BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

Prototipo para la Notación Z

TESIS

QUE PARA OBTENER EL TÍTULO DE

LICENCIADO EN COMPUTACIÓN

PRESENTA

LÓPEZ ROBLES MIGUEL

ASESOR

LIC. JOSÉ DE JESÚS LAVALLE MARTÍNEZ

Puebla, Pue.

Contenido

Introducción

En los últimos años los sistemas de cómputo han ido substituyendo a aquellos sistemas de control tradicional de tal forma que ahora estos nuevos sistemas de cómputo y principalmente los sistemas de software asociados con ellos se encuentran a cargo del control del tráfico aéreo, líneas de producción, transferencias de capital, misiones espaciales e incluso pruebas nucleares, por ejemplo. Como es de esperarse en muchas de estas actividades se juega la seguridad e integridad física de miles de personas por un lado y por el otro la estabilidad económica de empresas e instituciones financieras. De esta forma la seguridad y confiabilidad que deberán tener estos sistemas de software ha sido un tema preocupante para todos aquellos involucrados en su diseño. En este sentido, se han hecho ya algunos trabajos encaminados al diseño de software seguro y se ha llegado a lo que hoy conocemos como Ingeniería del Software. Dentro de la Ingeniería del Software se cubren facetas que van desde la definición inicial del sistema hasta la producción de la documentación para el sistema completo. De aqui y del estudio del proceso de producción de software se ha desarrollado la idea del ciclo de vida del software el cual se divide (de forma general) en las siguientes etapas :

- Análisis de requerimientos.
- Especificación del sistema.
- Diseño.
- Implantación.
- Integración y prueba.
- Soporte y mejoramiento.

La experiencia en el diseño de grandes proyectos muestra que el gran volumen de los costos de un proyecto de software está fuertemente ligado a la corrección de errores durante las etapas de implantación y prueba. Sin embargo, investigaciones minuciosas indican que estos errores son consecuencia de imprecisiones ocurridas durante las primeras etapas del proyecto, es decir, los grandes costos durante las fases de implantación y prueba son en su mayoría debidos a una falsa percepción del problema que se realiza, durante las fases de especificación y diseño [PoB+91].

Por otra parte, los costos de la corrección de errores en un producto que ya ha sido entregado al cliente son muy elevados [PoB+91]. Esta combinación de costos de rectificación altos y los errores en las primeras etapas del ciclo de vida del software, ha conducido a dar un esfuerzo mejor en el desarrollo de técnicas más poderosas que pueden aplicarse a estas etapas, y así reducir la tasa de errores y como consecuencia, los errores del producto terminado. Los métodos ó técnicas de especificación formal intentan ser una herramienta poderosa para ayudar al diseño de sistemas de software y que se aplicarían en las primeras etapas del ciclo de vida del software, principalmente en la etapa de especificación. Para tales propósitos, los métodos de especificación formal combinan:

- Una notación matemática basada en principios de la teoría de conjuntos y de la lógica de predicados, relaciones y funciones.
- Formalidad. Entenderemos por formalidad a la precisión y capacidad para argumentar las propiedades de un sistema.
- Abstracción. La capacidad de decir lo que se quiere hacer sin dar detalles de como se realizará.

Así se logra que:

- El significado de una especificación sea clara y sin ambigüedades.
- Los conceptos de la notación matemática permiten una mayor precisión que el lenguaje natural, sin sacrificar el detalle que sea requerido.
- Demostrar propiedades del sistema gracias a las técnicas de prueba matemática.

El empleo de estas técnicas está dirigido a la producción de especificaciones claras de las funciones esperadas por un sistema de computo (en general); además el uso de métodos formales se puede ver como una forma de proporcionar un alto grado de confianza a un sistema que será obtenido a partir de su especificación. A la fecha existen varios métodos de especificación formal y entre ellos se encuentra el **lenguaje**

de especificación formal Z o la **Notación Z** o simplemente **Z** como también se conoce.

Este trabajo está encaminado a la producción de un prototipo que de soporte a la Notación Z. El material sobre Z expuesto en [PoB+91] ha sido la base para el desarrollo del mismo. Algunas de las funciones que realizará el prototipo son:

- Revisión de la sintaxis de las especificaciones, énfasis especial se da a la concordancia de "tipos" con el fin de hallar posibles expresiones "sin sentido".

- Determinar las precondiciones de cuando una operación es o no aplicable.

tales funciones son básicas en cualquier prototipo para la notación Z.

En el primer capítulo se da una explicación de la notación Z así como de las teorías en las que se basa y de los conceptos que constituyen una especificación.

A partir del segundo capítulo se aborda la implantación del prototipo para la notación Z, en él que se incluye las fases del análisis léxico y análisis sintáctico.

El tercer capítulo muestra las ideas de la implantación de la comprobación de tipos, la cual se basa en las reglas para asignar tipos.

En el capítulo cuatro se analiza el cálculo y simplificación de precondiciones.

En el capítulo cinco se muestran las pruebas realizadas por el prototipo así como los resultados que se obtienen por el mismo.

Al final se incluye un apéndice que contiene la información sobre los comandos de LaTeX que son usados para la simbología de la notación Z.

Capítulo 1

La Notación Z

La notación Z fue desarrollada por el Grupo de Investigación en Programación (Programming Research Group) del Laboratorio de Cómputo de la Universidad de Oxford.

La notación Z es un lenguaje de especificación basado en estados[WoJ+97]: un sistema está caracterizado por un conjunto de estados entre los que existe un estado inicial, y un conjunto de operaciones que definen las transiciones de un estado a otro. La notación Z es usada para modelar una gran variedad de sistemas secuenciales.

Sustenta su base teórica[PoB+91][WoJ+97] en:

- La teoría de conjuntos, cuyos conceptos permiten la modelación de estructuras de datos.
- La lógica de predicados de primer orden, que se usa para describir de forma abstracta el efecto de cada operación en el sistema.
- Las teorías de relaciones y funciones.

Aún cuando los conceptos de estas teorías son la base de Z, la notación clasica en ellas no es la que se usará. Z como lenguaje de especificación hace uso de símbolos gráficos y "cajas" que habitualmente no se encuentran disponibles en un sistema editor estándar. Debido a esto se han creado algunas herramientas para LaTeX[SpM290] que nos proporcionan la facilidad de poder escribir especificaciones en un formato de texto y poder visualizarlas después de compilarlas en la forma usual de LaTeX [LaL94].

Introduzcamos ahora el lenguaje matemático usado por Z.

1.1 Teoría de conjuntos

En Z se tienen las operaciones usuales entre conjuntos, con algunas restricciones que son dadas mediante su sistema de tipos y que se tratarán más adelante.

Un conjunto puede definirse mediante la siguiente notación:

$$A == Expresi\acute{o}n$$

donde $Expresi\acute{o}n$ puede ser la lista de elementos que define al conjunto (forma extensional), puede ser un predicado que describa las características de los elementos del conjunto (forma intencional) o una expresión que involucre operaciones entre conjuntos.

Los operadores entre conjuntos son los usuales, unión ($\cup$), intersección ($\cap$), etc. En el apéndice A se describe una lista de estos operadores. Además se cuenta con los operadores:

'#' es un operador prefijo que aplicado a un conjunto finito calcula el número de elementos que contiene tal conjunto. Po ejemplo, sea $A == \{1, 2, 3, 4\}$, el número de elementos de A está dado por $\#A = 4$.

'..' describe un subconjunto de números enteros, por ejemplo, $2\,..\,7 = \{2, 3, 4, 5, 6, 7\}$.

1.1.1 Usando predicados para la descripción de conjuntos

Describir conjuntos listando todos sus elementos esta bien para conjuntos cuyo número de elementos es relativamente pequeño, pero resulta difícil, tedioso y a vecees imposible en otro caso. Así podemos auxiliarnos de predicados para la descripción de las características que deben tener los elementos de algún conjunto en particular.

En la teoría de conjuntos, usamos la notación

$$\{(a, b) \mid a \in A \wedge b \in B \wedge a < b\}$$

para describir al conjunto de "todos los pares ordenados (a, b) tal que a pertenece al conjunto A y b al conjunto B y además $a < b$". La sintaxis usada por la notación Z (y que es equivalente a la usual en teoría de conjuntos) es:

$$\{a : A;\ b : B \mid a < b\}$$

En general, la descripción de un conjunto se realiza mediante la sintaxis [PoB+91] [WoJ+97]:

$$\{Declaraci\acute{o}n[\mid Predicado][\bullet\ Expresi\acute{o}n]\}$$

donde:

Declaración introduce una o más variables locales y el dominio de trabajo. Las n-variables declaradas determinan una tupla de n elementos; el orden en que son declaradas las variables es el orden en el que aparecen en la tupla. Es importante decir que la tupla puede ser ignorada para la descripción del conjunto en función de la sintaxis, como se describe más adelante.

Predicado restringe los valores que pueden tomar las variables declaradas. La estructura de un predicado es la siguuiente

$$[Q][Decls][\bullet]\,explog$$

donde :

Q es un cuantificador: exitencial ($\exists$) o universal ($\forall$).

Decls es la lista de variables sobre las que actua el cuantificador.

explog es una restricción sobre las variables declaradas tanto en la parte declarativa del predicado así como sobre las variables declaradas en la construcción del conjunto.

Nótese que el cauntificador, la parte declarativa y el símbolo $\bullet$ son opcionales.

Expresión es una operación sobre las variables declaradas.

De acuerdo a la sintaxis tenemos que un conjunto puede definirse de alguna de las siguientes formas alternativas.

1. $\{Declaraci\acute{o}n \mid Predicado\}$

   El conjunto se conforma de aquellos valores o tuplas de valores de variables introducidas en *Declaración* para los cuales el *Predicado* toma el valor veritativo verdadero. Si se declara más de una variable, el orden en el que aparecen las variables declaradas indicará el orden de la tupla que se forma, esto es, si $Declaraci\acute{o}n$ tiene la forma:

   $$x : X;\ y : Y;\ z : Z$$

deberemos interpretarla como:

$$(x, y, z) \in X \times Y \times Z$$

según la notación de la teoría clásica de conjuntos. De la misma forma si ahora tenemos la declaración $y : Y;\ x : X;\ z : Z$ será equivalente a $(y, x, z) \in Y \times X \times Z$.

Ejemplos.

(a) $\{n : \mathbb{N} \mid n < 4\} = \{0, 1, 2, 3, 4\}$ En este caso, se declara sólo una variable así que el conjunto está formado por aquellos números naturales que son estrictamente menores que cuatro. La forma equivalente en la teoria clásica de conjuntos sería:

$$\{n \in \mathbb{N} \mid n < 4\}$$

(b) $\{m, n : \mathbb{N} \mid m + n = 5\} = \{(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)\}$
El conjunto se conforma por aquellos pares ordenados tales que la suma de sus componentes es igual a 5. El orden en el que aparecen las variables es importante e indica como se forma la tupla. La forma equivalente en la notación de la teoría clásica de conjuntos es:

$$\{(m, n) \in (\mathbb{N} \times \mathbb{N}) \mid m + n = 5\},\ \textit{o}$$
$$\{(m, n) \in \mathbb{N}^2 \mid m + n = 5\}$$

2. $\{\textit{Declaración} \bullet \textit{Expresión}\}$ El conjunto se conforma por aquellos elementos generados a partir de evaluar la expresión *Expresión* en el dominio definido por la *Declaración*. En este caso, la tupla determinada por las variables declaradas no es tomada en cuenta para determinar el conjunto.

Ejemplos.

(a) $\{n : \mathbb{N} \bullet 4 * n\} = \{0, 4, 8, 12, 16, ...\}$

(b) $\{x : X;\ y : Y;\ z : Z \bullet (x, y, z)\}$ En este caso, *Expresión* evalúa a un elemento del producto cartesiano formado por X, Y y Z.

3. $\{Declaración \mid Predicado \bullet Expresión\}$

   El conjunto se conforma por los elementos generados al evaluar la expresión sobre el dominio definido en *Declaración* y que es restringido por aquellos valores que cumplen el predicado.

   Ejemplo:

   $\{n : \mathbb{N} \mid n < 6 \bullet n * n\} = \{0, 1, 4, 9, 16, 25\}$

A primera vista parece complicado saber cuando la declaración de variables determina si el conjunto está conformado por la tupla formada por esa declaración, pero la siguiente regla puede usarse para salvar este problema: **"Si la sintaxis usada para describir un conjunto contiene • *Expresión* entonces el conjunto de variables declaradas sólo servirán para definir el dominio de trabajo de cada variable"**.

1.2 Sistema de tipos

Z propone la idea de que los objetos en el universo puedan ser categorizados en diferentes clases[PoB+91] y que además, estas clases de objetos sean disjuntas. Estas clases de objetos a la que nos referimos son los tipos del lenguaje Z.

El sistema de tipos requiere que cuando una variable sea introducida, sea declarada junto con información sobre su tipo. El tipo determina el conjunto de posibles valores para dicha variable. Cualquier expresión, variable ó dato en Z debe pertenecer a uno y sólo un tipo.

La sintaxis para declarar una variable es la siguiente:

$Variable : Tipo$

donde:

Variable es el nombre de la variable a ser declarada.

Tipo es el tipo de la variable declarada.

Puede declararse una serie de variables del mismo tipo usando la sintaxis:

$V_1, \ldots, V_n : Tipo$

o variables de diferentes tipos, mediante:

$$V_1 : T_1; \ \ldots; \ V_n : T_n$$

El sistema de tipos impone restricciones sobre la forma en la que podemos definir y usar conjuntos. Estas restricciones indican cuando una expresión es correcta. Dada una expresión en Z de la forma:

$$A \ \mathit{Operador} \ B$$

donde:

$$\mathit{Operador} : (T_1 \times T_2) \rightarrow T_3$$

se dice que esta expresión **tiene sentido** si y sólo si

$$\mathit{tipo}(A) = T_1 \ \mathit{y} \ \mathit{tipo}(B) = T_2$$

y además $\mathit{tipo}(A \ \mathit{Operador} \ B) = T_3$.

De otra forma esa expresión carece de sentido y se considera una expresión sintácticamente errónea.

1.2.1 Tipos básicos

La declaración de tipos básicos permite la existencia de entidades cuya característica principal es que su estructura interna es irrelevante, esto es, no es importante como sea su implantación en cuanto se asuma que se tiene un tipo de tal naturaleza, por ejemplo, al definir el tipo NUMERO_TELEFONICO se asumirá que existe, y no tendremos que preocuparnos si este se representa internamente como una secuencia de caracteres numéricos, alfanuméricos o en BCD.

Lo que si se restringe es que los tipos básicos sean disjuntos. No se permite que una variable pertenezca a dos tipos diferentes.

La sintaxis para definir tipos básicos es la siguiente:

$$[TB_1, TB_2, \ldots, TB_n]$$

esta sintaxis hará de los TB_i, $i = 1, \ldots, n$; tipos básicos disponibles para la declaración de variables.

1.2.2 Tipos de datos simples

En muchos casos es necesario definir algún tipo con un número pequeño de elementos o valores posibles para una variable declarada de tal tipo. De esta forma Z proporciona definiciones de tipos de datos simples donde estas declaraciones tienen la siguiente sintaxis:

$$Nombre\_de\_tipo ::= Elm_1 \mid Elm_2 \mid \ldots \mid Elm_n$$

donde $Elm_1, Elm_2, \ldots, Elm_n$ son los posibles valores que proporciona el tipo *Nombre_-de_tipo* y que además son diferentes.

Por ejemplo el siguiente tipo puede usarse como un indicador de éxito o fracaso.

$$Mensaje ::= Exito \mid Fracaso$$

$$reporte : Mensaje$$

$$reporte = Exito$$

1.2.3 Constructores

A partir de los tipos básicos pueden definirse nuevos tipos usando los constructores de tipos. Estos son:

- El constructor **Potencia** ($\mathbb{P}$), que indicará al conjunto potencia.
- El constructor **Producto cartesiano** ($\times$) que es el producto cartesiano de los tipos involucrados.
- Combinaciones de los dos anteriores.

El resultado es un nuevo tipo si y sólo si el constructor se aplica a un tipo.

1.3 Relaciones

Una relación es un conjunto de pares ordenados, puede verse también como un subconjunto del producto cartesiano, de esta forma, una relción R de X en Y, es un elemento de:

$$\mathbb{P}(X \times Y)$$

La notación Z ha empleado "flechas" para denotar relaciones, esto es debido a que podemos ver graficamente a una relación como una flecha que asocia elementos de dos conjuntos. El tipo de cada flecha determina el tipo de la relación, en particular, cuando hablemos de funciones, estaremos definiendo el símbolo (o la flecha) que identifica a cada función.

El símbolo de relación se define de la siguiente forma:

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

donde $X \leftrightarrow Y$ denota el conjunto de todas las relaciones entre X y Y. El símbolo $\leftrightarrow$ no debe confundirse con el símbolo lógico $\Leftrightarrow$.

Definición. Sea R una relación de tipo $A \leftrightarrow B$, entonces llamaremos a A el conjunto fuente y a B el conjunto destino. Un elemento de R se denota como (a, b) donde $a \in A$ y $b \in B$, otra forma equivalente para denotar un elemento de una relación es $a \mapsto b$. El símbolo $\mapsto$ es usado para indicar una relación entre los elementos a y b.

Ejemplo. Sean

$$\begin{aligned}&\mathit{Ciudad} == \{\mathit{Paris}, \mathit{Londres}, \mathit{Madrid}, \mathit{Barcelona}, \mathit{Oxford}\} \ y\\ &\mathit{Pais} == \{\mathit{Inglaterra}, \mathit{Francia}, \mathit{España}\}\end{aligned}$$

El conjunto de todas las relaciones de *Ciudad* a *Pais* es:

$$\begin{aligned}\mathit{Ciudad} \leftrightarrow \mathit{Pais} == \{&\emptyset, \{(\mathit{Paris}, \mathit{Inglaterra})\},\\ &\{(\mathit{Paris}, \mathit{Francia})\}, \{(\mathit{Paris}, \mathit{España})\},\\ &\{(\mathit{Londres}, \mathit{Inglaterra})\}, \ldots, \{(\mathit{Paris}, \mathit{Francia}),\\ &(\mathit{Londres}, \mathit{Inglaterra})\},\\ &\{(\mathit{Madrid}, \mathit{España}), (\mathit{Barcelona}, \mathit{España})\}, \ldots\}\end{aligned}$$

Podemos declarar las relaciones:

$$\mathit{CapitalDe} : \mathit{Ciudad} \leftrightarrow \mathit{Pais}$$

como el conjunto que relaciona a una ciudad capital con su correspondiente país; y

$$\mathit{ReinoUnido} : \mathit{Ciudad} \leftrightarrow \mathit{Pais}$$

el conjunto de pares $(\mathit{ciudad}, \mathit{pais})$ tal que geográficamente pertenecen al Reino Unido.

De las declaraciones anteriores podemos ver que

$$\begin{array}{l} \mathit{CapitalDe} \in \mathit{Ciudad} \leftrightarrow \mathit{Pais} \\ \mathit{ReinoUnido} \in \mathit{Ciudad} \leftrightarrow \mathit{Pais} \end{array}$$

además podemos hacer afirmaciones como:

$$\mathit{Madrid} \mapsto \mathit{España} \in \mathit{CapitalDe}$$

$$\mathit{Oxford} \mapsto \mathit{Inglaterra} \notin \mathit{CapitalDe} \quad \text{pero}$$

$$\mathit{Oxford} \mapsto \mathit{Inglaterra} \in \mathit{ReinoUnido}$$

1.3.1 Operaciones entre relaciones

Puesto que una relación es un conjunto de pares ordenados entonces todas aquellas operaciones entre conjuntos son aplicables a relaciones con las mismas restricciones impuestas por el sistema de tipos.

Además de las operaciones usuales de conjuntos, se definen las siguientes. Sean $R : X \leftrightarrow Y$, $S : \mathbb{P}(X)$ y $T : \mathbb{P}(Y)$.

Dominio

$$\operatorname{dom} R = \{x : X \mid (\exists\, y : Y \bullet x \mapsto y \in R)\}$$

Rango

$$\operatorname{ran} R = \{y : Y \mid (\exists\, x : X \bullet x \mapsto y \in R)\}$$

Relación Inversa

$$R^{-1} = \{x : X;\ y : Y \mid x \mapsto y \in R \bullet y \mapsto x\}$$

Restricción Dominio

$$S \lhd R = \{x : X;\ y : Y \mid x \in S \land x \mapsto y \in R\}$$

Restricción Rango

$$R \rhd T = \{x : X;\ y : Y \mid y \in T \land x \mapsto y \in R\}$$

Substracción Dominio

$$S \lhd R = \{x : X;\ y : Y \mid x \notin S \land x \mapsto y \in R\}$$

Substracción Rango

$$R \rhd T = \{x : X;\ y : Y \mid y \notin T \land x \mapsto y \in R\}$$

Composición de Relaciones

$$R \fatsemi S = \{x : X;\ y : Y;\ z : Z \mid x \mapsto y \in R \land y \mapsto z \ \in S \bullet x \mapsto z\}$$

Imagen Relacional

$$R(\!|\ S\ |\!) = \{x : X;\ y : Y \mid x \in S \land x \mapsto y \in R \bullet y\}$$

1.4 Funciones

Otra herramienta básica de la notación Z son las funciones. Una función es una relación con la propiedad de que cada elemento del dominio debe relacionarse a lo más con un elemento del rango, formalmente esto es

$$(x \mapsto y) \in R \land (x \mapsto z) \in R \Rightarrow y = z$$

1.4.1 Clasificación

En la notación Z se clasifica a las funciones de la siguiente forma:

Parciales Las funciones parciales son la clase más general de funciones, así cualquier función es una función parcial. Son llamadas parciales debido a que el dominio de la función no necesariamente debe ser todo el conjunto fuente.

La definición y notación de una función parcial es

$$\begin{aligned} X \nrightarrow Y == \{&R : X \leftrightarrow Y \mid \\ &(\forall x : X;\ y, z : Y \bullet x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z)\} \end{aligned}$$

Totales Una función total es aquella cuyo dominio es todo el conjunto fuente. Formalmente se define de la siguiente forma

$$X \rightarrow Y == \{f : X \nrightarrow Y \mid domf = X\}$$

Podemos usar la notación fx para denotar que para cualquier x en el dominio de f, fx es el único elemento del rango al cual mapea f. Esta notación es equivalente a la tradicional $f(x)$.[1]

Inyectivas Una función inyectiva tiene la propiedad de que dos elementos del dominio no están mapeados con el mismo elemento del rango, es decir,

$$x \mapsto y \in f \wedge z \mapsto y \in f \Rightarrow x = z$$

Dentro de las funciones inyectivas tendremos las siguientes definiciones:

Inyectivas Parciales

$$X \rightarrowtail\!\!\!\!\!\nrightarrow Y == \{f : X \nrightarrow Y \mid \\ (\forall x_1, x_2 : \mathrm{dom} f \bullet f\ x_1 = f\ x_2 \Rightarrow x_1 = x_2)\}$$

Inyectivas Totales

$$X \rightarrowtail Y == \{f : X \rightarrow Y \mid \\ (\forall x_1, x_2 : \mathrm{dom} f \bullet f\ x_1 = f\ x_2 \Rightarrow x_1 = x_2)\}$$

Suprayectivas Una función suprayectiva es aquella función cuyo rango es todo el conjunto destino, es decir, es una función donde cada elemento del conjunto destino es la imagen de al menos un elemento del dominio de la función.

Suprayectivas Parciales

$$X \nrightarrow\!\!\!\!\!\twoheadrightarrow Y == \{f : X \nrightarrow Y \mid \mathrm{ran} f = Y\}$$

Suprayectivas Totales

$$X \twoheadrightarrow Y == \{f : X \rightarrow Y \mid \mathrm{ran} f = Y\}$$

[1] Cuando la aridad de la función sea mayor de 1, es necesario usar paréntesis para agrupar los argumentos.

Biyectivas Una función biyectiva es aquella que es inyectiva y además suprayectiva. Formalmente tenemos que una función biyectiva se define como:

$$X \rightarrowtail\!\!\!\!\twoheadrightarrow Y == \{f : X \rightarrow Y \mid f \in X \rightarrowtail Y \land f \in X \twoheadrightarrow Y\}$$

Finitas Una función se dice finita si su dominio es un conjunto finito. Podemos dar la notación para estas funciones:

Finitas parciales

$$X \mathrel{+\!\!\!\!\twoheadrightarrow} Y$$

Finitas parciales inyectivas

$$X \mathrel{\rightarrowtail\!\!\!\!\!\!+\!\!\!\!\twoheadrightarrow} Y$$

1.4.2 Operaciones sobre funciones

Puesto que una función es una clase especial de relación, todas las operaciones definidas para relaciones pueden aplicarse a funciones. Sin embargo, debe tenerse cuidado ya que una operación no siempre producirá una función. Por ejemplo, si tomamos la unión de las dos funciones siguientes:

$$f == \{0 \mapsto 1, 1 \mapsto 1\}$$
$$g == \{0 \mapsto 2, 2 \mapsto 1\}$$

$$f \cup g = \{0 \mapsto 1, 1 \mapsto 1, 0 \mapsto 2, 2 \mapsto 1\}$$

no se obtiene como resultado una función.

Existe un operador adicional que se usa específicamente para generar una función con información nueva y garantiza que la relación resultante es una función.

La operación se llama *sobreescritura funcional*. Sean $f, g : X \mathrel{+\!\!\!\!\rightarrow} Y$, entonces la sobreescritura funcional esta dada por:

$$f \oplus g = ((\mathrm{dom}\, g) \lhd f) \cup g$$

Esta operación remueve de f todos los pares cuyo primer elemento esta en el dominio de g y agrega todos los pares de la función g.

Ejemplo. Sean f y g como en el ejemplo anterior entonces

$$f \oplus g = \{1 \mapsto 1, 0 \mapsto 2, 2 \mapsto 1\}$$

ahora la relación resultante es una función.

1.4.3 Notación lambda

Otra forma de escribir funciones es usando la notación lambda λ. Una expresión λ toma la siguiente forma:

$$\lambda\, decls[\mid pred] \bullet term$$

Esta expresión representa a una función que toma como argumentos el resultado de la expresión $decls[\mid pred]$ y lo mapea al valor definido por $term$.

La expresión lambda

$$\lambda\, x : T \mid pred \bullet term$$

es equivalente a la función descrita por la expresión de conjuntos

$$\{x : T \mid pred \bullet x \mapsto term\}$$

La notación lambda es una forma abreviada de describir funciones.

Ejemplo.

$$\lambda\, n : \mathbb{N} \mid n < 5 \bullet n^2$$

es equivalente a :

$$\{n : \mathbb{N} \mid n < 5 \bullet n \mapsto n^2\} = \{(0,0), (1,1), (2,4), \ldots\}$$

1.4.4 Secuencias

Una secuencia es una colección de objetos donde: el orden en el que aparecen los objetos es importante y puede haber repetición de objetos.

Por lo tanto, una secuencia es usada para modelar situaciones donde el orden y el número de repeticiones de un objeto son importantes, por ejemplo al considerar la estructura 'cola'. La notación extensional para una secuencia es:

$$\langle Obj_1, \ldots, Obj_n \rangle$$

donde puede darse que:

$$Obj_i = Obj_k \qquad cuando\ i \neq k$$

La característica a resaltar de una secuencia es la posición donde aparecen sus elementos.

Así, la secuencia:

$$\langle \mathit{rojo}, \mathit{verde}, \mathit{azul} \rangle$$

contiene información sobre la posición de los elementos.

Posición	Elemento
1	rojo
2	verde
3	azul

La información anterior puede escribirse mediante una función cuyo dominio es $\{1, 2, 3\}$

$$\langle \mathit{rojo}, \mathit{verde}, \mathit{azul} \rangle = \{1 \mapsto \mathit{rojo}, 2 \mapsto \mathit{verde},\ 3 \mapsto \mathit{azul}\}$$

Así, una secuencia es vista como una función de algún segmento inicial $1 \mathrel{..} n$ de $\mathbb{N}$ a los elementos de la secuencia.

Formalmente podemos definir, para cualquier conjunto X

$$\operatorname{seq} X == \{f : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!\!\!\;\shortmid\ \ X \mid \operatorname{dom} f = 1 \mathrel{..} \#f\}$$

1.5 La estructura de una especificación

Una especificación en Z es una serie de párrafos donde cada párrafo puede ser cualquiera de los bloques de construcción de especificación. El universo de tipos básicos y los tipos de datos simples son dos de estos bloques. Los restantes bloques se describen a continuación.

1.5.1 Descripción axiomática

Una descripción axiomática introduce una o más variables globales y opcionalmente especifica una restricción sobre sus valores en su parte del predicado.

La forma de una descripción axiomática es:

$$\begin{array}{|l}
\mathit{Declaraciones} \\
\hline
\mathit{Predicado}
\end{array}$$

Ejemplo:

$$\begin{array}{|l} maxlines : \mathbb{Z} \\ \hline maxlines > 0 \end{array}$$

Las variables introducidas por una descripción axiomática no pueden haber sido definidas con anterioridad.

Además, si en alguna parte de la especificación se desea agregar un predicado que describe una restricción a alguna variable global ya definida se puede hacer sólo escribiendo el predicado.

$$\begin{array}{|l} maxlines < 1000 \end{array}$$

1.5.2 Definiciones abreviadas

Las definiciones abreviadas son usadas para definir constantes mediante el uso de alguna construcción definida con anterioridad, así, la definición:

$$Nom == Expr$$

introduce un nuevo nombre para $Expr$, el nuevo nombre Nom es una constante global de la especificación con el mismo tipo y valor que la expresión $Expr$.

A fin de hacer más corta la escritura de una especificación es posible tomar un conjunto S del tipo T y usarlo para declarar variables, digamos x, entonces

$$x : S$$

indica que x es un elemento de S y como S es del tipo T entonces x será del tipo T, además, el valor de x está restringido por el predicado de S.

Ejemplo:

$$Pares == \{x : \mathbb{Z} \mid \exists\, y : \mathbb{Z} \bullet y > 0 \land x = 2 * y\}$$

y si queremos introducir la variable $alg\acute{u}n\_par$ tendremos que declarar

$$alg\acute{u}n\_par : \mathbb{Z}$$

y a continuación escribir un predicado que exprese los requerimientos de $alg\acute{u}n\_par$:

$$\exists\, y : \mathbb{Z} \bullet y > 0 \land alg\acute{u}n\_par = 2 * y$$

entonces una forma abreviada de hacerlo es definir

$$alg\acute{u}n\_par : Pares$$

lo cual describe los dos requerimientos anteriores.

1.5.3 Definiciones genéricas

El fin de las definiciones genéricas es contar con la capacidad de definir construcciones matemáticas donde no importa el tipo de sus componentes. De esta manera cuando se haga uso de ellas en la especificación se debe proveer del tipo en cuestión.

Las definiciones genéricas toman la siguiente forma:

$[X_1, ..., X_k]$
Declaraciones
Predicado

donde:

$X_1, ..., X_k$ son llamados *argumentos o parámetros genéricos* y pueden instanciarse con algún tipo en el transcurso de la especificación.

Declaraciones introduce una o más construcciones genéricas las cuales se consideran globales, cada declaración hace uso de combinaciones de los argumentos genéricos para expresar su tipo temporal. Una lista de declaraciones $D_1, \ldots, D_n$ puede escribirse usando el símbolo ; como separador.

Predicado define las características de las definiciones, esta parte también se llama *"parte axiomática"*. Una serie de predicados $P_1, \ldots, P_n$ se escriben usando el símbolo ; como separador, el significado es la conjunción de los n predicados.

Ejemplo:

$[X]$
$\emptyset : \mathbb{P}\, X$
$\emptyset = \{x : X \mid x \neq x\}$

El reemplazo de los parámetros genéricos por conjuntos actuales es conocido como instanciación.

1.5.4 Esquemas

El concepto de esquema ocupa un papel importante dentro de la notación Z. Un esquema es el medio por el cual todos los conceptos matemáticos hasta ahora definidos se usan para poder realizar la modelación de un sistema a través de una especificación.

Los esquemas son usados para describir aspectos estáticos, como son:

- Los estados que ocupa el sistema, que llamaremos el conjunto de estados admisibles.
- Las relaciones invariantes que se mantienen cuando el sistema transita de un estado a otro.

Los aspectos dinámicos incluyen:

- Las operaciones que son posibles.
- Las relaciones entre sus entradas y salidas.
- Las transiciones de estados que ocurren.

La forma de un esquema[PoB+91] es:

Nombre
Declaraciones

Predicado

donde:

Nombre Es el nombre del esquema.

Declaraciones Introduce variables locales y/o referencias a esquemas previamente definidos. Una lista de declaraciones $D_1, \ldots, D_n$ se escribe usando el símbolo ; como separador.

Predicado Especifica restricciones sobre las variables introducidas. Estas restricciones son llamadas *"propiedades del esquema"*. De igual forma, una serie de predicados $P_1, \ldots, P_n$ o $P_1 \wedge \ldots \wedge P_n$ puede escribirse usando el símbolo ; como separador. Si la parte del predicado se omite se asume que tiene un valor de verdadero, es decir, las variables declaradas pueden tomar los valores permitidos por sus tipos.

Un esquema puede definirse de forma horizontal

$$Nombre \mathrel{\widehat=} [Declaraciones[|\ Predicado]]$$

y el significado es el mismo que en la forma vertical (anterior) o

$$Nombre \mathrel{\widehat=} Expresi\acute{o}n\_de\_Esquema$$

donde esta última se usa para combinar esquemas ya definidos mediante el uso de los operadores del cálculo de esquemas.

Cuando intentamos escribir una especificación estamos intentando construir un modelo matemático de las propiedades que describen al sistema y en particular a objetos, que consideraremos cambiantes con el paso del tiempo; tales propiedades son las que nos sirven para describir los estados admisibles del sistema[2]. De esta forma el conjunto , de estados admisibles queda descrito por la sintaxis de esquema donde:

Nombre es el nombre del esquema que describe los estados posibles del sistema

Declaraciones representa el conjunto de propiedades del sistema. Las variables declaradas se llaman *"variables de estado"*.

Predicado Expresa las relaciones existentes entre las variables de estado definidas. También se llama *"estado invariante"*. Cuando hagamos referencia a la *"parte axiomática"* estaremos hablando del predicado del bloque.[3]

Un cambio de estado está dado por el cambio en los valores de las variables de estado. Por otra parte, cuando se define una operación, debe declararse cualquier entrada sufijándola con ?, las salidas se sufijan con !, los componentes posteriores a la operación se priman así como la relación o relaciones existentes entre ellos.

El cálculo de esquemas

El concepto de esquema[PoB+91] [WoJ+97] ocupa un lugar central en la notación Z, siendo éste el bloque de construcción de especificaciones más importante, de aquí que se han definido operadores que se aplican a esquemas, con lo que se puede expresar un esquema como la combinación de otros esquemas definidos previamente. En consecuencia, la información de una especificación se puede dividir en pequeñas piezas o bloques de especificación, donde idealmente cada pieza describe alguna característica

[2] También son llamados estados abstractos.

[3] Esto también se aplica a las descripciones axiomáticas y definiciones genéricas.

del sistema con cierta independencia de otros aspectos o de otros bloques de especificación, pero que en algún nivel todos los bloques se encuentran relacionados entre sí, en otras palabras se da lugar a la **modularidad**.

A diferencia de las operaciones entre conjuntos, en el cálculo de esquemas se permite la combinación de esquemas cuya parte declarativa contenga diferentes declaraciones, teniendo cuidado que después de la combinación no existan variables con el mismo nombre y tipo diferente. Si se diera el caso, entonces antes de realizar la operación es necesario renombrar alguna de las variables que están en conflicto.

Ejemplo. Sean los esquemas A y B definidos como:

A
$c? : T$
$\vdots$

B
$c! : S$
$\vdots$

Entonces si intentamos combinar A y B tendremos un conflicto en la declaración de la variable c, lo cual puede arreglarse renombrando c en A o en B.

Veamos entonces las operaciones más representativas del cálculo de esquemas.

Inclusión de esquemas

Una de las operaciones más simples es la de *incluir* el nombre de un esquema dentro de la declaración de otro:

S_1
$Decls_1$
$Pred_1$

S_2
S_1
$Decls_2$
$Pred_2$

El efecto de la inclusión del esquema S_1 entre las declaraciones del esquema S_2 es que las declaraciones de S_1 son incluidas entre las de S_2 y el predicado de S_1 es apendizado al de S_2 formando una o más conjunciones adicionales.

$$\begin{array}{|l} \multicolumn{1}{l}{S_2} \\ \hline Decls_1 \\ Decls_2 \\ \hline Pred_1 \\ Pred_2 \\ \hline \end{array}$$

Decoración

Como se mencionó, los caracteres ?, ! y $'$ pueden agregarse a los nombres de variables para denotar variables de entrada, variables de salida y estados posteriores a una operación respectivamente, a este proceso se le conoce como **decoración**.

Una decoración puede ser aplicada a un esquema, en particular la decoración $'$ es la que se aplica a esquemas. El efecto de aplicar una decoración a un esquema es el de decorar todas las variables en su declaración y predicado.

Ejemplo. Sea el esquema

$$\begin{array}{|l} \multicolumn{1}{l}{S} \\ \hline a : T \\ b : R \\ \hline a = expr_1 \\ b = expr_2 \\ \hline \end{array}$$

apliquémosle la decoración S'.

El efecto de la decoración es:

$$\begin{array}{|l} \multicolumn{1}{l}{S'} \\ \hline a' : T \\ b' : R \\ \hline a' = expr_1' \\ b' = expr_2' \\ \hline \end{array}$$

El símbolo Δ como prefijo del nombre de un esquema es la decoración usada cuando se requiera sugerir un cambio de estado. Sea S un esquema, ΔS es:

ΔS

S

S'

Si alguna operación en un sistema no causa un cambio de estado entonces el prefijo usado es Ξ. Sea S un esquema, ΞS es:

ΞS

ΔS

$No\_hay\_cambio\_de\_estado$

donde el predicado apropiado en lugar de $No\_hay\_cambio\_de\_estado$ establece que las variables primadas y no primadas son iguales.

Lo importante a destacar es que las decoraciones Δ y Ξ son sólo una convención y no forman parte de la notación Z.

Disyunción-conjunción

Se pueden combinar esquemas usando el operador de esquemas $\vee$ ('or'). En general, el significado de $S \mathrel{\widehat{=}} S_1 \vee S_2 \vee \ldots \vee S_n$ es: las declaraciones de S están formadas por la unión de las declaraciones de $S_1, S_2, \ldots, S_n$ y el predicado de S se forma tomando los predicados de $S_1, S_2, \ldots, S_n$ colocando entre ellos el operador lógico $\vee$. Además las ocurrencias del operador $\wedge$ ('and') que estaban implícitas en los predicados de $S_1, S_2, \ldots, S_n$ deben hacerse explícitas.

Dado que el operador lógico $\vee$ es asociativo y conmutativo, entonces el $\vee$ para esquemas tendrá estas propiedades también.

Cuando realizamos la conjunción de esquemas, usando el operador $\wedge$ ('and'), $S \mathrel{\widehat{=}} S_1 \wedge S_2 \wedge \ldots \wedge S_n$ las declaraciones de S están formadas por la unión de las declaraciones de $S_1, S_2, \ldots, S_n$ y el predicado de S es la conjunción de los predicados de $S_1, S_2, \ldots, S_n$.

Ocultamiento

Cuando el operador \ es aplicado a un esquema el argumento del lado derecho del operador es una tupla de variables las cuales se desea "ocultar" de la parte declarativa del esquema.

Formalmente,

$$[Decl\_Part \mid Pred] \setminus (v_1, \ldots, v_n)$$

donde $v_i \in Decl\_Part, i = 1, \ldots, n$

es equivalente a:

$$[RedDecl \mid \exists\, v_1 : T_1;\ \ldots;\ v_n : T_n \bullet Pred]$$

donde:

Decl_Part Es el conjunto de declaraciones del esquema al cual se le aplica el operador de ocultamiento.

Pred Es el predicado del esquema en cuestión.

$v_1, \ldots, v_n$ Son las variables que se desean ocultar, estas variables deben estar definidas en la parte declarativa del esquema, de otra forma, la operación de ocultamiento está *indefinida*. El símbolo ; se usa para separar declaraciones de variables de tipos diferentes.

RedDecl Es el conjunto de declaraciones del esquema resultante y son las declaraciones en *Decl_Part* menos las correspondientes a las v_i.

$T_1, \ldots, T_n$ Son los tipos de las v_i declarados en *Decl_Part*.

Ejemplo. Sea el esquema S definido como sigue:

S
$a : \mathbb{Z}$
$b : \mathbb{N}$
$a = b + 10$

después de aplicar la operación

$$Hide \mathrel{\widehat=} S \setminus (a)$$

el esquema *Hide* tiene la forma

$$\begin{array}{|l}
\textit{Hide} \\
\hline
b : \mathbb{N} \\
\hline
\exists\, a : \mathbb{Z} \bullet a = b + 10 \\
\hline
\end{array}$$

Composición

Si los esquemas S y T describen dos operaciones y cada una de estas operaciones provoca un cambio de estado en el sistema entonces

$$S \fatsemi T$$

describe el cambio de estado que resulta cuando la operación S es seguida de la operación T. Supongamos que el esquema *Estado* describe el espacio de estados del sistema entonces

$$\textit{Estado} \xrightarrow{S} \textit{Estado}^1 \xrightarrow{T} \textit{Estado}^2$$

es la secuencia de estados por la que nos llevará $S \fatsemi T$.

Para que una composición esté definida es necesario que el conjunto de variables primadas declaradas en S correspondan a las mismas variables no primadas declaradas en T.

Formalmente, la composición puede calcularse de la forma:

$$S \fatsemi T = \exists\, \textit{Estado}'' \bullet S[{}''/{}'] \wedge T[{}''/\ ]$$

donde:

- *Estado* representa el conjunto de estados admisibles del sistema.
- $S[{}''/{}']$ indica la substitución de todas las referencias de variables primadas en S por referencias doblemente primadas.
- $T[{}''/\ ]$ indica la substitución de las referencias no primadas en T por referencias doblemente primadas.
- La parte declarativa del nuevo esquema está formada por la unión de las declaraciones de S y T.
- El predicado del nuevo esquema está formado por la conjunción de los predicados de S y T y se verá afectado por la cuantificación existencial de las variables que resulten doblemente primadas.

Ejemplo. Supóngase que se tienen los esquemas

$$
\begin{array}{|l}
\multicolumn{1}{l}{S} \\
\hline
a, a' : A \\
b, b' : B \\
\hline
P \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{T} \\
\hline
a, a' : A \\
b, b' : B \\
c : C \\
\hline
Q \\
\hline
\end{array}
$$

calculemos la composición $Comp \mathrel{\widehat{=}} S \fatsemi T$.

Puesto que las declaraciones de las referencias primadas y no primadas en ambos esquemas son las mismas, entonces la composición está definida. Así el esquema composición es el siguiente:

$$
\begin{array}{|l}
\multicolumn{1}{l}{Comp} \\
\hline
a, a' : A \\
b, b' : B \\
c : C \\
\hline
\exists\, a'' : A, b'' : B \bullet \\
\qquad\qquad P[a''/a', b''/b'] \land \\
\qquad\qquad Q[a''/a, b''/b] \\
\hline
\end{array}
$$

Precondiciones

Una parte fundamental en el diseño de una especificación es el cálculo de precondiciones. Es importante destacar que en el diseño de las operaciones de un sistema generalmente sólo se toman en cuenta las condiciones necesarias para que estas operaciones se lleven a cabo. En otras palabras sólo se consideran las condiciones normales de funcionamiento.

Desafortunadamente, cuando programamos un sistema se definen este tipo de operaciones que llamaremos parciales y no hay una preocupación por analizar aquellas condiciones en las cuales la operación no pueda llevarse a cabo, es decir, nos quedamos a la mitad del camino en la construcción de un sistema robusto. Esto nos lleva a que después de un tiempo de que el sistema se ha puesto en marcha falle inexplicablemente, debido a que se presentó alguna condición que no se tomó en cuenta al momento de su diseño.

El cálculo de precondiciones de una operación es el medio usado para calcular las condiciones precisas bajo las cuales una operación es **aplicable**.

El cálculo de precondiciones nos puede llevar en casos extremos; a saber, que una operación no es aplicable bajo cualquier circunstancia o por el contrario, que siempre es aplicable; en el caso de no serlo, esto nos conduce a la revisión y correción del diseño.

Si el cálculo de las precondiciones nos entrega las condiciones bajo las cuales una operación es aplicable, entonces de estas precondiciones podremos obtener aquellas condiciones bajo las cuales una operación no es aplicable y esto nos servirá para la definición de operaciones de manejo de errores.

Conceptualmente, el operador encargado de realizar este cálculo es pre. Este operador nos devolverá otro esquema que representa las precondiciones del esquema al que le fue aplicado. Más adelante se describe la forma en que son calculadas tales precondiciones.

1.6 Ejemplo de especificación usando la notación Z

El siguiente ejemplo muestra de forma breve como se usan los conceptos antes descritos con el propósito de escribir una especificación.

1.6.1 Especificación informal

Se requiere un sistema que registre la fecha de cumpleaños de un grupo de personas. El sistema debe proporcionar las siguientes operaciones:

1. Registrar una nueva fecha.
2. Buscar una fecha dado el nombre de una persona.
3. Buscar los nombres de las personas que cumplen años en determinada fecha.

1.6.2 Definición de tipos

Necesitamos los tipos que representen el nombre de una persona, así como una fecha. Entonces declaramos los tipos básicos

$$[NOMBRE, FECHA]$$

1.6.3 Descripción de estados abstractos

Requerimos un medio donde registrar las fechas y nombres; cada vez que un nuevo par $(nombre, fecha)$ es agregado, el sistema cambiará de estado. Entonces el espacio de estados está definido como:

$$
\begin{array}{|l}
\hline
\textit{LibroDeCumpleaños} \\
Conocidos : \mathbb{P}\, NOMBRE \\
Cumpleaños : NOMBRE \nrightarrow FECHA \\
\hline
Conocidos = \operatorname{dom} Cumpleaños \\
\hline
\end{array}
$$

donde:

Conocidos el conjunto de personas registradas.

Cumpleaños asocia a cada nombre una sóla fecha, pero puede permitirse que dos nombres compartan la misma fecha.

1.6.4 Estado inicial

Es el estado con el cual inicia el sistema. Cuando el sistema inicia por primera vez no hay ninguna persona registrada. Así el estado inicial está dado por el siguiente esquema.

$$
\begin{array}{|l}
\hline
\textit{LibroInicial} \\
LibroCumpleaños' \\
\hline
Conocidos' = \emptyset \\
\hline
\end{array}
$$

1.6.5 Operaciones parciales

Una primera operación es la de agregar una nueva persona al libro

$$
\begin{array}{|l}
\hline
\textit{AgregaCumpleaños} \\
\Delta \textit{LibroCumpleaños} \\
\textit{nombre}? : \textit{NOMBRE} \\
\textit{fecha}? : \textit{FECHA} \\
\hline
\textit{nombre}? \notin \textit{Conocidos} \\
\textit{Cumpleaños}' = \textit{Cumpleaños} \cup \{\textit{nombre}? \mapsto \textit{fecha}?\} \\
\hline
\end{array}
$$

El predicado de la operación *AgregaCumpleaños* debe asegurar que:

$$\textit{Conocidos}' = \textit{Conocidos} \cup \{\textit{nombre}?\}$$

lo cual probamos a continuación:

$$
\begin{aligned}
\textit{Conocidos}' &= \operatorname{dom} \textit{Cumpleaños}' \\
&= \operatorname{dom}(\textit{Cumpleaños} \cup \{\textit{nombre}? \mapsto \textit{fecha}?\}) \\
&= \operatorname{dom} \textit{Cumpleaños} \cup \operatorname{dom}\{\textit{nombre}? \mapsto \textit{fecha}?\} \\
&= \operatorname{dom} \textit{Cumpleaños} \cup \{\textit{nombre}?\} \\
&= \textit{Conocidos} \cup \{\textit{nombre}?\}
\end{aligned}
$$

Pruebas como la anterior son las que nos permiten aseguran que la especificación corresponde con lo que se requiere.

La siguiente operación a desarrollar es: dado un nombre encontrar la fecha correspondiente.

$$
\begin{array}{|l}
\hline
\textit{EncuentraCumpleaños} \\
\Xi \textit{LibroCumpleaños} \\
\textit{nombre}? : \textit{NOMBRE} \\
\textit{fecha}! : \textit{FECHA} \\
\hline
\textit{nombre}? \in \textit{Conocidos} \\
\textit{fecha}! = \textit{Cumpleaños}\ \textit{nombre}? \\
\hline
\end{array}
$$

La última operación que realiza el sistema es: Dada una fecha queremos los nombres de las personas cuyo cumpleaños corresponde a la fecha dada.

$$
\begin{array}{|l}
\hline
\textit{QuienesCumplenAños} \\
\Xi \textit{LibroCumpleaños} \\
\textit{fecha?} : \textit{FECHA} \\
\textit{nombres!} : \mathbb{P}\, \textit{NOMBRE} \\
\hline
\textit{nombres!} = \{n : \textit{Conocidos} \mid \textit{Cumpleaños}\ n = \textit{fecha?}\} \\
\hline
\end{array}
$$

1.6.6 Operaciones totales

Es tiempo de definir algunas operaciones que nos ayuden a manejar errores. Definamos los mensajes de error y éxito

$$\textit{MENSAJE} ::= \textit{OK} \mid \textit{REGISTRADO} \mid \textit{NO\_REGISTRADO}$$

La siguiente operación nos regresa un mensaje de éxito.

$$
\begin{array}{|l}
\hline
\textit{Exito} \\
\textit{resultado!} : \textit{MENSAJE} \\
\hline
\textit{resultado!} = \textit{OK} \\
\hline
\end{array}
$$

Las operaciones que revisan si una persona está registrada se presentan a continuación.

$$
\begin{array}{|l}
\hline
\textit{PRegistrada} \\
\Xi \textit{LibroCumpleaños} \\
\textit{nombre?} : \textit{NOMBRE} \\
\textit{resultado!} : \textit{MENSAJE} \\
\hline
\textit{nombre?} \in \textit{Conocidos} \\
\textit{resultado!} = \textit{REGISTRADO} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{PNRegistrada} \\
\Xi \textit{LibroCumpleaños} \\
\textit{nombre?} : \textit{NOMBRE} \\
\textit{resultado!} : \textit{MENSAJE} \\
\hline
\textit{nombre?} \notin \textit{NOMBRE} \\
\textit{resultado!} = \textit{NO\_REGISTRADO} \\
\hline
\end{array}
$$

Ahora ya estamos listos para definir las operaciones que manejan errores. Para agregar un nuevo par $(nombre, cumpleaños)$

$$
\begin{aligned}
TAgregaCumpleaños \mathrel{\widehat=} & (AgregaCumpleaños \wedge Exito) \vee \\
& PRegistrada
\end{aligned}
$$

Como ejemplo expandiremos $TAgregaCumpleaños$.

$$
\begin{array}{|l}
\hline
\textit{TAgregaCumpleaños} \\
\Delta LibroCumpleaños \\
nombre? : NOMBRE \\
fecha? : FECHA \\
resultado! : MENSAJE \\
\hline
(nombre? \notin Conocidos \wedge \\
\quad Cumpleaños' = Cumpleaños \cup \{nombre? \mapsto fecha?\} \wedge \\
\quad resultado! = OK) \vee \\
(nombre? \in Conocidos \wedge \\
\quad Cumpleaños' = Cumpleaños \wedge \\
\quad resultado! = REGISTRADO) \\
\hline
\end{array}
$$

Para encontrar un cumpleaños

$$
\begin{aligned}
TEncuentraCumpleaños \mathrel{\widehat=} & (EncuentraCumpleaños \wedge Exito) \vee \\
& PNRegistrada
\end{aligned}
$$

Para encontrar los nombres de las personas que cumplen años en una fecha dada

$$TQuienesCumplenAños \mathrel{\widehat=} QuienesCumplenAños \wedge Exito$$

Así el sistema queda completamente descrito por la siguiente expresión.

$$
\begin{aligned}
TLibro \mathrel{\widehat=} & TAgregaCumpleaños \vee TEncuentraCumpleaños \vee \\
& TQuienesCumplenAños
\end{aligned}
$$

Finalmente, es importante mencionar que el proceso descrito anteriormente forma una parte del proceso de construcción de un sistema en términos de un programa ejecutable. Después del proceso de especificación se sigue otro proceso conocido como **refinamiento**, el cual nos proporciona un conjunto de teoremas con los que podemos ir paso a paso trasladando del nivel de abstracción de nuestra especificación hacia un nivel de concretitud como lo es un programa ejecutable. Estos teoremas del refinamiento también nos ayudarán a garantizar que las condiciones requeridas por el sistema se cumplan cuando pasemos de un nivel a otro, de tal forma que se pueda estar seguro que el programa final cumple con lo establecido en la especificación original.

Aunque la notación Z no describe una serie de pasos para escribir una especificación, los pasos seguidos en la descipción del ejemplo anterior mustra la forma en la que podría escribirse una espeficicación con el objeto de conseguir un documento que pueda leerse y comprenderse com mayor facilidad.

Capítulo 2

Análisis Léxico y Sintáctico

2.1 Introducción

La función principal de un analizador léxico es la de leer una secuencia de caracteres y transformarla en entidades léxicas que serán usadas por el analizador sintáctico. Estas entidades léxicas pueden llevar información adicional que especifica su categoría, (número, identificador, palabra reservada, etc.) La terminología en inglés para una entidad léxica es la de **'token'** y debido a la gran difusión de este término, en lo subsecuente, haremos uso de él.

La tarea de transformar secuencias de caracteres a tokens está en función de un conjunto de reglas que establece el lenguaje que se pretende analizar, que en nuestro caso es el lenguaje Z.

Generalmente, el analizador léxico se implanta como una rutina que es llamada por el analizador sintáctico cada vez que éste necesita que se le provea del siguiente token.

El análisis sintáctico es el proceso por medio del cual se puede determinar si una cadena de tokens pertenece o no al lenguaje generado por una gramática determinada. Para tal fin, un analizador sintáctico puede construir un árbol de análisis sintáctico para la entrada y de esta forma podrá garantizar que tal cadena pertenece al lenguaje [AhA+90].

La mayoría de los analizadores sintácticos caen dentro de dos clases de métodos: *ascendentes* y *descendentes*[WaD80] [CoJ+87]. Estos términos se refieren básicamente a la forma en que se construye el árbol de análisis sintáctico. En los métodos ascendentes el árbol se empieza a construir a partir de las hojas y avanza hacia la raíz. En los métodos descendentes, por el contrario, el árbol se empieza a construir por la raíz y avanza hacia las hojas. En este trabajo se diseña un analizador sintáctico descendente recursivo.

2.2 Análisis sintáctico descendente recursivo

En el análisis sintáctico descendente recursivo se ejecutan un conjunto de procedimientos recursivos para poder procesar la entrada[AhA+90].

Esta técnica[WaD80]tiene las siguientes características:

1. A cada símbolo no terminal de la gramática le corresponde un procedimiento.
2. La secuencia de procedimientos llamados al momento de procesar la entrada define implícitamente un árbol de análisis sintáctico.
3. La codificación de reglas gramaticales a procedimientos recursivos es casi inmediata.

Por otra parte, para poder aplicar este método de análisis sintáctico la gramática a usar debe cumplir con el requisito[AhA+90] [WaD80] de no ser recursiva izquierda, es decir, la gramática no debe contener producciones del estilo

$$A \rightarrow A\alpha$$

donde A es un no terminal y α es una cadena formada por símbolos terminales y no terminales.

Por tal motivo se decide implantar y transformar la gramática de tal forma que el método de análisis sintáctico pueda ser aplicado.

2.2.1 Eliminación de recursividad izquierda

Una gramática es **recursiva izquierda** si existe un no terminal A tal que existe una derivación[1]

$$A \Rightarrow^{+} A\alpha$$

para alguna cadena α.

La recursión directa por la izquierda se puede eliminar mediante el siguiente procedimiento [AhA+90]:

[1] El símbolo $\Rightarrow^{+}$ significa "deriva a"

1. Se agrupan las producciones de A en la forma

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

donde ninguna β_i comienza con A.

2. Finalmente se sustituyen las producciones de A por

$$A \to \beta_1 A' \mid \ldots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \in$$

Existe un algoritmo tratado en el libro de Aho [AhA+90] el cual sistemáticamente elimina la recursión por la izquierda en una gramática y que consiste en aplicar de forma iterativa el procedimiento descrito antes sobre cada producción en la gramática, el inconveniente de tal algoritmo es que la gramática no debe tener producciones $\in$, es decir, de la forma $A \to \in$, ni tampoco debe tener ciclos: producciones de la forma $A \to A$. En nuestro caso resultó suficiente el aplicar el método que se ha descrito.

2.2.2 Factorización por la izquierda

La factorización por la izquierda es una transformación gramatical usada básicamente para retrasar la decisión sobre que regla gramatical usar, cuando no está claro cual de dos producciones de un no terminal A aplicar debido a que las dos comienzan con el mismo terminal.

El procedimiento para factorización que fue aplicado a nuestra gramática es el siguiente.

Para cada no terminal A, encuéntrese el prefijo α más largo común a dos o más alternativas. Si $\alpha \neq \in$, es decir, existe un prefijo común no trivial, sustitúyanse todas las producciones de A,

$$A \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

donde γ representa todas las alternativas que no comienzan con α por

$$A \to \alpha A' \mid \gamma$$

$$A' \to \beta_1 \mid \ldots \mid \beta_n$$

2.3 Implantación

En el desarrollo del prototipo se ha seleccionado como lenguaje de programación a Prolog [WiJ][ReS+90]. Una de las ventajas del uso de este lenguaje es que su forma clausular hace inmediata la transición de reglas gramaticales a predicados, además el trabajo de selección de la regla gramatical que se aplicará, dada una cadena de entrada, es realizado por el mecanismo de "backtracking".

Por ejemplo, la producción

$$A \rightarrow terminal\ B$$

en Prolog se representa como:

$$a : -match(terminal), b.$$

donde:

a es el nombre del predicado asociado con el no terminal A

:- es el símbolo equivalente a $\rightarrow$

match es el procedimiento que verifica el terminal *terminal*

b es el nombre predicado asociado con B.

Cabe señalar que dentro de la sintaxis de Prolog[WiJ], los identificadores que inician con mayúsculas se consideran variables y cuando inician con _ se trata de la variable anónima.

El primer paso hacia la implantación fue transformar la gramática para que pudiera aplicarse el método de análisis sintáctico ya descrito.

Para poder realizar algunas tareas especiales aprovechando la fase de análisis sintáctico, se sigue el esquema de agregar a las producciones de la gramática acciones que desarrollen estas tareas[CoJ+87]. Dentro de las acciones que se realizan se encuentran:

1. Construcción del árbol de sintaxis abstracta para una expresión aritmética o lógica.

2. Llamado a la rutina que infiere el tipo de una expresión. Esto con el objetivo de conocer el tipo de variables locales o globales que son declaradas.

3. Llamado a la rutina que verifica que una expresión tenga sentido.

4. Agregar y extraer información de la tabla de símbolos.

5. El manejo de errores léxicos, sintácticos y semánticos, que involucra el desplegado de avisos del primer error encontrado.

6. Llamado a la rutina que calcula las precondiciones, en el caso de que así se requiera.

2.3.1 Analizador léxico

Puesto que la tarea del analizador léxico es la de entregar tokens y alguna información adicional sobre el token a analizar, un token es representado[2] por:

$t\ ([Símbolo, Atributo])$

donde:

t es el nombre de la cláusula que representa un símbolo terminal, esta cláusula está escrita al estilo de la sintaxis de Edinburgo[WiJ].

Símbolo es el token leído y el cual puede ser uno de los siguientes:

Word Un identificador es aquella secuencia de caracteres alfanuméricos cuyo primer caracter necesariamente es alfabético.

Número Es la secuencia de caracteres numéricos.

Símbolo Gráfico Es la secuencia de uno o más símbolos gráficos, esto es, aquellos caracteres que no pertenecen a los alfanuméricos.

Comando LaTeX Es la secuencia de caracteres que definen algún comando LaTeX.

Atributo información adicional que acompaña al token que se ha leído:

ctex si el token leído es un comando de LaTeX.

simb si el token pertenece al conjunto de caracteres gráficos.

Nombre es el nombre de la palabra que se ha leído.

[2]De aquí en adelante cuando se hable de representación asumiremos que es la implantación en Prolog de una entidad conceptual, salvo en aquellos casos en los que se indique explícitamente.

Las reglas[3] para la construcción de los diferentes tipos de tokens son:

Identificador Un identificador está formado por una secuencia de caracteres alfanuméricos donde el primer caracter es un caracter alfabético. En principio no hay limite para la longitud de un identificador. Además puede usarse el símbolo _ como parte de un identificador. Se hace diferencia entre mayúsculas y minúsculas, los identificadores Sum, sum, y sUm se consideran como tres identificadores distintos.

$$\begin{array}{rcl} Ident & \rightarrow & letra\ [letra \mid digito \mid \_]^* decoracion \\ letra & \rightarrow & a \mid \ldots \mid z \mid A \mid \ldots \mid Z \\ digito & \rightarrow & 0 \mid \ldots \mid 9 \end{array}$$

Decoraciones De acuerdo a la sintaxis de Z, el nombre de un identificador se puede sufijar con los símbolos $'$, !, ?. De esta forma estos símbolos forman parte del nombre de un identificador.

$$decoracion \rightarrow \ ' \mid ? \mid !$$

Número Esta formado por una secuencia de caracteres numéricos además de un punto en caso de ser necesario.

$$num \rightarrow digito\ digito^*[.digito^*]$$

Palabras Reservadas Se consideran palabras reservadas aquellas secuencias de comandos al estilo de LaTeX, donde el primer caracter de dichos comandos es \. En el apéndice A se da una lista detallada de los comandos LaTeX usados para la escritura de especificaciones. En el caso de encontrarse un token que sea el nombre de un comando de este tipo, pero su primer caracter no es \ entonces se reporta que se usó una palabra reservada pero que ésta no inició con \.

La implantación del analizador léxico se encuentra en el predicado

$$nextntokens(LTok, N)$$

donde:

nextntokens Es el nombre del predicado que corresponde a la implantación del analizador léxico.

[3]El significado de $prod^*$ en una regla gramatical es el de repeticiones de esa regla 0 o más veces.

LTok Es la lista de tokens que ha colectado el analizador.

N Es un número que usa internamente el analizador y que le sirve para llevar un conteo de los tokens analizados, al inicio debe ser 0.

El funcionamiento del analizador léxico está determinado por el siguiente caracter no blanco en la entrada.

% Indica el inicio de un comentario, se saltan los caracteres en la entrada hasta el fin de línea.

**** Se ha identificado que existe en la entrada un comando LaTeX, así se procede a leer la siguiente cadena de caracteres S. A continuación se describen las operaciones que son llevadas a cabo por el analizador en función de la cadena S leída:

Blanco mensaje de error

begin,, end Se lee la siguiente secuencia de tokens: símbolo {, símbolo terminal T, símbolo }. El token devuelto es la concatenación de la cadena begin o end y el síimbolo terminal T.

otra cadena en este caso la cadena leída es el token devuelto.

Una vez que se ha leído el token en la entrada, la información adicional devuelta es un atributo que indica la lectura de un comando LaTeX.

No LaTeX Si el símbolo leído no fue \ entonces en función de este caracter se realizan las siguientes operaciones.

Letra se realiza la lectura de una secuencia de caracteres alfanuméricos que corresponden a un identificador. Además, se informa que se ha leído un identificador.

número se realiza la lectura de una secuencia de caracteres numéricos y la información adicional es el valor numérico leído.

gráfico se lee una secuencia de caracteres gráficos hasta encontrar un blanco, esta secuencia de caracteres gráficos se considera como un token por lo que se indica que se leyó un símbolo especial.

2.3.2 Analizador sintáctico

La implantación del analizador sintáctico inicia con la descripción de las cláusulas que lo conformarán. Como se mencionó en la sección 2.2, debemos usar un procedimiento por cada regla gramatical que describe el lenguaje[PoB+91].

Las producciones tienen la siguiente representación:

$$
\begin{aligned}
&simb\_nter(L, M, Opcionales) :- \\
&\qquad c1(L, L_1, Opc1), \\
&\qquad \vdots \\
&\qquad cn(L_{n-1}, M, Opcn).
\end{aligned}
$$

donde:

simb_nter Representa el símbolo no terminal del lado izquierdo del símbolo $\rightarrow$ en una producción.

L La lista de tokens donde el primer elemento en la lista es el primer token a analizar[WaD80].

M La lista de tokens resultante después del análisis.

Opcionales Es una secuencia de argumentos que tienen un significado específico. Por ejemplo, para la producción que describe una palabra

$$ident(L, M, Word)$$

Opcionales en este caso es *Word* y sirve para reconocer una palabra y además para saber cuál fue la palabra reconocida.

c_i el conjunto de cláusulas que componen la regla gramatical, algunas de ellas pueden realizar tareas adicionales como las de agregar un símbolo a la tabla de símbolos. Los argumentos de cada cláusula coinciden con los descritos antes. En caso de que en la regla gramatical exista la ocurrencia de un símbolo terminal, entonces la cláusula equivalente es:

$$match(t(T), List, R).$$

donde:

List Lista de tokens a ser analizados.

R Lista de tokens después de realizar el análisis.

t(T) es el terminal que aparece en la regla gramatical, T tiene la forma descrita en la sección 2.3.1.

match es el predicado encargado de verificar que el siguiente token en la entrada (el elemento en la cabeza de *List*) corresponda con el símbolo terminal ($t(T)$) definido por la regla gramatical que es aplicada por el analizador sintáctico.

Además, *match* tiene la tarea de hacer un llamado al analizador léxico para que proporcione los siguientes tokens cuando la lista a procesar se encuentre con un sólo elemento.

Ejemplo. La regla gramatical que reconoce a un esquema es:

Schema_Box → `\begin{schema}{Nombre}`
Decl_Part
`\where`
Axiom_Part
`\end{schema}`

La figura 2.1 muestra la representación de la regla gramatical presentada. La descripción de la implantación correspondiente a la regla gramatical es la siguiente.

1. La cabeza del predicado es la representación del no terminal *Schema_Box*.

2. Los símbolos $:-$ equivalen al símbolo $\rightarrow$ en la producción.

3. Las dos cláusulas siguientes son la representación para la secuencia de símbolos terminales `\begin{schema}{`

4. **ident** representa a *Nombre*, en el caso de esta cláusula la variable *NSchema* pertenece al grupo de argumentos opcionales, su utilidad es la de obtener el nombre del esquema que se analiza.

5. **scheboxp** se encarga de revisar primero al terminal `\where` y después `\end \{schema}`.

6. Las demás cláusulas pertenecen al grupo de acciones adicionales que desarrollan tareas específicas, entre ellas, la comprobación de tipos.

$$
\begin{array}{l}
schemabox(L, M) :- \\
\qquad match(t([beginschema, ctex]), L, L1), \\
\qquad match(t(['\{', simb]), L1, L2), \\
\qquad ident(L2, L3, NSchema), \\
\qquad match(t(['\}', simb]), L3, L4), \\
\qquad declpart(L4, L5, Decls), \\
\qquad scheboxp(L5, M, Tree), \\
\qquad (\backslash + \ Tree == null \\
\qquad -> (expand\_schema(Decls, DecEx), \\
\qquad \quad local(DecEx), \\
\qquad \quad block\_type\_check(Tree), \\
\qquad \quad unlocal \\
\qquad \quad) \\
\qquad ;\ true \\
\qquad), \\
\qquad addsimbol(NSchema, schema, [Decls, Tree]).
\end{array}
$$

Figura 2.1: Representación de la regla gramatical que reconoce un esquema.

El funcionamiento del analizador sintáctico es el siguiente:

- En esta primera versión del prototipo el analizador sintáctico sólo tiene la capacidad de dar una respuesta de éxito o fracaso durante el proceso de análisis. Cuando se detecte un error, se envía un mensaje a la pantalla y se detiene el análisis.
- Reconoce como comentarios aquellas líneas que comienzan con el símbolo %.
- Cualquier información que no esté comprendida dentro de algún bloque de especificación o que no sea un comentario, causa un mensaje de fracaso en el intento de reconocer una especificación.

2.3.3 Tabla de símbolos

La tabla de símbolos está representada por una estructura de un árbol binario ordenado alfabeticamente [CoJ+87]. Cada nodo de este árbol tiene la siguiente estructura.

simbol(Name,Class,Adc,LSimbol,RSimbol).

donde:

simbol es el nombre de la cláusula que describe un nodo del árbol.

Name el nombre del símbolo en la tabla.

Class la clase de símbolo:

> **varg** Variable global.
>
> **cte** Constante.
>
> **schema** El nombre del símbolo es el de un esquema.
>
> **gendef** El nombre del símbolo corresponde a una lista de parámetros genéricos, ver sección 1.5.3.
>
> **bastype** Un tipo básico.
>
> **simtype** Un tipo con datos simples.

$$
\begin{array}{l}
insertsimbol(Name, Class, Adc, simbol(Name, Class, Adc, \_, \_)) :-!.\\
insertsimbol(Name, Class, Adc, simbol(Name2, \_, \_, Bef, \_)) :-\\
\quad strcmp(Name, Name2, R),\\
\quad R < 0,\\
\quad insertsimbol(Name, Class, Adc, Bef).\\
insertsimbol(Name, Class, Adc, simbol(Name2, \_, \_, \_, Aft)) :-\\
\quad strcmp(Name, Name2, R),\\
\quad R > 0,\\
\quad insertsimbol(Name, Class, Adc, Aft).
\end{array}
$$

Figura 2.2: Código que inserta un símbolo en la tabla.

Adc El significado de este campo está en función de la clase de símbolo en el nodo y es usado para guardar el tipo de *Name*. Para los tipos de datos simples, es una lista conteniendo en el primer elemento el nombre del tipo y el resto de la lista son los valores para ese tipo. Cuando se trata de un esquema, también tendremos una lista, pero de dos elementos, el primero es la lista de declaraciones del esquema y el segundo, es el árbol de sintaxis abstracta para el predicado.

LSimbol Es el símbolo cuyo nombre es alfabéticamente menor a *Name*, y es ubicado en el nodo izquierdo.

RSimbol Es el símbolo cuyo nombre es alfabéticamente mayor a *Name*, por lo que es ubicado en el nodo derecho.

La figura 2.2 muestra la implantación del predicado que inserta un símbolo en la tabla. La figura 2.3 muestra un ejemplo de la tabla de símbolos con información sobre:

$[NAME, DATE]$

book

$s : \mathbb{P}\, NAME$

En el análisis de tipos, es necesario que algunos símbolos sean almacenados temporalmente, eso es debido a que existen declaraciones que afectan de forma local a

$$
\begin{array}{l}
simbol(NAME, basic, null, \\
\quad\quad simbol(book, schema, [[d(s, power(NAME))], []], \\
\quad\quad\quad \_G5489, \\
\quad\quad\quad simbol(DATE, basic, null, \_G6390, \_G6391), \\
\quad\quad \_G3456 \\
\quad\quad).
\end{array}
$$

Figura 2.3: Tabla de símbolos con algunos elementos. Los identificadores _GXXXX indican variables no instanciadas.

alguna expresión, en especial cuando se trata de predicados cuantificados y en el caso de los esquemas. Por tal motivo, se ha definido una tabla de símbolos locales que sirve para tal propósito.

Esta tabla de símbolos locales, es una pila donde cada elemento de la pila es una lista de declaraciones; puesto que pueden existir varios niveles de localidad, cada vez que se empila un elemento las declaraciones en esta lista son las que se encuentran inmediatamente accesibles. Las dos operaciones básicas sobre la pila (tabla local) son empilar (push) y desempilar (pop) una lista de definiciones.

Cuando se requiere información sobre el tipo de un símbolo la búsqueda se realiza de la siguiente forma:

1. Si existen símbolos en la tabla de símbolos locales entonces se desempila la lista en el tope y se realiza la búsqueda. Si se tiene éxito entonces se regresa el tipo del símbolo encontrado. Si no, entonces se desempila la próxima lista y se repite el proceso. En el caso de no haber encontrado una ocurrencia de lo que se desea se pasa al siguiente paso. La razón de realizar la búsqueda en todas las listas empiladas es porque puede darse el caso de que una definición local afecte a sucesivas definiciones locales, esto puede verse en un predicado donde las variables cuantificadas tienen un alcance en el predicado que afecta la cuantificación y a su vez ese predicado puede tener otras variables cuantificadas. Por ejemplo cuando se analiza

   $$\forall x : \mathbb{N} \bullet (\exists y : \mathbb{N} \bullet y = 5 * x)$$

   la lista que contiene a la variable x es empilada en la tabla local (llamamos a esta lista L_1), continua el análisis y como se encuentra otro predicado que introduce la variable y, entonces la siguiente lista local que contiene a y es

```
getglobalsimb(Name, Class, Type) :-
        gettable(Table),
        existsimbol(Name, Table, Class, Type).
existsimbol(Name, simbol(Nins, Class, T, _, _), Class, Type) :-
        Name == Nins,
        (Class == basic
        -> Type = [Nins, []]
        ; Type = T
        ).
existsimbol(_Name, simbol(_Name2, _, _, B, A), undef, undef) :-
        var(B), var(A).
existsimbol(Name, simbol(Name2, _, _, Bef, _), Class, Type) :-
        strcmp(Name, Name2, R),
        R < 0,
        existsimbol(Name, Bef, Class, Type).
existsimbol(Name, simbol(Name2, _, _, _, Aft), Class, Type) :-
        strcmp(Name, Name2, R),
        R > 0,
        existsimbol(Name, Aft, Class, Type).
```

Figura 2.4: Código que consulta la tabla de símbolos globales.

empilada (L_2). En el momento de analizar $y = 5 * x$, la lista disponible es L_2 pero en ella no existe x así que se busca una referencia a x en la lista empilada antes: L_1. De esta forma se obtiene la información sobre x.

2. Si no se cumplió la condición del paso 1 el símbolo que se busca no esta definido localmente, entonces ahora lo que se realiza es una búsqueda en la tabla de símbolos globales. Si se tiene éxito se regresa el tipo del símbolo deseado, de otra forma, un error es reportado y el proceso de análisis termina.

Las figuras 2.4, 2.5, 2.6 muestran la implantación de los procesos descritos.

$$
\begin{aligned}
&getlocalsimb(Table, Name, Type) :-\\
&\qquad search(Name, Table, Type).\\
&search(Name, [Tb1 \mid Tbn], Type) :-\\
&\qquad getsimbol(Name, Tb1, T),\\
&\qquad (T == not\_found\\
&\qquad - > search(Name, Tbn, Type)\\
&\qquad ;\ Type = T\\
&\qquad).\\
&search(\_, [], undef).
\end{aligned}
$$

Figura 2.5: Código que encuentra información de un símbolo en la tabla de símbolos locales.

$$
\begin{aligned}
&gettypeofsimbol(Name, [LType, GType], Class) :-\\
&\qquad localdefs(Vars),\\
&\qquad getlocalsimb(Vars, Name, LType),\\
&\qquad getglobalsimb(Name, C, GType),\\
&\qquad (var(GType)\\
&\qquad - > (GType = undef, Class = [])\\
&\qquad ;\ Class = C\\
&\qquad).
\end{aligned}
$$

Figura 2.6: Código que encuentra información de un símbolo en la tabla de símbolos (locales o globales).

2.3.4 Arbol de sintaxis abstracta

Puesto que la notación Z se basa en una notación matemática, nos tendremos que enfrentar muy a menudo con expresiones matemáticas, además de poder realizar un análisis sintáctico es necesario realizar un análisis semántico como lo es la comprobación de tipos. Para tal efecto es necesario una vez realizado el análisis sintáctico guardar la información de la estructura de una expresión matemática (en general un predicado). De esta forma, se ha pensado en una estructura de árbol de sintaxis abstracta que represente la expresión que se quiere analizar.

Estructura

Un árbol de sintaxis abstracta[AhA+90] tiene la siguientes características:

- Es binario.
- La raíz de cada subarbol representa un operador.
- Las ramas constituyen los operandos.

En la figura 2.7 se muestran los árboles para dos expresiones.

La estructura del árbol de sintaxis abstracta (en la sintaxis de Prolog) es la siguiente:

$$rama(RamaIzq, RamaDer, [TDI, I])$$

donde:

rama es el nombre de la cláusula para un nodo en el árbol.

RamaIzq es la rama izquierda del nodo actual, esta puede ser **null** si el nodo es una hoja.

RamaDer es la rama derecha del nodo, esta también puede ser **null**.

TDI indica el tipo de información contenida en el nodo y puede ser:

> **oper** indica que el nodo es un operador aritmético, de conjuntos, funciones y relaciones.
>
> **oplog** indica que el nodo es un operador lógico, $\wedge, \vee, \Rightarrow$, etc.

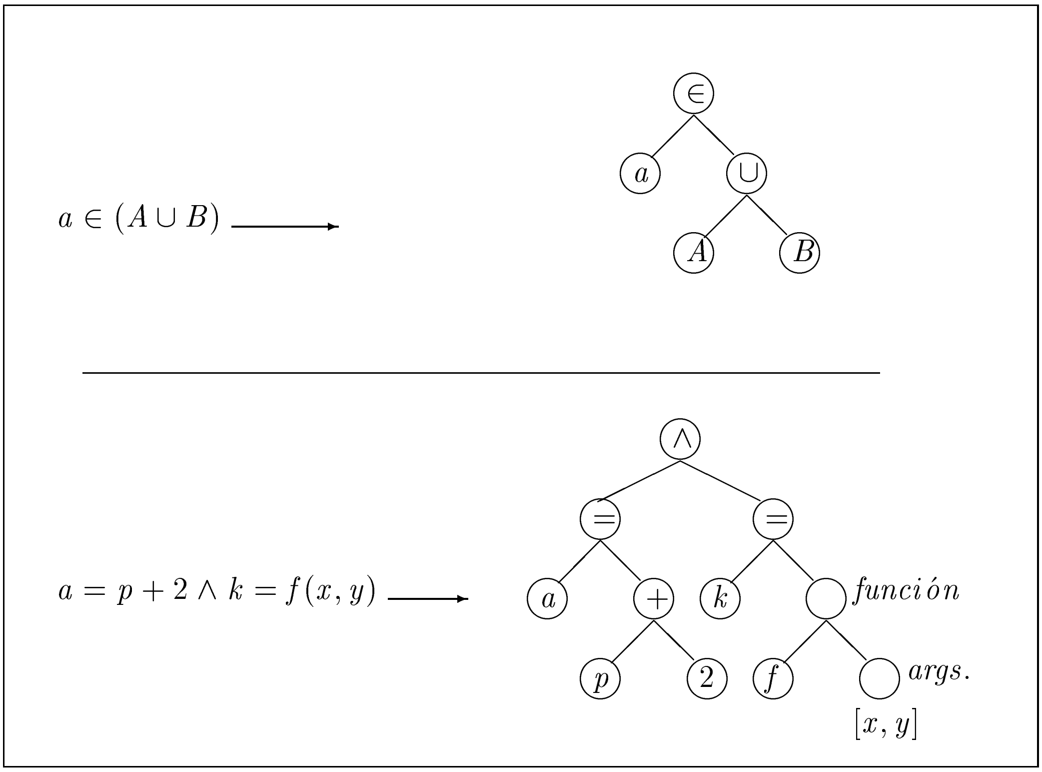

Figura 2.7: Arboles de sintaxis abstracta para las expresiones dadas.

$$
\begin{array}{l}
parse\_aritmetic\ (L, M, LHS, Tree) :- \\
\quad infun(L, L1, Op), \\
\quad nondel\_pop(Opant), \\
\quad prec([oper, Op], P1), \\
\quad prec(Opant, P2), \\
\quad (P1 > P2 \\
\quad -> (push([oper, Op]), \\
\qquad expression1(L1, L2, RHS), \\
\qquad pop(\_), \\
\qquad exp1p(L2, M, rama(LHS, RHS, [oper, Op]), Tree) \\
\qquad) \\
\quad ;\ !, fail \\
\quad).
\end{array}
$$

Figura 2.8: Código en Prolog del algoritmo para construcción del árbol de sintaxis abstracta.

opnd el nodo es un operando. Dentro de los operandos podemos tener, variables, constantes, números, tuplas y conjuntos.

I es la información en el nodo, es decir, $+, -, *, \rightarrow$, etc. para el caso de **oper**, $\wedge, \vee$, etc. para **oplog** y números, identificadores, etc. para el caso de **opnd**.

El árbol de sintaxis abstracta se construye a nivel de un predicado de Z, esto es, el uso de la regla gramatical que describe un predicado en Z tiene asociada la acción de construir el árbol.

Algoritmo de construcción

El algoritmo [Web] usado para la construcción del árbol es del tipo iterativo, así que dada la naturaleza recursiva de Prolog, se modificó para llevarlo a una forma recursiva. En la figura 2.8 se muestra el código en Prolog de tal algoritmo.

Los parámetros de *parse_aritmetic* son:

L Cadena de tokens de entrada.

M Cadena de tokens después del análisis.

LHS El árbol correspondiente a la subexpresión del lado izquierdo del operador, inicialmente corresponde al árbol que se construye con el primer token en una expresión.

Tree El árbol construido.

Para describir el funcionamiento del algoritmo usaremos como ejemplo la expresión

$$a + b * c + d$$

donde $prec(+) < prec(*)$.

- Cuando se realiza el primer llamado a *parse_aritmetic* se ha procesado el primer token (a) por el analizador sintáctico, así que este primer llamado tiene la forma[4]

  $$parse\_aritmetic([t([+, simb]) \mid Resto], M, r(null, null, [opnd, a]), Tree)$$

  - *infun* checa que el siguiente token sea un operador infijo, y consume $+$, después del llamado a *infun* tendremos:

    $$infun([t([+, simb]) \mid \_], [t([word, b]), \_], +)$$

  - Puesto que el análisis de la expresión ha iniciado, entonces la pila esta vacía; después del llamado a las siguientes tres rutinas tenemos:

    $$nondel\_pop([opnd, \_]),$$
    $$prec([oper, +], 3),$$
    $$prec([opnd, \_], 0),$$

  - La prioridad del operador que se ha procesado ($+$) resulta ser mayor a la del "operador anterior", así se procede a guardar este operador en la pila, y se hace un llamado a *expression*1:

    $$push([oper, +]),$$
    $$expression1([t([opnd, b]) \mid \_], L2, RHS),$$

[4]Deberá interpretarse a las variables que inicien con mayúsculas como variables en el contexto de Prolog.

– El llamado a *expression*1 hace que el analizador sintáctico se encargue de procesar el resto de la expresión, esta rutina efectúa el llamado a *parse_aritmetic* hasta encontrar un operador cuya prioridad sea menor o igual a la del operador inmediato anterior, entonces

$$
\begin{aligned}
&expression1([t([opnd, b]) \mid \_], [t([opnd, +]) \mid \_],\\
&\qquad r(r(null, null, [opnd, b]), r(null, null, [opnd, c]), [oper, *])\\
&\qquad),
\end{aligned}
$$

es el resultado después del llamado. Esto es porque la prioridad de $+$ es menor que la de $*$.

– Es en este instante cuando se desempila el operador que se guardó antes del llamado a *expression*1 $(+)$ y nuevamente la pila queda vacía.

– Finalmente se efectúa un llamado a *exp*1*p* donde nuevamente se hace otro llamado a *parse_aritmetic*.

$$parse\_aritmetic([t([+, simb]) \mid Resto], M, arbol\_anterior, Tree)$$

  * Otra vez, el operador que se procesa es $+$.
  * Puesto que la pila está vacía, $+$ es el operador con mayor prioridad.
  * Se salva el operador $+$.
  * El llamado a *expression*1 regresa el último operando de la expresión inicial, entonces

$$expression1([t([opnd, d])], [], r(null, null, [opnd, d]))$$

    es el resultado después del llamado.
  * Se desempila el operador guardado.
  * *exp*1*p* al no tener más entrada que procesar regresa el mismo árbol que se le envió como LHS.

– En la figura 2.9 muestra el árbol correspondiente a la expresión de inicio.

La gramática que describe la notación Z, se ha escrito de tal forma que el algoritmo pueda construir un árbol para una subexpresión (expresiones entre paréntesis, listas de expresiones como lo son tuplas, secuencias y conjuntos), con sólo empilar un operador ficticio cuya prioridad es 0 (la más baja) y con esto estaremos emulando el caso donde se trata una nueva expresión.

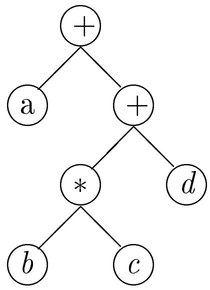

Figura 2.9: Arbol de sintaxis abstracta resultante del ejemplo.

Si bien este algoritmo realiza el trabajo más fuerte en la construcción del árbol, tiene otras acciones adicionales y que corresponden a otras reglas gramaticales. Por ejemplo, las que identifican operandos y construyen el correspondiente nodo; es importante mencionar que los conjuntos, tuplas y secuencias se consideran operandos.

2.4 Manejo de errores

Nuestra implantación del prototipo es recursiva, así que el intérprete de Prolog SWI-Prolog[WiJ] proporciona algunos predicados que son usados para el manejo de errores.

Los predicados son:

block(*Label*, *Meta*, *ExitValue*) Ejecuta *Meta* en un bloque. *Label* es el nombre del bloque en que se ejecuta *Meta*. *ExitValue* es unificado con el segundo argumento de un llamado a *exit* y que es invocado por *Meta*.

exit(*Label*, *Value*) Un llamado a *exit* hace que el bloque más interno con el cual es unificado *Label*, termine. *ExitValue* del predicado *block* es unificado con *Value*. Si la unificación falla entonces el bloque falla.

El concepto de **bloque** que es usado en estos predicados es semejante al de subrutina. Esto es, el predicado *block* ejecuta *Meta* como si fuera una subrutina y *exit* se encarga de terminar la ejecución de esa subrutina. Lo importante a destacar es que la subrutina que se ejecuta es de naturaleza recursiva y no importa que profundidad de recursión exista, cuando se hace un llamado a *exit* la subrutina inicial termina su ejecución. La figura 2.10 muestra la forma en que se aplican estos predicados.

De esta forma los bloques usados son:

```
block_specification :-
    block(specification, nextntokens(Ltoks, 0), Error),
    (var(Error)
    -> (block(specification, spec(Ltoks, [t([eof | _])]), SError),
        (var(SError)
        -> write_ln('Parsing and Type Checking OK')
        ; write('Error :'), write(SError)
        )
        )
    ; write(Error),
    fail
    ).
```

Después cuando se analiza un identificador :

```
ident(L, M, [Word]) :- match(t([argm, ctex]), L, L1),
    (match(t([word, Word]), L1, M), check_reservada(Word)
    -> true
    ; swritef(S,' Un identificador es esperado'),
    exit(specification, S)
    ).
```

Figura 2.10: Rutinas de manejo de errores

- Análisis sintáctico. Este es el bloque más externo. La etiqueta usada para identificarlo es **specification**
- Comprobación de Tipos. Es ejecutado por el anterior. Se identifica con **type-checking**
- Cálculo de precondiciones. También es ejecutado por el primero. Este bloque se identifica con la etiqueta **precondition**.

Es importante destacar que el prototipo finaliza su análisis en el primer error encontrado. Cada uno de los bloques de comprobación y precondiciones hace que el de sintaxis termine. En pantalla es reportado el error causante de la interrupción.

Dentro del proceso de análisis sintáctico se interactúa con el proceso de comprobación de tipo en dos formas, la primera para determinar el tipo de las variables declaradas en el bloque analizado y después para comprobar que la expresión (que representa la relación entre las variables) sea consistente de acuerdo al sistema de tipos. En cualquier de los dos casos el árbol de sintáxis abstracta que se construye en la fase de análisis sintáctico es el producto que se exporta a la fase de comprobación de tipos y posteriormente es una pieza fundamental para llevar a cabo el cálculo de precondiciones cuando es requerido mediante el cálculo de esquemas.

Capítulo 3

Comprobación de Tipos

Además de los aspectos sintácticos de un lenguaje, también es deseable comprobar algunos aspectos semánticos. Esta comprobación llamada comprobación estática, garantiza la detección de alguna clase de errores de programación. Algunos de los ejemplos de comprobación estática son: comprobaciones de tipos y comprobaciones del flujo de control [AhA+90].

Puesto que Z es un lenguaje altamente tipificado [SpM1], la buena escritura de una especificación requiere que no existan expresiones sin sentido y para esto se presenta el diseño de un comprobador de tipos.

El diseño de un comprobador de tipos para un lenguaje se basa en la información acerca de las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tiposa las construcciones del lenguaje.

3.1 Reglas de asignación de tipos

A diferencia del esquema explicativo usado hasta el momento donde se da un ejemplo para ilustrar la implantación de los conceptos teóricos que se exponen, ahora primero se listan las reglas de asignación de tipos y la parte que ejemplifica la implantación de las mismas se pospone hasta la sección 3.3, ya que antes debemos explicar algunos predicados que se usan para poder llevar a cabo la implantación de las reglas.

La notación usada para las reglas de asignación de tipos es:

$$\frac{\textit{Expresión}}{\textit{Tipo}} \quad [\ \textsf{Información}\ ]$$

donde:

Expresión Es la expresión de la que se desea obtener su tipo. En el caso de expresiones de la forma $a : Exp$, se desea obtener el tipo de *Exp* el cual será el tipo de la variable a.

Tipo Es el tipo de *Expresión*.

Información Es una breve explicación sobre el bloque de construcción al que se le puede aplicar la regla, el conjunto al que debe pertenecer algún componente de *Expresión*, la condición que debe cumplir *Expresión*, etc.

Las reglas de asignación de tipos para declaración de varièbles son[JoW][SpM1] :

$\dfrac{x : T}{T}$ [T tipo básico]	Cuando declaramos variables usando tipos básicos, el tipo de la variable es T.
$\dfrac{x : X}{T}$ [tipo(X) = $\mathbb{P}$(T), T básico]	Sie X es un conjunto y sus elementos son de tipo T, la declaración implica que $x \in X$ por lo tanto x es de tipo T. Por ejemplo, en la declaración $n : \mathbb{N}$, el tipo de $\mathbb{N}$ es $\mathbb{P}\,\mathbb{Z}$ entonces los elementos de $\mathbb{N}$ tienen tipo $\mathbb{Z}$ por lo tanto $tipo(n) = \mathbb{Z}$.
$\dfrac{x : a \mathinner{..} b}{\mathbb{Z}}$ [Rango]	La expresión $a \mathinner{..} b$ genera un subconjunto de $\mathbb{Z}$ así por la regla anterior $tipo(x) = \mathbb{Z}$.

$\dfrac{numero}{\mathbb{Z}}$ [Cte. numérica]	El tipo del número 123 es $\mathbb{Z}$.
$\dfrac{x : \mathbb{P}\, T}{\mathbb{P}\, tipo(T)}$ [Potencia]	Si declaramos $cbasic : \mathbb{P}\, Colores$, donde $Colores$ es un tipo básico; $cbasic \subset Colores \Rightarrow cbasic \in \mathbb{P}\, Colores$ así su tipo es $\mathbb{P}\, Colores$.
$\dfrac{x : A \times B}{tipo(A) \times tipo(B)}$ [P. Cartesiano]	La variable $x \in A \times B$, representa un par ordenado, si A y B son básicos entonces su tipo es $A \times B$.
$\dfrac{x == Exp}{tipo(Exp)}$ [Def. Abreviada]	Una expresión abreviada sirve para darle un nombre corto a una expresión que es muy extensa (en escritura), por lo que su tipo es el mismo de la expresión que abrevia.
$\dfrac{x : A\ RF\ B}{\mathbb{P}(A \times B)}$ [RF $\in \{\leftrightarrow, \rightarrow, \ldots\}$]	Una relación es un subconjunto de un producto cartesiano, por lo que de acuerdo a la regla de $\mathbb{P}$ tenemos que el tipo de una relación es $\mathbb{P}(A \times B)$ donde A y B son básicos. Lo mismo aplica a las funciones ya que son un caso particular de relaciones.

$\dfrac{x : \text{seq}\, X}{\mathbb{P}(\mathbb{Z} \times \mathit{tipo}(X))}$ [Secuencia]	En el capítulo 2 se vio que una secuencia de elementos de un conjunto X no es más que una función de $\mathbb{N}$ a X, por la regla anterior podemos decir que el tipo de la secuencia es $\mathbb{P}(\mathbb{Z} \times \mathit{tipo}(X))$.
Conjunto que se describe elemento por elemento.	
$\dfrac{\{\mathit{Exp}_1, \mathit{Exp}_2, \ldots, \mathit{Exp}_n\}}{\mathbb{P}\ \mathit{tipo}(\mathit{Exp}_1)}$	Un conjunto tiene tipo $\mathbb{P}\, X$ donde X es el tipo de los elementos del conjunto mismo: $\mathit{tipo}(\mathit{Exp}_1) = \ldots = \mathit{tipo}(\mathit{Exp}_n)$.
Conjuntos que se describen mediante predicados.	
$\dfrac{\{\mathit{Decl} \bullet \mathit{Expr}\}}{\mathbb{P}\ \mathit{tipo}(\mathit{Expr})}$	De la sec. 2, los elementos del conjunto quedan descritos por *Expr* y por la regla $\mathbb{P}$ se tiene que el tipo del conjunto es $\mathbb{P}\, \mathit{tipo}(\mathit{Expr})$. *Decl* es el conjunto de declaraciones que definen el dominio de trabajo.
$\dfrac{\{\mathit{Decl} \mid \mathit{Expr}\}}{\mathbb{P}\ (\mathit{tipo}(d_1) \times \ldots \times \mathit{tipo}(d_n))}$	De la sec. 2, los elementos del conjunto quedan descritos por la tupla formada en *Decl*, la cual tiene la forma $d_1 : T_1;\ , \ldots ;\ d_n : T_n$.

$$\frac{\{Decl \mid Pred \bullet Expr\}}{\mathbb{P}\ tipo(Expr)}$$	De la sec. 2, los elementos del conjunto quedan descritos por $Expr$ y por la regla $\mathbb{P}$ se tiene que el tipo del conjuntoes $\mathbb{P}\, tipo(Expr)$.
Tupla forma extensional	
$$\frac{(Exp_1, \ldots, Exp_n)}{tipo(Exp_1) \times \ldots \times tipo(Exp_n)}$$	El tipo de una tupla queda determinado por el producto cartesiano de los tipos de cada elemento en la tupla ($tipo(Exp_i)$; $i = 1, \ldots, n$).
$$\frac{\langle Exp_1, \ldots, Exp_n \rangle}{\mathbb{P}(\mathbb{N} \times tipo(Exp_1))}$$	El tipo de una secuencia dada en forma extensional, queda completamente determinado por medio del tipo de sus componentes el cual debe ser el mismo: $tipo(Exp_1) = \ldots = tipo(Exp_n)$.
$$\frac{Id[X], Id[A]}{tipo(Id[A/X])} \quad [\ \mathsf{Def.Gen}\ ]$$	Suponemos que $Id[X]$ previamente se ha definido como una construcción genérica donde X es un parámetro genérico, cuando usemos ésta construcción con una expresión no genérica como A el tipo de la construcción se obtiene substituyendo ($[A/X]$) las ocurrencias de X en Id por A.

$\dfrac{A\ opset\ B}{tipo(A)}$ [opset $\in \{\cup, \cap, \ldots\}$]	Para el caso de las operaciones entre conjuntos, el sistema de tipos requiere que $tipo(A) = tipo(B)$.
$\dfrac{t \in T}{true}$ [Membresía]	Para el caso del operador $\in$, deberá cumplise que $tipo(T) = \mathbb{P}\ tipo(t)$ y en este caso el valor de *true* indica la consistencia de la expresión.
$\dfrac{Exp_A = Exp_B}{true}$ [Igualdad]	Cuando se tiene una igualdad, necesariamente el tipo de las expresiones involucradas debe ser el mismo: $tipo(Exp_A) = tipo(Exp_B)$. Derivar *true* indica la consistencia de la igualdad.
Operaciones sobre relaciones.	
$\dfrac{\mathrm{dom}\, R}{\mathbb{P}\, X}$ [Dominio]	$R : X \leftrightarrow Y$, X y Y tipos.
$\dfrac{\mathrm{ran}\, R}{\mathbb{P}\, Y}$ [Rango]	$R : X \leftrightarrow Y$, X y Y tipos.

$\dfrac{S \lhd R}{tipo(R)}$ [$\mathsf{tipo(S) = tipo(}\mathrm{dom}\,\mathsf{R)}$]	Restricción Dominio. Se obtiene una relación excluyendo los pares $(x, y) \in R$ tales que $x \notin S$. El tipo de la nueva relación es el de R.
$\dfrac{R \rhd S}{tipo(R)}$ [$\mathsf{tipo(S) = tipo(}\mathrm{ran}\,\mathsf{R)}$]	Restricción rango. Este operador obtiene la relación resultante de eliminar de R los pares (x, y) tal que $y \notin S$. De esta forma el tipo de la nueva relación es el tipo de R.
$\dfrac{R \ntriangleleft S}{tipo(R)}$ [$\mathsf{tipo(S) = tipo(}\mathrm{dom}\,\mathsf{R)}$]	Substracción dominio. La relación resultante se obtiene excluyendo los pares $(x, y) \in R$ tales que $x \in S$. El tipo de la nueva relación es el de R.
$\dfrac{R \ntriangleright S}{tipo(R)}$ [$\mathsf{tipo(S) = tipo(}\mathrm{ran}\,\mathsf{R)}$]	El tipo de la nueva relación es $tipo(R)$. Sus elementos son los (x, y) tales que $y \notin S$.
$\dfrac{R(\!\mid S \mid\!)}{tipo(\mathrm{ran}\,R)}$ [$\mathsf{tipo(S) = tipo(}\mathrm{dom}\,\mathsf{R)}$]	Imagen relacional. El resultado de aplicar este operador es un conjunto del tipo de $\mathrm{ran}\,R$. Este conjunto contiene aquellos $y \in \mathrm{ran}\,R$ tal que si $(x, y) \in R$ entonces $x \in S$.

Para fines prácticos, en el caso de enteros ($\mathbb{Z}$), además de la asignación del tipo se obtiene información sobre el entero tratado, para el caso de rangos se obtiene la información de los límites del mismo.

3.2 Inferencia de tipos en una expresión

Una vez que se tiene el árbol de sintaxis abstracta descrito en la sección 2.3.4, correspondiente a una expresión dada, pueden realizarse las siguientes tareas:

- Obtener el tipo de la expresión para poder asignarlo a una variable. Esto generalmente sucede cuando se analiza la parte declarativa de un párrafo.
- Comprobar que la expresión tenga sentido. Esta operación se realiza cuando se analiza la parte axiomática de un párrafo

3.2.1 Obtención del tipo de una expresión

La parte declarativa de cualquier párrafo en Z se describe mediante las siguientes reglas gramaticales:

$$\begin{array}{rcl} Decl\_part & \rightarrow & Basic\_Decl[;\ Basic\_Decl]^* \\ Basic\_Decl & \rightarrow & Decl\_Name : Expresion \\ Decl\_Name & \rightarrow & Ident[, Ident]^* \\ Ident & \rightarrow & identificador \end{array}$$

En este sentido, una vez que se tienen los identificadores que representan variables, se construye el árbol de sintaxis abstracta para la expresión y finalmente se infiere el tipo de esta expresión. Esto en general se logra mediante un recorrido del árbol: primero se visita la raíz, el nodo izquierdo y finalmente el nodo derecho y aplicando las reglas de asignación de tipos , cuando se pueda aplicar alguna. El tipo obtenido y el identificador declarado son agregados a la tabla de símbolos.

3.2.2 Comprobación de expresiones

Cuando se analiza la parte axiomática de un párrafo, el primer paso lo resuelve el analizador sintáctico construyendo el árbol de sintaxis abstracta para la expresión en la parte axiomática del párrafo en cuestión. El hecho de obtener este árbol indica que se ha terminado el análisis sintáctico del párrafo actual y lo que resta es el análisis semántico del mismo. La diferencia entre esta etapa y la anterior es que aquí no obtenemos ningún tipo para la expresión analizada, en su lugar sólo sabemos si la expresión o parte de ella tiene sentido o no. Más adelante, en este capítulo se explicará la forma de atacar cada párrafo de Z.

3.3 Implantación

Las tareas de inferencia y comprobación de tipos se realizan mediante los predicados *gettypetree*/2 y *typeofoper*/4, los cuales son la implantación de las reglas de asignación de tipos dadas al inicio de este capítulo.

3.3.1 Predicado *gettypetree*/2

Este predicado es el encargado de obtener el tipo del árbol de sintaxis abstracta que se recorre. Existe un predicado de este tipo para cada regla de asignación aplicable a un nodo. Además se encarga de resolver las sustituciones necesarias para el caso de las definiciones genéricas. Existen también cláusulas de este predicado que mezclan una o más reglas de asignación para poder inferir el tipo. El proceso de selección de cuál regla es aplicada es responsabilidad del backtracking de Prolog. Si ninguna regla puede ser aplicada entonces existe una regla (que es la última en la lista a seleccionar) que sirve para enviar un mensaje de error.

La forma del predicado es:

$$gettypetree(Tree, Type)$$

donde:

Tree Es el árbol que es recorrido para obtener su tipo.

Type Es el tipo de la expresión que se infiere.

Para describir como se realiza el paso de una regla de asignación hacia el predicado en Prolog, tomaremos como ejemplo un par de reglas.

Ejemplo 1. Tomaremos la regla para obtener el tipo de una secuencia. La regla de asignación es:

$$\frac{\langle Exp_1, \ldots, Exp_n \rangle}{\mathbb{P}(\mathbb{N} \times tipo(Exp_1))} \quad [\ \mathsf{tipo}(\mathsf{Exp}_1) = \ldots = \mathsf{tipo}(\mathsf{Exp}_\mathsf{n})\ ]$$

1. Para empezar debemos encontrar la representación de la secuencia como un árbol. Así que esta representación es:

   $$rama([Exp_1, \ldots, Exp_n], null, [opnd, seq])$$

   de lo anterior tendremos que la raíz incluye la información de un operando, el nodo izquierdo contiene la lista de expresiones de la secuencia y no hay información en el nodo derecho.

2. Puesto que en una secuencia, todos sus elementos deben tener el mismo tipo, entonces tendremos que obtener el tipo de cada subexpresión en la lista. Esto es logrado consiguiendo n-soluciones a $gettypetree/2$ una para cada expresión. Si escribimos el siguiente predicado

   $$\begin{array}{l} gettypeinlistexpr([], []). \\ gettypeinlistexpr([Exp_1 \mid Exp_n], [T_{Exp_1} \mid T_{Exp_n}]) :- \\ \qquad gettypetree(Exp_1, T_{Exp_1}), \\ \qquad gettypeinlistexpr(Exp_n, T_{Exp_n}). \end{array}$$

   conseguir la solución de este predicado implica el conocimiento de los tipos de cada expresión.

3. Conociendo la lista de tipos, ahora lo que debe realizarse es verificar que efectivamente todos los tipos obtenidos sean iguales. Para tal efecto escribamos el siguiente predicado que realiza dicha tarea.

   $$\begin{array}{l} sametypes(\_, []). \\ sametypes(Type_1, [Type_1 \mid Type_n]) :- \\ \qquad sametypes(Type_1, Type_n). \end{array}$$

4. Cuando se ha verificado que coincidan los tipos en la lista, el paso final es regresar el tipo de toda la expresión, por lo que el predicado correspondiente a la regla de asignación es

   $$\begin{array}{l} gettypetree(rama(Lista, null, [opnd, seq]), \\ \qquad\qquad power(cross(z([0, inf]), Type_1))) :- \\ \qquad gettypeinlistexpr(Lista, [Type_1 \mid Types]), \\ \qquad sametypes(Type_1, Types). \end{array}$$

   donde $power(cross(z([0, inf]), Type_1))$ es la representación en Prolog de $\mathbb{P}(\mathbb{N} \times Type_1)$.

Es importante destacar que para efecto del ejemplo, no se ha descrito la acción que se toma cuando los elementos de la lista no tienen tipos idénticos. Brevemente se puede decir que se tiene que reportar un mensaje de error.

Ejemplo 2. Veamos ahora el caso para obtener el tipo de una expresión con operadores, tal como, $\mathbb{P}(A \times B)$.

Para lograr inferir el tipo de esta expresión deben usarse una serie de reglas de asignación.

$$\begin{array}{l} \mathit{gettypetree}(\mathit{rama}(\mathit{Left}, \mathit{Right}, [\mathit{oper}, \mathit{Oper}]), \mathit{Type}) : - \\ \qquad\qquad \mathit{gettypetree}(\mathit{Left}, \mathit{Type}_1), \\ \qquad\qquad \mathit{gettypetree}(\mathit{Right}, \mathit{Type}_2), \\ \qquad\qquad \mathit{typeofoper}(\mathit{Oper}, \mathit{Type}_1, \mathit{Type}_2, \mathit{Type}). \end{array}$$

El código anterior, es el resultado de la mezcla de varias reglas de asignación.

1. Se trata de una expresión en la que se deben conocer los tipos de ambos lados del operador. Posiblemente uno de estos subarboles, el derecho, es instanciado a *null*.

2. De esta forma con

   $$\mathit{gettypetree}(\mathit{Left}, \mathit{Type}1)$$

   estamos infiriendo el tipo del subarbol izquierdo; para tal fin pueden aplicarse varias reglas de asignación al momento de encontrar una solución a ese antecedente.

3. El siguiente paso es obtener el tipo del subarbol derecho lo cual es realizado mediante la solución del antecedente

   $$\mathit{gettypetree}(\mathit{Right}, \mathit{Type}2)$$

4. Finalmente, una vez inferidos los tipos de los subarboles izquierdo y derecho, lo que queda finalmente es verificar que estos tipos cumplan con la definición del operador que representa la raíz. Esto se logra obteniendo la solución a

   $$\mathit{typeofoper}(\mathit{Oper}, \mathit{Type}1, \mathit{Type}2, \mathit{Type})$$

   que es descrito en la siguiente sección. Básicamente lo que hace este predicado es aplicar la regla de asignación que coincida con el operador *Oper* y la variable *Type* es instanciada al tipo resultante de realizar la operación.

3.3.2 Predicado *typeofoper/4*

En expresiones donde intervienen operadores es necesario verificar que la aplicación de tales operadores corresponda con su definición, la definición de los operadores obedece, nuevamente, a las reglas de asignación de tipos.

Las cláusulas del predicado *typeofoper/4* tienen la siguiente forma

$$typeofoper(Oper, LType, RType, Type)$$

donde:

Oper es el operador que es analizado.

LType es el tipo de la subexpresión a la izquierda del operador.

RType el tipo de la subexpresión a la derecha del operador, este tipo puede estar instanciado a *null*, lo que significa que el operador es un operador prefijo.

Type Es el tipo de la expresión *LType Oper Rtype*.

En algunos casos el tipo que se infiere es **ok**, que indica que el operador ha sido usado correctamente, esto sucede para los operadores de relaciones aritméticas $<, >, \leq, \geq, \neq$, y para los operadores $\subset, \subseteq, \in, \notin$.

El siguiente ejemplo muestra la implantación de algunas reglas de asignación.

Ejemplo

- Constructor potencia.

$$\frac{x : \mathbb{P}\, T}{\mathbb{P}\, tipo(T)}$$

Implantación.

$$typeofoper(power, Type, \_, power(Type)).$$

- Membresia en conjuntos.

$$\frac{t \in T}{true}$$

Implantación.

$$typeofoper(in, Type1, power(Type1), ok).$$

- Operaciones sobre Conjuntos.

$$\frac{A \;\; opset \;\; B}{tipo(A) = tipo(B)} \quad [\,\mathsf{opset} \in \{\cup, \cap, \setminus, \ldots\}\,]$$

Implantación

$$\begin{array}{l} typeofoper(Oper, power(Type), power(Type), power(Type)) :- \\ \qquad\qquad member(Oper, [cup, backslash, cap, setminus]). \end{array}$$

- Dominio.

$$\frac{\operatorname{dom} R}{\mathbb{P}\, X} \quad [\,\mathsf{R : X \leftrightarrow Y}\,]$$

Implantación.

$$typeofoper(dom, power(cross(TDom, \_)), \_, power(TDom)).$$

Predicados de Z

Hasta el momento, dentro de las expresiones que se consideraron, no se había hecho referencia a como se trata un predicado para la inferencia de tipos en el cuerpo de este. Un predicado puede estar cuantificado existencialmente, universalmente o ser la combinación de uno o más predicados unidos mediante operadores lógicos ($\wedge, \vee, \Rightarrow, \Leftrightarrow$).

Tomando en cuenta la representación en Prolog de un predicado-Z vista en la sección 2.3.4, podemos describir como se trata dicho predicado para la inferencia de tipos.

La implantación de esta inferencia es realizada por el predicado-Prolog *typechecking*/1 cuya forma es:

typechecking(*Tree_Pred*)

donde *Tree_Pred* es el árbol que representa un predicado.

La forma de este predicado puede ser una de las combinaciones siguientes o combinaciones de ellas:

- $\forall$ *Declaraciones* $\bullet$ *Predicado*
- $\forall$ *Declaraciones* $\mid$ *Predicado*
- $\forall$ *Declaraciones* $\mid$ $Predicado_1 \bullet Predicado_2$
- Las tres representaciones anteriores pero con cuantificación existencial ($\exists$)
- $Predicado_1\ Op \ldots Op\ Predicado_n$ donde $Op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- Una expresión aritmética, de conjuntos, relaciones y funciones.

Así que *typechecking*/1 funciona como sigue:

1. Para el caso de cuantificación universal y existencial, se tiene un conjunto de declaraciones de la forma $d(Nombre, Tipo)$. El alcance de estas declaraciones es sobre el predicado que se analiza, esto es, sólo estan definidas para las expresiones en $Predicado_1$ y $Predicado_2$ (si es que existen uno o ambos); por lo que se guardan en la tabla de símbolos locales. Después de esto, se hace es un llamado recursivo al mismo *typechecking*/1 para que procese los predicados 1 y 2. La presencia de dichas declaraciones se debe a un análisis en la parte sintáctica[1].

2. Por otra parte, si se encuentra que algún predicado es compuesto por operadores lógicos, entonces *typechecking*/1 funciona de manera similar a *gettypetree*/2 en la inferencia de expresiones con operadores, es decir, se analiza el predicado a la derecha del operador lógico y después el predicado a la izquierda. Esto se realiza hasta que el predicado analizado sea una expresión que pueda ser manejada por *gettypetree*/2 o *typeofoper*/4.

[1] Recuérdese que cuando se realiza el análisis sintáctico algunas tareas como la de inferencia de tipos se desarrolla aún cuando el análisis sintáctico no haya terminado completamente.

$$
\begin{array}{l}
typechecking(null) :- !.\\
typechecking(rama(null, null, [oplog, Op])) :-\\
\qquad member(Op, [true, false]).\\
typechecking(rama(Decls, [ST, Pred], [oplog, Quant])) :-\\
\qquad member(Quant, [all, exist, exist\_1]),\\
\qquad local(Decls), !,\\
\qquad typechecking(ST),\\
\qquad typechecking(Pred),\\
\qquad unlocal.\\
typechecking(rama(Left, Right, [oplog, \_])) :-\\
\qquad typechecking(Left), !,\\
\qquad typechecking(Right).\\
typechecking(rama(Left, Right, [oper, Op])) :-\\
\qquad gettypetree(Left, TL),\\
\qquad gettypetree(Right, TR),\\
\qquad typeofoper(Op, TL, TR, Type), !,\\
\qquad \backslash + Type == error\_type.
\end{array}
$$

Figura 3.1: Implantación de *typechecking*/1.

En general, la parte axiomática de un párrafo en Z está definida por medio de un predicado y por lo que se explica arriba, el proceso de inferencia y comprobación de tipos en su nivel más general se da a través de *typechecking*/1 y este predicado es el encargado de conmutar entre *gettypetree*/2 y *typeofoper*/4 cuando sea necesario. Excepto en el caso de la parte declarativa de un párrafo, en la que la rutina encargada de la inferencia de tipos es *gettypetree*/2. La figura 3.1 muestra la implantación de *typechecking*/1.

Cuando se ha detectado algún error, se reporta enviando un mensaje a la pantalla y el proceso de análisis es abortado inmediatamente.

3.3.3 Unificación con parámetros genéricos (PG)

Dentro de las construcciones permitidas por Z, se encuentran aquellas que pueden definirse de forma genérica, esto es, construcciones que no dependen de un determinado tipo, sino que por el contrario, puedan servir para realizar operaciones independientemente del tipo de los conjuntos, secuencias, etc. a los que se apliquen. Ejemplo de tales construcciones es el siguiente [PoB+91]:

$$
\begin{array}{|l}
[X] \\
\hline
head : \operatorname{seq} X \rightarrow X \\
tail : \operatorname{seq} X \rightarrow \operatorname{seq} X \\
\hline
\forall\, s : \operatorname{seq} X \bullet \\
\qquad heads = s1 \land \\
\qquad tails = \lambda\, n : 1 \mathrel{..} \#s - 1 \bullet s(n+1) \\
\hline
\end{array}
$$

Ejemplo En algún momento en la especificación puede usarse la construcción anterior

$$c : CHAR;\ s : \operatorname{seq} CHAR$$
$$c = heads$$

y

$$p : \operatorname{seq} \mathbb{Z};\ vector : \operatorname{seq} \mathbb{Z}$$
$$p = tail\ vector$$

Para poder realizar la comprobación de tipos de las expresiones anteriores es necesario primero determinar el tipo de *head s* y de *tail vector*[2]. La información de que disponemos para inferir tipos en el ejemplo es

1. *tail* es una función que toma una secuencia de tipo X y regresa otra secuencia del mismo tipo.

2. El tipo de *tail* en términos de los **PG**, el cual es:

$$\mathbb{P}(\mathbb{P}(\mathbb{N} \times X) \times \mathbb{P}(\mathbb{N} \times X))$$

[2]Para propósitos de explicación tomaremos como ejemplo a la construcción *tail*.

3. El tipo de *vector* que es:

   $$\mathbb{P}(\mathbb{N} \times \mathbb{Z})$$

4. Finalmente, el tipo de p

   $$\mathbb{P}(\mathbb{N} \times \mathbb{Z})$$

Del tipo de una función $\mathbb{P}(\mathit{tipo}(\mathrm{dom}) \times \mathit{tipo}(\mathrm{ran}))$ podemos saber el tipo del rango y dominio. Así el primer paso es unificar el tipo del dominio (en términos de **PG**) con el tipo del argumento de la función (en términos no genéricos) que en nuestro ejemplo es *vector*.

El proceso de unificación en este caso, se realiza como sigue.

1. Dados

   - El tipo en términos de los **PG**.
   - El tipo no genérico.
   - La lista de **PG**.

   obtener una lista de pares $(\mathit{Gen}, \mathit{Subst})$ donde

   Gen es un **PG**.

   Subst es la expresión por la que es substituido *Gen*.

2. Con la lista de substituciones y el tipo en términos genéricos se construye un árbol binario que lo represente. Se realiza un recorrido de este árbol y conforme se recorre se construye un nuevo árbol con la misma información excepto cuando se encuentre con un **PG**, en este caso, la información en el nuevo árbol es la correspondiente a *Subst* que se encuentra en la lista de substituciones. Finalmente, el árbol construido en el recorrido se traslada a la forma de las expresiones originales[3]. En nuestro ejemplo la lista de substituciones es:

   $$[['X', \mathbb{Z}]]$$

   y el tipo unificado es:

   $$\mathit{power}(\mathit{cross}(\mathbb{N}, \mathbb{Z}))$$

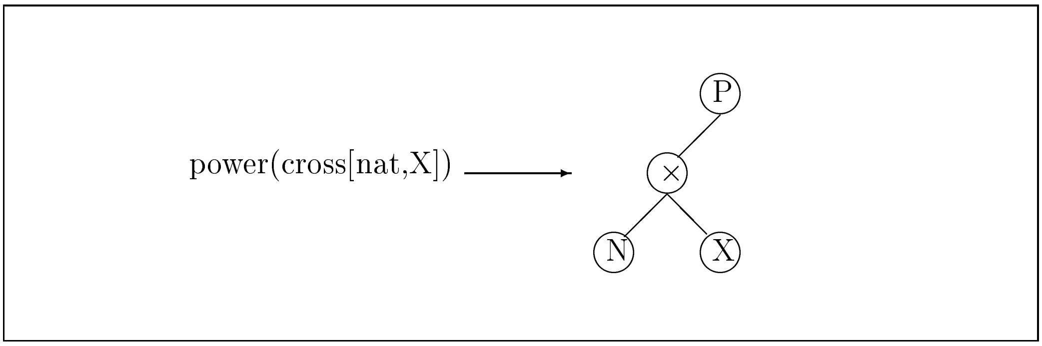

Figura 3.2: Arbol equivalente a una expresión genérica.

La figura 3.2 muestra la representación del tipo genérico por medio de un árbol binario.

El traslado de término hacia un árbol binario es llevado a cabo con ayuda del operador inter-construido de SWI-Prolog = .. [WiJ]. El funcionamiento de tal operador se describe a continuación:

Term = ..List *List* es una lista, donde su primer elemento es el nombre de la cláusula en *Term* y el resto de los elementos de la lista son los argumentos en *Term*. *Term* o *List* puede ser una variable. Si la aridad de *Term* es 0 entonces *List* sólo contiene un elemento. La figura 3.3 muestra el uso de este operador.

La idea de pasar de un término a un árbol es tomar la expresión a convertir, aplicar una primera vez el operador = .. y obtener una primera lista

$$[const, arg1, arg2]$$

donde:

const Es el constructor $\mathbb{P}$ o $\times$ presente en la expresión de tipo.

arg1, arg2 Son los argumentos a los que se aplica *const*.

Si **arg1** y **arg2** tienen *aridad* > 0 entonces se aplica nuevamente el operador, la raíz del árbol es **const** y las ramas el resultado de la aplicar el operador a **arg1** y **arg2**. El proceso termina cuando se encuentra que la aridad de ambos argumentos es 0.

[3] Que corresponden a la forma de términos en Prolog.

```
?- cross(a,b) =.. List
List = [cross, a, b]

?- power(cross(a,b)) =.. List
List = [power, cross(a,b)]

?- Term = [cross,power(a),b]
Term = cross(power(a),b)
```

Figura 3.3: Uso del operador = .. en una sesión de SWI-Prolog.

En el ejemplo anterior los tipos a ser substituidos por los parámetros genéricos fueron dados de forma implícita. También estos tipos pueden darse de forma explícita, en este caso la sintaxis es:

$tail[\mathbb{Z}]\ s$

Como puede verse, después de la construcción genérica se escriben entre paréntesis cuadrados la lista de expresiones a sustituir. Para nuestro ejemplo sólo es una. Aquí el proceso de unificación es realizado como sigue:

1. La información que se tiene es: la lista de **PG**y la lista de entidades por las que son substituidos cada **PG**. Entonces se tiene que construir la lista de substituciones $(Gen, Subst)$, que para el ejemplo nuevamente resulta ser:

   $[['X', num]]$

2. Se procede de forma similar al paso 2 del proceso de unificación de parámetros genéricos descrito en la sección 3.3.3.

3.4 Análisis de párrafos de Z

Ahora se describe la forma en que se lleva a cabo el análisis de tipos para cada párrafo de Z.

3.4.1 Tipos básicos, tipos de datos simples.

Cuando se analiza la definición de tipos básicos en realidad el único trabajo que se realiza es verificar que el identificador que representa a un tipo básico no se haya definido con anterioridad, esta tarea es desarrollada por las rutinas que manejan la tabla de símbolos, descritas en la sección 2.3.3. El mismo trabajo es hecho para el caso del análisis de los tipos de datos simples, la única diferencia es que cada elemento del tipo definido es buscado en la tabla de símbolos por una posible redeclaración.

3.4.2 Definición abreviada

Como se mencionó antes, una definición abreviada introduce una constante global, la regla gramatical asociada a una definición abreviada es:

$$Def{-}Abr \rightarrow \; Defhls == Expresi\acute{o}n$$

donde:

Defhls es el nombre de la constante global.

Expresión es el valor inicial de la constante.

Para poder determinar el tipo de la constante es necesario evaluar el árbol de sintaxis de *Expresión*. Una vez que se ha evaluado y obtenido el tipo de *Expresión*, el nombre de la constante y su tipo se almacenan en la tabla de símbolos.

3.4.3 Definición axiomática

El análisis de la definición axiomática (así como de los restantes párrafos), se divide en dos partes:

1. Análisis de las declaraciones de variables, en este caso hablamos de variables globales. Una vez que se ha identificado una declaración

   $$v_1, \ldots, v_n : Expr$$

   se obtiene el tipo de las n-variables declaradas, posteriormente se actualiza la tabla de símbolos con una entrada $(v_i, Tipo)$ para cada $i, (i \in [1, n])$, esto es porque los símbolos declarados pueden usarse para una nueva declaración. Este proceso continúa hasta que no haya más declaraciones.

2. Análisis de la parte axiomática. Inicia con el árbol de sintaxis abstracta que representa al predicado que constituyen las restricciones a los valores de las variables introducidas en la parte declarativa. El paso siguiente es el llamado

a *typechecking*/1 para que verifique la consistencia del predicado. Si se detecta algún error, entonces se envía un mensaje de fracaso, el mensaje corre por cuenta de *typechecking*/1. De otra forma se envía un mensaje de éxito y se continúa el análisis del siguiente párrafo.

3.4.4 Definición genérica

Las construcciones genéricas definidas en este párrafo se consideran globales, pero sólo hasta el final del análisis es cuando se agregan a la tabla de símbolos. De esta forma, el proceso de análisis toma un grado local, como se ve en la siguiente explicación.

Parámetros genéricos. Debido a la naturaleza genérica de este párrafo, es necesario que se definan los parámetros que deberán instanciarse en algún momento de la especificación con expresiones no genéricas. Así el primer paso es obtener estos parámetros, que son una lista de identificadores que son usados como "tipos" locales a la definición genérica. Entonces, dada la lista de parámetros genéricos

$$X_1, \ldots, X_n$$

tendremos que formar las declaraciones

$$d(X_1, X_1), \ldots, d(X_n, X_n)$$

donde el significado de $d(X_i, X_i)$ es el de declarar a la variable X_i cuyo tipo es X_i. Después de obtener esta lista de declaraciones, lo que sigue es agregarlas como declaraciones locales.

Declaraciones. El siguiente proceso es obtener una lista de declaraciones, donde la estructura de esta lista es $d(v_1, T_1), \ldots, d(v_m, T_m)$; un elemento de la lista debe estar presente por cada variable declarada. El efecto de haber definido los parámetros genéricos como declaraciones locales se ve reflejado cuando se intenta buscar el tipo de una expresión que seguramente hace referencia a combinaciones de los argumentos formales[4]. Esta lista de declaraciones se almacena en la tabla local.

Predicado. Una vez que todos los identificadores encontrados han sido almacenados en la tabla de símbolos, el proceso de análisis es el correspondiente al descrito en la sección 3.3.2 junto con el presentado en la sección 3.3.3.

[4]Recuérdese que el proceso de búsqueda por identificadores previamente definidos inicia en la tabla local y continua en la global.

3.4.5 Esquemas

La unidad de especificación de mayor uso es el esquema, ya que pueden construirse fragmentos de especificación reusables (Modularidad).

Un esquema puede escribirse en dos formas:

1. Por medio de una "caja"

   Nombre

   Parte Declarativa

   Parte Axiomática

2. Mediante el uso de los operadores del cálculo de esquemas para combinar una serie de esquemas previamente definidos.

   $$Nombre \mathrel{\widehat{=}} Expresi\acute{o}n\_c\acute{a}lculo\_esquemas$$

Cajas Cuando se encuentra un esquema definido por medio de una caja el análisis es el siguiente:

Declaraciones A diferencia de la parte declarativa de los dos párrafos anteriores, en ésta pueden declararse referencias a otros esquemas que previamente se han definido con cualquiera de las dos formas de declaración. El alcance de las declaraciones del esquema es local. Nuevamente se obtiene la lista de declaraciones de la forma $d(v_1, T_1), \ldots, d(v_n, T_n)$ donde posiblemente algún v_i es una referencia a un esquema, en cuyo caso el elemento de la lista tiene la forma

$$d(nombre, schema(Decoraci\acute{o}n))$$

donde **Decoración** es:

- Una serie de apóstrofos que indican estados posteriores a una operación.
- Δ que indica un cambio de estado $\Delta S == S;\ S'$.
- Ξ que indica que el esquema no causa ningún cambio de estado.

Puesto que estamos tratando con un ambiente local, el paso a seguir es el de actualizar la tabla local con esta lista de declaraciones, pero antes de este paso es necesario realizar una expansión de las referencias a esquemas para que solamente sean variables las que se almacenen en la tabla local. El proceso de expansión se realiza hasta que no haya más referencias a esquemas. Cuando se expande un esquema se expande también su declaración, con esto se asegura que se expande cualquier esquema referido. Las ocurrencias recursivas de un esquema no son permitidas ya que Z requiere que antes sea declarado. Para los casos de ΔS y ΞS, primero se expande a S; S' y después cada una de estas referencias es expandida como se explicó antes. En el caso de encontrarse alguna referencia a un esquema desconocido se reporta un mensaje error y el análisis termina inmediatamente.

Predicado Para el caso del predicado se sigue el esquema visto en la sección 3.3.2.

En el caso de que no exista ningún error, el esquema y la información sobre sus partes declarativa y axiomática se agrega a la tabla de símbolos global.

Cálculo de esquemas Cuando un esquema se define combinando otros esquemas, por medio del cálculo de esquemas el proceso a seguir es colectar las definiciones de los esquemas componentes de la expresión, realizar la concatenación de estas y revisar que no exista un conflicto entre variables, esto es, dos variables con el mismo nombre declaradas en esquemas diferentes deben pertenecer al mismo tipo, de no ser así, un mensaje de error se envía al usuario. Los predicados de cada esquema son combinados por el operador łógico equivalente al cálculo de esquemas. Cuando de trate del operador de ocultamiento $\setminus$ se verifica que las variables incluídas en la tupla a ocultar estén debidamente declaradas por el esquema al que se le aplica la operación. En el caso de la composición se verifica que el tipo de cada variable de estado primada en el esquema a la izquierda de $\fatsemi$ coincida con el tipo de su correspondiente variable no primada en el esquema a la derecha del operador. Una vez que se ha obtenido el nuevo esquema, la tabla de símbolos se actualiza con la información sobre el nuevo esquema. Además, este nuevo esquema es escrito en un archivo de salida[5] para que el usuario pueda revisarlo posteriormente.

La parte central en la implantación del proceso de comprobación de tipos es la implantación del conjunto de reglas de asignación de tipos, después, la selección de la regla que será aplicada, ya sea para la obtención del tipo de una expresión o para determinar la consistencia de la misma (con respecto a los tipos de las variables y

[5]El formato de este archivo de salida es LaTeX, el nombre del archivo es el mismo que el de entrada con extensión PRE.

operadores que intervienen en la misma), se realiza por medio del mecanismo de backtracking de Prolog. Cuando se obtiene el tipo de una variable, este tipo es almacenado en la tabla de símbolos junto con la variable en cuestión en otro caso el árbol de sintaxis abstracta que la representa la expresión que se verifica, es la entidad almacenada en la tabla de símbolos, posiblemente para su uso posterior con objeto de realizar el calculo y simplificación de precondiciones.

Capítulo 4

Precondiciones

El camino hacia el diseño de un sistema seguro debe garantizar que cuando ocurra un cambio de estado en el sistema, el estado final de esa transición debe estar definido. En otras palabras, deberemos de conocer aquellas condiciones que deberán de cumplirse para que se lleve a cabo una transición de estado. Conociendo estas condiciones podremos saber si una operación puede realizarse exitosamente o en su defecto cuando no puede ser aplicada ya que conducirá al sistema a un estado indefinido.

Para Z, tanto el conjunto de estados admisibles así como las operaciones que, en principio, generan un cambio de estado en el sistema que se especifica, deberán modelarse por medio de esquemas, de tal suerte que dentro del cálculo de esquemas, existe el operador *pre*, que sólo puede aplicarse a operaciones, y el cual es usado como un medio para calcular estas condiciones.

4.1 Definiciones

Definición Las precondiciones de una operación son el conjunto de condiciones bajo las cuales una operación es **aplicable**.

Definición Se dice que una operación es **aplicable** para aquellas combinaciones de estados anteriores y variables de entrada tal que existe un estado posterior y un conjunto de variables de salida que satisfacen las relaciones definidas por la operación misma.

Si los estados del sistema son modelados por medio del esquema *Estado* y *Salida* es el conjunto de variables de salida asociadas con una operación *Oper* entonces usamos

$$\text{pre } Oper$$

para denotar las precondiciones de Oper, y además las precondiciones de *Oper* quedan definidas por la expresión

$$\text{pre } Oper = \exists\, Estado';\ Salida \bullet Oper$$

4.2 Cálculo y simplificación

El proceso de calcular las precondiciones es realizado de acuerdo a los siguientes pasos, los cuales se siguen de la definición de aplicabilidad:

Supongamos que se desea calcular las precondiciones de

Oper
Declaraciones
Predicado

1. Dividimos *Declaraciones* en dos partes

   Ant_Ent Una lista de estados anteriores y variables de entrada

   Post_Sal Lista de estados posteriores y variables de salida[1]

2. Las precondiciones de *Oper* están dadas por

Pre
Ant_Ent
$\exists\, Post\_Sal \bullet$ $\quad Predicado$

Ejemplo. Para ver como funciona este algoritmo para calcular las precondiciones, consideremos los siguientes esquemas

S
$a : \mathbb{N}$ $b : \mathbb{N}$
$a \neq b$

[1] Los estados posteriores son aquellos que se encuentran decorados mediante un apóstrofe, las variables de salida son las decoradas con el símbolo !.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{T} \\
S \\
c : \mathbb{N} \\
\hline
b \neq c \\
\hline
\end{array}$$

calculemos las precondiciones de *Inc*

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{Inc} \\
\Delta T \\
in? : \mathbb{N} \\
out! : \mathbb{N} \\
\hline
a' = a + in? \\
b' = b \\
c' = c \\
out! = c \\
\hline
\end{array}$$

Aplicando el primer paso tendremos que dividir la parte declarativa en dos listas[2]:

$$Ant\_Ent = [\, T, in? : \mathbb{N}]$$

$$Post\_Sal = [\, T', out! : \mathbb{N}]$$

Así las precondiciones de *Inc* están dadas por

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{PreInc} \\
T \\
in? : \mathbb{N} \\
\hline
\exists\, out! : \mathbb{N};\;\; T' \bullet \\
\qquad a' = a + in? \\
\qquad b' = b \\
\qquad c' = c \\
\qquad out! = c \\
\hline
\end{array}$$

Como puede verse en el ejemplo, el predicado del esquema precondición resultante puede ser muy complicado. Así que el siguiente paso es el de simplificar este predicado.

[2]Como se vió en el Capítulo 2, $\Delta S == S;\ S'$.

La simplificación del predicado resultante está basada en la aplicación de la regla de **un punto** [PoB+91], la cual escribimos a continuación.

$$\frac{\exists\, x : T \bullet p \wedge t = x}{t \in T \wedge p[t/x]} \quad [\ \textsf{Regla de un punto}\ ]$$

La línea horizontal indica que si encontramos una expresión de la forma $\exists\, x : T \bullet p \wedge t = x$ se puede reescribir como $t \in T \wedge p[t/x]$.

Para poder aplicar esta regla a nuestro predicado tendremos que seguir los siguientes pasos

1. Expandir cualquier esquema en la lista *Post_Sal*
2. Si en el predicado encontramos una ecuación que involucre a algún componente declarado en *Post_Sal*, entonces utilizamos la regla de un punto para eliminar ese componente. Este paso es repetido tantas veces como sea posible.
3. Ahora se revisan las expresiones dentro de las declaraciones para eliminar posibles redundancias, es decir, expresiones que se tienen como premisas y que aparecen en el predicado. Por ejemplo si en el predicado aparece la expresión $v \in G$ y en las declaraciones se tiene $v : G$ entonces la primera expresión puede eliminarse del predicado.
4. Si $Post\_Sal_1$ y $Predicado_1$ son el resultado del paso anterior entonces las precondiciones están dadas ahora por

$$Ant\_Ent$$
$$\exists\, Post\_Sal_1 \bullet$$
$$\quad Predicado_1$$

Ejemplo El predicado del cálculo anterior es

$$\exists\, out! : \mathbb{N};\ T' \bullet$$
$$\quad a' = a + in?$$
$$\quad b' = b$$
$$\quad c' = c$$

$out! = c$

Siguiendo con el procedimiento antes descrito, el primer paso nos indica que debemos expandir todos los esquemas en $Post\_Sal$

$$
\begin{aligned}
&\exists\, out! : \mathbb{N};\ S';\ c' : \mathbb{N} \mid b' = c' \bullet \\
&\quad a' = a + in? \\
&\quad b' = b \\
&\quad c' = c \\
&\quad out! = c
\end{aligned}
$$

Vemos que aún existe otro esquema, así que tenemos que expandirlo nuevamente

$$
\begin{aligned}
&\exists\, out! : \mathbb{N};\ a', b' : \mathbb{N};\ c' : \mathbb{N} \mid b' \neq c' \wedge a' \neq b' \bullet \\
&\quad a' = a + in? \\
&\quad b' = b \\
&\quad c' = c \\
&\quad out! = c
\end{aligned}
$$

Como ya no hay más esquemas a expandir, procedemos al siguiente paso. La regla de un punto puede aplicarse cuatro veces para producir el predicado

$$
\begin{aligned}
&a + in? \neq b \\
&b \neq c \\
&a + in? \in \mathbb{N} \\
&b \in \mathbb{N} \\
&c \in \mathbb{N}
\end{aligned}
$$

Finalmente, el último paso es revisar las declaraciones, para eliminar información redundante. Así podemos ver que T asegura que $b \in \mathbb{N}$, $c \in \mathbb{N}$ y $b \neq c$; después como $a \in \mathbb{N}$ y $in? \in \mathbb{N}$ entonces $a + in? \in \mathbb{N}$. De esta forma con el predicado simplificado, el esquema pre Inc es:

pre Inc
$in? : \mathbb{N}$ T
$a + in? \neq b$

4.3 Implantación

La importancia de conocer las precondiciones de una operación nos lleva a desarrollar un sistema que de forma automática realice el cálculo y posteriormente la simplificación de las precondiciones. El proceso de simplificación es aplicado al predicado del esquema que resulta del cálculo de las precondiciones de alguna expresión del cálculo de esquemas. El objetivo de este proceso es el de llevar dicho predicado a una forma equivalente en la que las condiciones redundantes que pudiesen estar presentes sean eliminadas. Cuando hablamos de la simplificación que es realizada por una persona (en papel) estamos involucrando a la heurística como un proceso por medio del cual la mayoría de las veces se logra llegar a una forma simplificada. Nuestro caso es el de un proceso automático de simpificación que principalmente involucra la aplicación repetida de reglas de reescritura con las que se eliminan tales condiciones. Pero como no contamos con la heurística entonces en general no podremos llegar a la forma que se obtiene cuando directamente interviene una persona.

Sin embargo, dado que la implantación de los procedimientos descritos en la sección anterior nos reducen el trabajo de cálculo y el de simplificación (sobre todo cuando tratamos con esquemas que tienen varios niveles de inclusión), entonces eso nos motiva a la implantación de tales tareas.

4.3.1 Cálculo de precondiciones

Dentro de las tareas que deben realizarse para efectuar el cálculo de precondiciones están: la separación de la parte declarativa del esquema, en dos grupos, uno que contiene referencias a los estados anteriores, variables de entrada y variables locales, y otro grupo que contiene las referencias a los estados posteriores y variables de salida.

Recordando la implantación del conjunto de declaraciones de un esquema, se tiene una estructura de lista en Prolog

$$[d(Nombre_1, Tipo_1), \dots, d(Nombre_n, Tipo_n)]$$

donde:

$Nombre_i$ es el nombre de la variable o de un esquema. Este nombre entra dentro de la categoría de átomo de Prolog. Si el nombre termina con ? entonces es una variable de entrada y si termina con ! entonces estamos hablando de una variable de salida

$Tipo_i$ Es el tipo de la variable y cuando se trata de un esquema éste tiene la forma:

$$schema(Decoración)$$

$$
\begin{array}{l}
is\_out(Name) : - \\
\quad atom\_chars(Name, List), \\
\quad member(0'!, List).
\end{array}
$$

Figura 4.1: Verifica si una variable es de salida.

donde *Decoración* es una secuencia de letras p y cada p equivale a un apóstrofe[3]. También puede ser Δ o Ξ que equivalen a $S;\ S'$ (suponiendo que $Nombre_i$ es S).

El problema de determinar si una variable es de entrada o salida radica en observar primero que la referencia no sea un esquema y después que el nombre de la variable termine con una decoración del tipo ? o !. La figura 4.1 muestra el código en Prolog del predicado que verifica si una variable es de salida.

Para localizar las referencias a estados anteriores, el primer paso es verificar que el tipo de la declaración sea *schema*(*Decoración*) esto garantiza que se trata de un esquema. El siguiente paso es llevar a la forma $S;\ S'$, las ocurrencias Δ y Ξ de *Decoración* y si no está decorado el esquema entonces expandirlo y aplicar nuevamente este proceso, la implantación se muestra en la figura 4.2. Después, lo único que resta es seleccionar entradas, salidas, estados anteriores y posteriores, la figura 4.3 muestra tal implantación.

Hasta este momento hemos calculado las precondiciones de una operación.

4.3.2 Simplificación

De acuerdo con el procedimiento de simplificación expuesto anteriormente, el primer paso en la simplificación del predicado de una precondición es expandir los esquemas en el grupo *Post_Sal*, de tal forma que sólo queden referencias a variables. Entonces, dada la lista de declaraciones donde cada elemento de la lista tiene la forma:

$d(Nombre, Tipo)$

- Si *Nombre* es una variable entonces se continúa con el siguiente elemento.

[3] En Z el símbolo ' es usado para decoraciones. Un esquema decorado con un ' es considerado como una referencia a un estado posterior.

```
expande_delta([], []).
expande_delta([d(Name, schema(Dec)) | Rd], [d(Name, schema([])),
        d(Name, schema(p)) | OD]) :-
    member(Dec, [delta, xi]),
    expand_delta(Rd, OD).
expand_delta([d(Name, schema([])) | Rd], ExpandedD) :-
    getglobalsimb(Name, [Dec, _]),
    expand_delta(Dec, ExDec),
    expand_delta(Rd, ExRd),
    append(ExDec, ExRd, ExpandedD).
expand_delta([D | R], [D | Od]) :-
    expand_delta(R, Od).
```

Figura 4.2: Expandiendo a la forma $S;\ S'$.

```
after_outs([], []).
after_outs([d(Name, schema(D)) | Rem],
        [d(Name, schema(D)) | Rv]) :-
    \+ D == [],
    after_outs(Rem, Rv).
after_outs([d(Name, T) | Rem], [d(Name, T) | Rv]) :-
    is_out(Name),
    after_outs(Rem, Rv).
after_outs([_ | R], AO) :-
    after_outs(R, AO).
```

Figura 4.3: Dividiendo declaraciones.

- Si $Tipo = schema(p)$ entonces, de la tabla de símbolos se obtienen las declaraciones y el predicado del esquema de nombre *Nombre* y

  1. Todas las variables de las declaraciones y predicado se decoran con p (cumpliendo con la definición de S').
  2. Si se encuentra una referencia a otro esquema, entonces se expande primero y después sus variables se decoran con p.
  3. Se concatenan las variables de los puntos anteriores para formar una lista de variables.

  Se analiza la siguiente declaración.

- El proceso termina cuando se ha llegado a la lista vacía.

El siguiente paso es buscar ecuaciones en el predicado donde intervienen variables de la lista previamente obtenida. Este paso es la aplicación de la regla de un punto. Para poder llevarlo a cabo:

1. Se toma la primera variable en la lista, digamos V, como lo que deseamos es encontrar una ecuación de la forma $V = Expr$ y dado que un predicado es representado como:

   $$rama(Izq, Der, [oplog, Op])$$

   estamos interesados en buscar ecuaciones como:

   $$rama(V, Expr, [oper, =])$$

   en *Izq* y *Der*.

2. Si no existe tal expresión entonces se continúa con la siguiente variable. De otra forma se elimina la expresión $V = Expr$ del predicado.

3. Ahora que se tiene la expresión *Expr* entonces se buscan en *Izq* y *Der* todas las ocurrencias de V y se sustituyen por *Expr*, esto es realizado de la siguiente forma:

   - Se busca la expresión:

     $$rama(V, OtraExpr, [oper, OtroOper])$$

     se asume que $OtroOper \neq' =' $ porque esas ecuaciones ya han sido eliminadas.

- Entonces se substituye por:

$$rama(Expr, OtraExpr, [oper, OtroOper])$$

El proceso de búsqueda y sustitución es realizado mediante un recorrido del árbol que representa el predicado, empezando por la raíz, el nodo izquierdo y finalmente el nodo derecho.

4. Cuando se han substituído todas las ocurrencias de V, entonces la expresión:

$$Expr \in Tipo$$

debe ser agregada al predicado mediante un conectivo $\wedge$ y la declaración:

$$d(Nombre, Tipo)$$

puede ser eliminada de la lista de declaraciones.

5. Este proceso es seguido hasta analizar la última declaración en la lista.

La figura 4.4 muestra el predicado que aplica la regla de un punto.

Finalmente, el último paso consiste en eliminar expresiones consideradas como hipótesis y que se encuentran en la parte declarativa. Se construye una lista de tales expresiones, las declaraciones de la forma:

$$x : T$$

son convertidas a:

$$rama(rama(null, null, [opnd, x]), rama(null, null, [type, T]), [oper, \in])$$

Esto implica que los esquemas existentes en las declaraciones deben ser expandidos hasta obtener sólo variables.

Después, cada elemento en la lista es buscado en el predicado de la precondición. Si se localiza entonces este es eliminado, de otra forma se continúa con el siguiente elemento en la lista. La figura 4.5 muestra el predicado de búsqueda de expresiones.

Como último paso, dentro de las variables que formaban el grupo $Post\_Sal$ pudieron quedar algunas que en ningún lugar en el predicado se hace referencia a ellas entonces pueden ser eliminadas de la lista $Post\_Sal$, así en la figura 4.6 se muestra el código que localiza la ocurrencia de una variable en el predicado y en la figura 4.7 se encuentra la implantación del cálculo y simplificación de precondiciones.

$$
\begin{array}{l}
onepointrule([], Pred, [], Pred).\\
onepointrule([d(N, T) \mid DN], Pred, NewD, NewP) :-\\
\quad search\_asign(rama(null, null, [opnd, N]), Pred, Subst),\\
\quad elim\_asign(rama(rama(null, null, [opnd, N]), Subst, [oper, =]),\\
\qquad\qquad\qquad Pred, NP1),\\
\quad elim\_asign(rama(Subst, rama(null, null, [opnd, N]), [oper, =]),\\
\qquad\qquad\qquad NP1, NP),\\
\quad substituye(rama(null, null, [opnd, N]), Subst, NP, NNP),\\
\quad onepointrule(DN, NNP, NewD, Ap),\\
\quad (\backslash + \ is\_typesimple(T)\\
\quad -> agregar(d(N, T), Subst, Ap, NewP)\\
\quad ;\ NewP = Ap\\
\quad).\\
onepointrule([D \mid DN], Pred, [D \mid NewD], NewP) :-\\
\quad onepointrule(DN, Pred, NewD, NewP).
\end{array}
$$

Figura 4.4: Implantación de la Regla de un Punto.

$$
\begin{array}{l}
search\_asign(A, rama(A, B, [oper, =]), B).\\
search\_asign(A, rama(B, A, [oper, =]), B).\\
search\_asign(A, rama(C, \_D, [oplog, \_]), B) :-\\
\quad search\_asign(A, C, B).\\
search\_asign(A, rama(\_C, D, [oplog, \_]), B) :-\\
\quad search\_asign(A, D, B).
\end{array}
$$

Figura 4.5: Búsqueda de una expresión.

```
exists_refer(d(Name,_), rama(null, null, [opnd, Name])).
exists_refer(Decl, rama(A, B, [oper, _Oper])) :-
    exists_refer(Decl, A);
    exists_refer(Decl, B).
exists_refer(Decl, rama(A, B, [oplog, _Oper])) :-
    exists_refer(Decl, A);
    exists_refer(Decl, B).
```

Figura 4.6: Búsqueda de una variable.

```
precondition(Decl, Pred,
        [NDecls, rama(ExsDecPart, [null, NewP], [oplog, exist])]) :-
    expand_delta(Decl, EDecl),
    after_outs(EDecl, Aftouts),
    after_predicate(Aftouts, Aftpred),
    elim_decls(EDecl, Aftouts, NDecls),
    expand_schema(Aftouts, ExpAftouts), !,
    onepointrule(ExpAftouts,
        rama(Aftpred, Pred, [oplog, land]), Ne     Dcl, NTP),
    getdeclsandpreds(NDecls, LDecls, LPred),
    getexprindecls(LDecls, ListD),
    getexprinpreds(LPred, ListP),
    append(ListD, ListP, DecPred),
    elim_hipot(DecPred, NTP, NewP),
    del_refs_notused(NewDcl, NewP, ExsDecPart).
```

Figura 4.7: Cálculo y simplificación de precondiciones.

El diseño de un sistema robusto requiere del conocimiento de las condiciones que deben cumplirse antes que una operación pueda ser aplicada; a partir de estas precondiciones se pueden diseñar operaciones que contemplen los casos en los que una operación no puede aplicarse. La implantación de la regla de un punto realiza el trabajo de simplificación. La información del esquema que contiene las precondiciones de una operación se almacena en la tabla de símbolos como un esquema más en la especificación y además en un archivo para un análisis posterior.

Capítulo 5

Pruebas y resultados

Los ejemplos que se han desarrollado para propósitos de prueba del prototipo se han seleccionado, de tal forma que cubran con la mayor parte de la notación soportada. El primer ejemplo es el que se ha descrito en la sección 1.6.

2. La siguiente especificación ha sido tomada de [PoB+91]. Se requiere un sistema que ayude a la adquisición de vocabulario para un estudiante que esté interesado en aprender un segundo lenguaje. El sistema registra pares de palabras; el primer elemento de este par es una palabra del lenguaje "nativo" y el segundo una palabra del lenguaje "extranjero" que se pretende aprender. Las palabras que se agreguen al vocabulario deberán satisfacer reglas ortográficas de ambos lenguajes, el propósito de las reglas es eliminar palabras que posiblemente no pertenezcan a alguno de los dos lenguajes.

Los siguientes conjuntos representan todas las posibles secuencias de caracteres sobre los alfabetos de los lenguajes "nativo" y "extranjero" respectivamente.

$$[\mathit{Native}, \mathit{Foreign}]$$

Las reglas ortográficas son caracterizadas por subconjuntos de "Native" y "Foreign".

$$\begin{array}{|l} \mathit{OrthoNative} : \mathbb{P}\, \mathit{Native} \\ \mathit{OrthoForeign} : \mathbb{P}\, \mathit{Foreign} \end{array}$$

Para cualquier par de lenguajes, las definiciones de esos subconjuntos deben ser proporcionadas en alguna etapa durante el proceso de implantación. Se requiere que en cualquier momento el vocabulario contenga sólo palabras que satisfacen las reglas ortográficas, es decir, cada palabra nativa o extranjera en el vocabulario debe ser un elemento de los subconjuntos apropiados.

El siguiente conjunto consta de mensajes usados para reportarle al usuario el resultado de las operaciones que son efectuadas.

$$
\begin{aligned}
\mathit{Message} ::= {} & \mathit{Ok} \mid \mathit{AlreadyKnownPair} \mid \mathit{NewPairEntered} \\
\mid {} & \mathit{ErrorInForeignWord} \mid \mathit{ErrorInNativeWord} \mid \mathit{ErrorInBothWords} \\
\mid {} & \mathit{UnknownNativeWord} \mid \mathit{UnknownForeignWord} \\
\mid {} & \mathit{VocablsEmpty} \mid \mathit{NoCorrectResponses}
\end{aligned}
$$

Se necesita decir el significado de un conjunto que tenga un orden alfabéticon total. Un orden alfabético debe tener las siguientes propiedades:

- Si un elemento x está antes en el orden que un elemento y y y es un elemento anterior a z entonces x es anterior a z.

- Para dos elementos distintos x y y, x es anterior a y en el orden o y es anterior a x, pero no ambos.

La siguiente construcción especifica tales características.

$$
\begin{array}{|l}
[X] \\
\hline
\mathit{alphorder} : \mathbb{P}(X \rightarrow X) \\
\hline
\mathit{alphorder} = \{\_\mathit{Rel}\_ : X \leftrightarrow X \mid \\
\qquad \forall\, x, y, z : X \bullet \\
\qquad (x \mathrel{\mathit{Rel}} y \wedge y \mathrel{\mathit{Rel}} z \Rightarrow x \mathrel{\mathit{Rel}} z) \wedge \\
\qquad (x \mathrel{\mathit{Rel}} y \wedge y \mathrel{\mathit{Rel}} x \Rightarrow x = y) \wedge \\
\qquad (x \neq y \Rightarrow x \mathrel{\mathit{Rel}} y \vee y \mathrel{\mathit{Rel}} x) \\
\quad \} \\
\hline
\end{array}
$$

La siguiente función toma un conjunto finito y lo convierte a una secuencia cuya longitud es el número de elementos de tal conjunto, y donde sus elementos están ordenados de acuerdo a un orden alfabético dado.

$$
\begin{array}{|l}
[X] \\
\hline
putinorder : (\mathbb{F}\, X \times alphorder[X]) \rightarrow \mathrm{seq}\, X \\
\hline
putinorder = \{nset : \mathbb{F}\, X;\ order : alphorder[X];\ nseq : \mathrm{seq}\, X \mid \\
\quad \#nseq = \#nset \land nset = \mathrm{ran}\, nseq \land \\
\quad (\forall\, i, j : 1 \mathinner{.\,.} \#nset \mid i < j \bullet (nseq\ i, nseq\ j) \in \\
\quad order) \bullet (nset, order) \mapsto nseq \\
\quad \} \\
\hline
\end{array}
$$

Ahora se declara el órden alfabético correspondiente a cada lenguaje. Además, se definen dos funciones (*SetInNativeWordOrder*, *SetInForeignWordOrder*) que convierten conjuntos de palabras de ambos lenguajes a sus correspondientes secuencias ordenadas alfabéticamente.

$$
\begin{array}{|l}
\_NativeWordOrder\_ : alphorder[Native]; \\
\_ForeignWordOrder\_ : alphorder[Foreign]; \\
SetInNativeWordOrder : \mathbb{F}\, Native \rightarrow \mathrm{seq}\, Native; \\
SetInForeignWordOrder : \mathbb{F}\, Foreign \rightarrow \mathrm{seq}\, Foreign \\
\hline
SetInNativeWordOrder = \\
(\lambda\, nset : \mathbb{F}\, Native \bullet putinorder(nset, NativeWordOrder)); \\
SetInForeignWordOrder = \\
(\lambda\, nset : \mathbb{F}\, Foreign \bullet putinorder(nset, ForeignWordOrder))
\end{array}
$$

Se necesita definir un orden alfabético sobre estas funciones, de tal forma que respete el orden de las palabras en ambos lenguajes cuando como usando como criterio el orden sobre el primer elemento de un pareja: $(nativo, extranjero)$.

$$
\begin{array}{|l}
\_NativeDictOrder\_ : alphorder[Native \times \mathrm{seq}\, Foreign]; \\
\_ForeignDictOrder\_ : alphorder[Foreign \times \mathrm{seq}\, Native] \\
\hline
\forall\, n1, n2 : Native;\ fseq1, fseq2 : \mathrm{seq}\, Foreign \bullet \\
\quad n1 \neq n2 \Rightarrow \\
\qquad (n1, fseq1)\ NativeDictOrder\ (n2, fseq2) \Leftrightarrow \\
\qquad n1\ NativeWordOrder\ n2; \\
\forall\, f1, f2 : Foreign;\ nseq1, nseq2 : \mathrm{seq}\, Native \bullet \\
\quad f1 \neq f2 \Rightarrow \\
\qquad (f1, nseq1)\ ForeignDictOrder\ (f2, nseq2) \Leftrightarrow \\
\qquad f1\ ForeignWordOrder\ f2
\end{array}
$$

Y ahora definimos las funciones que ponen las listas de palabras en los ordenes correctos.

$$
\begin{array}{|l}
SetInNativeDictOrder : \\
\quad (Native \rightarrowtail\!\!\!\!\!\!+\; \operatorname{seq} Foreign) \rightarrow \operatorname{seq}(Native \times \operatorname{seq} Foreign); \\
SetInForeignDictOrder : \\
\quad (Foreign \rightarrowtail\!\!\!\!\!\!+\; \operatorname{seq} Native) \rightarrow \operatorname{seq}(Foreign \times \operatorname{seq} Native) \\
\hline
SetInNativeDictOrder = \\
\quad (\lambda\, tset : (Native \rightarrowtail\!\!\!\!\!\!+\; \operatorname{seq} Foreign) \bullet \\
\qquad putinorder(tset, NativeDictOrder)); \\
SetInForeignDictOrder = \\
\quad (\lambda\, tset : (Foreign \rightarrowtail\!\!\!\!\!\!+\; \operatorname{seq} Native) \bullet \\
\qquad putinorder(tset, ForeignDictOrder))
\end{array}
$$

En el siguiente esquema se define el vocabulario como una relación entre pares de palabras correctamente formadas.

$$
\begin{array}{|l}
\hline
WellFormedVocab \\
\hline
Vocab : OrthoNative \leftrightarrow OrthoForeign; \\
NativeWordsKnown : \mathbb{F}\, OrthoNative; \\
ForeignWordsKnown : \mathbb{F}\, OrthoForeign \\
\hline
NativeWordsKnown = \operatorname{dom} Vocab; \\
ForeignWordsKnown = \operatorname{ran} Vocab \\
\hline
\end{array}
$$

El espacio de estados se describe con:

$$
\begin{array}{|l}
\hline
WordForWord \\
\hline
WellFormedVocab; \\
\hline
\end{array}
$$

Ahora el estado inicial:

$$
\begin{array}{|l}
\hline
InitWord_F or_W ord \\
\hline
WordForWord' \\
\hline
Vocab' = \emptyset; \\
\hline
\end{array}
$$

La siguiente operación agrega un par de palabras al vocabulario.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{EnterPair}} \\
\Delta \mathit{WellFormedVocab}; \\
n? : \mathit{OrthoNative}; \\
f? : \mathit{OrthoForeign} \\
\hline
\mathit{Vocab}' = \mathit{Vocab} \cup \{n? \mapsto f?\} \\
\hline
\end{array}
$$

Cuando se intenta agregar un par que ya está presente en el vocabulario debe reportarse al usuario tal evento, así como cuando un nuevo par ha sido agregado.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{ReportIfAlreadyKnown}} \\
\mathit{Vocab} : \mathit{OrthoNative} \leftrightarrow \mathit{OrthoForeign}; \\
n? : \mathit{OrthoNative}; \\
f? : \mathit{OrthoForeign}; \\
\mathit{report}! : \mathit{Message} \\
\hline
n? \mapsto f? \in \mathit{Vocab} \Rightarrow \mathit{report}! = \mathit{AlreadyKnownPair}; \\
n? \mapsto f? \notin \mathit{Vocab} \Rightarrow \mathit{report}! = \mathit{NewPairEntered} \\
\hline
\end{array}
$$

Usando el cálculo de esquemas podemos entonces definir una operación robusta que nos permita agregar el par de palabras $(\mathit{native}, \mathit{foreign})$

$$\mathit{AddPair} \mathrel{\widehat=} \mathit{EnterPair} \land \mathit{ReportIfAlreadyKnown}$$

Las precondiciones para la operación anterior son calculadas mediante la siguiente expresión.

$$\mathit{PAddPair} \mathrel{\widehat=} \mathrm{pre}\ \mathit{AddPair}$$

A continuación se muestran los resultados obtenidos por el prototipo con respecto a la expansión de los esquemas que se expresaron como una operación del cálculo de esquemas y los pasos seguidos para el cálculo de precondiciones.

Después de expandir la expresión

$$\mathit{AddPair} \mathrel{\widehat=} \mathit{EnterPair} \land \mathit{ReportIfAlreadyKnown}$$

el esquema resultante se presenta a continuación.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddPair}} \\
\Delta \textit{WellFormedVocab}; \\
\textit{Vocab} : \textit{power}(\textit{cross}(\textit{Native}, \textit{Foreign})); \\
n? : \textit{Native}; \\
f? : \textit{Foreign}; \\
\textit{report}! : \textit{Message} \\
\hline
\textit{Vocabp} = \textit{Vocab} \cup \{n? \mapsto f?\} \wedge \\
n? \mapsto f? \in \textit{Vocab} \Rightarrow \textit{report}! = \textit{AlreadyKnownPair} \wedge \\
n? \mapsto f? \notin \textit{Vocab} \Rightarrow \textit{report}! = \textit{NewPairEntered} \\
\hline
\end{array}
$$

El cálculo descrito por

$$\textit{PAddPair} \mathrel{\widehat=} \text{pre } \textit{AddPair}$$

es realizado por el prototipo mediante los siguientes pasos:

- Se inicia con el esquema *AddPair* mostrado anteriormente.
- La expansión de las expresiones ΔS y ΞS es el siguiente paso:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Expand } \Delta} \\
\textit{WellFormedVocab}; \\
\textit{WellFormedVocab}'; \\
\textit{RecordOfProgress}; \\
\textit{RecordOfProgress}'; \\
\textit{Vocab} : \textit{power}(\textit{cross}(\textit{Native}, \textit{Foreign})); \\
n? : \textit{Native}; \\
f? : \textit{Foreign}; \\
\textit{report}! : \textit{Message} \\
\hline
\textit{Vocabp} = \textit{Vocab} \cup \{n? \mapsto f?\} \wedge \\
n? \mapsto f? \in \textit{Vocab} \Rightarrow \textit{report}! = \textit{AlreadyKnownPair} \wedge \\
n? \mapsto f? \notin \textit{Vocab} \Rightarrow \textit{report}! = \textit{NewPairEntered} \\
\hline
\end{array}
$$

- Las precondiciones están dadas por la expresión:

$$\exists \textit{State}';\ \textit{Output} \bullet \textit{Oper}$$

en nuestro caso, se extraen de la parte declarativa de *Expand* Δ las referencias a estados posteriores (referencias primadas) y las variables de salida; después se cuantifican existencialmente y pasan a formar parte del predicado.

$$
\begin{array}{|l}
\hline
\textit{Pre} \\
\textit{WellFormedVocab}; \\
\textit{RecordOfProgress}; \\
\textit{Vocab} : \textit{power}(\textit{cross}(\textit{Native}, \textit{Foreign})); \\
n? : \textit{Native}; \\
f? : \textit{Foreign} \\
\hline
\exists\, \textit{WellFormedVocab};\ \textit{RecordOfProgress}';\ \textit{report}! : \textit{Message} \bullet \\
\qquad \textit{Vocabp} = \textit{Vocab} \cup \{n? \mapsto f?\} \wedge \\
\qquad n? \mapsto f? \in \textit{Vocab} \Rightarrow \textit{report}! = \textit{AlreadyKnownPair} \wedge \\
\qquad n? \mapsto f? \notin \textit{Vocab} \Rightarrow \textit{report}! = \textit{NewPairEntered} \\
\hline
\end{array}
$$

- Ahora nos concentraremos en la parte axiomática del último esquema, expandiremos las referencias a esquemas hasta obtener sólamente variables. La notación para esquemas es usada por el prototipo para denotar los pasos que se siguen en la simplificación del predicado que resultó del cálculo de las precondiciones, de esta forma la expresión:

  $$\exists\, \textit{Decls} \bullet \textit{Pred}$$

  es representada por el prototipo como:

$$
\begin{array}{|l}
\hline
\textit{Step}(n) \\
\textit{Decls} \\
\hline
\textit{Pred} \\
\hline
\end{array}
$$

  donde:

  Step(n) - es una etiqueta donde n indica el número de veces que se ha aplicado la regla de un punto al predicado.

  Decls - son las declaraciones que inicialmente están cuantificadas existencialmente, en cada paso se van modificando.

  Pred - es el predicado que es objeto de simplificación.

Siguiendo con el ejemplo, el predicado que inicialmente se simplifica es:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Step(0) \quad}} \\
Vocabp : power(cross(Native, Foreign)); \\
NativeWordsKnownp : power(Native); \\
ForeignWordsKnownp : power(Foreign); \\
CumuMaxMarksp : z(inf); \\
CumuMarksScoredp : z(inf); \\
AveragePercentp : z(inf); \\
report! : Message \\
\hline
NativeWordsKnownp = \mathrm{dom}(Vocabp) \land \\
ForeignWordsKnownp = \mathrm{ran}(Vocabp) \land \\
Vocabp = Vocab \cup \{n? \mapsto f?\} \land \\
n? \mapsto f? \in Vocab \Rightarrow report! = AlreadyKnownPair \land \\
n? \mapsto f? \notin Vocab \Rightarrow report! = NewPairEntered \\
\hline
\end{array}
$$

- Ahora se aplica la regla de un punto

$$
\frac{\begin{array}{l} \exists\, Vocabp : \mathbb{P}(Native \times Foreign) \bullet p \land \\ Vocabp = Vocab \cup \{n? \mapsto f?\} \end{array}}{\begin{array}{l} Vocab \cup \{n? \mapsto f?\} \in \mathbb{P}(Native \times Foreign) \land \\ p[Vocab \cup \{n? \mapsto f?\} / Vocabp] \end{array}} \quad [\ \textsf{Regla de un punto}\ ]
$$

para obtener:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Step(1) \quad}} \\
NativeWordsKnownp : power(Native); \\
ForeignWordsKnownp : power(Foreign); \\
CumuMaxMarksp : z(inf); \\
CumuMarksScoredp : z(inf); \\
AveragePercentp : z(inf); \\
report! : Message \\
\hline
NativeWordsKnownp = \mathrm{dom}(Vocab \cup \{n? \mapsto f?\}) \land \\
ForeignWordsKnownp = \mathrm{ran}(Vocab \cup \{n? \mapsto f?\}) \land \\
n? \mapsto f? \in Vocab \Rightarrow report! = AlreadyKnownPair \land \\
n? \mapsto f? \notin Vocab \Rightarrow report! = NewPairEntered \\
\hline
\end{array}
$$

- A continuación la regla de un punto se aplica sobre *NativeWordsKnownp* y *ForeignWordsKnownp*:

$$\frac{\exists\, NativeWordsKnownp : \mathbb{P}(Native) \bullet p \wedge \\ NativeWordsKnownp = \mathrm{dom}(Vocab \cup \{n? \mapsto f?\})}{\mathrm{dom}(Vocab \cup \{n? \mapsto f?\}) \in power(Native) \wedge \\ p[\mathrm{dom}(Vocab \cup \{n? \mapsto f?\})/NativeWordsKnownp]} \quad [\text{ R. de un pto. }]$$

$$\frac{\exists\, ForeignWordsKnownp : \mathbb{P}(Foreign) \bullet p \wedge \\ ForeignWordsKnownp = \mathrm{ran}(Vocab \cup \{n? \mapsto f?\})}{\mathrm{ran}(Vocab \cup \{n? \mapsto f?\}) \in power(Foreign) \wedge \\ p[\mathrm{ran}(Vocab \cup \{n? \mapsto f?\})/ForeignWordsKnownp]} \quad [\text{ R. de un pto. }]$$

para obtener finalmente:

$$\begin{array}{|l}
\hline
\textit{PAddPair} \\
\textit{WellFormedVocab}; \\
\textit{RecordOfProgress}; \\
Vocab : power(cross(Native, Foreign)); \\
n? : Native; \\
f? : Foreign \\
\hline
\exists\, report! : Message \bullet \\
\quad Vocab \cup \{n? \mapsto f?\} \in power(cross(Native, Foreign)) \wedge \\
\quad \mathrm{dom}(Vocab \cup \{n? \mapsto f?\}) \in power(Native) \wedge \\
\quad \mathrm{ran}(Vocab \cup \{n? \mapsto f?\}) \in power(Foreign) \wedge \\
\quad n? \mapsto f? \in Vocab \Rightarrow report! = AlreadyKnownPair \wedge \\
\quad n? \mapsto f? \notin Vocab \Rightarrow report! = NewPairEntered \\
\hline
\end{array}$$

3. La siguiente especificación fue tomada de [WoJ+97]. Se desea especificar el funcionamiento de un *buffer* que sirva para la implantación de una cola. Este buffer puede almacenar un conjunto finito de valores. El tipo de los elementos de la cola es caracter, deben especificarse las operaciones de entrada y salida de la cola.

El siguiente conjunto define el tipo de los elementos de la cola.

$[CHAR]$

La siguiente construcción define dos funciones, la primera regresa el primer elemento en una secuencia de tipo X y la segunda la secuencia menos el primer elemento.

$$
\begin{array}{l}
[X] \\
\hline
head : \operatorname{seq} X \rightarrow X; \\
tail : \operatorname{seq} X \rightarrow \operatorname{seq} X \\
\hline
\forall\, s : \operatorname{seq} X \bullet \\
\qquad head\ s = s1 \wedge \\
\qquad tail\ s = (\lambda\, n : 1 \mathrel{..} \#s - 1 \bullet s(n+1))
\end{array}
$$

Nuestro *buffer* es modelado por el esquema:

$$
\begin{array}{l}
Buffer \\
\hline
buffer : \operatorname{seq} CHAR; \\
size : \mathbb{N}; \\
max\_size : \mathbb{N} \\
\hline
size = \#buffer; \\
size \leq max\_size
\end{array}
$$

El número de elementos presentes es igual a la longitud de la secuencia y nunca deberá exceder el tamaño máximo del buffer.

El estado inicial del sistema está caracterizado por el buffer vacío.

$$
\begin{array}{l}
BufferInit \\
\hline
Buffer' \\
\hline
buffer' = \langle\rangle
\end{array}
$$

La capacidad del buffer no puede ser cambiada después de ser accedido.

$$
\begin{array}{l}
UpdateBuffer \\
\hline
\Delta Buffer \\
\hline
max\_size' = max\_size
\end{array}
$$

Si el buffer no está lleno entonces un elemento es agregado. El nuevo elemento es apendizado[1] al final de la secuencia que representa el buffer.

[1] El operador $\frown$ es usado para concatenar dos secuencias.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{BufferIn0}}} \\
\textit{UpdateBuffer}; \\
x? : \textit{CHAR} \\
\hline
\textit{size} < \textit{max\_size}; \\
\textit{buffer}' = \textit{buffer} \frown \langle x? \rangle \\
\hline
\end{array}
$$

La siguiente expresión calcula las precondiciones de la operación de entrada.

$PreBufferIn0 \mathrel{\widehat=} \mathrm{pre}\ BufferIn0$

Extraer un elemento es posible sólo si el buffer no está vacío. El elemento obtenido es el que se encuentra en la cabeza del buffer.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{BufferOut0}}} \\
\textit{UpdateBuffer}; \\
x! : \textit{CHAR} \\
\hline
\textit{buffer} \neq \langle \rangle; \\
\textit{buffer}' = \textit{tail buffer}; \\
x! = \textit{head buffer} \\
\hline
\end{array}
$$

Hasta el momento los esquemas definen las operaciones de una inserción o extracción exitosa.
Para dar una interfaz informativa de lo que sucede en el sistema, consideraremos tres posibles mensajes

$Report ::= ok \mid full \mid empty$

Un mensaje de éxito se define con la operación

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Success}}} \\
\textit{report}! : \textit{Report} \\
\hline
\textit{report}! = \textit{ok} \\
\hline
\end{array}
$$

Cuando el tamaño del buffer es igual a *max_size* debe producirse un mensaje informando que el buffer está lleno

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BufferInError \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Buffer; \\
report! : Report \\
\hline
size = max_size; \\
report! = full \\
\hline
\end{array}$$

Un mensaje de buffer vacío se produce cuando se intenta extraer un elemento de un buffer con cero elementos

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad BufferOutError \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Buffer; \\
report! : Report \\
\hline
buffer = \langle\rangle; \\
report! = empty \\
\hline
\end{array}$$

Las dos operaciones de mensajes de error no deben cambiar el estado del sistema. Esto es especificado mediante el uso de la decoración Ξ.

Finalmente, podremos definir las versiones robustas que realizan un manejo de errores de las operaciones de *entrada/salida* del buffer.

$$BufferIn \mathrel{\widehat=} (BufferIn0 \land Success) \lor BufferInError$$

$$BufferOut \mathrel{\widehat=} (BufferOut0 \land Success) \lor BufferOutError$$

A continuación se presentan los resultados obtenidos por el prototipo correspondientes a las expresiones del cálculo de esquemas, así como del cálculo de precondiciones.

Empezamos con la expansión de la expresión:

$$BufferIn \mathrel{\widehat=} (BufferIn0 \land Success) \lor BufferInError$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BufferIn}} \\
\hline
\textit{UpdateBuffer}; \\
x? : \textit{CHAR}; \\
\Xi \textit{Buffer}; \\
\textit{report}! : \textit{Report} \\
\hline
(\textit{size} < \textit{max\_size} \land \\
\textit{bufferp} = \textit{buffer} \frown \langle x? \rangle \land \\
\textit{report}! = \textit{ok}) \lor \\
(\textit{size} = \textit{max\_size} \land \\
\textit{report}! = \textit{full}) \\
\hline
\end{array}
$$

La expansión de la expresión

$$\textit{BufferOut} \mathrel{\widehat{=}} (\textit{BufferOut}0 \land \textit{Success}) \lor \textit{BufferOutError}$$

está descrita por el siguiente esquema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BufferOut}} \\
\hline
\textit{UpdateBuffer}; \\
x! : \textit{CHAR}; \\
\Xi \textit{Buffer}; \\
\textit{report}! : \textit{Report} \\
\hline
(\textit{buffer} \neq \langle\rangle \land \\
\textit{bufferp} = \textit{tail}\ \textit{buffer} \land \\
x! = \textit{head}\ \textit{buffer} \land \\
\textit{report}! = \textit{ok}) \lor \\
(\textit{buffer} = \langle\rangle \land \\
\textit{report}! = \textit{empty}) \\
\hline
\end{array}
$$

Ahora se presenta la salida producida por el prototipo y que corresponde a los pasos que sigue para calcular:

$$\textit{PreBufferIn}0 \mathrel{\widehat{=}} \text{pre } \textit{BufferIn}0$$

- El esquema $BufferIn0$ es:

$$
\begin{array}{|l}
\hline
\textit{Begin precondition calculus} \\
UpdateBuffer; \\
x? : CHAR \\
\hline
size < max\_size \land \\
bufferp = buffer \frown \langle x? \rangle \\
\hline
\end{array}
$$

- Se necesita expandir todas las expresiones de la forma ΔS y Ξ a la forma $S;\ S'$. $BufferIn0$ aparentemente no tiene ninguna de tales expresiones, pero $UpdateBuffer$ sí, entonces procedemos a expandir este último esquema.

$$
\begin{array}{|l}
\hline
\textit{Expanding } \Delta \\
Buffer; \\
Buffer'; \\
x? : CHAR \\
\hline
size < max\_size \land \\
bufferp = buffer \frown \langle x? \rangle \\
\hline
\end{array}
$$

- El siguiente paso es aplicar la definición de aplicabilidad

  $\exists\, State';\ Output \bullet Oper$

  esto es, separamos los estados posteriores (primados) y las variables de salida, se cuantifican existencialmente y pasan a formar parte del predicado.

$$
\begin{array}{|l}
\hline
Pre \\
Buffer; \\
x? : CHAR \\
\hline
\exists\, Buffer' \bullet \\
\quad size < max\_size \land \\
\quad bufferp = buffer \frown \langle x? \rangle \\
\hline
\end{array}
$$

- Ahora se procede a la simplificación del predicado del esquema anterior aplicando la regla de un punto descrita en la sección 4.2. De esta forma, se construye un nuevo esquema donde su parte declarativa está formada por la parte declarativa del predicado de *Pre*, y la parte axiomática está formada por las restricciones del predicado, así que después de expandir *Buffer* nos queda el siguiente esquema:

$$
\begin{array}{|l}
\hline
\textit{Step}(0) \\
\textit{bufferp} : \textit{power}(\textit{cross}(z([0, \textit{inf}]), \textit{CHAR})); \\
\textit{sizep} : z(\textit{inf}); \\
\textit{max\_sizep} : z(\textit{inf}) \\
\hline
\textit{sizep} = \#(\textit{bufferp}) \wedge \\
\textit{sizep} \leq \textit{max\_sizep} \wedge \\
\textit{size} < \textit{max\_size} \wedge \\
\textit{bufferp} = \textit{buffer} \frown \langle x? \rangle \\
\hline
\end{array}
$$

- Se aplica la regla de un punto

$$
\frac{\exists\, \textit{bufferp} : \mathbb{P}(\mathbb{N} \times \textit{CHAR}) \bullet p \wedge \textit{bufferp} = \textit{buffer} \frown \langle x? \rangle}{\textit{buffer} \frown \langle x? \rangle \in \mathbb{P}(\mathbb{N} \times \textit{CHAR}) \wedge p[\textit{buffer} \frown \langle x? \rangle / \textit{bufferp}]} \quad [\ \mathsf{R1P}\ ]
$$

de esta forma tendremos

$$
\begin{array}{|l}
\hline
\textit{Step}(1) \\
\textit{sizep} : z(\textit{inf}); \\
\textit{max}_s\textit{izep} : z(\textit{inf}) \\
\hline
\textit{sizep} = \#(\textit{buffer} \frown \langle x? \rangle) \wedge \\
\textit{sizep} \leq \textit{max\_sizep} \wedge \\
\textit{size} < \textit{max\_size} \\
\hline
\end{array}
$$

- Nuevamente se aplica la regla de un punto

$$
\frac{\exists\, \textit{sizep} : \mathbb{Z} \bullet p \wedge \textit{sizep} = \#(\textit{buffer} \frown \langle x? \rangle)}{\#(\textit{buffer} \frown \langle x? \rangle) \in \mathbb{Z} \wedge p[\#(\textit{buffer} \frown \langle x? \rangle) / \textit{sizep}]} \quad [\ \mathsf{R1P}\ ]
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\_\, Step(2)} \\
max\_sizep : z(inf) \\
\hline
\#(buffer \frown \langle x? \rangle) \leq max\_sizep \land \\
size < max\_size \\
\hline
\end{array}
$$

- Finalmente el esquema que contiene las precondiciones de *PreBufferIn*0 es:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\_\, PreBufferIn0} \\
Buffer; \\
x? : CHAR \\
\hline
\exists\, max\_sizep : z(inf) \bullet \\
\qquad buffer \frown \langle x? \rangle \in power(cross(z([0, inf]), CHAR)) \land \\
\qquad \#(buffer \frown \langle x? \rangle) \in z(inf) \land \\
\qquad \#(buffer \frown \langle x? \rangle) \leq max\_sizep \land \\
\qquad size < max\_size \\
\hline
\end{array}
$$

La salida que genera el prototipo y que se ha mostrado en las pruebas realizadas, está escrita en formato LaTeX, con esto, se tiene la posibilidad de que los esquemas resultantes de expresiones del cálculo de esquemas puedan ser utilizados nuevamente como parte de la especificación o como parte de un documento.

Conclusiones

La fase de diseño en los sistemas de software es crucial porque durante ésta se deben tomar decisiones que en determinado momento tendrán que afectar de forma directa el funcionamiento del mismo. Los métodos formales son una herramienta poderosa con la que es posible la especificación del funcionamiento de un sistema de forma breve, clara y sin ambigüedades. Además los conceptos matemáticos nos ayudan a probar que el significado de una especificación corresponda al funcionamiento real del sistema que queremos especificar.

Como una primera aproximación en el camino de construir un sistema seguro, el cálculo de precondiciones juega un papel importante ya que este proceso nos conduce al diseño de sistemas que tendrán la capacidad de resolver favorablemente situaciones que podrían conducirlos a un estado indefinido gracias a que se han investigado las condiciones que lo llevarían a tal estado. Para tal efecto, el prototipo resulta una herramienta para atacar el problema tanto de verificar especificaciones y posteriormente para la determinación de precondiciones.

El uso de Internet resultó de valiosa ayuda porque la información encontrada sobre el tema aportó algunas ideas para atacar algunos problemas de implantación. Debido a la gran difusión de la notación Z, mucha gente alrededor del planeta se encuentra trabajando en prototipos, la mayoría de los trabajos encontrados en el Web implantan la comprobación de tipos, probadores de teoremas usando lógica de alto orden, otros intentan "animar" especificaciones trasladando el código Z directamente a un lenguaje de programación y entonces ejecutando el programa en este lenguaje (aunque sólo se trabaja con un subconjunto de la notación Z) y hasta el momento de la redacción del presente trabajo, no se encontró información sobre trabajos relacionados con el cálculo de precondiciones y dada la importancia del tema el prototipo propone una herramienta que automatiza el cálculo de precondiciones además que este trabajo puede servir para despertar el interés de la comunidad dedicada al estudio de Z y poder recibir la aportación de ideas para la continuidad del mismo.

El contar con esta herramienta nos ayudará en el aprendizaje de la metodología de especificación bajo Z. La implantación nos muestra que es posible la construcción de sistemas concretos que en principio se proponen como modelos teoricos y abstarctos, de tal forma que en un futuro pueda considerarse a Z como un lenguaje de programación poderoso que no sólo sirva para la especificación formal de un sistema a nivel de documentación, sino también como el lenguaje que nos permita la traducción al código ejecutable a través de un compilador y donde estemos seguros que la aplicación final cumpla cabalmente con los requerimientos iniciales del sistema.

Finalmente, dentro de los trabajos a futuro podemos mencionar:

- La implantación de un módulo que permita trasladar una especificación al ambiente de un probador de teoremas tal como HOL y de esta forma ser capaz de probar propiedades

- La posibilidad de refinar especificaciones para conseguir algun grado de concretitud en algún lenguaje de programación.

- La migración a un ambiente más amigable que incluya ventanas, un editor de especificaciones, etc. donde el prototipo se escriba en un lenguaje que peermita la construcción de un archivo ejecutable.

Apéndice A

Zed-Csp

La descripción de los comandos que soportan los símbolos de la notación Z ha sido extraida de [SpM290] en donde se explican los pasos a seguir cuando lo que se desea es vizualizar una especificación. En las siguientes secciones se presentan los símbolos gráficos de la notación Z y enseguida su equivalente en términos de LaTeX[1].

A.1 Párrafos

Todos los párrafos en una especificación deberán acotarse por medio de

```
\begin{spec}
\end{spec}
```

A.1.1 Tipos básicos

$$[T_1, \ldots, T_n]$$

```
\begin{zed}
[T_1,\ldots,T_n]
\end{zed}
```

A.1.2 Tipos de datos simples

$$\mathit{Tipo} ::= E_1 \mid \ldots \mid E_n$$

[1] Usando el estilo zed-csp.sty.

```
\begin{zed}
Tipo ::= E_1 | \ldots | E_n
\end{zed}
```

A.1.3 Definición abreviada

$$id == Expr$$

```
\begin{zed}
id == Expr
\end{zed}
```

A.1.4 Definición axiomática

$$\begin{array}{|l} D_1;\ \ldots;\ D_n \\ \hline P_1;\ \ldots;\ P_n \end{array}$$

```
\begin{axdef}
D_1 ; \ldots; D_n
\where
P_1 ; \ldots; P_n
\end{axdef}
```

A.1.5 Definición genérica

$$\begin{array}{|l} [X_1, \ldots, X_n] \\ \hline D_1;\ \ldots;\ D_n \\ \hline P_1;\ \ldots;\ P_n \\ \hline \end{array}$$

```
\begin{gendef}[X_1,ldots,X_n]
D_1 ; \ldots; D_n
\where
P_1 ; \ldots; P_n
\end{gendef}
```

A.1.6 Esquemas

Nombre
$D_1; \ \ldots; \ D_n$
$P_1; \ \ldots; \ P_n$

```
\begin{schema}{Nombre}
D_1 ; \ldots; D_n
\where
P_1 ; \ldots; P_n
\end{schema}
```

Definición en forma horizontal

$$S \mathrel{\widehat=} Expr\_Schema$$

```
S \defs Expr\_Schema
```

A.2 Predicados

$\forall Decls \mid P \bullet Q$	`\forall Decls	P @ Q`
$\exists Decls \mid P \bullet Q$	`\exists Decls	P @ Q`
$(\lambda Decl \mid P \bullet E)$	`(\lambda Decl | P @ E)`
$p \land q$	`p \land q`
$p \lor q$	`p \lor q`
$p \Rightarrow q$	`p \implies q`
$p \Leftrightarrow q$	`p \iff q`
true	true
false	false

A.3 Conjuntos

$\{Exp_1, \ldots, Exp_n\}$	`\{Exp_1, \ldots, Exp_n\}`
$\{Decls \mid Pred \bullet Exp\}$	`\{Decls	Pred @ Exp\}`
$A = B$	`A = B`
$A \neq B$	`A \neq B`
$a \in A$	`a \in A`

$a \notin A$	`a \notin A`
$\emptyset$	`\emptyset`
$A \subset B$	`A \subset B`
$A \subseteq B$	`A \subseteq B`
$\mathbb{P}\, B$	`\power B`
$\mathbb{F}\, B$	`\finite B`
$\# A$	`\# A`
$A \cup B$	`A \cup B`
$A \cap B$	`A \cap B`
$A \setminus B$	`A \setminus B`
$A \times B$	`A \cross B`

A.4 Decoraciones

ΔS	`\Delta S`
ΞS	`\Xi S`
$a?$	`a?`
$a!$	`a!`

A.5 Operadores binarios

$\mapsto$	`\mapsto`	mod	`\mod`
$\ldotp\ldotp$	`\upto`	$\oplus$	`\oplus`
$+$	`+`	$\lhd$	`\dres`
$-$	`-`	$\rhd$	`\rres`
$\setminus$	`\setminus`	$\ntriangleleft$	`\ndres`
$\frown$	`\cat`	$\ntriangleright$	`\nrres`
$*$	`*`	$\fatsemi$	`\comp`
div	`\div`		

A.6 Relaciones infijas

$\neq$	`\neq`	⇸	`\pfun`
$\notin$	`\notin`	$\rightarrow$	`\fun`
$\subseteq$	`\subseteq`	⤔	`\pinj`
$\subset$	`\subset`	$\rightarrowtail$	`\inj`
$<$	`<`	⤀	`\psurj`
$>$	`>`	$\twoheadrightarrow$	`\surj`
$\leq$	`\leq`	⤖	`\bij`
$\geq$	`\geq`	⇻	`\ffun`
$\leftrightarrow$	`\rel`	⤕	`\finj`

A.7 Operadores prefijos

$\mathbb{P}$	`\power`	dom	`\dom`
$\mathbb{F}$	`\finset`	ran	`\ran`
seq	`\seq`		

Apéndice B

Gramática.

La primera sección presenta la gramática como está descrita en [PoB+91]. Después se presenta modificada, para aceptar comandos LaTeXy eliminando la recursi'on izquierda.

B.1 Gramática base

$$
\begin{array}{rcl}
\mathit{SPECIFICATION} & \rightarrow & \mathit{PARAGRAPH\ NL} \ldots \mathit{NL\ PARAGRAPH} \\
\mathit{PARAGRAPH} & \rightarrow & \mathit{DEFNOBOX} \\
 & | & \mathit{AXIOMATICBOX} \\
 & | & \mathit{SCHEMABOX} \\
 & | & \mathit{GENERICBOX} \\
\mathit{DEFNOBOX} & \rightarrow & \mathit{KINDDEFNOBOX} \\
\mathit{KINDDEFNOBOX} & \rightarrow & [\mathit{IDENT}, \ldots, \mathit{IDENT}] \\
 & | & \mathit{DEFLHS} == \mathit{EXPRESSION} \\
 & | & \mathit{IDENT} ::= \mathit{BRANCH}' \,|'\, \ldots' \,|'\, \mathit{BRANCH} \\
 & | & \mathit{PREDICATE} \\
\mathit{AXIOMATICBOX} & \rightarrow &
\end{array}
$$

$$
\begin{array}{|l}
\mathit{DECL\_PART} \\
\hline
\mathit{AXIOM\_PART}
\end{array}
$$

$\mathit{SCHEMABOX} \rightarrow$

SCHEMA_NAME

DECL_PART

AXIOM_PART

$$
\begin{array}{rcl}
SCHEMABOX & \rightarrow & SCHEMANAME \mathrel{\widehat=} SCHEMAEXP \\
GENERICBOX & \rightarrow &
\end{array}
$$

GENFORMALSDECL_PART

AXIOMPART

$$
\begin{array}{rcl}
DECL\_PART & \rightarrow & BASICDCL\ SEP\ \ldots SEP\ BASICDCL \\
AXIOM\_PART & \rightarrow & PREDICATE\ SEP\ \ldots SEP\ PREDICATE \\
SEP & \rightarrow & ; \\
DEFLHS & \rightarrow & IDENT \\
BRANCH & \rightarrow & IDENT \\
SCHEMAEXP & \rightarrow & \forall\, SCHEMATEXT \bullet SCHEMAEXP \\
 & | & \exists\, SCHEMATEXT \bullet SCHEMAEXP \\
 & | & SCHEMAEXP1 \\
SCHEMAEXP1 & \rightarrow & [SCHEMATEXT] \\
 & | & SCHEMAREF \\
 & | & \mathrm{pre}\ SCHEMAEXP1 \\
 & | & SCHEMAEXP1 \wedge SCHEMAEXP1 \\
 & | & SCHEMAEXP1 \vee SCHEMAEXP1 \\
 & | & SCHEMAEXP1 \Rightarrow SCHEMAEXP1 \\
 & | & SCHEMAEXP1 \Leftrightarrow SCHEMAEXP1 \\
 & | & SCHEMAEXP1 \backslash (DECLNAME, \ldots, DECLNAME) \\
 & | & SCHEMAEXP1 \mathbin{;} SCHEMAEXP1 \\
 & | & (SCHEMAEXP) \\
SCHEMATEXT & \rightarrow & DECL\_PART'\ |'\ PREDICATE \\
SCHEMAREF & \rightarrow & SCHEMANAME\ DECORATION \\
BASICDCL & \rightarrow & DECLNAME, \ldots, DECLNAME : EXPRESSION
\end{array}
$$

$$
\begin{array}{rcl}
 & | & SCHEMAREF \\
PREDICATE & \rightarrow & \forall\, SCHEMATEXT \bullet PREDICATE \\
 & & \exists\, SCHEMATEXT \bullet PREDICATE \\
 & | & PREDICATE1 \\
PREDICATE1 & \rightarrow & EXPRESSION\ REL\ EXPRESSION\ REL \ldots REL \\
 & & EXPRESSION \\
 & | & SCHEMAREF \\
 & | & PREREL\ EXPRESSION \\
 & | & [true \mid fasle] \\
 & | & \not{P}REDICATE1 \\
 & | & PREDICATE1 \wedge PREDICATE1 \\
 & | & PREDICATE1 \vee PREDICATE1 \\
 & | & PREDICATE1 \Rightarrow PREDICATE1 \\
 & | & PREDICATE1 \Leftrightarrow PREDICATE1 \\
 & | & (PREDICATE) \\
REL & \rightarrow & [= \mid \in] \\
 & | & INREL \\
EXPRESSION0 & \rightarrow & \lambda\, SCHEMATEXT \bullet EXPRESSION \\
 & | & EXPRESSION \\
EXPRESSION & \rightarrow & EXPRESSION\ INGEN\ EXPRESSION \\
 & | & EXPRESSION1 \times EXPRESSION1\ \times \ldots \times \\
 & & EXPRESSION1 \\
 & | & EXPRESSION1 \\
EXPRESSION1 & \rightarrow & EXPRESSION1\ INFUN\ EXPRESION1 \\
 & | & \mathbb{P}\, EXPRESSION3 \\
 & | & PREGEN\ EXPRESSION3 \\
 & | & -EXPRESSION3 \\
 & | & EXPRESSION3 (\!|\ EXPRESSION0\ |\!) \\
 & | & EXPRESSION2 \\
EXPRESSION2 & \rightarrow & EXPRESSION2\ EXPRESSION3 \\
 & | & EXPRESSION3 \\
EXPRESSION3 & \rightarrow & VARNAME[GENACTUALS] \\
 & | & number
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \mathit{SCHEMAREF} \\
 & | & \mathit{SETEXP} \\
 & | & < [\mathit{EXPRESSION}, \ldots, \mathit{EXPRESSION}] > \\
 & | & (\mathit{EXPRESSION}, \ldots, \mathit{EXPRESSION}) \\
 & | & (\mathit{EXPRESSION}0) \\
\mathit{SETEXP} & \rightarrow & \{[\mathit{EXPRESSION}, \ldots, \mathit{EXPRESSION}]\} \\
 & | & \{\mathit{SCHEMATEXT} \bullet \mathit{EXPRESSION}\} \\
\mathit{DECLNAME} & \rightarrow & \mathit{IDENT} \\
 & | & \mathit{OPNAME\ WORD} \\
\mathit{OPNAME} & \rightarrow & \_\mathit{WORD}\_ \\
\mathit{INSYM} & \rightarrow & \mathit{INFUN} \\
 & | & \mathit{INGEN} \\
 & | & \mathit{INREL} \\
\mathit{PRESYM} & \rightarrow & \mathit{PREGEN} \\
\mathit{DECORATION} & \rightarrow & \mathit{STROKE} \\
\mathit{GENFORMALS} & \rightarrow & [\mathit{IDENT}, \ldots, \mathit{IDENT}] \\
\mathit{GENSACTUALS} & \rightarrow & [\mathit{EXPRESSION}, \ldots, \mathit{EXPRESSION}] \\
\mathit{STROKE} & \rightarrow & [' | ! \,|?] \\
\mathit{VARNAME} & \rightarrow & \mathit{IDENT\ GENS} \\
\mathit{IDENT} & \rightarrow & \mathit{WORD\ DECORATION} \\
\mathit{SCHEMANAME} & \rightarrow & \mathit{WORD} \\
\mathit{INFUN} & \rightarrow & [\mapsto | \ldotp\ldotp | + | - | * | \cup | \cap | \oplus | \setminus | \frown | \lhd | \rhd | \unlhd] \\
\mathit{PREGEN} & \rightarrow & [\# | \mathbb{P} | \mathrm{dom} | \mathrm{ran} | \mathbb{F} | \mathrm{seq}]
\end{array}
$$

B.2 Gramática modificada

$$
\begin{array}{rcl}
\mathit{SPECIFICATION} & \rightarrow & \texttt{\textbackslash begin\{spec\}} \\
 & & \mathit{PARAGRAPH} \\
 & & \mathit{SPEC}' \\
\mathit{SPEC}' & \rightarrow & \texttt{\textbackslash end\{spec\}} \\
\mathit{SPEC}' & \rightarrow & \mathit{MANYPARAGRAPH} \\
 & & \texttt{\textbackslash end\{spec\}}
\end{array}
$$

MANYPARAGRAPH	$\rightarrow$	*PARAGRAPH*
		MANYPARAGRAPH
MANYPARAGRAPH	$\rightarrow$	ϵ
PARAGRAPH	$\rightarrow$	*DEFNOBOX*
PARAGRAPH	$\mid$	*AXIOMATICBOX*
PARAGRAPH	$\mid$	*SCHEMABOX*
PARAGRAPH	$\mid$	*GENERICBOX*
DEFNOBOX	$\rightarrow$	`\begin{zed}`
		KINDDEFNOBOX
		ANKINDDEFNOBOX
		`\end{zed}`
ANKINDDEFNOBOX	$\rightarrow$	`\also/`
		KINDDEFNOBOX
		ANKINDDEFNOBOX
ANKINDDEFNOBOX	$\rightarrow$	ϵ
KINDDEFNOBOX	$\rightarrow$	*BASICTYPES*
	$\mid$	*DEFLHS == EXPRESSION*
	$\mid$	*IDENT* ::= *BRANCH ANBRANCH*
	$\mid$	*PREDICATE*
BASICTYPES	$\rightarrow$	[*IDENTSIMPLE ANIDENT*
ANIDENT	$\rightarrow$	*COMMA IDENTSIMPLE ANIDENT*
	$\mid$	]
ANBRANCH	$\rightarrow$	$\mid$ *BRANCH ANBRANCH*
	$\mid$	ϵ
AXIOMATICBOX	$\rightarrow$	`\begin{axdef}`
		AXDEF_DECLS
		AXIOMATICBOXP
AXIOMATICBOXP	$\rightarrow$	`\where`
		AXIOMPART
		`\end{axdef}`
AXIOMATICBOXP	$\rightarrow$	`\end{axdef}`
SCHEMABOX	$\rightarrow$	`\begin{schema}{`*IDENT* }
		DECLPART

$$
\begin{array}{rcl}
 & & \mathit{SCHEBOXP} \\
\mathit{SCHEMABOX} & \rightarrow & \mathit{SCHEMANAME} \mathrel{\widehat{=}} \mathit{SCHEMAEXP} \\
\mathit{SCHEBOXP} & \rightarrow & \texttt{\textbackslash where} \\
 & & \mathit{AXIOMPART} \\
 & & \texttt{\textbackslash end\{schema\}} \\
\mathit{SCHEBOXP} & \rightarrow & \texttt{\textbackslash end\{schema\}} \\
\mathit{GENERICBOX} & \rightarrow & \texttt{\textbackslash begin\{gendef\}\{}\mathit{GENFORMALS}\texttt{\}} \\
 & & \mathit{DECLPART} \\
 & & \mathit{GENBOXP} \\
\mathit{GENBOXP} & \rightarrow & \texttt{\textbackslash where} \\
 & & \mathit{AXIOMPART} \\
 & & \texttt{\textbackslash end\{gendef\}} \\
\mathit{GENBOXP} & \rightarrow & \texttt{\textbackslash end\{gendef\}} \\
\mathit{DECLPART} & \rightarrow & \mathit{BASICDCL}\ \mathit{ANBASICDCL} \\
\mathit{ANBASICDCL} & \rightarrow & \mathit{SEP}\ \mathit{BASICDCL}\ \mathit{ANBASICDCL} \\
 & | & \epsilon \\
\mathit{AXIOMPART} & \rightarrow & \mathit{PREDICATE}\ \mathit{ANPREDICATE} \\
\mathit{ANPREDICATE} & \rightarrow & \mathit{SEP}\ \mathit{PREDICATE}\ \mathit{ANPREDICATE} \\
 & | & \epsilon \\
\mathit{SEP} & \rightarrow & \mathit{PCOMMA} \\
\mathit{DEFLHS} & \rightarrow & \mathit{IDENT} \\
 & | & \mathit{PREGEN}\ \mathit{IDENT} \\
 & | & \mathit{IDENT}\ \mathit{INGEN}\ \mathit{IDENT} \\
\mathit{BRANCH} & \rightarrow & \mathit{IDENT} \\
\mathit{SCHEMAEXP} & \rightarrow & \forall\, \mathit{SCHEMATEXT} \bullet \mathit{SCHEMAEXP} \\
 & | & \exists\, \mathit{SCHEMATEXT} \bullet \mathit{SCHEMAEXP} \\
 & | & \mathit{SCHEMAEXP}1 \\
\mathit{SCHEMAEXP}1 & \rightarrow & [\mathit{SCHEMATEXT}]\mathit{SCHEXP}1P \\
 & | & \mathit{SCHEMAREF}\ \mathit{SCHEXP}1P \\
 & | & \not\mathit{SCHEMAEXP}1\ \mathit{SCHEXP}1P \\
 & | & \mathrm{pre}\ \mathit{SCHEMAEXP}1\ \mathit{SCHEXP}1P \\
 & | & (\mathit{SCHEMAEXP})\mathit{SCHEXP}1P \\
\mathit{SCHEXP}1P & \rightarrow & [\land|\lor|\Rightarrow|\ \mathit{iff}\ |\ \setminus\ |\ \fatsemi]\mathit{SCHEMAEXP}1\ \mathit{SCHEXP}1P
\end{array}
$$

$$
\begin{array}{rcl}
SCHEXP1P & \rightarrow & \epsilon \\
SCHEMATEXT & \rightarrow & DECLPART\ SCHTEXTP \\
SCHTEXTP & \rightarrow & '\,|'\ PREDICATE \\
 & | & \epsilon \\
SCHEMAREF & \rightarrow & SCHEMANAME\ DECORATION \\
 & | & \Delta SCHEMANAME \\
 & | & \Xi SCHEMANAME \\
GENS & \rightarrow & [GENSACTUALS] \\
 & | & \epsilon \\
AXDEF\_DECLS & \rightarrow & BASICDCL\ AXDEF\_NEXTDECL \\
AXDEF\_NEXTDECL & \rightarrow & SEP\ BASICDCL\ AXDEF\_NEXTDECL \\
 & | & \epsilon \\
BASICDCL & \rightarrow & DECLNAME\ ANDECLNAME\ :EXPRESSION \\
 & | & SCHEMAREF \\
ANDECLNAME & \rightarrow & COMMA\ DECLNAME\ ANDECLNAME \\
ANDECLNAME & \rightarrow & \epsilon \\
PREDICATE & \rightarrow & [\forall \mid \exists] SCHEMATEXT\ PREDICATEP \\
PREDICATE & \rightarrow & PREDICATE1 \\
PREDICATEP & \rightarrow & \bullet\ PREDICATE \\
PREDICATEP & \rightarrow & \epsilon \\
PREDICATE1 & \rightarrow & EXPRESSION\ ANEXPRESSION\ PREDICATE1P \\
 & | & SCHEMAREF\ PREDICATE1P \\
 & | & [true \mid fasle] PREDICATE1P \\
 & | & \not PREDICATE1\ PREDICATE1P \\
 & | & (PREDICATE) PREDICATE1P \\
PREDICATE1P & \rightarrow & [\vee | \wedge | \Rightarrow | \Leftrightarrow] PREDICATE1 PREDICATE1P \\
PREDICATE1P & \rightarrow & \epsilon \\
ANEXPRESSION & \rightarrow & REL\ EXPRESSION\ ANEXPRESSION \\
ANEXPRESSION & \rightarrow & \epsilon \\
REL & \rightarrow & [= | \in] \\
REL & \rightarrow & INREL \\
EXPRESSION0 & \rightarrow & \lambda\, SCHEMATEXT \bullet EXPRESSION \\
EXPRESSION0 & \rightarrow & EXPRESSION
\end{array}
$$

$$
\begin{array}{rcl}
EXPRESSION & \rightarrow & EXPRESSION1\ EXPR\_P \\
EXPR\_P & \rightarrow & \times EXPRESSION1\ ANEXP1 EXPP \\
EXPR_P & \rightarrow & EXPP \\
EXPP & \rightarrow & INGEN\ EXPRESSION\ EXPP \\
EXPP & \rightarrow & \epsilon \\
ANEXP1 & \rightarrow & \times EXPRESSION1\ ANEXP1 \\
ANEXP1 & \rightarrow & \epsilon \\
EXPRESSION1 & \rightarrow & EXPRESSION2\ EXP1P \\
 & | & \mathbb{P}\, EXPRESSION3\ EXP1P \\
 & | & PREGEN\ EXPRESSION3\ EXP1P \\
 & | & -EXPRESSION3\ EXP1P \\
 & | & EXPRESSION3 (\!|\ EXPRESSION0\ |\!) EXP1P \\
EXP1P & \rightarrow & PARSE_A RITMETIC \\
EXP1P & \rightarrow & \epsilon \\
EXPRESSION2 & \rightarrow & EXPRESSION3\ EXP2P \\
EXP2P & \rightarrow & EXPRESSION3\ EXP2P \\
EXP2P & \rightarrow & \epsilon \\
EXPRESSION3 & \rightarrow & VARNAME \\
 & | & numero \\
 & | & [\mathbb{Z} \mid \mathbb{N}] \\
 & | & SCHEMAREF \\
 & | & SETEXP \\
 & | & SEQUENCE \\
 & | & (EXPRESSION0\ FLEXP3 \\
FLEXP3 & \rightarrow & COMMA\ TUPLE \\
FLEXP3 & \rightarrow &) \\
TUPLE & \rightarrow & MEXPRESSION) \\
PARSE\_ARITMETIC & \rightarrow & INFUN\ EXPRESSION1\ EXP1P \\
SEQUENCE & \rightarrow & \langle MEXPRESSION \rangle \\
SEQUENCE & \rightarrow & \langle \rangle \\
MEXPRESSION & \rightarrow & EXPRESSION\ LEXPRESSION \\
LEXPRESSION & \rightarrow & COMMA\ EXPRESSION\ LEXPRESSION \\
LEXPRESSION & \rightarrow & \epsilon
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{SETCLASS} & \rightarrow & \mathit{SCHEMATEXT} \\
\mathit{SETCLASS} & \rightarrow & \mathit{MEXPRESSION} \\
\mathit{SETEXP} & \rightarrow & \{\mathit{SETCLASS}\ \mathit{SETEXPP} \\
\mathit{SETEXP} & \rightarrow & \{\} \\
\mathit{SETEXPP} & \rightarrow & \bullet\ \mathit{EXPRESSION}\} \\
\mathit{SETEXPP} & \rightarrow & \} \\
\mathit{DECLNAME} & \rightarrow & \mathit{IDENT} \\
\mathit{DECLNAME} & \rightarrow & \mathit{OPNAME}\ \mathit{WORD} \\
\mathit{OPNAME} & \rightarrow & \_\mathit{WORD}\_ \\
\mathit{INSYM} & \rightarrow & \mathit{INFUN} \\
\mathit{INSYM} & \rightarrow & \mathit{INGEN} \\
\mathit{INSYM} & \rightarrow & \mathit{INREL} \\
\mathit{PRESYM} & \rightarrow & \mathit{PREGEN} \\
\mathit{PRESYM} & \rightarrow & \mathit{PREREL} \\
\mathit{DECORATION} & \rightarrow & \mathit{STROKE}\ \mathit{MULTISTROKE} \\
\mathit{DECORATION} & \rightarrow & \epsilon \\
\mathit{MULTISTROKE} & \rightarrow & \mathit{STROKE}\ \mathit{MULTISTROKE} \\
\mathit{MULTISTROKE} & \rightarrow & \epsilon \\
\mathit{GENFORMALS} & \rightarrow & \mathit{IDENT}\ \mathit{MANYIDENT} \\
\mathit{MANYIDENT} & \rightarrow & \mathit{COMMA}\ \mathit{IDENT}\ \mathit{MANYIDENT} \\
\mathit{MANYIDENT} & \rightarrow & \epsilon \\
\mathit{GENSACTUALS} & \rightarrow & \mathit{MEXPRESSION} \\
\mathit{STROKE} & \rightarrow & [' \mid ! \mid ?] \\
\mathit{VARNAME} & \rightarrow & \mathit{IDENT}\ \mathit{GENS} \\
\mathit{VARNAME} & \rightarrow & \emptyset \\
\mathit{IDENTSIMPLE} & \rightarrow & \mathit{WORD} \\
\mathit{IDENT} & \rightarrow & \mathit{WORD}\ \mathit{DECORATION} \\
\mathit{SCHEMANAME} & \rightarrow & \mathit{WORD} \\
\mathit{INFUN} & \rightarrow & [\ \mapsto \mid \ldotp\ldotp \mid + \mid - \mid * \mid \cup \mid \cap \mid \oplus \mid \setminus \mid \frown \mid \lhd \mid \rhd \mid \unlhd] \\
\mathit{PREGEN} & \rightarrow & [\# \mid \mathbb{P} \mid \mathrm{dom} \mid \mathrm{ran} \mid \mathbb{F} \mid \mathrm{seq}\,]
\end{array}
$$

Bibliografía

[PoB+91] Ben Potter, Jane Sinclair and David Till. **An Introduction to Formal Specification and Z**, Prentice Hall, 1991.

[LaL94] Leslie Lamport. **A Document Preparation System LaTeX: User's Guide And Reference Manual**, Addison-Wesley, 1994.

[JoW] Wendy Jonsthon. **A type Checker for Object-Z**. Technical Report No. 96-24, The University of Queensland, Australia.

[WiJ] Jan Wielemaker. **SWI-Prolog 2.7** *Reference Manual*, University of Amsterdam.

[SpM1] Michael Spivey. **Richer Types for Z**, Oxford University Computing Laboratory.

[CoJ+87] Jacques Cohen and Timothy J. Hickey. **Parsing and Compiling Using Prolog**, ACM Transactions on Programming Languages and Systems, Vol. 9 No.2 April 1987.

[WaD80] David H. D. Warren. **Logic Programming and Compiler Writing**, Software-Practice and Experience, Vol. 10, 97-125 (1980).

[AhA+90] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. **Compilers: Principles, Techniques and Tools**, Addison Wesley, 1990.

[Web] **Operator precedence parser for Wesley**
http://www.cs.um.edu.mt/ hzarb/CSM201/lecture6/node6.html

[ReS+90] Steve Reeves and Michael Clarke. **Logic for Computer Science**, Addison Wesley, 1990.

[SpM290] Mike Spivey. **A guide to the zed style option**, December 1990.

[WoJ+97] Jim Woodcock and Jim Davies. **Using Z (Online)**, http://www.comlab.ox.ac.uk/igdp/usingz/text/online/index.html 1997.