Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

Soft set based rewrite system for symbolic biological systems modeled with Pathway Logic in Maude

Trabajo de Fin de Máster
Curso 2020–2021

Autor
Rocío M. Santos Buitrago

Director
Adrián Riesco Rodríguez

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

Soft set based rewrite system for symbolic biological systems modeled with Pathway Logic in Maude

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación
(Software Systems and Computation)

Autor
Rocío M. Santos Buitrago

Director
Adrián Riesco Rodríguez

Convocatoria: *Junio 2021*
Calificación: *7,5 Notable*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

12 de julio de 2021

Dedicatoria

A mis padres, Beatriz y Gustavo, por su apoyo incondicional a lo largo de mi vida

Agradecimientos

Estudiar este máster ha sido un desafío importante en mi vida, que habría sido difícil de superar si no hubiera sido por el apoyo y el ánimo de mucha gente.

Quiero expresar mi agradecimiento, en primer lugar, a mi tutor, Adrián Riesco, por su guía, dedicación y apoyo en este trabajo de fin de máster: GRACIAS!!!

Estoy muy agradecida a todo el profesorado del máster, en especial en este último año, por su comprensión ante mis excepcionales circunstancias personales.

También quiero dar las gracias a Francisco J. López Fraguas, mi profesor de programación declarativa en el grado, y también tengo un agradecimiento especial a todos los miembros del grupo Maude en UCM. Tengo buenos recuerdos de la disposición y paciencia de Alberto Verdejo para resolver mis dudas. También muchas gracias a José Meseguer, como padre de Maude.

Por supuesto estaré siempre agradecida a Narciso Martí, por sus incontables y muy buenos consejos académicos y personales.

Por último, quiero agradecer a mi familia por su cariño y paciencia. A mis padres y mis abuelos por apoyar mis intereses académicos y por animarme a conseguir todos mis propósitos. A mis hermanos Bea, Marta, Patu y Gus, por estar siempre a mi lado y animarme en cada reto.

Resumen

Sistema de reescritura basado en soft sets para sistemas biológicos simbólicos modelados con Pathway Logic en Maude

Pathway Logic es una herramienta para tratar con sistemas biológicos simbólicos desarrollada en SRI International. Está basada en redes de Petri e implementada en el lenguaje de reescritura Maude. Con esta herramienta se han desarrollado numerosos modelos de rutas de señalización celular. Estos modelos constituyen una base de conocimiento formal que contiene información sobre los cambios que tienen lugar en las proteínas dentro de una célula en respuesta a la reacción de ligandos/receptores, sustancias químicas o diversas tensiones. El objetivo principal de este trabajo consiste en desarrollar e implementar nuevas variantes de reescrituras basadas en soft sets en el contexto de Pathway Logic. La instrucción de reescritura estándar en Maude permite examinar los términos a través del árbol de reescrituras partiendo de un estado inicial. Otro objetivo es mejorar y actualizar la entrada/salida. Para esa reescritura de términos, se proponen nuevas funciones de elección en el árbol de búsquedas con soft sets, que proporcionan ventajas sobre la toma de decisiones bajo información incompleta. La implementación propuesta se lleva a cabo con el lenguaje Maude y se utilizan las nuevas funcionalidades en el metaintérprete y la gestión de entrada/salida. Por último, este trabajo incluye un análisis de los resultados del sistema propuesto respecto del sistema de reescritura estándar y de otros enfoques existentes en la literatura.

Palabras clave

Lógica de reescritura, Maude, Metalenguaje, Pathway Logic, Sistemas biológicos simbólicos, Soft sets, Información incompleta, Toma de decisiones, Sistema de búsqueda.

Abstract

Soft set based rewrite system for symbolic biological systems modeled with Pathway Logic in Maude

Pathway Logic is a tool for dealing with symbolic biological systems developed at SRI International. It is based on Petri nets and the Maude rewriting language. Numerous cellular signaling pathway models have been developed with this tool. These models constitute a formal knowledge base that contains information about the changes that take place in proteins within a cell in response to the reaction of ligands/receptors, chemicals or various stresses. The main objective of this dissertation is to develop and implement new rewriting variants based on soft sets in the context of Pathway Logic. The standard rewrite instruction in Maude allows the examination of terms through the rewrite tree starting from an initial state. Another objective is to improve and upgrade the input/output. For this rewriting of terms, new choice functions are proposed in the search tree by using soft sets, which provide advantages over decision making under incomplete information. The proposed implementation is carried out with the Maude language and the new functionalities in the meta-interpreter and the input/output management are used. Finally, this dissertation includes an analysis of the results of the proposed system with respect to the standard rewriting system and other existing approaches in the literature.

Keywords

Rewriting logic, Maude, Metalanguage, Pathway Logic, Symbolic biological systems, Soft sets, Incomplete information, Decision making, Search system.

Índice

1. Introducción **1**

1.1. Motivación . . . 1
1.2. Objetivos . . . 2
1.3. Plan de trabajo . . . 3
1.4. Organización de la memoria . . . 3

2. Estado de la cuestión **5**

2.1. Maude . . . 5
2.2. Reescritura con objetos externos . . . 8
2.3. Metaintérprete . . . 9
2.4. Reflexión y `META-LEVEL` en Maude . . . 10
2.5. Soft sets . . . 12
2.5.1. Soft sets incompletos . . . 13
2.5.2. Toma de decisiones con *soft sets* . . . 14
2.6. Sistemas biológicos simbólicos y Pathway Logic . . . 17
2.6.1. Modelos en Pathway Logic: ecuaciones . . . 18
2.6.2. Modelos en Pathway Logic: reglas de reescritura . . . 22

3. Descripción del trabajo **25**

3.1. Soft sets y funciones de elección . . . 26
3.2. Definición y reescritura de *softdish* . . . 28
3.2.1. Operador `rewStrat` . . . 28
3.2.2. Operador `narrowStrat` . . . 29
3.3. Módulos de Pathway Logic . . . 31
3.4. Entorno de ejecución con IO y metaintérprete en PLSS . . . 32
3.5. Comandos de simplificación con ecuaciones y reglas *soft* . . . 37
3.6. Comando de búsqueda con narrowing . . . 39

3.7. Otros comandos de PLSS . . . 42
3.7.1. Comando de carga de ficheros con instrucciones . . . 43
3.7.2. Comando de asignación de la función de elección . . . 44
3.7.3. Comando de selección del modelo de Pathway Logic . . . 44
3.7.4. Comando del sistema de ayuda . . . 45
3.8. Integración de los módulos en PLSS . . . 45
3.9. Instalación y ejecución de PLSS . . . 47

4. Análisis de resultados **53**
4.1. Análisis y comparación con las funciones de elección . . . 53
4.2. Análisis y comparación con otros sistemas de búsqueda . . . 55
4.2.1. Caso 1: error en el análisis de los términos . . . 56
4.2.2. Caso 2: sin solución . . . 56
4.2.3. Caso 3: ejemplo del comando `softnarrowsearch` con solución . . . 57
4.3. Análisis y comparación entre comandos de reescritura/estrechamiento . . . 58
4.3.1. Caso: modelo biológico Tgfb1 . . . 58
4.4. Análisis de sistemas biológicos en Pathway Logic . . . 60

5. Trabajos relacionados **63**

6. Conclusiones y trabajo futuro **67**
6.1. Conclusiones del trabajo . . . 67
6.2. Líneas de Trabajo Futuro . . . 68

7. Introduction **69**
7.1. Motivation . . . 69
7.2. Objectives . . . 70
7.3. Work plan . . . 70
7.4. Organization of the dissertation . . . 71

8. Conclusions and future work **73**
8.1. Conclusions of the work . . . 73
8.2. Lines of future work . . . 74

Bibliografía **75**

Índice de figuras

2.1. Ruta de señalización TGF-β . . . 18
2.2. Representación esquemática de una célula . . . 21
2.3. Pathway Logic Assistant . . . 22
2.4. Regla de reescritura `931` en Pathway Logic Assistant . . . 23

3.1. Diagrama de estados de PLSS . . . 34
3.2. Dependencia de ficheros Maude . . . 46
3.3. Dependencia de otros módulos . . . 47
3.4. Dependencia de módulos específicos de Pathway Logic . . . 48
3.5. Dependencia de módulos de uso general de PLSS . . . 49

Índice de tablas

2.1. Representación tabular del *soft set* incompleto 14
2.2. Tablas completadas para un *soft set* incompleto 16
2.3. Soluciones para el problema representado por (F_0, E_0) 16

4.1. Modelos biológicos implementados en PLSS 61

Índice de listados

2.1. Definición en Maude de la sopa de localizaciones . . . 19
2.2. Conjunto de modificaciones en Maude . . . 20
2.3. Definición de las modificaciones en Maude . . . 20
2.4. Placa de preparación `SmallDish` . . . 21
2.5. Dish `Tgfb1Dish` . . . 21
2.6. Regla de reescritura `931.TgfbR1.TgfbR2.by.Tgfb1` . . . 23
3.1. Funciones `undefZero` y su función auxiliar `compUndefZero` . . . 27
3.2. Operadores `rewStrat` y `next` . . . 28
3.3. Función `allReachableTerms` . . . 29
3.4. Función `allReachableNTerms` . . . 30
3.5. Función `putNarrowingAtt` . . . 30
3.6. Módulo `QQ1` . . . 31
3.7. Módulo `TGFB1DISH` . . . 32
3.8. Reglas en el módulo `ALLRULES` . . . 32
3.9. Comando `red` en PLSS. . . . 37
3.10. Comando `softrew` en PLSS. . . . 38
3.11. Regla `softnarrowstratComm`. . . . 39
3.12. Regla `softNarrowSearchComm` en el caso 1. . . . 40
3.13. Regla `softNarrowSearchComm` en el caso 2. . . . 41
3.14. Regla `softNarrowSearchComm` en el caso 3. . . . 42
3.15. Comando `load` en PLSS. . . . 43
4.1. Dish `EgfDish` . . . 61
4.2. Dish `NgfDish` . . . 62

Capítulo 1

Introducción

"Bueno, algunas veces yo he creído hasta
seis cosas imposibles antes del desayuno"
— Lewis Carroll, Alicia en el País de las Maravillas

Este capítulo comienza con una breve introducción al tema de este proyecto. A continuación se dedica una sección a los objetivos y al plan de trabajo. Por último, se describe la organización de la memoria.

1.1. Motivación

Muchos problemas de la vida real requieren el uso de datos imprecisos o desconocidos. Su análisis debe implicar la aplicación de principios matemáticos capaces de captar estas características. En el campo de la Inteligencia Artificial, la teoría de *soft sets* proporciona un marco de trabajo apropiado para la toma de decisiones en las situaciones de falta de información.

El lenguaje Maude permite especificar la parte estática de un sistema mediante una lógica ecuacional y la parte dinámica mediante reglas de reescritura. Las especificaciones son ejecutables y Maude dispone de un comando de búsqueda en anchura. Sin embargo, esta búsqueda estándar de Maude recorre todos los términos alcanzables del árbol de búsqueda, sin permitir estrategias avanzadas, por lo que no es adecuada para algunas aplicaciones donde es deseable elegir priorizar una rama entre todas las posibles. A partir de esta situación, en este proyecto se propone implementar un entorno de ejecución con diferentes estrategias en el sistema de reescritura que permitan escoger la mejor ruta de reescritura según ciertos criterios. Además, teniendo en cuenta que la información disponible puede ser incompleta, se han definido las estrategias mediante la teoría de *soft sets*.

Por otra parte, Maude es un lenguaje de especificación que permite definir y expresar de forma natural una amplia gama de aplicaciones. Pathway Logic es una herramienta basada en Maude y diseñada para tratar con sistemas biológicos simbólicos. Un sistema biológico simbólico es una red de elementos en el que sus relaciones se definen de forma abstracta. De esta forma, los métodos formales basados en lógica de reescritura son apropiados para tratar y analizar el comportamiento de estos sistemas.

Con Pathway Logic se han desarrollado numerosos modelos de rutas de señalización celular. De forma abreviada, una ruta de señalización celular es un sistema biológico en

el que definimos las partes o localizaciones que existen en una célula (e.g., membrana celular, citoplasma, núcleo, etc.) y los elementos moleculares que pueden aparecer en las distintas partes de la célula (e.g., proteínas, genes, etc.). En la ruta de señalización también definimos el comportamiento dinámico o interacciones entre los elementos de la célula (e.g., la unión entre un ligando y un receptor, la activación de una proteína, etc.). Pathway Logic facilita la comprensión de los sistemas biológicos complejos y la verificación de hipótesis en el diseño de experimentos.

Con la intención de proporcionar una aplicación con utilidad real, la implementación del sistema de reescritura propuesto se ha desarrollado con los modelos biológicos de Pathway Logic. De esta forma, su sistema de toma de decisiones basado en *soft sets* nos permite escoger dentro de todos los cambios dinámicos posibles, aquellos que sean más probables, incluso cuando no se disponga de toda la información celular. Se ha utilizado este sistema de búsqueda con algunos modelos biológicos de Pathway Logic, por ejemplo con la ruta de señalización celular del factor de crecimiento epitelial, para determinar la existencia de modificaciones de algunas proteínas asociadas a células cancerígenas.

En el desarrollo de este sistema se han utilizado las nuevas funcionalidades de la nueva versión de Maude con el metaintérprete y la gestión de entrada/salida, que permiten manejar de forma eficiente un entorno de ejecución dentro de Maude.

1.2. Objetivos

El objetivo principal de este proyecto consiste en desarrollar e implementar variantes de reescrituras y búsquedas dirigidas basadas en *soft sets* en el contexto de Pathway Logic. Para gestionar los cambios que se producen en los sistemas biológicos y la necesidad de interacción con objetos externos, la implementación se lleva a cabo con el lenguaje Maude 3.1, aprovechando las nuevas funcionalidades de esta versión.

Se pretenden definir reescrituras dirigidas que nos permitan elegir las rutas de los términos alcanzables a partir de estrategias basadas en *soft sets*. Estas estrategias se utilizan incluso en las situaciones de información incompleta de algunos de los atributos. Los atributos en los términos y en las reglas de reescritura nos permiten definir los criterios para la elección del mejor término, que en nuestro caso de rutas de señalización biológica, será el término o patrón final más probable.

El estrechamiento (*narrowing*) es una alternativa de la reescritura que funciona cambiando encaje de patrones por unificación. Se han implementado comandos de búsqueda con técnicas de reescritura y de estrechamiento, en las que se utilizan *soft sets* para dirigir las búsquedas, tanto normales como simbólicas.

Las metas intermedias para alcanzar este objetivo se pueden especificar en:

1. Implementar una aplicación de entrada/salida en Maude con su sistema de comandos propios.

2. Integrar la aplicación con los modelos de Pathway Logic.

3. Definir estrategias de los *soft sets* para guiar la reescritura.

4. Extender los ejemplos para incluir *soft sets* y analizar el funcionamiento de las distintas estrategias de búsqueda.

1.3. Plan de trabajo

La realización del proyecto se ha apoyado en el plan de trabajo con el director. Las primeras reuniones se dedicaron a la concreción del tema y objetivos del proyecto. Después, cada dos semanas hemos tenido reuniones en las que se repasaban las tareas definidas en la reunión anterior. Por parte del director se realizaban correcciones y se proponían mejoras. En cada reunión también se concretaba el trabajo a realizar durante las dos semanas siguientes.

Aparte de las reuniones ordinarias, las dudas puntuales sobre cualquier aspecto del trabajo se han resuelto por parte del director por correo electrónico.

Los hitos que se establecieron para alcanzar los objetivos son:

1. Búsqueda y definición inicial del tema del trabajo. Esta tarea se extendió desde finales de septiembre de 2019 hasta mediados de octubre de 2019.

2. Profundizar en el lenguaje Maude. Durante los dos meses siguientes me dediqué a estudiar a fondo los temas de Maude necesarios para este proyecto.

3. Investigación sobre Pathway Logic y *soft sets*. Este estudio se alargó hasta marzo de 2020.

4. Diseño e implementación de un prototipo básico de la aplicación. Una versión preliminar funcionó a finales de julio de 2020.

5. Ampliación progresiva de las funcionalidades (tanto en los comandos como en las estrategias). Me dediqué a esta tarea durante el último trimestre de 2020.

6. Realización de pruebas. Los meses de enero y febrero de 2021 llevé a cabo las pruebas del programa.

7. Redacción de la memoria del proyecto. De forma intensiva, empecé la redacción en marzo de este año.

1.4. Organización de la memoria

La memoria está dividida en los siguientes capítulos:

- **Capítulo 1**: Introducción. Se introduce el tema del TFM, se describen los objetivos del trabajo y se detalla el plan de trabajo seguido para la consecución de los objetivos.

- **Capítulo 2**: Estado de la cuestión. En este capítulo se introducen las áreas en las que se fundamenta el trabajo: el lenguaje de programación Maude, el metaintérprete de Maude, la teoría de *soft sets* y Pathway Logic.

- **Capítulo 3**: Descripción del trabajo. En este capítulo se describe el trabajo realizado en el proyecto **PLSS**: **P**athway **L**ogic con **S**oft **S**ets.

- **Capítulo 4**: Análisis de resultados. En este capítulo se realiza un análisis comparativo de los resultados obtenidos con los del uso estándar de la reescritura.

- **Capítulo 5**: Trabajos relacionados. En este capítulo se discuten los trabajos relacionados con el proyecto.
- **Capítulo 6**: Conclusiones y trabajo futuro. Por último, se han expuesto las conclusiones y las líneas para un trabajo futuro.

El código elaborado para este proyecto está disponible en el repositorio de GitHub `https://github.com/rsantosb/TFM-PLSS`, bajo la licencia MIT (`https://opensource.org/licenses/MIT`).

Capítulo 2

Estado de la cuestión

"Cuando comas brotes de bambú,
recuerda al hombre que los plantó"
— Proverbio chino

En este capítulo se introducen las áreas en las que se fundamenta el trabajo. La sección 2.1 trata del lenguaje de programación Maude. Las secciones 2.2 y 2.3 se refieren a la reescritura con objetos externos y al metaintérprete de Maude. La sección 2.4 describe la reflexión y el metanivel en Maude. La sección 2.5 trata sobre la teoría de *soft sets*. Por último, la sección 2.6 introduce Pathway Logic, el entorno de trabajo que se ha utilizado para la modelización de los sistemas biológicos.

2.1. Maude

Maude (Clavel et al., 2020) es un lenguaje de programación para especificaciones formales mediante el uso de términos algebraicos. Una importante ventaja de Maude es que es ejecutable, por lo que las especificaciones sirven como prototipos. Las declaraciones de programación básicas de Maude son ecuaciones y reglas, que en ambos casos tienen una semántica de reescritura simple donde las instancias del patrón del lado izquierdo se reemplazan por las instancias correspondientes del lado derecho. Maude tiene una biblioteca de módulos predefinidos entre los que destacamos:

- El módulo `QID` proporciona el tipo `Qid` de los identificadores que empiezan con comilla (`'`), junto con operadores básicos para estos identificadores, tales como la concatenación, la indexación de un identificador por un entero o la eliminación del primer carácter después de la comilla (Clavel et al., 2020).

  También resulta útil disponer de un tipo de datos de listas de estos identificadores con comilla. El módulo `QID-LIST` extiende el módulo `QID` con el tipo `QidList` de las listas de identificadores con comilla (Clavel et al., 2020).

- El módulo `CONFIGURATION` proporciona tipos básicos y constructores para modelar sistemas basados en objetos. Una configuración es una "sopa" o multiconjunto de objetos y mensajes que representa un posible estado del sistema (Clavel et al., 2020).

- El módulo `META-LEVEL`, que se comenta en detalle en la sección 2.3.

Un *módulo funcional* es un programa de Maude que únicamente tiene ecuaciones, dicho de otra forma, se declaran con ecuaciones una o más funciones. Las ecuaciones se especifican en Maude con la palabra clave `eq` y terminan con un punto. Se emplean de izquierda a derecha como las reglas de simplificación.

Por ejemplo, vamos a construir un conjunto de personas con sus nombres y edades. Para ello definimos inicialmente las clases `Persona` y `CjtoPersonas` con la palabra reservada `sorts`, indicando con `subsort` que el tipo `Persona` es un subtipo de `CjtoPersonas`:

```
sorts Persona CjtoPersonas .
subsort Persona < CjtoPersonas .
```

Después definimos el constructor de personas con la instrucción `op` con los dos argumentos, donde el nombre será del tipo `Qid` y la edad será un número natural:

```
op {_,_} : Qid Nat -> Persona [ctor] .  *** Nombre y edad
```

Los conjuntos de personas se definen ahora como una sopa de elementos o, dicho de otra forma, como una lista asociativa, conmutativa, idempotente y cuyo elemento neutro es la lista vacía:

```
op cjtoVacio : -> CjtoPersonas [ctor] .
op __ : CjtoPersonas CjtoPersonas -> CjtoPersonas [ctor assoc comm idem    id:
    cjtoVacio] .
```

De esta forma, ya podemos escribir y simplificar términos del tipo `Persona` y del tipo `CjtoPersonas` usando el comando `red`:

```
Maude> red {'Manuel, 10} .
reduce in CJTOPERSONAS : {'Manuel,10} .
result Persona: {'Manuel,10}

Maude> red {'Manuel, 10} {'Juan, 27} .
reduce in CJTOPERSONAS : {'Juan,27} {'Manuel,10} .
result CjtoPersonas: {'Juan,27} {'Manuel,10}
```

Podemos calcular el número de personas en un conjunto con el operador `contarPersonas` y sus ecuaciones:

```
op contarPersonas : CjtoPersonas -> Nat .
eq contarPersonas(cjtoVacio) = 0 .
eq contarPersonas(P:Persona CP:CjtoPersonas)
   = s(contarPersonas(CP:CjtoPersonas)) .
```

Observamos que el operador se define recursivamente. La función aplicada sobre el conjunto vacío devuelve el valor cero y cuando se aplica sobre un conjunto de personas de la forma `P:Persona CP:CjtoPersonas`, el operador `contarPersonas` devuelve el resultado de contar el número de elementos del conjunto definido con la variable `CP:CjtoPersonas` más uno (`s_` denota la función sucesor en los números naturales).

Por supuesto, las ecuaciones de nuestro programa deben ser "buenas" *reglas de simplificación*, en el sentido de que su resultado final debe existir y ser único. Este es de hecho el caso de la definición funcional anterior.

En Maude, las ecuaciones pueden ser condicionales; es decir, que solo se aplicarán cuando se cumpla la condición especificada. Por ejemplo, podemos determinar si una persona es mayor de edad usando una ecuación condicional:

```
op mayorEdad : Persona -> Bool .
ceq mayorEdad({Q:Qid, N:Nat}) = true if N:Nat >= 18 .
eq mayorEdad(P:Persona) = false [owise] .
```

donde `ceq` es la palabra clave de Maude que introduce ecuaciones condicionales. Observamos que en la segunda ecuación utilizamos el atributo `owise` para indicar el valor a tomar cuando no se cumpla la condición de la ecuación anterior.

Un *módulo de sistema* es un programa de Maude que tiene reglas y además puede tener ecuaciones. Como las ecuaciones, las reglas también se calculan reescribiendo de izquierda a derecha. Se entienden las reglas como transiciones locales en un sistema concurrente. Las reglas de reescritura se introducen con la palabra clave `rl` y las reglas condicionales con `crl`.

Continuando con el ejemplo anterior, definimos el operador `eliminar` para eliminar una persona de una lista de personas y definimos una regla de reescritura `eliminarPersona` que nos permita realizar esta tarea:

```
op eliminar : Qid CjtoPersonas -> CjtoPersonas .
rl [eliminarPersona] :
   eliminar(Q:Qid, {Q:Qid, N:Nat} CP:CjtoPersonas)
=> CP:CjtoPersonas .
```

Esta regla toma ventaja de que un conjunto de personas es una sopa asociativa y conmutativa. Así que cuando queremos eliminar la persona con nombre `Q:Qid` de un conjunto de personas que tiene una persona de la forma `{Q:Qid, N:Nat}`, es decir, que su nombre es `Q:Qid`, entonces el conjunto reescrito será el resto del conjunto de personas, es decir, `CP:CjtoPersonas`.

A continuación definimos la unión de conjuntos de personas a través operador `anadir`, pero queremos que la unión de los conjuntos se realice con reglas de reescritura, no de forma ecuacional. Utilizamos el operador auxiliar `noPerteneceConjunto` para determinar si un nombre de persona está en un conjunto de personas. Devolverá un valor verdadero si esa persona no pertenece al conjunto.

```
op noPerteneceConjunto : Qid CjtoPersonas -> Bool .
eq noPerteneceConjunto(Q:Qid, {Q:Qid, N:Nat} CP:CjtoPersonas) = false .
eq noPerteneceConjunto(Q:Qid, CP:CjtoPersonas) = true [owise] .
```

En la primera ecuación se dice que cuando un nombre (`Qid`, primer argumento del operador `noPerteneceConjunto`) es el nombre de la persona que tenemos en el segundo argumento, entonces el resultado de la función es falso. Con el atributo `owise` de la segunda ecuación, indicamos que en cualquier otro caso de no poder aplicar la primera ecuación, el resultado será verdadero.

Definimos ahora varias reglas de reescritura para distinguir los siguientes casos: (1) cuando el primer conjunto es vacío, en este caso el resultado de la reescritura será el segundo conjunto; (2) cuando los dos conjuntos tienen un elemento con nombre común `Q:Qid`, entonces se elimina esa persona del primer conjunto y se actualiza la edad para esa persona en el segundo conjunto; y (3) cuando el primer conjunto tiene una persona que no está en

el segundo conjunto, se elimina esa persona del primer conjunto y se incluye en el segundo:

```
op anadir : CjtoPersonas CjtoPersonas -> CjtoPersonas .

rl [anadirPersonas] :
   anadir(cjtoVacio, CP:CjtoPersonas)
=> CP:CjtoPersonas .

rl [anadirPersonas] :
   anadir({Q:Qid, N:Nat} CP:CjtoPersonas, {Q:Qid, N1:Nat} CP1:CjtoPersonas)
=> anadir(CP:CjtoPersonas, {Q:Qid, N:Nat} CP1:CjtoPersonas) .

crl [anadirPersonas] :
   anadir({Q:Qid, N:Nat} CP:CjtoPersonas, CP1:CjtoPersonas)
=> anadir(CP:CjtoPersonas, {Q:Qid, N:Nat} CP1:CjtoPersonas)
if noPerteneceLista(Q:Qid, CP1:CjtoPersonas) .
```

De esta forma, en cada paso de reescritura se va reduciendo el número de elementos del primer conjunto, hasta que queda vacío, se aplica la primera regla y se acaba el proceso. Observamos que la última regla es condicional, con lo que se ejecutará cuando haya *matching* con los términos de la parte inicial de la regla y además se cumpla la condición.

Finalmente escribimos ejemplos de reescrituras, ejecutados por el comando `rew`. En el primer ejemplo se elimina la persona que se llama `'Manuel` del conjunto de personas y en el segundo ejemplo se añade el conjunto `{'Manuel, 9} {'Juan, 27}` al conjunto `{'Jose, 15} {'Luis, 4}`:

```
Maude> rew eliminar('Manuel, {'Manuel, 9} {'Juan, 27} {'Jose, 15} {'Luis, 4}) .
result CjtoPersonas: {'Jose,15} {'Juan,27} {'Luis,4}

Maude> rew anadir({'Manuel, 9} {'Juan, 27}, {'Jose, 15} {'Luis, 4}) .
result CjtoPersonas: {'Jose,15} {'Juan,27} {'Manuel,9} {'Luis,4}
```

Maude utiliza esta información para generar un algoritmo de encaje de multiconjuntos, donde el operador de unión de multiconjuntos se empareja módulo asociatividad y conmutatividad (Clavel et al., 2020). Por tanto, un programa que contiene estas reglas de reescritura es intuitivamente muy simple y tiene una semántica de reescritura sencilla.

Los sistemas que se especifican con estas reglas pueden ser concurrentes y no deterministas; es decir, a diferencia de las ecuaciones, no se supone que todas las secuencias de reescritura puedan conducir al mismo resultado. De hecho, la regla `eliminar` es no determinista: si hay varias personas con el mismo nombre no sabemos a quién se elimina. Además, algunos sistemas pueden no tener estados finales. En estos sistemas, su objetivo podría ser el tener interacciones continuas con su entorno como sistemas reactivos.

2.2. Reescritura con objetos externos

Los objetos externos representan entidades externas a Maude. Son capaces de comunicar Maude con otros elementos tales como ficheros o *sockets*.

La reescritura con objetos externos debe iniciarse con el comando de reescritura externa `erewrite` o `erew`. Es un comando similar al comando de reescritura `rew`, pero nos permite intercambiar mensajes con objetos externos que no están presentes en la configuración (Clavel et al., 2020).

El módulo predefinido STD-STREAM de Maude representa los canales de entrada/salida de UNIX (entrada estándar stdin, salida estándar stdout y error estándar stderr). Maude los trata como objetos externos.

El módulo CONFIGURATION permite definir configuraciones. Una configuración es una "sopa" de objetos y mensajes que representa un posible estado del sistema. Para que un programa se comunique con objetos externos, se precisa definir un portal:

```
sort Portal .
subsort Portal < Configuración .
op <> : -> Portal [ctor] .
```

Maude gestiona los flujos estándar de entrada/salida en modo texto y orientado a líneas. Los comandos fundamentales son parejas de mensajes que solicitan un servicio y su correspondiente respuesta. En particular tenemos:

- Los objetos de la entrada estándar se pueden obtener con el mensaje getLine(). Cuando el mensaje ya ha sido procesado, la entrada estandar stdin devuelve un mensaje del tipo gotLine().
- La salida estándar y el error estándar aceptan los mensajes write(). Cuando el mensaje ya ha sido procesado, se devuelve un mensaje del tipo wrote().

El programa debe utilizar los mensajes de las configuraciones para realizar las tareas necesarias y para permitir interacciones con objetos externos. Los objetos externos corresponden a entidades con un determinado estado.

2.3. Metaintérprete

Conceptualmente, un metaintérprete es un objeto externo que funciona como un intérprete independiente de Maude, con su propia base de datos de módulos y vistas, que envía y recibe mensajes. El módulo META-INTERPRETER del fichero meta-interpreter.maude contiene los comandos y mensajes de respuesta que cubren casi la totalidad del intérprete Maude (Clavel et al., 2020).

En nuestro programa se utiliza un metaintérprete para insertar módulos y vistas, que nos permite realizar cálculos en esos módulos. Como respuesta, el metaintérprete responde con mensajes de reconocimiento de las operaciones realizadas o que contienen resultados.

La sintaxis del metaintérprete es la misma que la usada para metarrepresentar de términos y módulos en el módulo META-LEVEL. La API para los metaintérpretes definida en el módulo META-INTERPRETER incluye varios tipos y constructores, un identificador de objeto integrado interpreterManager y una gran colección de comandos y mensajes de respuesta.

En el módulo META-INTERPRETER todos los mensajes siguen el formato estándar de los mensajes Maude, donde los dos primeros argumentos son los identificadores de objeto del objetivo y del remitente. El identificador de objeto interpreterManager indicado arriba se refiere a un objeto externo especial que se encarga de crear nuevos metaintérpretes en el contexto de ejecución actual. Estos metaintérpretes tienen identificadores de objeto de la forma interpreter(n) para cualquier número natural n (Clavel et al., 2020).

2.4. Reflexión y META-LEVEL en Maude

Una característica importante de Maude es la reflexión. Intuitivamente, significa que los programas de Maude se pueden metarrepresentar como datos, que luego se pueden manipular y transformar como si fuesen datos "normales" (Clavel et al., 2020). El uso de módulos metarrepresentados nos permite razonar con módulos y crear nuevos comandos.

El módulo predefinido `META-LEVEL` de Maude proporciona funciones para "subir" y "bajar" de nivel y para ejecutar funciones a nivel objeto al metanivel, que explicaré con detalle más adelante. De esta forma, sería lo mismo sumar `2 + 2` que metarrepresentar el módulo `NAT` de los números naturales (con la función `upModule`), metarrepresentar `2 + 2` (con la función `upterm`), ejecutarlo (con la función `metareduce`) y bajarlo (con la función `downTerm`).

Muchas de las operaciones del módulo `META-LEVEL` son parciales. Dicho de otra forma, son funciones que pueden no tener un tipo cuando se aplican con ciertos argumentos. En estos casos se tratarán como indefinidos, no como errores. Estas indefiniciones se corresponden con excepciones o casos no contemplados en la semántica que hayamos definido.

Este módulo es totalmente funcional, ya que sus funciones de descenso son deterministas, aunque puedan manipular entidades intrínsecamente no deterministas como las teorías de reescritura (Clavel et al., 2020). Debido a su naturaleza funcional, aunque sea muy potente, significa que no tiene noción de estado.

De acuerdo con Clavel et al. (2020), a continuación se describen algunas de las operaciones más importantes del módulo `META-LEVEL` que se utilizan en el presente trabajo:

- La operación `metaParse` refleja el comando `parse` de Maude.

  ```
  op metaParse : Module QidList Type? ~> ResultPair? [special (...)] .
  ```

  Esta función analiza si la lista de tokens (`QidList`) se corresponde con el tipo (`Type?`), respecto del módulo recibido en el primer argumento. La constante `anyType` permite generar términos de cualquier tipo.

  Si `metaParse` tiene éxito, la función devuelve la metarrepresentación del término analizado con su tipo correspondiente. En caso contrario, devuelve:

  - `noParse(`n`)` cuando no hubo análisis. El valor n es el índice del primer token incorrecto (contando desde 0), o el número de tokens en el caso de un final de entrada inesperado. Es del tipo `ResultPair?`.
  - `ambiguity(`$r1$`,`$r2$`)` cuando existen varios análisis. Los valores $r1$ y $r2$ son los pares de resultados correspondientes a dos análisis distintos.

- Las operaciones `upTerm` y `downTerm` son fundamentales en el uso de la reflexión con Maude:

  - La función `upTerm` toma un término `t` y devuelve la metarrepresentación de su forma canónica.
  - La función `downTerm` toma la metarrepresentación de un término `t` como primer argumento y un término `t'` como segundo argumento. La función devuelve:

- La forma canónica de `t`, si `t` es un término del mismo tipo que `t'`.
- En caso contrario, devuelve la forma canónica de `t'`.

Esta es la sintaxis de los comandos `upTerm` y `downTerm`:

```
op upTerm : Universal -> Term [poly (1) special (...)] .
op downTerm : Term Universal -> Universal [poly (2 0) special (...)] .
```

Existen algunas otras funciones para moverse entre el metanivel y el nivel objeto. En particular, el módulo `META-LEVEL` proporciona las funciones `upModule` y `upView`. El operador `upModule` mueve el módulo indicado al metanivel. Toma como argumentos la metarrepresentación del nombre de un módulo y un valor booleano, y devuelve la metarrepresentación del módulo. Cuando el segundo argumento es `true`, entonces la metarrepresentación incluirá las declaraciones correspondientes que importa el módulo. En caso contrario, solo contendrá las declaraciones declaradas explícitamente en el módulo.

El siguiente operador `upView` toma como argumento la metarrepresentación del nombre de una vista, cuando dicha vista está en la base de datos de vistas de Maude. El operador `upView` devuelve la metarrepresentación correspondiente.

La declaración de estos operadores es:

```
op upModule : Qid Bool ~> Module [special (...)] .
op upView : Qid ~> View [special (...)] .
```

- La operación parcial `metaReduce` se encarga de reducir un término con la simplificación ecuacional. Su sintaxis es la siguiente:

```
op metaReduce : Module Term ~> ResultPair [special (...)] .
```

Toma como argumentos la metarrepresentación de un módulo `R` y la metarrepresentación de un término `t`. Cuando `t` es un término de `R`, `metaReduce(R,t)` devuelve la metarrepresentación de la forma canónica de `t`, con el uso de las ecuaciones del módulo `R`, junto con la metarrepresentación de su correspondiente clase. La estrategia de reducción que utiliza `metaReduce` es similar a la empleada por el comando `reduce`.

Cuando alguno de los argumentos que recibe no es correcto, la operación `metaReduce` es indefinida, es decir, el término no se reduce y no se evalúa a un término de tipo `ResultPair`.

- La operación parcial `metaSearch` realiza una búsqueda en la que los términos se pueden reescribir con reglas, que en nuestro programa PLSS serán del tipo *soft* reglas. La estrategia de búsqueda utilizada por `metaSearch` coincide con la del comando `search` a nivel objeto.

La operación `metaSearch` toma como argumentos la metarrepresentación de un módulo (`Module`), la metarrepresentación del término inicial de la búsqueda (`Term`), la metarrepresentación del patrón a buscar (`Term`), la metarrepresentación de una condición a satisfacer (`Condition`), la metarrepresentación del tipo de búsqueda a realizar (`Qid`), un valor (`Bound`) y un número natural (`Nat`):

```
op metaSearch : Module Term Term Condition Qid Bound Nat ~> ResultTriple?
    [special (...)] .
```

Los valores `Qid` permiten definir los tipos de búsqueda con argumentos similares a los tipos de búsqueda empleados por el comando `search`:

- `'*` para una búsqueda que implique cero o más reescrituras (`=>*`).
- `'+` para una búsqueda que consta de una o más reescrituras (`=>+`).
- `'!` para una búsqueda que solo coincide con formas canónicas (`=>!`).

La profundidad máxima de la búsqueda se representa con el argumento `Bound` y el número de la solución que se muestra se consigue con el argumento `Nat`.

- La función `metaNarrowingSearch` se encarga de realizar la búsqueda con estrechamiento (*narrowing*), en la cual se analizan todos los cómputos de un conjunto de estados gracias a la descripción de los mismos mediante técnicas simbólicas (Clavel et al., 2020; Escobar et al., 2005). La sintaxis de esta función es:

```
op metaNarrowingSearch : Module Term Term Qid Bound Qid Nat ->
    NarrowingSearchResult? [special ...] .
```

Los argumentos de entrada en `metaNarrowingSearch` son:

- La metarrepresentación del módulo (`Module`), el término inicial (primer `Term`) o estado inicial del que se parte y el término del patrón a alcanzar (segundo `Term`);
- El tipo de búsqueda (`Qid`), que en nuestro caso es `'+` (1 o más pasos). A continuación, `Bound` representa la longitud máxima de narrowing, que es `unbounded` para nuestro programa, indicando que no se impone límite en el número de reescrituras;
- El siguiente argumento `Qid` puede ser `'none` o `'match`. La constante `'none` significa que se aplica una reducción estándar sin ningún plegado.
- Por último, el argumento `Nat` se corresponde con el índice de la solución elegida. De esta forma, se pueden proporcionar todas las soluciones de forma secuencial, como hacen otros comandos de metanivel en Maude.

La función devuelve la solución del tipo `NarrowingSearchResult` cuando la búsqueda es exitosa. Si no hubo resultado, la función devuelve `failure`.

2.5. Soft sets

Muchos problemas de la vida real requieren el uso de datos imprecisos o inciertos. Su análisis necesita la aplicación de principios matemáticos capaces de captar estas características. La teoría de conjuntos difusos (*fuzzy sets*) supuso un cambio paradigmático en las matemáticas al permitir un grado de pertenencia parcial. Existe una vasta literatura sobre los conjuntos difusos y sus aplicaciones desde la publicación del artículo de Zadeh (1965). Las definiciones de los *soft sets* y de los *soft sets* incompletos y su aplicación a las

tomas de decisión se han obtenido fundamentalmente de los trabajos de Alcantud (2016a); Alcantud y Santos-García (2017); Santos-Buitrago et al. (2019).

De las generalizaciones de los *fuzzy sets* nos interesa especialmente la aplicación de la teoría de los conjuntos blandos (*soft sets*) y sus extensiones a los problemas de la toma de decisiones. Los *soft sets* fueron introducidos por Molodtsov (1999). Algunas referencias relevantes del desarrollo de su teoría se deben a: Aktaş y Çağman (2007) que definen las propiedades básicas de los *soft sets*, Alcantud (2016a,b) que los utiliza para las tomas de decisión y estudia las relaciones de los *soft sets* con los *fuzzy sets* y otras extensiones, y Ali et al. (2009) que definen nuevas operaciones con los *soft sets*. Ali et al. (2015) definen *soft sets* ordenados en red para situaciones en las que existe algún orden entre los elementos del conjunto de parámetros. Qin et al. (2013) combinan los conjuntos de intervalos y los *soft sets* y Zhang (2014) estudia los *interval soft sets* y sus aplicaciones. Maji et al. (2001) introducen los *fuzzy soft sets*. Wang et al. (2014) introducen los *hesitant fuzzy soft sets*. Han et al. (2014), Qin et al. (2011), y Zou y Xiao (2008) se ocupan de los *soft sets* incompletos. También hay interesantes modelos híbridos en la literatura reciente.

2.5.1. Soft sets incompletos

Se adopta la descripción y terminología habitual para los *soft sets* y sus extensiones: el conjunto U denota el universo de objetos y el conjunto E denota el conjunto universal de parámetros.

Definición 1 (Molodtsov (1999)) *Un par* (F, A) *es un* soft set *sobre* U *cuando* $A \subseteq E$ *y* $F : A \longrightarrow \mathcal{P}(U)$*, donde* $\mathcal{P}(U)$ *denota el conjunto de todos los subconjuntos de* U*.*

Un *soft set* sobre U es considerado como una familia de subconjuntos parametrizados del universo U, siendo el conjunto A los parámetros. Para cada parámetro $e \in A$, $F(e)$ es el subconjunto de U aproximado por e o el conjunto de elementos e-aproximados del *soft set*. Muchos investigadores han desarrollado esta noción y definen otros conceptos relacionados (Maji et al., 2003; Feng y Li, 2013). Para modelar situaciones cada vez más generales, se ha propuesto la siguiente definición de *soft sets* incompletos.

Definición 2 (Han et al. (2014)) *Un par* (F, A) *es un soft set incompleto sobre* U *cuando* $A \subseteq E$ *y* $F : A \longrightarrow \{0, 1, *\}^U$*, donde* $\{0, 1, *\}^U$ *es el conjunto de todas las funciones de* U *a* $\{0, 1, *\}$.

Obviamente, todo *soft set* puede considerarse un *soft set* incompleto. El símbolo $*$ en la definición 2 permite capturar la *falta de información*: cuando $F(e)(u) = *$ interpretamos que se desconoce si u pertenece al subconjunto de U aproximado por e. Como en el caso de los *soft sets*, cuando $F(e)(u) = 1$ (resp., $F(e)(u) = 0$), interpretamos que u pertenece (resp., no pertenece) al subconjunto de U aproximado por e.

Cuando los conjuntos U y A son finitos, los *soft sets* y los *soft sets* incompletos pueden representarse por matrices o en forma tabular. Las filas se corresponden con objetos en U y las columnas se corresponden con parámetros en A. En el caso de un *soft set*, estas representaciones son binarias (es decir, todos los elementos o celdas son 0 ó 1).

El siguiente ejemplo de la práctica real ilustra un *soft set* incompleto. Después lo utilizamos para explicar los fundamentos de toma de decisiones en términos prácticos.

Ejemplo 1 *Sea $U = \{h_1, h_2, h_3\}$ el universo de casas y $E_0 = \{e_1, e_2, e_3, e_4\}$ el conjunto de parámetros (atributos o características de la casa). La siguiente información define un* soft set *incompleto (F_0, E_0):*

(a) $h_1 \in F_0(e_1) \cap F_0(e_3)$ y $h_1 \notin F_0(e_4)$, pero se desconoce si $h_1 \in F_0(e_2)$ o no.

(b) $h_2 \in F_0(e_2)$ y $h_2 \notin F_0(e_3) \cup F_0(e_4)$, pero se desconoce si $h_2 \in F_0(e_1)$ o no.

(c) $h_3 \in F_0(e_1) \cap F_0(e_4)$ y $h_3 \notin F_0(e_2) \cup F_0(e_3)$.

(d) $h_4 \notin F_0(e_1) \cup F_0(e_2) \cup F_0(e_4)$, pero se desconoce si $h_4 \in F_0(e_3)$ o no.

La tabla 2.1 captura la información que define (F_0, E_0). De esta forma, se obtiene la representación tabular del *soft set* incompleto (F_0, E_0).

	e_1	e_2	e_3	e_4
h_1	1	$*$	1	0
h_2	$*$	1	0	0
h_3	1	0	0	1
h_4	0	0	$*$	0

Tabla 2.1: Representación tabular del *soft set* incompleto (F_0, E_0) definido en el ejemplo 1.

2.5.2. Toma de decisiones con *soft sets*

Maji et al. (2002) fueron pioneros en la toma de decisiones basadas en *soft sets*. Establecieron el criterio de que un objeto puede ser seleccionado si maximiza el valor de elección del problema. En relación con esto, Zou y Xiao (2008) argumentaron que en el proceso de recopilación de datos puede haber datos desconocidos, imprecisos o inexistentes. Por lo tanto, se deben tener en cuenta los *soft sets* estándar bajo información incompleta, lo que exige la inspección de *soft sets* incompletos.

Cuando un *soft set* (F, A) se representa en forma matricial a través de la matriz $(t_{ij})_{k\times l}$, donde k y l son los cardinales de U y A respectivamente, entonces el valor de elección (o *choice value*) de un objeto $h_i \in U$ es $c_i = \sum_j t_{ij}$. Se hace una elección adecuada cuando el objeto seleccionado h_k verifica $c_k = \text{máx}_i\, c_i$. En otras palabras, los objetos que maximizan el valor de elección son los resultados satisfactorios de este problema de decisión.

En lo que respecta a la toma de decisiones incompleta basada en *soft sets*, los enfoques más utilizados son los de Zou y Xiao (2008), Qin et al. (2011), Han et al. (2014) y Alcantud y Santos-García (2017). Estas son las ideas de sus métodos:

(a) Zou y Xiao (2008) iniciaron el análisis de *soft sets* y *fuzzy soft sets* bajo información incompleta. En el primer caso, proponen calcular todos los valores de elección posibles para cada objeto, y luego calcular sus respectivos valores de decisión d_i por el método del promedio ponderado. Para ello, el peso de cada valor de elección posible se calcula con la información completa existente. En particular, proponen algunos indicadores sencillos que pueden utilizarse eventualmente para priorizar las alternativas, a saber, $c_{i(0)}$ (el valor de elección si se supone que todos los datos que faltan son 0), $c_{i(1)}$ (el valor de elección si se supone que todos los datos que faltan son 1) y d_{i-p} (el valor de elección corresponde a $(c_{i(0)} + c_{i(1)})/2$).

(b) Inspirándose en el enfoque de análisis de datos de Zou y Xiao, Qin et al. (2011) proponen una nueva forma de completar los datos que faltan en un *soft set* incompleto. Para ello, introducen la relación entre los parámetros. Así pues, dan prioridad a la asociación entre parámetros antes que a la probabilidad de que aparezcan objetos en $F(e_i)$. De esta manera, adjuntan un *soft set* completo con cualquier *soft set* incompleto. Sin embargo, el procedimiento de Qin et al. (2011) presupone que hay asociaciones entre algunos de los parámetros. En su propuesta, cuando no se alcanza un umbral dado exógenamente, los datos se rellenan según el enfoque de Zou y Xiao.

Qin et al. indican que su procedimiento puede utilizarse para implementar aplicaciones que impliquen *soft sets* incompletos, pero no hacen ninguna declaración explícita en cuanto a la toma de decisiones. No obstante, parece apropiado complementar su procedimiento de llenado con una priorización de los objetos de acuerdo con sus valores de elección Q_i, como es frecuente en la toma de decisiones basada en los *soft sets*.

(c) Han et al. (2014) explican que su método es bueno cuando los objetos en U están relacionados entre sí. Estos autores desarrollan y comparan varios criterios de obtención para la toma de decisiones de *soft sets* incompletos que se generan por intersección restringida.

(d) Por último, Alcantud y Santos-García (2017) proponen considerar todas las formas posibles de "completar" un *soft set* incompleto y tomar la decisión a partir de estas soluciones completadas. Este sistema es apropiado cuando no se tiene información a priori sobre la relación entre los objetos y los parámetros.

Para explicar la idea de este método, la aplicamos sobre el ejemplo 1 y obtenemos los valores de elección s_i de cada opción. Si se tienen en cuenta todas las posibilidades, en última instancia uno de las cuatro tablas representadas en la tabla 2.2 contiene la información completa que se necesita para tomar la decisión. En esta tabla hemos eliminado el objeto h_4 debido al cribado de dominación previo que se explica a continuación. Como no sabemos cuál será la correcta, debemos asumir que todas estas tablas son equiparables según el principio de indiferencia de Laplace. Por lo tanto, es sensato calcular qué objetos deben ser seleccionados de acuerdo con la toma de decisiones en cada uno de estos casos, y luego seleccionar el objeto que sea óptimo en la mayoría de los casos.

Los investigadores pueden llevar a cabo una operación de cribado antes de seleccionar una opción final. En primer lugar, calculamos el valor máximo c_0 de todos los valores de elección $c_{j(0)}$ a través de las opciones u_j. Si este valor es estrictamente mayor que el valor de elección $c_{k(1)}$ de una alternativa u_k, esta alternativa puede ser eliminada de la matriz/tabla inicial. La razón es que si se supone que todos los datos que faltan para el u_k son 1, hay otra opción i que verifica que cuando todos los datos que faltan para u_i se suponen que son 0, la opción i sigue teniendo un valor de elección mayor que la opción k. Este argumento sugiere la siguiente definición de dominancia entre opciones.

Definición 3 (Alcantud y Santos-García (2017)) *Sea* (F, A) *un* soft set *incompleto sobre* U. *Una opción* i *domina una opción* k *cuando* $c_{k(1)} < c_{i(0)}$.

Claramente, si empleamos cualquier solución basada en el valor de la elección, podemos descartar libremente las opciones dominadas. Por ejemplo, si utilizamos d_j, $c_{j(0)}$, $c_{j(1)}$ ó

	C_{v_1} matrix				
	e_1	e_2	e_3	e_4	c_i
h_1	1	0	1	0	2
h_2	0	1	0	0	1
h_3	1	0	0	1	2

	C_{v_2} matrix				
	e_1	e_2	e_3	e_4	c_i
h_1	1	0	1	0	2
h_2	1	1	0	0	2
h_3	1	0	0	1	2

	C_{v_3} matrix				
	e_1	e_2	e_3	e_4	c_i
h_1	1	1	1	0	3
h_2	0	1	0	0	1
h_3	1	0	0	1	2

	C_{v_4} matrix				
	e_1	e_2	e_3	e_4	c_i
h_1	1	1	1	0	3
h_2	1	1	0	0	2
h_3	1	0	0	1	2

Tabla 2.2: Las cuatro tablas completadas para el *soft set* incompleto (F_0, E_0) según el paso 4 de nuestro algoritmo, con los respectivos valores de elección para cada alternativa.

d_{j-p} como indicador de cualquier opción j, la opción k no puede maximizar el indicador seleccionado cuando la opción i lo domina. Esta simplificación es básicamente intrascendente en el caso de las soluciones de Zou y Xiao, sin embargo, podemos aplicarla en otros algoritmos computacionalmente costosos para reducir los cálculos.

Siguiendo con el ejemplo 1, observamos que las casas 1 y 3 dominan la casa 4, por lo que h_4 se elimina y queda una tabla de 3×4. En dicha tabla recortada tenemos $w = 2$, y enumeramos las celdas con valor $*$ como $((1, 2), (2, 1))$.

Por cada $v \in \{0, 1\}^w = \{v_1 = (0, 0), v_2 = (0, 1), v_3 = (1, 0), v_4 = (1, 1)\}$ surge una tabla factible completada. Estas cuatro tablas están representadas en la tabla 2.2, junto con los valores de elección de las casas de cada tabla. Observamos que h_1 adjunta el valor de elección más alto en todas estas cuatro tablas, h_2 adjunta el valor de elección más alto solo en C_{v_2}, y h_3 adjunta el valor de elección más alto exactamente en C_{v_1} y C_{v_2}.

	s_i	d_i	d_{i-p}	$c_{i(0)}$	$c_{i(1)}$	Q_i
h_1	1,00	2,50	2,50	2	3	3
h_2	0,25	2,00	1,50	1	2	1
h_3	0,50	2,00	2,00	2	2	2
h_4	0	0,33	0,5	0	1	1
Óptimo	$\{h_1\}$	$\{h_1\}$	$\{h_1\}$	$\{h_1, h_3\}$	$\{h_1\}$	$\{h_1\}$

Tabla 2.3: Soluciones para el problema representado por (F_0, E_0) en ejemplo 1 según varios indicadores.

La tabla 2.3 contiene los indicadores de las propuestas de solución que hemos mencionado aplicados al ejemplo 1. También se representan las alternativas óptimas para cada procedimiento. Además, obsérvese que el hecho de que las casas 1 y 3 dominen la casa 4 se deriva de la tabla 2.3, al comparar el máximo de la columna $c_{i(0)}$—que se alcanza en 1 y 3—y los valores de la columna $c_{i(1)}$ que son estrictamente menores que dicho máximo.

2.6. Sistemas biológicos simbólicos y Pathway Logic

Como hemos visto, Maude proporciona numerosas herramientas de análisis para teorías de reescritura: ejecución de reglas de reescrituras, búsqueda en anchura, comprobación de modelos (*model checking*) en lógica temporal lineal, demostración inductiva de teoremas y muchos otros. Con la utilización de estas funcionalidades es posible estudiar el comportamiento de los sistema biológicos para comprobar, por ejemplo, si es posible alcanzar un cierto estado desde el estado inicial y analizar si el sistema verifica algunas propiedades temporales.

La idea de transición entre estados permite modelar los sistemas biológicos en Maude de manera natural: mientras que las células son un conjunto de multiconjuntos que representan las diferentes componentes que aparecen en una célula real, las reacciones bioquímicas se representan por medio de reglas de reescritura (Bernardo et al., 2008).

Pathway Logic es un marco de análisis cualitativo implementado en Maude (Talcott, 2008, 2016). Pathway Logic se utiliza para modelar y analizar procesos biológicos, como las redes de señalización de células o redes metabólicas (es decir, el conjunto de procesos físicos y del metabolismo que deciden las características fisiológicas y bioquímicas en una célula). Las redes de señalización se inician o "disparan" cuando una molécula de señalización externa a la célula activa un receptor en la membrana celular y esta unión ligando/receptor desencadena alteraciones en cadena en la moléculas dentro de la célula.

Los modelos de Pathway Logic pueden analizarse directamente mediante ejecución, búsqueda y verificación de modelos (Talcott y Dill, 2006; Talcott et al., 2003; Talcott y Knapp, 2017). Actualmente las capacidades de Pathway Logic incluyen:

1. Modelos con diferentes niveles de detalle. Las moléculas biológicas, sus estados, las partes o localizaciones de la célula y sus roles en los procesos moleculares o celulares pueden ser modelados con diferentes niveles de abstracción.

2. Rutas generadas dinámicamente con el uso de búsquedas y verificación de modelos. Dada una especificación de un sistema concurrente, esta se puede: ejecutar para encontrar un comportamiento posible; utilizar la búsqueda para comprobar si se puede alcanzar un estado que cumpla una condición determinada; o la verificación del modelo para ver si se satisface una propiedad temporal; y, cuando no se cumple la propiedad, se puede mostrar el contraejemplo (Knapp et al., 2005).

3. Transformación a redes de Petri para análisis y visualización. Pathway Logic Assistant es una aplicación implementada en Java que proporciona una herramienta gráfica para Pathway Logic. Pathway Logic Assistant proporciona una representación visual interactiva de los modelos de Pathway Logic y, entre otras, facilita las siguientes tareas: muestra la red de reacciones de señalización para una determinada placa de preparación (o *dish*), formula y envía consultas para encontrar y comparar rutas, etc. (Talcott y Dill, 2005). Dado un estado inicial, Pathway Logic Assistant selecciona las reglas correspondientes del conjunto de reglas y representa la red de reacciones resultante como una red de Petri. De esta forma, se consigue una representación gráfica natural y se consiguen algoritmos eficientes para responder a las consultas.

2.6.1. Modelos en Pathway Logic: ecuaciones

Para ilustrar cómo Pathway Logic puede tratar las rutas de señalización, se muestra a continuación un modelo abreviado de transducción de señales intracelulares. Una base de conocimiento formal contendrá la información sobre los cambios que ocurren en las proteínas en una célula en respuesta a sustancias químicas y otros elementos. También contienen los cambios de la exposición a ligandos/receptores, siendo los principales responsables en el inicio del proceso de transducción de señales. Los ligandos son moléculas que envían una señal al unirse al centro activo de una proteína, en cambio, los receptores son proteínas, capaces de identificar el mensajero químico.

TGFB1 (Factor de crecimiento transformante beta 1) es uno de los modelos implementados en Pathway Logic. El modelo TGFB1 contiene un total de 57 reglas y 968 datums. La evidencia experimental de cada regla se suministra en forma de datums, que representan el resultado de un experimento publicado en una revista especializada (Talcott, 2008). Las reglas y las evidencias forman parte del modelo de estímulos que se puede descargar del sitio web de Pathway Logic (`http://pl.csl.sri.com`).

La figura 2.1 muestra la complejidad de las reacciones que tienen lugar en la ruta de señalización TGF-β.

Los modelos de Pathway Logic se organizan en cuatro capas: clases y operaciones, componentes, reglas y consultas. En primer lugar, la capa de clases y operaciones declara

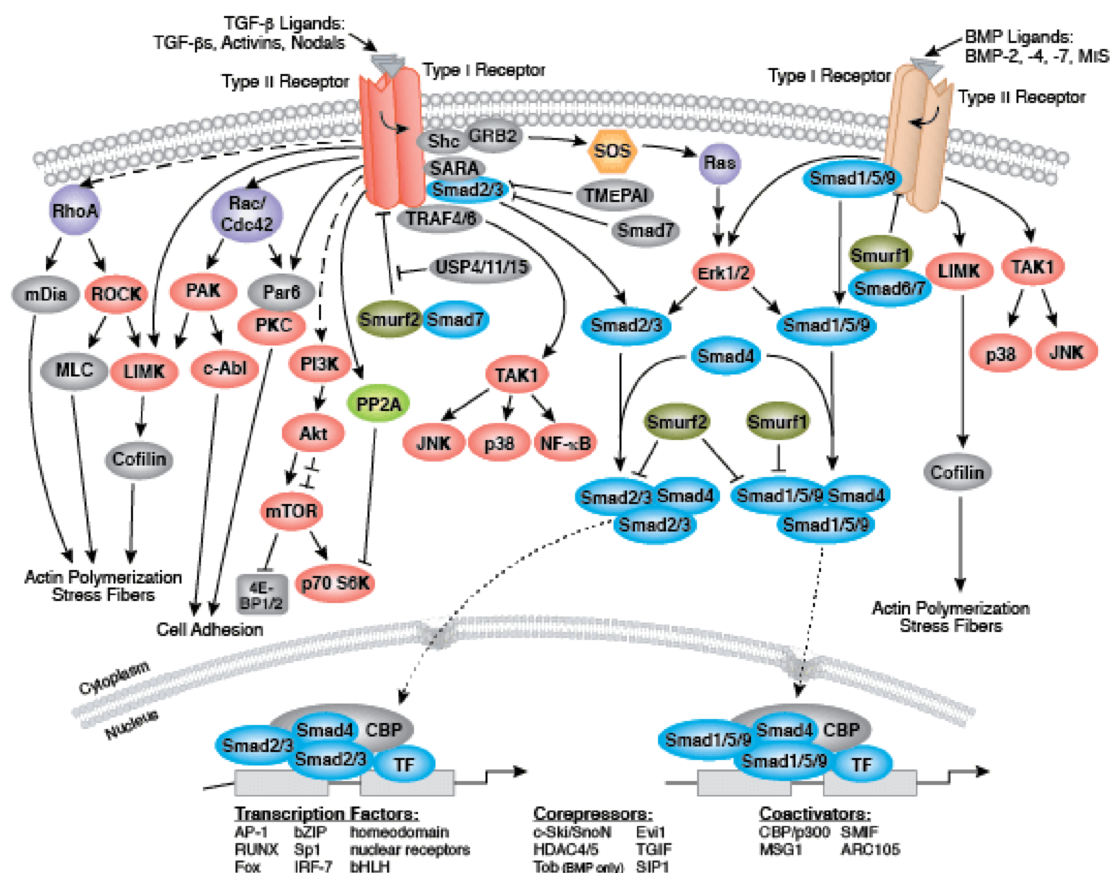

Figura 2.1: Ruta de señalización TGF-β. Fuente: Cell Signaling Technology, `https://www.cellsignal.com/pathways/tgf-beta-signaling-pathway`.

las principales relaciones de clases y subordinación, constituye el análogo lógico de la ontología. De esta forma, analizaremos de arriba a abajo los aspectos involucrados en un modelo de Pathway Logic. Las placas de preparación o *dishes* se definen como envoltorios de multiconjuntos o términos de la clase "sopa" (`Soup`). Los elementos de una sopa son las diferentes partes o localizaciones de la célula con sus respectivos contenidos. En Maude, el constructor `PD` para *dishes* y el constructor de localizaciones se definen como:

```
op PD : Soup -> Dish [ctor] .
op {_|_} : LocName Soup -> Location [ctor format (n d d d d d)] .
```

Cada una de las localizaciones se construye con el operador `{_|_}`, donde el primer argumento es el identificador o nombre de la localización y el segundo argumento es la sopa (`Soup`) de sus componentes.

De esta manera, el operador de `PD` se aplica a un multiconjunto de partes de la célula y se obtiene un elemento de la clase `Dish`. Por ejemplo, el siguiente código en Maude define una placa con dos localizaciones: el núcleo (`NUc`) que contiene las proteínas `Rb1`, `Myc`, y `Tp53`; y la membrana celular (`CLm`) que contiene las proteínas `EgfR` y `PIP2`:

```
PD( {NUc | Rb1 Myc Tp53}  {CLm | EgfR PIP2} ) .
```

En un segundo nivel, se define la sopa de localizaciones:

Listado 2.1: Definición en Maude de la sopa de localizaciones

```
sorts MtSoup Soup .
subsort MtSoup < Soup .
op empty : -> MtSoup [ctor] .
op __ : Soup Soup -> Soup [ctor assoc comm id: empty] .
```

Es decir, se definen las clases de `Dish` y `MtSoup` y además `MtSoup` es una subclase de la clase `Soup`. La sopa vacía se define con el operador constante `empty`. Por último, el operador `__` define un multiconjunto de entidades no ordenadas (en términos matemáticos, una estructura de datos asociativa y conmutativa de elementos cuyo elemento neutro es la lista vacía `empty`). Un ejemplo de un término de la clase `Soup` es:

```
{NUc | Rb1 Myc Tp53}  {CLc | Erks Erk1}  {CLm | EgfR PIP2}
```

donde hay un conjunto de tres localizaciones (`NUc`, `CLc` y `CLm`) con sus respectivos contenidos. En cuanto a la definición de cada localización, se utiliza la clase `Location` para especificar los elementos en las diferentes localizaciones de la célula:

```
op {_|_} : LocName Soup -> Location [ctor] .
```

El operador `{_|_}` tiene dos argumentos: el identificador de la localización y su contenido (es decir, una sopa de elementos como proteínas, sustancias químicas y genes). Los diferentes elementos pueden estar contenidos en diferentes partes o ubicaciones de la célula: fuera de la célula (`XOut`), dentro o a través de la membrana celular (`CLm`), adheridos al interior de la membrana celular (`CLi`), en el citoplasma (`CLc`) y en el núcleo (`NUc`). Podemos indicar que el núcleo `NUc` contiene el gen `Tp53-gene` (la transcripción del gen está `on`) y las proteínas `Rb1`, `Myc`, `Tp53` y `NProteasome` (Santos-Buitrago et al., 2017):

```
{NUc | [Tp53-gene - on] Rb1 Myc Tp53 NProteasome}
```

En la definición del operador `{_|_}`, hemos visto que se recibe un término del tipo `Soup` como segundo argumento. En este caso, esta sopa es una lista del contenido de esa parte o localización de la célula. Finalmente, señalamos que cada uno de los elementos de una sopa puede tener modificaciones. Por ejemplo, el término `[Rac1 - GDP]` indica que la proteína `Rac1` se une al guanosín difosfato (`GDP`).

La clase `Modification` se utiliza para representar una modificación de la proteína post-traducción (por ejemplo, activación, unión, fosforilación). Las modificaciones en Maude se aplican utilizando el operador `[_-_]`.

```
op [_-_] : Protein ModSet -> Protein  [right id: none ] .
```

Las modificaciones son un conjunto de transformaciones individuales que pertenecen a la clase `ModSet` (Talcott, 2008). Un conjunto de modificaciones se define en Maude de forma análoga a las sopas definidas anteriormente:

Listado 2.2: Conjunto de modificaciones en Maude

```
sorts Site  Modification ModSet .
subsort Modification < ModSet .
op none : -> ModSet .
op __ : ModSet ModSet -> ModSet [assoc comm id: none] .
```

Hay numerosas modificaciones posibles: `acetyl!site` (acetilado en un sitio específico), `act` (activado), `degraded` (degradado), `dimer` (dimerizado), `GDP` (ligado al GDP), `GTP` (ligado al GTP), `K48ubiq` (ligado covalentemente a la ubiquitina polimerizada mediante enlaces K48), `K63ubiq` (ligado covalentemente a la ubiquitina polimerizada mediante enlaces K63), `p50` (un producto de separación de 50kD), `phos` (fosforilado), `phos!` (fosforilado en un sitio específico), `sumo` (sumoilado), `ubiq` (ubiquitado), `Yphos` (fosforilado en tirosina), `off` (no transcribe el ARNm) y `on` (transcribe el ARNm) (Talcott, 2016).

En Maude, cada una de estas modificaciones se define de esta manera:

Listado 2.3: Definición de las modificaciones en Maude

```
op act : -> ACT [ctor metadata "((description \"activated\") (abbrev +))"] .
op phos : -> AAMOD [ctor metadata "((term \"phosphorylated residue\")
    (abbrev p))"] .
op ubiq : -> AAMOD [ctor metadata "((term \"ubiquitinylated on lysine\")
    (abbrev ub))"] .
op GDP : -> SMBIND [ctor metadata "((term \"Guanosine 5'-diphosphate\")
    (abbrev GDP))"] .
op GTP : -> SMBIND [ctor metadata "((term \"Guanosine 5'-triphosphate\")
    (abbrev GTP))"] .
```

donde `ACT`, `AAMOD`, y `SMBIND` son subclases de la clase `Modification`. En las opciones del operador, una palabra clave `metadata` permite incluir meta-información adicional sobre un modificador.

La figura 2.2 muestra una representación esquemática de una célula muy simple (Talcott, 2008). Diferentes elementos aparecen en diferentes partes o localizaciones de la célula: fuera de la célula (`XOut`), dentro o a través de la membrana celular (`CLm`), adheridos al in-

terior de la membrana celular (CLi), en el citoplasma (CLc) y en el núcleo (NUc). Se representan algunas proteínas: el factor de crecimiento epidérmico (Egf), la cinasa PI3 (Pi3k), la cinasa activadora ERK 1 (Mek1), etc. Algunas componentes aparecen con distintos modificadores: activación (act), fosforilación sobre la tirosina (Yphos), y unión al GDP (GDP). Esta célula se representada en Maude con el siguiente SmallDish:

Listado 2.4: Placa de preparación SmallDish

```
eq SmallDish =
  PD( {XOut | Egf}  {CLi | Pi3k [Cdc42 - GDP]}
    {NUc | Rb1 Myc Tp53}
    {CLc | [Mek1 - act] [Ilk - act] Erks Erk1}
    {CLm | EgfR PIP2 [Gab1 - Yphos]}) .
```

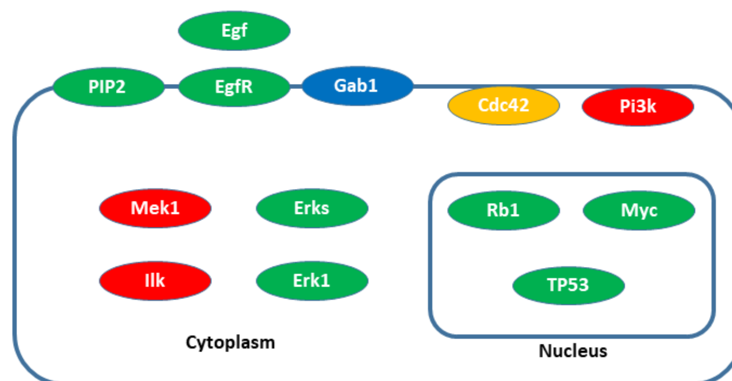

Figura 2.2: Representación esquemática de una célula (Santos-Buitrago et al., 2017). Las proteínas activadas están marcadas en rojo, las fosforiladas en azul y las unidas a GDP en amarillo. Las proteínas que no tienen modificaciones se muestran en verde.

Con la ayuda de Pathway Logic Assistant, en la figura 2.3 se muestra una representación de una red de Petri con la ruta de señalización. Los rectángulos se corresponden con las transiciones (reacciones bioquímicas) y los óvalos son las ocurrencias (entidades biológicas), que aparecen más oscurecidas cuando son las ocurrencias iniciales. Los reactivos de una regla se conectan con flechas hacia las reglas y los productos se conectan con flechas que llegan desde las reglas. Cuando tenemos flechas punteadas, ese elemento será a la vez reactivo y producto, es decir, de entrada y salida. En el ejemplo de la figura, tenemos la proteína Jak1 en el citoplasma y la proteína transmembrana Gp130 en la localización GP130C. Las dos proteínas son reactivos en la reacción 1229c. El resultado de esta regla es que la proteína Gp130 no se altera y Jak1 cambia de localización, desde el citoplasma a GP130C).

Por último, detallamos la codificación completa de un ejemplo específico de la placa de preparación Tgfb1Dish (Santos-Buitrago et al., 2017):

Listado 2.5: Dish Tgfb1Dish

```
  op Tgfb1Dish : -> Dish .
  eq Tgfb1Dish = PD( {XOut | Tgfb1} {Tgfb1RC | TgfbR1 TgfbR2} {CLo | empty}
    {CLm | empty} {CLi | [Cdc42 - GDP] [Hras - GDP] [Rac1 - GDP] }
    {CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3
        Smad4 Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16}
    {NUc | Ctdsp1 Ets1 Smad7 Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene
        Col3a1-gene  Ctgf-gene Fn1-gene Mmp2-gene Pai1-gene Smad6-gene Smad7-
```

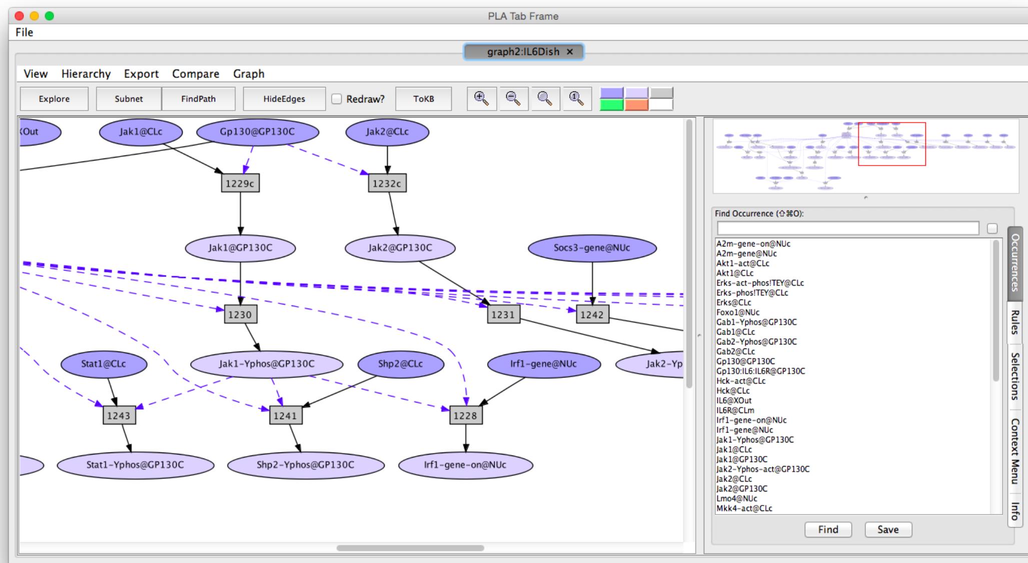

Figura 2.3: Vista general de una ruta de señalización con Pathway Logic Assistant.

```
gene Tgfb1-gene Timp1-gene Cst6-gene Dst-gene Mmp9-gene Mylk-gene Pthlh
-gene Gfi1-gene Csrp2-gene RoRc-gene}) .
```

En este *dish* se define un estado inicial (llamado `Tgfb1Dish`) con varias localizaciones y elementos:

- el exterior (localización `XOut`) que contiene el factor de crecimiento transformante beta1 (`Tgfb1`);
- la localización `Tgfb1RC` que contiene el receptor beta del factor de crecimiento transformante I y II (`TgfbR1` y `TgfbR2`);
- la localización `CLo`, que contiene los elementos adheridos al exterior de la membrana de plasma, está vacía;
- la membrana (localización con la etiqueta `CLm`) también está vacía;
- el interior de la membrana (localización con la etiqueta `CLi`) contiene tres proteínas unidas a GDP: `Cdc42`, `Hras` y `Rac1`;
- el citoplasma (localización con la etiqueta `CLc`) contiene las proteínas `Abl1`, `Akt1`, `Atf2`, `Erks`, etc.; y
- el núcleo (localización con la etiqueta `NUc`) contiene varios genes (`Smad7`, `Tgfb1`, `Cst6`, etc.) y proteínas (`Ctdsp1`, `Ets1`, etc.).

2.6.2. Modelos en Pathway Logic: reglas de reescritura

Las reglas de reescritura describen el comportamiento de las proteínas y otros componentes dependiendo de los estados de modificación y los contextos biológicos. Cada regla

representa un paso en un proceso biológico como las reacciones metabólicas o las reacciones de señalización intracelular o intercelular (Eker et al., 2002, 2003; Santos-Buitrago et al., 2017; Talcott, 2006).

El conjunto de reglas de transición se construye a partir de los hallazgos experimentales publicados en revistas prestigiosas. En el siguiente ejemplo, los resultados de Nakao et al. (1997) determinan el comportamiento de las señales del TGF-β desde la membrana al núcleo a través de los receptores.

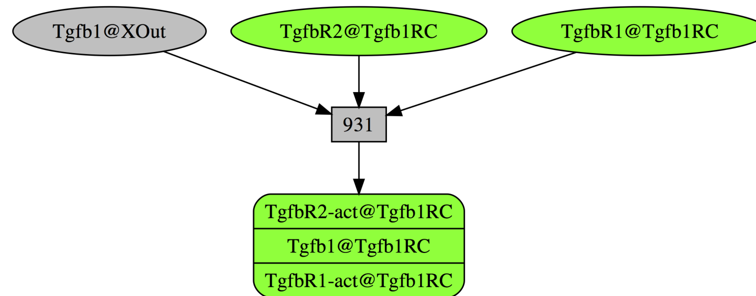

Figura 2.4: Representación esquemática de la regla de reescritura `931.TgfbR1.TgfbR2.by.Tgfb1` en Pathway Logic Assistant (Santos-Buitrago et al., 2017).

Esta regla de reescritura 931 establece: *en presencia del receptor I del factor de crecimiento transformante beta `Tgfb1` en el exterior de la célula (`XOut`), los receptores `TgfbR1` y `TgfbR2` se activan (`TgfbR1-act` y `TgfbR2-act`) y se unen entre sí y a `Tgfb1` (`[TgfbR1 - act] : [TgfbR2 - act] : Tgfb1`).* En la sintaxis de Maude, este proceso de señalización se expresa mediante la siguiente regla de reescritura (Santos-Buitrago et al., 2017):

Listado 2.6: Regla de reescritura `931.TgfbR1.TgfbR2.by.Tgfb1`

```
rl[931.TgfbR1.TgfbR2.by.Tgfb1]:
   {XOut | xout Tgfb1 }
   {Tgfb1RC | tgfb1rc TgfbR1 TgfbR2 }
=> {XOut | xout }
   {Tgfb1RC | tgfb1rc
   ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) } .
```

Capítulo 3

Descripción del trabajo

En este capítulo se describe el trabajo realizado en el proyecto **PLSS**: **P**athway **L**ogic con **S**oft **S**ets. Como se ha descrito en la sección 2.6, Pathway Logic es una herramienta diseñada para tratar con sistemas biológicos simbólicos desarrollada en SRI International (Talcott, 2008). Pathway Logic dispone de varios modelos con diferentes niveles de detalle y permite la generación dinámica de rutas de señalización mediante búsquedas y verificación de modelos.

Se han desarrollado numerosos modelos de rutas de señalización celular con esta herramienta. El modelo STM7 es una base de conocimiento formal que contiene información sobre los cambios que ocurren en las proteínas dentro de una célula en respuesta a la exposición a ligandos/receptores, sustancias químicas o diversas tensiones. Para simplificar, STM7 se divide en 32 mapas, cada uno de los cuales representa un estímulo. Estos mapas se llaman platos o *dishes* porque describen un estado inicial que corresponde al estado de las células en un plato de cultivo al comienzo de un experimento más un estímulo. En nuestro caso, se utilizan distintos mapas que se corresponden con los siguientes modelos biológicos: (1) modelo del factor de crecimiento transformante beta 1 (Tgfb1), (2) factor de crecimiento epidérmico (Egf) y (3) factor de crecimiento de hepatocitos (Hgf), (4) interleucina-6 (IL6) y (5) factor de crecimiento nervioso (Ngf).

Este trabajo proporciona una extensión del modelo STM7, de forma que se permite trabajar con datos imprecisos o desconocidos. Realmente, este trabajo se podría emplear para extender cualquier otro sistema (distinto de Pathway Logic) desplegando la configuración y los comandos con *soft sets* que aquí se proponen.

Con una base de conocimiento de Pathway Logic, en este trabajo se pretende implementar una variante de reescritura basada en *soft sets*. El comando `search` estándar en Maude permite realizar búsquedas a través del árbol de reescrituras partiendo de un estado inicial. La implementación propuesta se realizará con ayuda del metaintérprete de Maude y proporcionará las ventajas de los *soft sets* sobre la toma de decisiones bajo información incompleta. En las secciones 2.1 y 2.3, se comentaron brevemente los aspectos fundamentales del lenguaje de reescritura Maude y su metaintérprete (Clavel et al., 2020).

De forma resumida, la implementación de este proyecto se puede dividir en los siguientes bloques:

1. Importación y adaptación de los módulos de Tgfb1 procedentes del modelo STM7 de Pathway Logic.

2. Implementación en Maude de los *soft sets* y desarrollo de nuevos operadores y funciones asociadas.

3. Definición de *softdish*, que será un plato o dish de Pathway Logic con unos atributos que permitirán establecer una elección entre las nuevas reglas de ejecución basadas en *soft sets*.

4. Elaboración de un entorno de ejecución para PLSS utilizando la gestión de la entrada/salida y el metaintérprete.

5. Implementación de los comandos de búsqueda y simplificación con ecuaciones y reglas *soft*.

6. Implementación de otros comandos: carga de ficheros de usuario con comandos, carga de distintas estrategias para los comandos de simplificación, carga de los distintos modelos biológicos de Pathway Logic y sistema de ayuda del programa.

7. Integración de todas las partes en el proyecto PLSS.

En las secciones que vienen a continuación se desarrollan cada uno de los bloques enumerados[1]. Después se muestra el procedimiento de instalación y la ejecución del programa con algunos ejemplos.

3.1. Soft sets y funciones de elección

En la sección 2.5 se han descrito los fundamentos teóricos de los *soft sets*. Ahora se comenta la implementación en Maude de los *soft sets* y el desarrollo de los operadores y funciones asociados.

Un *soft set* se puede representar como una matriz cuyos elementos son ceros, unos y asteriscos. En nuestro caso, las filas corresponden con todos los posibles términos en los que se puede reescribir nuestro término inicial y las columnas a cada uno de los atributos que se consideran. Por tanto, desde el punto de vista de la codificación, la implementación de los *soft sets* consiste en definir matrices. En el módulo `MATRIX` se definen la fila de una matriz como una listas de valores y la matriz como una lista de filas:

```
fmod MATRIX is pr VALUE .
 sorts Row SoftSet .  subsort Value < Row < SoftSet .

 op mtRow : -> Row [ctor] .
 op _,_ : Row Row -> Row [ctor assoc id: mtRow] .

 op mt : -> SoftSet [ctor] .
 op __ : SoftSet SoftSet -> SoftSet [ctor assoc id: mt] .
endfm
```

[1] El código completo del proyecto está disponible en `https://github.com/rsantosb/TFM-PLSS`. En la presente memoria se incluyen únicamente los fragmentos de código necesarios para la explicación del diseño y desarrollo del programa.

Después se definen las funciones que calculan los valores de elección (*choice values*). Los valores de elección asignan un valor numérico (natural o racional) a cada posible término reescrito (o término alcanzable por reescritura) y el *mejor* término alcanzable será el que tenga un mayor valor. Existen varias formas de definir estos valores de elección. Según se explica en la sección 2.5, uno de los valores de elección es $c_{i(0)}$, que es el valor de elección si se supone que todos los datos que faltan son ceros. A continuación se detalla el código Maude para $c_{i(0)}$, con un fragmento del módulo `PREDEF-VALUE-FUNCTIONS` en el que se define el funcionamiento de las funciones `undefZero` y su función auxiliar `compUndefZero`:

Listado 3.1: Funciones `undefZero` y su función auxiliar `compUndefZero`

```
*** Replaces * by 0 and adds up all values
op undefZero : SoftSet -> Nat .
eq undefZero(M) = compUndefZero(M, 0, 0, 0) .

*** Current - Best value - Selected Row
op compUndefZero : SoftSet Nat Nat Nat -> Nat .
eq compUndefZero(mt, C, BV, S) = S .
ceq compUndefZero(R M, C, BV, S) =
             if N >= BV
             then compUndefZero(M, s(C), N, C)
             else compUndefZero(M, s(C), BV, S)
             fi
 if N := addUndefZero(R) .

op addUndefZero : Row -> Nat .
eq addUndefZero(mtRow) = 0 .
eq addUndefZero((0, R)) = addUndefZero(R) .
eq addUndefZero((1, R)) = s(addUndefZero(R)) .
eq addUndefZero((*, R)) = addUndefZero(R) .
```

De forma análoga se codifican las funciones `undefOne` y `undefSemi` que calculan los mejores valores de elección cuando se sustituyen los asteriscos por 1 o por 1/2.

Las dos nuevas funciones propuestas definen también el valor de elección para una fila como la suma de los valores asignados a cada elemento de la fila. Los elementos 0 y 1 suman su propio valor. Sin embargo, en el caso de los asteriscos, las nuevas funciones tomarán como referencia la distribución de los valores desconocidos en su fila o columna, respectivamente.

De esta forma, la estrategia `undefWRow` asigna el siguiente valor al grupo de elementos desconocidos de la fila i:

$$\frac{\sum_{i=0}^{N} i \cdot \binom{N}{i}}{2^N}$$

donde N es el número de asteriscos en la fila. Es decir, la función `undefWRow` asigna al grupo de asteriscos un valor que depende del número de elementos desconocidos en esa fila N. Para todas las posibles combinaciones de sustituciones de ceros y unos en el valor desconocido (es decir, 2^N), se calcula el número de casos en los que los unos suman un valor i, que es $\binom{N}{i}$. Después se multiplica por el valor i correspondiente y se suman, de forma que obtenemos el valor esperado para el grupo de elementos desconocidos de la fila de acuerdo con su distribución en la fila de la matriz.

Dicho de otra forma, el choice value en la estrategia `undefWRow` para la fila i es:

$$\text{cv}(i) = N_1 + \frac{\sum_{i=0}^{N} i \cdot \binom{N}{i}}{2^N}$$

donde N_1 es el número de unos en la fila y N es el número de asteriscos en la fila.

Por último, la nueva estrategia `undefWCol` asigna el valor a un elemento desconocido de la fila i y columna j basándose en la distribución de valores desconocidos en el atributo correspondiente (es decir, en la columna). El choice value que se asigna a cada fila será también la suma de los valores asignados a cada elemento. Este valor de cada elemento es igual a cero y uno para cada cero y uno, respectivamente. En el caso de un dato desconocido, entonces se le asigna $\frac{N_1}{N_1+N_0}$, donde N_0 y N_1 son respectivamente el número de ceros y unos en la columna, siempre que $N_1 + N_0 \neq 0$, es decir, que todos los elementos de la columna no sean desconocidos. En caso contrario, se asigna el valor $1/2$. Dicho de otra forma, en la estrategia `undefWCol` asignamos a cada asterisco la proporción de unos dentro de los valores conocidos en su columna.

3.2. Definición y reescritura de *softdish*

El elemento fundamental en PLSS es el *softdish*. Siguiendo el artículo de Santos-Buitrago et al. (2019), se define un *softdish* como un plato o dish de Pathway Logic junto con unos atributos que permitirán establecer una elección entre las distintas reglas de ejecución basadas en *soft sets*.

Para realizar la reescritura de un *softdish*, se necesita construir la matriz del *soft set* asociado buscando todos los términos alcanzables desde el término inicial con el valor de sus atributos y, después, se elige aplicar el término cuyo valor de elección sea mayor.

En la implementación he realizado dos versiones de reescritura (`rewStrat` y `narrowStrat`), que se explican con detalle en las siguientes subsecciones.

3.2.1. Operador `rewStrat`

En esta primera versión de reescritura, la implementación de la reescritura de un *softdish* se realiza con el operador `rewStrat`, que a su vez se apoya en el operador auxiliar `next`, que es el que calcula el siguiente término reescrito (ver listado 3.2).

Listado 3.2: Operadores `rewStrat` y `next`

```
op rewStrat : Module Term TermList Qid -> Term .
ceq rewStrat(M, T, ATTSN, Q) = T'
  if T' := next(M, T, ATTSN, Q) .
eq rewStrat(M, T, ATTSN, Q) = T [owise] .

op next : Module Term TermList Qid ~> Term .
ceq next(M, T, ATTSN, Q) = T'
 if TL := allReachableTerms(M, T) /\
    TL =/= empty /\
    MX := matrixFromTerms(TL, ATTSN) /\
    N := computeValue(MX, Q) /\
    T' := TL [N] .
```

Como se observa en el código, el operador `next` recibe como argumentos un módulo, un término y unos atributos. El operador `next` se apoya en las siguientes funciones:

- `allReachableTerms`: Calcula la lista de todos los términos alcanzables desde el término actual.
- `matrixFromTerms`: Construye la matriz asociada al *soft set*.

- `computeValue`: Calcula la fila (regla de reescritura) correspondiente con el mayor valor de decisión (con la función de choice value que se haya escogido).

De esta forma, si la lista de términos `TL` alcanzables que calcula `allReachableTerms` es no vacía, entonces `next` toma el término `N`-ésimo, donde `N` es la mejor opción que se encarga de escoger `computeValue`, con la función de elección `Q`, a partir de la matriz del *soft set* que define `matrixFromTerms`.

La función `allReachableTerms` se implementa en el listado 3.3 haciendo uso de las funciones `metaReduce` y `metaSearch`.

Listado 3.3: Función `allReachableTerms`

```
op allReachableTerms : Module Term -> TermList .
ceq allReachableTerms(M, T) = allReachableTerms(M, T, V, 0, empty)
 if Ty := getType(metaReduce(M, T)) /\
    V := qid("V:" + string(Ty)) .

op allReachableTerms : Module Term Variable Nat TermList -> TermList .
ceq allReachableTerms(M, T, V, N, TL) = TL
 if metaSearch(M, T, V, nil, '+, 1, N) == failure .
ceq allReachableTerms(M, T, V, N, TL) =
    allReachableTerms(M, T, V, s(N), (TL, T'))
 if {T', Ty, SB} := metaSearch(M, T, V, nil, '+, 1, N) .
```

Esta función `allReachableTerms` utiliza una función auxiliar con el mismo nombre. Básicamente, la función va añadiendo cada término alcanzable a la lista hasta que se produce un fallo en la búsqueda de un nuevo término. En ese momento, la función `allReachableTerms` devuelve la lista conseguida.

La función `matrixFromTerms` construye la matriz asociada al *soft set*. Para cada término de la lista de términos alcanzables, `matrixFromTerms` construirá una fila en la matriz. La función auxiliar `rowFromTerm` recibe cada término de la lista y los atributos, para formar la fila correspondiente en la matriz:

```
op matrixFromTerms : TermList TermList -> SoftSet .
eq matrixFromTerms(empty, ATTS) = mt .
eq matrixFromTerms((T, TL), ATTS) =
    rowFromTerm(T, ATTS) matrixFromTerms(TL, ATTS) .
```

La función `computeValue` recibe como argumento la matriz del *soft set* y el nombre de la función de decisión que se desea utilizar:

```
op computeValue : SoftSet Qid -> Nat .
eq computeValue(MX,'undefZero) = undefZero(MX) .
eq computeValue(MX,'undefOne)  = undefOne(MX) .
eq computeValue(MX,'undefSemi) = undefSemi(MX) .
eq computeValue(MX,'undefWRow) = undefWRow(MX) .
eq computeValue(MX,'undefWCol) = undefWCol(MX) .
```

3.2.2. Operador `narrowStrat`

En esta subsección se presenta la implementación del operador `narrowStrat` de un *softdish* basado en `narrowing`. Mientras la reescritura aplica encaje de patrones y luego reglas, el estrechamiento o `narrowing` utiliza unificación y reglas.

Su función principal es narrowStrat que, junto con su auxiliar nextN, se encargan de realizar la reescritura[2]. Las funciones matrixFromTerms y computeValue, definidas en la subsección anterior para el operador rewStrat, se utilizan también en el nuevo operador narrowStrat.

La nueva función nextN implementa un código similar a next, salvo que invoca la función allReachableNTerms. La función allReachableNTerms en el listado 3.4 es la versión narrowing de la función allReachableTerms y hace uso de la función metaReduce y la función de búsqueda con narrowing (metaNarrowingSearch).

Listado 3.4: Función allReachableNTerms

```
op allReachableNTerms : Module Term -> TermList .
ceq allReachableNTerms(M, T) = allReachableNTerms(M, T, V, 0, empty)
 if Ty := getType(metaReduce(M, T)) /\
    V := qid("V:" + string(Ty)) .

op allReachableNTerms : Module Term Variable Nat TermList -> TermList .
ceq allReachableNTerms(M, T, V, N, TL) = TL
 if metaNarrowingSearch(putNarrowingAtt(M), T, V, '+, unbounded, 'none, N) ==
     failure .
ceq allReachableNTerms(M, T, V, N, TL) =
    allReachableNTerms(M, T, V, s(N), (TL, T'))
 if {T', Ty, SB, VQid:Qid, VSubs2:Substitution, VQid2:Qid} :=
     metaNarrowingSearch(putNarrowingAtt(M), T, V, '+, unbounded, 'none, N) .
```

En la búsqueda mediante narrowing con el comando metaNarrowingSearch, el módulo M se ha modificado con la función putNarrowingAtt.

Para que las reglas de reescritura definidas en el fichero de reglas (e.g., Tgfb1RulesSS.maude) puedan ser localizadas por metaNarrowingSearch, será necesario que al final de cada una de ellas aparezca el atributo narrowing. La función putNarrowingAtt permite añadir el atributo narrowing *on the fly* y de forma automática en todas las reglas del módulo M.

Con la función putNarrowingAtt implementada en el listado 3.5 se extraen todas las reglas del módulo dado y se sustituyen por las misma reglas con el atributo narrowing. Se hace una comprobación de la existencia de este atributo en la regla con la función booleana hasNarrowing. Para completar la tarea, la función auxiliar getRules toma el conjunto de reglas del módulo y la función setRules cambia las reglas nuevas por las anteriores.

Listado 3.5: Función putNarrowingAtt

```
op putNarrowingAtt : Module -> Module .
ceq putNarrowingAtt(M) = setRules(M, RS')
 if RS' := putNarrowingAtt(getRules(M)) .

op putNarrowingAtt : RuleSet -> RuleSet .
eq putNarrowingAtt(none) = none .
ceq putNarrowingAtt(rl L => R [AtS] . RS) = rl L => R [AtS] . putNarrowingAtt(
    RS)
 if hasNarrowing(AtS) .
ceq putNarrowingAtt(crl L => R if C [AtS] . RS) = crl L => R if C [AtS] .
    putNarrowingAtt(RS)
 if hasNarrowing(AtS) .
ceq putNarrowingAtt(rl L => R [AtS] . RS) = rl L => R [narrowing AtS] .
    putNarrowingAtt(RS)
 if not hasNarrowing(AtS) .
ceq putNarrowingAtt(crl L => R if C [AtS] . RS) = crl L => R if C [narrowing
    AtS] . putNarrowingAtt(RS)
```

[2] Las funciones rewStrat, narrowStrat y sus funciones auxiliares se definen en el módulo SOFT-SET-FUN.

```
 if not hasNarrowing(AtS) .

op hasNarrowing : AttrSet -> Bool .
eq hasNarrowing(narrowing AtS) = true .
eq hasNarrowing(AtS) = false [owise] .

op setRules : SModule RuleSet -> SModule .
eq setRules(mod H is IL sorts SS . SSDS ODS MAS EqS RS endm, RS') =  mod H is
    IL sorts SS . SSDS ODS MAS EqS RS endm .

op getRules : SModule -> RuleSet .
eq getRules(mod H is IL sorts SS . SSDS ODS MAS EqS RS endm)  =  RS .
```

3.3. Módulos de Pathway Logic

En este apartado se aborda la importación y adaptación de los módulos de los distintos modelos biológicos definidos en la base de conocimiento formal STM7 de Pathway Logic. De acuerdo con la sección 2.6, en los módulos QQn, donde n varía de 1 a 5, se definen la sintaxis y la semántica de algunos modelos biológicos soportados. Se utilizarán los siguientes modelos biológicos: (1) Tgfb1 o factor de crecimiento transformante beta 1, (2) Egf o factor de crecimiento epidérmico, (3) Hgf o factor de crecimiento de hepatocitos, (4) IL6 o interleucina-6 y (5) Ngf o factor de crecimiento nervioso.

En concreto, en el módulo QQ1 se definen la sintaxis y la semántica del modelo Tgfb1[3]:

Listado 3.6: Módulo QQ1

```
mod QQ1 is
 inc ALLRULES .
 inc ALLOPS .
 inc SOFT-SET-FUN .
 inc TGFB1DISH .
endm
```

Este módulo importa los siguientes módulos: (1) ALLOPS que especifica los operadores que definen los elementos y componentes de la célula; (2) TGFB1DISH que establece el dish o estado inicial en este modelo; (3) ALLRULES que incluye todas las reglas de reescritura que gobiernan la dinámica de la célula; y (4) SOFT-SET-FUN que define las estrategias de decisión posibles con el *soft set* utilizado.

El módulo ALLOPS a su vez incluye los módulos en los que se definen las constantes para los químicos, genes y proteínas que pueden existir en la célula y también sus modificaciones (MODIFICATIONOPS).

El módulo ALLOPS también importa el módulo LOCATIONOPS donde se definen las localizaciones de la célula. Por ejemplo, se define la constante NUc que representa el núcleo de la célula. El último módulo que incluye es THEOPS, en el cual se establece toda la sintaxis de la célula.

En el módulo THEOPS y los módulos que se incluyen sucesivamente, se definen las clases y operadores que nos permiten modelizar la célula. La sopa se construye con el operador __, de forma que una sopa es un conjunto asociativo y conmutativo de elementos (Things) tales como genes, proteínas modificadas, compuestos, etc.

[3] Los módulos QQ2, QQ3, ... se definen de forma similar para los restantes modelos.

En el módulo `MODIFICATION` se incluye la sintaxis para las modificaciones. Se construye con el operador `[_-_]`, donde su primer argumento es el elemento a modificar (por ejemplo, una proteína) y el segundo argumento es el conjunto de modificadores que afectan a ese elemento. Estos elementos se definen también como una sopa o multiconjunto asociativo y commutativo de modificadores.

A continuación, el módulo `TGFB1DISH` define el dish o estado inicial en el modelo Tgfb1 de STM7. El listado 3.7 contiene la definición de la constante `Tgfb1Dish` de la clase `Dish` como un conjunto de localizaciones con sus contenidos (Talcott, 2008). Por ejemplo, se puede comprobar que la membrana de la célula está vacía (`{CLm | empty}`); que el citoplasma (`CLc`) contiene, entre otras, las proteínas `Abl1`, `Akt1` y `Atf2`; y que en el interior de la membrana celular (`CLi`) está presente la proteína `Hras` con la modificación `GDP` (`[Hras - GDP]`):

Listado 3.7: Módulo `TGFB1DISH`

```
mod TGFB1DISH is inc ALLOPS .

op Tgfb1Dish : -> Dish .
eq Tgfb1Dish = PD(
 {XOut | Tgfb1 } {Tgfb1RC | TgfbR1 TgfbR2  } {CLo | empty } {CLm | empty }
 {CLi | [Cdc42 - GDP] [Hras - GDP] [Rac1 - GDP] }
 {CLc | Abl1 Akt1 Atf2 Erks Fak1 Jnks Mekk1 Mlk3 P38s Pak2 Pml Smad2 Smad3
     Smad4 Smurf1 Smurf2 Tab1 Tab2 Tab3 Tak1 Traf6 Zfyve16 }
 {NUc | Ctdsp1 Ets1 Smad7 Cdc6-gene Cdkn1a-gene Cdkn2b-gene Col1a1-gene
     Col3a1-gene Ctgf-gene Fn1-gene Mmp2-gene Pai1-gene Smad6-gene Smad7-gene
     Tgfb1-gene Timp1-gene Cst6-gene Dst-gene Mmp9-gene Mylk-gene Pthlh-gene
     Gfi1-gene Csrp2-gene RoRc-gene } ) .
endm
```

El módulo `ALLRULES` incluye todas las reglas de reescritura que gobiernan la dinámica de la célula. El significado de estas reglas se explicó con detalle en la seccion 2.6. En el listado 3.8 se muestran dos de sus reglas (Talcott, 2008).

Listado 3.8: Reglas en el módulo `ALLRULES`

```
rl[931.TgfbR1.TgfbR2.by.Tgfb1]:
  {XOut    | xout Tgfb1                                  }
  {Tgfb1RC | tgfb1rc  TgfbR1 TgfbR2                      }
  =>
  {XOut    | xout                                        }
  {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) } .

rl[1719.Abl1.irt.Tgfb1]:
  {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) }
  {CLc     | clc      [Fak1 - fak1mods] Abl1             }
  =>
  {Tgfb1RC | tgfb1rc ([TgfbR1 - act] : [TgfbR2 - act] : Tgfb1) }
  {CLc     | clc      [Fak1 - fak1mods] [Abl1 - act]     } .
```

Por último, el módulo `SOFT-SET-FUN` define las estrategias de decisión posibles con el *soft set* utilizado, que se comenta en la siguiente sección.

3.4. Entorno de ejecución con IO y metaintérprete en PLSS

En esta sección se describe la programación del entorno de ejecución para PLSS utilizando los flujos de entrada/salida estándar y metaintérpretes. Con PLSS se ha desarrollado

un lenguaje propio simple con su propia gramática y una transformación de análisis de los términos en Maude. El lenguaje de PLSS incluye las siguientes características:

- Se pueden declarar clases y operaciones, y especificar ecuaciones en los términos que podemos construir con ellas.
- Se pueden incluir módulos previamente definidos y almacenar módulos en la base de datos del metaintérprete para que puedan ser utilizados después.
- Se pueden reducir los términos a su forma canónica utilizando las ecuaciones de un módulo. La sintaxis definida para PLSS permite los siguientes comandos: `red` para la simplificación ecuacional, `softrew` para la simplificación con *soft* reglas, `softnarrowsearch` para el análisis mediante narrowing (o estrechamiento) y, por último, `softnarrow` para la simplificación de la búsqueda mediante narrowing.
- Se puede seleccionar el modelo biológico de Pathway Logic con el que se desee trabajar.
- Se puede incluir la estrategia de decisión que se desee utilizar.
- Se puede solicitar información sobre los comandos con el comando `help`.
- Se puede ejecutar una serie de instrucciones almacenadas en un fichero de texto externo.
- Se puede salir del programa con los comandos `exit` y `q`.

Con PLSS, se puede almacenar la información en un objeto que puede pedir entradas al usuario a través del flujo de entrada estándar. Para poder analizar usando el metaintérprete, se necesita colocar el módulo con la sintaxis de PLSS en el metaintérprete. Una vez insertado, se analizan las entradas. Cuando el flujo estándar acoge un mensaje `getLine`, entonces responde con la cadena que ha introducido el usuario hasta que se encuentra un retorno de línea. Se pueden analizar las entradas de varias líneas, para esto necesitamos pedir nuevas líneas hasta que la entrada se complete. Además, en cualquier momento podemos obtener un error sintáctico o una ambigüedad, en cuyo caso se necesita informar del error.

Una vez que se introduce la entrada, se pueden utilizar distintos comandos. Si la entrada corresponde a un comando de simplificación, el término debe ser analizado y luego reducido por el metaintérprete.

Para simplificar el proceso, se utiliza un atributo `state` que lleva la cuenta de las distintas alternativas. Este objeto también almacena más información (e.g., el identificador del metaintérprete, el nombre del último módulo introducido y la entrada parcial introducida). Algunos de los estados más significativos son:

```
sort State .

*** Initial state, metainterpreter is created
op init : -> State [ctor] .
*** Loading from standard database
op load-std-db : -> State [ctor] .
*** Loading grammar
op load-grammar : -> State [ctor] .
*** Create user metainterpreter
op load-user-mi : -> State [ctor] .
```

```
*** Waiting input from the user
op idle : -> State [ctor] .
*** Parsing command
op parseComm : -> State [ctor] .
*** Execute command
op executeComm : -> State [ctor] .
*** Waits for a wrote msg and returns to idle with getLine
op print&idle : -> State [ctor] .
*** Waits for a wrote msg and keeps executing commands.
op print&executeComm : -> State [ctor] .
```

En la figura 3.1 se representa el diagrama de estados del entorno de ejecución, que muestra la secuencia de estados por los que pasa un objeto a lo largo de su vida.

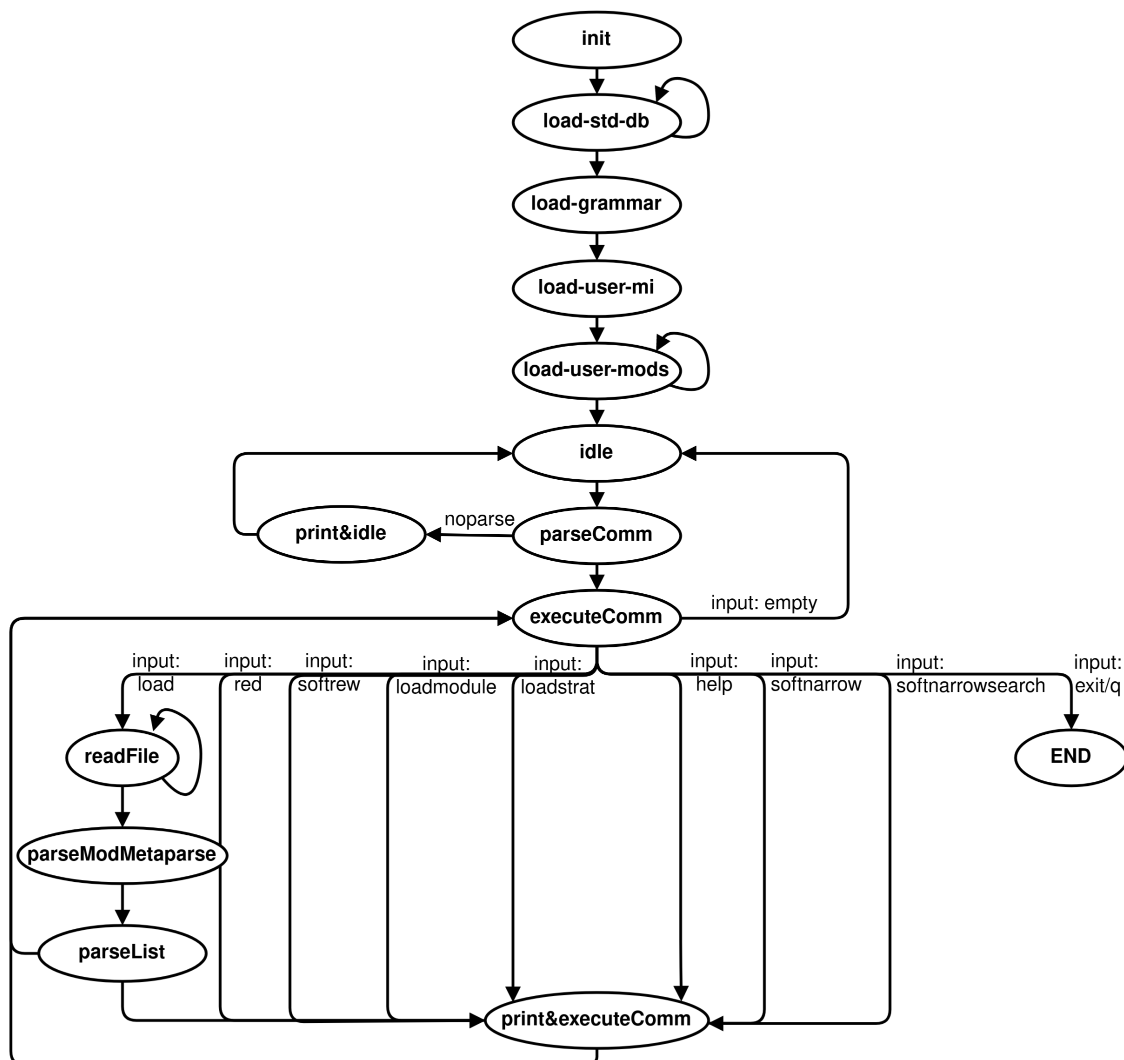

Figura 3.1: Diagrama de estados de PLSS.

Para guiar los pasos intermedios se utilizan varios mensajes, donde `plss` será el objeto y `PLSS` su clase. Estos son algunos ejemplos:

```
insertModule(MI, plss, upModule(Q, false)) .
write(stdout, plss, STR)
parseTerm(MI, plss, 'GRAMMAR, none, tokenize(Text), '@Input@) .
createInterpreter(interpreterManager, plss, none) .
```

El entorno de PLSS se inicia utilizando la configuración `run` que está formado por las constantes de configuración, con ello se creará un intérprete y se enviará el prompt del programa al flujo de salida:

```
< plss : PLSS | file: null, out: "", state: init, load: modList,
        uload: umodList, input: empty, module: userModule, strat: 'undefZero >
```

Los atributos que se asocian al objeto pertenecen a la clase `Attribute` y se definen junto con sus valores:

```
*** Attributes
op uload:_ : List{Preload} -> Attribute [ctor gather (&)] .
op load:_ : List{Preload} -> Attribute [ctor gather (&)] .
op module:_ : Maybe{Module} -> Attribute [ctor] .
op file:_ : Maybe{Oid} -> Attribute [ctor] .
op input:_ : TermList -> Attribute [ctor] .
op state:_ : State -> Attribute [ctor] .
op out:_ : String -> Attribute [ctor] .
op umi:_ : Oid -> Attribute [ctor] .
op mi:_ : Oid -> Attribute [ctor] .
op strat:_ : Maybe{Qid} -> Attribute [ctor] .
```

Una vez escrito el mensaje y creado el metaintérprete, se inserta la lista de módulos que configuran la sintaxis de PLSS:

```
eq modList = module('TRUTH-VALUE) module('BOOL-OPS) module('TRUTH)
             module('BOOL) module('EXT-BOOL) module('NAT) module('RANDOM)
             module('COUNTER) module('INT) module('RAT) module('FLOAT)
             module('STRING) module('CONVERSION) module('BOUND)
             module('QID) view('Qid) module('TRIV) view('TRIV)
             module('LIST) module('QID-LIST)
             module('COMMON-SIGN) module('COMMAND-SIGN) module('PLSS-SIGN) .
```

Los módulos y vistas que se utilizan se van cargando con reglas similares a la regla `init-module`:

```
rl [init-module] :
   createdInterpreter(plss, interpreterManager, MI)
   < plss : PLSS | state: init, load: module(Q) LP, AtS >
=> < plss : PLSS | state: load-std-db, mi: MI, load: LP, AtS >
   insertModule(MI, plss, upModule(Q, false)) .
```

En esta regla se carga el primer módulo de la lista de módulos (disponible en el atributo `load:`. El segundo argumento del mensaje creado `createdInterpreter` es el remitente `interpreterManager`. Después PLSS carga el módulo con la gramática a través de la regla `load-predef-finished`:

```
rl [load-predef-finished] :
   insertedModule(plss, MI)
   < plss : PLSS | state: load-std-db,  load: nil, AtS >
=> < plss : PLSS | state: load-grammar, load: nil, AtS >
   insertModule(MI, plss, GRAMMAR) .
```

De la misma forma, a continuación se crea el metaintérprete de usuario y los módulos de su sintaxis:

```
rl [user-metainterpreter-created] :
```

```
   createdInterpreter(plss, interpreterManager, MI)
   < plss : PLSS | state: load-user-mi, uload: module(Q) LP, AtS >
=> < plss : PLSS | state: load-user-mods, umi: MI, uload: LP, AtS >
   insertModule(MI, plss, upModule(Q, false)) .

rl [load-user-predef-module-metainterpreter] :
   insertedModule(plss, MI)
   < plss : PLSS | state: load-user-mods, uload: module(Q) LP, AtS >
=> < plss : PLSS | state: load-user-mods, uload: LP, AtS >
   insertModule(MI, plss, upModule(Q, false)) .
```

Una vez que se han insertado todos los módulos del metaintérprete, se envía un mensaje `getLine` al objeto `stdin`, que contiene el prompt del programa (`PLSS>`).

```
rl [user-predef-module-metainterpreter-finished] :
   insertedModule(plss, MI)
   < plss : PLSS | state: load-user-mods, uload: nil, AtS >
=> < plss : PLSS | state: idle, AtS >
   getLine(stdin, plss, "PLSS>") .
```

En ese momento el programa se sitúa en el estado `idle`, preparado para recibir comandos. Cuando el usuario introduce algunas entradas, el objeto `stdin` envía un mensaje del tipo `gotLine` con la cadena introducida. Las entradas se analizan por el metaintérprete y toda la lista `TermList` se almacena en el atributo `input`. Por ejemplo, el comando `exit` responde con un mensaje de despedida:

```
rl [exit] :
   < plss : PLSS | state: executeComm, input: ('exit.@Command@), AtS >
=> write(stdout, plss, "Thanks for using PLSS!\n") .
```

Si el análisis tiene éxito, el metaintérprete envía un mensaje `parsedTerm` con un término de tipo `ResultPair`. Este término se envía a la siguiente función y se sitúa en el estado `parseComm` para que continúe la ejecución.

Por otra parte, cuando falla el análisis, se envía un mensaje del metaintérprete del tipo `parsedTerm` con el término `noParse`, indicando la posición en la que se produjo el fallo en el análisis. En el caso de tener una entrada incompleta, se indica la posición final. En este caso, esta estrada incompleta se añade a la entrada actual y se pide al usuario que complete el comando.

```
rl [parseCommError] :
   parsedTerm(plss, MI, noParse(N))
   < plss : PLSS | mi: MI, state: parseComm, AtS >
=> < plss : PLSS | mi: MI, state: print&idle, AtS >
   write(stdout, plss, "The introduced command does not exit.\n") .
```

Como puede verse en la figura 3.1, cada vez que concluye un comando el programa vuelve al estado `idle` tras notificarlo al usuario, independientemente del resultado del análisis.

En la siguiente sección se comenta con detalle la codificación de los comandos de simplificación y de la manipulación de estrategias.

3.5. Comandos de simplificación con ecuaciones y reglas *soft*

Se ha definido una gramática de comandos a partir de la cual se realiza el análisis sintáctico. Los comandos definen un tipo especial "burbuja" que simplemente corresponde a una secuencia de tokens que serán analizados posteriormente. Si queremos extender la herramienta, solo tenemos que modificar la gramática, incluir el nuevo comando y luego definir reglas para ejecutarlos.

La sintaxis de los comandos se definen en el fichero `grammar.maude`, como se muestra a continuación para el comando `red`:

```
op red_ : @Bubble@ -> @Command@ [ctor] .
```

Como se puede observar, este comando recibe una burbuja como argumento de entrada.

Por una parte, el comando `red` realiza la simplificación ecuacional del término introducido. Su código se muestra en el listado 3.9. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe una burbuja como argumento y la regla `redComm` realiza las siguientes tareas:

1. Análisis de la burbuja con el operador `metaParse`:

```
{T', Ty} := metaParse(M, downQidList(T), anyType)
```

2. Reducción del término analizado con el operador `metaReduce`:

```
{T'', Ty'} := metaReduce(M, T')
```

3. Presentación mejorada del término reducido con el operador `metaPrettyPrint`:

```
QIL := metaPrettyPrint(M, T'')
```

4. Con el operador `printTokens`, preparación del mensaje a mostrar con la lista de tokens devueltos:

```
STR := "\nResult: " + printTokens(QIL) + "\n"
```

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

Listado 3.9: Comando `red` en PLSS.

```
crl [redComm] :
    < plss : PLSS | state: executeComm, input: ('red_['bubble[T]], INPLST),
        module: M, AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
    write(stdout, plss, STR)
 if {T', Ty} := metaParse(M, downQidList(T), anyType) /\
```

```
    {T'', Ty'} := metaReduce(M, T') /\
    QIL := metaPrettyPrint(M, T'') /\
    STR := "\nResult: " + printTokens(QIL) + "\n" .
```

Por otra parte, el comando `softrew` realiza la simplificación del término introducido con las *soft* reglas, es decir, con las reglas escogidas de acuerdo con la función de elección definida en los *soft sets*. Su código se muestra en el listado 3.10. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe una burbuja como argumento y la regla `softrewstratComm` realiza las siguientes tareas:

1. Análisis de la burbuja con el operador `metaParse` y asignación mediante encaje de patrones con las variables `DISH` y `ATTS` de los valores correspondientes:

```
{'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish)
```

2. Reescritura del término analizado con el operador `rewStrat`, que se comentó en la sección 3.2:

```
T' := rewStrat(M, DISH, ATTS, MQ)
```

3. Presentación mejorada del término reducido con el operador `metaPrettyPrint`:

```
QIL := metaPrettyPrint(M, T'')
```

4. Con el operador `printTokens`, preparación del mensaje a mostrar con la lista de tokens devueltos:

```
STR := "\nResult: " + printTokens(QIL) + "\n"
```

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

Listado 3.10: Comando `softrew` en PLSS.

```
crl [softrewstratComm] :
    < plss : PLSS | state: executeComm, input: ('softrew_['bubble[T]], INPLST)
        , module: M, strat: MQ, AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, module: M, strat:
      MQ, AtS >
    write(stdout, plss, STR)
 if {'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish) /\
    T' := rewStrat(M, DISH, ATTS, MQ) /\
    QIL := metaPrettyPrint(M, T') /\
    STR := "\nResult: " + printTokens(QIL) + "\n" .
```

Por último se ha implementado un comando con la función de realizar el análisis mediante el estrechamiento. El comando `softnarrow` reescribe el término introducido para la simplificación de la búsqueda mediante *narrowing*. En la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe una burbuja como argumento y la regla `softnarrowstratComm` realiza las siguientes tareas:

1. Análisis de la burbuja con el operador `metaParse` y asignación mediante matching a las variables `DISH` y `ATTS` de los valores correspondientes:

```
{'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish)
```

2. Reescritura del término analizado con el operador `narrowStrat`, que se comentó en la sección 3.2:

```
T' := narrowStrat(M, DISH, ATTS, MQ)
```

3. Presentación mejorada del término reducido con el operador `metaPrettyPrint`:

```
QIL := metaPrettyPrint(M, T')
```

4. Con el operador `printTokens`, se prepara el mensaje a mostrar con la lista de tokens devueltos:

```
STR := "\nResult: " + printTokens(QIL) + "\n"
```

Al final envía el mensaje `STR` al metaintérprete y PLSS pasa al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de entrada.

El listado 3.11 muestra el código de esta regla.

Listado 3.11: Regla `softnarrowstratComm`.

```
crl [softnarrowstratComm] :
    < plss : PLSS | state: executeComm, input: ('softnarrow_['bubble[T]],
       INPLST), module: M, strat: MQ, AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, module: M, strat:
      MQ, AtS >
    write(stdout, plss, STR)
 if {'__[DISH, ATTS], 'SoftDish} := metaParse(M, downQidList(T), 'SoftDish) /\
    T' := narrowStrat(M, DISH, ATTS, MQ) /\
    QIL := metaPrettyPrint(M, T') /\
    STR := "\nResult: " + printTokens(QIL) + "\n" .
```

3.6. Comando de búsqueda con narrowing

En esta sección se describe el comando de búsqueda `softnarrowsearch` mediante estrechamiento. De esta forma, se analizarán todos los cálculos de un conjunto de estados gracias a la descripción de los mismos mediante técnicas simbólicas.

El comando `softnarrowsearch` se encarga de realizar la búsqueda *narrowing* de un término inicial a un término final. En lo referente a los estados, después del análisis del comando, se pasa al estado `executeComm`. Este comando recibe dos burbujas como argumentos de entrada. Estos argumentos se corresponden con un término `SoftDish` inicial y

otro término `SoftDish` que será el patrón a alcanzar. Ambos términos pueden contener variables, que serán asociadas cuando se complete la ejecución de `metaNarrowingSearch`.

La regla `softNarrowSearchComm` realizará las tareas necesarias para que el comando `softnarrowsearch` se pueda ejecutar. En la implementación de esta búsqueda, se han codificado tres reglas para cada uno de los siguientes casos:

1. Error en el análisis sintáctico de los términos.
2. No existe resultado en la búsqueda.
3. Se encuentra al menos una solución.

En el primer caso, cuando existe un error de parseo en alguno de los argumentos de entrada del `metaNarrowingSearch`, se resuelve mediante el uso de disyunción de dos condiciones negativas. Cada condición consiste en comprobar que el `metaParse` de cada uno de los dos términos (`T` y `T1`) no pertenece al tipo `ResultPair`:

```
( not (metaParse(M, downQidList(T),  'SoftDish) :: ResultPair)  or
  not (metaParse(M, downQidList(T1), 'SoftDish) :: ResultPair))
```

Cuando se ejecuta esta regla, se muestra el siguiente mensaje de error:

```
STR := "\nError en los términos parseados para MetaNarrowingSearch.\n\n"
```

El listado 3.12 muestra el código de esta regla.

Listado 3.12: Regla `softNarrowSearchComm` en el caso 1.

```
crl [softNarrowSearchComm] :
< plss : PLSS | state: executeComm, input: ('softnarrowsearch__['bubble[T],
         'bubble[T1]], INPLST), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
   write(stdout, plss, STR)
if (not (metaParse(M, downQidList(T), 'SoftDish) :: ResultPair) or
  not (metaParse(M, downQidList(T1), 'SoftDish) :: ResultPair)) /\
  STR := "\nError en los términos parseados para MetaNarrowingSearch.\n\n" .
```

El segundo caso se encarga de comprobar que la búsqueda con `metaNarrowingSearch` no encuentra solución. Para ello, se comprueba que el resultado no sea del tipo `NarrowingSearchResult`, es decir, que sea la constante `failure` o que exista ambigüedad (`failure`):

```
not (metaNarrowingSearch(M, T', T1', '*, unbounded, 'none, 0) ::
    NarrowingSearchResult)
```

Cuando se ejecuta esta regla, se muestra el siguiente mensaje:

```
STR := "\nNo solución en la búsqueda con MetaNarrowingSearch.\n\n"
```

El listado 3.13 muestra el código de esta regla.

Listado 3.13: Regla softNarrowSearchComm en el caso 2.

```
crl [softNarrowSearchComm] :
< plss : PLSS | state: executeComm, input: ('softnarrowsearch__['bubble[T],
         'bubble[T1]], INPLST), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
   write(stdout, plss, STR)
if  {T', Ty} := metaParse(M, downQidList(T), 'SoftDish) /\
  {T1', T1y} := metaParse(M, downQidList(T1), 'SoftDish) /\
  not (metaNarrowingSearch(putNarrowingAtt(M), T', T1', '*, unbounded, 'none,
      0) :: NarrowingSearchResult) /\
  STR := "\nNo solución en la búsqueda con MetaNarrowingSearch.\n\n" .
```

En el tercer caso, el comando `metaNarrowingSearch` encuentra al menos una solución, que será del tipo `NarrowingSearchResult`. La asignación de las salidas de la solución de este comando se realiza con la siguiente instrucción:

```
{T'', 'SoftDish, VSubs:Substitution, VQid:Qid, VSubs2:Substitution, VQid2:Qid }
     := metaNarrowingSearch(M, T', T1', '*, unbounded, 'none, 0)
```

Los argumentos de entrada del comando `metaNarrowingSearch` son:

- El primer argumento de entrada (`M`) corresponde a la metarrepresentación del módulo en que se realice la búsqueda.
- Los siguientes argumentos (`T'` y `T1'`) son, respectivamente, las metarrepresentaciones del término inicial y la del patrón a alcanzar.
- Se continúa con la metarrepresentación de la flecha de búsqueda, en nuestro caso `'*`, correspondiente a `=>*` en la ejecución del comando `search` (es decir, usar 0 o más pasos).
- El siguiente argumento es la longitud máxima de narrowing. En nuestro caso es `unbounded` del tipo `Bound`.
- Después, podrán aparecer las constantes `'match` o `'none`. La diferencia entre ellas, consiste en que la constante `'none` indica que se aplica una reducción estándar sin ningún plegado, mientras que con la constante `'match` se aplica la reducción de plegado.
- El último argumento será un numero natural, que representa la solución elegida. De esta forma, como en muchos otros comandos de metanivel, se pueden proporcionar todas las soluciones de forma secuencial.

Los elementos de la salida de `NarrowingSearchResult` se corresponden con:

- El primer elemento `Term` (T'') es el término patrón, es decir, el término al que se desea llegar, junto con la última sustitución.
- El segundo elemento se corresponde con `Type` ('SoftDish), que es la metarrepresentación del tipo del término.
- `Substitution` (`VSubs:Substitution`) es equivalente a la sustitución para conseguirlo, junto con la última sustitución incorporada.

- El siguiente elemento es un `Qid` (`VQid:Qid`) que equivale a la metarrepresentación del identificador utilizado para crear nuevas variables.

- `Substitution` (`VSubs2:Substitution`) es la última sustitución y por último, otro `Qid` (`VQid2:Qid`) que metarrepresenta el identificador utilizado para el unificador de variante.

El listado 3.14 muestra el código de esta regla.

Listado 3.14: Regla `softNarrowSearchComm` en el caso 3.

```
crl [softNarrowSearchComm] :
< plss : PLSS | state: executeComm, input: ('softnarrowsearch__['bubble[T],
    'bubble[T1]], INPLST), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module: M, AtS >
   write(stdout, plss, STR)
if  {T', Ty} := metaParse(M, downQidList(T), 'SoftDish) /\
   {T1', T1y} := metaParse(M, downQidList(T1), 'SoftDish) /\
   metaNarrowingSearch(putNarrowingAtt(M), T', T1', '*, unbounded, 'none, 0) ::
        NarrowingSearchResult /\
   {T'', 'SoftDish, VSubs:Substitution, VQid:Qid, VSubs2:Substitution,
       VQid2:Qid} :=
   metaNarrowingSearch(putNarrowingAtt(M), T', T1', '*, unbounded, 'none, 0) /\
   QIL := metaPrettyPrint(M, T'') /\
   QIL1 := printSB(M, VSubs:Substitution) /\
   QIL2 := printSB(M, VSubs2:Substitution) /\
   QIL3 := metaPrettyPrint(M, T') /\ QIL4 := metaPrettyPrint(M, T1') /\
   STR := "\nResult (SoftDish-term): " + printTokens(QIL) +
     "\nSubstitution1: " + printTokens(QIL1) +
     "\nSubstitution2: " + printTokens(QIL2) +
     "\nTerm-ini (T): "  + printTokens(QIL3) +
     "\nTerm-patron (T1): " + printTokens(QIL4) + "\n\n" .
```

Para mostrar adecuadamente las sustituciones del resultado se ha creado la función `printSB`. Con esta función, dada una sustitución, se devuelve una lista de `Qid`s.

```
op printSB : Module Substitution -> QidList .
eq printSB (M, none) = 'none .
eq printSB (M, V:Variable <- T:Term) =
   getName(V:Variable)  '<-  metaPrettyPrint(M, T:Term) .
eq printSB (M, V:Variable <- T:Term ; S:Substitution) =
   printSB (M, V:Variable <- T:Term)  printSB(M, S:Substitution) .
```

Al final de cada uno de estos casos, el mensaje `STR` se envia al metaintérprete y PLSS pasa al estado `print&executeComm` para que muestre la cadena correspondiente y pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.7. Otros comandos de PLSS

En esta sección se describen los comandos de carga de ficheros con instrucciones (`load`), de asignación de la función de elección (`loadstrat`), de carga de los distintos módulos que contienen los modelos biológicos de Pathway Logic (`loadmodule`) y el sistema de ayuda (`help`).

3.7.1. Comando de carga de ficheros con instrucciones

El comando `load` realiza la carga de ficheros con instrucciones. Su código se muestra en el listado 3.15. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

El comando recibe como argumento un token con el nombre del fichero a cargar y se realizan las siguientes tareas:

1. La regla `loadComm` recibe el nombre del fichero como argumento y lo abre. También pasa a PLSS al estado `readFile`.
2. La regla `openedFile` coloca el nombre del fichero en el atributo `file:`.
3. La regla `readingModule` va leyendo cada una de las líneas del fichero. El texto leído lo almacena en el atributo `out:`. Cuando se encuentra con una línea vacía, envía el mensaje de cerrar el fichero.
4. La regla `closedFile` cierra el fichero con el nombre dado y pasa al estado `parseModMetaparse`. Además vacía el atributo `file:`, que antes contenía el nombre del fichero.
5. Las reglas `parseModuleMetaparse`, `parseModuleNoBubblesOK` y `parseModuleNoBubblesError` llevan a cabo el análisis del contenido del módulo y en caso de error muestra el mensaje correspondiente.

Al final se colocan todas las instrucciones en el atributo `input:` y pasa PLSS al estado `print&executeComm`. De esta forma, se irán ejecutando uno por uno todos los comandos que se han colocado en la entrada.

Listado 3.15: Comando `load` en PLSS.

```
crl [loadComm] :
 < plss : PLSS | state: executeComm, input: ('load_['token[T]], INPLST), AtS >
=> < plss : PLSS | state: readFile, input: INPLST, AtS >
   openFile(fileManager, plss, STR, "r")
 if STR := string(downQid(T)) .

rl [openedFile] : openedFile(plss, fileManager, FHIn)
   < plss : PLSS | file: null, state: readFile, AtS >
=> < plss : PLSS | file: FHIn, state: readFile, AtS >
   getLine(FHIn, plss) .

rl [readingModule] : gotLine(plss, FHIn, Text)
   < plss : PLSS | file: FHIn, out: Read, state: readFile, AtS >
=> if Text == ""
   then < plss : PLSS | file: FHIn, out: Read, state: readFile, AtS >
        closeFile(FHIn, plss)
   else < plss : PLSS | file: FHIn, out: (Read + Text), state: readFile, AtS >
        getLine(FHIn, plss)
   fi .

rl [closedFile] : closedFile(plss, FHIn)
   < plss : PLSS | file: FHIn, state: readFile, AtS >
=> < plss : PLSS | file: null, state: parseModMetaparse, AtS > .

rl [parseModuleMetaparse] :
   < plss : PLSS | out: Read, state: parseModMetaparse, mi: MI, AtS >
=> < plss : PLSS | out: Read, state: parseList, mi: MI, AtS >
   parseTerm(MI, plss, 'GRAMMAR, none, tokenize(Read), '@Input@) .
```

```
crl [parseModuleNoBubblesOK] :
    parsedTerm(plss, MI, {T, Ty})
    < plss : PLSS | out: Read, state: parseList, mi: MI, input: INPLST, AtS >
 => < plss : PLSS | out: "", state: executeComm, mi: MI, input: (TL, INPLST),
     AtS >
 if TL := extractTerms(T) .

crl [parseModuleNoBubblesError] :
    parsedTerm(plss, MI, noParse(N))
    < plss : PLSS | out: Read, state: parseList, mi: MI, AtS >
 => < plss : PLSS | out: "", state: print&executeComm, mi: MI, AtS >
    write(stdout, plss, printTokens(QIL))
 if QIL := showMsg(tokenize(Read), N) .
```

3.7.2. Comando de asignación de la función de elección

El comando `loadstrat` realiza la asignación de la función de elección de los *soft sets*. Para la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

```
crl [loadstratComm] :
   < plss : PLSS | state: executeComm, input: ('loadstrat_['token[T]], INPLST)
       , strat: MQ, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, strat: downQid(T),
     AtS >
    write(stdout, plss, STR)
 if STR := "\nLoaded strategy function ( " +  string(downQid(T)) + " )\n" .
```

El comando recibe un token con el nombre de la función de estrategia como argumento y entonces la regla `loadstratComm` coloca ese término en el atributo `strat: downQid(T)`.

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.7.3. Comando de selección del modelo de Pathway Logic

El comando `loadmodule` realiza la asignación del modelo biológico con el que se desea trabajar. Estos módulos vienen representados como: `QQ1` es el equivalente al modelo que corresponde con `TGFB1`, el módulo `QQ2` hace referencia al modelo `EGF`, el `QQ3` con `HGF`, el `QQ4` con `IL6` y por último el módulo `QQ5` se corresponde con el modelo `NGF`.

Estos modelos biológicos son necesarios para la ejecución de los comandos de simplificación y búsqueda con los términos del tipo `SoftDish`. En la codificación de este comando, después del análisis, se pasa al estado `executeComm`.

```
crl [loadmoduleComm] :
   < plss : PLSS | state: executeComm, input: ('loadmodule_['token[T]], INPLST
       ), module: M, AtS >
=> < plss : PLSS | state: print&executeComm, input: INPLST, module:
    putNarrowingAtt(upModule(downQid(T), true)), AtS >
    write(stdout, plss, STR)
 if STR := "\nLoaded Pathway Logic module ( " +  string(downQid(T)) + " )\n" .
```

El comando recibe como argumento un `token` con el nombre de un módulo que representa el modelo biológico. La regla `loadmoduleComm` se encarga de colocar ese término en el atributo correspondiente: `module: putNarrowingAtt(upModule(downQid(T), true))`.

De esta forma, todas las reglas del nuevo módulo que se introduce tendrán añadido el atributo `narrowing`, por la función `putNarrowingAtt` que se explica en la subsección 3.2.2 del operador `narrowStrat`.

Al final envía el mensaje `STR` al metaintérprete y PLSS pasa al estado `print&executeComm` para que muestre el resultado y pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.7.4. Comando del sistema de ayuda

El comando `help` muestra la ayuda para PLSS:

```
 rl [help] :
    < plss : PLSS | state: executeComm, input: ('help.@Command@, INPLST), AtS >
 => < plss : PLSS | state: print&executeComm, input: INPLST, AtS >
    write(stdout, plss, help + "\n") .
```

El comando no recibe argumentos, simplemente muestra el contenido de la ayuda que está almacenado en la constante `help`[4]:

```
op help : -> String .
eq help =
  "\n   ======================== \n" +
  "   AYUDA DEL PROGRAMA: PLSS\n" +
  "   ========================\n\n" +
  "   1. load + ( Fichero de texto ): \n" +
  "      Carga el fichero y ejecuta los comandos incluidos en el fichero. \n" +
  "      Cuando exista más de una instrucción, cada una deberá estar \n" +
  "      entre paréntesis. \n" +
  "      Ejemplo: \n" +
  "      PLSS> load softsetfile.txt \n" +
  "       \n" +
  .../...
```

Al final envía el mensaje `STR` al metaintérprete y pasa PLSS al estado `print&executeComm` para que muestre el resultado y que pase de nuevo al estado `idle` a la espera de un nuevo comando o bien al estado `executeComm` para ejecutar el resto de comandos de la entrada.

3.8. Integración de los módulos en PLSS

En esta sección se describen los módulos y ficheros desarrollados para este proyecto. En la figura 3.2 se muestran de forma gráfica las dependencias (importación) entre los distintos ficheros del proyecto.

Los módulos que se han desarrollado están incluidos en los siguientes ficheros:

- `plss.maude`: módulo principal del proyecto PLSS.
- `grammar.maude`: definición de la gramática de PLSS. En el módulo `COMMAND-SIGN` se definen los comandos (`load`, `red`, etc.).
- `plallSS.maude`: fichero con los operadores y reglas adaptadas de Pathway Logic.

[4] Se recuerda que el código completo del proyecto está disponible en `https://github.com/rsantosb/TFM-PLSS`.

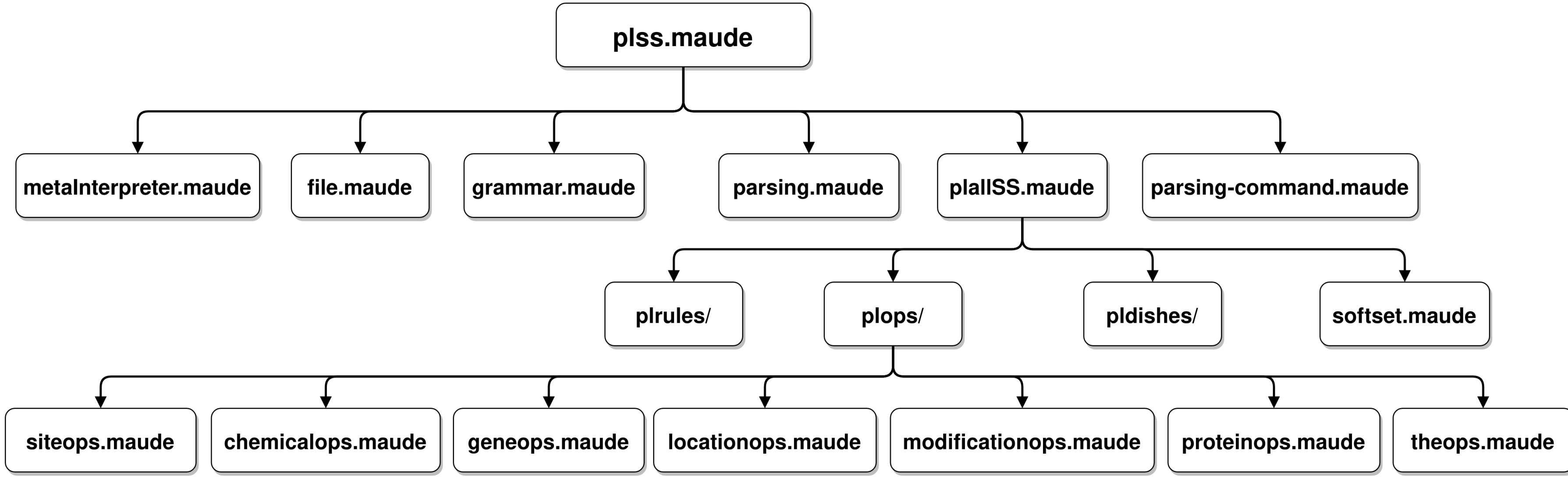

Figura 3.2: Dependencia de ficheros Maude. La carpeta `plops` es compartida por los cinco módulos que PLSS puede cargar. Las carpetas `pldishes` y `plrules` contienen los discos y las reglas para estos cinco modelos de Pathway Logic.

- Las carpetas `ops`, `rules` y `SDishes` contienen los ficheros auxiliares para trabajar con los modelos de Pathway Logic.
- `parsing-command.maude` realiza el análisis de los comandos. Además incluye el módulo `COMMAND-PROCESSING`.
- `parsing.maude` realiza el resto de tareas de análisis y también contiene módulos auxiliares de la base de datos, mensajes, errores y otras operaciones con módulos.
- `softset.maude`: fichero de definición de *soft sets* con los comandos de reescritura `rewStrat` y `narrowStrat` junto con sus funciones de estrategias.
- `softsetfile.txt`, `softsetfile1.txt`, `softsetfile2.txt`: ejemplos de ficheros con instrucciones para importar con la instrucción `load`.
- `test_plss.txt`: ejemplos de ejecución de comandos para utilizar dentro de PLSS con sus salidas.

Además de los ficheros descritos, se han utilizado los ficheros `metainterpreter.maude` y `file.maude` incluidos en la distribución de Maude 3.1.

En lo referente a los módulos, las figuras 3.3-3.5 muestran la relación de dependencia entre ellos. La figura 3.3 muestra la dependencia entre otros módulos, por ejemplo los para su correcta gramática, la figura 3.4 se centra en los módulos específicos de Pathway Logic y la figura 3.5 agrupa los módulos de uso general de PLSS.

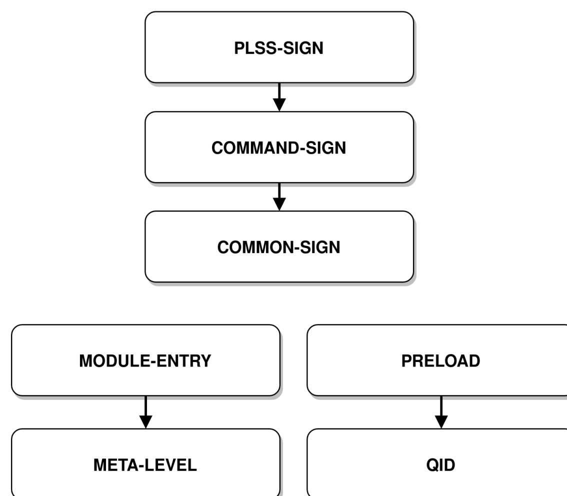

Figura 3.3: Dependencia de otros módulos.

3.9. Instalación y ejecución de PLSS

En esta sección se explica el procedimiento de instalación y el funcionamiento del programa con algún ejemplo.

El único requisito previo para el uso de PLSS es la instalación de Maude (Clavel et al., 2020). Es necesaria la versión 3.1 o posterior. Después solo hay que descargar los ficheros del repositorio en un único directorio.

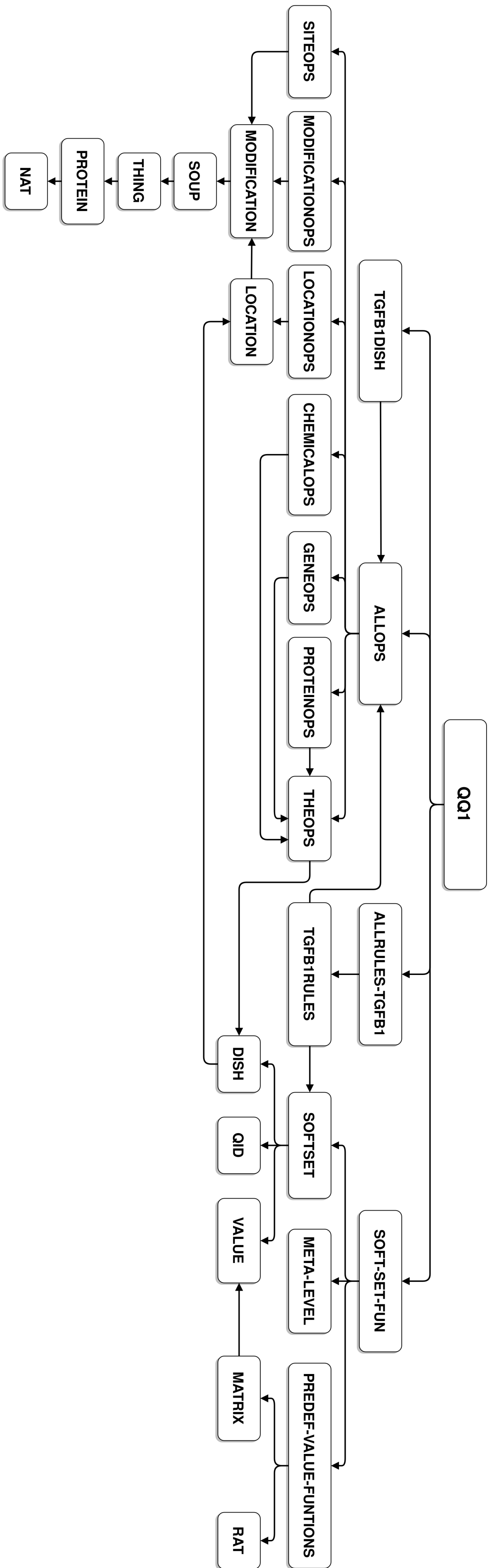

Figura 3.4: Dependencia de módulos específicos de Pathway Logic. En PLSS hay hasta 5 módulos que representan distintos modelos biológicos en Pathway Logic. Además de los contenidos, la única diferencia entre ellos es el nombre del fichero de dishes y rules. En esta figura se representa el modelo TGFB1.

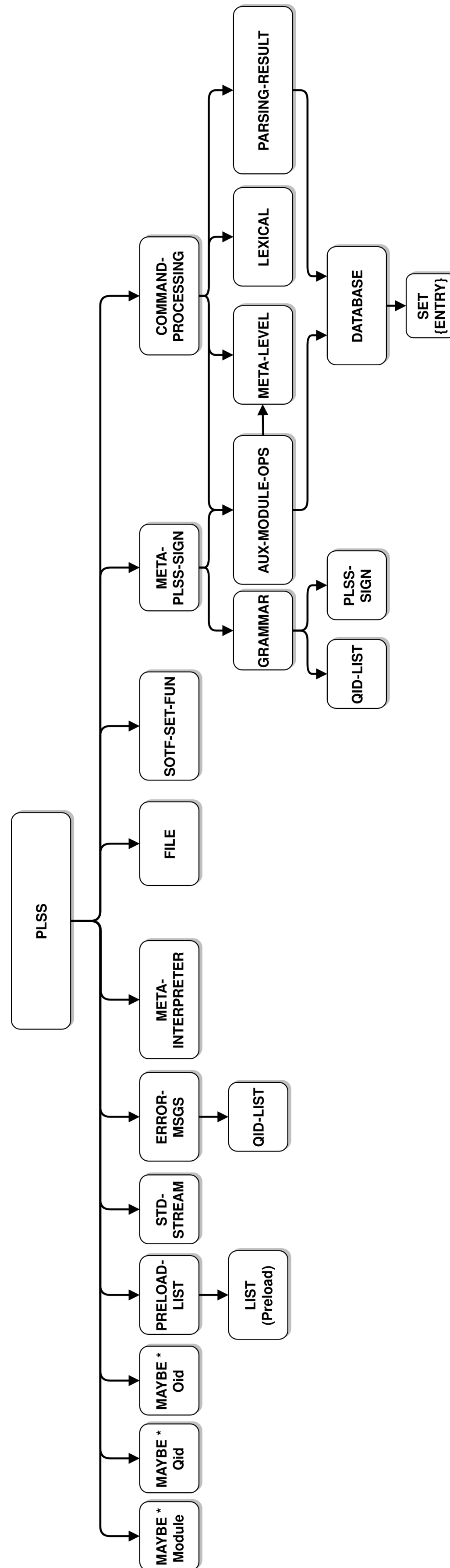

Figura 3.5: Dependencia de módulos de uso general de PLSS.

Respecto del funcionamiento del programa, una vez descargado el código del repositorio, en primer lugar se debe ejecutar el módulo principal `plss.maude` en Maude con la opción `allow-files` para poder abrir ficheros externos:

```
$ maude -allow-files plss
               \||||||||||||||||||/
             --- Welcome to Maude ---
               /||||||||||||||||||\
      Maude 3.1 built: Oct 12 2020 20:13:41
      Copyright 1997-2020 SRI International
             Wed Jan  6 19:04:10 2021
==========================================
erewrite in PLSS : run .
PLSS>
```

Se ejecutó el comando `erewrite` para la configuración `run`, que está formado por las constantes de configuración, con ello se creará un intérprete y se enviará el prompt del programa al flujo de salida:

```
< plss : PLSS | file: null, out: "", state: init, load: modList, uload:
    umodList, input: empty, module: userModule, strat: 'undefZero >
```

A continuación, se muestra un ejemplo con el comando `softrew`:

```
PLSS> softrew IL6Dish ([il6r = 1])
```

La reescritura con el comando `softrew` se realiza sobre un término del tipo `SoftDish`, es decir, un dish con unos atributos. En este caso, el dish es `IL6Dish` (definido en el módulo de los dishes `IL6DISH`), con un único atributo `il6r` con valor `1`.

El resultado que nos muestra PLSS es:

```
Result softrew: PD(
{CLc | Akt1 Erks Gab1 Gab2 Hck Jak1 Jak2 Shp2 Stat1[Mkk4 - act][Tyk2 - Yphos]}
{CLi | empty}
{CLm | empty}
{GP130C |(Gp130 : IL6 : IL6R)[Pkcd - act][Rac1 - GTP][Vav1 - Yphos][Vav2 -
    Yphos][Vav3 - Yphos]}
{NUc | Foxo1 Irf1-gene Lmo4[A2m-gene - on][RankL-gene - on][Socs3-gene - on][
    Stat3 - act]}
{XOut | empty}) [il6r = 1]
```

Se puede observar que las reglas de reescritura transforman por completo el plato inicial. Con la regla `SS1221.IL6R.irt.IL6` el disco inicial se transforma en las localizaciones `Xout` y `CLm` vacías, pasando las proteínas `IL6` y `Gp130` a estar ligadas en la localización `GP130C` junto con el receptor `IL6R`. Esta regla deja el atributo `il6r` con valor uno.

Con la regla `1237.Stat3.to.Gp130C.irt.IL6`, dada la anterior ligadura y teniendo `Stat3` en el `CLc`, la localización `GP130C` muta a la siguiente forma: `[Stat3 - Yphos]`. Dada esta situación, si además `CLc` posee `Pkcd` junto con los cambios mencionados antes, transmuta mediante la regla `1236.Pckd.irt.IL6` en `GP130C` con `[Pkcd - act]`.

También se utiliza la regla `1235.Rac1.irt.IL6` para transformar `[Rac1 - GDP]` de `CLi` a `[Rac1 - GTP]`. Si se aplica la regla `1239.Stat3.Sphos.irt.IL6` y después, con la regla `1238.Stat3.to.nuc.irt.IL6`, `[Stat3 - Yphos]` se moverá al núcleo como `[Stat3 - act]`.

Como en la localización `CLc` del dish del que partimos se encuentran `Vav1`, `Vav2` y `Vav3`; la regla `1234.Vavs.irt.IL6` los transforma en `Gp130C` como `[Vav1 - Yphos][Vav2 - Yphos][Vav3 - Yphos]`.

Los genes del núcleo se activan con las siguientes reglas:

- La regla `1222.A2m-gene.irt.IL6` activa `[A2m-gene - on]`, como premisa el núcleo debe tener: `Foxo1 [Stat3 - act] Lmo4 A2m-gene`.
- Con la regla `1233.Mkk4.irt.IL6`, se activará `Mkk4` si en `GP130C` existe la ligadura inicial y además se encuentra `Rac1` con la forma `GTP`.
- La regla `1240.RankL-gene.irt.IL6` activa el gen `RankL`.
- La regla `1242.Socs3-gene.irt.IL6` activa `[Socs3-gene - on]`.
- Dado un `Tyk2` en `CLc`, con la regla `1244.Tyk2.irt.IL6`, `Tyk2` pasa al estado `Yphos`.

Por último, existen dos comandos `exit` y `q` que terminarán con la ejecución y muestran un mensaje de despedida:

```
PLSS> exit

Thanks for using PLSS!
rewrites: 4234 in 580ms cpu (89892ms real) (7299 rewrites/second)
result Portal: <>
```

Capítulo 4

Análisis de resultados

"Los datos son una cosa preciosa y
durarán más tiempo que los propios sistemas"
— Tim Berners-Lee

En este capítulo se realiza un análisis comparativo entre las funciones de elección para los comandos de simplificación. Después, se realiza un análisis entre los distintos comandos de simplificación y reescritura. También existe una sección para del análisis del comando de búsqueda. Por último, se explican los diferentes pathways que soporta la aplicación y se comentan algunos casos concretos de búsquedas biológicas sobre esas rutas de señalización.

4.1. Análisis y comparación con las funciones de elección

El objetivo de esta sección es ver el comportamiento de cada una de las funciones de elección. Tenemos cinco formas de elegir el mejor término a reescribir y queremos confirmar que esos criterios son distintos. Estas funciones también llamadas estrategias, se definen en la sección 3.1. Con `loadstrat` se cargan las diferentes estrategias para utilizarse en los comandos `softrew` y `softnarrow`.

En primer lugar elegimos el modelo con la ruta de señalización con la que queramos trabajar. En este caso, cargamos el modelo `IL6` con la instrucción `loadmodule QQ4`. De esta forma tendremos a disposición los dishes y las reglas asociadas a este modelo.

En una primera ejecución realizamos un experimento para tener una comparativa entre dos estrategias de elección `undefZero` y `undefOne`. Para elegir la función de elección deseada, utilizamos el comando `loadstrat`, por ejemplo, escribimos `loadstrat undefZero`.

Pretendemos realizar la reescritura de un término con dos estrategias y analizar sus resultados:

```
softrew (SmallDish  ([il6r = 1], [hck = 0], [akt1 = *]))
```

La reescritura con el comando `softrew` se realiza sobre un término del tipo `SoftDish`, es decir, un dish con unos atributos/valores. En este caso, el dish es `SmallDish`, que es un dish con pocas proteínas definido en el módulo de los dishes `IL6DISH`, y los atributos que se han utilizado son `il6r`, `hck` y `akt1`, que tienen valores `1`, `0` y `*`, respectivamente.

El resultado que nos proporciona PLSS es:

```
Result softrew: PD(
{CLc | Hck[Akt1 - act]}
{CLm | empty}
{GP130C | Gp130 : IL6 : IL6R}
{XOut | empty})[akt1 = 1],[hck = 0],[il6r = 1]
```

Esto significa que de todos los términos alcanzables por reescritura a partir del término inicial `SmallDish` y los atributos definidos, la estrategia `undefZero` nos escoge el mejor de ellos. Fruto de esta reescritura, se ha activado la proteina `Akt1`.

En este caso, para cada uno de los dos términos alcanzables, se construye la fila correspondiente de la matriz con los valores de los atributos en cada término. Las columnas corresponden al valor de cada uno de los atributos. De esta forma, tenemos la siguiente tabla/matriz:

```
1   1   0
1   *   *
```

A partir de esta matriz, la estrategia `undefZero` convierte los valores `*` a ceros y suma todos los elementos de la fila para construir su *choice value*. Después, nuestro comando en PLSS elegirá el término con un mayor *choice value*. En este caso, la primera fila o término tiene un *choice value* 2, y la segunda fila un *choice value* 1. Por tanto, se elige el primer término.

Por otra parte, si ejecutamos ahora el mismo `SoftDish`, pero con la estrategia `undefOne`, es decir:

```
loadstrat undefOne
softrew (SmallDish  ([il6r = 1], [hck = 0], [akt1 = *]))
```

obtenemos la misma lista de términos alcanzables y la misma matriz anterior. Sin embargo, la estrategia `undefOne` convierte los valores `*` a unos y suma todos los elementos de la fila para construir su *choice value*. Por lo tanto, en este caso, la primera fila o término tiene un *choice value* 2, y la segunda fila un *choice value* 3. Por tanto, se elige el segundo término:

```
Result softrew: PD(
{CLc |[Akt1 - act][Hck - act]}
{CLm | empty}
{GP130C | Gp130 : IL6 : IL6R}
{XOut | empty})[akt1 = *],[hck = *],[il6r = 1]
```

Este resultado tiene también activada la proteína `Akt1` en el citoplasma, pero en este plato se activa también la proteína `Hck`.

A continuación, vamos a analizar el comportamiento de las demás funciones de elección: `undefSemi`, `undefWRow` y `undefWCol`. De nuevo, estas funciones de elección nos permiten escoger el mejor término dentro de la lista de todos los términos alcanzables. La elección también se basa en tomar el término con el mayor *choice value*. La diferencia para cada estrategia está en cómo se calcula estos valores de elección.

En la siguiente tabla vamos a considerar una reescritura en la que existen cinco términos alcanzables para cuatro atributos:

```
1 0 0 1
1 1 * *
1 0 * *
1 0 1 0
1 * * *
```

Para esta matriz, calculamos a continuación los *choice values* para todos los términos y con cada una de las estrategias:

```
             cv0  cv1  cvS  cvR  cvC
1 0 0 1  |   2    2    2    2    2
1 1 * *  |   2    4    3    2,75 3
1 0 * *  |   1    3    2    1,75 2
1 0 1 0  |   2    2    2    2    2
1 * * *  |   1    4    2,5  1,75 2,25
```

donde `cv0` es el *choice value* que corresponde a la estrategia `undefZero` y `cv1`, `cvS`, `cvR` y `cvC` corresponden respectivamente a las estrategias `undefOne`,`undefSemi`, `undefWRow` y `undefWCol`.

La estrategia `undefSemi` asigna 1/2 a cada asterisco. De esta forma, en el segundo término, que tiene dos asteriscos, su *choice value* será $2 + 2 \cdot (1/2) = 3$. Es decir, el primer sumando corresponde a los unos del primer término y el segundo sumando indica que hay dos asteriscos. Como este es el mayor *choice value* de todos los términos, el comando devuelve este término como resultado.

La estrategia `undefWRow` asigna un valor a cada asterisco de un término analizando los valores de esa fila. Para todas las posibles combinaciones de sustituciones de ceros y unos en el valor desconocido, se calcula el valor esperado para el grupo de elementos desconocidos de la fila de acuerdo con su distribución en la fila de la matriz. De esta forma, en el tercer término, que tiene dos asteriscos, su *choice value* será $1 + (2 \cdot 2 + 1 \cdot 1)/4 = 1,75$. Es decir, el primer sumando dentro del paréntesis corresponde a los unos del primer término y el segundo sumando indica que hay dos asteriscos.

La estrategia `undefWCol` asigna un valor a cada asterisco de un término analizando los valores de su columna. En el segundo término hay dos asteriscos. Para cada asterisco, se calcula $\frac{N_1}{N_1+N_0}$, donde N_0 y N_1 son respectivamente el número de ceros y unos en la columna, siempre que $N_1 + N_0 \neq 0$, es decir, que todos los elementos de la columna no sean desconocidos. En caso contrario, se asigna el valor 1/2. Por tanto, el *choice value* de este término será $2 + 2 \cdot (1/2) = 3$, ya que en cada columna de los asteriscos existe un cero y un uno. Como este es el mayor *choice value* de todos los términos, el comando devuelve este término como resultado con la estrategia `undefWCol`.

4.2. Análisis y comparación con otros sistemas de búsqueda

En esta sección se analizan y se muestran las diferencias entre los casos del comando de búsqueda creado para este programa. En concreto, hacemos referencia a `softnarrowsearch`. Se ejecutarán varios ejemplos, comentado y analizando en detalle la salida mostrada.

El comando `softnarrowsearch` se encarga de realizar la búsqueda con narrowing o estrechamiento. Recibe dos bubbles como entrada, que deberán corresponderse con términos

del tipo `SoftDish`. El primero argumento es el término inicial y el segundo argumento es el patrón a alcanzar. Cuando exista solución, el comando devuelve el término encontrado con las sustituciones que deriven de las posibles variables que aparezcan en los términos.

```
softnarrowsearch ( Término-SoftDish-inicial ) ( Término-SoftDish-patrón )
```

En las siguientes subsecciones se van a comentar algunos ejemplos. En el primer caso se muestra un ejemplo con error en los términos. En el segundo caso aparece un ejemplo de búsqueda sin solución. Al final se ejecuta un caso con solución.

4.2.1. Caso 1: error en el análisis de los términos

En primer lugar, se ejecuta el siguiente comando y se muestra la salida.

```
PLSS> softnarrowsearch (TinyDish6)
            (TinyDish1 ([cdc6 = 0]))

Error en los términos parseados para MetaNarrowingSearch.
```

En este ejemplo, el comando `softnarrowsearch` llegar desde primer término (`TinyDish6`) a una solución que se ajuste al patrón (`TinyDish1 ([cdc6 = 0])`). En primer lugar, PLSS comprueba que los tipos de los dos argumentos correspondan con el tipo `SoftDish`. Los dishes `Tgfb1Dish`, `TinyDish`, `TinyDish1` y `SmallDish` se declaran en el fichero `Tgfb1Dish.maude`).

Sin embargo, el primer término `TinyDish6` es del tipo `Dish`, no es del tipo `SoftDish`. Por lo tanto, el comando `softnarrowsearch` nos devuelve un mensaje de error, provocado por el término inicial.

4.2.2. Caso 2: sin solución

En este ejemplo se ejecuta el comando que aparece a continuación y se muestra la salida.

```
PLSS> softnarrowsearch (TinyDish ([cdc6 = *]))
            (TinyDish1 ([cdc6 = 1]))

No solución en la búsqueda con MetaNarrowingSearch.
```

De acuerdo con el comentario de la subsección anterior, el comando `softnarrowsearch` intentará encontrar al menos una solución a partir del término inicial (`TinyDish ([cdc6 = *])`) que se ajuste al patrón deseado (`TinyDish1 ([cdc6 = 1])`).

A continuación, se muestran los dishes utilizados (`TinyDish` y `TinyDish1`) correspondientes a cada término.

```
eq TinyDish = PD(
 {XOut     | Tgfb1}
 {Tgfb1RC | TgfbR1 TgfbR2}
 {CLc     | Erks}) .

eq TinyDish1 = PD(
 {XOut   | Tgfb1}
 {Tgfb1RC | TgfbR1 TgfbR2}) .
```

Lo primero que comprueba el comando `softnarrowsearch` es que los dos términos de los argumentos se correspondan con el tipo `SoftDish`. Ambos dishes existen y tienen atributos, por lo que sí son del tipo esperado. Ahora se hará un matching del tipo de `NarrowingSearchResult` con el tipo del resultado de la función `metaNarrowingSearch` con los metatérminos iniciales recibidos. Pero en este caso, como no existe solución, el tipo que devuelve esta función no se corresponde con el buscado como nos muestra el mensaje. Por lo tanto, el comando nos devuelve un mensaje de que no hay solución en la búsqueda.

4.2.3. Caso 3: ejemplo del comando `softnarrowsearch` con solución

En primer lugar mostramos el comando para este caso y los argumentos que hemos empleado:

```
PLSS> softnarrowsearch (SD:SoftDish)
            (PD({ XOut | Tgfb1 }) ([cdc6 = 1]))
```

Este ejemplo, deseamos saber si existe un `SoftDish` con el que sea posible llegar a otro `SoftDish` cuyo plato contenga la proteína `Tgfb1` en el exterior y que además el atributo `cdc6` tenga un valor 1.

En este caso encuentra una solución y nos la muestra:

```
Result (SoftDish-term): #1:SoftDish
Substitution1: SD <- #1:SoftDish
Substitution2: #1 <- PD({XOut | Tgfb1})[cdc6 = 1]
Term-ini (T): SD:SoftDish
Term-patron (T1): PD({XOut | Tgfb1})[cdc6 = 1]
```

Se analiza con detalle el contenido. La primera instrucción que se muestra es la siguiente:

```
Result (SoftDish-term): #1:SoftDish
```

En esta aparece el resultado aunque no completo, para ello es necesario utilizar las sustituciones que aparecen después. En las siguientes instrucciones se pueden observar las sustituciones que nos conducirán al `SoftDish` buscado.

```
Substitution1: SD <- #1:SoftDish
Substitution2: #1 <- PD({XOut | Tgfb1})[cdc6 = 1]
```

Si en la primera instrucción se escribe `Substitution1` y se cambia `#1` por `Substitution2`, se obtiene el término deseado:

```
[ Solución inicial ]  -> Result (SoftDish-term): #1:SoftDish
[ Substitution1 ]   -> Result (SoftDish-term): (SD <- #1:SoftDish)
[ Substitution2 ]   -> Result (SoftDish-term): (SD <- (#1 <- PD({XOut | Tgfb1
    })[cdc6 = 1]):SoftDish)
[ Solución final ]  -> Result (SoftDish-term): (SD:SoftDish) <- (PD({ XOut |
    Tgfb1 }) ([cdc6 = 1]))
```

Como se puede observar en la etiqueta `Solución final`, contiene la solución del término `SoftDish` con un dish que contiene un `PD( XOut | Tgfb1 )` y con un atributo `([cdc6 = 1])`.

4.3. Análisis y comparación entre comandos de reescritura/estrechamiento

En nuestro programa PLSS se han implementado varios comandos de reducción y de reescritura con reglas *soft*. En este apartado se realizan y analizan varias ejecuciones con estos comandos. En estos ejemplos se varían tanto los platos y atributos, como los valores que pueden tomar los atributos. Con esta aportación se pretende mostrar la versatilidad y utilidad del programa. Después, se realiza un análisis de las distintas salidas.

En primer lugar, recordamos el comando `softrew` de PLSS que se dedica a realizar la reescritura del término con *soft* reglas y el comando `softnarrow` que emplea el *narrowing* con unificación y reglas. Los dos comandos reciben una burbuja o *bubble* como entrada (`T`), que deberá ser un término del tipo `SoftDish`. La salida de los dos comandos será el término con el mejor *choice value* para los atributos de entre todas las reescrituras posibles.

Por tanto, la principal diferencia entre ellos es que el comando `softnarrow` realiza la simplificación mediante narrowing, es decir, utiliza el estrechamiento.

A continuación, se analizarán dos casos con distintos modelos y dishes, que utilizan una misma estrategia. En estos casos, la estrategia empleada es `undefZero`, que se carga al iniciar PLSS con la siguiente instrucción:

```
PLSS> loadstrat undefZero
Loaded strategy function ( undefZero )
```

4.3.1. Caso: modelo biológico Tgfb1

En primer lugar, se ejecutan los comandos de reducción y reescritura mencionados anteriormente y se muestra su salida. Para comenzar, se define el plato `TinyDish2` que se va a utilizar:

```
eq TinyDish2 = PD(
 {XOut | Tgfb1}
 {NUc | Cdc6-gene Cdkn2b-gene Tgfb1-gene Col3a1-gene}
 {Tgfb1RC | TgfbR1 TgfbR2}) .
```

La definición de los platos que necesitemos utilizar deben estar incluidos en el fichero `Tgfb1Dish.maude`.

Ahora continuamos con la ejecución. En la primera instrucción, con `loadmodule` cargamos el módulo a emplear (`QQ1` corresponde al modelo biológico Tgfb1 de Pathway Logic).

```
PLSS> loadmodule QQ1
PLSS> red TinyDish2 ([cdc6 = 0], [col3a1 = *], [tgfb1 = 0])
PLSS> softrew TinyDish2 ([cdc6 = 0], [col3a1 = *], [tgfb1 = 0])
PLSS> softnarrow TinyDish2 ([cdc6 = 0], [col3a1 = *], [tgfb1 = 0])
```

La salida de estos comandos será:

```
Loaded Pathway Logic module ( QQ1 )

Result: PD(
{NUc | Cdc6-gene Cdkn2b-gene Col3a1-gene Tgfb1-gene}
{Tgfb1RC | TgfbR1 TgfbR2}
```

```
{XOut | Tgfb1})
[cdc6 = 0],[col3a1 = *],[tgfb1 = 0]

Result softrew: PD(
{NUc | Cdkn2b-gene[Cdc6-gene - on][Col3a1-gene - on][Tgfb1-gene - on]}
{Tgfb1RC | Tgfb1 :[TgfbR1 - act] :[TgfbR2 - act]}
{XOut | empty})
[cdc6 = 1],[col3a1 = 1],[tgfb1 = 1]

Result softnarrow: PD(
{NUc | Cdkn2b-gene[Cdc6-gene - on][Col3a1-gene - on][Tgfb1-gene - on]}
{Tgfb1RC | Tgfb1 :[TgfbR1 - act] :[TgfbR2 - act]}
{XOut | empty})
[cdc6 = 1],[col3a1 = 1],[tgfb1 = 1]
```

En el plato `TinyDish2` aparecen una serie de localizaciones como el núcleo, `Tgfb1RC` y el exterior de la célula. En cada una de las localizaciones existen ciertos genes o proteínas, por ejemplo, en el exterior de la célula se encuentra la proteína `Tgfb1` o en el núcleo los genes `Cdc6`, `Cdkn2b`, `Tgfb1-gene` y `Col3a1`.

Hemos definido algunos atributos relacionados con las funciones para el control del crecimiento celular, la proliferación celular y la apoptosis. En estas ejecuciones se usan los siguientes: el atributo `cdc6` corresponde a un regulador imprescindible en la replicación del ADN, además es esencial en el proceso de activación en el ciclo celular; el atributo `col3a1` (colágeno de tipo III) es una proteína compuesta por tres cadenas de péptidos idénticos, cada una de las cuales se denomina cadena alfa 1 de colágeno de tipo III; y, por último, el atributo `tgfb1` (factor de crecimiento transformante beta 1) es una proteína de secreción en la célula.

En este caso, los tres comandos se han aplicado al mismo `SoftDish`, es decir, exactamente al mismo plato y con los mismos atributos, para así analizar los cambios que resultan de la ejecución de cada uno. El dish es el `TinyDish2` y los atributos son los descritos anteriormente (`cdc6`, `col3a1` y `tgfb1`). El estado inicial de los atributos es:

```
([cdc6 = 0], [col3a1 = *], [tgfb1 = 0])
```

En lo que respecta a los comandos `red`, `softrew` y `softnarrow` podemos ver que a pesar de recibir la misma entrada, el único resultado que no coincide es el `red`, ya que su función es reducir el término para la simplificación ecuacional. En este caso, si ejecutamos el comando `red`, nos devuelve el mismo término que recibió.

Lo que intenta la instrucción `softrew` es que dado un `SoftDish`, ha de encontrar el mejor término a reescribir entre todos los posibles. En este caso, con el mismo `SoftDish`, utilizando las *soft* reglas que hacen matching, nos indica los siguientes cambios:

1. En la parte del plato estos fueron los cambios:

```
{XOut | Tgfb1}
 -> {XOut | empty}

{NUc | Cdc6-gene Cdkn2b-gene Tgfb1-gene Col3a1-gene}
 -> {NUc | Cdkn2b-gene[Cdc6-gene - on][Col3a1-gene - on]
          [Tgfb1-gene - on]}

{Tgfb1RC | TgfbR1 TgfbR2}
 -> {Tgfb1RC | Tgfb1 :[TgfbR1 - act] :[TgfbR2 - act]}
```

2. En la parte de los atributos, los cambios son:

```
[tgfb1 = 0] -> [tgfb1 = 1]
[cdc6 = 0] -> [cdc6 = 1]
[col3a1 = *] -> [col3a1 = 1]
```

Las *soft* reglas que se han utilizado en la reescritura son:

- `rl[SS1724.Cdc6-gene.irt.Tgfb1]` para el gen `cdc6`, teniendo el atributo `cdc6` con valor 0.
- `rl[SS1717.Tgfb1-gene.irt.Tgfb1]` para el gen `Tgfb1`, teniendo cualquier valor en el atributo `tgfb1`.
- `rl[SS1714.Col3a1-gene.irt.Tgfb1]` para el gen `Col3a1`, teniendo también cualquier valor en el atributo `col3a1`.

Inicialmente, actúa la regla `rl[931.TgfbR1.TgfbR2.by.Tgfb1]` que requiere que exista un dish que posea la proteína `Tgfb1` en la localización `XOut` y que los dos receptores `TgfbR1` TgfbR2 (Receptor de la proteína TGF-beta 1 y 2) estén presentes en la localización `Tgfb1RC`. En esa situación, actúa la regla `rl[931.TgfbR1.TgfbR2.by.Tgfb1]`, que modifica el `XOut` transformándolo en la sopa que hubiese pero la proteína modifica a la localización `Tgfb1RC` ligando los dos receptores activados entre sí junto a la proteína `Tgfb1`.

Después, en la salida de la ejecución se observa en el núcleo que los genes (`Cdc6-gene`, `Tgfb1-gene` y `Col3a1-gene`) se han activado, excepto el gen `Cdkn2b-gene` pues no hay ningún estado en que logre activarse. Los genes activados lo hacen con la localización `Tgfb1RC` en la situación comentada con la regla anterior aplicada.

Todos los cambios que realizan las reglas están sujetos a que las condiciones del dish se cumplan. El valor de los tres atributos mutan a 1, después de que estas reglas sean aplicadas.

El comando `softnarrow` realiza una tarea similar a `softrew` excepto que lo hace mediante narrowing. Pero en este caso la solución del término `SoftDish` devuelto es la misma que para el comando anterior.

4.4. Análisis de sistemas biológicos en Pathway Logic

Según se explicó en la sección 2.6, Pathway Logic es una herramienta que permite utilizar técnicas de modelado formal para entidades y procesos biológicos basándose en la lógica de reescritura de Maude. Pathway Logic es muy eficiente para la comprensión de sistemas biológicos complejos y permite diseñar experimentos de comprobación de ciertas hipótesis sobre funciones *in vivo*.

En este proyecto se han utilizado varias bases de conocimiento formal que contienen información sobre los cambios que ocurren en las proteínas y genes dentro de una célula en respuesta a la exposición a los ligandos del receptor, sustancias químicas y otros estímulos.

En nuestro programa PLSS se han adaptado los siguientes pathways completos con sus dishes y reglas:

- **TGFB1**: factor de crecimiento transformante beta 1,
- **EGF**: factor de crecimiento epidérmico,
- **HGF**: factor de crecimiento de hepatocitos,
- **IL6**: interleucina-6,
- **NGF**: factor de crecimiento nervioso.

La tabla 4.1 muestra la información relevante de cada uno de estos pathways[1]. Para cada modelo, la tabla incluye el plato inicial, fichero de reglas y el número de reglas y datums empleados en cada uno de los modelos biológicos.

Modelo	Dish	Fichero de reglas	# Reglas	# Datos
TGFB1	Tgfb1Dish	`Tgfb1RulesSS.maude`	57	968
EGF	EgfDish	`EgfRulesSS.maude`	129	2281
HGF	HgfDish	`HgfRulesSS.maude`	48	329
IL6	IL6Dish	`IL6RulesSS.maude`	23	282
NGF	NgfDish	`NgfRulesSS.maude`	46	301

Tabla 4.1: Modelos biológicos implementados en PLSS.

Con estos modelos, se pueden observar y analizar los cambios que ocurren cuando se agregan péptidos, sustancias químicas o tensiones a las células cultivadas inicialmente.

Los platos o *dishes* que describen un estado inicial se corresponden con el estado de las células en un plato de cultivo al comienzo de un experimento, junto con un estímulo. El estado inicial de la base de conocimiento en cada uno de estos platos es similar: las células no están tratadas y están en reposo. A continuación, cada plato se obtiene agregando al estado base un ligando o estímulo y sus receptores asociados.

El plato del modelo TGFB1 se presentó en el listado 2.5. Los platos que corresponden a los pathways Egf y Ngf se presentan en los listados 4.1 y 4.2 (Talcott, 2008).

Listado 4.1: Dish `EgfDish`

```
eq EgfDish =
  PD( {XOut | Egf} {EgfRC | EgfR} {CLo | empty} {CLm | Erbb2 Pag1 Plscr1}
    {CLi | [Cdc42 - GDP] [Hras - GDP] [Kras - GDP] [Nras - GDP] [Rac1 - GDP] [
        Rala - GDP] [Ralb - GDP] [Rap1a - GDP] [Rap2b - GDP] [Rit1 - GDP] Gnai1
         Gnai3 Pld1 Pld2 Src}
    {CLc | (Mlst8 : Mtor : Raptor) [Gsk3s - act] Abl1 Akt1 Araf ArhGap5 ArhGef4
         ArhGef7 Atf2 Bmx Braf Cbl Cblb Cin85 Crk CrkL Csk Dbl Dok1 Dok2
        Eif4ebp1 EndA1 Eps8 Eps15 Erk5 Erks Fak1 Fak2 Flna Freud1 Gab1 Gab2
        Git1 Grb2 Hpk1 Ipo7 IqGap1 Jak1 Jak2 Jnks Lkb1 Mapkapk2 Mek1 Mek5 Mekk1
         Mekk2 Mekk3 Mlk3 Nckipsd P38s Pdpk1 Pi3k Pkcd Pkcz Plcg1 Plce1 Ptk6
        Pxn Raf1 RalGds RapGef1 Rasa1 RasGrp3 Rela Rictor Rin1 Rps6 Rsk1 Rsk2
        Rsk3 S6k1 Sh2d3a Sh2d3c Shc1 Shoc2 Shp2 Sin1 Smad3 Sos1 Stat1 Stat3
        Stat5a Stat5b Tnk2 Tns3 Ube2l3 Vav1 Vav2 Vav3 Ywhaz Y196 Y1122}
    {NUc | Atf1 Creb1 Elk1 Fos HistH3 Jun Msk1 Msk2 Myc} ) .
```

[1] Cada dato viene representado como el resultado de un experimento publicado en una revista médico-científica y cada regla es proporcionada por una evidencia experimental en forma de referencia.

Listado 4.2: Dish `NgfDish`

```
eq NgfDish = PD({XOut | Ngf} {NgfRC | NgfR} {TrkaC | Trka}
  {CLm | empty}   {CLi | [Hras - GDP] [Rap1a - GDP] [Rit1 - GDP]}
  {CLc | Pi3k Akt1 Aps Araf Arms Braf Crk CrkI CrkL Erk5 Erks Frs2 Gab1 Gab2
      Ikba Ikk1 Ikk2 Irak1 Jnks Matk Mek1 Mek2 Myd88 P38s Pkcd Plcg1 Pxn Raf1
      RapGef1 Rela Ripk2 Sh2b1 Shc1 Sqstm1 Traf6 Znf274}
  {NUc | Egr1-gene Fos-gene Elk1 Foxo1 Foxo3}) .
```

Por último, cabe destacar la importancia de los sistemas de búsqueda de Pathway Logic para determinar las situaciones biológicas que conducen al cáncer en cada una de las rutas biológicas. En este sentido, las causas fundamentales de cáncer son procesos biológicos como la apóptosis, la proliferación celular, la estabilidad cromosómica y la transcripción de genes. Estos procesos van asociados al comportamiento de determinadas proteínas con modificaciones, que Pathway Logic nos ayuda a encontrar.

En este ejemplo, queremos saber si desde el siguiente `SoftDish`:

```
(PD({XOut  | Hgf} {HgfRC | HgfR}) ([erks = 1]))
```

es posible llegar a otro `SoftDish` en el que en la localización `HgfRC` se encuentre esta unión ligando/receptor `([HgfR - Yphos] : Hgf)` además de una sopa que desconocemos y cuál sería el valor del atributo `erks` para este caso concreto.

Esta búsqueda la podemos realizar con la instrucción `softnarrowsearch` y la siguiente línea de comando:

```
 PLSS> softnarrowsearch (PD({XOut  | Hgf} {HgfRC | HgfR}) ([erks = 1]))
            (PD(S:Soup {HgfRC | ([HgfR - Yphos] : Hgf)}) ([erks = V:Value]))
```

Obtenemos el siguiente resultado:

```
 Result softnarrowsearch (SoftDish-term):
  PD({HgfRC | Hgf :[HgfR - Yphos]}{XOut | empty})[erks = 1]

 Substitution1: none
 Substitution2: S <- {XOut | empty}
                V <-(1) .Value

 Term-ini (T):    PD({XOut | Hgf}{HgfRC | HgfR})[erks = 1]
 Term-patron (T1):  PD(S:Soup{HgfRC |[HgfR - Yphos] : Hgf})[erks = V:Value]
```

Las únicas variables que desconocemos son `S` y `V`, es decir, la sopa y el valor del atributo `erks`. En la segunda sustitución, podemos observar que la sopa `S` toma el valor de la localización `{XOut | empty}` y el valor del atributo `erks` designado por la variable `V` es `1`.

Capítulo 5

Trabajos relacionados

"Si no eres terco, te rendirás de tus propios experimentos antes de tiempo. Y si no eres flexible, te darás golpes contra una pared y no verás una solución distinta al problema que intentas resolver"
— Jeff Bezos

Este trabajo es multidisciplinar y hay trabajos relacionados en cada una de las áreas de conocimiento: lógica de reescritura, biología computacional y *soft computing*. Dentro de la lógica de reescritura, también existen trabajos relacionados sobre el entorno de ejecución, la utilización de la reflexión, el narrowing, etc.

En Santos-Buitrago et al. (2019) se presenta una versión preliminar de integración entre Pathway Logic y *soft sets*. Esta primera versión presentaba una implementación ligada a Full Maude, una extensión de Maude escrita en el propio Maude usada para desarrollo de extensiones de sintaxis que quedó obsoleta con el lanzamiento de Maude 3. Además de adaptar la idea para hacerla funcionar en la última versión de Maude he implementado un sistema de entrada/salida interactivo fácilmente extensible usando un módulo Maude como gramática y reglas de reescritura para gestionar cada comando; comandos para carga de ficheros de texto, que pueden contener lotes de comandos; nuevos tipos de funciones de elección de *soft sets*; nuevos estrategias de búsqueda; y un conjunto de *benchmarks* para analizar el comportamiento de los comandos anteriores, que ha supuesto redefinir las reglas de los correspondientes modelos.

Es importante recordar que Maude cuenta con su propio lenguaje de estrategias (Clavel et al., 2020, Capítulo 10). Este lenguaje permite un control de la aplicación de las reglas de reescritura y se define en capa de especificación por encima de las de ecuaciones y reglas, consiguiendo así un control independiente y "limpio" del proceso de reescritura. En nuestro caso, el programa PLSS no elige las reglas de reescritura a ejecutar ni su orden, sino que escoge, de la lista de términos posibles a reescribir, el término que tiene mejores propiedades de acuerdo con la función de elección escogida, por lo que podemos considerarlo como una nueva estrategia independiente.

Por último, en lo referente a Pathway Logic y la manipulación de sus modelos biológicos con Maude, los trabajos fundamentales se deben a Carolyn Talcott. Para la realización de búsquedas y reescrituras sobre Pathway Logic, los trabajos relacionados que he seguido son de Adrián Riesco con Carolyn Talcott y otros (Riesco et al., 2020; Santos-Buitrago et al., 2017; Santos-García et al., 2016). En Santos-García et al. (2016) se generalizan las

búsquedas usando variables para "agrupar" conjuntos de resultados; en Santos-Buitrago et al. (2017) se implementó una búsqueda "hacia atrás", que dado un resultado buscaba posibles estados iniciales que pudiesen alcanzar esa solución; por último, en Riesco et al. (2020) se define una noción más débil de encaje de patrones para alcanzar más resultados.

Existen muchas otras herramientas de análisis y visualización de redes biológicas. En primer lugar hay que destacar a Pathway Logic Assistant, que es una extensión de Pathway Logic que permite la visualización de las redes de señalización y el análisis de los sistemas biológicos con ayuda gráfica. Además de las herramientas relacionadas con Pathway Logic, los sistemas de modelización y análisis más cercanos a nuestro trabajo son los sistemas de modelización basados en reglas Kappa, BioNetGen y Mød:

- La plataforma *Kappa* (Boutillier et al., 2018) incluye un conjunto de técnicas interactivas de análisis de modelos basados en reglas. De una forma similar a Pathway Logic, permite manejar de forma efectiva la complejidad de los sistemas biológicos. Utiliza los agentes de forma similar al uso del módulo `CONFIGURATION` en PLSS. Dispone de varias herramientas de modelización, simulación y análisis dentro del framework de Kappa. También existe una interfaz de usuario con herramientas de visualización interactiva para los modelos definidos.

- *BioNetGen* (Harris et al., 2016) es una aplicación de código abierto desarrollada en Perl, C++ y Python. Permite la especificación y simulación de sistemas bioquímicos basado en reglas. Entre sus modelos también incluye las redes de señalización molecular. Los modelos se escriben con su propio lenguaje BNGL (BioNetGen language). Los modelos de BNGL se pueden exportar a otros formatos, como MATLAB y el lenguaje SBML (Systems Biology Markup Language). SBML es un lenguaje estándar basado en XML utilizado para representar modelos de procesos biológicos. BioNetGen dispone de una interfaz visual llamada RuleBender, que permite verificar depuración y comparación de modelos.

- *Mød* (Andersen et al., 2016) es una aplicación para el modelado con patrones de reacciones químicas a partir de descripciones de gramáticas gráficas. El software está implementado en C++ y Python. Mød permite visualizar moléculas y reacciones químicas. Permite explorar el espacio químico basado en un sistema de reescritura de grafos.

En resumen, las ventajas que ofrece Pathway Logic frente a las otras plataformas similares son:

1. la versatilidad que aporta el lenguaje subyacente Maude para el diseño y funcionalidades de la aplicación, que permite trabajar con modelos muy diferentes;

2. la potencia de las herramientas formales desarrolladas por Maude: verificación, unificación, model checking, etc.

Otras aplicaciones de análisis de redes biológicas están basados en sistemas dinámicos numéricos que se resuelven a través de ecuaciones diferenciales. Dentro de este tipo de soluciones, destaco:

- *Bio-SPICE* (Biological Simulation Program for Intra- and Inter-Cellular Evaluation) (Carvalho et al., 2018) permite la simulación de procesos en células vivas. El cuadro de mando de Bio-SPICE se basa en la plataforma NetBeans y permite que las distintas fuentes de datos de datos, modelos y motores de simulación de los distintos grupos de investigación se puedan combinar y ejecutar de forma distribuida. SRI International participa en el desarrollo de varias de sus herramientas. Bio-SPICE utiliza SBML como lenguaje de intercambio entre las diferentes herramientas. Dispone de varias herramientas de simulación, como los simuladores de ecuaciones diferenciales ordinarias continuas o simuladores estocásticos.

- *BioUML* (Kolpakov, 2002) es una aplicación de código abierto desarrollada en Java. Dispone de herramientas de visualización, simulación y análisis. Los datos experimentales puede obtenerlos a partir del acceso a bases de datos celulares. BioUML permite el ajuste de parámetros y soporta muchas otras técnicas de análisis de *big data* necesarias para tratar grandes cantidades de datos en bruto.

Capítulo 6

Conclusiones y trabajo futuro

"Comience por el principio", dijo el rey con gran gravedad,
"y continúe hasta que llegue al final: entonces deténgase"
— Lewis Carroll, Alicia en el País de las Maravillas

Por último, en este capítulo se presentan las conclusiones de este proyecto. Se incluyen también las dificultades encontradas en el desarrollo del trabajo. En la segunda sección se enumeran las posibles mejoras del proyecto para un trabajo futuro.

6.1. Conclusiones del trabajo

Una vez cerrado el proyecto, es el momento de presentar las conclusiones. En primer lugar, en base a las pruebas realizadas sobre el programa final, se ha conseguido alcanzar de forma satisfactoria el objetivo esencial que se fijó en las fases de planificación y especificación del proyecto. El objetivo fundamental consiste en desarrollar e implementar nuevas variantes de reescrituras basadas en *soft sets* en el contexto de Pathway Logic.

Los objetivos intermedios del proyecto se pueden concretar en: (1) implementación de una aplicación en Maude con su sistema de comandos propios; (2) integración con Pathway Logic; y (3) definición de estrategias de reescritura para guiar la reescritura.

Cada uno de ellos ha planteado los retos de investigar en ese área concreta e integrar esos conocimientos dentro del lenguaje Maude. Durante la ejecución del proyecto, son de destacar el esfuerzo en el estudio del metanivel y de la modelización de sistemas biológicos basados en reglas de Pathway Logic.

Por otra parte, concretando los objetivos parciales, la aplicación PLSS permite ejecutar dentro de Maude sus propios comandos (por ejemplo, simplificación, reescritura, etc.). Esta es una funcionalidad novedosa de las últimas versiones de Maude mediante su metaintérprete. En cuanto a comandos originales, se han implementado comandos de búsqueda que utilizan nuevas estrategias de *soft sets* para guiar la búsqueda.

Además se han adaptado el modelo de la ruta de señalización celular TGFB1 (factor de crecimiento transformante beta 1) y otros modelos desarrollados en Pathway Logic. La integración de su estructura y su base de conocimiento ha sido posible gracias a que el diseño de Pathway Logic está basado en el lenguaje Maude.

Por último, se han validado los resultados haciendo un análisis comparativo con otros sistemas de reescritura. Se muestran las diferencias entre las distintas funciones de estrategia dependiendo del valor de los atributos.

También se ha analizado el rendimiento del programa en lo referente al tiempo de ejecución. Se observa que todas las funciones, excepto `undefWCol`, se ejecutan en un tiempo similar a la reescritura estándar de Maude.

Las principales dificultades encontradas en el desarrollo del proyecto han sido: (1) el aprendizaje de las nuevas tecnologías implicadas en el proyecto, especialmente el manejo de los *soft sets* y Pathway Logic; y (2) la dificultad conceptual de la reflexión en Maude y del manejo del metaintérprete.

El código elaborado para este proyecto está disponible en el repositorio de GitHub `https://github.com/rsantosb/TFM-PLSS`, bajo la licencia MIT (`https://opensource.org/licenses/MIT`).

6.2. Líneas de Trabajo Futuro

Durante la elaboración del trabajo han aparecido numerosos puntos de mejora e ideas que sería interesante realizar en un futuro. Algunas de esas líneas de trabajo futuro son:

- Implementar nuevas funciones de estrategia para la toma de decisión, que se pueden ajustar mejor a otros tipos de problemas.
- Incorporar un lenguaje de especificaciones que permita a los usuarios del programa el desarrollo de funciones propias.
- Trabajar con otras extensiones de *fuzzy sets* o *soft sets*.
- Desarrollar nuevos comandos y funcionalidades en PLSS (por ejemplo, algunos comandos que mejoren la visualización de los resultados y ayuden a su mejor entendimiento).
- Investigar sobre los atributos que puedan ser significativos en la dinámica de la modelización celular.

Chapter 7

Introduction

"Why, sometimes I've believed as many as
six impossible things before breakfast"
— Lewis Carroll, Alice in Wonderland

This chapter begins with a brief introduction to the topic of this project. It is followed by a section on the objectives and work plan. Finally, the organization of the dissertation is described.

7.1. Motivation

Many real-life problems require the use of inaccurate or unknown data. Their analysis must involve the application of mathematical principles capable of capturing these characteristics. In the field of Artificial Intelligence, the theory of *soft sets* provides an appropriate framework for decision making in situations of lack of information.

The Maude language allows specifying the static part of a system by means of equational logic and the dynamic part by means of rewrite rules. The specifications are executable and Maude has a breadth-first search command. However, this standard Maude search runs through all reachable terms in the search tree, without allowing advanced strategies, so it is not suitable for some applications where it is desirable to choose to prioritize one branch among all possible ones. Based on this situation, this project proposes to implement an execution environment with different strategies in the rewriting system that allow to choose the best rewriting path according to certain criteria. Moreover, taking into account that the available information may be incomplete, the strategies have been defined using the theory of *soft sets*.

On the other hand, Maude is a specification language that allows a wide range of applications to be defined and expressed in a natural way. Pathway Logic is a tool based on Maude and designed to deal with symbolic biological systems. A symbolic biological system is a network of elements in which their relationships are defined abstractly. Thus, formal methods based on rewrite logic are appropriate for dealing with and analyzing the behavior of these systems.

Numerous models of cell signaling pathways have been developed with Pathway Logic. In short, a cell signaling pathway is a biological system in which we define the parts or locations that exist in a cell (e.g., cell membrane, cytoplasm, nucleus, etc.) and the

molecular elements that can occur in the different parts of the cell (e.g., proteins, genes, etc.). In a signaling pathway we also define the dynamic behavior or interactions between the elements of the cell (e.g., binding between a ligand and a receptor, activation of a protein, etc.). Pathway Logic facilitates the understanding of complex biological systems and the verification of hypotheses in the design of experiments.

With the intention of providing an application with real utility, the implementation of the proposed rewrite system has been developed with Pathway Logic's biological models. In this way, its decision making system based on *soft sets* allows us to choose among all the possible dynamic changes, those that are more probable, even when not all the cellular information is available. This search system has been used with some Pathway Logic biological models, for example with the epithelial growth factor cell signaling pathway, to determine the existence of modifications of some proteins associated with cancer cells.

In the development of this system, the new functionalities of the new version of Maude have been used with the meta-interpreter and the input/output management, which allow the efficient handling of an execution environment within Maude.

7.2. Objectives

The main objective of this project is to develop and implement variants of rewrites and directed searches based on *soft sets* in the context of Pathway Logic. In order to handle the changes that occur in biological systems and the need for interaction with external objects, the implementation is carried out with the Maude 3.1 language, taking advantage of the new functionalities of this version.

The aim is to define directed rewritings that allow us to choose the paths of reachable terms from the paths of the reachable terms from strategies based on *soft sets*. These strategies are used even in the situations of incomplete information of some of the attributes. The attributes in the terms and in the rewrite rules allow us to define the criteria for choosing the best term, which in our case of biological signaling paths, will be the most probable final term or pattern.

Narrowing is an alternative to rewriting that works by exchanging pattern matching for unification. Search commands have been implemented with both rewriting and narrowing techniques, where *soft sets* are used to direct both normal and symbolic searches.

Intermediate goals for achieving this objective can be specified at:

1. Implement an input/output application in Maude with its own command system.
2. Integrate the application with Pathway Logic models.
3. Define strategies of *soft sets* to guide rewriting.
4. Extend the examples to include *soft sets* and analyze how different search strategies work.

7.3. Work plan

The realization of the project has been supported by the work plan with the director. The first meetings were dedicated to the concretion of the theme and objectives of the

project. Afterwards, every two weeks we had meetings in which the tasks defined in the previous meeting were reviewed. On the part of the director, corrections were made and improvements proposed. In each meeting, the work to be done during the following two weeks was specified.

Apart from the regular meetings, occasional doubts about any issue of the work have been resolved by the director by e-mail.

The milestones that were established to achieve the objectives are:

1. Search and initial definition of the subject of the work. This task was extended from the end of September 2019 to mid-October 2019.

2. Going deeper into the Maude language. For the next two months, I dedicated myself to studying Maude's topics needed for this project in depth.

3. Research on Pathway Logic and *soft sets*. This research was extended until March 2020.

4. Design and implementation of a basic prototype of the application. The preliminary version was operational at the end of July 2020.

5. Progressive extension of the functionalities (both in the commands and in the strategies). I dedicated myself to this task during the last quarter of 2020.

6. Testing. In January and February 2021, I carried out the testing of the program.

7. Drafting of the project report. Intensively, I started writing this report in March of this year.

7.4. Organization of the dissertation

The dissertation is divided into the following chapters:

- **Chapter 1**: Introduction. The theme of the TFM is introduced, the objectives of the work are described, and the work plan followed to achieve the objectives is detailed.

- **Chapter 2**: State-of-the-art. This chapter introduces the areas on which the work is based: the Maude programming language, the Maude meta-interpreter, *soft sets* theory, and Pathway Logic.

- **Chapter 3**: Description of the work. This chapter describes the work done in the project **PLSS**: **P**athway **L**ogic with **S**oft **S**ets.

- **Chapter 4**: Results Analysis. This chapter includes a comparative analysis of the results obtained with those of the standard use of rewriting.

- **Chapter 5**: Related work. This chapter discusses the works related to the project.

- **Chapter 6**: Conclusions and Future Work. Finally, the conclusions and lines for future work have been set out.

The code developed for this project is available in the GitHub repository `https://github.com/rsantosb/TFM-PLSS`, under the MIT license (`https://opensource.org/licenses/MIT`).

Chapter 8

Conclusions and future work

"Begin at the beginning," the King said, very gravely,
"and go on till you come to the end: then stop"
— Lewis Carroll, Alice in Wonderland

This chapter presents the conclusions of this project. It also includes the difficulties encountered in the development of the work. The second section lists possible improvements to the project for future work.

8.1. Conclusions of the work

Once the project is closed, it is time to present the conclusions.

Firstly, on the basis of the tests carried out on the final programme, the essential objective set in the planning and specification phases of the project has been successfully achieved. The fundamental objective is to develop and implement new variants of rewrites based on *soft sets* in the context of Pathway Logic.

The intermediate objectives of the project can be specified as: (1) implementation of an application in Maude with its own command system; (2) integration with Pathway Logic; and (3) definition of rewriting strategies to guide the rewrite.

Each of them has posed the challenges of researching that particular area and integrating that knowledge into the Maude language. During the execution of the project, the effort in the study of the meta-level and the modelling of biological systems based on Pathway Logic rules is noteworthy.

Moreover, by specifying the partial objectives, the PLSS application allows you to execute your own commands within Maude (e.g. simplification, rewriting, etc.). This is a new feature of the latest versions of Maude through its meta-interpreter.

In addition, the TGFB1 (transforming growth factor beta 1) cell signaling pathway model and other models developed in Pathway Logic have been adapted. The integration of its structure and knowledge base has been possible thanks to the fact that the design of Pathway Logic is based on the Maude language.

The definition of new rewriting strategies to guide rewriting has been based on the proposals in the recent Santos-Buitrago et al. (2019). In fact, those described in that

article have also been implemented in PLSS (`undefZero`, `undefOne` and `undefSemi`).

Finally, the results have been validated by making a comparative analysis with other rewriting systems. The differences between the various strategy functions are shown depending on the value of the attributes.

The performance of the programme in terms of execution time has also been analysed. It was found that all functions, except `undefWCol`, run at a similar time to Maude's standard rewrite.

The main difficulties encountered in the development of the project have been: (1) the learning of the new technologies involved in the project, especially the handling of the *soft sets* and Pathway Logic; and (2) the conceptual difficulty of reflecting on Maude and handling the meta-interpreter.

The code developed for this project is available in the GitHub repository `https://github.com/rsantosb/TFM-PLSS`, under the MIT license (`https://opensource.org/licenses/MIT`).

8.2. Lines of future work

During the elaboration of the work, numerous points of improvement and ideas have appeared that would be interesting to carry out in the future. Some of these lines of future work are:

- Implement new strategy functions for decision making, which can be better adjusted to other types of problems.
- Incorporate a specification language that allows program users to develop their own functions.
- Work with other fuzzy sets or *soft sets* extensions.
- Develop new commands and functionalities in PLSS (e.g., some commands to improve the visualization of the results and help to better understand them).
- Research on the attributes that can be significant in the dynamics of cell modelling.

Bibliografía

Cuanto menos se lee,
más daño hace lo que se lee.

Miguel de Unamuno

Aktaş, H. y Çağman, N. Soft sets and soft groups. *Information Sciences*, vol. 177, páginas 2726–2735, 2007.

Alcantud, J. C. R. A novel algorithm for fuzzy soft set based decision making from multiobserver input parameter data set. *Information Fusion*, vol. 29, páginas 142–148, 2016a.

Alcantud, J. C. R. Some formal relationships among soft sets, fuzzy sets, and their extensions. *International Journal of Approximate Reasoning*, vol. 68, páginas 45–53, 2016b.

Alcantud, J. C. R. y Santos-García, G. A new criterion for soft set based decision making problems under incomplete information. *International Journal of Computational Intelligence Systems*, vol. 10, páginas 394–404, 2017.

Ali, M., Feng, F., Liu, X., Min, W. K. y Shabir, M. On some new operations in soft set theory. *Computers & Mathematics with Applications*, vol. 57(9), páginas 1547–1553, 2009.

Ali, M., Mahmood, T., Rehman, M. M. U. y Aslam, M. F. On lattice ordered soft sets. *Applied Soft Computing*, vol. 36, páginas 499–505, 2015.

Andersen, J. L., Flamm, C., Merkle, D. y Stadler, P. F. A software package for chemically inspired graph transformation. En *International Conference on Graph Transformation*, páginas 73–88. Springer, 2016.

Bernardo, M., Degano, P. y Zavattaro, G., editores. *Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008, Bertinoro, Italy, June 2-7, 2008, Advanced Lectures*, vol. 5016 de *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-68892-1.

Boutillier, P., Maasha, M., Li, X., Medina-Abarca, H. F., Krivine, J., Feret, J., Cristescu, I., Forbes, A. G. y Fontana, W. The Kappa platform for rule-based modeling. *Bioinformatics*, vol. 34(13), páginas i583–i592, 2018.

Carvalho, R. V., Verbeek, F. J. y Coelho, C. J. Bio-modeling using Petri nets: A computational approach. En *Theoretical and Applied Aspects of Systems Biology*, páginas 3–26. Springer, 2018.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R. y Talcott, C. L. *Maude manual (version 3.1)*, 2020. `http://maude.cs.illinois.edu/w/images/6/62/Maude-3.1-manual.pdf`.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J. y Sönmez, M. K. Pathway Logic: Symbolic analysis of biological signaling. En *Proceedings of the 7th Pacific Symposium on Biocomputing, PSB 2002, Lihue, Hawaii, USA, January 3-7, 2002* (editado por R. B. Altman, A. K. Dunker, L. Hunter y T. E. Klein), páginas 400–412. 2002.

Eker, S., Laderoute, K., Lincoln, P., Sriram, M. G. y Talcott, C. L. Representing and simulating protein functional domains in signal transduction using Maude. En *Computational Methods in Systems Biology, First International Workshop, CMSB 2003, Roverto, Italy, February 24-26, 2003, Proceedings* (editado por C. Priami), vol. 2602 de *Lecture Notes in Computer Science*, páginas 164–165. Springer, 2003. ISBN 3-540-00605-2.

Escobar, S., Meseguer, J. y Thati, P. Natural narrowing for general term rewriting systems. En *International Conference on Rewriting Techniques and Applications. RTA 2005* (editado por J. Giesl), vol. 3467 de *Lecture Notes in Computer Science*, páginas 279–293. Springer, 2005. ISBN 978-3-540-25596-3.

Feng, F. y Li, Y. Soft subsets and soft product operations. *Information Sciences*, vol. 232, páginas 44–57, 2013.

Han, B.-H., Li, Y., Liu, J., Geng, S. y Li, H. Elicitation criterions for restricted intersection of two incomplete soft sets. *Knowledge-Based Systems*, vol. 59, páginas 121–131, 2014.

Harris, L. A., Hogg, J. S., Tapia, J.-J., Sekar, J. A., Gupta, S., Korsunsky, I., Arora, A., Barua, D., Sheehan, R. P. y Faeder, J. R. BioNetGen 2.2: Advances in rule-based modeling. *Bioinformatics*, vol. 32(21), páginas 3366–3368, 2016.

Knapp, M., Briesemeister, L., Eker, S., Lincoln, P., Poggio, A., Talcott, C. L. y Laderoute, K. Pathway Logic helping biologists understand and organize pathway information. En *Fourth International IEEE Computer Society Computational Systems Bioinformatics Conference Workshops & Poster Abstracts (CSB 2005 Workshops), Stanford, California, USA, 8-11 August, 2005*, páginas 155–156. IEEE Computer Society, 2005. ISBN 0-7695-2442-7.

Kolpakov, F. A. BioUML-Framework for visual modeling and simulation of biological systems. En *Proceedings of the International Conference on Bioinformatics of Genome Regulation and Structure*, vol. 130, página 133. 2002.

Maji, P., Biswas, R. y Roy, A. Fuzzy soft sets. *Journal of Fuzzy Mathematics*, vol. 9, páginas 589–602, 2001.

Maji, P., Biswas, R. y Roy, A. An application of soft sets in a decision making problem. *Computers and Mathematics with Applications*, vol. 44, páginas 1077–1083, 2002.

MAJI, P., BISWAS, R. y ROY, A. Soft set theory. *Computers and Mathematics with Applications*, vol. 45, páginas 555–562, 2003.

MOLODTSOV, D. Soft set theory - first results. *Computers and Mathematics with Applications*, vol. 37, páginas 19–31, 1999.

NAKAO, A., AFRAKHTE, M., MORN, A., NAKAYAMA, T., CHRISTIAN, J. L., HEUCHEL, R., ITOH, S., KAWABATA, M., HELDIN, N.-E., HELDIN, C.-H. ET AL. Identification of Smad7, a TGFβ-inducible antagonist of TGF-β signalling. *Nature*, vol. 389(6651), páginas 631–635, 1997.

QIN, H., MA, X., HERAWAN, T. y ZAIN, J. Data filling approach of soft sets under incomplete information. En *Intelligent Information and Database Systems* (editado por N. Nguyen, C.-G. Kim y A. Janiak), vol. 6592 de *Lecture Notes in Computer Science*, páginas 302–311. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20041-0.

QIN, K., MENG, D., PEI, Z. y XU, Y. Combination of interval set and soft set. *International Journal of Computational Intelligence Systems*, vol. 6(2), páginas 370–380, 2013.

RIESCO, A., SANTOS-BUITRAGO, B., KNAPP, M., SANTOS-GARCÍA, G., GALILEA, E. H. y TALCOTT, C. L. Fuzzy matching for cellular signaling networks in a choroidal melanoma model. En *Practical Applications of Computational Biology & Bioinformatics, 14th International Conference (PACBB 2020), LÁquila, Italy, 17-19 June, 2020* (editado por G. Panuccio, M. Rocha, F. Fdez-Riverola, M. S. Mohamad y R. Casado-Vara), vol. 1240 de *Advances in Intelligent Systems and Computing*, páginas 80–90. Springer, 2020.

SANTOS-BUITRAGO, B., RIESCO, A., KNAPP, M., ALCANTUD, J. C. R., SANTOS-GARCÍA, G. y TALCOTT, C. L. Soft set theory for decision making in computational biology under incomplete information. *IEEE Access*, vol. 7, páginas 18183–18193, 2019.

SANTOS-BUITRAGO, B., RIESCO, A., KNAPP, M., SANTOS-GARCÍA, G. y TALCOTT, C. L. Reverse inference in symbolic systems biology. En *11th International Conference on Practical Applications of Computational Biology & Bioinformatics, PACBB 2017, Porto, Portugal, 21-23 June, 2017* (editado por F. Fdez-Riverola, M. S. Mohamad, M. P. Rocha, J. F. D. Paz y T. Pinto), vol. 616 de *Advances in Intelligent Systems and Computing*, páginas 101–109. Springer, 2017. ISBN 978-3-319-60815-0.

SANTOS-GARCÍA, G., TALCOTT, C. L., RIESCO, A., SANTOS-BUITRAGO, B. y RIVAS, J. D. L. Role of nerve growth factor signaling in cancer cell proliferation and survival using a reachability analysis approach. En *10th International Conference on Practical Applications of Computational Biology & Bioinformatics, PACBB 2016, Sevilla, Spain, 1–3 June 2016* (editado por M. S. Mohamad, M. P. Rocha, F. Fdez-Riverola, F. J. D. Mayo y J. F. D. Paz), vol. 477 de *Advances in Intelligent Systems and Computing*, páginas 173–181. Springer, 2016.

TALCOTT, C. L. Symbolic modeling of signal transduction in Pathway Logic. En *Proceedings of the Winter Simulation Conference WSC 2006, Monterey, California, USA, December 3-6, 2006* (editado por L. F. Perrone, B. Lawson, J. Liu y F. P. Wieland), páginas 1656–1665. WSC, 2006. ISBN 1-4244-0501-7.

TALCOTT, C. L. Pathway Logic. En *Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008, Bertinoro, Italy, June 2-7, 2008, Advanced*

Lectures (editado por M. Bernardo, P. Degano y G. Zavattaro), vol. 5016 de *Lecture Notes in Computer Science*, páginas 21–53. Springer, 2008. ISBN 978-3-540-68892-1.

Talcott, C. L. The Pathway Logic formal modeling system: Diverse views of a formal representation of signal transduction. En *IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2016, Shenzhen, China, December 15-18, 2016* (editado por T. Tian, Q. Jiang, Y. Liu, K. Burrage, J. Song, Y. Wang, X. Hu, S. Morishita, Q. Zhu y G. Wang), páginas 1468–1476. IEEE Computer Society, 2016. ISBN 978-1-5090-1611-2.

Talcott, C. L. y Dill, D. L. The Pathway Logic Assistant. En *Proceedings of the Third International Workshop on Computational Methods in Systems Biology, CMSB'05, Edinburgh, Scotland, April 2005* (editado por G. Plotkin), vol. 3, páginas 228–239. 2005.

Talcott, C. L. y Dill, D. L. Multiple representations of biological processes. En *Transactions on Computational Systems Biology* (editado por C. Priami y G. D. Plotkin), vol. 4220 de *Lecture Notes in Computer Science*, páginas 221–245. Springer, 2006. ISBN 3-540-45779-8.

Talcott, C. L., Eker, S., Knapp, M., Lincoln, P. y Laderoute, K. Pathway Logic modeling of protein functional domains in signal transduction. En *Proceedings of the 2nd IEEE Computer Society Bioinformatics Conference, CSB 2003, Stanford, California, August 11-14, 2003* (editado por P. Markstein y Y. Xu), páginas 618–619. IEEE Computer Society, 2003. ISBN 0-7695-2000-6.

Talcott, C. L. y Knapp, M. Explaining response to drugs using Pathway Logic. En *Computational Methods in Systems Biology - 15th International Conference, CMSB 2017, Darmstadt, Germany, September 27-29, 2017, Proceedings* (editado por J. Feret y H. Koeppl), vol. 10545 de *Lecture Notes in Computer Science*, páginas 249–264. Springer, 2017. ISBN 978-3-319-67471-1.

Wang, F., Li, X. y Chen, X. Hesitant fuzzy soft set and its applications in multicriteria decision making. *Journal of Applied Mathematics*, página Article ID 643785, 2014.

Zadeh, L. Fuzzy sets. *Information and Control*, vol. 8, páginas 338–353, 1965.

Zhang, X. On interval soft sets with applications. *International Journal of Computational Intelligence Systems*, vol. 7(1), páginas 186–196, 2014.

Zou, Y. y Xiao, Z. Data analysis approaches of soft sets under incomplete information. *Knowledge-Based Systems*, vol. 21(8), páginas 941–945, 2008.