Debilitamiento (*weakening*))**.** *Si* $\Gamma \vdash \mathtt{e} : \mathtt{C}$ *, entonces,* $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e} : \mathtt{C}$.

Prueba. Inducción sobre $\Gamma \vdash \mathtt{e} : \mathtt{C}$

Caso (T-VAR)

Tenemos $\mathtt{e} = \mathtt{y}$ con $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{y} : \mathtt{C}$ y $\mathtt{y} : \mathtt{C} \in \Gamma$. El resultado es trivial pues $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{y} : \mathtt{C}$ por (T-VAR).

Caso (T-ATR)

Tenemos $\mathtt{e} = \mathtt{e_0.f_i}$ con $\Gamma \vdash \mathtt{e_0.f_i} : \mathtt{C_i}$, $\Gamma \vdash \mathtt{e_0} : \mathtt{D_0}$, $\mathtt{C}\ \mathtt{f_i} \in \mathtt{atributos}(\mathtt{D_0})$. Entonces, $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e_0} : \mathtt{D_0}$ por hipótesis de inducción, y como $\mathtt{atributos}(\mathtt{D_0}) = \vec{\mathtt{C}}\ \vec{\mathtt{f}}$ entonces $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e_0.f_i} : \mathtt{C}$ por (T-ATR).

Caso (T-INVMET)

Tenemos $\mathtt{e} = \mathtt{e_0.m}(\vec{\mathtt{e}})$ con $\Gamma \vdash \mathtt{e_0} : \mathtt{D_0}$, $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{D_0}) = \vec{\mathtt{E}} \to \mathtt{C}$, $\Gamma \vdash \vec{\mathtt{e}} : \vec{\mathtt{D}}$ con $\vec{\mathtt{D}} \leq \vec{\mathtt{E}}$. Sea $\Gamma \vdash \mathtt{e_0.m}(\vec{\mathtt{e}}) : \mathtt{C}$, entonces por hipótesis de inducción $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e_0} : \mathtt{D_0}$ y como $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{D_0}) = \vec{\mathtt{E}} \to \mathtt{C}$ entonces $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e_0.m}(\vec{\mathtt{e}}) : \mathtt{C}$ por (T-INVMET).

Caso (T-NEW)

Tenemos $\mathtt{e} = \mathtt{new\ C}(\vec{\mathtt{e}})$ con $\Gamma \vdash \mathtt{new\ C}(\vec{\mathtt{e}}) : \mathtt{C}$, $\mathtt{atributos}(\mathtt{C}) = \vec{\mathtt{D}}\ \vec{\mathtt{f}}$, $\mathtt{Gamma} \vdash \vec{\mathtt{e}} : \vec{\mathtt{C}}$, $\vec{\mathtt{C}} \leq \vec{\mathtt{D}}$. Sabemos por hipótesis de inducción que $\Gamma, \mathtt{x} : \mathtt{B} \vdash \vec{\mathtt{e}} : \vec{\mathtt{C}}$ lo que implica que $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{new\ C}(\vec{\mathtt{e}}) : \mathtt{C}$ por (T-NEW).

Caso (T-CAST-AR)

Tenemos $\mathtt{e} = (\mathtt{D})\mathtt{e}_0$, con $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}$, $\mathtt{C} \leq \mathtt{D}$. Por hipótesis de inducción $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e}_0 : \mathtt{C}$ lo que implica que $\Gamma, \mathtt{x} : \mathtt{B} \vdash (\mathtt{D})\mathtt{e}_0 : \mathtt{D}$ por (T-CAST-AR).

Caso (T-CAST-AB)
Tenemos $\mathtt{e} = (\mathtt{D})\mathtt{e}_0$, con $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}$, $\mathtt{D} \leq \mathtt{C}$, $\mathtt{D} \neq \mathtt{C}$.
Por hipótesis de inducción $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e}_0 : \mathtt{C}$ lo que implica que $\Gamma, \mathtt{x} : \mathtt{B} \vdash (\mathtt{D})\mathtt{e}_0 : \mathtt{D}$ por (T-CAST-AB).

Caso (T-CAST-EST)
Tenemos $\mathtt{e} = (\mathtt{D})\mathtt{e}_0$, con $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}$, $\mathtt{D} \not\leq \mathtt{C}$ $\mathtt{C} \not\leq \mathtt{D}$.
Por hipótesis de inducción $\Gamma, \mathtt{x} : \mathtt{B} \vdash \mathtt{e}_0 : \mathtt{C}$ lo que implica que $\Gamma, \mathtt{x} : \mathtt{B} \vdash (\mathtt{D})\mathtt{e}_0 : \mathtt{D}$ (con advertencia estúpida) por (T-CAST-EST).

□

Agregar nuevas variables a nuestro contexto no afectan a los tipos ya inferidos por éste, por tanto se preservan los tipos al crecer el contexto.

Lema 4. *Si* $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}_0) = \vec{\mathtt{D}} \to \mathtt{D}$ *y* $\mathtt{cuerpoMetodo}(\mathtt{m}, \mathtt{C}_0) = (\vec{\mathtt{x}}, \mathtt{e})$ *entonces para alguna* $\mathtt{D}_0$ *con* $\mathtt{C}_0 \leq \mathtt{D}_0$ *existe alguna* $\mathtt{C} \leq \mathtt{D}$ *tal que* $\vec{\mathtt{x}} : \vec{\mathtt{D}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e} : \mathtt{C}$.

Prueba. Inducción sobre $\mathtt{cuerpoMetodo}(\mathtt{m}, \mathtt{C}_0)$

Caso base: $\mathtt{m}$ está definido en $\mathtt{C}_0$, digamos como $\mathtt{M} = \mathtt{D}\ \mathtt{m}(\vec{\mathtt{D}}\ \vec{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e};\}$. Como la tabla de clases está bien formada entonces se cumple $\mathtt{M}\ \mathtt{OK}\ \mathtt{en}\ \mathtt{C}_0$. En particular como $\mathtt{cuerpoMetodo}(\mathtt{m}, \mathtt{C}_0) = (\vec{\mathtt{x}}, \mathtt{e})$ entonces $\vec{\mathtt{x}} : \vec{\mathtt{D}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathtt{e} : \mathtt{C}$ con $\mathtt{C} \leq \mathtt{D}$, que es lo que necesitábamos.

Paso inductivo: $\mathtt{m}$ no está definida en $\mathtt{C}_0$, Tenemos
$\mathtt{TC}(\mathtt{C}_0) = \mathtt{class}\ \mathtt{C}_0\ \mathtt{extends}\ \mathtt{E}\ \{\ldots\}, \mathtt{tipoMetodo}(\mathtt{m}, \mathtt{C}_0) = \mathtt{tipoMetodo}(\mathtt{m}, \mathtt{E})$. Por hipótesis de inducción como $\mathtt{tipoMetodo}(\mathtt{m}, \mathtt{E}) = \vec{\mathtt{D}} \to \mathtt{D}$ y
$\mathtt{cuerpoMetodo}(\mathtt{m}, \mathtt{E}) = (\vec{\mathtt{x}}, \mathtt{e})$. se tiene $\vec{\mathtt{x}} : \vec{\mathtt{D}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e} : \mathtt{C}$ con $\mathtt{C} \leq \mathtt{D}$ y $\mathtt{E} \leq \mathtt{D}_0$ para $\mathtt{C}_0 \leq \mathtt{E} \leq \mathtt{D}_0$ y terminamos.

□

3.5.1. Preservación de tipos

Proposición 1 (Preservación de tipos)**.** *Si* T *es una tabla de clases bien formada,* $\Gamma \vdash \mathtt{e} : \mathtt{C}$ *y* $\mathtt{e} \to_{\mathtt{T}} \mathtt{e}'$ *entonces existe* $\mathtt{C}'$ *tal que* $\mathtt{C}' \leq \mathtt{C}$ *y* $\Gamma \vdash \mathtt{e}' : \mathtt{C}'$.

Prueba. Por inducción sobre $\mathtt{e} \to_{\mathtt{T}} \mathtt{e}'$.

Caso (E-ATRNEW)
Tenemos $\mathtt{e} = \mathtt{new}\ \mathtt{C}_0(\vec{\mathtt{v}}).\mathtt{f}_\mathtt{i}$, $\mathtt{e}' = \mathtt{v}_\mathtt{i}$ y $\mathtt{atributos}(\mathtt{C}_0) = \vec{\mathtt{D}}\ \vec{\mathtt{f}}$. De la forma de $\mathtt{e}$, vemos que la regla final en la derivación de $\Gamma \vdash \mathtt{e} : \mathtt{C}$ debe ser (T-ATR) con premisa $\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\vec{\mathtt{v}}) : \mathtt{D}_0$, y que $\mathtt{C} = \mathtt{D}_\mathtt{i}$.
De manera similar, la última regla en la derivación de $\Gamma \vdash \mathtt{new}\ \mathtt{C}_0(\vec{\mathtt{v}}) : \mathtt{D}_0$ debe

ser
(T-NEW) con las premisas $\Gamma \vdash \vec{v} : \vec{C}$ con $\vec{C} \leq \vec{D}$ y con $D_0 = C_0$. En particular, $\Gamma \vdash v_i : C_i$, finalizando el caso pues $C_i \leq D_i$.

Caso (E-INVNEW)
Tenemos $e = (\texttt{new } C_0(\vec{v})).m(\vec{u})$, $e' = e_0[\vec{u} := \vec{x}, \texttt{new } C_0(\vec{v}) := \texttt{this}]$ y $\texttt{cuerpoMetodo}(m, C_0) = (\vec{x}, e_0)$. Las últimas reglas de la derivación en $\Gamma \vdash e : C$ son (T-INVMET) y (T-NEW), con premisas $\Gamma \vdash \texttt{new } C_0(\vec{v}) : C_0$, $\Gamma \vdash \vec{v} : \vec{C}$, $\vec{C} \leq \vec{D}$, $\texttt{tipoMetodo}(m, C_0) = \vec{D} \to C$. Por el lema 4, tenemos que $\vec{x} : \vec{D}, \texttt{this} : D_0 \vdash e_0 : B$ para algunas D_0 y B con $C_0 \leq D_0$ y $B \leq C$.
Por el lema 3, $\Gamma, \vec{x} : \vec{D}, \texttt{this} : D_0 \vdash e_0 : B$. Entonces, por el lema 2, $\Gamma \vdash e_0[\vec{x} := \vec{u}, \texttt{this} := \texttt{new } C_0(\vec{v})] : E$ para alguna $E \leq B$. Por transitividad de $\leq$, obtenemos que $E \leq C$ dando $C' = E$ completando el caso.

Caso (E-CASTNEW)
Tenemos $e = (D)(\texttt{new } C_0(\vec{v}))$, $C_0 \leq D$ y $e' = \texttt{new } C_0(\vec{v})$. La prueba de $\Gamma \vdash (D)(\texttt{new } C_0(\vec{v})) : C$ debe terminar con (T-CAST-AR) pues terminando con (T-CAST-AB) o (T-CAST-EST) podría contradecir la aseveración de $C_0 \leq D$. Las premisas de (T-CAST-AR) deben ser $\Gamma \vdash \texttt{new } C_0(\vec{v}) : C_0$ y $D = C_0$, terminando el caso.

Ahora veremos los casos para **Orden de evaluación**:

Caso (E-ACCATR)
Tenemos $e = e_0.f_i$, $e' = e_0'.f_i$ y $e_0 \to_T e_0'$.
Por (T-ATR) tenemos que $\Gamma \vdash e_0 : C_0$ y $\texttt{atributos}(C_0) = \vec{C}\ \vec{f}$. Por hipótesis de inducción tenemos que $\Gamma \vdash e_0' : C_0'$ para alguna $C_0' \leq C_0$. Por la derivación de `atributos` sabemos que $\texttt{atributos}(C_0') = \vec{C}\ \vec{f}, \vec{D}\ \vec{g}$. Por lo tanto, concluimos $\Gamma \vdash e_0'.f_i : C_i$ por (T-ATR).

Caso (E-INVMET)
Tenemos $e = e_0.m(\vec{e})$, $e' = e_0'.m(\vec{e})$ y $e_0 \to_T e_0'$.
Sea $\Gamma \vdash e_0 : C_0$, por hipótesis de inducción tenemos que $\Gamma \vdash e_0' : C_0'$ para alguna $C_0' \leq C_0$. Por (T-INVMET) tenemos que $\texttt{tipoMetodo}(m, C_0) = (\vec{D}, C)$ $\Gamma \vdash \vec{e} : \vec{C}$, con $\vec{C} \leq \vec{D}$. Por la derivacion de `tipoMetodo` podemos concluir que $\texttt{tipoMetodo}(m, C_0') = (\vec{D}, C)$, terminando el caso.

Caso (E-INVMET-ARGS)
Tenemos $e = v_0.m(\vec{v}, e_0, \vec{e})$, $e' = v_0.m(\vec{v}, e_0', \vec{e})$ y $e_0 \to_T e_0'$.
Sea $\Gamma \vdash e_0 : C_0$ y $\Gamma \vdash v_0.m(\vec{v}, e_0, \vec{e}) : C$. Para asegurar la preservación de tipos dado que sólo modificamos los argumentos del método la regla (T-INVMET) nos pide que $\vec{C} \leq \vec{D}$ se cumpla, pero por hipótesis de inducción tenemos que $\Gamma \vdash e_0' : C_0'$ para alguna $C_0' \leq C_0$, terminando el caso.

Caso (E-NEW-ARGS)

Tenemos $\mathtt{e} = \mathtt{new\ C}(\vec{\mathtt{v}}, \mathtt{e}_0, \vec{\mathtt{e}}), \quad \mathtt{e}' = \mathtt{new\ C}(\vec{\mathtt{v}}, \mathtt{e}'_0, \vec{\mathtt{e}}), \quad \mathtt{e}_0 \to_{\mathtt{T}} \mathtt{e}'_0$.
Análogo al anterior, pero con (T-NEW).

Caso (E-CAST)
Tenemos $\mathtt{e} = (\mathtt{D})\mathtt{e}_0, \quad \mathtt{e}' = (\mathtt{D})\mathtt{e}'_0, \quad \mathtt{e}_0 \to_{\mathtt{T}} \mathtt{e}'_0$, existen tres subcasos.

- Subcaso (T-CAST-AR): $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0, \quad \mathtt{C}_0 \leq \mathtt{D}, \quad \mathtt{D} = \mathtt{C}$.

  Por hipótesis de inducción tenemos $\Gamma \vdash \mathtt{e}'_0 : \mathtt{C}'_0$ para alguna $\mathtt{C}'_0 \leq \mathtt{C}_0$. Por transitividad de $\leq$ tenemos $\mathtt{C}'_0 \leq \mathtt{C}$. Por lo tanto, para (T-CAST-AR) concluimos $\Gamma \vdash (\mathtt{C})\mathtt{e}'_0 : \mathtt{C}$ (sin advertencia estúpida adicional).

- Subcaso (T-CAST-AB): $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0, \quad \mathtt{D} \leq \mathtt{C}_0, \quad \mathtt{D} = \mathtt{C}$

  Por hipótesis de inducción tenemos $\Gamma \vdash \mathtt{e}'_0 : \mathtt{C}'_0$ para alguna $\mathtt{C}'_0 \leq \mathtt{C}_0$. Si $\mathtt{C}'_0 \leq \mathtt{C}$ o $\mathtt{C} \leq \mathtt{C}'_0$, entonces $\Gamma \vdash (\mathtt{C})\mathtt{e}'_0 : \mathtt{C}$ por (T-CAST-AR) o (T-CAST-AB) (sin advertencia estúpida adicional.

  Por otro lado, si ambas $\mathtt{C}'_0 \not\leq \mathtt{C}$ y $\mathtt{C} \not\leq \mathtt{C}'_0$ ocurren, entonces obtenemos $\Gamma \vdash (\mathtt{C})\mathtt{e}'_0 : \mathtt{C}$ con advertencia estúpida por (T-CAST-EST).

- Subcaso (T-CAST-EST): $\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0, \quad \mathtt{D} \not\leq \mathtt{C}_0, \quad \mathtt{C}_0 \not\leq \mathtt{D}, \quad \mathtt{D} = \mathtt{C}$.

  Por hipótesis de inducción tenemos $\Gamma \vdash \mathtt{e}'_0 : \mathtt{C}'_0$ para alguna $\mathtt{C}'_0 \leq \mathtt{C}_0$. Entonces, ambas $\mathtt{C}'_0 \not\leq \mathtt{C}$ y $\mathtt{C} \not\leq \mathtt{C}'_0$ se mantiene. Por lo tanto obtenemos $\Gamma \vdash (\mathtt{C})\mathtt{e}'_0 : \mathtt{C}$ con advertencia estúpida.

□

La regla de conversión de tipos explícita estúpida es un tecnicismo necesario para poder probar la preservación de tipos con una semántica operacional estructural.

3.5.2. Progreso

Respecto a la propiedad de progreso esta vez debemos observar que un programa bien tipado podría bloquearse debido a un uso indebido en la conversión de tipos explícita, por ejemplo la expresión:

$$(\mathtt{C})(\ \mathtt{new\ Object}())$$

con $\mathtt{C} \neq \mathtt{Object}$ está bien tipada debido a la regla de conversión de tipos explícita estúpida, pero queda bloqueada puesto que la regla (E-CASTNEW) exige verificar que $\mathtt{Object} \leq \mathtt{C}$ lo cual no se cumple.[13]

Proposición 2 (Progreso)**.** *Sea T una tabla de clases bien formada. Si $\vdash e : C$ entonces sucede una y sólo una de las siguientes condiciones:*

[13] De hecho cuando se bloquea una expresión bien tipada, ésta es la única causa posible.

1. *e es un valor.*

2. *e contiene una expresión de la forma* $(\texttt{C})\ \texttt{new}\ \texttt{D}(\vec{\texttt{v}})$ *donde* $\texttt{D} \not\leq \texttt{C}$.

3. *Existe* e' *tal que* $e \to_T e'$.

Prueba.

Recordemos que un valor es de la forma: $\texttt{v} = \texttt{new}\ \texttt{C}(\vec{\texttt{v}})$.

La demostración de nuestra proposición es por inducción sobre $\vdash \texttt{e} : \texttt{C}$

Caso (T-VAR)
No aplica.

Caso (T-ATR)
Tenemos $\vdash \texttt{e} : \texttt{C}_\texttt{i}$ con $\texttt{e} = \texttt{e}_0.\texttt{f}_\texttt{i}$ y $\vdash \texttt{e}_0 : \texttt{C}$ con $\texttt{atributos}(\texttt{C}) = \vec{\texttt{C}}\ \vec{\texttt{f}}$. Claramente $\texttt{e}_0.\texttt{f}_\texttt{i}$ no es un valor, y por hipótesis de inducción tenemos tres casos:

- $\texttt{e}_0$ es un valor, digamos $\texttt{e}_0 = \texttt{new}\ \texttt{C}(\vec{\texttt{v}})$, así $\texttt{e} = \texttt{new}\ \texttt{C}(\vec{\texttt{v}}).\texttt{f}_\texttt{i}$ y entonces $\texttt{e}_0.\texttt{f}_\texttt{i} \to_\texttt{T} \texttt{v}_\texttt{i}$. Por lo tanto, $\texttt{e}' = \texttt{v}_\texttt{i}$, cumpliendo así con *3*.

- $\texttt{e}_0$ contiene una expresión $(\texttt{C})\texttt{new}\ \texttt{D}(\vec{\texttt{v}})$ con $\texttt{D} \not\leq \texttt{C}$, claramente $\texttt{e}_0.\texttt{f}_\texttt{i}$ cumple *2*.

- Existe $\texttt{e}'_0$ tal que $\texttt{e}_0 \to_\texttt{T} \texttt{e}'_0$ por lo tanto $\texttt{e}_0.\texttt{f}_\texttt{i} \to_\texttt{T} \texttt{e}'_0.\texttt{f}_\texttt{i}$ aplicando (E-ACCATR), cumpliendo con *3*.

Caso (T-INVMET)
Tenemos $\vdash \texttt{e} : \texttt{C}$ con $\texttt{e} = \texttt{e}_0.\texttt{m}(\vec{\texttt{e}})$, $\vdash \texttt{e}_0 : \texttt{C}_0$, $\vdash \vec{\texttt{e}} : \vec{\texttt{C}}$, $\vdash \texttt{e}_0.\texttt{m}(\vec{\texttt{e}}) : \texttt{C}$, $\texttt{tipoMetodo}(\texttt{m}, \texttt{C}_0) = \vec{\texttt{D}} \to_\texttt{T} \texttt{C}$ y $\vec{\texttt{C}} \leq \vec{\texttt{D}}$.
Por hipótesis de inducción para $\vdash \texttt{e}_0 : \texttt{C}_0$ tenemos tres casos:

- $\texttt{e}_0$ es un valor, es decir, $\texttt{e}_0 = \texttt{new}\ \texttt{C}(\vec{\texttt{v}})$ entonces $\texttt{e} = (\texttt{new}\ \texttt{C}(\vec{\texttt{v}})).\texttt{m}(\vec{\texttt{e}})$. Aplicaremos la hipótesis de inducción a $\vec{\texttt{e}}$

  - $\vec{\texttt{e}}$ son valores, es decir $\vec{\texttt{e}} = \vec{\texttt{w}}$. Entonces $\texttt{e} = (\texttt{new}\ \texttt{C}(\vec{\texttt{v}})).\texttt{m}(\vec{\texttt{w}})$ aplicando la regla (E-INVNEW) concluimos con que e cumple *3*.
  - Si $\vec{\texttt{e}}$ cumple con *2* entonces e cumple con *2*.
  - Si alguna $\texttt{e}_\texttt{i} \in \vec{\texttt{e}}$ cumple con $\texttt{e}_\texttt{i} \to_\texttt{T} \texttt{e}'_\texttt{i}$ entonces se aplica la regla (E-NEW-ARGS) y e cumple con *3*.

- $\texttt{e}_0$ cumple con *2* y por lo tanto e también cumple con *2*.

- Si existe $\texttt{e}'_0$ tal que $\texttt{e}_0 \to_\texttt{T} \texttt{e}'_0$ entonces se aplica la regla E-INVMET y e cumple con *3*, con $\texttt{e}' = \texttt{e}'_0.\texttt{m}(\vec{\texttt{e}})$.

Caso (T-NEW)
Tenemos $\vdash \mathtt{e} : \mathtt{C}$ con $\mathtt{e} = \mathtt{new\ C}(\vec{\mathtt{e}})$, $\vdash \vec{\mathtt{e}} : \vec{\mathtt{C}}$, $\vdash \mathtt{new\ C}(\vec{\mathtt{e}}) : \mathtt{C}$, $\mathtt{atributos}(\mathtt{C}) = \vec{\mathtt{D}}\ \vec{\mathtt{f}}$ y $\vec{\mathtt{C}} \leq \vec{\mathtt{D}}$. Aplicar nuestra hipótesis de inducción a $\vec{\mathtt{e}}$, tenemos nuevamente tres casos:

- Si $\vec{\mathtt{e}}$ son valores, es decir $\vec{\mathtt{e}} = \vec{\mathtt{v}}$ tenemos que $\mathtt{e}$ es un valor y cumple con *1*.
- Si existe $\mathtt{e_i} \in \vec{\mathtt{e}}$ que cumple con *2* entonces $\mathtt{e}$ cumple con *2*.
- Si tenemos $\mathtt{e} = \mathtt{new\ C}(\vec{\mathtt{v}}, \mathtt{e_i}, \vec{\mathtt{e}})$ entonces se aplica (E-NEW-ARGS) y finalmente $\mathtt{e}$ cumple con *3* con $\mathtt{e'} = \mathtt{new\ C}(\vec{\mathtt{v}}, \mathtt{e'_i}, \vec{\mathtt{e}})$.

Caso (T-CAST-AR)
Tenemos $\mathtt{e} = (\mathtt{C})\mathtt{e_0}$, $\vdash \mathtt{e_0} : \mathtt{D}$, y $\mathtt{D} \leq \mathtt{C}$.
Nuevamente obtenemos tres casos a la hipótesis de inducción sobre $\mathtt{e_0}$:

- Si $\mathtt{e_0}$ es un valor de la forma $\mathtt{new\ C}(\vec{\mathtt{v}})$ entonces podemos aplicar (E-CASTNEW), de este modo $\mathtt{e}$ cumple *3* con $\mathtt{e'} = \mathtt{e_0}$.
- Si $\mathtt{e_0}$ cumple *2* entonces $\mathtt{e}$ cumple *2*.
- Si existe $\mathtt{e'_0}$ tal que $\mathtt{e_0} \rightarrow_{\mathtt{T}} \mathtt{e'_0}$ entonces $\mathtt{e}$ cumple *3*, con $\mathtt{e'} = (\mathtt{C})\mathtt{e'_0}$.

Caso (T-CAST-AB)
Tenemos $\mathtt{e} = (\mathtt{C})\mathtt{e_0}$, con $\vdash \mathtt{e_0} : \mathtt{D}$, $\mathtt{C} \leq \mathtt{D}$, y $\mathtt{C} \neq \mathtt{D}$.
Nuevamente obtenemos tres casos a la hipótesis de inducción sobre $\mathtt{e_0}$:

- Si $\mathtt{e_0}$ es un valor entonces sabemos que $\mathtt{D} \not\leq \mathtt{C}$, por lo tanto $\mathtt{e}$ cumple *2*.
- Si $\mathtt{e_0}$ cumple *2* entonces $\mathtt{e}$ cumple *2*.
- Si existe $\mathtt{e'_0}$ tal que $\mathtt{e_0} \rightarrow_{\mathtt{T}} \mathtt{e'_0}$ entonces $\mathtt{e}$ cumple *3, con* $\mathtt{e'} = (\mathtt{C})\mathtt{e'_0}$.

Caso (T-CAST-EST)
Tenemos $\mathtt{e} = (\mathtt{C})\mathtt{e_0}$, $\vdash \mathtt{e_0} : \mathtt{D}$, $\mathtt{C} \not\leq \mathtt{D}$, $\mathtt{D} \not\leq \mathtt{C}$.

Análogo al caso anterior.

□

3.5.3. Correspondencia con JAVA

Para finalizar hacemos explícita la correspondencia deseada entre un programa en Java Peso Pluma y su versión en JAVA:

- Cada programa sintácticamente correcto en Java Peso Pluma también lo es en JAVA.
- Un programa sintácticamente correcto le puede ser asignado un tipo con Java Peso Pluma (sin usar la regla de conversión de tipos explícita estúpida) si y sólo si es posible asignarle un tipo en JAVA.

- Un programa bien tipado en Java Peso Pluma se comporta igual que en JAVA, por ejemplo la evaluación de un programa en Java Peso Pluma no llega a un paso final, es decir a un valor, si y sólo si, dicho programa no llega a un valor al compilarlo y ejecutarlo en JAVA.

Por supuesto que las afirmaciones anteriores no pueden probarse al no existir una formalización de JAVA. Sin embargo, es muy útil enunciar la correspondencia de manera precisa para dejar en claro el objetivo que estamos tratando de alcanzar. Lo cual facilita una forma de juzgar contraejemplos.

No bastaba con decir que el intérprete JPP usa la misma sintaxis que el lenguaje de programación JAVA, excluyendo de la sintaxis aquello que el intérprete excluye de JAVA. Por ello fue necesario definir de manera formal lo que se incluye como parte de la sintaxis del intérprete JPP aclarando que si es válido en dicho intérprete forzosamente lo es en el lenguaje JAVA.

Por otro lado, en la semántica definimos el subtipado como una relación fundamental para las clases, reglas de tipado que dan la certeza de contar con un tipo correcto siempre y agregamos algunas definiciones auxiliares para terminar la estructuración de las instrucciones en el intérprete. En conjunto lo anterior compone una semántica lo suficientemente pequeña para ajustarse al intérprete pero lo suficientemente sustancial para ajustarse a la semántica usada por el lenguaje JAVA.

El comportamiento dinámico o en tiempo de ejecución también lleva sus propias reglas que aseguran que los tipos son congruentes aún cuando el estado de evaluación de una instrucción ha avanzado en uno o más pasos, para asegurar esto se definieron las reglas de la semántica dinámica y se demostró el teorema de preservación de tipos. Ciertamente dicha demostración se sustenta en el libro [10], pero cabe mencionar que en este trabajo se hizo la demostración completa.

De igual forma, el progreso que conllevan los pasos de evaluación debe asegurar que se llega a algún estado conocido o a un valor. Para ello también se hizo la demostración completa del teorema de progreso para el intérprete JPP, ratificando que todos los estados que pueden ser alcanzados por algún paso de evaluación son bien conocidos y de manera sólida se sabe qué hacer a continuación, asegurando un progreso continuo.

En resumen, se termina este capítulo dando las definiciones y demostraciones necesarias para contar con un formalismo suficientemente sustentado, y así continuar con la implementación que dará una demostración práctica del intérprete JPP.

Capítulo 4

Implementación

Una vez desarrollados todos los aspectos teóricos de JPP, podemos proceder a realizar la implementación. Para este trabajo se ha elegido realizar la implementación en el lenguaje HASKELL, pues su paradigma puramente funcional nos ayudará a lograr el objetivo del trabajo.

4.1. Análisis léxico

Lo primero que debemos hacer es crear un programa que nos ayude a reconocer todos los componentes de un programa en JPP. Antes de comenzar con clases y objetos hay que reconocer los componentes más básicos, es decir, aquellos componentes atómicos que podemos encontrar en un programa, como lo son: punto, coma, paréntesis, llaves, punto y coma, etc... Además de reconocer los nombres de clases y las palabras reservadas de JPP (por ahora a los nombres de métodos y nombres de variables los veremos genéricamente como iguales, más adelante les daremos reconocimiento)

A estos componentes los denominaremos *tokens*[1] pues es común conocer por este nombre a un elemento individual en un lenguaje de programación.

Nuestros tokens se dividen en cuatro categorías:

- **Nombre de clase:** serán todas aquellas palabras que comiencen por mayúscula.
- **Palabras reservadas:** class, extends, super, return, this y new.
- **Símbolos de puntuación y separación:** . , ; ( ) { } =
- **Otro:** que puede ser el nombre de un método o variable

[1] Un token o lexema es un bloque de texto categorizado.

Este analizador léxico lo crearemos en un módulo de HASKELL al que llamaremos *Lexer.hs*. Lo primero será definir lo que es un token mediante un nuevo tipo, a continuación presentamos el código para ello:

```
1     module Lexer where
2
3     import Char
4
5     data Token = Punto             -- .
6                | Coma              -- ,
7                | PuCo              -- ;
8                | ParA              -- (
9                | ParC              -- )
10               | LlaA              -- {
11               | LlaC              -- }
12               | Igual             -- =
13               | Rsv PalabraRsv    -- Palabras reservadas
14               | Clase String
15               | Otro String
16                 deriving(Show,Eq)
17
```

Ahora definimos las palabras reservadas:

```
18     data PalabraRsv = Class
19                     | Extends
20                     | Super
21                     | Return
22                     | This
23                     | NewPal
24                     deriving (Show,Eq)
25
```

Una vez hechas estas definiciones procederemos a crear dos funciones

```
lexerPal :: String ->[Token]
lexer :: String ->[Token]
```

ambas reciben una cadena y regresan una lista de tokens, la diferencia es que la primera manejará las palabras y la segunda los símbolos.

El programa recibido será una cadena, y la manera de trabajar de estas dos funciones será: siempre empieza `lexer` en busca de un símbolo, si lo encuentra al principio de la cadena, lo codifica y continúa con el resto de la cadena. Si por el contrario no encuentra un símbolo, deja el trabajo a `lexerPal`, el cual verifica de qué tipo de palabra se trata (palabra reservada, nombre de clase u otro) Una vez analizada y codificada, permite a `lexer` continuar, lo cual se hace hasta terminar con la cadena.

El código de las funciones `lexer` y `lexerPal` es el siguiente:

```
26      lexerPal :: String ->[Token]
27
28      lexerPal palabra =
29              let (inicio,resto) = break (notAlpha) palabra
30
31                  reservada  = case inicio of
32                               ``new''      ->Rsv NewPal
33                               ``this''     ->Rsv This
34                               ``super''    ->Rsv Super
35                               ``class''    ->Rsv Class
36                               ``return''   ->Rsv Return
37                               ``extends''  ->Rsv Extends
38                               otherwise    ->checa inicio
39
40              in reservada:(lexer resto)
41
42              where notAlpha t = not(isAlpha t)
43                  checa (x:xs)| isUpper x = Clase (x:xs)
44                              | otherwise = Otro (x:xs)
45
46
47      lexer :: String ->[Token]
48
49      lexer [] = []
50
51      lexer ('.':xs) = Punto:(lexer xs)
52      lexer (',':xs) = Coma:(lexer xs)
53      lexer (';':xs) = PuCo:(lexer xs)
54      lexer ('(':xs) = ParA:(lexer xs)
55      lexer (')':xs) = ParC:(lexer xs)
56      lexer ('{':xs) = LlaA:(lexer xs)
57      lexer ('}':xs) = LlaC:(lexer xs)
58      lexer ('=':xs) = Igual:(lexer xs)
59
60
61      lexer ('\n':xs) = (lexer xs)
62      lexer ('\r':xs) = (lexer xs)
63      lexer (' ':xs) = (lexer xs)
64
65      lexer lista = lexerPal lista
```

Las líneas `61,62` y `63` se ocupan de quitar caracteres como espacios en blanco y saltos de línea del programa, dichos caracteres no son relevantes ya que nuestro programa usa otro tipo de separadores como paréntesis, coma, punto y coma, etc.
La expresión de la línea `29` se sirve de la función *break*, el trabajo de dicha función

consiste en dividir una lista en dos, según el primer elemento que cumpla el criterio establecido.

Con lo anterior hemos completado el programa *Lexer.hs* que se encarga de separar un código de JPP en tokens, veamos un ejemplo sencillo de ejecución:

```
Lexer> lexer ",identificador;{clase).metodo{new}this"
[Coma,Otro "identificador",PuCo,LlaA,Otro "clase",
ParC,Punto,Otro "metodo",LlaA,Rsv NewPal,LlaC,Rsv This]
```

El código dado a la función lexer ciertamente es caótico, sin embargo, sólo se busca el reconocimiento de sus caracteres. La salida muestra una lista con los elementos ya reconocidos, ordenados según el orden original de la entrada.

4.2. Análisis sintáctico

El análisis léxico de un programa como:

```
1     class C extends Object{
2
3           C (){super();}
4
5           Object metodo(){return new Object();}
6
7     }
```

regresa una lista de tokens de la siguiente forma:

```
lexer ("class C extends Object{C (){super();}
Object metodo(){return new Object();}}")

[Rsv Class,Clase "C",Rsv Extends,Clase "Object",LlaA,
Clase "C",ParA,ParC,LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,
Clase "Object",Otro "metodo",ParA,ParC,LlaA,Rsv Return,
Rsv NewPal,Clase "Object",ParA,ParC,PuCo,LlaC,LlaC]
```

Aún no sabemos si los tokens forman un programa *correcto*, para asegurar esto, comenzaremos por separar los elementos que componen a un programa: clases, objetos, métodos, constructor, etc... es decir, haremos el análisis sintáctico.

4.2.1. Definiciones

A continuación, haremos las definiciones necesarias en un nuevo módulo de Haskell al que llamaremos *Parser.hs*

Iniciamos con definiciones de tipo que nos ayudarán a entender mejor el código:

```
1     module Parser where
2
3     import Lexer
4
5     type NomClase = String
6     type NomMetodo = String
7     type NomObjeto = String
8     type NomAtributo = String
9     type NomVariable = String
10    type Objeto = (NomClase,NomObjeto)
11
```

Ahora, definiremos las expresiones, en nuestro intérprete todo aquello que tome una de las siguientes formas será considerado una expresión:

```
12     -- « Expresiones »
13
14     data Expr = New (NomClase,[Expr])          -- new C(e)
15               | Cast (NomClase,Expr)           -- (C)e
16               | Acceso (Expr,NomAtributo)      -- e.atr
17               | Metodo (Expr,NomMetodo,[Expr]) -- e.m(e1,...,en)
18               | Variable NomVariable           -- variable
19                 deriving(Show,Eq)
20
```

Las definiciones hechas en las líneas `5,6,7,8` y `9` tienen como finalidad asignar un alias a los parámetros que serán recibidos en las expresiones. Como un ejemplo del porque es importante esto, notemos que una expresión escrita de la forma "`New (String,[Expr])`", en lugar de "`New (NomClase,[Expr])`", contiene un código más obscuro.

Para terminar con nuestras definiciones, precisaremos la forma que deben tener los métodos, el constructor, la clase y finalmente la tabla de clases.

Para `C m` $(\vec{\texttt{C}}\ \vec{\texttt{x}})\{\texttt{return e};\}$ tenemos

```
21
22    type MetodoDecl = (NomClase,NomMetodo,[Objeto],Expr)
23
```

Para $\texttt{C}(\vec{\texttt{C}}\ \vec{\texttt{x}})\{\texttt{super}(\vec{\texttt{e}});\ \texttt{this.f}_0 = \texttt{e}_0; \ldots; \texttt{this.f}_\texttt{i} = \texttt{e}_\texttt{i}\}$ tenemos

```
24
25    type ConstructorDecl = (NomClase,[Objeto],[Expr],[(Expr,Expr)])
26
```

Para `class C extends D` $\{\texttt{K}\ \vec{\texttt{M}}\}$ tenemos

```
27
```

```
28    type ClassDecl = (NomClase,[Objeto],ConstructorDecl,[MetodoDecl])
29
```

Para una lista de clases tenemos

```
30
31    type TablaClases = [ClassDecl]
32
```

Ahora viene el trabajo más importante, ¿cómo convertir un código JPP a una tabla de clases? Recordemos que el código de JPP será simplemente una lista de tokens, así que requeriremos de diversas funciones auxiliares para la separación correcta de ellos. Antes hagamos un pequeño análisis que servirá de pauta para el reconocimiento de paréntesis en el código.

```
1     class C extends D{
2
3       C f;
4
5       C (D g,C f){
6           super(g);
7           this.f = f;
8       }
9
10      B m(B x){
11          return e;
12      }
13    }
```

Se tiene una clase `C` con superclase `D`, la cual sólo tiene un atributo `f` de tipo `C`, el constructor es simple, la superclase `D` recibe sólo un parámetro e inicializa `f`; cuenta con un método llamado `m` el cual da como resultado algo de tipo `B` y recibe un sólo parámetro `x` cuyo tipo también es `B`.

Notemos una cosa, sólo hay tres formas que se encierran entre paréntesis, estas son:

- `D g, C f, B x`: definición de parámetros.
- `g`: paso de parámetros.
- `e`: expresión, no propiamente entre paréntesis, pero los podría contener.

Cabe resaltar que nuestros separadores no son espacios en blanco ni saltos de línea, sino llaves y paréntesis, lo cual significa que debemos hacer funciones, para que cada vez que nos encontremos con paréntesis esas funciones trabajen buscando y reconociendo dichas formas. También es importante mencionar que la definición de parámetros es de tipo `Objeto` (`NomClase,NomObjeto`) y que el paso de parámetros es de tipo `Expr` (`Variable NomVariable`).

La siguiente función se encarga de reconocer la definición de parámetros:

```
33    objetos :: [Token] ->([Objeto],[Token])
34
35    objetos (Clase nomC:Otro nomO:Coma:resto) =
36      let (varsM,resV) = objetos resto
37      in ((nomC,nomO):varsM,resV)
38
39    objetos (Clase nomC:Otro nomO:ParC:LlaA:resto) =
40      ([(nomC,nomO)],resto)
41
42    objetos (ParC:LlaA:resto) = ([],resto)
43
44    objetos _ = error ``error en parametros''
45
```

La función `objetos` además de regresar la lista de objetos, devuelve los tokens restantes, pues aún no han sido analizados y a ella no le corresponde hacerlo, a su vez elimina también los separadores que ya no necesitamos "){". Las líneas `35,36` y `37` se encargan de reconocer aquellos patrones del código que comienzan con el nombre de una clase y continúan con un alias que será el nombre de la variable. Las líneas `39` y `40` se encargan de reconocer el caso en que se ha llegado al último parámetro.

Una vez definidos los parámetros que serán recibidos se usará dicha definición al realizar el paso de parámetros, este paso implica el uso de expresiones, así que, haremos primero una función para reconocer expresiones y después nos enfocaremos al paso de parámetros.

Algoritmo para identificar expresiones

Las expresiones resultan ser todo un reto, pues finalmente deseamos crear una función que sin importar lo complejo de la estructura de la expresión realice un reconocimiento perfecto de ella. Para esto lo primero que haremos será un algoritmo que separe las expresiones guiándonos por los paréntesis, pues aquello que se encuentre dentro de unos paréntesis podrá ser otra expresión completamente ajena de la original.

Una vez encontrado un paréntesis, se hará un recorrido para identificar todos los tokens encerrados por dicho paréntesis, para lograr esta identificación mantendremos un contador que nos dirá cuántos paréntesis hay abiertos. Los tokens presentes antes de cerrar el último paréntesis se almacenan en una lista aparte, así al final, tendremos dos listas (tokens dentro, tokens siguientes) Nuestra función será llamada *cortaPP* (corta Por Paréntesis).

Algoritmo cortaPP

1. Si hay un paréntesis abierto y lo siguiente que leemos es un paréntesis que cierra, devolvemos la lista de tokens dentro y la lista restante omitiendo el paréntesis.

2. Si hay más de un paréntesis abierto y leemos un paréntesis que cierra, hacemos recursión disminuyendo en uno el contador de paréntesis abiertos y pasando el paréntesis leído a tokens dentro.

3. Sin importar la cantidad de paréntesis abiertos, al leer un paréntesis que abre hacemos recursión aumentando en uno el contador de paréntesis abiertos y tomando al paréntesis leído como uno de los tokens encerrados por los paréntesis principales.

4. Si leemos cualquier otra cosa sólo hacemos recursión pasando lo leído a tokens dentro.

5. Si ya no hay mas tokens por leer (lista vacía [ ]), mandamos error.

Ahora veamos la implementación del algoritmo anterior usando código en lenguaje HASKELL.

```
46      cortaPP :: ([Token],[Token]) ->Int ->([Token],[Token])
47      cortaPP (td,(ParC:ts)) 1 = (td,ts)
48      cortaPP (td,(ParC:ts)) n = cortaPP (td++[ParC],ts) (n-1)
49      cortaPP (td,(ParA:ts)) n = cortaPP (td++[ParA],ts) (n+1)
50      cortaPP (td,(x:ts)) n = cortaPP (td++[x],ts) n
51      cortaPP (_,[]) _ = error (``cortaPP ->Parentesis mal estructurados'')
52
```

Los siguientes dos algoritmos serán mutuamente recursivos. El primero es de la función *cambiaCorte*, la cual recibe una tupla de tokens, los primeros son aquellos encerrados por los paréntesis y los segundos son los tokens que aún no han sido analizados. Dicha tupla es obtenida tras ejecutar la función *cortaPP*.

El trabajo de la función *cambiaCorte* será identificar si la expresión cortada por paréntesis tenía alguna finalidad (es decir, separaba dos expresiones) o simplemente se trataba de una expresión encerrada entre paréntesis.

El segundo algoritmo pertenece a la función *expresion* y recibirá una lista de tokens y la transformará en una expresión de JPP.

ALGORITMO cambiaCorte

1. Si afuera de los paréntesis ya no hay tokens por analizar se invoca a la función *expresión* con la lista de tokens que se encuentran dentro del paréntesis.

2. Si en la tupla, la lista de tokens que representa aquellos que aún no han sido analizados sólo contiene los tokens “punto” y “otro” (correspondiente a la categoría `Otro`), entonces se trata de un acceso a atributo, por lo tanto construimos la expresión.

3. Si en la tupla, la lista de tokens que representa aquellos que aún no han sido analizados contiene los tokens “punto”, “otro”, “punto” y algo más, entonces a ese acceso a atributo se le solicita aún algo, que bien puede

ser otro acceso a atributo o bien una llamada a método, por tanto, se pasan los tokens "punto" y "otro" a la lista de tokens analizados y se hace recursión.

4. Si en la tupla, la lista de tokens que representa aquellos que aún no han sido analizados contiene los tokens "punto", "otro", "paréntesis que abre" y algo más; entonces `cortamos por paréntesis` lo que resta es decir `algo más` y si de ese corte no hay más tokens que analizar, entonces construimos la expresión como una invocación de método. Si aún hay tokens por analizar, hacemos recursión creando una nueva tupla, concatenando todo excepto el resto devuelto por el corte, pues esos serán los tokens por analizar en la recursión.

5. Si no pasa ninguno de los casos anteriores, se regresa error.

Veamos el código en HASKELL

```
53      cambiaCorte :: ([Token],[Token]) ->Expr
54      cambiaCorte (e1,[]) = expresion e1
55      cambiaCorte (e1,[Punto,Otro atr]) = Acceso (expresion e1,atr)
56      cambiaCorte (e1,(Punto:Otro atr:Punto:resto)) =
57            cambiaCorte (e1++[Punto,Otro atr],(Punto:resto))
58      cambiaCorte (e1,(Punto:Otro met:ParA:resto)) =
59            let (e3,e4) = cortaPP ([],resto) 1
60            in (if e4==[]
61                then Metodo (expresion e1,met,map (expresion) (separaExpr e3))
62                else cambiaCorte (e1++[Punto,Otro met,ParA]++e3++[ParC],e4))
63      cambiaCorte (e1,_) = error (``cambiaCorte ->Expresion mal estructurada'')
64
```

ALGORITMO expresion

1. Si la lista de tokens comienza por los tokens "paréntesis que abre", "clase", "paréntesis que cierra", construimos la conversión explícita de tipos haciendo recursión con el resto de la lista.

2. Si la lista de tokens comienza por el token "paréntesis que abre", dejamos el trabajo a `cambiaCorte` de lo que regrese cortaPP.

3. Si la lista de tokens sólo tiene el token "otro", entonces es una variable.

4. Si la lista de tokens sólo tiene la palabra reservada "This", entonces es la variable This.

5. Si la lista de tokens comienza por la palabra reservada "This" y algo más, entonces ejecutamos la función `cambiaCorte` con la cabeza de la lista y el resto.

6. Si es igual que el paso anterior pero en vez de la palabra reservada "This" es "otro", hacemos lo mismo que el paso anterior pero con "otro".

7. Si la lista de tokens comienza con los tokens "new", "clase", "paréntesis que abre" y algo más, ejecutamos `cortePP` del resto(algo más) y en caso de que no tenga tokens siguientes, entonces construimos la expresión *new*, haciendo recursión en toda la lista de tokens encontrados dentro de los paréntesis, en caso de que tenga tokens siguientes, llamamos a `cambiaCorte` creando una nueva tupla concatenando todo excepto el resto devuelto por el corte, pues esos serán los tokens siguientes en `cambiaCorte`.

8. Si no pasa ninguno de los casos anteriores, se regresa un error.

Veamos el código en HASKELL

```
65      expresion :: [Token] ->Expr
66      expresion (ParA:Clase c:ParC:resto) = Cast (c,expresion resto)
67      expresion (ParA:resto) = cambiaCorte (cortaPP ([],resto) 1)
68      expresion [Otro var] = Variable var
69      expresion [Rsv This] = Variable ``This''
70      expresion (Rsv This:resto) = cambiaCorte ([Rsv This],resto)
71      expresion (Otro var:resto) = cambiaCorte ([Otro var],resto)
72      expresion (Rsv NewPal:Clase c:ParA:resto) =
73          let (e1,e2) = cortaPP ([],resto) 1
74          in (if e2==[]
75              then New (c,map (expresion) (separaExpr e1))
76              else cambiaCorte ([Rsv NewPal,Clase c,ParA]++e1++[ParC],e2))
77      expresion _ = error (``expresion ->Expresion mal estructurada'')
78
```

En el código anterior se usa una función llamada *separaExpr*, dicha función tiene la finalidad de separar las expresiones que pudieran venir dentro de paréntesis separadas por comas, esto pasa en una llamada a constructor o método que pudiera contener muchos parámetros, cada parámetro es una expresión. La implementación de esa función es la siguiente:

```
79      separaExpr :: [Token] ->[[Token]]
80      separaExpr [] = []
81      separaExpr expr = let (exp,exps) = separaExprAux ([],expr) 0
82                        in exp:separaExpr exps
83
84      separaExprAux :: ([Token],[Token]) ->Int ->([Token],[Token])
85      separaExprAux (e,[]) 0 = (e,[])
86      separaExprAux (e,[]) _ = error (``Faltan parentesis''++(show e))
87      separaExprAux (e,(Coma:res)) 0 = (e,res)
88      separaExprAux (e,(ParA:res)) n = separaExprAux (e++[ParA],res) (n+1)
89      separaExprAux (e,(ParC:res)) n = separaExprAux (e++[ParC],res) (n-1)
90      separaExprAux (e,(r:rs)) n = separaExprAux (e++[r],rs) n
91
```

Con esto concluimos el reconocimiento de expresiones, pues se ha completado la implementación de los algoritmos necesarios para identificar las separaciones hechas por los paréntesis dentro del código de JPP y el algoritmo que reconoce y construye las expresiones a partir de la lista recibida de tokens.

Ahora ya contamos con las herramientas necesarias para reconocer la definición de parámetros y las expresiones.

A continuación mostraremos la función que reconoce las expresiones dentro de la llamada a *super*:

```
92      expS :: [Token] ->([Expr],[Token])
93      expS (Rsv Super:ParA:resto) = let (exprs,resB) = break (==PuCo) resto
94                                    in (expSaux (init exprs),tail resB)
96      expS _ = error ``Super no encontrado''
97
98      expSaux :: [Token] ->[Expr]
99      expSaux tok = map (expresion) (separaExpr tok)
100
```

Ahora reconocemos las expresiones dentro de *super*; sin embargo, como vimos formalmente en la sección 3.4.4 dichas expresiones deben ser exclusivamente variables.

Dentro del constructor tenemos también las asignaciones, el único lugar donde JPP permite asignaciones es cuando se inicializan los atributos de la clase, el código necesario para reconocerlas es el siguiente:

```
101      asignaciones :: [Token] ->([(Expr,Expr)],[Token])
102      asignaciones tokens = let (asignT,resto) = break (==LlaC) tokens
103                            in (asignacionesAux asignT,(tail resto))
104
105      asignacionesAux :: [Token] ->[(Expr,Expr)]
106      asignacionesAux [] = []
107      asignacionesAux asig =
108          let (a,as) = break (==PuCo) asig
109              (e1,e2) = break (==Igual) a
110          in (expresion e1,expresion (tail e2)):(asignacionesAux (tail as))
111
```

Este código toma todo lo restante en el constructor, y en la lista de tokens busca las asignaciones, regresando finalmente una lista de parejas de expresiones (la pareja representa las expresiones a ambos lados de la igualdad) y los tokens restantes sin la llave que cierra al constructor.

Ahora nos falta reconocer los métodos, para terminar así toda la clase. Para esto nos valdremos de tres funciones:

- La función *metodo* que recibe una lista de tokens y regresa la declaración del mismo (de tipo *MetodoDecl*) encontrada en la lista.

- La función *exprM* que recibe una lista de tokens y regresa la expresión encontrada dentro del método.
- La función *parseMaux* que recorre el resto de la clase y va identificando todos los métodos en ella, al final regresa una lista de *MetodoDecl* y los tokens que faltan por identificar, los cuales pueden definir otras clases.

El código de estas funciones se muestra a continuación:

```
112     metodo :: [Token] ->MetodoDecl
113     metodo (Clase nomC:Otro nomM:ParA:resto) =
114        let (varsM,resV) = objetos resto
115        in (nomC,nomM,varsM,exprM resV)
116
117     exprM :: [Token] ->Expr
118     exprM (Rsv Return:resto) =
119        let (exp,vac) = break (==PuCo) resto
120        in case vac of
121           [PuCo] -> expresion exp
122           _ -> error ``Error en las llaves de los metodos''
123
124     parseMaux :: ([MetodoDecl],[Token]) ->([MetodoDecl],[Token])
125     parseMaux (md,tok) =
126        let (pm,resto) = break (==LlaC) tok
127        in case pm of
128           [] -> (md,(tail resto))
129           _ -> parseMaux (md++[metodo pm],(tail resto))
130
```

Finalmente, debemos crear el código que conjunte todas las funciones que hemos definido para terminar así con el análisis sintáctico.

Como podemos notar, una clase se compone de atributos, constructor y métodos. Y un archivo puede contener la definición de una o más clases.

El reconocimiento de las clases estará dividido en cuatro funciones, cada una de ellas se encargará de reconocer uno de los aspectos que componen a las clases, veamos:

- *parser:* Se encargará de reconocer todas las clases que se definen en el programa, pues recordemos que una tabla de clases se compone de varias clases.
- *parseO:* Esta función tiene como propósito el reconocer los objetos que son atributos de clase.
- *parseC:* Esta función se encargará de reconocer el constructor de la clase.
- *parseM:* Se encargará de reconocer los métodos de la clase.

A continuación presentamos el código:

```
131     -- PARSER
132     parser :: [Token] ->TablaClases
133     parser [] = []
134     parser (Rsv Class:Clase nomC:Rsv Extends:Clase nomE:LlaA:resto) =
135         let (objs,resO) = parseO resto
136             (cons,resC) = parseC resO
137             (mets,resM) = parseM resC
138         in (if (uno cons)==nomC
139             then (nomE,objs,cons,mets):(parser resM)
140             else error ``Clase y constructor no coinciden...'')
141     parser e = error (``Esperaba la definicion de clase nueva''++(show e))
142     uno (x,_,_,_) = x
143
```

La función anterior debe recibir específicamente el inicio de un programa de la forma `class C extends D` $\{\ldots\}$ se encarga de separar los tres elementos, como lo son los objetos que son atributos de la clase, el constructor y los métodos, invocando a sus respectivas funciones.

En el constructor se hace la validación para asegurar que el nombre de clase coincida.

```
144     -- PARSER OBJETOS
145     parseO :: [Token] ->([Objeto],[Token])
146     parseO (Clase nomC:Otro nomO:PuCo:resto) = let (objs,resO) = parseO resto
147                                                in ((nomC,nomO):objs,resO)
148     parseO resto = ([],resto)
149
```

La función anterior debe recibir específicamente la declaración de atributos de clase de la forma $\vec{\texttt{C}}\ \vec{\texttt{x}};$. Así mismo, dicha función hace recursión buscando en cada paso recursivo el mismo patrón hasta identificar todos los atributos de clase.

```
150     -- PARSER CONSTRUCTOR
151     parseC :: [Token] ->(ConstructorDecl,[Token])
152
153     parseC (Clase nomC:ParA:resto) = let (varsC,resV) = objetos resto
154                                          (expSs,resS) = expS resV
155                                          (asigns,resC) = asignaciones resS
156                                      in ((nomC,varsC,expSs,asigns),resC)
157     parseC _ = error ``error antes de constructor''
158
```

La función anterior debe recibir específicamente la declaración del constructor de la clase de la forma $\texttt{C}\ (\vec{\texttt{C}}\ \vec{\texttt{y}})\{\texttt{super}(\vec{\texttt{e}});\texttt{this}.\vec{\texttt{x}} = \vec{\texttt{y}};\}$ identificando las variables que son parámetros del constructor, las expresiones pasadas como parámetros al constructor de la superclase y las asignaciones ligadas a los atributos de la clase.

```
159     -- PARSER METODOS
160     parseM :: [Token] ->([MetodoDecl],[Token])
161     parseM mts = parseMaux ([],mts)
162
```

La función anterior llama a la función auxiliar para reconocimiento de métodos, la cual recibe una lista vacía para depositar en ella los métodos en orden.

Si no se cumple la estructura exacta en la construcción de la clase, alguna de las funciones marcará el error ocurrido. Con esto damos por terminado el análisis sintáctico y presentamos un ejemplo de su funcionamiento:

```
1     Parser>parser (lexer ``class C extends D{
2                             C f;
3                             C (D g,C f){
4                               super(g);
5                               this.f = f;}
6                             B m(B x){return e;}
7                           }'')
8
9     [(``D'',[(``C'',''f'')],
10
11    (``C'',[(``D'',''g''),(``C'',''f'')],[Variable ``g''],
12    [(Acceso (Variable ``This'',''f''),Variable ``f'')]),
13
14    [(``B'',''m'',[(``B'',''x'')],Variable ``e'')])]
```

La lista en el renglón 9 muestra el nombre de la clase padre y una lista con los atributos de la clase C, tal cual indica la declaración de tipo *claseDecl*.

En los renglones 11 y 12 la lista muestra el nombre de la clase actual, una lista de definición de parámetros, una lista de paso de parámetros al constructor de la superclase y una lista de asignaciones siguiendo la declaración de tipo *constructorDecl*.

La lista en el renglón 14 muestra el tipo y nombre del método, una lista con la declaración de parámetros y la expresión del método, tal cual indica la declaración de tipo *metodoDecl*.

4.3. Tipado

Para la verificación de tipos haremos un nuevo módulo de Haskell al que llamaremos *VerificadorDeTipos.hs*, este módulo realizará el análisis de tipos para un programa en JPP.

4.3.1. Definiciones

Iniciamos importando los dos módulos que ya tenemos terminados y definiendo dos nuevos tipos, los cuales nos servirán para facilitar el análisis en el tipado

de los programas:

```
1     module VerificadorDeTipos where
2
3     import Lexer
4     import Parser
5
6     type TipoDecl = (Expr,NomClase)
7
8     type Ctx = [TipoDecl]
9
```

El tipo *TipoDecl* se refiere a una expresión y su tipo, por su parte el tipo *Ctx* se refiere al contexto necesario para asegurar que no existen variables libres o no cazadas en una expresión.

Bien, lo siguiente que haremos será implementar las definiciones auxiliares que se encuentran en la sección 3.4.3, el código para éstas es el siguiente:

```
10      -- « ATRIBUTOS »
11
12      atributos :: TablaClases ->NomClase ->[Objeto]
13
14      atributos _ ``Object'' = []
15
16      atributos tabla clase = (atributos tabla sClase)++(atrClase tabla clase)
17          where sClase = superClase tabla clase
18
```

Cumpliendo con

$$\frac{\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{atributos(D)} \;=\; \vec{\texttt{D}}\ \vec{\texttt{g}}}{\texttt{atributos(C)} \;=\; \vec{\texttt{D}}\ \vec{\texttt{g}},\ \vec{\texttt{C}}\ \vec{\texttt{f}}}$$

$$\texttt{atributos(Object)} \;=\; \varnothing$$

Nos servimos de las funciones auxiliares *atrClase* y *superClase* para obtener los atributos de la clase y a su vez para obtener el nombre de la clase padre respectivamente, esto último con el fin de obtener los atributos de la clase padre.

Las definiciones se muestran a continuación:

```
19      -- Obtenemos los atributos de la clase indicada
20      atrClase :: TablaClases ->NomClase ->[Objeto]
21      atrClase [] clase = error (clase++'': Clase no encontrada'')
```

```
22      atrClase ((_,atr,(claseH,_,_,_),_):resto) clase
23          | (clase==claseH) = atr
24          | otherwise = atrClase resto clase
25
26      -- Obtenemos el nombre de la super clase de la clase indicada
27      superClase :: TablaClases ->NomClase ->NomClase
28      superClase _ ``Object'' = ``Object''
29      superClase [] clase = error (clase++'': Clase no encontrada'')
30      superClase ((claseP,_,(claseH,_,_,_),_):resto) clase
31          | (clase==claseH) = claseP
32          | otherwise = superClase resto clase
33
```

La siguiente definición auxiliar sirve para encontrar los métodos de una clase en nuestra tabla de clases, a continuación podemos observar el código:

```
34      -- « TIPOMETODO »
35
36      tipoMetodo :: TablaClases ->NomMetodo ->NomClase ->([NomClase],NomClase)
37
38      tipoMetodo _ metodo ``Object'' = error (metodo++'': Metodo no encontrado'')
39
40      tipoMetodo tabla metodo clase | (mEnLista metodo mc) =
41                                        let (t,m,atr,e) = mLista metodo mc
42                                        in (tipoObj atr,t)
43                                    | otherwise =
44              tipoMetodo tabla metodo (superClase tabla clase)
45          where mc = (metodosClase tabla clase)
46
```

Cumpliendo con

$$\frac{\begin{array}{c}\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \\ \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \in \vec{\texttt{M}}\end{array}}{\texttt{tipoMetodo}(\texttt{m},\texttt{C}) \;=\; \vec{\texttt{B}} \to \texttt{B}}$$

$$\frac{\begin{array}{c}\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \\ \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \notin \vec{\texttt{M}}\end{array}}{\texttt{tipoMetodo}(\texttt{m},\texttt{C}) \;=\; \texttt{tipoMetodo}(\texttt{m},\texttt{D})}$$

Nos servimos de las funciones auxiliares *metodosClase* para obtener una lista con los métodos definidos en la clase especificada, *mLista* para seleccionar un método dentro de la lista anterior, *mEnLista* para saber si el método especificado es elemento de la lista de métodos y *tipoObj* para obtener sólo los tipos $\vec{\texttt{C}}$ de una lista de objetos de la forma $\vec{\texttt{C}}\ \vec{\texttt{x}}$.

Las definiciones se muestran a continuación:

```
47      -- Obtenemos los metodos de la clase indicada
48      metodosClase :: TablaClases ->NomClase ->[MetodoDecl]
49      metodosClase _ ``Object'' = []
50      metodosClase [] clase = error (clase++'': Clase no encontrada'')
51      metodosClase ((_,_,(claseH,_,_,_),met):resto) clase
52          | (clase==claseH) = met
53          | otherwise = metodosClase resto clase
54
55      -- Obtenemos el metodo indicado de la lista (no usa caso base)
56      mLista :: NomMetodo ->[MetodoDecl] ->MetodoDecl
57      mLista met ((t,m,atr,e):resto) | met==m = (t,m,atr,e)
58                                     | otherwise = mLista met resto
59
60      -- Buscamos el metodo indicado en la lista
61      mEnLista :: NomMetodo ->[MetodoDecl] ->Bool
62      mEnLista _ [] = False
63      mEnLista met ((_,m,_,_):resto) = (met==m)||(mEnLista met resto)
64
65      -- Obtenemos el tipo de los objetos
66      tipoObj :: [Objeto] ->[NomClase]
67      tipoObj [] = []
68      tipoObj ((cn,_):resto) = cn:(tipoObj resto)
69
```

La siguiente definición auxiliar sirve para encontrar la expresión que compone el cuerpo de un método en nuestra tabla de clases y se define como sigue:

```
70      -- « CUERPOMETODO »
71
72      cuerpoMetodo :: TablaClases ->NomMetodo ->NomClase ->([NomObjeto],Expr)
73
74      cuerpoMetodo _ metodo ``Object'' = error (metodo++'': Metodo no encontrado'')
75
76      cuerpoMetodo tabla metodo clase | (mEnLista metodo mc) =
77                                          let (t,m,atr,e) = mLista metodo mc
78                                          in (nomObj atr,e)
79                                      | otherwise =
80                  cuerpoMetodo tabla metodo (superClase tabla clase)
81          where mc = (metodosClase tabla clase)
82
```

Cumpliendo con

$$\frac{\texttt{TC(C)} \;=\; \texttt{class C extends D}\;\{\vec{\texttt{C}}\;\vec{\texttt{f}};\;\texttt{K}\;\vec{\texttt{M}}\} \qquad \texttt{B}\;\texttt{m}(\vec{\texttt{B}}\;\vec{\texttt{x}})\{\texttt{return e;}\} \in \vec{\texttt{M}}}{\texttt{cuerpoMetodo}(\texttt{m},\texttt{C}) \;=\; (\vec{\texttt{x}},\texttt{e})}$$

$$\frac{\texttt{TC(C)} \;=\; \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\} \qquad \texttt{B m}(\vec{\texttt{B}}\ \vec{\texttt{x}})\{\texttt{return e;}\} \notin \vec{\texttt{M}}}{\texttt{cuerpoMetodo(m, C)} \;=\; \texttt{cuerpoMetodo(m, D)}}$$

Nos servimos de la función auxiliar *nomObj* que toma una lista de objetos y regresa una lista que contiene solo los nombres de las variables. La implementación es como sigue:

```
83      -- Obtenemos el nombre de los objetos
84      nomObj :: [Objeto] ->[NomObjeto]
85      nomObj [] = []
86      nomObj ((_,obj):resto) = obj:(nomObj resto)
87
```

4.3.2. Tipado de expresiones

En la sección 3.4.4 vimos las reglas de tipado de JPP, ellas nos indican formalmente cómo realizar las pruebas necesarias para asegurar que un programa se encuentre correctamente tipado.

El tipado de expresiones cuenta con siete reglas:

- Variables.
- Acceso a atributos.
- Invocación de métodos.
- Creación de objetos.
- Conversión explícita de tipos hacia arriba.
- Conversión explícita de tipos hacia abajo.
- Conversión explícita de tipos estúpida.

Es importante permitir los tres tipos de conversión explícita de tipos para asegurar una buena tipificación. Ahora lo que necesitamos es modelar estas reglas en una función en Haskell, a esta función la llamaremos *tipode* y recibirá tres elementos:

- La tabla de clases `T`
- El contexto `Γ`
- La expresión a verificar `e`.

Cumpliendo con (`tipode T Γ e`) = `C` si y sólo si $\Gamma \vdash_{\texttt{T}} \texttt{e} : \texttt{C}$. Finalmente la función regresará un nombre de clase, pues no importa que expresión reciba, su tipo deberá ser una clase (recordemos que no existen tipos nativos como int,

boolean, float, etc...)

El código de la función *tipode* es el siguiente:

```
88      tipode :: TablaClases ->Ctx ->Expr->NomClase
89
90      tipode _ [] (Variable x) = error (``Variable ``++x++
91                                        `` ->Tipo indefinido en el contexto'')
92
93      tipode tabla ((e,cn):ts) (Variable x)
94          | e==(Variable x) = cn
95          otherwise = tipode tabla ts (Variable x)
96
97      tipode tabla ctx (Acceso (e,on)) =
98          let tipoExp = tipode tabla ctx e
99              objetos = (atributos tabla tipoExp)
100             (claseN,objN) = daObjeto objetos on
101         in claseN
102
103     tipode tabla ctx (Metodo (exp,metN,exps)) =
104         let tipoExp = tipode tabla ctx exp
105             tipoExps = map (tipode tabla ctx) exps
106             (rec,reg) = tipoMetodo tabla metN tipoExp
107         in if (subclases tabla tipoExps rec)
108            then reg
109            else error (`` tipos incompatibles en el metodo ->``
110            ++metN++(show rec)++(show tipoExps))
111
112     tipode tabla ctx (New (clase,exps)) =
113         let objetos = (atributos tabla clase)
114             tipoExps = map (tipode tabla ctx) exps
115             tipoObjs = (map fst objetos)
116         in if (subclases tabla tipoExps tipoObjs)
117            then clase
118            else error (`` tipode NEW `` ++
119                        (show tipoExps)++
120                        `` </:'' ++
121                        (show tipoObjs))
122
123     tipode tabla ctx (Cast (cn,expr)) =
124         let tipoExpr = tipode tabla ctx expr
125         in if (subclase tabla cn tipoExpr)
126            then cn -- conversion explicita de tipos hacia arriba
127            else if(subclase tabla tipoExpr cn)&&(cn /= tipoExpr)
128                 then cn -- conversion explicita de tipos hacia abajo
129                 else cn -- conversion explicita de tipos estupida
```

```
130
```

Esta función realiza las verificaciones de las reglas de tipado, de este modo hemos modelado adecuadamente las reglas de tipado para expresiones.

Para implementar la función *tipode* nos servimos de tres funciones auxiliares, las cuales describiremos a continuación:

- *daObjeto* que toma una lista de objetos y regresa el objeto especificado.
- *subClase* que indica si existe la relación de subclase, entre las dos clases especificadas dentro de la tabla de clases.
- *subClases* la cual verifica la relación de subclase en los elementos de dos listas.

El código de dichas funciones es el siguiente:

```
131     -- Obtenemos el objeto solicitado
132     daObjeto :: [Objeto] ->NomObjeto ->Objeto
133     daObjeto [] o = error (``objeto ``++o++'' no encontrado'')
134     daObjeto ((cn,on):os) obj | on==obj = (cn,on)
135                               | otherwise = daObjeto os obj
136
137     -- C <: D
138     subclase :: TablaClases ->NomClase ->NomClase ->Bool
139     subclase _ ``Object'' ``Object'' = True
140     subclase _ ``Object'' _ = False
141     subclase tabla c d = (c==d)|| -- reflexividad
142                          (subclase tabla (superClase tabla c) d)
143                          -- La transitividad se cumple en la recursion
144
145     -- C's <: D's
146     subclases :: TablaClases ->[NomClase] ->[NomClase] ->Bool
147     subclases _ [] [] = True
148     subclases tabla (x:xs) (y:ys) = (subclase tabla x y)&&
149                                     (subclases tabla xs ys)
150     subclases _ _ _ = error
151                       ``Parametros incompatibles en NEW o METODO''
152
```

4.3.3. Declaración correcta de métodos

La función *okMetodo* se utilizará para verificar si un método `m` perteneciente a la clase `C` es correcto en la tabla `T`. Cumpliendo con: (`okMetodo T C m`) = `True` si y sólo si `m ok en C` de acuerdo a lo especificado en `T`. La implementación es como sigue:

```
153     okMetodo :: TablaClases ->NomClase ->MetodoDecl ->Bool
154
155     okMetodo tabla c (regr,nomMet,objetos,expr) |
156              (elem (regr,nomMet,objetos,expr) metodosSup) =
157                    let (csms,cms) = (tipoMetodo tabla nomMet supClase)
158                        (csmc,cmc) = (tipoMetodo tabla nomMet c)
159                        ctx = (map (objAtipo) (objetos++[(c,''This'')]))
160                        tipoExpr = (tipode tabla ctx expr)
161                    in (cms==cmc)&&
162                        (csms==csmc)&&
163                        (subclase tabla tipoExpr cms)
164              | otherwise =
165                    let (csmc,cmc) = (tipoMetodo tabla nomMet c)
166                        ctx = (map (objAtipo) (objetos++[(c,''This'')]))
167                        tipoExpr = (tipode tabla ctx expr)
168                    in (subclase tabla tipoExpr cmc)
169              where supClase = (superClase tabla c)
170                    metodosSup = (metodosClase tabla supClase)
171
```

Cumpliendo con:

$$\frac{\begin{array}{c}\texttt{TC(C)} \;=\; \texttt{class C extends D}\{\ldots\} \\ \texttt{tipoMetodo}(\texttt{m},\texttt{D}) = \vec{\texttt{C}} \to \texttt{C}_0 \\ \vec{\texttt{x}} : \vec{\texttt{C}}, \texttt{this} : \texttt{C} \vdash \texttt{e}_0 : \texttt{C}_0' \\ \texttt{C}_0' \leq \texttt{C}_0\end{array}}{\texttt{C}_0\ \texttt{m}(\vec{\texttt{C}}\ \vec{\texttt{x}})\{\texttt{return e}_0;\}\ \texttt{OK en C}}$$

Los dos casos son necesarios, la función *tipoMetodo* marcará un error si no se verifica primero la existencia del método en la superclase. Si se presenta este caso, tendremos que asegurar la igualdad en los tipos que regresan las dos llamadas a *tipoMetodo*, uno para la clase actual y otro para la superclase. Por otro lado, si el método no fue definido en la superclase, entonces será suficiente con asegurar que el tipo de la expresión dentro del método es subtipo del tipo definido para el método.

Nos servimos de las siguientes funciones auxiliares:

- *objetoAtipo* la cual toma objetos y los transforma en declaraciones de tipo, esto con la finalidad de poder incluir dichos objetos en nuestro contexto.
- *map* y *elem* que son funciones nativas del lenguaje HASKELL. La primera permite aplicar una función a una lista de elementos y la segunda nos indica si un elemento especifico pertenece a una lista.

La implementación de *objetoAtipo* se muestra a continuación:

```
172     -- Cambiamos un objeto por un Tipo
```

```
173     objAtipo :: Objeto ->TipoDecl
174     objAtipo (cn,on) = (Variable on,cn)
175
```

4.3.4. Declaración correcta de clases

La última regla de tipado es para clases y será modelada por medio de la función *okClase*, la cual recibe la tabla de clases y una declaración de clase, posteriormente verificará si la clase se encuentra bien definida según la información de la tabla de clases.

```
176     okClase :: TablaClases ->ClassDecl ->Bool
177
178     okClase tabla (``Object'',atrs,(cn,objs,exprs,exprsIg),metodos) =
179         (exprs == []) &&
180         and (map (okMetodo tabla cn) metodos)
181
182     okClase tabla (supClase,atrs,(cn,objs,exprs,exprsIg),metodos) =
183         (tipoObj (take (length atrSup) objs))==(tipoObj atrSup) &&
184         (take (length gs) gsfs)==gs &&
185         and (map (okMetodo tabla cn) metodos)
186         where gs = (daNomVars exprs)
187               gsfs = map snd objs
188               atrSup = atributos tabla supClase
189
```

Cumpliendo con:

$$\frac{\begin{array}{c}\texttt{K} \;=\; \texttt{C}(\vec{\texttt{D}}\ \vec{\texttt{g}}, \vec{\texttt{C}}\ \vec{\texttt{f}})\{\texttt{super}(\vec{\texttt{g}});\ \texttt{this}.\vec{\texttt{f}} = \vec{\texttt{f}};\}\\ \texttt{atributos}(\texttt{D}) \;=\; \vec{\texttt{D}}\ \vec{\texttt{g}}\\ \vec{\texttt{M}}\ \texttt{OK en C}\end{array}}{\texttt{class C extends D}\{\vec{\texttt{C}}\ \vec{\texttt{f}};\ \texttt{K}\ \vec{\texttt{M}}\}\ \texttt{OK}}$$

Nos servimos de la función auxiliar *daNomVars* que toma una lista de variables y regresa una lista con sus nombres. Recordemos que la llamada a la superclase solo recibe variables, no es permitida ninguna otra expresión entre sus parámetros. Su implementación es como sigue:

```
190     daNomVars :: [Expr] ->[NomVariable]
191     daNomVars [] = []
192     daNomVars ((Variable n):vars) = n:(daNomVars vars)
193
```

Al igual que en el lenguaje JAVA, se permite el ingreso de comentarios dentro del código, para ello crearemos la función *comentarios* que toma una cadena y quita todo lo que se encuentre después de la instrucción \\ hasta el salto de línea, así como todo aquello que se encuentre dentro de las instrucciones

* *\ la función *hastaSalto* que recorre una cadena hasta un salto de línea y la función *hastaCierre* que detecta el cierre de la instrucción *\ permitiendo comentar secciones completas. La implementación es como sigue:

```
194     comentarios :: String ->String
195     comentarios [] = []
196     comentarios ('/':'/':resto) = comentarios (hastaSalto resto)
197     comentarios ('/':'*':resto) = comentarios (hastaCierre resto)
198     comentarios (x:resto) = x:(comentarios resto)
199
200     hastaSalto :: String ->String
201     hastaSalto [] = []
202     hastaSalto ('\n':resto) = resto
203     hastaSalto (_:resto) = hastaSalto resto
204
205     hastaCierre :: String ->String
206     hastaCierre [] = []
207     hastaCierre ('*':'/':resto) = resto
208     hastaCierre (_:resto) = hastaCierre resto
209
```

Con esto terminamos la implementación que se encargará de verificar el tipado en un programa.

Para la siguiente función usaremos una función perteneciente al lenguaje HASKELL llamada *do* cuya implementación usa *monadas*.[2] La función *do* devuelve el tipo *IO()*, que para fines prácticos podemos ver como la salida estándar. Nuestra función realizará la verificación de tipos en un programa, por esta razón la llamaremos *jppTipos*, dicha función recibirá el nombre de un archivo en cuyo contenido estará el programa a verificar.

Para este ejemplo mostraremos el código contenido en el archivo y las salidas del análisis léxico y sintáctico.

```
210     jppTipos :: String ->IO()
211     jppTipos archivo = do
212       contents <- readFile archivo
213       let sinCom = comentarios contents
214       let tokensLex = lexer sinCom
215       let tabla = parser tokensLex
216       putStr ''\n************************************************************''
217       putStr ''\n************************ Codigo ************************''
218       putStr ''\n************************************************************\n''
219       putStr contents
220       putStr ''\n\n''
```

[2]El término monada viene de una rama de las matemáticas abstractas conocida como *Teoría de categorías*. Esto refleja los fuertes principios matemáticos que subyacen en HASKELL[3]. La explicación del tema se encuentra fuera del alcance de este trabajo de tesis.

```
221       putStr ``\n************************************************************''
222       putStr ``\n************************* Lexer **************************''
223       putStr ``\n************************************************************\n''
224       print tokensLex
225       putStr ``\n\n''
226       putStr ``\n************************************************************''
227       putStr ``\n************************* Parser *************************''
228       putStr ``\n************************************************************\n''
229       print tabla
230       putStr ``\n\n''
231       putStr ``\n************************************************************''
232       putStr ``\n************************* Tipos **************************''
233       putStr ``\n************************************************************\n''
234       print (map (okClase tabla) tabla)
235
```

La función *jppTipos* recibe el nombre del archivo que contiene las clases del lenguaje JPP, lee el archivo y lo guarda como cadena en la variable *contents*, le quitamos los comentarios usando la función *comentarios* y comenzamos con el análisis léxico, para después realizar el análisis sintáctico. Finalmente verificamos que las clases que están bien formadas se encuentren correctamente tipadas.

Para demostrar el funcionamiento, guardamos en un archivo llamado *Prueba.jpp* el siguiente código de ejemplo que usamos en la sección 3.2.

```
1     class A extends Object { A() {super(); }}
2
3     class B extends Object { B() {super(); }}
4
5     class Par extends Object {
6         Object fst;
7         Object snd;
8
9         // Constructor:
10        Par(Object fst, Object snd){
11           super();
12           this.fst = fst;
13           this.snd =snd;
14        }
15
16        // Definicion de metodo:
17        Par setfst(Object newfst){
18           return new Par(newfst, this.snd);
19        }
20    }
21
```

Y ahora al ejecutar la función *jppTipos* con el archivo, la salida obtenida es la siguiente:

```
1   VerificadorDeTipos>jppTipos ‘‘Prueba.jpp’’
2
3   ************************************************************
4   ************************ Codigo ************************
5   ************************************************************
6
7   class A extends Object { A() {super(); }}
8
9   class B extends Object { B() {super(); }}
10
11  class Par extends Object {
12      Object fst;
13      Object snd;
14
15      // Constructor:
16      Par(Object fst, Object snd){
17         super();
18         this.fst = fst;
19         this.snd =snd;
20      }
21
22      // Definicion de metodo:
23      Par setfst(Object newfst){
24         return new Par(newfst, this.snd);
25      }
26  }
27
28  ************************************************************
29  ************************ Lexer ************************
30  ************************************************************
31  [Rsv Class,Clase ‘‘A’’,Rsv Extends,Clase ‘‘Object’’,LlaA,Clase ‘‘A’’,
32  ParA,ParC,LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,LlaC,Rsv Class,
33  Clase ‘‘B’’,Rsv Extends,Clase ‘‘Object’’,LlaA,Clase ‘‘B’’,ParA,ParC,
34  LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,LlaC,Rsv Class,Clase ‘‘Par’’,
35  Rsv Extends,Clase ‘‘Object’’,LlaA,Clase ‘‘Object’’,Otro ‘‘fst’’,
36  PuCo,Clase ‘‘Object’’,Otro ‘‘snd’’,PuCo,Clase ‘‘Par’’,ParA,
37  Clase ‘‘Object’’,Otro ‘‘fst’’,Coma,Clase ‘‘Object’’,Otro ‘‘snd’’,ParC,
38  LlaA,Rsv Super,ParA,ParC,PuCo,Rsv This,Punto,Otro ‘‘fst’’,Igual,
39  Otro ‘‘fst’’,PuCo,Rsv This,Punto,Otro ‘‘snd’’,Igual,Otro ‘‘snd’’,
40  PuCo,LlaC,Clase ‘‘Par’’,Otro ‘‘setfst’’,ParA,Clase ‘‘Object’’,
41  Otro ‘‘newfst’’,ParC,LlaA,Rsv Return,Rsv NewPal,Clase ‘‘Par’’,ParA,
42  Otro ‘‘newfst’’,Coma,Rsv This,Punto,Otro ‘‘snd’’,ParC,PuCo,LlaC,LlaC]
43
```

```
44  ************************************************************
45  ************************ Parser ************************
46  ************************************************************
47  [(‘‘Object’’,[],(‘‘A’’,[],[],[]),[]),(‘‘Object’’,[],(‘‘B’’,[],[],[]),[]),
48  (‘‘Object’’,[(‘‘Object’’,’’fst’’),(‘‘Object’’,’’snd’’)],
49  (‘‘Par’’,[(‘‘Object’’,’’fst’’),(‘‘Object’’,’’snd’’)],[],
50  [(Acceso (Variable ‘‘This’’,’’fst’’),Variable ‘‘fst’’),
51  (Acceso (Variable ‘‘This’’,’’snd’’),Variable ‘‘snd‘‘)]),
52  [(‘‘Par‘‘,‘‘setfst‘‘,[(‘‘Object‘‘,‘‘newfst‘‘)],New (‘‘Par‘‘,
53  [Variable ‘‘newfst‘‘,Acceso (Variable ‘‘This‘‘,‘‘snd‘‘)]))])]
54
55  ************************************************************
56  ************************ Tipos ************************
57  ************************************************************
58  [True,True,True]
59
60  VerificadorDeTipos>
61
```

El programa contenido en *Prueba.jpp* tiene tres clases `A, B y Par`, es por tal motivo que el verificado de tipos nos indica tres valores booleanos expresando que las tres clases son correctas en la tabla, línea 58.

A continuación veremos un ejemplo de error en el tipado:

```
1   VerificadorDeTipos>jpp ‘‘Prueba.jpp‘‘
2
```

En esta fase sólo se muestra el código contenido en el archivo que ha sido recibido por el intérprete, el cual contiene el error que se desea ejemplificar.

```
3   ************************************************************
4   ************************ Codigo ************************
5   ************************************************************
6   class A extends Par { A() {super(); }}
7
8   class B extends Object { B() {super(); }}
9
10  class Par extends Object {
11      Object fst;
12      Object snd;
13
14      // Constructor:
15      Par(Object fst, Object snd){
16         super();
17         this.fst = fst;
18         this.snd =snd;
```

```
19          }
20
21          // Definicion de metodo:
22          Par setfst(Object newfst){
23             return (Object)new Par(newfst, this.snd);
24          }
25      }
26
```

Tras el uso de la función *lexer* el intérprete ha transformado todos los caracteres a una lista de tokens la cual se muestra de la línea 27 a la 42.

```
27      ************************************************************
28      ************************* Lexer ****************************
29      ************************************************************
30      [Rsv Class,Clase ``A'',Rsv Extends,Clase ``Par'',LlaA,Clase ``A'',
31      ParA,ParC,LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,LlaC,Rsv Class,
32      Clase ``B'',Rsv Extends,Clase ``Object'',LlaA,Clase ``B'',ParA,ParC,
33      LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,LlaC,Rsv Class,Clase ``Par'',
34      Rsv Extends,Clase ``Object'',LlaA,Clase ``Object'',Otro ``fst'',
35      PuCo,Clase ``Object'',Otro ``snd'',PuCo,Clase ``Par'',ParA,
36      Clase ``Object'',Otro ``fst'',Coma,Clase ``Object'',Otro ``snd'',ParC,
37      LlaA,Rsv Super,ParA,ParC,PuCo,Rsv This,Punto,Otro ``fst'',Igual,
38      Otro ``fst'',PuCo,Rsv This,Punto,Otro ``snd'',Igual,Otro ``snd'',
39      PuCo,LlaC,Clase ``Par'',Otro ``setfst'',ParA,Clase ``Object'',
40      Otro ``newfst'',ParC,LlaA,Rsv Return,ParA,Clase ``Object'',ParC,
41      Rsv NewPal,Clase ``Par'',ParA,Otro ``newfst'',Coma,Rsv This,Punto,
42      Otro ``snd'',ParC,PuCo,LlaC,LlaC]
43
```

De la línea 44 a la 54 se muestra la lista regresado por la función *parser*. Se trata de la interpretación de los tokens anteriores, dando como resultado una lista de expresiones en JPP.

```
44      ************************************************************
45      ************************* Parser ***************************
46      ************************************************************
47      [(``Par'',[],(``A'',[],[],[]),[]),(``Object'',[],(``B'',[],[],[]),[]),
48      (``Object'',[(``Object'',``fst''),(``Object'',``snd'')],(``Par'',
49      [(``Object'',``fst''),(``Object'',``snd'')],[],
50      [(Acceso (Variable ``This'',``fst''),Variable ``fst''),
51      (Acceso (Variable ``This'',``snd''),Variable ``snd'')]),
52      [(``Par'',``setfst'',[(``Object'',``newfst'')],
53      Cast (``Object'',New (``Par'',[Variable ``newfst'',
54      Acceso (Variable ``This'',``snd'')]))])])]
```

```
55
```

Y finalmente se obtiene una lista de valores booleanos, cada uno de los cuales representa a cada clase recibida dentro del archivo contenedor del código. El orden en la lista corresponde con el orden dentro de dicho archivo. Por otro lado, el valor *True* indica que los tipos de dicha clase han sido verificados satisfactoriamente, y a su vez, el valor *False* significa que su respectiva clase no cumple con las reglas de tipado.

```
56      ************************************************************
57      ************************ Tipos *************************
58      ************************************************************
59      [False,True,False]
60
61      VerificadorDeTipos>
62
```

La clase `A` es incorrecta, pues es subclase de `Par`, y la clase `Par` recibe dos atributos en su constructor de tipo `Object`. Sin embargo, la expresión `super` dentro del constructor de la clase `A` no los recibe, y tampoco los recibe el constructor de la clase `A`. Estas dos fallas muestran que la clase `A` no es correcta en la tabla.

La clase `Par` por su parte realiza una conversión de tipos explícita, (sin embargo, este no es un error) puesto que como vimos anteriormente nuestro verificador de tipos permite realizar cualquiera de estas conversiones. Ahora bien, la expresión del método `setfst` es de tipo `Object`, el cual espera regresar algo de tipo `Par`, lo anterior es un error de tipos dado que `Object` $\not\leq$ `Par`. Si cambiamos el orden de estos dos tipos, es decir, si el método `setfst` regresara algo de tipo `Object` y la expresión fuera de tipo `Par`, el verificador daría por correctos los tipos representados en la clase `Par`, pues `Par` $\leq$ `Object`.

4.4. Evaluación

Las reglas de evaluación vistas en la sección 3.4.6, se dividen en cuatro casos:

- Acceso a atributos.
- Invocación de métodos.
- Conversión de tipos explícita.
- Orden de evaluación.

Donde se hacen transiciones de estados según la regla adecuada para el estado del que se trate, terminando la evaluación cuando tenemos un valor de la forma `new C(`$\vec{\mathtt{v}}$`)`. Los estados siguientes se modelarán usando recursión sobre la función *eval*, la cual se encargará de pasar de un estado a otro según las reglas definidas para la evaluación, excepto para el caso de *New*, pues éste requiere hacer recursión sobre todos sus parámetros.

Crearemos un módulo llamado *Evaluador.hs* en el que se escribirán las funciones necesarias para realizar los pasos de la evaluación.

Nuestra nueva función *eval*, requiere de un *Programa* para poder evaluar, es decir, necesita de la tabla de clases `T` donde se hicieron las definiciones y una expresión `e` que use dicha tabla para poder ser evaluada, de este modo, `eval(T, e)` $=$ `e`$'$ si y sólo si `e` $\rightarrow_{\mathtt{T}}$ `e`$'$. Recordemos que esa expresión es equivalente a la expresión dentro de un método *main* en JAVA. Primero definiremos el tipo *Programa* como sigue:

```
1      module Evaluador where
2
3      import Lexer
4      import Parser
5      import VerificadorDeTipos
6
7      type Programa = (TablaClases,Expr)
8
```

El programa se define como un par que contiene la tabla de clases y la expresión a evaluar. Ahora veamos el código de nuestra función *eval*, previo a cada definición en *eval* se incluye la regla o reglas de las que ha sido derivado el orden de evaluación según la sección 3.4.6.

```
9      eval :: Programa ->Expr
```

$$\frac{\mathtt{e}_0 \rightarrow_{\mathtt{T}} \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{f} \rightarrow_{\mathtt{T}} \mathtt{e}_0'.\mathtt{f}} \quad \text{(E-ACCATR)}$$

$$\frac{\mathtt{atributos(C)} = \vec{\mathtt{C}}\ \vec{\mathtt{f}}}{\mathtt{new\ C}(\vec{\mathtt{v}}).\mathtt{f_i} \rightarrow_{\mathtt{T}} \mathtt{v_i}} \quad \text{(E-ATRNEW)}$$

```
10      eval (tabla,(Acceso (e0,f))) =
11           let  (New (c,vs)) = eval (tabla,e0)
12                objs = (atributos tabla c)
13                posAtr = daPos f (map snd objs)
14           in vs!!posAtr
15
```

$$\frac{\mathtt{e}_0 \rightarrow_{\mathtt{T}} \mathtt{e}'_0}{\mathtt{e}_0.\mathtt{m}(\vec{\mathtt{e}}) \rightarrow_{\mathtt{T}} \mathtt{e}'_0.\mathtt{m}(\vec{\mathtt{e}})} \quad \text{(E-INVMET)}$$

$$\frac{\mathtt{e}_\mathtt{i} \rightarrow_{\mathtt{T}} \mathtt{e}'_\mathtt{i}}{\mathtt{v}_0.\mathtt{m}(\vec{\mathtt{v}}, \mathtt{e}_\mathtt{i}, \vec{\mathtt{e}}) \rightarrow_{\mathtt{T}} \mathtt{v}_0.\mathtt{m}(\vec{\mathtt{v}}, \mathtt{e}'_\mathtt{i}, \vec{\mathtt{e}})} \quad \text{(E-INVMET-ARGS)}$$

```
16      eval (tabla,(Metodo (e,m,es))) =
17           let (New (c,vs)) = eval (tabla,e)
18               us = map (evalExp tabla) es
19               (xs,e0) = (cuerpoMetodo m c tabla)
20           in eval (tabla, sustThis (New (c,vs)) (sust xs us e0))
21
```

$$\frac{\mathtt{e}_0 \rightarrow_{\mathtt{T}} \mathtt{e}'_0}{(\mathtt{C})\mathtt{e}_0 \rightarrow_{\mathtt{T}} (\mathtt{C})\mathtt{e}'_0} \quad \text{(E-CAST)}$$

$$\frac{\mathtt{C} \leq \mathtt{D}}{(\mathtt{D})(\mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}})) \rightarrow_{\mathtt{T}} \mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}})} \quad \text{(E-CASTNEW)}$$

```
22      eval (tabla,(Cast (d,e))) =
23           let (New (c,vals)) = eval (tabla,e)
24           in if (subclase c d tabla)
25              then New (c,vals)
26              else error ``cast erroneo''
27
```

$$\frac{\mathtt{e}_\mathtt{i} \rightarrow_{\mathtt{T}} \mathtt{e}'_\mathtt{i}}{\mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}}, \mathtt{e}_\mathtt{i}, \vec{\mathtt{e}}) \rightarrow_{\mathtt{T}} \mathtt{new}\ \mathtt{C}(\vec{\mathtt{v}}, \mathtt{e}'_\mathtt{i}, \vec{\mathtt{e}})} \quad \text{(E-NEW-ARGS)}$$

```
28      eval (tabla,(New (c,es))) = (New (c,(map (evalExp tabla) es)))
29
```

Nos servimos de un grupo de funciones auxiliares. La función *evalExp* hace la evaluación de una expresión según la tabla de clases que recibe como parámetro.

```
30      evalExp :: TablaClases ->Expr ->Expr
31      evalExp tabla expr = eval (tabla,expr)
32
```

La función *daPos* indica la posición de un elemento específico en una lista.

```
33      daPos :: (Eq a, Num b) =>a ->[a] ->b
34      daPos _ [] = error ``no existe elemento''
35      daPos atr (a:as) | atr==a = 0
36                       | otherwise = 1+(daPos atr as)
37
```

La función *sust* toma una lista de nombres de variables, una lista de expresiones de igual tamaño y una expresión para posteriormente sustituir todas las variables de la lista en la expresión recibida como tercer parámetro por sus respectiva expresión en la lista de expresiones. Es decir, $\mathsf{e}_0[\vec{\mathsf{x}} := \vec{\mathsf{u}}]$.

```
38      sust :: [NomVariable] ->[Expr] ->Expr ->Expr
39      sust [] [] expr = expr
40      sust (a:as) (e:es) expr = sust as es (sustAux a e expr)
41
```

La función *sustAux* que toma un nombre de variable y dos expresiones para poder sustituir todas las apariciones de dicha variable en la expresión específica por la expresión recibida como segundo parámetro.

```
42      sustAux :: NomVariable ->Expr ->Expr ->Expr
43      sustAux a e (Variable v) | a==v = e
44                               | otherwise = Variable v
45      sustAux a e (New (cn,exprs)) = New (cn,(map (sustAux a e) exprs))
46      sustAux a e (Cast (cn,expr)) = Cast (cn,(sustAux a e expr))
47      sustAux _ _ (Acceso (Variable ``This'',field)) = Acceso (Variable ``This'',field)
48      sustAux a e (Acceso (expr,field)) = (Acceso ((sustAux a e expr),field))
49      sustAux a e (Metodo (expr,mn,exprs)) =
50          (Metodo ((sustAux a e expr),mn,(map (sustAux a e) exprs)))
51
```

La función *sustThis* toma dos expresiones y sustituye la primera expresión en la segunda expresión siempre que se encuentre en el mismo contexto a la expresión *Variable This*.

```
52      sustThis :: Expr ->Expr ->Expr
53      sustThis expr (Variable ``This'') = expr
54      sustThis expr (New (cn,exprs)) = New (cn,(map (sustThis expr) exprs))
55      sustThis expr (Cast (cn,exprc)) = Cast (cn,(sustThis expr exprc))
56      sustThis expr (Acceso (expra,field)) = (Acceso ((sustThis expr expra),field))
57      sustThis expr (Metodo (exprm,mn,exprs)) =
58          (Metodo ((sustThis expr exprm),mn,(map (sustThis expr) exprs)))
59
```

4.5. Pruebas

Para realizar pruebas sobre JPP, agregaremos unas líneas a la función `jppTipos` para realizar la evaluación. Crearemos un caso particular de expresión para evaluar el cual se incluye directamente en el código que veremos a continuación:

```
234     putStr ``\n\n''
```

```
235     putStr ``\n************************************************************''
236     putStr ``\n********************** Evaluacion **********************''
237     putStr ``\n************************************************************\n''
238     let exprMain = (expresion (lexer (``new Par(new Object(),new Object()).snd'')))
239     putStr (``Expresion: ''++(show (eval (tabla,exprMain))++
240             ``\nClase: ''++(tipode tabla [] exprMain)))
241     putStr ``\n************************************************************\n''
242
```

En este caso seguiremos usando el archivo *Prueba.jpp*, y la variable `exprMain` almacenará la expresión a evaluar. Primero declaramos un objeto de la clase `Par` y después hacemos un acceso al atributo `snd`. Veamos como funciona [3] (modificamos el nombre del módulo de HASKELL de *VerificadorDeTipos* a *Evaluador*)

```
1     Evaluador>jppTipos ``Prueba.jpp''
2     ************************************************************
3     ************************ Codigo ************************
4     ************************************************************
5     class A extends Object { A() {super(); }}
6
7     class B extends Object { B() {super(); }}
8
9     class Par extends Object {
10        Object fst;
11        Object snd;
12
13        // Constructor:
14        Par(Object fst, Object snd){
15           super();
16           this.fst = fst;
17           this.snd =snd;
18        }
19
20        // Definicion de metodo:
21        Par setfst(Object newfst){
22           return new Par(newfst,this.snd);
23        }
24    }
25
```

Tras el uso de la función *lexer* el intérprete ha transformado todos los caracteres en una lista de tokens, la cual se muestra de la línea 29 a la 41.

```
26      ************************************************************
```

[3] En esta fase sólo se muestra el código contenido en el archivo que ha sido recibido por el intérprete

```
27      ************************ Lexer **************************
28      ************************************************************
29      [Rsv Class,Clase ``A'',Rsv Extends,Clase ``Object'',LlaA,Clase ``A'',
30      ParA,ParC,LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,LlaC,Rsv Class,
31      Clase ``B'',Rsv Extends,Clase ``Object'',LlaA,Clase ``B'',ParA,
32      ParC,LlaA,Rsv Super,ParA,ParC,PuCo,LlaC,LlaC,Rsv Class,
33      Clase ``Par'',Rsv Extends,Clase ``Object'',LlaA,Clase ``Object'',
34      Otro ``fst'',PuCo,Clase ``Object'',Otro ``snd'',PuCo,Clase ``Par'',
35      ParA,Clase ``Object'',Otro ``fst'',Coma,Clase ``Object'',Otro ``snd'',
36      ParC,LlaA,Rsv Super,ParA,ParC,PuCo,Rsv This,Punto,Otro ``fst'',
37      Igual,Otro ``fst'',PuCo,Rsv This,Punto,Otro ``snd'',Igual,
38      Otro ``snd'',PuCo,LlaC,Clase ``Par'',Otro ``setfst'',ParA,
39      Clase ``Object'',Otro ``newfst'',ParC,LlaA,Rsv Return,Rsv NewPal,
40      Clase ``Par'',ParA,Otro ``newfst'',Coma,Rsv This,Punto,Otro ``snd'',
41      ParC,PuCo,LlaC,LlaC]
42
```

De la línea 46 a la 51 se muestra aquello regresado por la función *parser*. Se trata de la interpretación de los tokens anteriores, dando como resultado una lista de expresiones en JPP.

```
43      ************************************************************
44      ************************ Parser *************************
45      ************************************************************
46      [(``Object'',[],(``A'',[],[],[]),[]),(``Object'',[],(``B'',[],[],[]),[]),
47      (``Object'',[(``Object'',``fst'')),(``Object'',``snd'')],(``Par'',
48      [(``Object'',``fst''),(``Object'',``snd'')],[],[(Acceso (Variable ``This'',
49      ``fst''),Variable ``fst''),(Acceso (Variable ``This'',``snd''),
50      Variable ``snd'')]),[(``Par'',``setfst'',[(``Object'',``newfst'')],
51      New (``Par'',[Variable ``newfst'',Acceso (Variable ``This'',``snd'')]))])]
52
```

La siguiente lista de valores booleanos nos indica que las tres clases contenidas en el archivo, cuentan con tipos correctamente definidos.

```
53      ************************************************************
54      ************************ Tipos **************************
55      ************************************************************
56      [True,True,True]
57
```

Finalmente, tras realizar la evaluación correspondiente de la expresión que incluimos en el código directamente y usando las tres clases contenidas en el archivo para la creación de la respectiva tabla de clases, obtenemos el valor resultante de dicha evaluación, el cual en este caso es una instancia de la clase *Object*.

```
58      ************************************************************
59      ********************** Evaluacion **********************
60      ************************************************************
61      Expresion: New (``Object'',[])
62      Clase: Object
63      ************************************************************
64
65      Evaluador>
66
```

Con estas funciones hemos terminado la implementación de JPP, ahora puede recibir programas, verificar sus estructuras, comprobar sus tipos y evaluarlos. Sin embargo, aún no recibimos la expresión que será evaluada en el programa, para ello usaremos la función *main*, cuya finalidad será encontrar dicha expresión recorriendo una cadena en busca el símbolo $. La función *main* utiliza la función auxiliar *mainAux*, la cual, una vez localizado el símbolo $ guarda los siguientes caracteres hasta encontrar otro símbolo $, esa será nuestra expresión a evaluar. Las funciones se muestran a continuación:

```
60      main :: String ->String ->(Expr,String)
61      main _ [] = error ``NO hay expresion para evaluar''
62      main p ('$':resto) = let (e,r) = mainAux ([],resto)
63                           in (e,p++r)
64      main p (x:xs) = main (p++[x]) xs
65
66      mainAux :: (String,String) ->(Expr,String)
67      mainAux (_,[]) = error ``Expresion no termina''
68      mainAux (expr,('$':resto)) = (expresion (lexer expr),resto)
69      mainAux (expr,(x:xs)) = mainAux ((expr++[x]),xs)
70
```

Llevar un registro de lo que ha sido verificado es indispensable, pues esta función debe ejecutarse antes de realizar los análisis del código quitando la expresión y sus marcadores para evitar errores en nuestros módulos.

Finalmente, crearemos la función *jpp*, de igual forma que con la función *jppTipos* ésta también recibirá el nombre del archivo contenedor del programa regresando la evaluación de la expresión correspondiente. Nuestra nueva función seguirá los siguientes pasos:

- Interpretar archivo.
- Obtener expresión a evaluar.
- Quitar comentarios.
- Verificar Sintácticamente.
- Verificar Semánticamente.

- Verificar que todas las clases sean correctas en la tabla de clases
- Verificar que el tipado de la expresión sea correcto.
- Evaluar la expresión.

La función es como sigue:

```
70      jpp :: String ->IO()
71      jpp archivo = do
72        contents <- readFile archivo
73        let (e,p) = main [] contents
74        let sinCom = comentarios p
75        let tokensLex = lexer sinCom
76        let tabla = parser tokensLex
77        if (and (map (okClase tabla) (tabla)))
78        then putStr ((show (eval (tabla,e))++`` : ''
79                      ++(tipode tabla [] e)))
80        else putStr ``Error en los tipos''
81
```

En la línea 72 del código se interpreta el archivo, guardando su contenido en la variable *contents*, posteriormente se hace la separación del programa y de la expresión a evaluar en la línea 73, de la línea 74 a la línea 79 se siguen los pasos anteriores.

Formalmente un programa es una tabla de clases seguida de la expresión a evaluar. De este modo, nuestro archivo *Prueba.jpp* se escribirá así:

```
1       class A extends Object { A() {super(); }}
2
3       class B extends Object { B() {super(); }}
4
5       class Par extends Object {
6           Object fst;
7           Object snd;
8
9           // Constructor:
10          Par(Object fst, Object snd){
11             super();
12             this.fst = fst;
13             this.snd =snd;
14          }
15
16          // Definicion de metodo:
17          Par setfst(Object newfst){
18             return new Par(newfst,this.snd);
19          }
```

```
20     }
21
22     $ new Par(new Object(),new Object()).snd $
23
```

Dando como resultado la expresión evaluada y su tipo:

```
Evaluador> jpp "Prueba.jpp"
New ("Object",[]) : Object
Evaluador>
```

Es sumamente importante obtener el tipo de la expresión antes de evaluar, pues el evaluador funciona correctamente sólo si la expresión se encuentra tipada correctamente. Por ejemplo, enviemos un parámetro a un objeto de la clase `Object` en la expresión a evaluar:

```
$ new Par(new Object(new A()),new Object()) $
```

y obtendremos el respectivo mensaje indicando un error en los parámetros, pues la clase `Object` no recibe ningún parámetro en su constructor:

```
Evaluador> jpp "Prueba.jpp"
New ("Par",[New ("Object",[New ("A",[])]),New ("Object",[])]) :
Program error: Parametros incompatibles en NEW o METODO

Evaluador>
```

A continuación crearemos dos clases de manera que nos ayuden a probar JPP, asegurándonos así de que no hay errores en la implementación. El programa es el siguiente:

```
1      // Ejemplo
2
3      class Auto extends Object{
4
5       Object cuatroRuedas;
6       Object unVolante;
7       Object dosPuertas;
8
9       Auto (Object ruedas,
10            Object volante,
11            Object puertas){
12            super();
13
14       this.cuatroRuedas = ruedas;
15       this.unVolante = volante;
16       this.dosPuertas = puertas;
```

```
17       }
18
19       Object getCuatroRuedas(){
20        return this.cuatroRuedas;
21       }
22      }
23
24      class Chevy extends Auto{
25
26       Object dosFaros;
27
28       Chevy (Object ruedasChevy,
29             Object volanteChevy,
30             Object puertasChevy,
31             Object farosCheby){
32             super(ruedasChevy,
33                   volanteChevy,
34                   puertasChevy);
35
36        this.dosFaros = farosChevy;
37       }
38
39       Chevy getChevy(Auto a){
40        return new Chevy(a.cuatroRuedas,
41                         a.unVolante,
42                         a.dosPuertas,
43                         this.dosFaros);
44       }
45      }
46
47      $
48      ((Auto)new Chevy(new Object(),
49                       new Object(),
50                       new Object(),
51                       new Object()).getChevy( new Chevy(new Object(),
52                                                         new Object(),
53                                                         new Object(),
54                                                         new Auto(new Object(),
55                                                                  new Object(),
56                                                                  new Object()
57                                                                 ).getCuatroRuedas()
58                                                        )
59                                              )
60      ).unVolante
61      $
62
```

Resultado:

```
Evaluador> jpp "Ejemplos.jpp"
New ("Object",[]) : Object
```

El programa anterior contruye dos clases: `Auto` y `Chevy` donde la segunda extiende a la primera. En la expresión incluida en el programa para su evaluación se integran la conversión explícita de tipos en la línea 48, la creación de objetos en casi todas las líneas, el acceso a un atributo en la línea 60 y la llamada a un método en las lineas 51 y 57. Dicha expresión y sus sub-expresiones implican una prueba completa para las reglas de evaluación, pues componen todos los posibles casos regresando finalmente un objeto de la clase `Object`. Con esto terminamos las pruebas de JPP y damos por finalizada la estructura del lenguaje. En el capítulo siguiente haremos una extensión que permitirá mejorar el detalle de las pruebas.

4.6. Extensión con números naturales

Los números naturales nos ayudarán a crear ejemplos más interesantes en JPP. Su construcción se hará creando objetos que contienen a su vez objetos, dicho de otro modo, un número se representará por la cantidad de objetos totales de una clase especifica.

```
0 = new Cero(new Object());
1 = new Nat(new Cero(new Object()));
2 = new Nat(new Nat(new Cero(new Object())));
3 = new Nat(new Nat(new Nat(new Cero(new Object()))));
  .
  .
  .
```

En este caso, un número estará representado por la cantidad de objetos de la clase `Nat` que contenga una expresión. En nuestras expresiones también están presentes las clases `Object` y `Cero`. La relación entre estas tres clases será como sigue:

$$\texttt{Cero} \leq \texttt{Nat} \leq \texttt{Object}$$

Es decir, `Object` es la superclase de `Nat` y de `Cero`. `Nat` es la superclase de `Cero`. La razón para usar tres clases es debido a las restricciones de tipos. Si sólo se usaran dos clases, en algún momento los tipos serían incorrectos.

Al construir los números de esta manera la clase `Cero` es subclase de la clase `Nat`, lo que implica que debe cumplir las siguientes condiciones:

- Debe recibir un parámetro en su constructor al igual que la clase `Nat`.

- Dicho parámetro debe ser de un objeto que cumpla la relación de subtipado con el tipo definido en la clase `Nat`, en este caso se trata de la clase `Object`.
- Dicho parámetro en la clase `Cero` no será usado, así que se invocará a la superclase y a ella será dado.

Por lo tanto, si un objeto de la clase `Nat` recibe una instancia de `Object` entonces el constructor de la clase `Cero` deberá recibir también un objeto de la clase `Object`. Tanto `Nat` como `Cero` recibirán genéricamente una instancia de la clase `Object`, lo anterior con la finalidad de no romper con las restricciones de tipos que JPP define.

Las clases serán construidas en un archivo llamado *NatBool.jpp*, donde se incluirá una expresión a evaluar cuando dichas clases estén terminadas. La clase `Nat` se construye como sigue:

```
1      class Nat extends Object{
2
3      Object p;
4
5         Nat(Object n){
6           super();
7           this.p = n;
8         }
9
10     }
11
```

La construcción de la clase `Nat` es de forma recursiva. Dicha clase contiene un atributo de la clase `Object`, de esta forma se permitirá recibir instancias de la misma clase `Nat` cuando el constructor sea invocado, así la clase se define en términos de sí misma. El constructor simplemente inicializará el atributo de clase con el parámetro recibido para que el sucesor y el predecesor de un número natural sean la agregación o sustracción, respectivamente, de una instancia de la clase `Nat`. El método `suc` que regresa el siguiente número natural creará una nueva instancia de la clase `Nat` que contendrá las instancias existentes, su construcción se muestra a continuación:

```
10      Nat suc(){
11        return new Nat(this);
12      }
13
```

Por otro lado el método `pred` quitará una instancia de la clase `Nat` a nuestra expresión, es importante notar que se realizará una *conversión de tipos explícita hacia abajo* lo cual es ilegal en términos de evaluación [4] no así con las reglas de tipado, asumimos entonces que el atributo `p` jamás será una instancia de `Object`, veamos su implementación:

[4] Según las reglas de evaluación de la sección 3.4.6.

```
14      Nat pred(){
15        return (Nat)this.p;
16      }
17
```

Como caso base para la clase `Nat` se creará la clase `Cero`, la cual extiende de la clase `Nat`.

```
56      class Cero extends Nat{
57
58         Cero(Object n){
59           super(n);
60         }
61
62         Nat pred(){
63           return this;
64         }
65
66      }
67
```

La clase `Cero` no define un método `suc`, ya que usa el método `suc` definido en la super clase. En el caso en el que se sobrescriba le método `pred`, se evitará salir del conjunto de los número naturales, ya que la resta no es cerrada en este conjunto.

El constructor de la clase `Cero` debe recibir un parámetro de la misma forma que lo hace la super clase, dicho parámetro será entregado a la super clase al invocar al método *super*.

Para facilitar el uso de los números naturales en JPP necesitamos agregar en nuestro módulo *Lexer.hs* la función *LexerDig*, esta función buscará tokens de la forma $0, 1, 2, 3 \ldots$ y los transformará en objetos de la clase `Nat` o `Cero`. De este modo abreviamos la escritura de los números naturales. La función es como sigue:

```
68      ------------------------------------------------------------------------
69      ----------------------------- NUMEROS ----------------------------------
70      ------------------------------------------------------------------------
71      lexerDig :: String ->[Token]
72      lexerDig palabra =
73               let (digitos,resto) = break (notDigit) palabra
74               in (lexer (num2str (hasDigito 0 digitos))++(lexer resto))
75               where
76               notDigit d = not(isDigit d)
77               aDigito c = (ord c)-(ord '0')
78               hasDigito dig [] = dig
79               hasDigito dig (c:cs) = hasDigito (10*dig + (aDigito c)) cs
80
```

```
81      num2str :: Int ->String
82              num2str 0 = ``new Cero(new Object())''
83              num2str n = (num2str (n-1))++``.suc()''
84
```

La función *lexerDig* utiliza las siguientes funciones auxiliares:

- *notDigit* que indica si un carácter no es un dígito.
- *aDigito* que toma un carácter y regresa su dígito respectivo.
- *hasDigito* que toma un número y una cadena de dígitos, la función aumenta los decimales según la posición del dígito en la cadena.

La función *nat2str* transforma un número entero en elementos de la clase `Cero`, llamando al método `suc` tantas veces como el valor del número. Una vez construida la estructura numérica utilizando las clases `Cero` y `Nat`, se procede a permitir que *lexer* haga la interpretación a la nueva cadena ya sin números, para la cual utilizará la tabla de clase.

Además de esto la función `lexer` deberá cambiar el último caso de su definición a:

```
65      lexer (x:xs) | (isDigit x) = lexerDig (x:xs)
66                   | otherwise = lexerPal (x:xs)
67
```

Para recibir como resultado algo igualmente amigable, agregaremos en el módulo *Evaluador.hs* la función *nat2num* que toma un objeto de la clase `Nat` o `Cero` y lo transforma en un número.

```
70      nat2num :: Expr ->Int
71      nat2num (New (``Cero'',_)) = 0
72      nat2num (New (``Nat'',num)) = 1+(nat2num (head num))
73
```

Y finalmente, llamamos a la función `nat2num` en la función `jpp` así:

```
74      jpp :: String ->IO()
75      jpp archivo = do
76        contents <- readFile archivo
77        let sinCom = comentarios contents
78        let (e,p) = main [] sinCom
79        let tokensLex = lexer p
80        let tabla = parser tokensLex
81        let bien = (and (map (okClase tabla) (tabla)))
82        let resultado = (eval (tabla,e))
83        let tipo = (tipode tabla [] e)
```

```
84        if (bien)
85        then if esNat resultado
86             then putStr ((show (nat2num resultado))++`` : ''++tipo)
87             else putStr ((show resultado)++`` : ''++ tipo)
88        else putStr (``Error en los tipos''++
89                     (show (map (okClase tabla) (tabla))))
90
```

Ahora, ya es posible escribir una expresión dentro del archivo *NatBool.jpp*, por ejemplo escribiendo $ 3 $ como la expresión a evaluar obtenemos:

```
Evaluador> jpp "Nat.jpp"
3 : Nat
Evaluador>
```

Aún siendo números, internamente JPP los considera objetos, por lo cual, no podemos hacer operaciones con dichos números. Las operaciones serán definidas como métodos que transformarán a los objetos de las clases numéricas antes mencionadas. A continuación veremos las definiciones de las operaciones básicas para naturales en cada clase respectivamente:

La operación suma en la clase `Cero` se define como:

```
70      Nat suma(Nat n){
71        return n; -- n+0
72      }
73
```

La operación suma en la clase `Nat` se define como:

```
18      Nat suma(Nat n){
19        return this.pred().suma(n.suc());
20      }
21
```

Nos servimos de la igualdad $n + m = (n - 1) + (m + 1)$, dicha igualdad nos permitirá disminuir un número mientras incrementamos el otro. Haciendo recursión hasta que n tomé el valor de "cero", se obtendrá la suma de los dos números originales, almacenando dicho resultado en la variable m.

Para la operación resta $(-)$ y menor que $(<)$ necesitamos un método auxiliar al que llamaremos *disminuye*, el cual restará dos número entre sí, esto lo hará disminuyendo ambos hasta llegar a "cero" y cuando eso suceda regresará el valor del número pasado como parámetro, por ejemplo $4.disminuye(5)$ regresa 1, dicho método se define en la clase `Cero` como:

```
74      Nat disminuye(Nat n){
75        return n; -- regresa el parametro recibido, pues el invocador es el cero
76      }
77
```

El método *disminuye* en la clase `Nat` se define como:

```
22      Nat disminuye(Nat n){
23        return (this.pred()).disminuye(n.pred());
24        -- Asi se hace recursion en los respectivos
25        -- predecesores de los naturales originales.
26      }
27
```

La operación resta ($-$) se define en la clase `Cero` como:

```
78      Nat resta(Nat n){
79        return this;
80        -- Cero menos el numero recibido como parametro, siempre sera cero
81      }
82
```

La operación resta ($-$) se define en la clase `Nat` como: [5]

```
28      Nat resta(Nat n){
29        return n.disminuye(this);
30      }
31
```

La operación mayor que ($>$) definida en la clase `Cero` se puede ver como sigue:

```
83      Nat mayor(Nat n){
84        return n;
85      }
86
```

Donde cualquier número natural es mayor que el cero, por lo tanto regresaremos simplemente el parámetro recibido como el mayor número natural de nuestra pareja.

La operación mayor que ($>$) en la clase `Nat` se define como:

```
32      Nat mayor(Nat n){
33        return (this.pred().mayor(n.pred())).suc();
34      }
35
```

Obtenemos el predecesor de dos números, tomamos al mayor y conseguimos su sucesor; es decir, el mayor de entre tres y siete será equivalente a tomar el mayor entre dos y seis más uno. Si el menor de estos números es el que invoca al método *mayor*, entonces se obtendrá esa misma cantidad de invocaciones al

[5] Dado que $4.resta(6)$ se define como $4 - 6$, el método *disminuye* tomará los parámetros invertidos, es decir $6.disminuye(4)$, por la manera en que dicho método se encuentra definido.

método *suc*, de lo contrario se obtendrá el natural que ha sido recibido como parámetro menos el que invocó al método *mayor*, después el número recibido como parámetro hará tantas invocaciones al método *suc* como números le fueron restados.

La operación menor que ($<$) en la clase `Cero` se define como:

```
87      Nat menor(Nat n){
88        return this; -- El menor numero natural que obtenemos es el cero
89      }
```

La operación menor que ($<$) en la clase `Nat` se define como:

```
36      Nat menor(Nat n){
37        return (this.mayor(n)).disminuye(this.suma(n));
38      }
39
```

Se hace la suma de ambos números y al resultado de dicha suma se le resta el mayor de los números recibidos. Por ejemplo el menor entre los números cinco y cuatro sería nueve menos cinco.

La operación multiplicación ($*$) se define en la clase `Cero` como:

```
90      Nat multi(Nat n){
91        return this; -- Cero por cualquier numero es cero
92      }
93
```

La operación multiplicación se define en la clase `Nat` como:

```
40      Nat multi(Nat n){
41        return n.suma(this.pred().multi(n));
42        -- Es sumar el parametro a si mismo tantas veces
43        -- como el valor del natural this
44      }
45
```

Para la operación potencia usaremos el método auxiliar *potAux* que ayudará a tomar los parámetros invertidos, esto es ya que $2.pot(3)$ se define como 2^3, y el exponente se disminuirá de uno en uno con la recursión, para obtener $(2 * 2 * 2 * 1)$.

El método auxiliar *potAux* se define en la clase `Cero` como sigue:

```
94      Nat potAux(Nat n){
95        return new Nat(this); -- Regresamos 1
96      }
97
```

El método *potAux* se define en la clase `Nat` como:

```
46      Nat potAux(Nat n){
47        return n.multi(this.pred().potAux(n));
48      }
49
```

Este método lo que hace es multiplicar el parámetro por sí mismo, tantas veces como el valor del número natural `this`, para así obtener el parámetro elevado a la potencia `this`.

La operación potencia en la clase `Nat` se define como:

```
50      Nat pot(Nat n){
51        return n.potAux(this);
52      }
53
```

Como se mencionó anteriormente se intercambia el orden de los números para ser coherente con la interpretación de la función potencia, es decir, p^q es elevar p a la potencia ^{q}. No es necesario definir este método en la clase `Cero`, ya que no se cambiará su definición, pues en cualquier caso llamará al método *potAux*, y por lo tanto se usará la definición descrita en la superclase.

A continuación se presentan algunas pruebas para verificar que los resultados de las operaciones definidas para los números naturales regresan resultados correctos, es decir, las operaciones hacen cálculos según las definiciones de los números naturales.

Al ejecutar en el intérprete la instancia $14.suma(110)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
124 : Nat
Evaluador>
```

Donde $14 + 110 = 124$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $214.suma(110)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
324 : Nat
Evaluador>
```

Donde $214 + 110 = 324$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $214.resta(110)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
104 : Nat
Evaluador>
```

Donde $214 - 110 = 104$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $14.resta(110)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
0 : Nat
Evaluador>
```

Donde $14 - 110 = 0$ en el conjunto de los números naturales.

Al ejecutar en el intérprete la instancia $14.multi(11)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
154 : Nat
Evaluador>
```

Donde $14 * 11 = 154$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $1.multi(77)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
77 : Nat
Evaluador>
```

Donde $1 * 77 = 77$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $40.multi(0)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
0 : Nat
Evaluador>
```

Donde $40 * 0 = 0$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $40.pot(0)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
1 : Nat
Evaluador>
```

Donde $40^0 = 1$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $0.pot(101)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
0 : Nat
Evaluador>
```

Donde $0^{101} = 0$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $2.pot(3)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
8 : Nat
Evaluador>
```

Donde $2^3 = 8$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $3.pot(3)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
27 : Nat
Evaluador>
```

Donde $3^3 = 27$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $1.mayor(3)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
3 : Nat
Evaluador>
```

Donde el mayor entre 1 y $3 = 3$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $12.mayor(3)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
12 : Nat
Evaluador>
```

Donde el mayor entre 12 y $3 = 12$ y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $4.menor(0)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
0 : Nat
Evaluador>
```

Donde el menor entre 4 y 0 = 0 y el resultado también es un número natural.

Al ejecutar en el intérprete la instancia $14.menor(110)$ se puede observar el siguiente resultado:

```
Evaluador> jpp "Nat.jpp"
14 : Nat
Evaluador>
```

Donde el menor entre 14 y 110 = 14 y el resultado también es un número natural.

Las ejecuciones anteriores demuestran el buen funcionamiento de los objetos en el intérprete, pues al modelar a los números naturales se ha completado con la lectura de números en el código, la construcción de clases, construcción de métodos, invocación de métodos, invocación de atributos, así como el uso de herencia en dichas clases. Hasta aquí hemos construido las operaciones necesarias para realizar cálculos simples y de este modo comenzar a utilizar JPP, completando satisfactoriamente las pruebas que con los números naturales nos es posible realizar.

4.7. Extensión con booleanos

Para la creación de valores booleanos nos serviremos de los números naturales:

```
false = new Boolean(0);
true = new Boolean(1);
```

Al crear un objeto de la clase `Boolean` esta instancia recibirá un objeto de la clase `Nat`, el cual definirá el valor de nuestro objeto `Boolean`, si se tratase de un cero entonces nuestro objeto `Boolean` representará al valor *false* y si se tratara de un uno representará al valor *true*.

La construcción de la clase `Boolean` necesita de las clases definidas para los números naturales, para lo que las siguientes definiciones estarán en el mismo archivo *NatBool.jpp*. Nuestra nueva clase se definirá como sigue:

```
108     class Boolean extends Object{
109
110     Nat valor;
111
112         Boolean(Nat b){
113           super();
114           this.valor = b;
115         }
```

```
116
117         Boolean true(){
118           return new Boolean(1);
119         }
120
121         Boolean false(){
122           return new Boolean(0);
123         }
124
125     }
126
```

Siendo que `Boolean` extiende de `Object`. Se tiene como atributo de clase un número natural y el constructor simplemente se inicializa con el parámetro recibido. Los métodos `true` y `false` regresan una instancia de `Boolean` (verdadero o falso) según sea el caso.

A continuación se creará un nuevo método para los naturales, este método *igual* indicará si dos números naturales son iguales entre sí. Para ello, necesitamos de un método auxiliar llamado *igualAux* que nos servirá para asegurar que el objeto que hace la llamada al método *igual* sea el menor de los dos, la definición de este método en la clase `Cero` se puede observar a continuación:

```
98      Boolean igualAux(Nat n){
99        return new Boolean(this.pot(n));
100     }
101
```

Antes de entrar a la explicación del porqué se definirá de esta manera la igualdad en la clase `Cero` veamos la definición para la clase `Nat`:

```
125     Boolean igual(Nat n){
126       return this.menor(n).igualAux(this.mayor(n));
127     }
128
129     Boolean igualAux(Nat n){
130       return this.pred().igualAux(n.pred());
131     }
132
```

Siendo `Nat` una super clase, basta con definir dentro de ésta el método *igual*.

El método *igualAux* será siempre llamado por el menor número natural que tengamos que comparar, esto nos asegurará que si los números no fuesen iguales entonces el que se recibe como parámetro nunca llegará a ser cero, y sin importar que tan grande sea la diferencia sabemos que sólo si ambos naturales son iguales se tiene que el parámetro recibido también es un cero. De este modo se van

disminuyendo a la par para finalmente regresar el cero elevado a la potencia n, donde n es el valor del parámetro.

Si es un cero, entonces el método `potencia` devuelve un uno, pues cualquier número elevado a la potencia cero da como resultado uno, incluso el cero. Si es un número diferente de cero entonces se multiplicará cero n veces por si mismo, lo que evidentemente devuelve cero, para finalmente crear el valor booleano.

Se modelará el operador *if* haciendo uso de los números naturales, para que la decisión solicitada al método *if*, sea resuelta por el objeto que manda llamar a dicho método. Si el método *if* es llamado desde un valor `Nat` entonces devolvemos el primer parámetro del método, en cualquier otro caso regresamos el segundo parámetro, lo anterior está dado por la estructura que forma al operador, la cual se expresa a continuación:

`if * then * else *.` [6].

El método *if* en la clase Cero:

```
102     Object if(Object t, Object f){
103       return f;
104     }
105
```

El número natural cero visto como un valor booleano representa al valor falso.

El método *if* en la clase Nat se presenta a continuación:

```
54     Object if(Object t, Object f){
55       return t;
56     }
57
```

El número natural uno visto como un valor booleano representa el valor de verdadero.

El método *if* en la clase Boolean se presenta a continuación:

```
133     Object if(Object t, Object f){
134       return this.valor.if(t,f);
135     }
```

El valor del booleano mandará llamar al método *if*.

Con lo anterior es posible evaluar expresiones como la siguiente:

```
$ 12.igual(11).if(3.pot(3),2.multi(7)) $
```

[6] En los asteriscos (*) se incluyen expresiones

Visto de otro modo es la multipicación del dos con el siete, ya que doce no es igual a once:

```
Evaluador> jpp "NatBool.jpp"
14 : Object
```

Las líneas anteriores muestran como el tipo de la expresión cambió a `Object`, esto se debe a que el método *if* recibe parámetros de tipo `Object` para asegurar la generalidad de nuestro método. Lo anterior indica que el método `if` modela al IF usado comúnmente en otros lenguajes, pero no actúa igual, pues es posible que nuestro método regrese dos tipos totalmente diferentes según el valor de la condicional del *if*, lo cual no crea un error de tipos en nuestro lenguaje. Sin embargo, sólo se trata de una aproximación al operador IF que cumple con nuestro propósito.

4.8. La sucesión de Fibonacci

Como otro ejemplo del uso del intérprete JPP, implementaremos un programa que calcule la sucesión de Fibonacci, con el cual concluiremos las demostraciones de uso del intérprete.

A continuación se muestra el algoritmo para calcular la sucesión de Fibonacci:

- fibo de 0 = 0
- fibo de 1 = 1
- fibo de n = fibo de (n-2) + fibo de (n-1)

Lo primero que haremos será crear una nueva clase en el mismo archivo donde fueron definidos los número naturales y los booleanos, esto es debido a que necesitamos que se encuentren en la misma tabla de clases. Dicha clase es como sigue:

```
138     class Fibo extends Object{
139
140        Fibo(){
141          super();
142        }
143
144        Object fibo(Nat n){
145          return n.igual(0).
146                   if(0,n.igual(1).
147                          if(1,((Nat)new Fibo().fibo(n.resta(2)))
148                          .suma((Nat)new Fibo().fibo(n.resta(1)))));
149        }
```

```
150     }
```

Finalmente basta con llamar al método solicitando el número de la sucesión que queremos calcular:

```
$new Fibo().fibo(12)$
```

Para este ejemplo se solicitaron los primeros 13 números de la sucesión lo cual da como resultado el respectivo número de Fibonacci en cada llamada. Se modificó el número de la expresión anterior según cada caso:

```
Evaluador> jpp "NatBool.jpp"
0 : Object
Evaluador> jpp "NatBool.jpp"
1 : Object
Evaluador> jpp "NatBool.jpp"
1 : Object
Evaluador> jpp "NatBool.jpp"
2 : Object
Evaluador> jpp "NatBool.jpp"
3 : Object
Evaluador> jpp "NatBool.jpp"
5 : Object
Evaluador> jpp "NatBool.jpp"
8 : Object
Evaluador> jpp "NatBool.jpp"
13 : Object
Evaluador> jpp "NatBool.jpp"
21 : Object
Evaluador> jpp "NatBool.jpp"
34 : Object
Evaluador> jpp "NatBool.jpp"
55 : Object
Evaluador> jpp "NatBool.jpp"
89 : Object
Evaluador> jpp "NatBool.jpp"
144 : Object
Evaluador>
```

Recordemos que la construcción del IF en este caso regresa un tipo `Object`, lo cual nos permite demostrar el uso de la **conversión explícita de tipos**.

En este capítulo se ha completado la implementación del intérprete Java Peso Pluma al crear un grupo de clases que juntas, permiten construir un programa que calcula la sucesión de Fibonacci, como lo muestran los resultados anteriores. Este no es el límite, pues como se demostró en el capítulos 3 con

la formalización del lenguaje, ahora es posible crear programas tan complejos como nuestras necesidades lo soliciten.

Las clases creadas para demostrar la capacidad del intérprete, trazan un grupo de funcionalidades que en sí mismas dan la certeza de haber concluido satisfactoriamente la construcción de dicho intérprete.

Vale la pena recordar las definiciones hechas en el capítulo 3.4.3, funciones que identifican a los atributos de una clase, el tipado de un método y el cuerpo de un método. Así como, las reglas de tipado expuestas en el capítulo 3.4.4 que comprenden el tipado de las variables, acceso a atributos, invocación de métodos, creación de objetos y conversión de tipos explícita. Las reglas de evaluación del capítulo 3.4.6 donde se incluye el acceso a atributos, la invocación de métodos, conversión explicita de tipos y el orden de la evaluación. Finalmente, las demostraciones del capítulo 3.5 las cuales han proporcionado gran parte de las bases de éste trabajo, que al seguir la formalización del lenguaje al pie de la letra termina por entregar un intérprete construido de la mejor manera.

Capítulo 5

Conclusiones

La construcción de un lenguaje de programación, es decir su diseño e implementación, propone un reto que combina análisis y destreza, toda vez que se desarrolla un sistema es menester identificar las funcionalidades y el comportamiento esperados para lograr un diseño capaz de cubrir las metas propuestas. Lograr dichas metas se torna difícil en un sistema con la complejidad contenida en un lenguaje de programación real, por ello, es necesario tener claro los alcances que el lenguaje debe tener, para que finalmente por medio de la interfaz resulte fácil el comunicarnos con un sistema de cómputo.

En este trabajo se construyó un lenguaje prototipo del paradigma orientado a objetos, en partícular del lenguaje Java, el cual toma algunos de los aspectos más importantes de éste, para modelarlos y formalizarlos, reconstruyendo así paso a paso los fundamentos de dicho lenguaje.

Nuestro lenguaje Java Peso Pluma se encuentra ligado directamente al lenguaje JAVA, lo cual provee clases, métodos, atributos, herencia y conversión explícita de tipos. Estos mecanismos son similares a los del lenguaje original permitiendo una correspondencia en el sentido de que todo programa en JPP que no utiliza una conversión de tipos insegura es un programa correcto en JAVA. Se excluyen muchos elementos característicos de JAVA como asignación, interfaces, mensajes a `super` y tipos básicos entre otros, con la finalidad de estudiar el núcleo del lenguaje y sobre él desarrollar un sistema que cuenta con una formalización que JAVA no permite.

La formalización del lenguaje se presentó con los temas referentes al sistema de tipos y subtipos descritos en el capítulo 2, en este trabajo se realizó un análisis general de los tipos, sus características y los posibles errores que pudieran generar; además, se implementaron las reglas de tipado para Java Peso Pluma. En el capítulo 3 y utilizando lo descrito anteriormente, se definió completamente nuestro lenguaje prototipo JPP, al fijar con claridad la sintaxis, el tipado, subtipado y semántica estática del mismo, así como la semántica operacional que se realiza sobre las expresiones del lenguaje. Se presentó de manera formal las demostraciones de los teoremas de Progreso y Preservación de tipos. El primero asegura que una expresión bien tipada siempre se evaluará de manera normal, es

decir, si la expresión se encuentra bien tipada no habrá errores en tiempo de ejecución. El segundo asegurando que aquellos programas tipados correctamente, durante los cambio en las expresiones, se mantendrán correctamente tipados.

En nuestra opinión la aportación más importante de este trabajo consiste en la implementación del lenguaje, pues si bien la formalización de Java Peso Pluma se puede localizar en documentos como artículos y libros en los que se estudian a detalle los aspectos que construyen al lenguaje, sin embargo actualmente no se tiene una implementación completa de éste. Es por esto que consideramos como un aporte significativo el haber no sólo formalizado sino implementado al lenguaje Java Peso Pluma. Aunado a lo anterior, otra aportación que consideramos importante es la implementación de los números naturales y los valores booleanos, puesto que nuestro prototipo no contaba con tipos primitivos, lo cual dificultaba mostrar ejemplos naturales de programas sencillos. Esta implementación permite ahora incluir ejemplos prácticos que permiten un mejor entendimiento del lenguaje. Debido a esto consideramos que este trabajo cuenta con la información necesaria para servir como material de apoyo en cursos de lenguajes de programación.

"Para que el cultivo de la historia de la ciencia adquiera cabal sentido y rinda todos los frutos que promete, se impone el examen de ciertas coyunturas, propias del desenvolvimiento científico. La revolución científica es quizá la circunstancia en que el desarrollo de la ciencia exhibe su plena peculiaridad, sin que importe gran cosa de qué materia se trate o la época considerada."

T.S. Kuhn [7]

La ciencia es una disciplina tan poderosa y a la vez tan sencilla que permite a cualquier persona formar parte de sus filas. Ella no pide fé ciega, o amor incondicional, porque la fé será recompensada con cada triunfo proveniente de ella y al hacer nuestro trabajo con amor no necesitamos nada más.

Apéndice A

Código fuente

A.1. Lexer.hs

```
1      module Lexer where
2
3      import Char
4
5      data Token = Punto             -- .
6                 | Coma              -- ,
7                 | PuCo              -- ;
8                 | ParA              -- (
9                 | ParC              -- )
10                | LlaA              -- {
11                | LlaC              -- }
12                | Igual             -- =
13                | Rsv PalabraRsv    -- Palabras reservadas
14                | Clase String
15                | Otro String
16                  deriving(Show,Eq)
17
18      data PalabraRsv = Class
19                      | Extends
20                      | Super
21                      | Return
22                      | This
23                      | NewPal
24                      deriving (Show,Eq)
25
26      lexerPal :: String ->[Token]
27
28      lexerPal palabra =
```

```
29              let (inicio,resto) = break (notAlpha) palabra
30
31                  reservada  = case inicio of
32                              ``new''      ->Rsv NewPal
33                              ``this''     ->Rsv This
34                              ``super''    ->Rsv Super
35                              ``class''    ->Rsv Class
36                              ``return''   ->Rsv Return
37                              ``extends''  ->Rsv Extends
38                              otherwise    ->checa inicio
39
40              in reservada:(lexer resto)
41
42              where notAlpha t = not(isAlpha t)
43                  checa (x:xs)| isUpper x = Clase (x:xs)
44                              | otherwise = Otro (x:xs)
45
46      ------------------------------------------------------------------------
47      ---------------------------- NUMEROS -----------------------------------
48      ------------------------------------------------------------------------
49      lexerDig :: String ->[Token]
50      lexerDig palabra =
51              let (digitos,resto) = break (notDigit) palabra
52              in (lexer (num2str (hasDigito 0 digitos))++(lexer resto))
53              where
54              notDigit d = not(isDigit d)
55              aDigito c = (ord c)-(ord '0')
56              hasDigito dig [] = dig
57              hasDigito dig (c:cs) = hasDigito (10*dig + (aDigito c)) cs
58
59      num2str :: Int ->String
60      num2str 0 = ``new Cero(new Object())''
61      num2str n = (num2str (n-1))++``.suc()''
62
63
64      lexer :: String ->[Token]
65
66      lexer [] = []
67
68      lexer ('.':xs) = Punto:(lexer xs)
69      lexer (',':xs) = Coma:(lexer xs)
70      lexer (';':xs) = PuCo:(lexer xs)
71      lexer ('(':xs) = ParA:(lexer xs)
72      lexer (')':xs) = ParC:(lexer xs)
73      lexer ('{':xs) = LlaA:(lexer xs)
74      lexer ('}':xs) = LlaC:(lexer xs)
```

```
75      lexer ('=':xs) = Igual:(lexer xs)
76
77
78      lexer ('\n':xs) = (lexer xs)
79      lexer ('\r':xs) = (lexer xs)
80      lexer (' ':xs) = (lexer xs)
81
82      lexer (x:xs) | (isDigit x) = lexerDig (x:xs)
83                   | otherwise = lexerPal (x:xs)
84
```

A.2. Parser.hs

```
1     module Parser where
2
3     import Lexer
4
5     type NomClase = String
6     type NomMetodo = String
7     type NomObjeto = String
8     type NomAtributo = String
9     type NomVariable = String
10    type Objeto = (NomClase,NomObjeto)
11
12     -- « Expresiones »
13
14     data Expr = New (NomClase,[Expr])           -- new C(e)
15               | Cast (NomClase,Expr)            -- (C)e
16               | Acceso (Expr,NomAtributo)       -- e.atr
17               | Metodo (Expr,NomMetodo,[Expr])  -- e.m(e1,...,en)
18               | Variable NomVariable            -- variable
19                 deriving(Show,Eq)
20
21
22    type MetodoDecl = (NomClase,NomMetodo,[Objeto],Expr)
23
24
25    type ConstructorDecl = (NomClase,[Objeto],[Expr],[(Expr,Expr)])
26
27
28    type ClassDecl = (NomClase,[Objeto],ConstructorDecl,[MetodoDecl])
29
30
31    type TablaClases = [ClassDecl]
32
33     objetos :: [Token] ->([Objeto],[Token])
34
35     objetos (Clase nomC:Otro nomO:Coma:resto) =
36       let (varsM,resV) = objetos resto
37       in ((nomC,nomO):varsM,resV)
38
39     objetos (Clase nomC:Otro nomO:ParC:LlaA:resto) =
40       ([(nomC,nomO)],resto)
41
```

```
42  objetos (ParC:LlaA:resto) = ([],resto)
43
44  objetos _ = error ``error en parametros''
45
46  cortaPP :: ([Token],[Token]) ->Int ->([Token],[Token])
47  cortaPP (td,(ParC:ts)) 1 = (td,ts)
48  cortaPP (td,(ParC:ts)) n = cortaPP (td++[ParC],ts) (n-1)
49  cortaPP (td,(ParA:ts)) n = cortaPP (td++[ParA],ts) (n+1)
50  cortaPP (td,(x:ts)) n = cortaPP (td++[x],ts) n
51  cortaPP (_,[]) _ = error (``cortaPP ->Parentesis mal estructurados'')
52
53  cambiaCorte :: ([Token],[Token]) ->Expr
54  cambiaCorte (e1,[]) = expresion e1
55  cambiaCorte (e1,[Punto,Otro atr]) = Acceso (expresion e1,atr)
56  cambiaCorte (e1,(Punto:Otro atr:Punto:resto)) =
57        cambiaCorte (e1++[Punto,Otro atr],(Punto:resto))
58  cambiaCorte (e1,(Punto:Otro met:ParA:resto)) =
59        let (e3,e4) = cortaPP ([],resto) 1
60        in (if e4==[]
61            then Metodo (expresion e1,met,map (expresion) (separaExpr e3))
62            else cambiaCorte (e1++[Punto,Otro met,ParA]++e3++[ParC],e4))
63  cambiaCorte (e1,_) = error (``cambiaCorte ->Expresion mal estructurada'')
64
65  expresion :: [Token] ->Expr
66  expresion (ParA:Clase c:ParC:resto) = Cast (c,expresion resto)
67  expresion (ParA:resto) = cambiaCorte (cortaPP ([],resto) 1)
68  expresion [Otro var] = Variable var
69  expresion [Rsv This] = Variable ``This''
70  expresion (Rsv This:resto) = cambiaCorte ([Rsv This],resto)
71  expresion (Otro var:resto) = cambiaCorte ([Otro var],resto)
72  expresion (Rsv NewPal:Clase c:ParA:resto) =
73      let (e1,e2) = cortaPP ([],resto) 1
74      in (if e2==[]
75          then New (c,map (expresion) (separaExpr e1))
76          else cambiaCorte ([Rsv NewPal,Clase c,ParA]++e1++[ParC],e2))
77  expresion _ = error (``expresion ->Expresion mal estructurada'')
78
79  separaExpr :: [Token] ->[[Token]]
80  separaExpr [] = []
81  separaExpr expr = let (exp,exps) = separaExprAux ([],expr) 0
82                    in exp:separaExpr exps
83
84  separaExprAux :: ([Token],[Token]) ->Int ->([Token],[Token])
85  separaExprAux (e,[]) 0 = (e,[])
86  separaExprAux (e,[]) _ = error (``Faltan parentesis''++(show e))
87  separaExprAux (e,(Coma:res)) 0 = (e,res)
88  separaExprAux (e,(ParA:res)) n = separaExprAux (e++[ParA],res) (n+1)
89  separaExprAux (e,(ParC:res)) n = separaExprAux (e++[ParC],res) (n-1)
90  separaExprAux (e,(r:rs)) n = separaExprAux (e++[r],rs) n
91
92  expS :: [Token] ->([Expr],[Token])
93  expS (Rsv Super:ParA:resto) = let (exprs,resB) = break (==PuCo) resto
94                                in (expSaux (init exprs),tail resB)
96  expS _ = error ``Super no encontrado''
97
98  expSaux :: [Token] ->[Expr]
99  expSaux tok = map (expresion) (separaExpr tok)
```

```
100
101     asignaciones :: [Token] ->([(Expr,Expr)],[Token])
102     asignaciones tokens = let (asignT,resto) = break (==LlaC) tokens
103                           in (asignacionesAux asignT,(tail resto))
104
105     asignacionesAux :: [Token] ->[(Expr,Expr)]
106     asignacionesAux [] = []
107     asignacionesAux asig =
108         let (a,as) = break (==PuCo) asig
109             (e1,e2) = break (==Igual) a
110         in (expresion e1,expresion (tail e2)):(asignacionesAux (tail as))
111
112     metodo :: [Token] ->MetodoDecl
113     metodo (Clase nomC:Otro nomM:ParA:resto) =
114        let (varsM,resV) = objetos resto
115        in (nomC,nomM,varsM,exprM resV)
116
117     exprM :: [Token] ->Expr
118     exprM (Rsv Return:resto) =
119        let (exp,vac) = break (==PuCo) resto
120        in case vac of
121           [PuCo] -> expresion exp
122           _ -> error ``Error en las llaves de los metodos''
123
124     parseMaux :: ([MetodoDecl],[Token]) ->([MetodoDecl],[Token])
125     parseMaux (md,tok) =
126        let (pm,resto) = break (==LlaC) tok
127        in case pm of
128           [] -> (md,(tail resto))
129           _ -> parseMaux (md++[metodo pm],(tail resto))
130
131     -- PARSER
132     parser :: [Token] ->TablaClases
133     parser [] = []
134     parser (Rsv Class:Clase nomC:Rsv Extends:Clase nomE:LlaA:resto) =
135         let (objs,resO) = parseO resto
136             (cons,resC) = parseC resO
137             (mets,resM) = parseM resC
138         in (if (uno cons)==nomC
139             then (nomE,objs,cons,mets):(parser resM)
140             else error ``Clase y constructor no coinciden...'')
141     parser e = error (``Esperaba la definicion de clase nueva''++(show e))
142     uno (x,_,_,_) = x
143
144     -- PARSER OBJETOS
145     parseO :: [Token] ->([Objeto],[Token])
146     parseO (Clase nomC:Otro nomO:PuCo:resto) = let (objs,resO) = parseO resto
147                                                in ((nomC,nomO):objs,resO)
148     parseO resto = ([],resto)
149
150     -- PARSER CONSTRUCTOR
151     parseC :: [Token] ->(ConstructorDecl,[Token])
152
153     parseC (Clase nomC:ParA:resto) = let (varsC,resV) = objetos resto
154                                          (expSs,resS) = expS resV
155                                          (asigns,resC) = asignaciones resS
156                                      in ((nomC,varsC,expSs,asigns),resC)
```

```
157     parseC _ = error ``error antes de constructor''
158
159     -- PARSER METODOS
160     parseM :: [Token] ->([MetodoDecl],[Token])
161     parseM mts = parseMaux ([],mts)
162
```

A.3. VerificadorDeTipos.hs

```
1     module VerificadorDeTipos where
2
3     import Lexer
4     import Parser
5
6     type TipoDecl = (Expr,NomClase)
7
8     type Ctx = [TipoDecl]
9
10     -- « ATRIBUTOS »
11
12     atributos :: TablaClases ->NomClase ->[Objeto]
13
14     atributos _ ``Object'' = []
15
16     atributos tabla clase = (atributos tabla sClase)++(atrClase tabla clase)
17         where sClase = superClase tabla clase
18
19     -- Obtenemos los atributos de la clase indicada
20     atrClase :: TablaClases ->NomClase ->[Objeto]
21     atrClase [] clase = error (clase++'': Clase no encontrada'')
22     atrClase ((_,atr,(claseH,_,_,_),_):resto) clase
23         | (clase==claseH) = atr
24         | otherwise = atrClase resto clase
25
26     -- Obtenemos el nombre de la super clase de la clase indicada
27     superClase :: TablaClases ->NomClase ->NomClase
28     superClase _ ``Object'' = ``Object''
29     superClase [] clase = error (clase++'': Clase no encontrada'')
30     superClase ((claseP,_,(claseH,_,_,_),_):resto) clase
31         | (clase==claseH) = claseP
32         | otherwise = superClase resto clase
33
34     -- « TIPOMETODO »
35
36     tipoMetodo :: TablaClases ->NomMetodo ->NomClase ->([NomClase],NomClase)
37
38     tipoMetodo _ metodo ``Object'' = error (metodo++'': Metodo no encontrado'')
39
40     tipoMetodo tabla metodo clase | (mEnLista metodo mc) =
41                                       let (t,m,atr,e) = mLista metodo mc
42                                       in (tipoObj atr,t)
43                                   | otherwise =
44               tipoMetodo tabla metodo (superClase tabla clase)
45         where mc = (metodosClase tabla clase)
46
47     -- Obtenemos los metodos de la clase indicada
```

```
48      metodosClase :: TablaClases ->NomClase ->[MetodoDecl]
49      metodosClase _ ``Object'' = []
50      metodosClase [] clase = error (clase++'': Clase no encontrada'')
51      metodosClase ((_,_,(claseH,_,_,_),met):resto) clase
52          | (clase==claseH) = met
53          | otherwise = metodosClase resto clase
54
55      -- Obtenemos el metodo indicado de la lista (no usa caso base)
56      mLista :: NomMetodo ->[MetodoDecl] ->MetodoDecl
57      mLista met ((t,m,atr,e):resto) | met==m = (t,m,atr,e)
58                                     | otherwise = mLista met resto
59
60      -- Buscamos el metodo indicado en la lista
61      mEnLista :: NomMetodo ->[MetodoDecl] ->Bool
62      mEnLista _ [] = False
63      mEnLista met ((_,m,_,_):resto) = (met==m)||(mEnLista met resto)
64
65      -- Obtenemos el tipo de los objetos
66      tipoObj :: [Objeto] ->[NomClase]
67      tipoObj [] = []
68      tipoObj ((cn,_):resto) = cn:(tipoObj resto)
69
70      -- « CUERPOMETODO »
71
72      cuerpoMetodo :: TablaClases ->NomMetodo ->NomClase ->([NomObjeto],Expr)
73
74      cuerpoMetodo _ metodo ``Object'' = error (metodo++'': Metodo no encontrado'')
75
76      cuerpoMetodo tabla metodo clase | (mEnLista metodo mc) =
77                                        let (t,m,atr,e) = mLista metodo mc
78                                        in (nomObj atr,e)
79                                      | otherwise =
80                  cuerpoMetodo tabla metodo (superClase tabla clase)
81          where mc = (metodosClase tabla clase)
82
83      -- Obtenemos el nombre de los objetos
84      nomObj :: [Objeto] ->[NomObjeto]
85      nomObj [] = []
86      nomObj ((_,obj):resto) = obj:(nomObj resto)
87
88      tipode :: TablaClases ->Ctx ->Expr->NomClase
89
90      tipode _ [] (Variable x) = error (``Variable ``++x++
91                                        `` ->Tipo indefinido en el contexto'')
92
93      tipode tabla ((e,cn):ts) (Variable x)
94          | e==(Variable x) = cn
95          otherwise = tipode tabla ts (Variable x)
96
97      tipode tabla ctx (Acceso (e,on)) =
98          let tipoExp = tipode tabla ctx e
99              objetos = (atributos tabla tipoExp)
100             (claseN,objN) = daObjeto objetos on
101         in claseN
102
103     tipode tabla ctx (Metodo (exp,metN,exps)) =
104         let tipoExp = tipode tabla ctx exp
```

```
105             tipoExps = map (tipode tabla ctx) exps
106             (rec,reg) = tipoMetodo tabla metN tipoExp
107         in if (subclases tabla tipoExps rec)
108            then reg
109            else error (`` tipos incompatibles en el metodo ->``
110            ++metN++(show rec)++(show tipoExps))
111
112     tipode tabla ctx (New (clase,exps)) =
113         let objetos = (atributos tabla clase)
114             tipoExps = map (tipode tabla ctx) exps
115             tipoObjs = (map fst objetos)
116         in if (subclases tabla tipoExps tipoObjs)
117            then clase
118            else error (`` tipode NEW `` ++
119                        (show tipoExps)++
120                        `` </:'' ++
121                        (show tipoObjs))
122
123     tipode tabla ctx (Cast (cn,expr)) =
124         let tipoExpr = tipode tabla ctx expr
125         in if (subclase tabla cn tipoExpr)
126            then cn -- conversion explicita de tipos hacia arriba
127            else if(subclase tabla tipoExpr cn)&&(cn /= tipoExpr)
128                 then cn -- conversion explicita de tipos hacia abajo
129                 else cn -- conversion explicita de tipos estupida
130
131      -- Obtenemos el objeto solicitado
132      daObjeto :: [Objeto] ->NomObjeto ->Objeto
133      daObjeto [] o = error (``objeto ``++o++'' no encontrado'')
134      daObjeto ((cn,on):os) obj | on==obj = (cn,on)
135                                | otherwise = daObjeto os obj
136
137      -- C <: D
138      subclase :: TablaClases ->NomClase ->NomClase ->Bool
139      subclase _ ``Object'' ``Object'' = True
140      subclase _ ``Object'' _ = False
141      subclase tabla c d = (c==d)|| -- reflexividad
142                           (subclase tabla (superClase tabla c) d)
143                           -- La transitividad se cumple en la recursion
144
145      -- C's <: D's
146      subclases :: TablaClases ->[NomClase] ->[NomClase] ->Bool
147      subclases _ [] [] = True
148      subclases tabla (x:xs) (y:ys) = (subclase tabla x y)&&
149                                      (subclases tabla xs ys)
150      subclases _ _ _ = error
151                        ``Parametros incompatibles en NEW o METODO''
152
153      okMetodo :: TablaClases ->NomClase ->MetodoDecl ->Bool
154
155      okMetodo tabla c (regr,nomMet,objetos,expr) |
156               (elem (regr,nomMet,objetos,expr) metodosSup) =
157                     let (csms,cms) = (tipoMetodo tabla nomMet supClase)
158                         (csmc,cmc) = (tipoMetodo tabla nomMet c)
159                         ctx = (map (objAtipo) (objetos++[(c,''This'')]))
160                         tipoExpr = (tipode tabla ctx expr)
161                     in (cms==cmc)&&
```

```
162                     (csms==csmc)&&
163                     (subclase tabla tipoExpr cms)
164             | otherwise =
165                   let (csmc,cmc) = (tipoMetodo tabla nomMet c)
166                       ctx = (map (objAtipo) (objetos++[(c,''This'')]))
167                       tipoExpr = (tipode tabla ctx expr)
168                   in (subclase tabla tipoExpr cmc)
169             where supClase = (superClase tabla c)
170                   metodosSup = (metodosClase tabla supClase)
171
172     -- Cambiamos un objeto por un Tipo
173     objAtipo :: Objeto ->TipoDecl
174     objAtipo (cn,on) = (Variable on,cn)
175
176     okClase :: TablaClases ->ClassDecl ->Bool
177
178     okClase tabla (``Object'',atrs,(cn,objs,exprs,exprsIg),metodos) =
179         (exprs == []) &&
180         and (map (okMetodo tabla cn) metodos)
181
182     okClase tabla (supClase,atrs,(cn,objs,exprs,exprsIg),metodos) =
183         (tipoObj (take (length atrSup) objs))==(tipoObj atrSup) &&
184         (take (length gs) gsfs)==gs &&
185         and (map (okMetodo tabla cn) metodos)
186         where gs = (daNomVars exprs)
187               gsfs = map snd objs
188               atrSup = atributos tabla supClase
189
190     daNomVars :: [Expr] ->[NomVariable]
191     daNomVars [] = []
192     daNomVars ((Variable n):vars) = n:(daNomVars vars)
193
194     comentarios :: String ->String
195     comentarios [] = []
196     comentarios ('/':'/':resto) = comentarios (hastaSalto resto)
197     comentarios ('/':'*':resto) = comentarios (hastaCierre resto)
198     comentarios (x:resto) = x:(comentarios resto)
199
200     hastaSalto :: String ->String
201     hastaSalto [] = []
202     hastaSalto ('\n':resto) = resto
203     hastaSalto (_:resto) = hastaSalto resto
204
205     hastaCierre :: String ->String
206     hastaCierre [] = []
207     hastaCierre ('*':'/':resto) = resto
208     hastaCierre (_:resto) = hastaCierre resto
209
210     jppTipos :: String ->IO()
211     jppTipos archivo = do
212       contents <- readFile archivo
213       let sinCom = comentarios contents
214       let tokensLex = lexer sinCom
215       let tabla = parser tokensLex
216       putStr ``\n************************************************************''
217       putStr ``\n************************ Codigo ************************''
218       putStr ``\n************************************************************\n''
```

```
219       putStr contents
220       putStr ``\n\n''
221       putStr ``\n************************************************************''
222       putStr ``\n************************* Lexer *************************''
223       putStr ``\n************************************************************\n''
224       print tokensLex
225       putStr ``\n\n''
226       putStr ``\n************************************************************''
227       putStr ``\n************************* Parser ************************''
228       putStr ``\n************************************************************\n''
229       print tabla
230       putStr ``\n\n''
231       putStr ``\n************************************************************''
232       putStr ``\n************************* Tipos *************************''
233       putStr ``\n************************************************************\n''
234       print (map (okClase tabla) tabla)
235
```

A.4. Evaluador.hs

```
1     module Evaluador where
2
3     import Lexer
4     import Parser
5     import VerificadorDeTipos
6
7     type Programa = (TablaClases,Expr)
8
9     eval :: Programa ->Expr
10    eval (tabla,(Acceso (e0,f))) =
11         let  (New (c,vs)) = eval (tabla,e0)
12              objs = (atributos tabla c)
13              posAtr = daPos f (map snd objs)
14         in vs!!posAtr
15
22     eval (tabla,(Cast (d,e))) =
23          let (New (c,vals)) = eval (tabla,e)
24          in if (subclase c d tabla)
25             then New (c,vals)
26             else error ``cast erroneo''
27
28     eval (tabla,(New (c,es))) = (New (c,(map (evalExp tabla) es)))
29
30     evalExp :: TablaClases ->Expr ->Expr
31     evalExp tabla expr = eval (tabla,expr)
32
33     daPos :: (Eq a, Num b) =>a ->[a] ->b
34     daPos _ [] = error ``no existe elemento''
```

```
35      daPos atr (a:as) | atr==a = 0
36                       | otherwise = 1+(daPos atr as)
37
38      sust :: [NomVariable] ->[Expr] ->Expr ->Expr
39      sust [] [] expr = expr
40      sust (a:as) (e:es) expr = sust as es (sustAux a e expr)
41
42      sustAux :: NomVariable ->Expr ->Expr ->Expr
43      sustAux a e (Variable v) | a==v = e
44                               | otherwise = Variable v
45      sustAux a e (New (cn,exprs)) = New (cn,(map (sustAux a e) exprs))
46      sustAux a e (Cast (cn,expr)) = Cast (cn,(sustAux a e expr))
47      sustAux _ _ (Acceso (Variable ``This'',field)) = Acceso (Variable ``This'',field)
48      sustAux a e (Acceso (expr,field)) = (Acceso ((sustAux a e expr),field))
49      sustAux a e (Metodo (expr,mn,exprs)) =
50          (Metodo ((sustAux a e expr),mn,(map (sustAux a e) exprs)))
51
52      sustThis :: Expr ->Expr ->Expr
53      sustThis expr (Variable ``This'') = expr
54      sustThis expr (New (cn,exprs)) = New (cn,(map (sustThis expr) exprs))
55      sustThis expr (Cast (cn,exprc)) = Cast (cn,(sustThis expr exprc))
56      sustThis expr (Acceso (expra,field)) = (Acceso ((sustThis expr expra),field))
57      sustThis expr (Metodo (exprm,mn,exprs)) =
58          (Metodo ((sustThis expr exprm),mn,(map (sustThis expr) exprs)))
59
60      main :: String ->String ->(Expr,String)
61      main _ [] = error ``NO hay expresion para evaluar''
62      main p ('$':resto) = let (e,r) = mainAux ([],resto)
63                           in (e,p++r)
64      main p (x:xs) = main (p++[x]) xs
65
66      mainAux :: (String,String) ->(Expr,String)
67      mainAux (_,[]) = error ``Expresion no termina''
68      mainAux (expr,('$':resto)) = (expresion (lexer expr),resto)
69      mainAux (expr,(x:xs)) = mainAux ((expr++[x]),xs)
70
71      nat2num :: Expr ->Int
72      nat2num (New (``Cero'',_)) = 0
73      nat2num (New (``Nat'',num)) = 1+(nat2num (head num))
74
75      jpp :: String ->IO()
76      jpp archivo = do
77        contents <- readFile archivo
78        let sinCom = comentarios contents
79        let (e,p) = main [] sinCom
80        let tokensLex = lexer p
```

```
81        let tabla = parser tokensLex
82        let bien = (and (map (okClase tabla) (tabla)))
83        let resultado = (eval (tabla,e))
84        let tipo = (tipode tabla [] e)
85        if (bien)
86        then if esNat resultado
87             then putStr ((show (nat2num resultado))++`` : ''++tipo)
88             else putStr ((show resultado)++`` : ''++ tipo)
89        else putStr (``Error en los tipos''++
90                     (show (map (okClase tabla) (tabla))))
91
```

Apéndice B

Código JPP

B.1. NatBool.jpp

```
1      class Nat extends Object{
2
3         Object p;
4
5         Nat(Object n){
6           super();
7           this.p = n;
8         }
9
10        Nat suc(){
11          return new Nat(this);
12        }
13
14        Nat pred(){
15          return (Nat)this.p;
16        }
17
18        Nat suma(Nat n){
19          return this.pred().suma(n.suc());
20        }
21
22        Nat disminuye(Nat n){
23          return (this.pred()).disminuye(n.pred());
24          -- Asi se hace recursion en los respectivos
25          -- predecesores de los naturales originales.
26        }
27
28        Nat resta(Nat n){
```

```
29          return n.disminuye(this);
30        }
31
32        Nat mayor(Nat n){
33          return (this.pred().mayor(n.pred())).suc();
34        }
35
36        Nat menor(Nat n){
37          return (this.mayor(n)).disminuye(this.suma(n));
38        }
39
40        Nat multi(Nat n){
41          return n.suma(this.pred().multi(n));
42          -- Es sumar el parametro a si mismo tantas veces
43          -- como el valor del natural this
44        }
45
46        Nat potAux(Nat n){
47          return n.multi(this.pred().potAux(n));
48        }
49
50        Nat pot(Nat n){
51          return n.potAux(this);
52        }
53
54        Object if(Object t, Object f){
55          return t;
56        }
57
58    }
59
60    class Cero extends Nat{
61
62        Cero(Object n){
63          super(n);
64        }
65
66        Nat pred(){
67          return this;
68        }
69
70        Nat suma(Nat n){
71          return n; -- n+0
72        }
73
74        Nat disminuye(Nat n){
```

```
75          return n; -- regresa el parametro recibido, pues el invocador es el cero
76        }
77
78        Nat resta(Nat n){
79          return this;
80          -- Cero menos el numero recibido como parametro, siempre sera cero
81        }
82
83        Nat mayor(Nat n){
84          return n;
85        }
86
87        Nat menor(Nat n){
88          return this; -- El menor numero natural que obtenemos es el cero
89        }
90        Nat multi(Nat n){
91          return this; -- Cero por cualquier numero es cero
92        }
93
94        Nat potAux(Nat n){
95          return new Nat(this); -- Regresamos 1
96        }
97
98        Boolean igualAux(Nat n){
99          return new Boolean(this.pot(n));
100       }
101
102       Object if(Object t, Object f){
103         return f;
104       }
105
106     }
107
108     class Boolean extends Object{
109
110       Nat valor;
111
112       Boolean(Nat b){
113         super();
114         this.valor = b;
115       }
116
117       Boolean true(){
118         return new Boolean(1);
119       }
120
```

```
121       Boolean false(){
122         return new Boolean(0);
123       }
124
125       Boolean igual(Nat n){
126         return this.menor(n).igualAux(this.mayor(n));
127       }
128
129       Boolean igualAux(Nat n){
130         return this.pred().igualAux(n.pred());
131       }
132
133       Object if(Object t, Object f){
134         return this.valor.if(t,f);
135       }
136    }
137
138    class Fibo extends Object{
139
140       Fibo(){
141         super();
142       }
143
144       Object fibo(Nat n){
145         return n.igual(0).
146                  if(0,n.igual(1).
147                         if(1,((Nat)new Fibo().fibo(n.resta(2)))
148                         .suma((Nat)new Fibo().fibo(n.resta(1)))));
149       }
150    }
```

Bibliografía

[1] Felleisen, Matthias and Friedman, Daniel P. *A Little Java, A Few Patterns*. Massachusetts: Institute of Technology, 1998.

[2] Haskell página oficial, `http://www.haskell.org/` 30 de Octubre del 2011, 17:30 hrs.

[3] Hudak, Paul. *The Haskell School of Expression: Learning Functional Programming through Multimedia*, UK: Cambridge University Press, 2000. 363 p.

[4] Hutton, Graham. *Programming in Haskell*, UK: Cambridge University Press, 2007. 171 p.

[5] Igarashi, Atsushi. Pierce, Benjamin C. and Wadler, Philip. *Featherweight Java: A Minimal Core Calculus for Java and GJ*. Philadelphia: ACM Transactions on Programming Languages and Systems, 2001. Vol. 23, No. 3, Pages 396-450.

[6] Knowles, Kenneth. *Featherweight Java*, Pennsylvania: Conflict Management and Peace Science 209G, Spring 2005, April 29, 2005.

[7] Kuhn, Thomas S. *La estructura de las revoluciones científicas*. Traducción de Agustín Contin. Argentina: Fondo de cultura económica, 2004. 318 p.

[8] Miranda, Perea Favio E. *Lenguajes de Programación y sus Paradigmas notas de clase*. México: Universidad Nacional Autónoma de México, 2010.

[9] Mitchell, John C. *Concepts in Programming Languages*. UK: Cambridge University Press, 2003. 450 p.

[10] Pierce, Benjamin C. *Types and Programming Languages*. Massachusetts: MIT Press, Cambridge, 2002. 623 p.

[11] RAE (Real Academia Española), `http://buscon.rae.es/draeI/SrvltConsulta?TIPO_BUS=3&LEMA=subsumir` 9 de Enero del 2012, 23:44 hrs.

[12] Turbak, Franklyn and Gifford, David with Sheldon, Mark A. *Design Concepts in Programming Languages* Massachusetts: MIT Press, Cambridge, 2008. 1322 p.