Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de una Biblioteca de Análisis Sintáctico y Validación de Problemas de Reescritura

Autor: Juan Pablo Herrero Gómez-Jareño
Tutor(a): Raúl Gutiérrez Gil

Madrid, Junio 2022

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Desarrollo de una Biblioteca de Análisis Sintáctico y Validación de Problemas de Reescritura

Junio 2022

Autor: Juan Pablo Herrero Gómez-Jareño
Tutor: Raúl Gutiérrez Gil
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

La proliferación de diversas competiciones internacionales en el ámbito de la reescritura de términos ha generado diferentes estándares para los formatos de entrada de los diferentes tipos de problemas de reescritura de cada categoría. El objetivo de este trabajo es desarrollar un servicio consistente en un procesador de lenguajes homogéneo para los diferentes problemas de reescritura presentes en estas competiciones, es decir, se creará un analizador léxico, sintáctico y semántico que reconozca los diferentes tipos de problemas utilizados en la base de datos de problemas de terminación TPDB ("Termination Problems Data Base") [1] y la base de datos de problemas de confluencia COPS ("Confluence Problems") [2] Además, se comprobará que la signatura de los sistemas de reescritura son consistentes. Finalmente, esta implementación se extiende a una biblioteca, lo que permitirá que cualquier herramienta pueda importar este servicio y se pueda beneficiar del analizador sintáctico desarrollado a través de una estructura de datos sencilla.

Abstract

The proliferation of several international competitions in the field of term rewriting has generated different standards for the input formats of the different types of rewriting problems in each category. The objective of this work is to develop a service consisting of a homogeneous language processor for the different rewriting problems present in these competitions, that is, a lexical, syntactic and semantic analyzer will be created which recognizes the different types of problems used in the “Termination Problems Data Base” (TPDB) [1] and the “Confluence Problems” database (COPS) [2] In addition, it will be verified that the signature of the rewriting systems are consistent. Finally, this implementation is extended to a library, which will allow any tool to import this service and benefit from the parser developed through a simple data structure.

Tabla de contenidos

1. Introducción **1**
1.1. Contexto . . . 1
1.1.1. Competición de terminación y complejidad . . . 1
1.1.2. Competición de confluencia . . . 2
1.1.3. Categorías en las competiciones . . . 2
1.2. Motivación . . . 3
1.3. Objetivos . . . 3

2. Estado del arte **5**
2.1. Formatos de las competiciones . . . 6
2.1.1. Confluencia . . . 6
2.1.2. Terminación . . . 9

3. Preliminares **13**
3.1. Sistemas de reescritura de términos . . . 14
3.2. Sistemas de reescritura de términos sensible al contexto . . . 14
3.3. Sistemas de reescritura con tipos . . . 14
3.4. Sistemas de reescritura condicional . . . 15
3.5. Sistemas de reescritura con axiomas . . . 15

4. Desarrollo **17**
4.1. Arquitectura del servicio . . . 18
4.2. Análisis Léxico . . . 19
4.2.1. Análisis Léxico con Parsec . . . 20
4.2.2. Definición de los tokens para cada formato . . . 22
4.3. Análisis Sintáctico . . . 24
4.3.1. Implementación del Analizador Sintáctico . . . 26
4.3.1.1. Analizador Sintáctico para los formatos de COPS . . 27
4.3.1.2. Analizador Sintáctico para los formatos de TPDB . . 31
4.3.1.3. Analizador Sintáctico para los formatos de TPDB definidos en XML . . . 34
4.4. Análisis Semántico . . . 40
4.4.1. Creación y exposición del módulo semántico . . . 40
4.4.2. Implementación de las condiciones semánticas . . . 42
4.5. Interfaces del servicio . . . 56
4.5.1. Biblioteca . . . 56

4.5.2. Validador 57
4.6. Repositorios del proyecto 61
4.7. Trabajo futuro 62

5. Conclusiones y Resultados 63
5.1. Resultados 63
5.1.1. Pruebas 64

6. Análisis de impacto 67

Bibliografía 69

Capítulo 1

Introducción

1.1. Contexto

En este trabajo se desarrolla un procesador de lenguajes, más específicamente un analizador, ya que de los dos procesos principales que componen un compilador que son el análisis y la síntesis del programa fuente, únicamente se realiza la parte correspondiente al primero. El análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Es la parte encargada de detectar si el programa fuente está mal formado en cuanto a la sintaxis, o no tiene una semántica consistente [3].

Los lenguajes que el servicio debe reconocer corresponden a los problemas contenidos en la base de datos de problemas de terminación y la base de datos de problemas de confluencia, por sus siglas en inglés, TPDB ("Termination Problems Data Base") y COPS ("Confluence Problems") respectivamente. Estos repositorios pertenecen a las competiciones terminación ("TermCOMP") y de confluencia ("Confluence Competition")

1.1.1. Competición de terminación y complejidad

La **Competición de Terminación y Complejidad (termCOMP)** se centra en el análisis automático de la terminación y el análisis automático de la complejidad para varios tipos de paradigmas de programación, incluyendo categorías para la reescritura de términos, los sistemas de transición de enteros, la programación imperativa, la programación lógica y la programación funcional. En todas las categorías, el concurso también acoge la participación de herramientas que proporcionen resultados certificables. El objetivo de la competición es demostrar la capacidad y los avances de las herramientas existentes en cada una de estas áreas.

El origen de esta competición se remonta a 2003, a un Taller de Terminación organizado en Valencia por Alberto Rubio, donde tras una demostración de herramientas, la comunidad decidió celebrar una competición anual de terminación y recopilar estadísticas para incitar al desarrollo de herramientas y nuevas técnicas de terminación. Desde 2004 la competición, conocida como termCOMP, se

ha estado organizado anualmente con entre diez y veinte herramientas participando en las diferentes categorías en terminación, complejidad y/o certificación. Sus organizadores han sido Claude Marché (de 2004 a 2007), René Thiemann (de 2008 a 2013), Johannes Waldmann (de 2014 a 2017) y Akihisa Yamada (desde 2018) Las últimas competiciones se han ejecutado en directo durante las principales conferencias del campo (en FLoC 2018, FSCD 2017, WST 2016, CADE 2015, VSL 2014, RDP 2013, IJCAR 2012, RTA 2011 y FLoC 2010) [4] [5].

1.1.2. Competición de confluencia

Si hablamos de la **Competición de Confluencia (CoCo)**, de forma similar que en el ámbito de la terminación, durante los últimos años se han ido desarrollando distintas herramientas para probar la confluencia automáticamente y propiedades relacionadas de la variedad de formatos de la reescritura. En las competición se trata de que cada herramienta participante, a la que se le presenta un conjunto de problemas, responda, en un tiempo determinado, si el sistema de reescritura de términos dado es confluente o no. Gana la herramienta que responda correctamente el mayor número de problemas. Estas herramientas compiten anualmente y son ejecutadas en los problemas de confluencia que encontramos en COPS.

Similar a TPDB en termCOMP, COPS es una base de datos en linea para problemas de confluencia en reescritura de términos, fue creada en 2012 para facilitar la organización de la Competición de Confluencia (CoCo) y el desarrollo de las herramientas de confluencia. Junto con CoCoWeb, portal para herramientas que participan en la competición de confluencia anual, dan soporte a investigadores interesados en desarrollar herramientas para la confluencia y propiedades relacionadas de los sistemas de reescritura [6] [7].

1.1.3. Categorías en las competiciones

En TPDB, la ejecución de los conjuntos de pruebas que ejecutan las herramientas de las que hemos hablado son agrupadas según el modelo computacional subyacente (reescritura o programación) y el objetivo del análisis (terminación o complejidad). Esta organización da como resultado los siguientes tres metacategorías desde "termCOMP 2014": terminación de reescritura, terminación de programas y análisis de complejidad. De las tres, es la primera (terminación de reescritura) la que nos atañe e interesa en este proyecto.

Hay muchos tipos diferentes de reescritura de términos y estrategias para aplicar reglas de reescritura. Muchos de ellos tienen sus propias categorías. En los sistemas de reescritura de términos estándar, existen categorías para la reescritura simple (TRS Standard), la reescritura relativa (TRS Relative), la reescritura de teorías de ecuaciones de módulo (TRS Equational), la reescritura de términos condicionales (TRS Conditional), la reescritura sensible al contexto (TRS Context Sensitive), reescritura Innermost (TRS Innermost) y reescritura Outermost (TRS Outermost). También hay una categoría para sistemas de reescritura de orden superior (HRS). Con respecto a los sistemas de reescritura de cadenas,

existen categorías para la reescritura simple (SRS Standard), la reescritura relativa (SRS Relative) y la reescritura cíclica (Cycle Rewrite). Además, hay varias de estas categorías tienen una variante certificada especial [4].

Por otra parte en la Competición de Confluencia (CoCo) encontramos actualmente trece categorías diferentes que se describen a continuación. La categoría COM que ocupa los problemas de conmutación. La categoría CSR los problemas sensibles al contexto. CTRS, categoría que se ocupa de la confluencia de los sistemas de reescritura de términos condicionales. Esta categoría la encontramos en su versión sin certificar y en su versión certificada que es la llamada CPF-CTRS. La categoría GCR para la confluencia base de sistemas de reescritura de términos con tipos (MSTRS). La categoría HRS dedicada a la confluencia de sistemas de reescritura de orden superior. La categoría INF que trata los problemas de infactibilidad. NFP que se ocupa de la propiedad de forma normal de los sistemas de reescritura de primer orden. SRS de la confluencia de los sistemas de reescritura de cadenas. La categoría TRS es la más antigua en CoCo y se ocupa de la confluencia de sistemas de reescritura de primer orden. En esta categoría también contamos con la versión sin certificar y la certificada correspondiente CPF-TRS. La categoría UNC se ocupa de la propiedad de "formas normales únicas con respecto a la conversión" de los sistemas de reescritura de primer orden. Y finalmente, la categoría UNR trata las "formas normales únicas con respecto a la propiedad de reducción" de los sistemas de reescritura de primer orden [7].

1.2. Motivación

Dada la gran variedad de categorías es llamativo el hecho de que actualmente no exista ningún servicio común que ofrezca, de forma externa y homogénea, la funcionalidad de lectura y análisis de los distintos formatos de problemas de entrada. A raíz de lo que se ha explicado, es fácil darse cuenta que ambas competiciones comparten tipos de problemas o son muy similares. Buscamos una estructura homogéneas para esos formatos.

Es por ello por lo que este proyecto resulta interesante, es una clara mejora a las herramientas que se desarrollan para este tipo de competiciones, ofrecer un analizador a modo de biblioteca que pueda reconocer los diversos formatos de problemas y sea fácilmente implementable por todas ellas.

1.3. Objetivos

Este trabajo consta de varios objetivos. El primero de ellos es adquirir los conocimientos necesarios sobre el lenguaje de programación Haskell y la biblioteca Parsec, sobre los que el servicio va a ser implementado. Otro objetivo es el de comprender el contexto es en que se desarrolla este trabajo, esto es conocer los formatos de problemas que se pretenden procesar y el ámbito al que pertenecen.

El último objetivo de este trabajo es elaborar un servicio capaz de analizar los distintos tipos de problemas de entrada que encontramos en TPDB y COPS. Para exponerlo a modo de aplicación, como validador, y de librería. Para que pueda

ser utilizado por otras herramientas que comparten los tipos de formatos de problemas que utilizan.

Capítulo 2

Estado del arte

En las competiciones de terminación y confluencia, encontramos muchas herramientas que concursan anualmente en las distintas categorías, sin embargo, actualmente no existe ningún servicio común a todas ellas que les ofrezca la funcionalidad de lectura y análisis de los distintos formatos de problemas de entrada. Hasta el momento, es cada herramienta la que implementa dicha funcionalidad. De todas ellas destacamos tres que resultan relevantes en el proyecto, son MU-TERM, CONFident e infChecker. Además, existen otras muchas herramientas que participan en estas competiciones. En las últimas ediciones de las competiciones podemos encontrar herramientas como MultumNonMulta, Wanda, SOL, matchbox, SizeChangeTool, AProVE, TTT, NaTT, o MU-TERM, que participaron en termCOMP. O CSI, FORT-h, FORTify, CONFident, infChecker, Toma, Hakusan, CoLL, CeTA, CO3, ACP, AGCP o NaTT en CoCo. No es raro que estas estén escritas en lenguajes funcionales, como es el caso de matchbox, TTT, MU-TERM, CONFident o infChecker. Por esto, tener una biblioteca que aúne el reconocimiento de formatos, es una buena idea.

MU-TERM se trata de una herramienta capaz de verificar automáticamente las propiedades de terminación en las distintas variantes de sistemas de reescritura de términos (TRS), incluyendo sistemas de reescritura condicionales, sensibles al contexto, TRS ecuacionales y de clasificación ordenada. Esta herramienta sigue una aproximación unificada, basado en la lógica, para describir los cálculos de reescritura [8]. **CONFident** es capaz de comprobar la existencia de confluencia en sistemas basados en la reescritura, por medio de su representación lógica [9]. Por último, **infChecker** es una herramienta que implementa un marco de trabajo pensado para verificar la infactibilidad. Verificar la infactibilidad es importante en el análisis de las propiedades computacionales de los sistemas de reescritura, como la confluencia o la terminación [10].

Estas tres herramientas concursan en sus competiciones correspondientes. Desde 2014, MU-TERM ha demostrado ser la herramienta más potente para probar la terminación operacional de la reescritura condicional y la terminación de la reescritura sensible al contexto, ganando cada año la subcategoría correspondiente en la Competición Internacional de Herramientas de Terminación. [8]. Por otro lado, CONFident e infChecker son herramientas que participan en la Com-

petición Internacional de Confluencia. infChecker lo hace desde 2019, cuando se incluyó por primera vez la categoría de infactibilidad (INF).

Dichas herramientas resultan relevantes para el proyecto debido a que se dispone de acceso a ellas y cuentan con una base de analizador sintáctico que puede ser extraído y aprovechado con el objetivo de mejorarlo para crear un procesador de lenguajes homogéneo. También es importante destacar que el proyecto se desarrolla en Haskell, lenguaje de en el que se encuentran escritas estas herramientas que nos sirven de base.

Haskell es un lenguaje de programación puramente funcional, esto quiere decir que la forma de programar en Haskell va enfocada a funciones que toman valores inmutables como entrada y producen nuevos valores como salida. La idea central detrás de la programación funcional es que dadas las mismas entradas, estas funciones siempre devuelven los mismos resultados.

A diferencia de los lenguajes de programación imperativos, Haskell no se basa en los cambios de estado mediante la mutación de variables, elimina el énfasis al código que modifica los datos. Además de no modificar datos, las funciones de Haskell generalmente no se comunican con el mundo externo; son las llamadas 'funciones puras'.

En Haskell se busca eliminar la mutabilidad o efectos secundarios, de hecho, se hace una fuerte distinción entre el código puro y las partes de los programas que leen o escriben archivos, se comunican a través de conexiones de red o tiene cualquier otro tipo de interacción hacia el exterior. El objetivo de esto es que sea más fácil organizar, razonar y probar nuestros programas [11].

2.1. Formatos de las competiciones

Como se ha explicado en el apartado de 1.1, Contexto , los tipos de problemas que vamos a tratar pertenecen a varias categorías de dos competiciones del ámbito de la reescritura de términos, la de terminación cuyos problemas encontramos en TPDB y la de confluencia, con los problemas que están recogidos en COPS. En cada categoría de cada competición se utilizan determinados formatos como se explica a continuación.

2.1.1. Confluencia

El primer formato que encontramos en COPS es el **TRS** ("term rewrite systems") que es el formato básico, ya que casi todos los demás derivan de este. Esta compuesto por un bloque de definición de variables, un bloque de reglas y uno de comentarios. Este formato es una simplificación del TRS que podemos encontrar en TPDB. Este formato es el que se utiliza en las categorías CPF-TRS, SRS, TRS, UNC y UNR..

```
(VAR x)
(RULES
    d(x,x) -> x
    f(x) -> d(x,f(x))
```

```
    a -> f(a)
)
(COMMENT
TRS example
)
```

Listing 2.1: Ejemplo de problema con formato TRS

Este formato cuenta con una posible extensión (**eTRS**) en la que se agrega una declaración de firma que especifica el conjunto de símbolos de función y sus aridades. En este formato, cada símbolo que aparece en las reglas debe declararse como un símbolo de función o una variable. Lo utilizan en las categorías UNC y UNR de la competición.

```
(VAR x y)
(SIG (f 2) (a 0) (b 0) (h 1))
(RULES
    f(x,x) -> x
    f(a,y) -> f(y,b)
)
```

Listing 2.2: Ejemplo de problema con formato eTRS

El siguiente formato que tenemos es **CTRS** (çonditional rewrite systems"), el cual es similar al TRS pero con la posibilidad de añadir condiciones en el bloque de reglas, y encontramos un nuevo bloque en el que se especifica el tipo de condición. Este formato lo encontramos en las categorías CPF-CTRS y CTRS.

```
(CONDITIONTYPE ORIENTED)
(VAR x)
(RULES
  not(x) -> false | x == true
  not(x) -> true  | x == false
)
```

Listing 2.3: Ejemplo de problema con formato CTRS

Otro formato que extiende el TRS básico es el **CSTRS** (çontext-sensitive term rewrite systems") en el cual aparece un nuevo bloque en el que se pueden declarar símbolos de función, siempre que estén también en el bloque de reglas, junto con una lista de enteros positivos estrictamente crecientes que corresponderán a las posiciones activas de los argumentos de dicha función, en las reglas. Es utilizado en las categoría CSR.

```
(VAR L X)
(REPLACEMENT-MAP
   (cons 1)
)
(RULES
   incr(nil) -> nil
   incr(cons(X,L)) -> cons(s(X),incr(L))
   adx(nil) -> nil
   adx(cons(X,L)) -> incr(cons(X,adx(L)))
   nats -> adx(zeros)
   zeros -> cons(0,zeros)
   head(cons(X,L)) -> X
```

```
   tail(cons(X,L)) -> L
)
```

Listing 2.4: Ejemplo de problema con formato CSTRS

Encontramos también el formato **CSCTRS** (çontext-sensitive conditional term rewrite systems") que extiende a los dos últimos (CTRS y CSTRS) ya que es una combinación de ambos. Este, al igual que el anterior, también lo utiliza la categoría CSR de CoCo.

```
(CONDITIONTYPE ORIENTED)
(VAR l x y)
(REPLACEMENT-MAP
   (cons 2) (le 1)
)
(RULES
   le(0,s(x)) -> true
   le(x,0) -> false
   le(s(x),s(y)) -> le(x,y)
   min(cons(x,nil)) -> x
   min(cons(x,l)) -> x | le(x,min(l)) == true
   min(cons(x,l)) -> min(l) | le(x,min(l)) == false
   min(cons(x,l)) -> min(l) | min(l) == x
)
```

Listing 2.5: Ejemplo de problema con formato CSCTRS

Finalmente tenemos unos últimos formatos que, por falta de tiempo, no abarcaremos en este trabajo. El formato **HRS** (“higher-order rewrite systems”) sigue el mismo estilo que el resto de formatos de “primer orden”, agregando declaraciones de tipo a variables y símbolos de función, así como sintaxis para la abstracción y aplicación [2], y es el que se utiliza en la categoría de CoCo que lleva el mismo nombre. El formato **MSTRS** (“many-sorted term rewrite systems”), por su parte, se utiliza en la categoría llamada GCR para probar la confluencia base de los sistemas de reescritura de términos de ordenación múltiple. Existen, además, un par de categorías (COM y INF) que utilizan formatos especiales que no son mas que modificaciones de los formatos que acabamos de explicar. En el caso de COM el formato que se utiliza deriva de TRS y en el caso de INF los formatos de los que se basa son el CTRS y el TRS.

En el portal de COPS encontramos especificados los diferentes formatos de problemas que podemos encontrar en la competición, junto con sus correspondientes gramáticas [2] .

Debido a que los formatos que vamos a reconocer son, como se ha explicado, muy similares. Derivan unos de otros. Lo primero que se ha hecho ha sido juntar todas las gramáticas en una única especificación, para poder trabajar fácilmente con ella. Como se explicará en el capítulo del desarrollo del servicio, será durante el análisis semántico donde se reconozca las características de cada formato particular. No perdemos información acerca del formato al que corresponde cada problema ya que no hay conflictos entre las particularidades de cada uno. Y no será necesario construir varios analizadores sintácticos distintos cuando estos formatos comparten la mayor parte de su gramática.

```
spec      ::= e | ( decl ) spec
decl      ::= (CONDITIONTYPE ctype) [(VAR idlist)] (REPLACEMENT-MAP cslist) (SIG funlist
    ) (RULES rulelist) [(COMMENT string)]
idlist   ::= e | id idlist
rulelist ::= e | rule rulelist
rule     ::= term -> term | term -> term | condlist
ctype    ::= SEMI-EQUATIONAL | JOIN | ORIENTED
condlist ::= cond | cond, condlist
cond     ::= term == term
term     ::= id | id() | id(termlist)
termlist ::= term | term, termlist
funlist  ::= e | fun funlist
fun      ::= (id int)
cslist   ::= csfun | csfun cslist
csfun    ::= (id intlist)
intlist  ::= e | int, intlist
```

Listing 2.6: Gramática para los formatos de problemas de COPS

2.1.2. Terminación

En el portal de TPDB solo encontramos las especificaciones de las gramáticas de los formatos **TRS** y **SRS** ("Strings Rewriting Systems") [1], que corresponden a las categorías con las que arrancó el concurso. A continuación se muestra la especificación de dicho formato TRS, utilizado, como decimos, en las distintas categorías desde que se inició la competición de terminación.

```
spec         ::= e | ( decl ) spec
decl         ::= VAR   idlist | THEORY   listofthdecl | RULES   listofrules
                  | STRATEGY   strategydecl | id   anylist
anylist      ::= e | id   anylist | string   anylist | ( anylist )   anylist | , anylist
idlist       ::= e | id   idlist
listofthdecl ::= e | (   thdecl   )   listofthdecl
thdecl       ::= id   idlist | EQUATIONS   eqlist
eqlist       ::= e | equation   eqlist
equation     ::= term   ==   term
listofrules  ::= e | rule   listofrules
rule         ::= term   ->   term | term   ->   term   |   condlist
                          | term   ->=   term | term   ->=   term   |   condlist
condlist     ::= cond | cond   ,   condlist
cond         ::= term   ->   term | term   -><-   term
term         ::= id | id   (   ) | id   (   termlist   )
termlist     ::= term | term   ,   termlist
strategydecl ::= INNERMOST | OUTERMOST | CONTEXTSENSITIVE   csstratlist
csstratlist  ::= e | ( id   intlist )   csstratlist
intlist      ::= e | int   intlist
```

Listing 2.7: Gramática para los formatos de problemas de TPDB

Aquí el formato TRS abarca directamente los sistemas de reescritura condicionales y sensibles al contexto que veíamos en los formatos CTRS, CSTRS y CSTRS cuyas gramáticas se encuentran por separado en COPS. Se puede observar algunas diferencias con anterior gramática, por ejemplo, en la especificación que hace de estos sistemas de reescritura condicionales y sensibles al contexto. Vemos a demás que esta gramática especifica y reconoce los sistemas de reescritura de términos ecuacionales, que no encontrábamos para los problemas de COPS.

El formato SRS, por su parte, es una simplificación de TRS en el que las funciones que aparecen solo pueden ser unarias, es decir, solo pueden tener un argumento. Este está en desuso y no se va a implementar en el servicio.

```
(VAR x y h i u v)
(THEORY (AC or))
(THEORY (C eq))
(RULES
  eq(0, 0) -> true
  eq(0, s(x)) -> false
  eq(s(x), 0) -> false
  eq(s(x), s(y)) -> eq(x, y)
  or(true, y) -> true
  or(false, y) -> y
  union(empty, h) -> h
  union(edge(x, y, i), h) -> edge(x, y, union(i, h))
  reach(x, y, empty, h) -> false
  reach(x, y, edge(u, v, i), h) -> if_reach_1(eq(x, u), x, y, edge(u, v, i), h)
  if_reach_1(true, x, y, edge(u, v, i), h) -> if_reach_2(eq(y, v), x, y, edge(u, v, i),
      h)
  if_reach_1(false, x, y, edge(u, v, i), h) -> reach(x, y, i, edge(u, v, h))
  if_reach_2(true, x, y, edge(u, v, i), h) -> true
  if_reach_2(false, x, y, edge(u, v, i), h) -> or(reach(x, y, i, h), reach(v, y, union(i
      , h), empty))
)
```

Listing 2.8: Ejemplo de problema con formato TRS

Durante el desarrollo y evolución de la competición se han ido introduciendo más formatos y categorías y a partir de la versión 7 (año 2009) [1] se introdujo la definición de estos formatos sobre el lenguaje de marcado XML. Con esto tenemos la versión estándar o tradicional y su versión en **XML**. El formato **HRS** que lo encontramos definido sobre XML, al igual que su homólogo en la Competición de Confluencia, no se va a poder incluir en el servicio.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="../../xml/xtcHTML.xsl"?>
<problem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
    noNamespaceSchemaLocation="../../xml/xtc.xsd" type="termination">
    <trs>
        <rules>
            <rule>
                <lhs>
                    <funapp>
                        <name>f</name>
                        <arg>
                            <funapp>
                                <name>s</name>
                                <arg>
                                    <var>x</var>
                                </arg>
                            </funapp>
                        </arg>
                        <arg>
                            <var>y</var>
                        </arg>
                        <arg>
                            <var>y</var>
                        </arg>
                    </funapp>
                </lhs>
                <rhs>
```

```
                    <funapp>
                        <name>f</name>
                        <arg>
                            <var>y</var>
                        </arg>
                        <arg>
                            <var>x</var>
                        </arg>
                        <arg>
                            <funapp>
                                <name>s</name>
                                <arg>
                                    <var>x</var>
                                </arg>
                            </funapp>
                        </arg>
                    </funapp>
                </rhs>
            </rule>
        </rules>
        <signature>
            <funcsym>
                <name>f</name>
                <arity>3</arity>
            </funcsym>
            <funcsym>
                <name>s</name>
                <arity>1</arity>
            </funcsym>
        </signature>
    </trs>
    <strategy>FULL</strategy>
    <metainformation>
        <originalfilename>./TRS/AG01/#3.29.trs</originalfilename>
    </metainformation>
</problem>
```

Listing 2.9: Ejemplo de problema con formato TRS-XML

A diferencia que para el formato TRS tradicional en TPDB no tenemos definida la gramática de su versión en XML. Si que encontramos su correspondiente XML Schema [1]. A partir de este XML Schema se ha construido la siguiente gramática.

```
spec         ::= e | <problem> predecl </problem> spec
predecl      ::= <trs> decl </trs> | <strategy> strategydecl </strategy>
decl         ::= <rules> listofrules </rules> | <signature> signlistdecl </signature>
               | ctypedecl | commentdecl
signlistdecl ::= e | <funcsym> signdecl </funcsym> signlistdecl
signdecl     ::= standarSig | standarSig thdecl | standarSig rpmapdecl
standarSig   ::= <name> id </name> <arity> int </arity>
thdecl       ::= e | <theory> th </theory>
ctypedecl        ::= e | <conditiontype> ctype </conditiontype>
ctype        ::= JOIN | ORIENTED | OTHER
th           ::= A | C | AC
listofrules  ::= e | <rule> rule </rule> listofrules | <relrules> relRulesList </
    relrules>
relRulesList ::= e | <rule> rule </rule> relRulesList
rule         ::= <lhs> term </lhs> <rhs> term </rhs>
               | <lhs> term </lhs> <rhs> term </rhs> <conditions> condlist </conditions>
condlist     ::= <condition> cond </condition> | <condition> cond </condition> condlist
cond         ::= <lhs> term </lhs> <rhs> term </rhs>
term         ::= <var> id </var> | <funapp> <name> id </name> termlist </funapp>
termlist     ::= e | <arg> term </arg> | <arg> term </arg> termlist
```

```
strategydecl ::= INNERMOST | OUTERMOST | FULL
rpmapdecl    ::= <replacementmap/> | <replacementmap>rpmap</replacementmap>
rpmap        ::= e | <entry> int </entry> rpmap
commentdecl  ::= e | <comment> string </comment>
```

Listing 2.10: Gramática para los formatos de problemas de TPDB definidos en XML

Capítulo 3

Preliminares

La reescritura de términos es una rama de la informática teórica que combina elementos de la lógica, el álgebra universal, la demostración automática de teoremas y la programación funcional. Su fundamento es la lógica ecuacional. Lo que distingue la reescritura de términos de la lógica ecuacional es que las ecuaciones son usadas se utilizan como reglas de sustitución dirigidas, por ejemplo la parte izquierda puede ser remplazada por la parte derecha, pero no vice versa. Esto constituye un modelo computacional Turing-completo, que está muy cerca de la programación funcional. Esto tiene aplicaciones en álgebra, teoría de la recursión, ingeniería de software y lenguajes de programación. En general, la reescritura de términos aplica en cualquier contexto donde sean necesarios métodos eficientes para razonar con ecuaciones. Dos de las propiedades más importantes de los sistemas de reescritura de términos son la terminación y la confluencia.

Como decimos la terminación es una de las propiedades importantes de los sistemas de reescritura de términos y trata de probar si ocurre que tras un numero finito de aplicaciones de reglas siempre llegamos a una a una expresión a la que no se aplican más reglas. A esa expresión se le llama entonces forma normal. Sin embargo, no es del todo trivial de demostrar. Además, la no terminación no siempre tiene que ser causada por una sola regla; también puede resultar de la interacción de varias reglas.

Si hay diferentes formas de aplicar reglas a un término t dado, lo que lleva a diferentes términos derivados t1 y t2, lo que la confluencia pretende comprobar es si pueden unirse t1 y t2, es decir, si siempre podemos encontrar un término común s que se pueda alcanzar tanto desde t1 como desde t2 a través de la aplicación de reglas. La confluencia de las reglas significa que el resultado de un cálculo es independiente de la estrategia de evaluación.

Para un sistema de reescritura de términos finitos, se puede encontrar una forma normal de un término dado mediante una simple búsqueda en profundidad. Si el sistema es también confluente, las formas normales son únicas, lo que hace que el problema verbal de la correspondiente teoría ecuacional sea decidible. Desafortunadamente, la terminación es una propiedad indecidible de los

sistemas de reescritura de términos. Esto es cierto incluso si solo se permiten símbolos de función unarios en las reglas, o solo una regla de reescritura. Sin embargo, en el caso restringido de los sistemas de reescritura básicos, es decir, los sistemas de reescritura cuyas reglas no deben contener variables, la terminación se vuelve decidible [12].

3.1. Sistemas de reescritura de términos

Una signatura Σ es un conjunto de símbolos de función $f \in \Sigma$, donde todo símbolo f de la signatura tiene asociado un número entero no negativo n que representa la aridad de f.

Dada una signatura Σ y un conjunto de variables X tal que $\Sigma \cap X = \emptyset$, el conjunto de Σ-términos sobre X se define inductivamente como:

- $X \subseteq T(\Sigma, X)$.
- para todo símbolo $f \in \Sigma$ cuya aridad $n \geq 0$ y $t_1, \ldots, t_n \in T(\Sigma, X)$, tenemos que $f(t_1, \ldots, t_n) \in T(\Sigma, X)$

Para los términos de aridad 0, también llamados constantes, escribiremos f en lugar de $f()$.

Una regla de reescritura en un par ordenado (ℓ, r) que se escribe $\ell \to r$, donde $\ell, r \in T(\Sigma, X)$, $\ell \notin X$ y las variables que aparecen en r deben aparecer en ℓ.

Un sistema de reescritura es un par (Σ, R), donde Σ es una signatura y R es un conjunto de reglas de reescritura.

3.2. Sistemas de reescritura de términos sensible al contexto

Un sistema de reescritura de términos sensible al contexto es un par $(\mathcal{R}, \mu)$ que consiste en un sistema de reescritura $\mathcal{R}$ sobre una signatura Σ y una función de reemplazamiento μ. La función de remplazamiento μ en una proyección entre símbolos de función $f \in \Sigma$ y conjuntos de números naturales que satisfacen $\mu(f) \subseteq \{1, \ldots, n\}$, donde n es la aridad de f.

3.3. Sistemas de reescritura con tipos

En un sistema de reescritura con tipos (Σ, R), partimos de un conjunto no vacío de de tipos S y a cada símbolo de función de la signatura Σ le asociamos un tipado $f : s_1 \times \cdots \times s_n \to s$, donde $s_1, \ldots, s_n, s \in S$ y n es la aridad de f.

Dada una signatura Σ y un conjunto de variables X tal que cada variable tiene asociado un tipo y $\Sigma \cap X = \emptyset$, el conjunto de Σ-términos sobre X se define inductivamente como:

- $X \subseteq T(\Sigma, X)$.

- para todo símbolo $f : s_1 \times \cdots \times s_n \to s \in \Sigma$ y $t_1, \ldots, t_n \in T(\Sigma, X)$, tenemos que $f(t_1, \ldots, t_n) \in T(\Sigma, X)$ donde t_i es de tipo $s_i \in S$ y $1 \leq i \leq n$.

Toda regla de reescritura $\ell \to r \in R$ debe cumplir que ℓ y r tengan el mismo tipo.

3.4. Sistemas de reescritura condicional

Un sistema de reescritura condicional (Σ, R) es un sistema de reescritura donde las reglas tienen la forma $\ell \to r \mid c$, donde c es una lista de condiciones de la forma: $u\ R\ v$, siendo R es una relación entre términos. En este trabajo solo consideramos los casos donde $R \in \{=, \to, \to\leftarrow\}$.

3.5. Sistemas de reescritura con axiomas

Un sistema de reescritura con axiomas (Σ, E, R) es un sistema de reescritura donde E contiene axiomas asociados a los símbolos de la signatura Σ. En nuestro caso, los axiomas considerados son la asociatividad y la conmutatividad.

Capítulo 4

Desarrollo

La implementación del proyecto se ha realizado íntegramente en Haskell. Además, se han utilizado la biblioteca Parsec y otras herramientas como Cabal, Visual Studio Code o Git para desarrollarlo.

Cabal, por sus siglas en inglés 'Common Architecture for Building Applications and Libraries', es un sistema para construir y empaquetar bibliotecas y programas de Haskell que provee una interfaz común para poder realizar estas tareas fácilmente. Así mismo, Visual Studio Code es un editor de código que ha permitido desarrollar y editar el código de forma sencilla y cómoda. Y Git, un sistema de control de versiones que se ha usado para mantener el registro de las modificaciones realizadas durante la evolución del servicio, así como, para sincronizarlo con el repositorio creado para hacerlo accesible.

Muy importante para el desarrollo de este trabajo ha sido Parsec, una biblioteca combinadora de analizadores monádicos para Haskell. En la programación funcional, un enfoque para construir analizadores descendentes recursivos es modelar analizadores como funciones y definir funciones de orden superior (o combinadores) que implementan construcciones gramaticales. Dichos analizadores forman una instancia de una mónada y proporcionan un método rápido y fácil de construir analizadores funcionales [13]. A diferencia de los generadores de analizadores que ofrecen un conjunto fijo de combinadores para expresar gramáticas, como los combinadores que ofrece Parsec se manipulan como valores de primera clase, estos se pueden combinar para definir nuevos combinadores que se ajusten al dominio de la aplicación. Otra ventaja es que nos permite utilizar un solo lenguaje, evitando tener que integrar de diferentes herramientas y lenguajes [14].

Un concepto importante que se acaba de mencionar, y que utilizamos a lo largo de todo el desarrollo de este proyecto, es el concepto de mónada. Las mónadas son omnipresentes en el código Haskell, nos permiten capturar, en una clase de tipos de Haskell, las nociones de encadenamiento e inyección junto con los tipos que queremos que contengan [11]. Es decir, nos permiten representar cálculos, definidos como una secuencia de pasos, y componer funciones con tipos incompatibles encapsulándolos en una clase de tipos que llamamos monádicos ya

que cuentan con ciertas características que nos permiten hacer esto. De manera conceptual podemos entenderlas como un objeto que transporta datos entre unidades funcionales que lo van transformando en cada paso.

Además de que los combinadores de Parsec están basados en este concepto, se hace uso constante de él y nos va a ser especialmente útil en muchos casos como a la hora de manejar las excepciones, ya que nos permite cortocircuitar una cadena de cálculos y propagar el error, o para propagar un estado entre funciones, o en la interacción con la entrada y salida del servicio.

4.1. Arquitectura del servicio

Partiendo del analizador base del que se dispone, la arquitectura del servicio se desarrolla con el objetivo de que sea fácilmente extensible a la hora de introducir formatos nuevos que quieran ser reconocidos. Gran parte del código es compartido y puede ser fácilmente reutilizado entre formatos, ya que estos comparten estructuras similares o iguales en algunos casos.

Las características propias de cada formato se implementan en módulos propios separados, que pueden ser llamados desde la aplicación principal. Esto lo podemos hacer así, en parte, gracias al uso de la biblioteca Parsec. Precisamente Parsec nos permite crear analizadores para las distintas fases del análisis y estos se manejan como ciudadanos de primera clase dentro del lenguaje, es decir, los propios analizadores los podemos meter en listas, pasar como parámetros y devolver como valores, lo que nos otorga la posibilidad de desacoplarlos fácilmente [15].

El resultado lo encontramos en la figura 4.1 donde podemos ver los distintos módulos que conforman el analizador. Los módulos de las interfaces, así como los correspondientes a las acciones semánticas y la gramática son comunes, por otro lado, los módulos correspondientes a las definiciones de tokens de cada lenguaje y a su sintaxis son propios de cada formato y no se comparten.

Esta arquitectura es resultado de que cada formato tenga sus propios módulos para el análisis léxico y sintáctico, que son específicos y trabajan sobre el fichero fuente. Sin embargo, el resto de partes de la aplicación trabajan sobre la estructura de datos definida en el módulo “Parser.Grammar”, que se explica más adelante, y que contiene la información tras dicho análisis sintáctico.

En el fondo se sigue una estructura muy similar a la arquitectura que comúnmente tiene todo procesador de lenguajes [3], que podemos ver en la figura 4.2. Pero, como se explica en apartados posteriores, por simplicidad y gracias al uso de Parsec las partes correspondientes al análisis léxico y sintáctico se entrelazan en este servicio. Si que hay una distinción clara entre la fase correspondiente al análisis sintáctico y la del semántico.

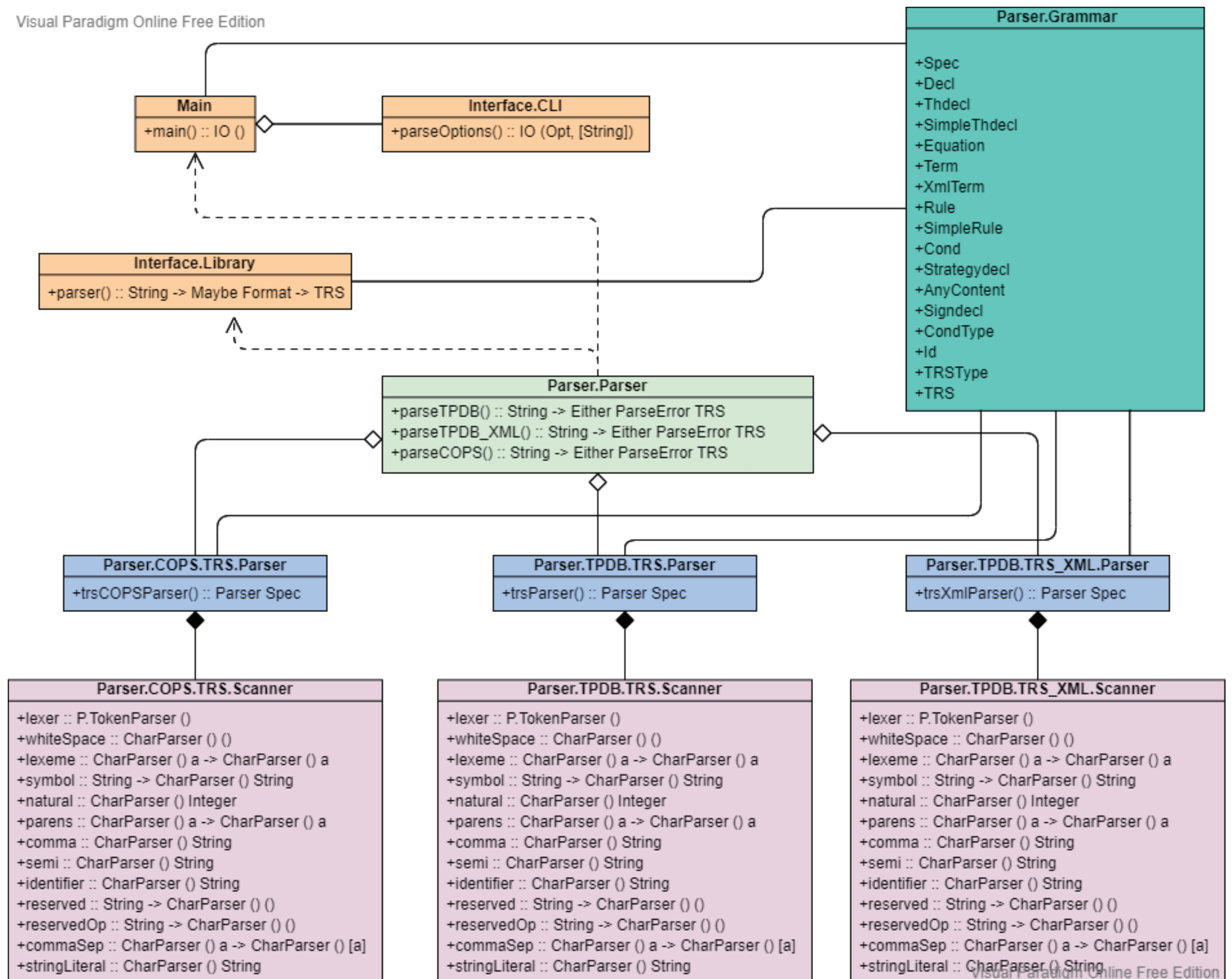


Figura 4.1: Arquitectura del servicio

4.2. Análisis Léxico

A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token que pasa a la fase siguiente, el análisis sintáctico. Los token esta formados por el nombre del token y un atributo del token. El primer componente es un símbolo abstracto, que es el que se utiliza durante el análisis sintáctico, y el segundo componente apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se utiliza en el análisis semántico [3].

Parsec proporciona las herramientas necesarias para implementar dicho análisis léxico. Como decimos, normalmente los analizadores sintácticos toman en su entrada una lista de tokens que les proporciona el analizador léxico, el cual, además, suele filtrar los espacios en blanco y los comentarios, sin embargo, Par-

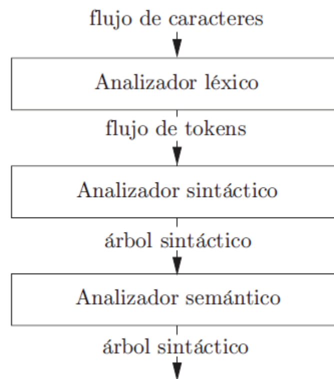


Figura 4.2: Estructura parcial de un compilador

sec ofrece otra aproximación para realizar esta tarea. Permite fusionar el análisis léxico y sintáctico en una misma especificación. Se pueden usar los combinadores de análisis, que proporciona la biblioteca, para lidiar con el análisis léxico también [15].

Para el servicio se utiliza esta aproximación ya que permite la implementación del análisis léxico de forma sencilla y, para los objetivos de este trabajo, realmente no se necesita desarrollar analizadores léxicos más complejos. Los formatos de los lenguajes a procesar no lo requieren. Esto no significa que los módulos, en los que son definidas para cada formato las listas de tokens a reconocer, no puedan ser reutilizados en caso de que otros compartan los mismos tokens.

4.2.1. Análisis Léxico con Parsec

Entre los módulos de Parsec encontramos el módulo estándar “ParsecToken”, el cual se ocupa de los problemas que normalmente se delegan a un escáner separado: espacios en blanco, comentarios, identificadores, palabras reservadas, números, cadenas, entre otros. Para poder definir las características de cada uno de nuestros formatos el módulo exporta una función llamada “makeTokenParser” que obtiene dichas características como argumento. Estas características se las vamos pasar a la función a través del tipo “LanguageDef”.

```
1  data LanguageDef st
2      = LanguageDef
3      { commentStart :: String
4      , commentEnd :: String
5      , commentLine :: String
6      , nestedComments :: Bool
7      , identStart :: CharParser st Char
8      , identLetter :: CharParser st Char
9      , opStart :: CharParser st Char
10     , opLetter :: CharParser st Char
11     , reservedNames :: [String]
12     , reservedOpNames:: [String]
13     , caseSensitive :: Bool
14     }
```

Listing 4.1: Definición tipo de dato ’LanguageDef’

```
srsDef :: P.LanguageDef st
srsDef = emptyDef {
     P.commentStart = ""
   , P.commentEnd = ""
   , P.commentLine = ""
   , P.nestedComments =
   , P.identStart =
   , P.identLetter =
   , P.opStart =
   , P.opLetter =
   , P.reservedNames= []
   , P.reservedOpNames = []
   , P.caseSensitive =
   }
```

Listing 4.2: Creación de un 'LanguageDef' vacío

La función “makeTokenParser” crea y devuelve un registro llamado “TokenParser”. Este contiene un conjunto de analizadores léxicos, que son los que se van a utilizar para reconocer los lexemas de los ficheros fuente [15]. Como se puede ver a continuación, importamos varias de las funciones de análisis léxico que ese registro ofrece. Lo hacemos en cada uno de los módulos "Scanner", donde se encuentra definido lo relativo al léxico de cada formato.

```
-- | Create lexer using 'srsDef' definition.
lexer :: P.TokenParser ()
lexer = P.makeTokenParser $ srsDef
```

Listing 4.3: Función 'makeTokenParser'

```
-- | white Space
whiteSpace :: CharParser () ()
whiteSpace= P.whiteSpace lexer

-- | Lexeme
lexeme :: CharParser () a -> CharParser () a
lexeme = P.lexeme lexer

-- | Symbol
symbol :: String -> CharParser () String
symbol = P.symbol lexer

-- | Natural
natural :: CharParser () Integer
natural = P.natural lexer

-- | Parens
parens :: CharParser () a -> CharParser () a
parens = P.parens lexer

-- | Brackets
brackets :: CharParser () a -> CharParser () a
brackets = P.brackets lexer

-- | Comma
comma :: CharParser () String
comma = P.comma lexer

-- | Semi
semi :: CharParser () String
semi = P.semi lexer
```

```
32
33 -- | Identifier
34 identifier :: CharParser () String
35 identifier= P.identifier lexer
36
37 -- | Reserved
38 reserved :: String -> CharParser () ()
39 reserved = P.reserved lexer
40
41 -- | Reserved option
42 reservedOp :: String -> CharParser () ()
43 reservedOp= P.reservedOp lexer
44
45 -- | Separated by comma
46 commaSep :: CharParser () a -> CharParser () [a]
47 commaSep = P.commaSep lexer
48
49 -- | String literal
50 stringLiteral :: CharParser () String
51 stringLiteral = P.stringLiteral lexer
```

Listing 4.4: Analizadores léxicos importados

4.2.2. Definición de los tokens para cada formato

Hay varios tokens diferentes que necesitamos reconocer. Para el servicio se crean tres especificaciones de tokens distintas. Una para el TRS de TPDB, otra para su versión en XML y la última para los formatos de COPS.

Para empezar, según la especificación del TRS en TPDB tenemos que un identificador será cualquier secuencia de caracteres no vacía. A excepción de paréntesis, dobles comillas y coma, que también son tokens, y las secuencias especiales correspondientes a los operadores que son '->', '==', '->=', '-><-' y '|'. Además, contamos con las palabras reservadas CONTEXTSENSITIVE, EQUATIONS, INNERMOST, OUTERMOST, RULES, STRATEGY, THEORY y VAR. Tenemos únicamente dos tipos, el tipo cadena ('string') que será cualquier secuencia de caracteres entre dobles comillas y el tipo entero ('int') que corresponde a una secuencia, no vacía, de dígitos. Como se decía, estos requisitos son definidos en el módulo 'Parser.TPDB.TRS.Scanner' de la siguiente forma:

```
1 -- | Parsec list of reserved symbols
2 trsDef :: P.LanguageDef st
3 trsDef = emptyDef {
4      P.commentStart = ""
5    , P.commentEnd = ""
6    , P.commentLine = ""
7    , P.nestedComments = True
8    , P.identStart = alphaNum <|> oneOf "<>~! $ %&/.:;-_{}[]^*+    '   ?=#@\\|"
9    , P.identLetter = P.identStart trsDef
10   , P.opStart = oneOf ")(\"-"
11   , P.opLetter = oneOf ")(\",><="
12   , P.reservedNames= [ "CONTEXTSENSITIVE", "EQUATIONS", "INNERMOST", "OUTERMOST" , "
       RULES"
13                      , "STRATEGY", "THEORY", "VAR"]
14   , P.reservedOpNames = ["->","==", "->=", "-><-", "|"]
15   , P.caseSensitive = True
16   }
```

Listing 4.5: Definición de tokens para TRS

Para el TRS con formato XML contamos con un conjunto de tokens que, en parte, se parece al del formato tradicional. Los patrones, tanto de los identificadores como de los tipos cadena y entero, son los mismos. Sin embargo, es este formato no tenemos operadores como en el formato anterior. Los operadores en este caso se describen mediante las etiquetas XML. Tampoco las palabras reservadas serán iguales, la lista ahora es la siguiente: problem, trs, rules, rule, relrules, lhs, rhs, conditions, condition, var, funapp, name, arg, strategy, signature, conditiontype, INNERMOST, OUTERMOST, FULL, funcsym, arity, theory, replacementmap, entry y comment. Además, tenemos los tokens propios de las etiquetas de este lenguaje de marcado (’<’, ’/’ y ’>’), que envolverán a nuestras palabras reservadas. Y, por último, los correspondientes a los comentarios, para los que se usan la secuencia de tokens ’<!–’ para la apertura y ’–>’ para el cierre de comentario.

Por lo tanto, para los formatos en XML la mayoría de los campos de este registro permanecen iguales a los de la definición anterior. Principalmente se ven modificados las listas de palabras reservadas (P.reservedNames), las de operadores (P.reservedOpNames) y los comentarios (P.commentStart y P.commentEnd). Se crea el módulo ’Parser.TPDB.TRS_XML.Scanner’ con estas modificaciones.

```
1  P.commentStart = "<!--"
2    , P.commentEnd = "-->"
```

Listing 4.6: Definición de comentarios en XML

```
1 , P.reservedNames= ["problem", "trs", "rules", "rule", "lhs", "rhs", "conditions",
2               "condition", "var", "funapp", "name", "arg", "strategy", "signature",
3               "conditiontype", "INNERMOST", "OUTERMOST", "FULL", "funcsym", "arity",
4               "theory", "replacementmap", "entry", "comment", "relrules"]
```

Listing 4.7: Definición de palabras reservadas en TRS-XML

```
1 , P.reservedOpNames = []
```

Listing 4.8: Definición de operadores en TRS-XML

Finalmente, en el conjunto de tokens que tenemos para los distintos formatos de problemas de COPS, los identificadores siguen siendo secuencias de caracteres, no vacías, que no pueden contener paréntesis, comillas dobles y coma, ya que son tokens también. Las secuencias especiales correspondientes a los operadores en este caso son: ’->’ y ’==’. Y tenemos las palabras reservadas RULES, VAR, ORIENTED, SEMI-EQUATIONAL, JOIN, CONDITIONTYPE, REPLACEMENT-MAP, COMMENT y SIG.

Partiendo también de la primera deinicion, tenemos las siguientes modificaciones:

```
1 , P.reservedNames= [ "RULES", "VAR", "ORIENTED", "SEMI-EQUATIONAL", "JOIN" , "
      CONDITIONTYPE", "REPLACEMENT-MAP", "COMMENT", "SIG"]
```

Listing 4.9: Definición de palabras reservadas para formatos de COPS

```
, P.reservedOpNames = ["->","=="]
```

Listing 4.10: Definición de operadores para formatos de COPS

4.3. Análisis Sintáctico

La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los nombres de los tokens producidos por el analizador del léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens [3]. En nuestro servicio definimos esa representación intermedia, en el módulo "Parser.Grammar", a través de la creación de nuestros propios tipos de datos.

Cada expresión y función en Haskell tiene un tipo. El tipo de un valor indica que comparte ciertas propiedades con otros valores del mismo tipo. Construir nuevos tipos de datos propios nos permite agregar estructura a los valores en nuestros programas. Podemos dar un nombre y un tipo distinto a una colección de valores relacionados. Definir nuestros propios tipos también impide mezclar accidentalmente valores de dos tipos que son estructuralmente similares pero que tienen nombres diferentes. En definitiva, creando tipos agregamos significado a los valores y el sistema de tipos nos proporciona esa abstracción que necesitamos.

En Haskell, los nombres de tipos y valores son independientes entre sí. Usamos un constructor de tipo (es decir, el nombre del tipo) en las declaraciones de los tipos, como se ve a continuación. Y usamos el constructor de su valor cuando los utilizamos en el código[11].

Dadas las especificaciones de los sistemas de reescritura vistas en el capítulo 3 de Preliminares. Y las gramáticas presentadas en la sección 2.1, que integran estas especificaciones en función de las categorías y los formatos problemas que se utilizan en cada competición. Creamos los siguientes tipos de datos para poder abstraer las características que se definen en las especificaciones de los sistemas, de las particularidades de la gramática.

```
------------------------------------------------------------------------------
-- Data
------------------------------------------------------------------------------

-- | Specification declaration
data Spec = Spec [Decl] -- ^ List of declarations
      deriving (Eq, Ord, Show, Data, Typeable)

-- | List of declarations
data Decl = Var [Id] -- ^ Set of variables
    | CType CondType -- ^ Type of conditional rules (XML format)
    | Strategy Strategydecl -- ^ Extra information
    | Context [(Id, [Int])] -- ^ Context-Sensitive strategy (COPS format)
    | Theory [Thdecl] -- ^ Set of rules
    | Signature [Signdecl] -- ^ Type of signature (XML format)
    | Rules [Rule] -- ^ Set of rules
    | AnyList Id [AnyContent] --AnyList Id [String]
    | Comment String -- ^ Extra information
```

```haskell
      deriving (Eq, Ord, Show, Data, Typeable)

-- | Theory declaration
data Thdecl = Thdecl SimpleThdecl [Thdecl]
      deriving (Eq, Ord, Data, Typeable)

-- | Simple theory declaration
data SimpleThdecl = Id Id [Id]
    | Equations [Equation]
      deriving (Eq, Ord, Data, Typeable)

-- | Equation declaration
data Equation = Term :==: Term
      deriving (Eq, Ord, Data, Typeable)

-- | Term declaration
data Term = T Id [Term]
    | XTerm XmlTerm
    deriving (Eq, Ord, Data, Typeable)

-- | XmlTerm declaration
data XmlTerm = Tfun Id [Term]
    | Tvar Id
    deriving (Eq, Ord, Data, Typeable)

-- | Rule declaration
data Rule = Rule SimpleRule [Cond]
      | COPSrule SimpleRule [Equation]
      deriving (Eq, Ord, Data, Typeable)

-- | Simple rule declaration
data SimpleRule = Term :-> Term
    | Term :->= Term
    deriving (Eq, Ord, Data, Typeable)

data Cond = Term :-><- Term
    | Arrow Term Term
    deriving (Eq, Ord, Data, Typeable)

-- | Strategy Declaration
data Strategydecl = INNERMOST
  | OUTERMOST
  | CONTEXTSENSITIVE [(Id, [Int])]
  | FULL
    deriving (Eq, Ord, Show, Data, Typeable)

data AnyContent = AnyId Id
  | AnySt String
  | AnyAC [AnyContent]
    deriving (Eq, Ord, Show, Data, Typeable)

-- | Condition Type
data CondType = JOIN
  | ORIENTED
  | OTHER
  | SEMIEQUATIONAL
  deriving (Eq, Ord, Show, Data, Typeable)

-- | Signature declaration
data Signdecl = S Id Int
  | Sth Id Int Id
  | Srp Id Int [Int]
    deriving (Eq, Ord, Data, Typeable)

-- | Identifier
type Id = String

```

```
86 -- | TSR Type
87 data TRSType = TRSStandard
88   | TRSEquational
89   | TRSConditional CondType
90   | TRSContextSensitive
91   | TRSContextSensitiveConditional CondType
92     deriving (Show)
93
94 data TRS
95   = TRS { trsSignature :: Map Id Int
96         , trsVariables :: Set Id
97         , trsRMap :: [(Id, [Int])]
98         , trsRules :: [Rule]
99         , trsType :: TRSType
100        , trsStrategy :: Maybe Strategydecl
101        , signatureBlock :: Bool
102        } deriving (Show)
```

Listing 4.11: Tipos de datos del módulo 'Parser.Grammar'

El módulo de análisis sintáctico creará y dará valores a esta estructura en función de la entrada que reciba. Y, como es usual, es sobre la que posteriormente trabajará el analizador semántico. Es equivalente a esa estructura de árbol de la que hablábamos. A continuación, se muestra una impresión de la estructura generada durante el análisis, en base a los valores del fichero de ejemplo que la acompaña.

```
[Var ["x","y","xs"],Theory ["AC" | "plus"],Rules [sum(x,y) -> S(int(x,y)),S(nil) -> 0,S
    (cons(x,xs)) -> plus(x,S(xs)),plus(x,0) -> x,plus(x,s(y)) -> s(plus(x,y)),int(0,0)
    -> cons(0,nil),int(0,s(y)) -> cons(0,int(s(0),s(y))),int(s(x),0) -> nil,int(s(x),s(
    y)) -> intlist(int(x,y)),intlist(nil) -> nil,intlist(cons(x,y)) -> cons(s(x),
    intlist(y))]]
```

Listing 4.12: Ejemplo de estructura obtenida tras el análisis sintáctico

```
1  (VAR x y xs)
2  (THEORY (AC plus))
3  (RULES
4    sum(x, y) -> S(int(x, y))
5    S(nil) -> 0
6    S(cons(x, xs)) -> plus(x, S(xs))
7    plus(x, 0) -> x
8    plus(x, s(y)) -> s(plus(x, y))
9    int(0, 0) -> cons(0, nil)
10   int(0, s(y)) -> cons(0, int(s(0), s(y)))
11   int(s(x), 0) -> nil
12   int(s(x), s(y)) -> intlist(int(x, y))
13   intlist(nil) -> nil
14   intlist(cons(x, y)) -> cons(s(x), intlist(y))
15 )
```

Listing 4.13: Problema utilizado para el ejemplo

4.3.1. Implementación del Analizador Sintáctico

Como ya se ha mencionado, con la aproximación que vamos a seguir de Parsec el análisis léxico y sintáctico se entrelazan. El método de análisis sintáctico que

se implementa en este servicio es de clase descendente recursivo. Encontraremos un procedimiento o función asociada con cada no terminal de la gramática correspondiente. En concreto se ajusta a una forma de análisis sintáctico de descenso recursivo conocida como análisis sintáctico predictivo. Todas las bibliotecas de combinadores están obligadas a utilizar un algoritmo de análisis predictivo, también conocido como derivación de izquierda a derecha o análisis LL.

Una condición necesaria para realizar este tipo de análisis sintáctico descendente y, por tanto, restricción importante en la mayoría de los analizadores combinadores existentes, como Parsec, es que no pueden manejar la recursividad a la izquierda. Lo primero que haría un analizador sintáctico recursivo a la izquierda es llamarse a sí mismo, lo que daría como resultado un bucle infinito. Toda gramática recursiva por la izquierda puede reescribirse en una gramática recursiva por la derecha. Con Parsec, también es posible definir una cadena combinadora que capture el patrón de diseño de la codificación de la asociatividad usando directamente la recursividad izquierda. De hecho, ya ofrece implementada una por defecto, evitando así una reescritura manual de la gramática [14].

Es por ello que, antes de implementar los analizadores sintácticos de cada gramática, se ha comprobado que estuvieran factorizadas y no fueran recursivas por la izquierda. También se ha tenido cuidado con otros posibles problemas que atañen a las gramáticas para los analizadores sintácticos, como es la ambigüedad. En nuestro caso no se ha tenido que prestar gran atención a la asociatividad y precedencia de los operadores, debido a que son características que no encontramos en nuestros formatos.

4.3.1.1. Analizador Sintáctico para los formatos de COPS

Se ha partido del analizador base con el que contábamos que reconocía los formatos recogidos en COPS. Este ha servido de referencia para los demás. También se le ha hecho alguna modificación para poder adaptarlo y usarlo en conjunto con el resto del código que se ha desarrollado. Usando la gramática correspondiente, expuesta en el apartado 2.1, se comprobó que el analizador funcionaba correctamente y cubría los casos requeridos.

En el módulo “Parser.COPS.TRS.Parser”, donde implementamos este analizador sintáctico, tenemos un conjunto de funciones que reconocen las distintas partes de dicha gramática. Estas funciones las vamos combinando y componiendo para poder reconocer la gramática en su totalidad.

La función ’trsCOPSParser’, es la función capaz de reconocer la gramática en su conjunto. Por ello, es la función que exportamos para poder usarla en los demás módulos. Lo que se hace en esta función, como se puede ver a continuación, es establecer que la especificación del formato está compuesta por varios bloques, al menos uno. Entre paréntesis, cada uno de ellos. Esto es exactamente lo que indican el combinador ’many1’ y el combinador ’parens’.

```
1 trsCOPSParser :: Parser Spec
2 trsCOPSParser = liftM Spec (many1 (whiteSpace >> parens decl))
```

Dichos bloques los analiza la función 'decl' que especifica, conforme a la gramática, que puede tratarse de un de los siguientes bloques: de condiciones, de variables, de reglas, de firmas de función, de estrategia, o comentarios. En la función utilizamos el combinador '<|>'. Este combinador es predictivo y solo prueba la alternativa que tiene a su derecha si la primera no ha consumido ninguna entrada. Es común verlo en conjunto al combinador 'try' que hace que el analizador al que se le añade simule no haber consumido entrada. Los dos juntos permiten una anticipación infinita en la gramática.

```
1 decl :: Parser Decl
2 decl = declVar <|> declRules <|>  declCSStrategy <|> declCondType <|> declSignature <|>
      declComment
```

Ahora entramos en el análisis de cada uno de los bloques que llamamos desde 'decl'. Y comenzamos con la función 'declVar', la cual analiza si el bloque que encontramos tiene la composición de un bloque de variables. Un bloque de definición de variables comienza con la palabra reservada 'VAR' y después puede contener varios identificadores. En esta función podemos ver como usando Parsec y sus combinadores realizamos el análisis léxico y sintáctico conjuntamente, ya que 'reserved' e 'identifier' son funciones que forman parte del conjunto de analizadores léxicos que obtenemos del módulo 'Scanner'. Sus funciones son las de reconocer las palabras reservadas y los identificadores, respectivamente. Usamos 'identifier' en la función auxiliar 'phrase' junto con el combinador 'many', ya que pueden ser varios los identificadores que encontremos declarados en el bloque.

```
1 -- | Variables declaration is formed by a reserved word plus a set of variables
2 declVar :: Parser Decl
3 declVar = reserved "VAR" >> do { idList <- phrase
4                                ; return . Var $ idList
5                                }
6
7 -- | A phrase
8 phrase = many identifier
```

En la función 'declRules' se analizan los bloques de reglas. Primero reconocemos la palabra reservada 'RULES' que encontramos siempre al principio de este bloque. Tras esto se examina cada una de las reglas que contiene. Cada regla está formada por dos términos y un operador que ha de ser '->'. Para reconocer los operadores se hace uso del analizador léxico 'reservedOp' cuyo funcionamiento es el mismo que el de 'reserved', solo que mira en la lista de operadores que habíamos definido.

Además, en una regla también se pueden especificar condiciones, si las ponemos a continuación del operador '|'. Como esto es opcional utilizamos el combinador 'option'. Este combinador tratar de aplicar el analizador que se le está pasando como segundo argumento, pero si falla sin consumir entrada devolverá el valor que le hemos dado en el primero.

En el caso de que tengamos varias condiciones tienen que ir separadas por comas. Para esto se define la función 'commaSep" que aplica a la condición el combinador 'sepEndBy', el cual, va a analiza que las condiciones aparezcan separadas por lo que se le pasa en el segundo argumento, que en este caso tiene que ser una coma ya que se le está pasando la función léxica 'comma'. Cada una de estas condiciones, a su vez, también están formadas por dos términos y un operador que en este caso debe ser '=='.

```
1  -- | Rules declaration is formed by a reserved word plus a set of rules
2  declRules :: Parser Decl
3  declRules = reserved "RULES" >> liftM Rules (many rule)
4
5  -- | Rule
6  rule :: Parser Rule
7  rule =
8   do sr <- simpleRule
9      conds <- option [] (reservedOp "|" >> commaSep' cond)
10     return (COPSrule sr conds)
11
12 -- | Simple rule
13 simpleRule =
14  do t1 <- term
15     op <- ruleOps
16     t2 <- term
17     return (op t1 t2)
18
19 -- | Rule options
20 ruleOps = (reservedOp "->" >> return (:->))
21
22 -- | Condition
23 cond =
24  do
25     t1 <- term
26     op <- condOps
27     t2 <- term
28     return (op t1 t2)
29
30 -- | Condition options
31 condOps = (reservedOp "==" >> return (:==:))
32
33 -- | Separated by comma
34 commaSep' :: Text.ParserCombinators.Parsec.Prim.GenParser Char () a
35              -> Text.ParserCombinators.Parsec.Prim.GenParser Char () [a]
36 commaSep' = (`sepEndBy` comma)
```

Para analizar los términos que pueden aparecer tanto en las reglas como en las condiciones definimos la función 'term'. Para poder reconocer un término esta función obliga a que esté compuesto de un identificador, que puede ir seguido de paréntesis. Estos paréntesis pueden, a su vez, contener una lista de términos. De haber varios términos irán separados por coma.

```
1 -- | A term
2 term :: Parser Term
3 term =
4  do n <- identifier
5     terms <- option [] (parens (commaSep' term))
6     return (T n terms)
```

La siguiente declaración a reconocer es la del bloque de estrategia. La fun-

ción 'declCSStrategy' analiza que el bloque empiece con la palabra reservada 'REPLACEMENT-MAP'. Tras esto comprueba que haya una lista, de al menos un elemento, en la que cada componente vaya entre paréntesis. Dentro debe contener un identificador y otra lista. Esta segunda lista, que puede ser una lista vacía, contiene números enteros separados por comas. Aquí empleamos un nuevo analizador del léxico llamado 'natural', para reconocer estos enteros. 'natural' comprueba que sea un número entero positivo y nos devuelve su valor.

```
1 -- | Context-sensitive strategy
2 declCSStrategy :: Parser Decl
3 declCSStrategy =
4  do reserved "REPLACEMENT-MAP"
5     strats <- many1$ parens (do a <- identifier
6                                 b <- commaSep' natural
7                                 return (a, map fromInteger b)
8                            )
9     return $ Context strats
```

Con la siguiente función 'declCondType' se analizan los bloques de condición, los cuales son muy sencillos de comprobar ya que solo se tiene que reconocer que el bloque empiece por la palabra reservada 'CONDITIONTYPE' la cual ha de ir seguida por una de las siguientes: SEMI-EQUATIONAL, JOIN u ORIENTED.

```
1 -- | Condition type declaration is formed by a reserved word plus SEMI-EQUATIONAL, JOIN,
     or ORIENTED
2 declCondType :: Parser Decl
3 declCondType = reserved "CONDITIONTYPE" >> liftM CType (semiEq <|> join <|> oriented)
4
5 -- | Semi-equational conditions
6 semiEq :: Parser CondType
7 semiEq = reserved "SEMI-EQUATIONAL" >> return SEMIEQUATIONAL
8
9 -- | Join conditions
10 join :: Parser CondType
11 join = reserved "JOIN" >> return JOIN
12
13 -- | Oriented conditions
14 oriented :: Parser CondType
15 oriented = reserved "ORIENTED" >> return ORIENTED
```

Entre las últimas declaraciones que podemos encontrar tenemos la de la signatura. En esta función, además de revisar la palabra 'SIG' con la que tiene que comenzar el bloque, se comprueba que venga una lista. La lista puede ser vacía o puede contener grupos de elementos entre paréntesis con, cada uno, un identificador y un entero en su interior.

```
1 -- | Signature declaration is formed by list of functions with arity
2 declSignature :: Parser Decl
3 declSignature = reserved "SIG" >> liftM Signature (many (parens fun))
4
5 -- | Function symbol
6 fun :: Parser Signdecl
7 fun =
8  do n <- identifier
9     m <- many1 digit
10    return (S n (read m))
```

Por último, se analiza el bloque de comentarios, donde la palabra reservada 'COMMENT' puede ir seguida de cualquier lista de caracteres, salvo el carácter ')' que indicará el fin de ese bloque. Para procesar esto utilizamos otro analizador léxico que no habíamos empleado hasta ahora, 'noneOf', al cual le pasamos como parámetro el carácter ')' para que consuma la entrada con cualquier carácter que le llegue, pero no este paréntesis de final de bloque, que debe ser reconocido por 'parens' en la función 'trsCOPSParser'.

```
1 declComment =
2  do reserved "COMMENT"
3     decls <- many $ noneOf ")"
4     return$ Comment decls
```

4.3.1.2. Analizador Sintáctico para los formatos de TPDB

Para la implementación de la gramática de los formatos de TPDB, al igual que hemos hecho para la de los de COPS, creamos un conjunto de nuevas funciones en un módulo al que ahora nombramos como "Parser.TPDB.TRS.Parser".

Al comienzo de este módulo definimos la función 'trsParser'. Esta es la función más general, capaz de reconocer el formato en su conjunto. Por ello, de nuevo, es la función que se exporta para que, más adelante, la podamos utilizar en los módulos pertinentes. Como vemos a continuación la función específica que un 'TRS' está compuesto por cualquier número de declaraciones, teniendo que haber al menos una. Como ya hemos mencionado es exactamente lo que hace el combinador 'many1'. Cada una de estas declaraciones también van siempre entre paréntesis, por lo que poniendo esas declaraciones como parámetro del analizador 'parens' nos cercioramos de ello. De lo contrario se devolverá un error.

```
1 -- |parse TRS specification
2 trsParser :: Parser Spec
3 trsParser = liftM Spec (many1 (whiteSpace >> parens decl))
```

Las posibles declaraciones que puede tomar la función anterior son las enumeradas en la función 'decl'. Especificamos entonces que en esta gramática debe tratarse con declaraciones de variables, de teoría, de reglas, de estrategia o listas que pueden contener cualquier ciertos elementos.

```
1 -- | A declaration is form by a set of variables, a theory, a set of rules, a strategy
     an extra information
2 decl :: Parser Decl
3 decl = declVar <|> declRules <|> declStrategy <|> declTheory <|> declAnylist
```

En esta gramática vemos que tanto la declaración de variables como la de reglas no varía demasiado, por lo que la implementación de las funciones que reconocen estas partes en casi igual.

```
1 -- | Variables declaration is formed by a reserved word plus a set of variables
2 declVar :: Parser Decl
3 declVar = reserved "VAR" >> do { idList <- phrase
```

```
4                                ; return . Var $ idList
5                                }
6
7  -- | A phrase
8  phrase = many identifier
9
10 -- | Rules declaration is formed by a reserved word plus a set of rules
11 declRules :: Parser Decl
12 declRules = reserved "RULES" >> liftM Rules (many rule)
13
14 -- | Rule
15 rule :: Parser Rule
16 rule =
17  do sr <- simpleRule
18     conds <- option [] (reservedOp "|" >> commaSep' cond)
19     return (Rule sr conds)
20
21 -- | Simple rule
22 simpleRule =
23  do t1 <- term
24     op <- ruleOps
25     t2 <- term
26     return (op t1 t2)
27
28 -- | Rule options
29 ruleOps = try (reservedOp "->" >> return (:->))
30         <|> (reservedOp "->=" >> return (:->=))
31
32 -- | Condition
33 cond =
34  do
35     t1 <- term
36     op <- condOps
37     t2 <- term
38     return (op t1 t2)
39
40 -- | Condition options
41 condOps = try (reservedOp "-><-" >> return (:-><-))
42         <|> (reservedOp "->" >> return (Arrow))
43
44 -- | A term
45 term :: Parser Term
46 term =
47  do n <- identifier
48     terms <- option [] (parens (commaSep' term))
49     return (T n terms)
```

Los cambios más relevantes a destacar los encontramos en los operadores de las reglas y en los de las condiciones. Son distintos que los que se usan en los problemas de confluencia.

La declaración de una estrategia sufre algunas modificaciones en este formato. Se utiliza 'reserved' para reconocer la palabra 'STRATEGY' que identifica el bloque, como se ha venido haciendo. Luego tenemos que volver a comprobar si nos llega una de las tres siguientes palabras reservadas: INNERMOST, OUTERMOST o CONTEXTSENSITIVE. En caso de que la que llegue sea 'CONTEXTSENSITIVE' podrá ir acompañada de una lista en la que cada elemento va entre parénesis y estará formado por un identificador seguido de uno o varios enteros. Si nos fijamos, los enteros ahora no han de ir separados por comas. En la función utilizamos, de nuevo, el analizador de números positivos 'natural' pero sin pasárselo a la función auxiliar 'commaSep".

```haskell
declStrategy :: Parser Decl
declStrategy = reserved "STRATEGY" >> liftM Strategy (innermost <|> outermost <|>
    contextsensitive)

-- | innermost strategy
innermost :: Parser Strategydecl
innermost = reserved "INNERMOST" >> return INNERMOST

-- | outermost strategy
outermost :: Parser Strategydecl
outermost = reserved "OUTERMOST" >> return OUTERMOST

-- | contextsensitive strategy
contextsensitive :: Parser Strategydecl
contextsensitive =
    do  reserved "CONTEXTSENSITIVE"
        strats <- many$ parens (do a <- identifier
                                   b <- many natural
                                   return (a, map fromInteger b)
                                )
        return $ CONTEXTSENSITIVE strats
```

Aunque en los problemas de terminación no encontramos una declaración de tipo de condición, como si teníamos en los de confluencia, tenemos un nuevo tipo de bloque de declaración que no encontrábamos antes. El bloque teoría, con la palabra reservada 'THEORY', está compuesto por listas de declaraciones de teoría cada una entre paréntesis. Dichas declaraciones pueden tratarse de listas de identificadores o de un bloque de ecuaciones, donde el bloque comienza con la palabra reservada 'EQUATIONS'. Una ecuación se define como una igualación de términos con el operador '=='.

```haskell
-- | Theory declaration is formed by a reserved word plus a set of theory declarations
declTheory :: Parser Decl
declTheory = reserved "THEORY" >> liftM Theory (many thdecl)

-- | Theory declaration
thdecl :: Parser Thdecl
thdecl =
 do sthd <- parens $ simpleThdecl
    listofthdecl <- option [] (many thdecl)
    return (Thdecl sthd listofthdecl)

-- | Simple theory declaration
simpleThdecl :: Parser SimpleThdecl
simpleThdecl = thlId <|> declEq

-- | Theory identificator list
thlId =
 do id <- identifier
    idlist <- option [] phrase
    return (Id id idlist)

-- | Equation declaration
declEq = reserved "EQUATIONS" >> liftM Equations (many eq)

-- | Equation
eq =
 do t1 <- term
    op <- eqOps
    t2 <- term
    return (op t1 t2)
```

```
31
32 -- | Equation options
33 eqOps = (reservedOp "==" >> return (:==:))
```

Por último, tenemos el bloque que suele ser usado para poner comentarios y que está formado por un identificador al que puede acompañar una lista. Como puede observe en la gramática correspondiente, dicha lista pude tener diversas composiciones. Aquí usamos 'stringLiteral', otro analizador léxico que reconoce cadenas de caracteres y trata automáticamente las secuencias de escape y los espacios.

```
1 declAnylist :: Parser Decl
2 declAnylist=
3  do
4     name <- identifier
5     decls <- many anyContent
6     return $ AnyList name decls
7
8 anyContent :: Parser AnyContent
9 anyContent = anyId <|> anySt <|> anyAC <|> (comma >> anyContent)
10
11 -- | Identifiers
12 anyId :: Parser AnyContent
13 anyId = liftM AnyId identifier
14
15 -- | Strings
16 anySt :: Parser AnyContent
17 anySt = liftM AnySt stringLiteral
18
19 -- | Others
20 anyAC :: Parser AnyContent
21 anyAC = liftM AnyAC (parens $ many anyContent)
```

4.3.1.3. Analizador Sintáctico para los formatos de TPDB definidos en XML

Para la versión moderna de los formatos de TPDB definidos sobre XML, volvemos a crear un nuevo módulo. Esta vez se va a llamar "Parser.TPDB.TRS_XML.Parser". Como puede observarse en su gramática, presentada en la sección 2.1 "Formatos de las competiciones", encontramos bastantes cambios respecto de la versión original del formato. Una notable diferencia es que, en este caso, se tiene que reconocer y tratar la estructura del lenguaje de marcado XML en conjunto con las características y los distintos bloques del propio TRS.

Utilizando los combinadores básicos y sin crear ninguna función adicional se podría haber incluido toda la lista de etiquetas existentes, tanto si son de apertura como de cierre, en la lista de palabras reservadas que se ha definido para este módulo. Sin embargo, se ha decidido que hacer esto no es lo más correcto ya que, además, estaríamos juntando las propias palabras reservadas que dicta el TRS con la composición y estructura de las etiquetas en XML. No tendríamos distinción entre una y otra. Estaríamos incrementando innecesariamente la lista de palabras reservadas y empeorando la simplicidad y legibilidad del código en la parte del análisis sintáctico, haciéndolo menos genérico y reutilizable, de cara a correcciones y extensiones de lo que ya existe.

Por las razones expuestas en el párrafo anterior se crea la siguiente función, la cual admite como argumentos la palabra reservada que contiene la etiqueta junto con el cuerpo que van a delimitar dichas etiquetas. La función se encargará de comprobar tanto que la etiqueta tenga el formato correcto en su apertura y en su cierre, como que la palabra reservada esté en la lista. Una vez comprobado, devolverá el valor del cuerpo.

```
1  -- | XML labels
2  reservedLb q p=between (try $ aux1 q) (try $ aux2 q) p
3
4  aux1 q=do{ (symbol "<")
5           ; (reserved q)
6           ; manyTill anyChar (try (symbol ">"))
7           }
8
9  aux2 q=do{ (symbol "<")
10          ; (symbol "/")
11          ; (reserved q)
12          ; (symbol ">")
13          }
```

Usamos el combinador 'between' para poder analizar que la estructura contenga lo que se le está poniendo como último parámetro y esté delimitada, es decir, comience y termine, por lo que le indicamos en el primer y segundo parámetro. Si es correcto y está bien delimitado nos devolverá el valor del cuerpo para que podamos analizarlo.

Como la palabra reservada que contiene la etiqueta debe ser la misma al principio y al final, esto es lo que le especificamos a 'between'. Por lo que solo necesitamos que nuestra función tenga un argumento para la palabra reservada. En la etiqueta de apertura comprobamos que empieza con el símbolo '<', seguido de la palabra reservada que corresponda. Gracias al combinador 'manyTill' somos capaces de escapar cualquier atributo que pueda venir en la etiqueta y que no nos interesa reconocer. A 'manyTill' se le pasa el analizador léxico 'anyChar' para que lo aplique y reconozca cualquier carácter (cero o más veces) hasta que encuentre el símbolo '>', que le suministramos como su segundo parámetro. De forma similar, para reconocer la etiqueta de cierre exigimos que empiece con el carácter '<' seguido de '/' y la misma palabra reservada que en la etiqueta de apertura seguida del carácter de cierre de etiqueta '>'.

Otra característica de la versión XML del formato TRS es que, un problema normalmente comienza y acaba con metainformación. Como no nos interesa analizarla se crea otra función auxiliar que reconoce y obvia esta metainformación. Por lo que encontramos en ejemplos de problemas, siempre va entre etiquetas que comienzan y terminan o bien por '<?' y '?>', o bien por '<metainformation>' y '</metainformation>'.

```
1  metainf = try (do{ whiteSpace
2                   ; string "<?"
3                   ; manyTill anyChar (try (string "?>"))
4                   })
5            <|> (do{ whiteSpace
6                   ; string "<metainformation>"
7                   ; manyTill anyChar (try (string "</metainformation>"))
```

```
8                  })
```

A continuación, comenzamos con la propia especificación del TRS. En este caso será la función 'trsXmlParser' la que se exporta al resto de módulos. Como se puede ver es aquí donde utilizamos la función 'metainf' que acabamos de mostrar.

Los problemas con formato XML vienen delimitados entre las etiquetas '<problem></problem>'. Dentro encontramos siempre los bloques '<trs></trs>' y '<strategy></strategy>' que tratamos a continuación. Además, puede haber más bloque de metainformación, para lo que usaremos nuestra función auxiliar.

```
1 trsXmlParser :: Parser Spec
2 trsXmlParser = whiteSpace >> (many $ try metainf) >> whiteSpace >>
3     (reservedLb "problem" $ do{d <- reservedLb "trs" (many decl)
4                              ; st <- strategy
5                              ; (many $ try metainf)
6                              ; whiteSpace
7                              ; return $ Spec (st:d)
8                              })
```

El bloque 'trs' está formado por la declaración de reglas, el de firmas de función y, opcionalmente, el bloque de comentarios y el de declaración del tipo de condición. Como venimos haciendo, la implementación para poder analizar sintácticamente esto es similar a la de las otras gramáticas pero con las funciones que reconocen las partes que ahora nos encontramos.

```
1 -- | A declaration is form by a set of rules, signatures, a condition type and comments
2 decl :: Parser Decl
3 decl = declRules <|> declSignature <|> ctypeDecl <|> declComment
```

La implementación para poder analizar sintácticamente las reglas, con sus condiciones y términos, es la siguiente.

```
1 -- | Rules declaration is formed by reserved tags plus a set of rules
2 declRules :: Parser Decl
3 declRules = reservedLb "rules" $ liftM Rules (rulesType)
4
5 rulesType =(try $ many (try $ reservedLb "rule" rule))
6         <|> (try $ reservedLb "relrules" $ (many (reservedLb "rule" relRule)) )
7
8
9 -- | Rule
10 rule :: Parser Rule
11 rule =
12  do sr <- simpleRule
13     conds <- option [] (reservedLb "conditions" (many $ reservedLb "condition" cond))
14     return (Rule sr conds)
15
16 -- | Simple rule
17 simpleRule =
18  do t1 <- reservedLb "lhs" term
19     op <- return (:->)
20     t2 <- reservedLb "rhs" term
21     return (op t1 t2)
22
23     -- | Relative rule
```

```
24      relRule :: Parser Rule
25      relRule =
26      do sr <- simpleRelRule
27          conds <- option [] (reservedLb "conditions" (many $ reservedLb "condition" cond)
                )
28          return (Rule sr conds)
29
30      -- | Simple relative rule
31      simpleRelRule =
32      do t1 <- reservedLb "lhs" term
33          op <- return (:->=)
34          t2 <- reservedLb "rhs" term
35          return (op t1 t2)
36
37
38  -- | Condition
39  cond =
40   do
41      t1 <- reservedLb "lhs" term
42      op <- condOps
43      t2 <- reservedLb "rhs" term
44      return (op t1 t2)
45
46  -- | Condition options
47  condOps = try (return (Arrow))
48
49  -- | A term
50  term :: Parser Term
51  term =
52   do xmlTerm <- (try termVar) <|> (reservedLb "funapp" termFun)
53      return (XTerm xmlTerm)
54
55  termVar :: Parser XmlTerm
56  termVar = liftM Tvar (reservedLb "var" identifier)
57
58  termFun :: Parser XmlTerm
59  termFun =
60   do n <- reservedLb "name" identifier
61      terms <- (many (try $ reservedLb "arg" term))
62      return (Tfun n terms)
```

Como vemos es muy similar a la implementación que se hizo de estas mirmos tipos de bloque en los otros formatos. Los cambios más importantes que se han hecho aquí es el reconocimiento de las etiquetas con la función 'reservedLb' en vez de usar 'reserved' para las palabras reservadas como teníamos antes, y la composición de los términos. Ahora un término puede ser un identificador marcado por las etiquetas '<var></var>' lo que significa que ese término hace referencia a una variable y es este el motivo de que no haya una declaración explicita de variables en este formato como si la había en los otros. Un término puede ser también una función, como indican las etiquetas '<funapp></funapp>', que estará formado por el identificador de dicha función marcado con la etiqueta '<name>' y la lista de argumentos, cada uno entre las etiquetas '<arg>' y '</arg>'. El argumento contiene a su vez un término.

Y, además, ahora se reconocen otro tipo de reglas que son las reglas relativas. Estás reglas a efectos de sintaxis se tratan casi igual que las normales, es decir, las únicas diferencias que encontramos son las etiquetas que las delimitan ('<relrules></relrules>') y que los términos que contienen usan el operador '->=' en lugar de '->'. El resto permanece igual.

Junto con el bloque de reglas que acabamos de ver tenemos la declaración de firmas de función, que aparece nueva en este formato. Esta declaración la forman los identificadores de las funciones que tenemos en las reglas junto con su aridad correspondiente. Además, es aquí, junto a la declaración de cada una de las signaturas, donde se debe especificar la declaración de la teoría o si dicha función tiene asignado un mapa de remplazo ('replacementmap'), que en el TRS tradicional se especificaba en el bloque 'CONTEXTSENSITIVE' o en los formatos de COPS en el 'REPLACEMENT-MAP'. Ahora la declaración de teoría consta de un identificador. Y en la del mapa de remplazo seguiremos teniendo que especificar una lista de enteros pero esta vez cada uno irá entre las etiquetas '<entry></entry>'.

```
1  -- | Signature declaration is formed by list of functions with arity
2  declSignature :: Parser Decl
3  declSignature = reservedLb "signature" $ liftM Signature (many (try $ reservedLb "
       funcsym" fun))
4
5  -- | Function symbol
6  fun :: Parser Signdecl -- (Id,Int)
7  fun = (sigTh) <|> (sigRpNull) <|> (sigRp)  <|> (sig)
8
9  sigRpNull = try (do{ n <- reservedLb "name" identifier
10                     ; m <- reservedLb "arity" natural
11                     ; try $ emptyReservedLb "replacementmap"
12                     ; return (Srp n (fromInteger m) [])
13                     })
14
15 sigRp = try (do { n <- reservedLb "name" identifier
16                 ; m <- reservedLb "arity" natural
17                 ; rp <- reservedLb "replacementmap" (many1 $ reservedLb "entry" natural)
18                 ; return (Srp n (fromInteger m) (map fromInteger rp))
19                 })
20
21 sig = try (do{ n <- reservedLb "name" identifier
22              ; m <- reservedLb "arity" natural
23              ; return (S n (fromInteger m))
24              })
25
26 sigTh = try (do { n <- reservedLb "name" identifier
27                 ; m <- reservedLb "arity" natural
28                 ; th <- reservedLb "theory" identifier
29                 ; return (Sth n (fromInteger m) th)
30                 })
```

Destacar que no pueden aparecer a la vez declaraciones de teoría y 'replacementmaps', ni en la misma signatura ni entre el resto de ellas. Es por eso que en nuestra estructura de datos de “Parser.Grammar”, el tipo 'Signature' tiene asociados constructores diferentes para los tres posibles casos: que haya teoría, que haya mapa de remplazos o que encontremos simplemente la firma de la función. Además, cuando se declara un mapa de remplazo, este puede aparecer vacío (sin contenido), por ello se ha creado la función 'emptyReservedLb' que reconoce la etiqueta vacía con la palabra reservada que le pasemos.

```
1  emptyReservedLb q = do { (symbol "<")
2                         ; (reserved q)
3                         ; (symbol "/")
4                         ; (symbol ">")
5                         }
```

La declaración de estrategia y la de los comentarios no tiene mucha complejidad. Tratamos un comentario como cualquier número de caracteres que estén entre las etiquetas '<comment></comment>', no analizaremos el contenido que tengan en su interior ya que no es relevante. El analizador 'noneOf' consumirá con éxito cualquier entrada hasta que encuentre el carácter '<' que le proporcionamos, para que justo después 'reservedLb' pueda reconocer correctamente la etiqueta de fin de comentario.

```
1 -- | Extra information
2 declComment = liftM Comment (reservedLb "comment" (many $ noneOf "<"))
```

La estrategia en esta gramática podemos ver que no aparece junto al resto de bloques, como lo hacía antes. Tenemos el bloque de estrategia al mismo nivel que las etiquetas '<trs></trs>'. Este bloque contendrá ahora una de las siguientes palabras reservadas: INNERMOST, OUTERMOST o FULL. Ya no aparece la palabra 'CONTEXTSENSITIVE' que teníamos en la anterior versión del TRS. Como se ha explicado, los mapas de remplazamiento ya no van declarados aquí, sino que se hace en el bloque de firmas. Así pues, la implementación para reconocer el bloque de estrategia queda de la siguiente forma:

```
1 strategy :: Parser Decl
2 strategy = reservedLb "strategy" $ liftM Strategy (innermost <|> outermost <|>
      contextsensitive)
3
4 -- | innermost strategy
5 innermost :: Parser Strategydecl
6 innermost = reserved "INNERMOST" >> return INNERMOST
7
8 -- | outermost strategy
9 outermost :: Parser Strategydecl
10 outermost = reserved "OUTERMOST" >> return OUTERMOST
11
12 -- | full strategy
13 contextsensitive :: Parser Strategydecl
14 contextsensitive = reserved "FULL" >> return FULL
```

Finalmente, en este formato aparece un nuevo bloque que no encontrábamos en el TRS tradicional pero que si tuvimos para los formatos de COPS. Se trata de la declaración de tipo de condición. Para esta gramática debe estar compuesto por la palabra reservada 'JOIN', 'ORIENTED' u 'OTHER' delimitada por las etiquetas '<conditiontype></conditiontype>'.

```
1 ctypeDecl :: Parser Decl
2 ctypeDecl = reservedLb "conditiontype" $ liftM CType (join <|> oriented <|> other)
3
4 -- | join condition type
5 join :: Parser CondType
6 join = reserved "JOIN" >> return JOIN
7
8 -- | oriented condition type
9 oriented :: Parser CondType
10 oriented = reserved "ORIENTED" >> return ORIENTED
11
```

```
12 -- | other condition type
13 other :: Parser CondType
14 other = reserved "OTHER" >> return OTHER
```

4.4. Análisis Semántico

El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. Una parte importante del análisis semántico es la recopilación y comprobación de tipos, verificando que cada operador tenga operandos que coincidan [3]. En el caso del analizador desarrollado en este trabajo esa comprobación no se va a implementar debido a que los formatos sobre los que trabajamos carecen de operaciones en las que se deban comprobar que los tipos sean los adecuados. Tampoco cuentan con declaraciones de variables en las que se les asigne un tipo a las mismas. Para los únicos tipos que encontramos en estos formatos, entero y cadena, es en las fases de análisis previas donde simplemente se comprueba que efectivamente se trata del tipo correspondiente.

Para realizar en análisis semántico recorremos la estructura que contiene todos los valores del fichero fuente, proporcionada por el analizador sintáctico.

4.4.1. Creación y exposición del módulo semántico

Para implementar y aplicar las condiciones semánticas a la estructura devuelta por cualquiera de los analizadores que hemos desarrollado se crea el módulo 'Parser.Parser'. Como este módulo es el que va a ser utilizado directamente por las interfaces del servicio, lo primero que hacemos es crear y exportar las siguientes funciones:

```
1  ----------------------------------------------------------------------------
2  -- Functions
3  ----------------------------------------------------------------------------
4
5  -- | Parses a TPDB problem
6  parseTPDB :: String -> Either ParseError TRS
7  parseTPDB = (checkConsistency (checkTPDBDeclaration)) . parseTRS
8
9  -- | Parses a term rewriting system in TPDB format
10 parseTRS :: String -> Either ParseError Spec
11 parseTRS s = sortSpec (doParse s trsParser)
12
13
14 -- | Parses a TPDB-XML problem
15 parseTPDB_XML :: String -> Either ParseError TRS
16 parseTPDB_XML = (checkConsistency (checkXMLDeclaration)) . parseTRS_XML
17
18 -- | Parses a term rewriting system in TPDB-XML format
19 parseTRS_XML :: String -> Either ParseError Spec
20 parseTRS_XML s = sortSpec (doParse s trsXmlParser)
21
22
23 -- | Parses a COPS problem
24 parseCOPS :: String -> Either ParseError TRS
25 parseCOPS = (checkConsistency (checkCOPSDeclaration)) . parseTRS_COPS
26
```

```
27 -- | Parses a term rewriting system in COPS format
28 parseTRS_COPS :: String -> Either ParseError Spec
29 parseTRS_COPS s = sortSpec (doParse s trsCOPSParser)
```

Haremos uso de una de estas funciones ’parseTPDB’, ’parseTPDB_XML’ o ’parseCOPS’, dependiendo de la gramática con la que queramos analizar un determinado fichero, es decir, del analizador sintáctico que queramos aplicar. Cada una de ellas recibe el contenido del fichero como una cadena de caracteres.

Lo primero que hacen estas funciones es aplicar el analizador sintáctico a esa cadena de caracteres. Para ello llaman a la función auxiliar que las acompaña: ’parseTRS’, ’parseTRS_XML’ y ’parseTRS_COPS’. Desde dicha función se ejecuta, a su vez, a la función ’doParse’ con la cadena de entrada y el tipo de analizador (’parser’) que se le está aplicando. El ’parser’ que está recibiendo la función es la función de análisis que creamos y exportamos en los módulos sintácticos vistos.

```
1 -- | Parses the system and returns the parsing error or the succesful parsed system.
2 -- (parse p filePath input) runs a character parser p without user state.
3 -- The filePath is only used in error messages and may be the empty string.
4 doParse :: String -> Parser a -> Either ParseError a
5 doParse s p = parse p "" s
```

En ’doParse’ recibimos dichos argumentos, la cadena ’s’ a analizar y el analizador a aplicar ’p’. Para ejecutar el análisis hacemos uso de la función ’parse’ de la biblioteca de Parsec. Esta función ejecuta el analizador de caracteres ’p’ devolviendo un ParseError (Left) o un valor de tipo ’a’ (Right). Aquí vemos claramente como, aprovechando el concepto de mónada mencionado al principio del capítulo, somos capaces de devolver el resultado del análisis si va bien, o cortocircuitar la cadena de cálculos y propagar el error si el análisis ha fallado. Esto lo hacemos gracias al uso de ’Either’, mónada que suele usarse para representar un cálculo que podría fallar, ya que cuenta con dos constructores ’Left’ y ’Right’ con los que encapsula el resultado. ’Left’ para contener los detalles del error en caso de fallo o ’Right’ para contener los resultados correctos.

Volviendo a la ejecución de la función ’doParse’, en caso de que haya ido bien estaremos devolviendo la estructura que vimos en el apartado 4.3. Lo último que hacemos, de nuevo en la función auxiliar correspondiente (’parseTRS’, ’parseTRS_XML’ o ’parseTRS_COPS’), es ordenar dicha estructura antes de devolver el resultado del análisis sintáctico a la función principal. Ordenar la estructura para que posteriormente nos resulte más sencillo verificar que las declaraciones encontradas sean correctas.

```
1 sortSpec :: Either ParseError Spec -> Either ParseError Spec
2 sortSpec (Left parseError) = Left parseError
3 sortSpec (Right (Spec decls))=  Right (Spec $ sort decls)
```

En la función ’sortSpec’ hacemos este reordenado utilizando la función ’sort’ que nos proporciona Haskell. ’sort’ ordena automáticamente el tipo de datos ’Decl’ que definimos en nuestro módulo ’Parser.Grammar’ y podemos ver en el ’Listing 4.18’ del apartado 4.3. Por defecto, la función ’sort’ los organiza la estructura

siguiendo el orden en el que hemos puesto cada uno de los constructores de tipo, que no fue casual.

Finalmente, una vez que tenemos los valores del análisis sintáctico ordenados en la estructura, podemos aplicarle las función 'checkConsistency', con la que se comprobarán los requisitos semánticos.

4.4.2. Implementación de las condiciones semánticas

En la función 'checkConsistency' recibimos la especificación del fichero que estamos analizando. A esta le vamos a aplicar el conjunto de funciones que veremos más adelante. Pero antes, comprobamos si el tipo y cantidad de declaraciones que encontramos en esa especificación es correcta.

```
1 checkConsistency :: (Either ParseError Spec -> Decl -> Either ParseError Spec) -> Either
      ParseError Spec -> Either ParseError TRS
2 checkConsistency _ (Left parseError) = Left parseError
3 checkConsistency checkDclFun (Right (Spec decls))
4   = case foldl checkDclFun (Right (Spec [])) decls of
5         Left parseError -> Left parseError
6         Right (Spec _) -> evalState (checkWellFormed decls) (TRS M.empty S.empty [] []
              TRSStandard Nothing)
```

Para hacer esta comprobación de declaraciones utilizamos la función 'foldl' que nos proporciona Haskell. A 'foldl' le pasamos 'checkDclFun' que tomará el valor de la función de comprobación de declaraciones que corresponda. La recibe como parámetro. Tenemos definida una función de comprobación de declaraciones por cada gramática, son 'checkTPDBDeclaration', 'checkXMLDeclaration' y 'checkCOPSDeclaration'.

La función 'foldl' utiliza la funcion recibida en 'checkDclFun' como función 'de paso', y toma un valor inicial para su 'acumulador', en este caso es la estructura de la especificación vacía ('(Right (Spec [ ]))'). 'foldl' también toma la lista de declaraciones 'decl' que llega en ese momento desde el sintáctico. El 'paso' toma el acumulador y un elemento de 'decl' y devuelve un nuevo valor de acumulador. 'foldl' lo que hace es llamar sucesivamente a la función 'de paso' sobre el acumulador y el elemento de 'decl' que extrae en cada paso. Y se pasa a sí mismo el nuevo valor del acumulador, de forma recursiva, para consumir el resto de la lista.

En las funciones 'checkTPDBDeclaration', 'checkXMLDeclaration' y 'checkCOPSDeclaration' expresamos los posibles ordenamientos de los bloques en cada gramática. Así como los bloques mínimos obligatorios que deben aparecer y aquellos que pueden repetirse, es decir, que podemos encontrar más de una vez. Las funciones devolverán el error en caso de que no se cumplan las condiciones requeridas.

```
1 -- | Checks declaration order for TPDB
2 checkTPDBDeclaration :: Either ParseError Spec -> Decl -> Either ParseError Spec
3 checkTPDBDeclaration (Left parseError) _ = Left parseError
4 checkTPDBDeclaration (Right (Spec [])) (Var vs) = Right . Spec $ [Var vs]
5 checkTPDBDeclaration (Right (Spec (Var vs:rest))) (Var vss) = Right . Spec $ (Var vss:
     Var vs:rest)
```

```
checkTPDBDeclaration (Right (Spec (Var vs:rest))) (Strategy st) = Right . Spec $ [Strategy st, Var vs]
checkTPDBDeclaration (Right (Spec (Var vs:rest))) (Theory th) = Right . Spec $ (Theory th:Var vs:rest)
checkTPDBDeclaration (Right (Spec (Var vs:rest))) (Rules rs) = Right . Spec $ (Rules rs:Var vs:rest)
checkTPDBDeclaration (Right (Spec (Strategy st:rest))) (Theory th) = Right . Spec $ (Theory th:Strategy st:rest)
checkTPDBDeclaration (Right (Spec (Strategy st:rest))) (Rules rs) = Right . Spec $ (Rules rs:Strategy st:rest)
checkTPDBDeclaration (Right (Spec (Theory th:rest))) (Theory thh) = Right . Spec $ (Theory thh:Theory th:rest)
checkTPDBDeclaration (Right (Spec (Theory th:rest))) (Rules rs) = Right . Spec $ (Rules rs:Theory th:rest)
checkTPDBDeclaration (Right (Spec (Rules rs:rest))) (Rules rss) = Right . Spec $ (Rules rss:Rules rs:rest)
checkTPDBDeclaration (Right (Spec (Rules rs:rest))) (AnyList id al) = Right . Spec $ ((AnyList id al):Rules rs:rest)
checkTPDBDeclaration (Right (Spec ((AnyList id al):rest))) (AnyList idd all) = Right . Spec $ ((AnyList idd all):(AnyList id al):rest)
checkTPDBDeclaration _ (Var _) = Left $ newErrorMessage (UnExpect "VAR block") (newPos "" 0 0)
checkTPDBDeclaration _ (Rules _) = Left $ newErrorMessage (UnExpect "RULES block") (newPos "" 0 0)
checkTPDBDeclaration _ (Theory _) = Left $ newErrorMessage (UnExpect "THEORY block") (newPos "" 0 0)
checkTPDBDeclaration _ (Strategy _) = Left $ newErrorMessage (UnExpect "STRATEGY block") (newPos "" 0 0)
checkTPDBDeclaration _ (AnyList _ _) = Left $ newErrorMessage (UnExpect "ANYLIST block") (newPos "" 0 0)
```

Listing 4.14: Comprobación bloques de declaraciones para la gramática de los formatos de problemas de TPDB

```
-- | Checks declaration order for TPDB xml
checkXMLDeclaration :: Either ParseError Spec -> Decl -> Either ParseError Spec
checkXMLDeclaration (Left parseError) _ = Left parseError
checkXMLDeclaration (Right (Spec [])) (CType ctype) = Right . Spec $ [CType ctype]
checkXMLDeclaration (Right (Spec [])) (Strategy stgy) = Right . Spec $ [Strategy stgy]
checkXMLDeclaration (Right (Spec [CType ctype])) (Strategy stgy) = Right . Spec $ [Strategy stgy, CType ctype]
checkXMLDeclaration (Right (Spec (Strategy stgy:rest))) (Signature sg) = Right . Spec $ (Signature sg:Strategy stgy:rest)
checkXMLDeclaration (Right (Spec (Signature sg:rest))) (Rules rs) = Right . Spec $ (Rules rs:Signature sg:rest)
checkXMLDeclaration (Right (Spec (Rules rs:rest))) (Rules rss) = Right . Spec $ (Rules rss:Rules rs:rest)
checkXMLDeclaration (Right (Spec (Rules rs:rest))) (Comment c) = Right . Spec $ (Comment c:Rules rs:rest)
checkXMLDeclaration _ (CType _) = Left $ newErrorMessage (UnExpect "CONDITIONTYPE block") (newPos "" 0 0)
checkXMLDeclaration _ (Strategy _) = Left $ newErrorMessage (UnExpect "STRATEGY block") (newPos "" 0 0)
checkXMLDeclaration _ (Rules _) = Left $ newErrorMessage (UnExpect "RULES block") (newPos "" 0 0)
checkXMLDeclaration _ (Signature _) = Left $ newErrorMessage (UnExpect "SIGNATURE block") (newPos "" 0 0)
checkXMLDeclaration _ (Comment _) = Left $ newErrorMessage (UnExpect "COMMENT block") (newPos "" 0 0)
```

Listing 4.15: Comprobación bloques de declaraciones para la gramática de los formatos de problemas de TPDB en XML

```
-- | Checks declaration order for COPS
checkCOPSDeclaration :: Either ParseError Spec -> Decl -> Either ParseError Spec
checkCOPSDeclaration (Left parseError) _ = Left parseError
checkCOPSDeclaration (Right (Spec [])) (Var vs) = Right . Spec $ [Var vs]
checkCOPSDeclaration (Right (Spec [])) (CType ctype) = Right . Spec $ [CType ctype]
checkCOPSDeclaration (Right (Spec [])) (Context rmap) = Right . Spec $ [Context rmap]
checkCOPSDeclaration (Right (Spec [])) (Signature sg) = Right . Spec $ [Signature sg]
checkCOPSDeclaration (Right (Spec [])) (Rules rs) = Right . Spec $ [Rules rs]
checkCOPSDeclaration (Right (Spec (Var vs:rest))) (Var vss) = Right . Spec $ (Var vss:
    Var vs:rest)
checkCOPSDeclaration (Right (Spec (Var vs:rest))) (CType ctype) = Right . Spec $ (CType
    ctype:Var vs:rest)
checkCOPSDeclaration (Right (Spec (Var vs:rest))) (Context rmap) = Right . Spec $ (
    Context rmap:Var vs:rest)
checkCOPSDeclaration (Right (Spec (Var vs:rest))) (Signature sg) = Right . Spec $ (
    Signature sg:Var vs:rest)
checkCOPSDeclaration (Right (Spec (Var vs:rest))) (Rules rs) = Right . Spec $ (Rules rs:
    Var vs:rest)
checkCOPSDeclaration (Right (Spec (CType ctype:rest))) (Var vs) = Right . Spec $ (Var vs
    :CType ctype:rest)
checkCOPSDeclaration (Right (Spec (CType ctype:rest))) (Context rmap) = Right . Spec $ (
    Context rmap:CType ctype:rest)
checkCOPSDeclaration (Right (Spec (CType ctype:rest))) (Rules rs) = Right . Spec $ (
    Rules rs:CType ctype:rest)
checkCOPSDeclaration (Right (Spec (Context rmap:rest))) (Rules rs) = Right . Spec $ (
    Rules rs:Context rmap:rest)
checkCOPSDeclaration (Right (Spec (Signature sg:rest))) (Rules rs) = Right . Spec $ (
    Rules rs:Signature sg:rest)
checkCOPSDeclaration (Right (Spec (Rules rs:rest))) (Rules rss) = Right . Spec $ (Rules
    rss:Rules rs:rest)
checkCOPSDeclaration (Right (Spec (Rules rs:rest))) (Comment c) = Right . Spec $ (
    Comment c:Rules rs:rest)
checkCOPSDeclaration _ (CType _) = Left $ newErrorMessage (UnExpect "CONDITIONTYPE block
    ") (newPos "" 0 0)
checkCOPSDeclaration _ (Var _) = Left $ newErrorMessage (UnExpect "VAR block") (newPos "
    " 0 0)
checkCOPSDeclaration _ (Context _) = Left $ newErrorMessage (UnExpect "REPLACEMENT-MAP
    block") (newPos "" 0 0)
checkCOPSDeclaration _ (Rules _) = Left $ newErrorMessage (UnExpect "RULES block") (
    newPos "" 0 0)
checkCOPSDeclaration _ (Signature _) = Left $ newErrorMessage (UnExpect "SIGNATURE block
    ") (newPos "" 0 0)
checkCOPSDeclaration _ (Comment _) = Left $ newErrorMessage (UnExpect "COMMENT block") (
    newPos "" 0 0)
```

Listing 4.16: Comprobación bloques de declaraciones para la gramática de los formatos de problemas de COPS

Con estas funciones obligamos, por ejemplo, a que al menos una sección de VAR y una de RULES sean obligatorias en la versión original del TRS de TPDB, o que en la versión XML aparezcan obligatoriamente las secciones de 'strategy', 'rules' y 'signature'. O, en los formatos de COPS, la definición obligatoria de 'RULES', añadiéndole 'SIG' si se trata de un eTRS, 'CONDITIONTYPE ' si un CTRS, 'REPLACEMENT-MAP' con un CSTRS y las dos últimas juntas si el formato corresponde a CSCTRS.

Continuando con el análisis semántico, volvemos a la función 'checkConsistency' donde si 'checkDclFun' no da error entramos a evaluar la función 'checkWellFormed'. Es esta función donde, a base de recorrer toda la estructura sintáctica, vamos a evaluar los requisitos semánticos más importantes y extraer toda la información relevante para devolverla posteriormente. Todo ello nos fuerza a

crear y mantener un estado que podamos consultar e ir modificando durante la ejecución de ’checkWellFormed’, hasta que termine de recorrer la totalidad de la estructura de valores del sintáctico.

Para poder tener y propagar este estado volvemos al concepto de mónada que explicamos al comenzar el capítulo. La mónada de estado nos permite administrar un estado mutable de manera ordenada. Haremos uso de la mónada estado para poder perpetuar implícitamente dicho estado que, como veremos a continuación, contendrá ciertas estructuras a las que nos interesa poder acceder, en cualquier momento, durante la ejecución de las funciones. En definitiva, este concepto de mónada estado nos sirve para poder leer algunas partes y modificar otras a través de asignaciones, de forma similar a como se hace en un lenguaje imperativo pero sin serlo.

```
1  Right (Spec _) -> evalState (checkWellFormed decls) (TRS M.empty S.empty [] []
       TRSStandard Nothing False)
```

Es por esto que para ejecutar la función ’checkWellFormed’ lo hemos hecho pasándola como parámetro de la función ’evalState’. Cada mónada tiene sus propias funciones de evaluación especializadas, ’evalState’ es una de las que tiene dicha monada estado. ’evalState’ recibe una función y un estado inicial y devuelve el resultado de la función. El estado que se le está pasando es un tipo de datos que se definió en el módulo “Parser.Grammar”. Se puede ver junto al resto de tipos de datos que utilizamos.

```
1 data TRS
2   = TRS { trsSignature :: Map Id Int
3         , trsVariables :: Set Id
4         , trsRMap :: [(Id, [Int])]
5         , trsRules :: [Rule]
6         , trsType :: TRSType
7         , trsStrategy :: Maybe Strategydecl
8         , signatureBlock :: Bool
9         } deriving (Show)
```

Como podemos ver, el tipo ’TRS’ que utilizamos en el estado está formado por varios campos. Los dos primeros campos son dos estructuras de datos una de tipo ’Map’ y otra de tipo ’Set’ que usaremos para guardar los identificadores de las funciones junto a su aridad (’trsSignature’) y los identificadores de las variables (’trsVariables’), respectivamente. Estas estructuras tendrán, durante el análisis semántico, una función similar a una tabla de símbolos. Aunque, en esta aplicación, realmente no tenemos una tabla de símbolos como tal, ya que ni por diseño del servicio ni por necesidad hemos creado una estructura de este tipo que fuera común a todas las fases de análisis.

A continuación, podemos observar dos listas ’trsRMap’ y ’trsRules’, que utilizamos para almacenar tanto la información que aparece en el bloque de mapas de remplazo como las reglas del bloque de reglas. Los que se define en los bloques de ’replacement-map’ son los argumentos que se quieren bloquear de las reglas definidas en el bloque de reglas, es por ello que la información de los mapas de remplazo se compone por una lista en la que cada elemento tiene el identificador

de la función a la que hará referencia el bloqueo de argumentos, acompañada de una lista de enteros indicando las posiciones bloqueadas.

Por último, tenemos 'trsType', 'trsStrategy' y 'signatureBlock'. 'trsStrategy' simplemente contendrá la información sobre el tipo de estrategia que tiene el problema, si es que tiene definida una. 'signatureBlock' nos servirá para saber si existe el bloque de declaración de firmas de función y, en ese caso, comprobar que las funciones que estamos utilizando en las reglas han sido previamente declaradas. Y 'trsType' es, a su vez, otra estructura que hemos definido para poder identificar y devolver el tipo de problema que estamos reconociendo. Dicha estructura, definida también en “Parser.Grammar”, tiene el siguiente aspecto:

```
1 data TRSType =
2   | TRSStandard
3   | TRSEquational
4   | TRSConditional CondType
5   | TRSContextSensitive
6   | TRSContextSensitiveConditional CondType
7     deriving (Show)
```

Es ahora, en la función 'checkWellFormed', cuando vamos a ir comprobando el resto de la semántica, hasta que terminemos de recorrer el contenido de la estructura sintáctica. Cuando encontremos que la estructura esté vacía porque hemos terminado de recorrerla simplemente devolvemos lo obtenido.

```
1 checkWellFormed :: [Decl] -> State TRS (Either ParseError TRS)
2 checkWellFormed [] = do { myTRS <- get
3                        ; return . Right $ myTRS}
```

Comenzamos el análisis con las declaraciones de variables. Primero obtenemos, de la mónada estado, la estructura de datos donde las almacenamos. Mirando dicha estructura comprobamos si las variables que nos llegan han sido ya declaradas, dando un error en tal caso. Hacemos la intersección de lo que hay en 'trsVariables' con lo que tenemos en la estructura del sintáctico, para dar el error en caso de que la intersección no sea vacía. Si no han sido declaradas previamente, las insertamos en nuestra lista de variables ('trsVariables').

```
1 checkWellFormed ((Var vs):rest) =
2    do { myTRS <- get
3       ; let vars = trsVariables myTRS
4       ; let vsSet = S.fromList vs
5       ; let duplicated = S.intersection vsSet vars
6       ; if (not . S.null $ duplicated) then
7           return . Left $ newErrorMessage (UnExpect $ "variable(s) already declared: "
                ++ (concat . intersperse ", " . S.elems $ duplicated)) (newPos "" 0 0)
8         else
9           do{ put $ myTRS { trsVariables = S.union vars vsSet }
10            ; checkWellFormed rest
11            }
12      }
```

Si lo que encontramos durante el recorrido es una declaración de tipo de condición ('CType') lo único que hemos de hacer es extraer esta información guar-

dando el tipo que ha sido declarado y cambiando nuestro 'trsType' de estandar (TRSStandard) a condicional ('TRSConditional').

```
1  checkWellFormed (CType SEMIEQUATIONAL:rest) = do { myTRS <- get
2                                                  ; put $ myTRS { trsType =
                                                       TRSConditional SEMIEQUATIONAL }
3                                                  ; checkWellFormed rest
4                                                  }
5  checkWellFormed (CType OTHER:rest) = do { myTRS <- get
6                                          ; put $ myTRS { trsType = TRSConditional
                                               SEMIEQUATIONAL }
7                                          ; checkWellFormed rest
8                                          }
9  checkWellFormed (CType JOIN:rest) = do { myTRS <- get
10                                        ; put $ myTRS { trsType = TRSConditional JOIN }
11                                        ; checkWellFormed rest
12                                        }
13 checkWellFormed (CType ORIENTED:rest) = do { myTRS <- get
14                                            ; put $ myTRS { trsType = TRSConditional
                                                  ORIENTED }
15                                            ; checkWellFormed rest
16                                            }
```

La siguiente declaración que nos podría llegar es la de estrategia. En el caso de contener las palabras 'INNERMOST', 'OUTERMOST' o 'FULL' (si se trata de un problema en formato XML), únicamente tendremos que introducir dicha información en el tipo 'trsStrategy' del estado.

```
1  checkWellFormed (Strategy INNERMOST:rest) = do { myTRS <- get
2                                                 ; put $ myTRS { trsStrategy = Just
                                                      INNERMOST }
3                                                 ; checkWellFormed rest
4                                                 }
5  checkWellFormed (Strategy OUTERMOST:rest) = do { myTRS <- get
6                                                 ; put $ myTRS { trsStrategy = Just
                                                      OUTERMOST }
7                                                 ; checkWellFormed rest
8                                                 }
9  checkWellFormed (Strategy FULL:rest) = do { myTRS <- get
10                                           ; put $ myTRS { trsStrategy = Just FULL }
11                                           ; checkWellFormed rest
12                                           }
```

De ser 'CONTEXTSENSITIVE' la palabra obtenida, tenemos comprobar que el tipo de formato que tenemos en ese momento no es un 'TRSEquational' ya que de lo contrario sería erróneo porque tendríamos un conflicto de formatos. La declaración 'CONTEXTSENSITIVE' solo puede aparecer en un formato de tipo 'TRSContextSensitive' o 'TRSContextSensitiveConditional'. También debemos guardar, en la lista que hemos creado para ello, la declaración de los mapas de reemplazo que encontramos en esta declaración, comprobando antes que no están repetidos. Y, por último, si en el tipo de formato lo que tenemos es 'TRSStandard' lo cambiaremos por 'TRSContextSensitive' o si es 'TRSConditional' por 'TRSContextSensitiveConditional'.

```
1  checkWellFormed (Strategy (CONTEXTSENSITIVE rmap):rest) =
2       do { myTRS <- get
3          ; if (length . nub . map fst $ rmap) == length rmap then
```

```
4             case trsType myTRS of
5               TRSEquational -> return . Left $ newErrorMessage (UnExpect $ "found theory
                     in non equational type") (newPos "" 0 0)
6               _ -> do{put $ myTRS { trsRMap = rmap
7                                   , trsStrategy = Just FULL
8                                   , trsType = case trsType myTRS of
9                                                 TRSStandard -> TRSContextSensitive
10                                                TRSConditional typ ->
                                                      TRSContextSensitiveConditional typ
11                                                TRSContextSensitive ->
                                                      TRSContextSensitive
12                                                TRSContextSensitiveConditional typ ->
                                                      TRSContextSensitiveConditional typ
13                                  }
14                     ;checkWellFormed rest
15                     }
16          else
17            return . Left $ newErrorMessage (UnExpect $ "duplicated symbols in
                  replacement map declaration") (newPos "" 0 0)
18        }
```

Exactamente igual hacemos si tenemos un 'REPLACEMENT-MAP' en la cadena que estamos analizando. Ya que 'REPLACEMENT-MAP' es exactamente lo mismo que 'CONTEXTSENSITIVE'. Se tratan de los mismos bloques que se usan para declarar las listas de mapas de reemplazo, la única diferencia es que en los formatos de confluencia la palabra reservada para indicarlo es 'REPLACEMENT-MAP' y en los de terminación 'CONTEXTSENSITIVE'. Por ello las acciones de realizamos son las mismas, aunque en este caso no necesitamos comprobar que el formato actual no sea un 'TRSEquational' ya que en la problemas de confluencia no existe.

```
1 checkWellFormed (Context rmap:rest) =
2      do { myTRS <- get
3         ; if (length . nub . map fst $ rmap) == length rmap then
4             do { put $ myTRS { trsRMap = rmap
5                              , trsType
6                                  = case trsType myTRS of
7                                      TRSStandard -> TRSContextSensitive
8                                      TRSConditional typ -> TRSContextSensitiveConditional
                                           typ
9                              }
10              ; checkWellFormed rest
11              }
12          else
13            return . Left $ newErrorMessage (UnExpect $ "duplicated symbols in
                  replacement map declaration") (newPos "" 0 0)
14        }
```

Cuando lo que tengamos sean declaraciones de teoría, del mismo modo, comprobamos que el formato actual no sea 'TRSConditional', 'TRSContextSensitive' o TRSContextSensitiveConditional. Y establecemos el tipo de formato como 'TRSEquational' en el estado.

```
1 checkWellFormed (Theory th:rest) =
2     do { myTRS <- get
3          ; case trsType myTRS of
4              TRSStandard -> do{put $ myTRS { trsType = TRSEquational }
5                                ;checkWellFormed rest
6                                }
```

```
              TRSEquational -> checkWellFormed rest
              _ -> return . Left $ newErrorMessage (UnExpect $ "replacementmap or
                  condition type in equational type") (newPos "" 0 0)
        }
```

En caso de que durante el análisis encontremos que se ha definido un bloque de declaración de funciones ('signature') lo primero que vamos a hacer es poner la variable booleana 'signatureBlock' del estado a 'True', para tenerlo en cuanta cuando analicemos las reglas. Una vez se ha modificado esta variable, entramos a analizar cada una de las funciones declaradas en el bloque llamando a nuestra función auxiliar 'checkSignatures', que recorrerá esta lista.

```
checkWellFormed (Signature sg:rest) = do { myTRS <- get
                                         ; put $ myTRS { signatureBlock = True}
                                         ; result <- checkSignatures sg
                                         ; case result of
                                            Left parseError -> return . Left $
                                                parseError
                                            Right _ -> checkWellFormed rest
                                         }

checkSignatures :: [Signdecl] -> State TRS (Either ParseError ())
checkSignatures [] = do { myTRS <- get
                        ; return . Right $ ()
                        }
checkSignatures (s:ss) = do { result <- checkSignature s
                            ; case result of
                                Left parseError -> return . Left $ parseError
                                Right _ -> checkSignatures ss
                            }
```

Según el tipo de firma de función que nos encontramos durante el recorrido del bloque 'Signature', aplicaremos unas acciones u otras. La firma de función que obtengamos pueden ser de tres tipos, debido a que en el bloque de signatura de la versión XML, a diferencia que en las otras dos gramáticas, es donde se declaran tanto las reglas de teoría como las de los mapas de remplazo.

Si el tipo signatura aparece simplemente como un identificador y la aridad correspondiente, lo que hacemos es comprobar que dicho identificador no está en la lista de variables y a continuación lo insertamos en la lista de funciones. De lo contrario lanzamos un error.

```
checkSignature (S id arity) =
     do { myTRS <- get
        ; let vars = trsVariables myTRS
        ; let funcs = trsSignature myTRS
        ; case (S.member id vars, M.lookup id funcs) of
            (False, Nothing) -> do { put $ myTRS { trsSignature = M.insert id arity $
                funcs }
                                   ; return . Right $ ()
                                   }
            (False, Just len) -> return . Left $ newErrorMessage (UnExpect $ "symbol "
                ++ id ++ " with arity " ++ (show arity) ++ " already in signature") (
                newPos "" 0 0)
            _ -> return . Left $ newErrorMessage (UnExpect $ "symbol declaration in
                variables " ++ id) (newPos "" 0 0)
        }
```

Si junto al identificador de la función y la aridad tenemos un identificador que contiene el tipo de teoría, se hace lo mismo que en el caso anterior, solo que está vez comprobamos que el formato que nos encontramos analizando sea 'TRSEquational'. O lo establecemos como tal si el formato actual es el 'TRSStandard'. Además comprobamos primero que el identificador de teoría sea válido, ya que debe contener la cadena “A”, “C” o “AC”.

```
1  checkSignature (Sth id arity thId) =
2      if (thId == "A") || (thId == "C") || (thId == "AC") then
3        do { myTRS <- get
4             ; let vars = trsVariables myTRS
5             ; let funcs = trsSignature myTRS
6             ; case (S.member id vars, M.lookup id funcs) of
7                 (False, Nothing) ->
8                       do { case trsType myTRS of
9                                TRSStandard -> do { put $ myTRS {trsSignature = M.insert id
                                     arity $ funcs
10                                                                , trsType = TRSEquational
11                                                                }
12                                                  ; return . Right $ ()
13                                                  }
14                               TRSEquational -> do { put $ myTRS {trsSignature = M.insert
                                     id arity $ funcs}
15                                                    ; return . Right $ ()
16                                                    }
17                               _ -> return . Left $ newErrorMessage (UnExpect $ "found
                                     theory in non equational type") (newPos "" 0 0)
18                           }
19                (False, Just len) -> return . Left $ newErrorMessage (UnExpect $ "symbol "
                      ++ id ++ " with arity " ++ (show arity) ++ " already in signature ")
                     (newPos "" 0 0)
20                _ -> return . Left $ newErrorMessage (UnExpect $ "symbols declaration in
                     variables " ++ thId) (newPos "" 0 0)
21            }
22     else
23       return . Left $ newErrorMessage (UnExpect $ "identifier '" ++ thId ++ "' is not
             valid theory declaration") (newPos "" 0 0)
```

La última posibilidad es que haya, junto al identificador y la aridad, una lista de enteros correspondientes a la definición de un mapa de remplazo. Como ya se mencionó dichos enteros estarán indicando las posiciones de los parámetros que se desean bloquear de esa función. En este último caso lo que hacemos es lo mismo que cuando lo analizamos para el TRS tradicional o en los formatos de COPS. Primero comprobamos que el formato actual no es 'TRSEquational'. Después, introducimos el identificador de la función y la lista de enteros en la 'rmap', que es lista del estado global donde los almacenamos. También modificamos en el estado el formato que tenemos ('trsType'), para indicar que es 'ContextSensitive'. Finalmente comprobamos que el identificador de la función no estuviera en la lista de variables y lo introducimos en la lista de funciones.

```
1  checkSignature (Srp id arity intlist) =
2      do { myTRS <- get
3         ; let vars = trsVariables myTRS
4         ; let funcs = trsSignature myTRS
5         ; let rmap = ((id, intlist):(trsRMap myTRS))
6         ; case trsType myTRS of
7             TRSEquational -> return . Left $ newErrorMessage (UnExpect $ "found theory
                  in non equational type") (newPos "" 0 0)
```

```
8              _ -> do{put $ myTRS { trsRMap = rmap
9                                  , trsType = case trsType myTRS of
10                                               TRSStandard -> TRSContextSensitive
11                                               TRSConditional typ ->
                                                     TRSContextSensitiveConditional typ
12                                               TRSContextSensitive ->
                                                     TRSContextSensitive
13                                               TRSContextSensitiveConditional typ ->
                                                     TRSContextSensitiveConditional typ
14                                 }
15                     ; case (S.member id vars, M.lookup id funcs) of
16                         (False, Nothing) -> do { put $ myTRS { trsSignature = M.insert
                               id arity $ funcs }
17                                               ; return . Right $ ()
18                                               }
19                         (False, Just len) -> return . Left $ newErrorMessage (UnExpect $
                                "symbol " ++ id ++ " with arity " ++ (show arity) ++ "
                               already in signature ") (newPos "" 0 0)
20                         _ -> return . Left $ newErrorMessage (UnExpect $ "symbols
                               declaration in variables " ++ id) (newPos "" 0 0)
21                     }
22         }
```

Con el análisis de las reglas de los problemas, terminamos con los bloques a analizar semánticamente. En comparación con el resto, es la parte donde más acciones semánticas comprobamos y aplicamos.

Cuando el analizador comienza a procesar un bloque de reglas lo primero que hacemos es comprobar cada una de las reglas declaradas dentro de los mismos. Esto lo hacemos en la función auxiliar 'checkRules'. Si la comprobación de las reglas no devuelve ningún error simplemente las añadimos a la lista de reglas que almacenamos en el estado.

```
1 checkWellFormed ((Rules rs):rest) = do {result <- checkRules rs
2                                         ; case result of
3                                             Left parseError -> return . Left $ parseError
4                                             Right _ -> do { myTRS <- get
5                                                           ; let rules = trsRules myTRS
6                                                           ; put $ myTRS {trsRules = (rs
                                                                ++ rules) }
7                                                           ; checkWellFormed rest
8                                                           }
9                                         }
```

En la función 'checkRules' vamos a ir recorriendo cada regla declarada. Cuando lleguemos al final y no queden más reglas, en el caso de que estemos trabajando con un 'TRSContextSensitive' o un 'TRSContextSensitiveConditional', es decir, que haya habido un bloque de declaración de mapas de reemplazo, comprobamos esas declaraciones de mapas de remplazo, que previamente analizamos y almacenamos en una lista del estado ('trsRMap'). Lo haremos en la función 'trsRMap'.

```
1 -- | Checks if the rules are well-formed wrt the extracted signature
2 checkRules :: [Rule] -> State TRS (Either ParseError ())
3 checkRules [] = do { myTRS <- get
4                      -- first, we extract the arity of symbols, then we check RMap
5                    ;case trsType myTRS of
6                         TRSContextSensitive -> checkRMap . trsRMap $ myTRS
```

```
7                          TRSContextSensitiveConditional _ -> checkRMap . trsRMap $ myTRS
8                          _ -> return . Right $ ()
9                     }
10 checkRules (r:rs) = do { myTRS <- get
11                         ; let vs = trsVariables myTRS
12                         ; if nonVarLHS vs r then -- lhs (left-hand side of the rule) is
                               non-variable
13                             if isCRule r || (not . hasExtraVars vs $ r) then -- extra
                                   variables not allowed in non-conditional rules
14                                 do { result <- checkTerms . getTerms $ r
15                                    ; case result of
16                                        Left parseError -> return . Left $ parseError
17                                        Right _ -> checkRules rs
18                                    }
19                             else
20                               return . Left $ newErrorMessage (UnExpect $ "extra
                                     variables in the rule " ++ (show r)) (newPos "" 0 0)
21                           else
22                             return . Left $ newErrorMessage (UnExpect $ "variable in the
                                    left-hand side of the rule " ++ (show r)) (newPos "" 0
                                   0)
23                         }
```

Para cada regla vamos a comprobar primero que el término de su izquierda no es una variable. Haremos dicha comprobación pasándole la regla y la lista de variables declaradas a la función 'nonVarLHS'.

```
1 -- | checks if the lhs is non-variable
2 nonVarLHS :: Set Id -> Rule -> Bool
3 nonVarLHS vs (Rule ((T idt _) :-> r) conds) = not . member idt $ vs
4 nonVarLHS vs (Rule ((T idt _) :->= r) conds) = not . member idt $ vs
5 nonVarLHS vs (Rule ((XTerm (Tfun idt _)) :-> r) conds) = not . member idt $ vs
6 nonVarLHS vs (Rule ((XTerm (Tvar idt)) :-> r) conds) = not . member idt $ vs
7 nonVarLHS vs (COPSrule ((T idt _) :-> r) eqs) = not . member idt $ vs
```

Después, hemos de verificar si la regla es condicional o no. Esto lo hacemos en la función 'isCRule'. Y en caso de no ser una regla que contenga condiciones debemos comprobar que el término de la parte derecha de la regla no tenga variables que no encontremos en su parte izquierda. En la función 'hasExtraVars' comprobamos si tiene esas variables extra. Combinándola con la función 'not' aplicamos la condición.

```
1 -- | checks if the rule is conditional
2 isCRule :: Rule -> Bool
3 isCRule (Rule _ []) = False
4 isCRule (COPSrule _ []) = False
5 isCRule _ = True
6
7 -- | checks if the non-conditional rule has extra variables
8 hasExtraVars :: Set Id -> Rule -> Bool
9 hasExtraVars vs (Rule (l :-> r) []) = not . S.null $ getVars vs r \\ getVars vs l
10 hasExtraVars vs (Rule (l :->= r) []) = not . S.null $ getVars vs r \\ getVars vs l
11 hasExtraVars vs (COPSrule (l :-> r) []) = not . S.null $ getVars vs r \\ getVars vs l
12 hasExtraVars _ _ = error $ "Error: hasExtraVars only applies to non-conditional rules"
```

Una vez hemos mirado si la regla cumple las restricciones obtenemos sus términos con la función 'getTerms' y se los pasamos a la función 'checkTerms' para analizar cada uno de ellos.

```
1  -- | gets all the terms from a rule
2  getTerms :: Rule -> [Term]
3  getTerms (Rule (l :-> r) conds) = (l:r:concatMap getTermsCond conds)
4  getTerms (Rule (l :->= r) conds) = (l:r:concatMap getTermsCond conds)
5  getTerms (COPSrule (l :-> r) eqs) = (l:r:concatMap getTermsEq eqs)
6
7  -- | Checks if the terms are well-formed wrt the extracted signature
8  checkTerms :: [Term] -> State TRS (Either ParseError ())
9  checkTerms [] = return . Right $ ()
10 checkTerms (t:ts) = do { result <- checkTerm t
11                        ; case result of
12                            Left parseError -> return . Left $ parseError
13                            Right _ -> checkTerms ts
14                        }
```

En 'checkTerms' recorremos la lista de términos y en 'checkTerm' analizamos cada uno. Para analizar cada término vemos que podemos encontrarnos tres casos distintos en función del formato que estemos tratando. Los términos de la gramática de la versión XML se tratan de forma distinta a los demás.

En el caso que sea un término que no proviene de la definición en XML lo que tendremos será un identificador y otra lista de términos. La parte de la función 'checkTerm' que le aplicamos es la siguiente:

```
1  -- | Checks if the term is well-formed wrt the extracted signature
2  checkTerm :: Term -> State TRS (Either ParseError ())
3  checkTerm (T id terms) =
4       do { myTRS <- get
5          ; let vars = trsVariables myTRS
6          ; let funcs = trsSignature myTRS
7          ; let signature = signatureBlock myTRS
8          ; let arglen = length terms
9          ; case (S.member id vars, M.lookup id funcs) of
10             (False, Nothing) -> do {if (signature) then
11                                       return . Left $ newErrorMessage (UnExpect $ "
                                             symbol: '" ++ id ++ "' not declared in
                                             signature ") (newPos "" 0 0)
12                                     else
13                                       do{ put $ myTRS { trsSignature = M.insert id (
                                             length terms) $ funcs }
14                                         ; checkTerms terms
15                                         }
16                                     }
17             (False, Just len) -> if (arglen == len) then
18                                     checkTerms terms
19                                  else
20                                     return . Left $ newErrorMessage (UnExpect $ "symbol "
                                          ++ id ++ " with arity " ++ (show arglen) ++ " in
                                          term " ++ (show $ T id terms)) (newPos "" 0 0)
21             (True, Nothing) -> if (arglen == 0) then
22                                  return . Right $ ()
23                                  else
24                                  return . Left $ newErrorMessage (UnExpect $ "arguments
                                        in variable " ++ id) (newPos "" 0 0)
25             -- next case is not possible
26             _ -> return . Left $ newErrorMessage (UnExpect $ "variable and function
                     symbols declaration " ++ id) (newPos "" 0 0)
27         }
```

En esta parte cogemos las listas de variables y de funciones de nuestro estado y buscamos en ellas el identificador que nos ha llegado. En el caso de no en-

contrarlo en ninguna lo introducimos en la lista de funciones, siempre que no hayamos encontrado durante el análisis un bloque específico de declaración de funciones porque si no tendríamos en las reglas una función sin declarar. En el caso de que no esté entre nuestras variables pero si lo encontremos en las funciones, siempre y cuando la aridad case con la que teníamos guardada, será correcto y llamaremos recursivamente a esta misma función 'checkTerm'. Para que compruebe la lista de términos que quedan junto al que acabamos de analizar. Por último, si el identificador no está en nuestra lista de funciones y si en la de variables, habremos terminado. Siempre y cuando la aridad que acompaña al identificador, es decir, la longitud de términos que lo acompañan sea igual a cero.

Ya hemos tratado el primero de los posibles tipos de términos que podemos encontrar. Los dos tipos siguientes, que pertenecen al caso especial de un término en la gramática de la versión XML, tienen como peculiaridad que el formato distingue desde el principio aquellos que corresponden con una función y aquellos que son una variable. Lo hace con las etiquetas '<funapp</funapp>' y '<var></var>', respectivamente. Es por esto que, en realidad, en el fragmento que queda de la función 'checkTerm' se realizan las mismas acciones que acabamos de ver pero separando las que son para una variable de las que son para las funciones.

```
checkTerm (XTerm (Tfun id terms)) =
     do { myTRS <- get
        ; let funcs = trsSignature myTRS
        ; let signature = signatureBlock myTRS
        ; let arglen = length terms
        ; case (M.lookup id funcs) of
            Nothing -> do {if (signature) then
                            return . Left $ newErrorMessage (UnExpect $ "symbol: '" ++
                                id ++ "' not declared in signature ") (newPos "" 0 0)
                            else
                            do { put $ myTRS { trsSignature = M.insert id (length terms)
                                 $ funcs }
                                ; checkTerms terms
                                }
                          }
            (Just len) -> if (arglen == len) then
                            checkTerms terms
                          else
                            return . Left $ newErrorMessage (UnExpect $ "symbol " ++ id
                                ++ " with arity " ++ (show arglen) ++ " in term " ++ (
                                show $ T id terms)) (newPos "" 0 0)
        }
checkTerm (XTerm (Tvar id)) = do { myTRS <- get
                                 ; let vars = trsVariables myTRS
                                 ; case (S.member id vars) of
                                    False -> do { put $ myTRS { trsVariables = S.insert
                                        id $ vars }
                                                ; return . Right $ ()
                                                }
                                    _ -> return . Right $ ()
                                  }
```

Como se puede ver las acciones son las mismas, lo único que cambia es que si el identificador es el de una variable no necesitamos comprobar que no vaya acompañada por más términos. No se da esa posibilidad ya que se comprueba

en el análisis sintáctico.

Queda analizar los mapas de remplazo, si es que los había. Lo hacemos siempre una vez se terminan de comprobar las reglas, ya que necesitamos haber extraído esa información antes. En la función 'trsRMap' vamos a recorrer cada una de las funciones que se declararon en ese mapa de remplazo, las cuales que teníamos almacenadas en una lista 'trsRMap' del estado. Lo que hacemos es obtener nuestra lista de funciones del estado y verificamos si la aridad de las reglas declaradas en el mapa de remplazo es congruente con la aridad de las funciones obtenidas en la declaración de reglas o, en caso de haber bloque de declaración de funciones, las obtenidas en ese bloque. Además, comprobamos que la lista de enteros que tenemos por cada identificador en los mapas de remplazo sea una lista creciente que empieza en uno y su valor máximo es la aridad de la función a la que corresponde. Por último, en caso de existir el bloque de declaración de funciones y no encontrar esa función en el conjunto del mapa de remplazo, se lanzará error.

```
1  -- | Checks if the replacement map satisfies arity restriction and increasing order
2  checkRMap :: [(Id, [Int])] -> State TRS (Either ParseError ())
3  checkRMap [] = return . Right $ ()
4  checkRMap ((f,[]):rmaps) = do { myTRS <- get
5                                ; let signature = signatureBlock myTRS
6                                ; case M.lookup f (trsSignature myTRS) of
7                                    Just arity -> checkRMap rmaps
8                                    Nothing -> if (signature) then
9                                                  return . Left $ newErrorMessage (
                                                      UnExpect $ "function symbol " ++ f
                                                      ++ " in replacement map (the symbol
                                                      does not appear in signature
                                                      declaration)") (newPos "" 0 0)
10                                              else
11                                                 checkRMap rmaps
12                               }
13 checkRMap ((f,rmap):rmaps) =
14      do { myTRS <- get
15         ; let signature = signatureBlock myTRS
16         ; case M.lookup f (trsSignature myTRS) of
17             (Just arity) -> let srmap = sort rmap in
18                            if (rmap == srmap) && (head rmap >= 1) && (last rmap <= arity)
                                   then
19                              checkRMap rmaps
20                            else
21                              return . Left $ newErrorMessage (UnExpect $ "replacement map
                                   for symbol " ++ f ++ " (must be empty" ++ (if arity > 0
                                   then " or an ordered list of numbers in [1.." ++ (show
                                  arity) ++ "] " else "") ++ ")") (newPos "" 0 0)
22             Nothing -> if (signature) then
23                            return . Left $ newErrorMessage (UnExpect $ "function symbol "
                                   ++ f ++ " in replacement map (the symbol does not appear
                                  in signature declaration)") (newPos "" 0 0)
24                        else
25                           checkRMap rmaps
26         }
```

4.5. Interfaces del servicio

El servicio desarrollado puede ser utilizado a través de dos interfaces diferentes. Como biblioteca para poder llamar a la función de análisis expuesta desde el código de la herramienta que la necesite. O por línea de comandos (CLI) a modo de programa ejecutable que funciona como validador, al que se le proporciona la ruta del fichero o carpeta a analizar y los parámetros deseados y devolverá el resultado del análisis.

Para exponer construir y empaquetar el servicio con estas dos interfaces se ha configurado Cabal con las siguientes opciones en las que se especifican los módulos que requieren cada una, sus dependencias y el módulo de la interfaz, entre otras cosas.

```
1 Executable rew-syntax-check
2   buildable: True
3   main-is: Main.hs
4   other-modules:
5     Interface.CLI
6     Parser.COPS.TRS.Parser
7     Parser.COPS.TRS.Scanner
8     Parser.TPDB.TRS.Parser
9     Parser.TPDB.TRS.Scanner
10    Parser.TPDB.TRS_XML.Parser
11    Parser.TPDB.TRS_XML.Scanner
12    Parser.Grammar
13    Parser.Parser
14  build-depends:        base > 4, pretty, parsec, syb, containers, mtl, filepath,
      directory
15  hs-source-dirs:       src
16  ghc-options: -O2 -threaded
17  extensions:
18
19 Library
20  exposed-modules: Interface.Library
21  other-modules:
22    Parser.COPS.TRS.Parser
23    Parser.COPS.TRS.Scanner
24    Parser.TPDB.TRS.Parser
25    Parser.TPDB.TRS.Scanner
26    Parser.TPDB.TRS_XML.Parser
27    Parser.TPDB.TRS_XML.Scanner
28    Parser.Grammar
29    Parser.Parser
30  build-depends:        base > 4, pretty, parsec, syb, containers, mtl
31  hs-source-dirs:       src
32  ghc-options: -O2
33  extensions:
```

4.5.1. Biblioteca

Para la interfaz de la biblioteca creamos el módulo “Interface.Library” en el cuál vamos a definir la función que podrá ser utilizada por otras herramientas para consumir nuestro servicio. Esta función recibe dos parámetros, una cadena de caracteres (’s’) que será la que se quiera analizar y un campo formato (’format’), en el que se puede indicar la gramática con la que se quiere analizar la cadena o dejarlo como nulo para que la función pruebe todos los formatos de los que

dispone y de como resultado el primero que sea capaz de analizar correctamente la cadena o error si ninguno ha sido capaz.

```
1  -- | Accepted formats
2  data Format = TPDB | COPS | XMLTPDB
3
4  parser :: String -> Maybe Format -> TRS
5  parser s format =
6      do
7          case format of
8              Just TPDB ->
9                    case parseTPDB s of
10                      Left parseerror
11                         -> error$ "Parse Error (Main): " ++ show parseerror
12                      Right sys
13                         -> sys
14             Just XMLTPDB ->
15                   case parseTPDB_XML s of
16                      Left parseerror
17                         -> error$ "Parse Error (Main): " ++ show parseerror
18                      Right sys
19                         -> sys
20             Just COPS ->
21                   case parseCOPS s of
22                      Left parseerror
23                         -> error$ "Parse Error (Main): " ++ show parseerror
24                      Right sys
25                         -> sys
26             Nothing -> anyParse s
27
28
29 aivailableFormats :: [(String -> Either ParseError TRS)]
30 aivailableFormats = [parseTPDB, parseTPDB_XML, parseCOPS]
31
32 anyParse :: String -> TRS
33 anyParse fdata = checkParser $ map (\(p) -> p fdata) aivailableFormats
34
35 checkParser [] = (error "Error (CLI): Format not supported")
36 checkParser ((Right x):_) = x
37 checkParser ((Left _):xs) = checkParser xs
```

Como decíamos, la función 'parser' es la que exportamos para que pueda ser utilizada. Esta función recibe la cadena a analizar y otro parámetro que se ha definido como 'Maybe Format'. Con esto conseguimos que se le pueda indicar un 'Nothing', en caso de querer probar automáticamente con todos los analizadores, o especificar uno de los tipos definidos en el 'Format', es decir, 'TPDB', 'COPS' o 'XMLTPDB'.

La función auxiliar 'anyParse' es con la que probamos un analizador tras otro en busca de alguno que consiga procesar la cadena. Los analizadores con los que vamos probando son los que tenemos en la lista 'aivailableFormats'.

4.5.2. Validador

La implementación del validador es algo más compleja. Nos basamos en la documentación que se encuentra disponible en la página `https://mail.haskell.org/pipermail/haskell/2004-January/013412.html` para desarrollar el manejo de las opciones del ejecutable.

Lo primero que hacemos es definir los nuevos tipos de datos correspondientes a las opciones que podemos recibir ('Opt').

```
-- | Accepted formats
data Format = TPDB | COPS | XMLTPDB

-- | Command line options
data Opt = Opt { inputName :: String -- ^ Input file name
               , inputContent :: IO String -- ^ Input file content
               , inputDir :: FilePath   -- ^ Input dir path
               , inputFormat :: Maybe Format -- ^ Input format (Nothing implies
                  automatic)
               }
```

A continuación se incluyen las opciones opciones por defecto dando valores a 'Opt'

```
-- | Default parameters
startOpt :: Opt
startOpt
  = Opt { inputName = "foo.trs"
        , inputContent = exitErrorHelp "use -i option to set input"
        -- a simple way to handle mandatory flags
        , inputDir = ""
        , inputFormat = Nothing
        }
```

Por último con las siguientes funciones podemos cambiar dichos valores por defecto, así como devolver un mensaje de ayuda y/o de error, si el usuario no introduce las opciones requeridas.

```
-- | Command line options
options :: [OptDescr (Opt -> IO Opt)]
options = [ Option "h" ["help"]
                   (NoArg (\opt -> exitHelp))
                   "Show usage info"
          , Option "i" ["input"]
                   (ReqArg (\arg opt -> do return opt {inputName = arg
                                                      ,inputContent = readFile arg})
                           "FILE"
                   )
                   "Input TPDB/COPS/XML file"
           , Option "" ["dir"]
                   (ReqArg (\arg opt -> do return opt {inputDir = arg})
                           "DIR"
                   )
                   "Input dir"
          , Option "" ["tpdb"]
                   (NoArg (\opt -> do return opt { inputFormat = Just TPDB })
                   )
                   "Parse a TPDB file"
          , Option "" ["xml"]
                   (NoArg (\opt -> do return opt { inputFormat = Just XMLTPDB })
                   )
                   "Parse a TPDB XML file"
         , Option "" ["cops"]
                   (NoArg (\opt -> do return opt { inputFormat = Just COPS })
                   )
                   "Parse a TPDB XML file"
          , Option "v" ["version"]
                   (NoArg (\_ -> do hPutStrLn stderr "rew-syntax-check, version 0.1"
```

```
31                                     exitWith ExitSuccess))
32                    "Print version"
33           ]
34
35 -- | Help information
36 showHelp :: IO ()
37 showHelp = do prg <- getProgName
38               hPutStrLn stderr (usageInfo prg options)
39               hFlush stderr
40
41 -- | -h (--help) Show help
42 exitHelp :: IO Opt
43 exitHelp = do showHelp
44               exitWith ExitSuccess
45
46 -- | Show error and help information
47 exitErrorHelp :: String -> IO a
48 exitErrorHelp msg = do hPutStrLn stderr msg
49                        hPutStrLn stderr ""
50                        showHelp
51                        exitFailure
52
53 -- | Parse options
54 parseOptions :: IO (Opt, [String])
55 parseOptions = do (optsActions, rest, errors) <- getArgs
56                    >>= return . getOpt RequireOrder options
57                    -- parse the arguments with the given options
58                   when (not (null errors)) $ do mapM_ (hPutStrLn stderr) errors
59                                                 -- show all errors in the stderr output
60                                                 showHelp
61                                                 exitFailure
62                   opts <- foldl (>>=) (return startOpt) optsActions
63                   -- apply actions to the default parameters
64                   return (opts, rest)
```

Para poder crear el validador, además del manejo de opciones que se ha creado en el módulo 'Interface.CLI', necesitaremos realizar la acción adecuada en función de las opciones que hayamos definido. Exportaremos la función 'parseOptions' para poder utilizarla en el módulo 'Main', que es donde vamos a realizar esta lógica.

El módulo 'Main' es el módulo principal del validador, desde aquí son lanzadas el conjunto de funciones requeridas para la ejecución del servicio. Como se puede ver al inicio de esta sección 4.5, este módulo es el indicado como punto de entrada en la configuración de Cabal.

Creamos una función 'main' para interactuar con la entrada y salida de datos. Primero llamamos a la función 'parseOptions' para obtener las opciones que el usuario introduzca por consola al ejecutar la aplicación. Una vez recogemos las opciones las comprobamos, llamando a la función 'parseFiles' en caso de que la ruta que se reciba corresponda a la de un directorio o a 'callParse' la ruta obtenida pertenece a un archivo.

```
1 -- | The 'main' function parses an input file or dir
2 main :: IO ()
3 main =
4    do (opts, _) <- parseOptions
5       let Opt { inputName = filename
6               , inputContent = input
7               , inputDir = dir
```

```
8                 , inputFormat = format } = opts
9
10       existDir <- doesDirectoryExist dir
11       if (existDir) then
12          do {dirPaths <- listDirectory dir
13             ;parseFiles dir (dirPaths) format
14             }
15       else
16          do{filedata <- input
17            ;let !trs = callParse filename filedata format
18            ;hPutStr stdout ("\n Success:\n" ++ show trs ++ "\n\n")
19            }
```

En el caso de haber recibido un único archivo, con 'callParse' simplemente lo analizamos según las opciones de formato recibidas y devolvemos el resultado. Si hemos recibido un formato específico llamaremos directamente al analizador correspondiente. De lo contrario, utilizaremos la función 'autoparse' en la que se mira la extensión del nombre del archivo y se compara con las extensiones que tenemos censadas en la lista 'aivailableFormats' ejecutando el analizador que corresponda a dicha extensión.

```
1 callParse :: String -> String -> Maybe Format -> TRS
2 callParse filename filedata format= do
3    case format of
4             Just TPDB ->
5                   case parseTPDB filedata of
6                      Left parseerror
7                         -> error$ "Parse Error (Main): " ++ show parseerror
8                      Right sys
9                         -> sys
10            Just XMLTPDB ->
11                  case parseTPDB_XML filedata of
12                     Left parseerror
13                        -> error$ "Parse Error (Main): " ++ show parseerror
14                     Right sys
15                        -> sys
16            Just COPS ->
17                  case parseCOPS filedata of
18                     Left parseerror
19                        -> error$ "Parse Error (Main): " ++ show parseerror
20                     Right sys
21                        -> sys
22            Nothing -> autoparse filename filedata
23
24
25 -- | File extensions
26 aivailableFormats :: [(String, String -> Either ParseError TRS)]
27 aivailableFormats = [(".trs", parseTPDB), (".xml", parseTPDB_XML), (".trs", parseCOPS)]
28
29 -- | Parse file into a TRS
30 autoparse :: String -> String -> TRS
31 autoparse fname = maybe (error "Error (CLI): File Extension not supported")
32                         parseWithFailure
33                         matchParser
34    where matchParser
35              = msum $ map (\(ext,p)-> if fname `endsWith` ext then Just p else Nothing)
                    aivailableFormats
36          endsWith
37              = flip isSuffixOf
38          parseWithFailure parser contents
39              = case parser contents of
40                  Left parseerror
41                     -> error$ "Parse Error (CLI): " ++ show parseerror
```

```
42                  Right sys
43                      -> sys
```

La otra opción es que la ruta pertenezca a un directorio, por lo que en la función 'parseFiles' hacemos lo mismo que acabamos de ver, pero recorriendo cada uno de los archivos del directorio. Utilizamos la función 'toParse' junto a, de nuevo, la lista 'aivailableFormats' para obviar los archivos del directorio que no nos interesan porque no se corresponden con algún formato a analizar. Tras esto por cada uno de los ficheros que nos vamos encontrando llamamos, de nuevo, a 'callParse' para que ejecute el análisis en función de las opciones de formato introducidas. Por último mostraremos los resultados en la consola además de guardarlos en un archivo de texto dentro del mismo directorio, el cuál nombramos como 'parser_results.txt'.

```
1  -- | Parse all files in the given directory
2  parseFiles :: FilePath -> [FilePath] -> Maybe Format -> IO ()
3  parseFiles _ [] _ = hPutStr stdout (" ------------- END OF DIR ------------- ")
4  parseFiles dirPath (filepath:rest) format= do
5        if ((toParse filepath aivailableFormats)) then
6           do {let absPath= dirPath </> filepath
7              ;input <- readFile absPath
8              ;hPutStr stdout  ("\n++ File:" ++ show absPath ++ " :\n")
9              ;let !trs = callParse filepath input format
10             ;hPutStr stdout  ("Success: " ++ show trs ++ "\n")
11             --Write results
12             ;let trsOut = ("\n++ File:" ++ show absPath ++ " :\n" ++ show trs ++ "\n")
13             ;let writePath = dirPath </> "parser_results.txt"
14             ;appendFile writePath trsOut
15             ;parseFiles dirPath rest format
16             }
17       else
18          parseFiles dirPath rest format
19
20
21 toParse filename [] = False
22 toParse filename ((ext,_):xs) | ext `isSuffixOf` filename = True
23                               | otherwise = toParse filename xs
```

4.6. Repositorios del proyecto

El servicio se encuentra disponible en un repositorio público de GitHub: `https://github.com/juanpahgj/tfg_2022.git`, en el que se puede ver toda la evolución del proyecto. Junto al código de la aplicación se encuentran los problemas que han sido utilizados en las pruebas y los reportes emitidos tras los análisis.

Para compilar el proyecto debe tenerse instalado Cabal. En la página de la herramienta se explica cómo hacerlo `https://cabal.readthedocs.io/en/3.4/getting-started.html`. Una vez instalado se debe ejecutar los mandatos "cabal new-configure" seguido de " cabal new-clean; cabal new-build" para la compilación y empaquetamiento.

El ejecutable resultante se llamará "rew-syntax-check.exe" y la herramienta lo genera en una subcarpeta dentro del directorio "./dist-newstyle/build/". El mismo ejecutable puede usarse tanto desde la consola de comandos, a modo de

validador, como desde otra herramienta, a modo de biblioteca. Contiene ambas interfaces.

El analizador base utilizado como punto de partida para este trabajo se encuentra disponible en: `https://github.com/raulgut/csrs-syntax-checker.git`

4.7. Trabajo futuro

Como trabajo futuro se podría extender el servicio para poder analizar formatos como el 'HRS', 'MSTRS' o 'SRS', que no ha dado tiempo a incluir en este trabajo. Así como incluir el reconocimiento de los formatos particulares que encontramos en las categorías como la de 'COM' e 'INF' de la competición de confluencia (CoCo).

Además, ya que actualmente se extrae y devuelve la información que contiene el problema que se analiza, eliminando las características particulares del formato, dicha información podría ser representada conforme a las características de otro formato. Una interesante funcionalidad que se podía añadir es la de traducción entre formatos.

Capítulo 5

Conclusiones y Resultados

En este trabajo se ha desarrollado un servicio que permite unificar las entradas de diferentes tipos de sistemas de reescritura, de las competiciones internacionales de terminación y confluencia, en una estructura de datos homogénea. Gracias a esto, podemos comprobar si un sistema escrito en formato TPDB o formato COPS es correcto y, en caso contrario, mostrar un error y la línea incorrecta. Este, además, se define como una biblioteca en la implementación, lo que puede permitir que otras herramientas hagan uso de ella como interfaz de entrada a sus herramientas.

El servicio desarrollado se ha escrito en Haskell utilizado un enfoque, que encontramos en la programación funcional, por el cual se modelan analizadores como funciones y se definen como de orden superior para implementan las construcciones gramaticales a base de combinarlas entre ellas (combinadores). El desarrollo se ha hecho utilizando Parsec, biblioteca para escribir analizadores en Haskell basada en estos combinadores de analizadores de orden superior.

5.1. Resultados

La aplicación resultante tiene alrededor de 1683 líneas de código, repartidas en los once módulos que la componen. Cuenta con la capacidad de reconocer un total de cuatro sistemas de reescritura de términos diferentes definidos sobre tres gramáticas distintas, lo que hace una cantidad de once tipos de formato distintos. Esto significa que la funcionalidad de análisis que este servicio ofrece, a las herramientas que participan en ambas competiciones, cubre el setenta por ciento de las categorías que acogen actualmente estas competiciones.

En las tablas 5.1 y 5.1 se exhiben dichas categorías correspondientes a cada competición. Se muestran en verde las categorías donde el servicio es capaz de reconocer el formato que utilizan, en amarillo las que la aplicación es capaz de reconocer el formato, pero todavía tiene algún error funcional que se debe corregir, y en rojo las que tienen formatos que no se reconocen.

Confluence Competition Categories
COM (commutation problems)
CPF-TRS (certified confluence of term rewriting)
CPF-CTRS (certified confluence of conditional rewriting)
CSR (confluence of context-sensitive rewriting)
CTRS (confluence of conditional rewriting)
GCR (ground confluence)
HRS (confluence of higher-order rewrite systems)
INF (infeasibility problems)
NFP (normal form property)
SRS (confluence of string rewriting)
TRS (confluence of term rewriting)
UNC (unique normal forms wrt conversion)
UNR (unique normal forms wrt reduction)

Cuadro 5.1: Categorías en la Competición de Confluencia

termCOMP - Termination of Rewriting Categories
TRS Standard
TRS Standard Certified
TRS Relative
TRS Relative Certified
SRS Standard
SRS Standard Certified
SRS Relative
SRS Relative Certified
TRS Equational
TRS Equational Certified
TRS Conditional
TRS Context Sensitive
TRS Innermost
TRS Innermost Certified
TRS Outermost
TRS Outermost Certified
HRS (union beta)

Cuadro 5.2: Categorías en la Competición de Terminación

5.1.1. Pruebas

Se han realizado pruebas del servicio en las que han sido analizados cerca de 4500 problemas distintos. Estos problemas has sido descargados de los repositorios de problemas que encontramos tanto en TPDB [1] como en COPS [2]. Todos los problemas analizados se pueden encontrar en el repositorio, dentro de la carpeta 'tests' que se encuentra junto al código del servicio. Entre todos los archivos se han encontrado un total de setenta y cuatro que no cumplen con las especificaciones del formato. Estos los podemos encontrar en las subcarepetas

nombradas como “files_with_error”. Junto al nombre del fichero se indica el error encontrado. El resto de los problemas han sido reconocidos como correctos. Se encuentran divididos en subcarpetas de la misma forma que se encontraban en los repositorios de las competiciones. En cada directorio, la herramienta ha generado un archivo “parser_results.txt” que contiene el resultado del análisis de los problemas que contiene la carpeta.

Capítulo 6

Análisis de impacto

En este trabajo se ha desarrollado un servicio con la funcionalidad de lectura y análisis de los distintos formatos de problemas que encontramos en las competiciones de terminación “termCOMP” y de confluencia “CoCo”. Ofrecer dicha funcionalidad de forma externa, homogénea y fácilmente implementable presenta una clara mejora funcional que afecta a herramientas utilizadas en el marco del análisis de programas y propiedades computacionales.

Propiedades como la terminación, advierten de si cabe la posibilidad de una ejecución indefinida en un programa u aplicación, lo que conlleva un gasto energético y de recursos. Por tanto, este proyecto y el servicio de análisis que en él se desarrolla contribuye, indirectamente, con el Objetivo de Desarrollo Sostenible número doce, "Producción y Consumo Responsable"que se muestra en la figura 6.1. Ayuda a evitar que se desaprovechen recursos y al ahorro de energía en los sistemas informáticos.

Figura 6.1: Objetivos de Desarrollo Sostenible

Bibliografía

[1] J.waldmann. (2009) Termination problems data base. [Online]. Available: https://termination-portal.org/wiki/TPDB

[2] Confluence problems (cops). [Online]. Available: https://cops.uibk.ac.at

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compiladores: principios, técnicas y herramientas*, 2nd ed. Pearson Educación, 2007.

[4] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada, "The termination and complexity competition," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 156–166.

[5] C. O. Peter Schneider-Kamp. (2008) Termination portal. [Online]. Available: https://termination-portal.org/wiki/Termination_Portal

[6] N. Hirokawa, J. Nagele, and A. Middeldorp, "Cops and cocoweb: Infrastructure for confluence tools," in *International Joint Conference on Automated Reasoning*. Springer, 2018, pp. 346–353.

[7] Confluence competition (coco). [Online]. Available: http://project-coco.uibk.ac.at/

[8] R. Gutiérrez and S. Lucas, "mu-term: Verify termination properties automatically (system description)," in *International Joint Conference on Automated Reasoning*. Springer, 2020, pp. 436–447.

[9] F. Mitterwallner, J. Hochrainer, and A. Middeldorp, "Coco 2021 participant: Fort-h 1.1," in *Proc. 10th International Workshop on Confluence*, 2021.

[10] R. Gutiérrez and S. Lucas, "Automatically proving and disproving feasibility conditions," in *International Joint Conference on Automated Reasoning*. Springer, 2020, pp. 416–435.

[11] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real world haskell: Code you can believe in*. .O'Reilly Media, Inc.", 2008.

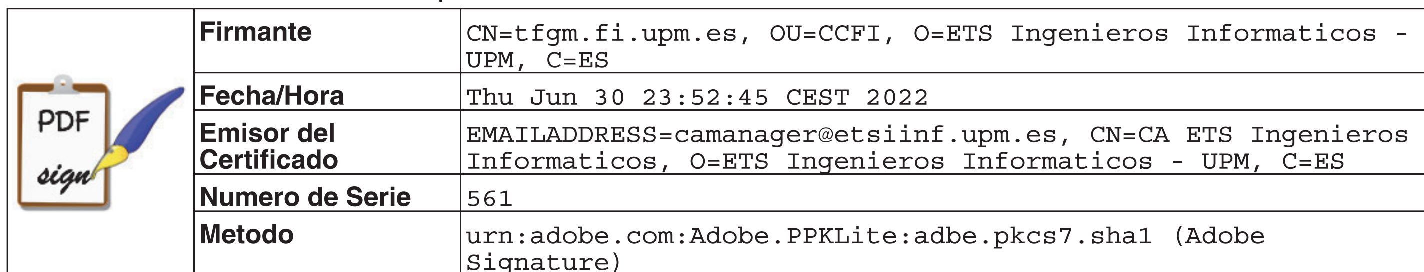
[12] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge university press, 1999.

[13] G. Hutton and E. Meijer, "Monadic parser combinators," 1996.

[14] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001.

[15] D. Leijen, "Parsec, a fast combinator parser," 2001.

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Thu Jun 30 23:52:45 CEST 2022
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)