Universidad Autónoma de Madrid

Escuela Politécnica Superior

TRABAJO FIN DE MÁSTER

Motor algebraico extensible para Chalkpy

Máster Universitario en Ingeniería Informática

Autor: Salcedo Valderrama, Laura

Tutor: Lago Fernández, Luis Fernando
Departamento de Ingeniería Informática

Fecha: Septiembre, 2018

MOTOR ALGEBRAICO EXTENSIBLE PARA CHALKPY

Autor: Salcedo Valderrama, Laura
Tutor: Lago Fernández, Luis Fernando

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Septiembre, 2018

¿Importa el destino? ¿O es el camino que emprendemos? Declaro que ningún logro tiene tan gran sustancia como el camino empleado para conseguirlo. No somos criaturas de destinos. Es el viaje el que nos da la forma. Nuestros pies encallecidos, nuestras espaldas fortalecidas por cargar el peso de nuestros viajes, nuestros ojos abiertos con el fresco deleite de las experiencias vividas.

Brandon Sanderson.

Agradecimientos

A mi tutor Luis por volver a apostar por Chalkpy y por mí.

A Rodri por entender que el TFM tenía preferencia sobre él y no dejar que me rindiese ni en los peores días.

A Villegas por empeñarse en que usara LaTeXy por tantas conversaciones.

A Dani que aunque no tiene mano izquierda ha evitado que me salgan más canas.

Y ti, que has buscado un rato para leerlo.

Resumen

Chalkpy es una herramienta para el apoyo a la docencia que permite la manipulación de expresiones algebraicas. Cuenta con dos versiones, la primera desarrollada con una librería con características afines a los sistemas de álgebra computacional (CAS), y la segunda formado por un motor resultado de un compilador desarrollado para la creación de distintos CAS. Ambas versiones formaron parte de un Trabajo de Fin de Grado.

Este Trabajo de Fin de Máster tiene como objetivo la creación de un nuevo motor que mantenga la funcionalidad existente en las versiones anteriores de Chalkpy: definición de operadores, definición de constates simbólicas y la manipulación de expresiones algebraicas y ecuaciones. Se le introducirán dos mejoras. La primera de ellas es la posibilidad de utilizar operaciones con definiciones complejas. La segunda es permitir la extensibilidad del motor, incluso a otros ámbitos diferentes a la matemática tradicional. Esta extensión se debe poder hacer de una manera sencilla para un usuario sin grandes conocimientos en programación.

Para su desarrollo, se ha planteado un análisis inicial de requisitos, seguido de un modelo de desarrollo iterativo e incremental, con versiones funcionales del motor tras cada incremento. El lenguaje elegido ha sido Prolog, un lenguaje de programación lógico, con una arquitectura de Modularización orientada a flujos de funciones. Esto implica una división del sistema en módulos, los cuales, dados unos datos de entrada y tras una transformación, generan los datos de salida.

El proyecto ha conseguido alcanzar los objetivos marcados y superar los hitos propuestos. Se han podido generar los diferentes ficheros que aportan la funcionalidad para operar la suma, el producto, el logaritmo, la exponencial, ecuaciones de primer grado, derivadas, sumatorios e, incluso, sistemas lineales de dos ecuaciones con dos incógnitas. Además estos operadores se han podido ir implementando de forma escalonada sin necesidad de modificar la lógica genérica del motor.

Palabras Clave — Chalkpy, Sistema de Álgebra Computacional, álgebra, Prolog

Abstract

Chalkpy is a teaching support tool that allows algebraic expression manipulation. At this moment there are two versions. The first one was developed using a library with similar functionalities to those of a Computer Algebra System (CAS). The second one is based on a compiler that generates a user defined CAS. Both versions were part of Final Degree Projects.

The main objective of this Master thesis is to develop a new algebraic engine that maintains the current features of the former Chalkpy versions: definition of operators and symbolic constants, and resolution of algebraic expressions and equations by user manipulations. Two improvements will be added. The first one is to allow complex operation definitions. The second one is to allow the system extension, in areas that go beyond traditional Mathematics. These extensions must be defined in a simple way, even by users that are not proficient in Programming.

For the development of these features, an initial requirements analysis has been set, being followed by an incremental, iterative development, leading to working versions of the system after each iteration. Prolog, a logical programming language, has been selected for the development, along with a Modular Architecture based in function flow. This implies a modular division within the system, with each module generating its own output data by applying a transformation to its input.

All the initial objectives have been achieved during the development of this project. Definition files that involve different mathematical functionality have been tested, including addition, product, the logarithm and exponential functions, first-order equations, derivatives, summatories and even linear equation systems. All these operators have been implemented step-by-step, without modifying the core system's logic.

Keywords— Chalkpy, Computer Algebra System, algebra, Prolog

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 2
- 1.2. Objetivos . . . 3
- 1.3. Estructura del documento . . . 4

2. Estado del Arte **5**
- 2.1. Del álgebra tradicional al computacional . . . 5
- 2.2. Sistemas de Álgebra Computacional . . . 7
- 2.3. Herramientas CAS . . . 7

3. Definición del proyecto **11**
- 3.1. Alcance . . . 11
- 3.2. Metodología . . . 12
- 3.3. Tecnología y Herramientas . . . 13
  - 3.3.1. Tecnologías . . . 13
  - 3.3.2. Herramientas . . . 14

4. Análisis **15**
- 4.1. Requisitos Funcionales . . . 15
- 4.2. Requisitos no funcionales . . . 17

5. Diseño e Implementación **19**
- 5.1. Arquitectura del proyecto . . . 19
- 5.2. Sistema de Álgebra Computacional . . . 21
  - 5.2.1. Expresiones algebraicas . . . 21
  - 5.2.2. Ecuación . . . 22
  - 5.2.3. Selección de términos . . . 23
  - 5.2.4. Definición de operadores . . . 25
  - 5.2.5. Operador evalúa . . . 29
  - 5.2.6. Definición de constantes simbólicas . . . 30

5.2.7. Operadores y constantes simbólicas predefinidos 30
5.2.8. Elemento neutro e inverso 31
5.2.9. Valores por defecto 32
5.3. Interfaz de pruebas 32
5.4. Funcionamiento interno del CAS 33

6. Pruebas **37**
6.1. Inspección de código 37
6.2. Caja blanca 38
6.3. Caja negra 38
6.3.1. Automatización 38

7. Resultados **41**
7.1. Álgebra de Boole 42
7.2. Sumatorio 44
7.3. Sistemas de dos ecuaciones 46

8. Conclusiones y Trabajo Futuro **53**

Bibliografía **54**

Glosario **59**

Acrónimos **61**

Anexos **63**

A. Ficheros de usuarios **65**

B. Principales predicados del motor de Chalkpy **77**

Índice de figuras

5.1. Modularización del sistema y su comunicación 20
5.2. Flujo típico de las funciones del sistema 21
5.3. Ejemplo de la representación de una expresión algebraica como lista . . . 22
5.4. Ejemplo de la representación de una ecuación como lista 22
5.5. Ejemplos de expresión algebraica y ecuación con selección. 24
5.6. Resolución en la interfaz de la ecuación $2 + x = 6$ 35

6.1. Ejemplo de entrada para pruebas de caja negra 39

7.1. Resolución de una expresión perteneciente al Álgebra de Boole. 44

Índice de códigos

5.1. Definición del operador suma. 25
5.2. Definición del símbolo de la suma. 26
5.3. Definición del tipo de representación para la suma. 26
5.4. Definición de la suma como operador n-ario. 27
5.5. Definición de argumentos personalizados para el operador opr. 27
5.6. Definición del proceso de operar para la suma. 29
5.7. Definición del operador evalúa 29
5.8. Definición de opera para el operador primitivo equal. 30
5.9. Definición del número uno como constante simbólica. 30
5.10. Definición del elemento neutro del operador suma. 31
5.11. Definición del elemento inverso del operador suma. 32
7.1. Definición del fichero de usuario para el Álgebra de Boole. 42
7.2. Definición del fichero de usuario para el sumatorio 45
7.3. Definición del fichero de usuario para los sistemas lineales de dos ecuaciones. 47
A.1. Definición del fichero de usuario básico. 65
A.2. Definición del fichero de usuario para la potencia. 68
A.3. Definición del fichero de usuario para el Álgebra de Boole. 69
A.4. Definición del fichero de usuario para el logaritmo y la exponencial. 71
A.5. Definición del fichero de usuario para las derivadas. 73
A.6. Definición del fichero de usuario para el sumatorio 74
A.7. Definición del fichero de usuario para los sistemas de dos ecuaciones. 75

Índice de tablas

5.1. Predicados correspondientes a la definición de argumentos de un operador 27
5.2. Predicados para las propiedades básicas que cumple un operador. 28
5.3. Operadores y constantes simbólicas predefinidos del motor. 31
5.4. Valores por defecto para la características a definir por el usuario. 32
5.5. Mapeo para la interfaz de pruebas entre las instrucciones y el teclado. . . . 33

7.1. Resolución del sumatorio $\sum_{i=4}^{6} 2^i$. 46

B.1. Predicados principales del módulo de operación de Chalkpy. 78
B.2. Predicados principales del módulo de validación de Chalkpy. 79
B.3. Predicados principales del módulo principal de Chalkpy. 80
B.4. Predicados principales del módulo de aplicación de Chalkpy. 80
B.5. Predicados principales del módulo de operativa de Chalkpy. 81
B.6. Predicados principales del módulo de impresión de Chalkpy. 81

1

Introducción

Chalkpy es una herramienta informática de apoyo a la docencia que permite la manipulación de expresiones algebraicas sencillas. En sus inicios proveía contenidos matemáticos a través de un navegador web y permitía llevar a cabo manipulaciones sencillas y dinámicas. No obstante, la ampliación de su funcionalidad presentaba un problema difícil de resolver. Por ello se desarrolló una nueva versión, Chalkpy 2.0, conformada por un motor que permitía la manipulación de expresiones mediante la definición de reglas de equivalencia. Esto se conoce como sistema de álgebra computacional [1], o CAS de sus siglas en inglés.

Este proyecto, desarrollado como Trabajo de Fin de Máster en la Escuela Politécnica Superior de la Universidad Autónoma de Madrid, tiene como propósito el desarrollo de un motor algebraico extensible. Es decir, un motor que permita, de una forma sencilla, ampliar su funcionalidad mediante la introducción, principalmente, de nuevas operaciones.

A continuación se explica la motivación para el desarrollo de este nuevo motor y los objetivos que se pretende alcanzar, seguido de una pequeña descripción del presente documento.

1.1. Motivación

Chalkpy nace como una herramienta de apoyo a la docencia que permite la manipulación de ecuaciones de primer grado. Su creación se enmarca en un Proyecto de Innovación Docente [2] y Trabajo de Fin de Grado [3] desarrollados en la Escuela Politécnica Superior de la UAM. Para dicho proyecto se utilizó la librería SymPy [4] de Python que, debido a su enfoque de resolución automatizada de expresiones, introducía simplificaciones que podían no interesar al usuario. Además, aunque cubría los objetivos perseguidos por dicho trabajo, tras las pruebas se detectó la necesidad de ampliar su funcionalidad para conformar una ayuda más amplia a los usuarios. Esto planteaba un problema: la lógica y la interfaz se habían desarrollado específicamente para la resolución de ecuaciones lineales por lo que ampliar su lógica y generalizarlo implicaba un cambio casi completo.

En un Proyecto de Innovación Docente [5] y Trabajo de Fin de Grado [6] posterior se buscó la solución al problema de Chalkpy, desarrollar un CAS a medida. Para su creación se implementó la generación automática de motores de álgebra computacional mediante un compilador desarrollado en Java. Se concretó un lenguaje de definición de expresiones y reglas que, al ser interpretado por el compilador, generaba código capaz de manipular las expresiones acorde a las reglas definidas. Además permitía la definición de operadores mediante la modificación de diferentes parámetros relacionados con su simbología, sus argumentos y sus propiedades matemáticas básicas. Todo ello se especificaba en un fichero de definiciones. De esta forma un usuario podía crear y modificar diferentes CAS utilizando únicamente dicho fichero, sin necesidad de tener amplios conocimientos de programación. Como caso particular de los motores generados se obtuvo el que conformaría la segunda versión de Chalkpy. Este CAS permitía la resolución de ecuaciones de primer grado y su extensión a diversos operadores y operaciones.

Chalkpy 2.0 cumplía con los objetivos buscados. Solucionaba los problemas que generaba la modificación del sistema para introducir nuevas funcionalidades, cuyos límites se encontraban en las expresiones matemáticas y las equivalencias que definiese el usuario. No obstante, cuando se intentó extender a operaciones con características complejas e, incluso, usarlo en otros ámbitos diferentes al álgebra tradicional, se constató que la complejidad aumentaba de forma considerable para un usuario sin experiencia en programación. Esto limitaba su uso a expresiones y transformaciones aritméticas sencillas. Se planteaba un nuevo problema, la extensibilidad del motor algebraico en la nueva versión de Chalkpy no era óptima.

Este Trabajo de Fin de Máster persigue la mejora del actual motor algebraico de Chalkpy haciendo que resulte más sencillo a un usuario ampliar la funcionalidad de la

herramienta, sin importar la complejidad de las operaciones a definir. Además, se pretende poder utilizar el motor en contextos diferentes al álgebra tradicional, adaptándose a su posibles especificaciones sin necesidad de modificar el núcleo. Todo ello debe hacerse de forma que el usuario no requiera tener grandes conocimientos en programación para su uso y ampliación.

1.2. Objetivos

El fin de este Trabajo de Fin de Máster es diseñar e implementar un CAS que mejore la usabilidad actual de Chalkpy y permita, de una forma sencilla, extender la funcionalidad de la herramienta a nuevos ámbitos. Esto daría lugar a una nueva versión, Chalkpy 3.0. Para ello se contemplan tres grandes objetivos que debe alcanzar el proyecto:

- Poder definir expresiones matemáticas.
- Poder definir equivalencias entre expresiones que permitan su transformación en diversos ámbitos.
- Desarrollo de una interfaz de pruebas que permita la verificación del CAS de una forma cómoda y sencilla.

Para alcanzar dichos objetivos se plantean unos objetivos parciales que marcarán la fase de análisis, y darán la base para desarrollar el catálogo inicial de requisitos. Dichos objetivos parciales son:

- Desarrollar las propiedades básicas de manipulación algebraica de forma general.
- Definir y poder operar con constantes simbólicas.
- Definir un operador mediante su simbología, sus argumentos, su operativa y sus propiedades matemáticas.
- Definir las manipulaciones concretas de los operadores de forma sencilla para el usuario y de forma transparente a la lógica del motor.
- Crear un programa de testeo, mediante comandos por teclado, que permita la creación y manipulación de expresiones acorde a las operaciones definidas.

1.3. Estructura del documento

Este documento se compone de los siguientes capítulos:

- **Capítulo 2:** en el estado del arte se expone la evolución y situación actual del álgebra computacional con el fin de mostrar al lector una panorámica de su uso. También se presentan diversas aplicaciones con un fin similar al presente proyecto.
- **Capítulo 3:** detalla el alcance del proyecto. Ilustra la metodología utilizada y las tecnologías y herramientas que han sido necesarias para llevar a cabo su desarrollo.
- **Capítulo 4:** expone el análisis mediante la enumeración del catálogo de requisitos, tanto funcionales como no funcionales.
- **Capítulo 5:** abarca el diseño e implementación del motor, explicando su arquitectura y componentes.
- **Capítulo 6:** define las pruebas que se han realizado tanto en las fases de desarrollo como al finalizar estas.
- **Capítulo 7:** muestra los resultados que se han obtenido.
- **Capítulo 8:** el último capítulo enumera las conclusiones del presente trabajo y apunta las posibles líneas de desarrollo futuro para la ampliación y mejora del proyecto.

Tras los capítulos enumerados se encuentra la bibliografía, el glosario de términos y acrónimos, y los apéndices con datos menos relevantes.

2

Estado del Arte

Desde la era babilónica hasta la actualidad el álgebra ha evolucionado constantemente. La tecnología ha tenido un fuerte impacto en esta rama de la matemática, tanto en su desarrollo como en sus métodos de enseñanza y aprendizaje. En este apartado se pretende dar una panorámica de dicha evolución, esta información se puede ampliar en las fuentes utilizadas [7] [8]. Para finalizar, y como resultado de la evolución que se detalla, se presentan algunos sistemas de álgebra computacional que tienen, o han tenido, un gran impacto en el desarrollo o uso actual de esta tecnología.

2.1. Del álgebra tradicional al computacional

Fue en la antigua Babilonia (2000 a.C.) donde surgieron las raíces del álgebra debido al desarrollo de un avanzado sistema aritmético que, basado en ideogramas, se usaba para la resolución de cálculos de forma algorítmica. De esta forma encontraron soluciones a problemas que hoy en día se calculan mediante ecuaciones. De una manera similar, los griegos usaban el álgebra geométrica para resolver ecuaciones mediante el uso y transformación de figuras geométricas a cuyos lados asociaban letras que representaban los términos de dichas ecuaciones. No obstante, fueron el matemático alejandrino Diofanto, en el siglo III, y el indio Brahmagupta, en el siglo VII, quienes consiguen un gran nivel de desarrollo en los inicios del álgebra. Este último, en su libro *"Brahmasphutasiddhanta"*, muestra la primera solución aritmética completa para ecuaciones cuadráticas que incluía los números negativos. Por su parte, Diofanto escribió una serie de libros llamados

"Arithmetica" que contenían una colección de problemas, entre ellos diversas ecuaciones algebraicas.

Considerado uno de los padre del álgebra, el matemático persa Al-Juarismi, en el siglo VIII, dio nombre a esta rama de la matemática mediante el uso de una operación llamada *al-ğabr* (restauración) y descrita en su obra *"Hisob al-jabr wa'l muqabalah"*, en castellano, *"Compendio de cálculo por reintegración y comparación"*. En él estudia de forma sistemática la resolución de ecuaciones de primer y segundo grado aunque sin el uso de la simbología algebraica actual, los coeficientes negativos ni el cero. A partir de este momento la evolución es continua, llegando al siglo XVI cuando, entre otras aportaciones, François Viète introduce el lenguaje simbólico para las expresiones polinómicas. Las vocales representaban lo desconocido y el resto de constantes simbolizaban parámetros o números conocidos.

En la Edad Moderna Gottfried Leibniz aspira a construir un lenguaje formalizado y crea una máquina que daba solución a sumas, restas, multiplicaciones y divisiones. Pero su ambición era construir una que pudiera determinar los valores verdaderos de los enunciados matemáticos. Plantea ciertos axiomas que, dos siglos más tarde, servirían como base para la creación, por parte de George Boole, del álgebra de Boole. Sin embargo, Charles Babbage siguió desarrollando el concepto, construyendo una máquina programable para cualquier tipo de cálculo, la Máquina analítica. Entre sus elementos se encontraba un procesador aritmético y una unidad de control para decidir la tarea a realizar. Ada Lovelace se encargaría de escribir diferentes programas para su uso en dicha máquina.

Es en el siglo XX cuando David Hilbert plantea un reto consistente en encontrar un algoritmo que determinara si una fórmula del cálculo de primer orden, o lógica de predicados, es un teorema. A esto se le conoce como *"Entscheidungsproblem"* o *"El problema de la decisión"* [9]. Poco después, Gödel enunció su teorema de incompletitud [10]: ningún sistema formado por un conjunto de axiomas podía ser consistente y completo a la vez. Dicho de otro modo, en este tipo de sistemas, si los axiomas no se contradicen entre sí, siempre existen enunciados imposibles de refutar o probar. Las conclusiones de este teorema siempre se aplican si los axiomas pertenecen a una teoría aritmética recursiva, es decir, el proceso de deducción se lleva a cabo mediante un algoritmo.

Influenciados por las ideas de Gödel, Alonzo Church y Alan Turing dieron una respuesta negativa al problema de la decisión. El primero usó el cálculo lambda [11], un sistema formal completo que modeliza la matemática y permite la definición de funciones computables. Con ello demostró que no existe ningún algoritmo que pueda decidir si dos expresiones del cálculo son iguales. Por otro lado, Turing concluyó que no se puede asegurar que una demostración tenga solución. Usó el problema de la parada y la máquina

de Turing [12], que realizaba manipulaciones con símbolos. En ocasiones, la máquina que diseñó no finalizaba sus ejecuciones por lo que no se podía saber el resultado.

Podemos considerar la máquina de Turing como la precursora de los CAS, en ambos casos hay una actuación automática en función de un símbolo de entrada. A pesar de esto, no será hasta 1963 cuando Martinus Veltman cree el primer ejemplo real de CAS, influenciado por la física teórica y la inteligencia artificial, en lenguaje ensamblador. Lo llamará *Schoonschip* [13].

2.2. Sistemas de Álgebra Computacional

Un sistema de álgebra computacional es un programa, o calculadora avanzada, que modeliza la idea de algoritmo algebraico y facilita el cálculo simbólico. Dicho de otro modo, es una herramienta que permite la manipulación de expresiones mediante ciertas reglas predefinidas. Estas reglas representan los pasos que realizaría una persona cuando intenta resolver una expresión algebraica. Para aplicarlas se usan los sistemas de reescritura de términos, los cuales transforman expresiones complejas en otras más sencillas de manipular. Aunque sobre estos sistemas hay una gran influencia del cálculo lambda, existen diversas implementaciones de CAS usando lenguajes orientados a la programación declarativa, que permiten y facilitan el cálculo simbólico, como Haskell [14], LISP [15] o Prolog [16].

Tras la creación del primer CAS, *Schoonschip*, no pasaron muchos años hasta la aparición de los primeros sistemas populares, *Reduce* [15] y *Macsyma*. De este último existe una versión actualizada y mantenida llamada *Maxima* [17]. No obstante, los más usados en la actualidad por ingenieros, matemáticos y científicos son *Maple* [18], *Mathematica* [19] y *Matlab* [20] que, aunque destaca por su cálculo numérico, permite el cálculo simbólico.

2.3. Herramientas CAS

La funcionalidad y la tecnología que se han usado como base de los sistemas de álgebra computacional ha ido creciendo y evolucionando con el tiempo. En la actualidad, aunque la mayoría de estos sistemas tienen un propósito de carácter general, existe un pequeño grupo de CAS que se centran en áreas especificas, la mayoría desarrollados en universidades como software libre. Otra característica a destacar es que, como norma general, la búsqueda de la solución está automatizada, si bien es cierto que suelen

mostrar los pasos dados para llegar a ella. Chalky, por su parte, ha pretendido ser un apoyo a los docentes, facilitando la transmisión de conocimiento de una forma dinámica y eficiente. Esto motiva que el motor de la versión 2.0, y el que se va a desarrollar en el actual proyecto, no sustituyan los cálculos manuales ni el conocimiento de las reglas algebraicas de un usuario, necesarias para resolver los problemas.

Centrándonos en los sistemas que se aplican en matemáticas, exceptuando casos muy concretos, su funcionalidad principal es presentar al usuario todas las posibles soluciones a los problemas que plantee. Chalkpy, por su parte, busca que sea el usuario quien encuentre la solución, llevando a cabo cualquier manipulación que crea conveniente y que esté definida en el motor. De este modo se centra en una resolución manual, recordando al uso del lápiz y papel, apoyando al usuario para evitar que cometa algún error.

A continuación de describe una serie de sistemas de álgebra computacional relevantes para el proceso de desarrollo de Chalkpy.

Reduce

Reduce es un sistema de álgebra computacional de propósito general desarrollado en la década de los 60 por Anthony Hearn, aunque posteriormente ha contado con la colaboración de incontables personas. Este sistema está desarrollado en un dialecto del lenguaje de programación Lisp.

Reduce se centra en la resolución automática de expresiones y permite la ampliación de funcionalidad mediante la definición de nuevas funciones. En contrapunto, el mecanismo para introducir esta nueva lógica precisa de un aprendizaje previo del lenguaje usado por el motor, llegando a resultar complejo para un usuario con poca experiencia en programación.

Maxima

Maxima es un motor de cálculo simbólico que se lanzó en 1982, evolucionando del sistema Macsyma desarrollado por el MIT. Aunque cuenta con interfaz gráfica está pensado para funcionar en modo consola. Está programado con un lenguaje de programación de la familia Lisp.

Como pasa con el caso anterior, permite la introducción de nueva funcionalidad. Y, aunque el uso de su lenguaje no es muy complejo, para resolver algunos problemas es necesario ser un usuario experimentado y avanzado en Lisp.

Mathematica

Mathematica es el sistema original del grupo *Wolfram Research*, propietarios de uno de los CAS más utilizados a día de hoy, WolframAlpha [21]. Con uno de los catálogos más extensos, en la actualidad cuenta con más de 5000 funciones y algoritmos.

Escrito con tecnología propia, *Wolfram Language*, su lenguaje simbólico, potencia el sistema y cubre la mayoría de los objetivos de Chalkpy. Esto es así porque el propósito de su lenguaje es la construcción de algoritmos accesibles a través de un lenguaje simbólico unificado, y la representación de elementos como expresiones simbólicas. La funcionalidad completa de este CAS está contenida en una herramienta que requiere el pago de una licencia.

YACAS

YACAS [22] es un sistema de álgebra computacional de propósito general y código abierto. Diseñado con un lenguaje propio, muy ligado a Lisp, permite que el usuario introduzca nuevas funciones. También destaca por la posibilidad de llevar a cabo manipulaciones manuales y no únicamente resoluciones automáticas.

Destacando sobre los anteriores, la extensibilidad del sistemas no es sencilla. Se requiere de un aprendizaje que agrupa los diferentes tipos de elementos que soporta, sus funciones y reglas, así como la estructuración de sus programas o la evaluación de las expresiones, entre otros.

PRESS

PRESS es el acrónimo de *PRolog Equation Solving System*, un sistema enfocado en la resolución de ecuaciones de forma automatizada. Fue desarrollado en 1974 como una herramienta para explorar la posibilidad de llevar a cabo demostraciones automáticas de teoremas.

Aunque no es un Sistema de Álgebra Computacional al uso, la parte dedicada a la resolución de ecuaciones tiene muchas características similares a las de los sistemas presentados anteriormente. Y, como la mayoría de los CAS, está enfocado en la resolución automatizada.

3

Definición del proyecto

En este capítulo se detallará el alcance del proyecto y la metodología elegida. Finalmente, se explicarán las tecnologías y herramientas utilizadas en todas las fases del proyecto.

3.1. Alcance

Chalkpy 2.0 permite la definición de operaciones mediante su simbología, su número de argumentos y sus propiedades básicas, operar con constantes simbólicas (el uno o el cero por ejemplo), resolver ecuaciones de primer grado y manipular expresiones con operaciones sencillas como la suma o la multiplicación. También permite modificar el motor según los requisitos del usuario de forma sencilla y con una necesidad mínima de conocimientos en programación, basta con saber definir operaciones en Java.

La nueva versión de Chalkpy debe mantener la funcionalidad existente en la anterior, permitiendo la definición de operaciones y expresiones, así como la manipulación de expresiones algebraicas y ecuaciones. Se presentan dos mejoras, la primera es añadir mecanismos que permitan el uso de operaciones más complejas que las usadas hasta ahora y, la segunda y objetivo principal del proyecto, permitir una amplia extensibilidad de forma sencilla. Este es el punto más crítico, para ello el motor debe contar con definiciones operacionales genéricas y que no requieran de un gran conocimiento en programación para su uso, este sigue siendo uno de los principales preceptos de Chalkpy.

El proyecto contará con varias pruebas de concepto que comprueben si se ha cumplido el alcance estipulado. Estas pruebas se llevarán a cabo mediante la definición y uso de operaciones tales como sumas, multiplicaciones, sumatorios y ecuaciones.

3.2. Metodología

Para la elección de la metodología a utilizar no solo se ha tenido en cuenta la definición del proyecto, su alcance y sus objetivos. También se ha considerado el desarrollo y los problemas a los que hubo que enfrentarse durante la creación del primer CAS para Chalkpy. Este conocimiento hace que se estimen las dificultades que plantea definir de forma genérica los operadores y propiedades básicas de manipulación algebraica de modo que puedan valer para diferentes expresiones y ámbitos. Por todo ello se ha decidido llevar a cabo un análisis inicial de requisitos, seguido de un modelo de desarrollo iterativo e incremental [23] donde, tras finalizar cada incremento, se obtendrá una versión funcional del motor. No obstante, se deja abierta la posibilidad de modificar la lógica desarrollada en incrementos anteriores siempre que sea necesario para la generalización de las definiciones.

Las fases en las que se dividirá cada iteración serán:

- **Análisis:** se enfoca en completar el catálogo inicial de requisitos funcionales y no funcionales. Es decir, aquellos que definen el funcionamiento del sistema, qué debe hacer, y sus propiedades, cómo debe hacerlo.

- **Diseño:** se centra en la creación de los algoritmos necesarios para la definición de las propiedades algebraicas básicas, la definición de operadores y las manipulaciones de las expresiones. Como se ha señalado, en esta fase se permite la modificación de algoritmos ya diseñados por necesidad de generalizarlos a nuevas operaciones o ámbitos.

- **Implementación:** codificación del diseño y de las pruebas específicas de la funcionalidad que se desarrolla. Al igual que en la fase anterior, y complementándola, se modificará la lógica ya implementada en función de los cambios decididos en el diseño.

- **Pruebas:** ejecución de todas las pruebas, independientemente de su tipo. Esta fase permite dar por cerrado los módulos o ficheros que estén completos y con un funcionamiento correcto.

- **Documentación:** recopilación de todas las notas y apuntes generados durante las fases anteriores para desarrollar la presente documentación.

Respecto a los incrementos se han dividido en función de los hitos que se marcan de inicio:

- Definición de la operación, las propiedades(conmutativa, asociativa, etc.) y los operadores básicos(suma, multiplicación, etc.).
- Implementación de nuevos operadores como la potencia.
- Resolución de ecuaciones de primer grado.
- Uso del motor para la resolución de operaciones en Álgebra de Boole.
- Implementación de derivadas.
- Implementación del sumatorio.

3.3. Tecnología y Herramientas

Para la realización del proyecto se han utilizado las herramientas y tecnologías que se exponen en las siguientes secciones.

3.3.1. Tecnologías

- **Prolog:** [24] lenguaje de programación lógico perteneciente a la programación declarativa. Una de las principales ventajas de este tipo de lenguajes es la posibilidad de definir varias declaraciones iguales con distinta funcionalidad, siendo el propio lenguaje quien, durante una ejecución, encuentra la declaración correcta en función de los parámetros con los que se trabaje.
- **AWK:** [25] lenguaje de programación enfocado al procesamiento de flujos de datos, o ficheros, en formato texto. Debido a su amplio uso de expresiones regulares se ha elegido como tecnología para el tratamiento de las salidas de SWI-Prolog en el testeo del motor resultante.

- **LaTeX:** [26] sistema para la creación de texto que permite al usuario centrarse en el contenido más que en el formato. Adicionalmente, aporta una gran capacidad física para representar ecuaciones o fórmulas complejas, y una estructuración sencilla. Ha sido utilizado para el desarrollo de este documento.

3.3.2. Herramientas

- **Vim:** [27] editor de texto integrado en todos los sistemas UNIX, se ha utilizado como entorno de desarrollo para el motor. Esta elección se debe, principalmente, a sus amplias opciones de configuración y eficiencia, ya que gran parte del trabajo iba a ser desarrollado en una maquina virtual.
- **SWI-Prolog:** [28] implementación de código abierto multiplataforma del lenguaje de programación Prolog. La razón principal por la que se ha decidido elegir esta implementación es porque cuenta con una extensa librería de predicados. Otro de los motivos de su elección es su continuo desarrollo y soporte.
- **Bitbucket:** [29] para el control de versiones se ha utilizado esta plataforma con soporte para Mercurial [30] y Git [31]. Este último ha sido el software de revisiones elegido para el seguimiento del avance y los cambios de todos los elementos del proyecto.

4

Análisis

Durante la fase de análisis se ha llevado a cabo la creación del catálogo de requisitos funcionales y no funcionales; guía que detalla las funcionalidades y características que debe recoger el nuevo motor de Chalkpy. Debido a la metodología utilizada, y el planteamiento del proyecto, se definió un primer catálogo de requisitos con las bases del motor y la funcionalidad genérica. A lo largo del proyecto, en las fases de análisis de las distintas iteraciones e incrementos, dicho catalogo se iría completando y modificando siempre que fuese necesario.

4.1. Requisitos Funcionales

Los requisitos funcionales definen las funciones, o componentes, de un sistema. Deben detallar las acciones o funcionalidades propias del sistema con las que debe contar durante su ejecución.

RF1 **Definición de expresiones.** Un usuario debe poder definir cualquier expresión algebraica utilizando números, variables alfanuméricas, constantes simbólicas y cualquiera de sus operadores. Esto incluye la utilización de otras expresiones como argumentos de un operador.

RF2 **Definición de ecuaciones.** Un usuario debe poder definir cualquier ecuación utilizando números, variables alfanuméricas, constantes simbólicas y cualquiera de los operadores definidos.

RF3 **Definición de operadores.** Un usuario debe poder definir operadores mediante su nombre, su simbología, su tipo de representación, sus argumentos, sus propiedades matemáticas y su operativa. El nombre del operador debe ser obligatorio.

RF4 **Definición de constantes simbólicas** Un usuario debe poder definir constantes simbólicas con un nombre y su valor numérico.

RF5 **Uso de constantes simbólica.** El usuario debe poder introducir cualquiera de las constantes simbólicas en sus expresiones.

RF6 **Operar constantes simbólicas.** Un usuario debe definir el comportamiento de sus operadores respecto a las constantes simbólicas.

RF7 **Orden en las definiciones.** La introducción de la definición de un operador en un fichero no debe obligar al usuario a seguir un orden específico, excepto cuando hagan referencia a la misma propiedad del operador. En este caso, entre ellas, deben seguir un orden de mayor a menor restricción en su comportamiento.

RF8 **Ejecución de manipulación.** Un usuario debe poder ejecutar cualquier manipulación que haya definido sobre los términos seleccionados.

RF9 **Manipulación de expresiones.** Un usuario debe poder manipular las expresiones o ecuaciones ya sea en su totalidad o cualquiera de los términos que la conforman.

RF10 **Propiedades primitivas.** El sistema debe saber llevar a cabo, de forma genérica, las propiedades básicas de las matemáticas, a saber: propiedad asociativa, propiedad conmutativa y propiedad distributiva. Dichas propiedades podrán ser aplicadas a un número arbitrario de elementos.

RF11 **Operar.** El sistema debe incluir la lógica genérica para operar cualquier operador.

RF12 **Constantes simbólica.** Para el sistema los números *1*, *0* y *-1* deben ser constantes simbólicas predefinidas.

RF13 **Valores por defecto.** El sistema debe dar un valor por defecto a aquellas características que no defina el usuario.

RF14 **Expresión matemática de trabajo.** El sistema solo permitirá trabajar sobre una expresión a la vez.

RF15 Términos seleccionables. El sistema debe permitir seleccionar cualquier término. Es decir, tanto la expresión general como cualquiera de los elementos que la conforman.

RF16 Representación interna. Todas las expresiones deberán ser implementadas como una lista.

RF17 Representación del usuario. El sistema debe poder representar cualquier expresión en la notación designada por el usuario.

RF18 Corrección matemática. El sistema no debe comprobar la corrección matemática de las operaciones o ejecuciones de las manipulaciones ejecutadas por el usuario siempre que cumplan con lo definido por este.

RF19 Control en las manipulaciones. El sistema deberá detectar si la expresión no cumple los requisitos necesarios para llevar a cabo una manipulación ejecutada por el usuario. Si esto ocurriese, el sistema nunca modificará la expresión.

RF20 Programa principal. El sistema contará con un programa principal que permita la realización de pruebas. En cada ejecución deberá mostrar la expresión actual en dos posibles formatos: el interno y el del usuario.

RF21 Ficheros de definiciones. El sistema debe admitir más de un fichero de definición de usuario.

4.2. Requisitos no funcionales

Los requisitos no funcionales definen las características del sistema. Imponen criterios o restricciones que debe cumplir el sistema definiendo los requerimientos mínimos de su ejecución. Suelen estar relacionados con la interfaz, la usabilidad, el rendimiento o la documentación.

RNF1. Requisitos del sistema. El sistema está pensado para ejecutarse en un entorno con la versión de SWI-Prolog 7.6 o superior.

RNF2. Tiempo de respuesta. El sistema no tendrá un tiempo de respuesta superior a 300 ms.

5

Diseño e Implementación

Este capítulo muestra las decisiones de diseño que han sido necesarias para la creación del motor, cumpliendo las especificaciones que marcan los requisitos recogidos en el Capítulo 4. Para ello se detallarán los aspectos más relevantes de estas decisiones, precedidos por una explicación de la organización de los diferentes elementos a desarrollar, denominado arquitectura del proyecto. Sobre este diseño, se marcarán aspectos esenciales de la implementación, o codificación, enfocada en hacer funcional el diseño elegido.

5.1. Arquitectura del proyecto

La primera decisión que se ha tomado respecto al diseño es separar el motor y la interfaz de pruebas por lo que serían dos sistemas diferenciados pero dependientes en el caso de este último. Es decir, la interfaz necesitará el motor para su funcionamiento y se desarrollará en base al CAS, no obstante, no será tenida en cuenta para la elección del diseño del sistema algebraico. Llegados a este punto se ha decidido optar por una arquitectura de Modularización o Descomposición Modular orientada a flujos de funciones [23].

La Descomposición Modular orientada a flujos de funciones se basa en la división del sistema en elementos funcionales denominados módulos. Cada uno de ellos agrupará las tareas con una funcionalidad o fin similar donde, en la mayoría de los casos, para unos datos de entrada dados y tras una transformación, generará los datos de salida. Hay

que tener en cuenta que este modelo puede llegar a presentar un problema, encontrar un formato común para que los datos sean reconocidos por todas las transformaciones. Pero en el caso del motor de Chalkpy siempre se va a trabajar con listas de elementos con un formato predefinido por lo que, por requisitos, ya existe una estandarización de los datos.

A continuación se detallan los diferentes módulos que compondrán el sistema. Serán un total de siete módulos: seis primitivos, formarán el motor desde un inicio, y uno no primitivo, lo creará el usuario y se añadirá a posteriori. Esto se refleja en la Figura 5.1, donde se muestra un diagrama con dichos módulos y la interacción de los datos entre ellos.

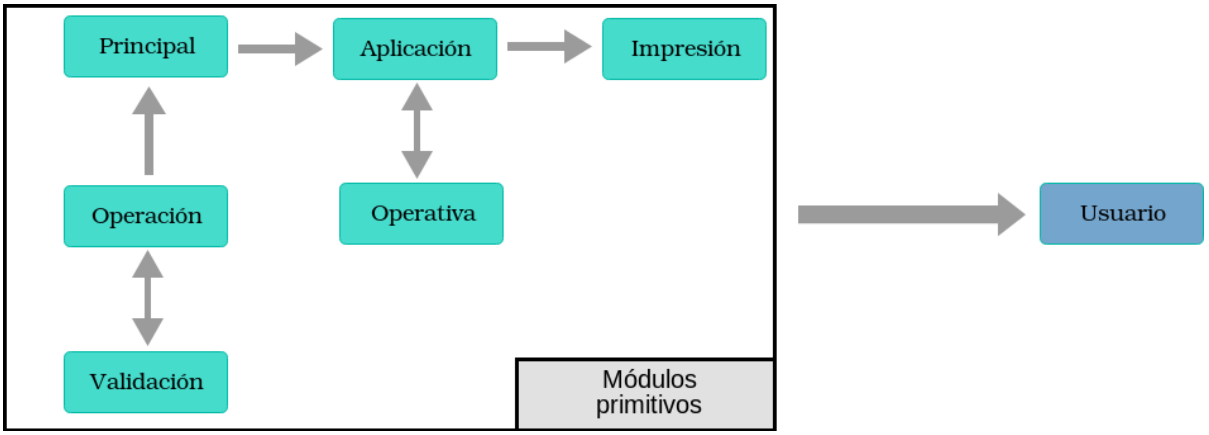

Figura 5.1: Modularización del sistema y su comunicación

1. **Módulo de operación:** se encarga de crear las expresiones algebraicas y ecuaciones. Además controla la selección de los términos.
2. **Módulo de validación:** comprueba que las expresiones definidas sean válidas según las especificaciones de las operaciones que la conforman.
3. **Módulo principal:** aunque se va a considerar un módulo podría definirse incluso como la API del motor en un futuro. Su única función es recoger los datos que serán necesarios para ejecutar las distintas manipulaciones.
4. **Módulo de aplicación:** se encarga de aplicar la manipulación deseada en el término seleccionado.
5. **Módulo de operativa:** comprende toda la lógica de las manipulaciones primitivas: asociar, disociar, conmutar, etc.
6. **Módulo de impresión:** lleva a cabo la transformación de la expresión interna para imprimirla con el formato definido por el usuario.

7. **Módulo de usuario:** contendrá todas las definiciones de operadores del usuario y, por tanto, será dado por él.

Finalmente podemos definir el flujo de funciones típico del sistema como se muestra en la Figura 5.2.

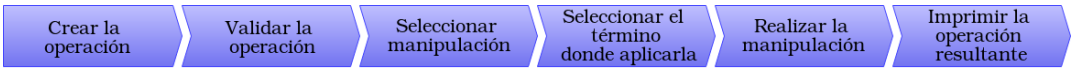

Figura 5.2: Flujo típico de las funciones del sistema

5.2. Sistema de Álgebra Computacional

En esta sección se detallarán las decisiones particulares tomadas sobre el diseño del CAS junto a su motivación. Algunas de estas pueden formar parte de las fases de análisis pero se ha decidido presentarlas en este capítulo, donde se exponen los posibles problemas que han surgido junto a las soluciones dadas.

5.2.1. Expresiones algebraicas

Como ya se ha visto en el catálogo de requisitos cualquier expresión que se defina debe tener una representación interna en formato de lista. Tanto en las expresiones algebraicas como en las ecuaciones que se deben manipular, pueden existir operadores cuyos argumentos sean otras operaciones. Por ello se ha decidido que el formato final de todas las expresiones algebraicas del sistema sea una lista de listas. Pero, a diferencia de la estructura del primer motor de Chalkpy, cada expresión será una única lista donde el primer elemento es la operación y el resto sus argumentos. Dichos argumentos pueden ser números, variables, constantes simbólicas u otras listas que representen expresiones internas. Un ejemplo de esta representación se muestra en la Figura 5.3 donde se representa la expresión $(x + 2 + 1) * 5{,}3$.

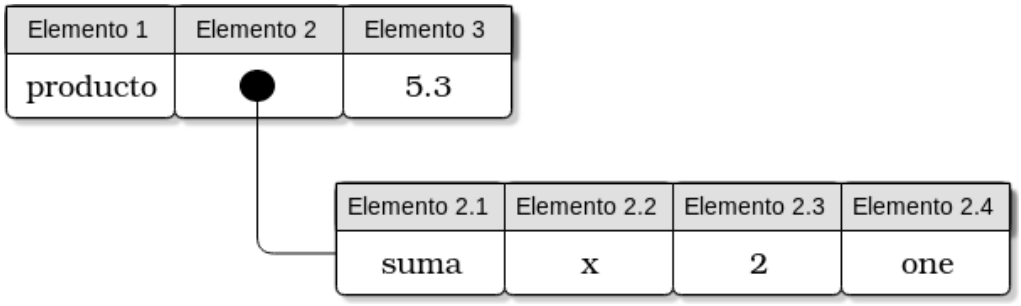

Figura 5.3: Ejemplo de la representación de una expresión algebraica como lista

5.2.2. Ecuación

Podemos definir el concepto de ecuación como la igualdad entre dos expresiones que contienen una o más variables. Además la igualdad ($=$) puede ser considerada como un operador que permite introducir elementos en ambas expresiones. Teniendo en cuenta esto, y que cualquier expresión se representa como una lista de listas cuyo primer argumento es la operación, descrito en el punto anterior, se podría utilizar la misma representación para las ecuaciones. El segundo y el tercer argumento de la lista serían las dos expresiones que componen la ecuación. De esta forma se convierte al operador de igualdad en una operación primitiva. En la Figura 5.4 se ejemplifica la representación interna de la ecuación $4 + x = 7$.

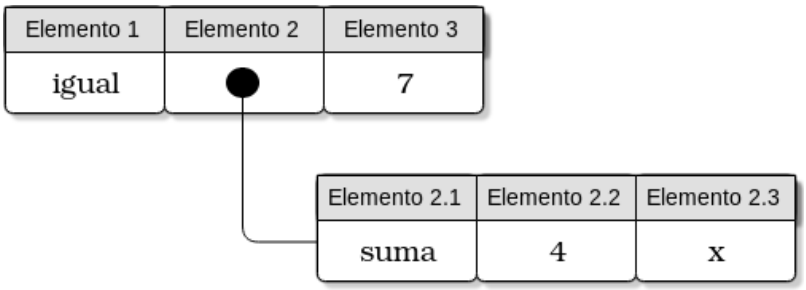

Figura 5.4: Ejemplo de la representación de una ecuación como lista

Llegado este punto se planteaba el problema de la operativa del igual. Como se ha marcado, su operar tiene que introducir expresiones a ambos lados de la ecuación. Debido a la representación elegida se puede traducir este funcionamiento como la creación de una nueva lista cuyo operador corresponde al de la nueva expresión y sus argumentos son, por una parte, los términos de la ecuación y, por el otro, los nuevos elementos a

introducir. Por ejemplo, queremos sumar un 2 a cada lado de la siguiente ecuación.

$$x + 2 = 6 \tag{5.1}$$

Podemos llevarlo a cabo creando una nueva ecuación de la siguiente forma:

$$Z + 2 = Z + 2 \tag{5.2}$$

Ahora solo sería necesario sustituir cada una de las Z por cada uno de los términos de la ecuación original, 5.1, dando como resultado:

$$(x + 2) + 2 = 6 + 2 \tag{5.3}$$

Esto nos permite también posicionar, respecto al término original, los nuevos elementos ya que podríamos añadirlos tanto al principio de los términos de la ecuación como al final. Para indicar al motor cuales son las nuevas expresiones que forman la ecuación, pudiendo marcar el orden, valdría crear una lista de la forma: [*operador*,@,*nuevos argumentos*]. El símbolo @ es el elegido para marcar donde se deben colocar los términos antiguos. Además, será el motor quien deba llevar a cabo las sustituciones con las dos expresiones que forman la ecuación original. El código de esta lógica se muestra más adelante, en el Código 5.8 perteneciente a la sección donde se explica el mecanismo que se ha implementado para llevar a cabo las sustituciones de elementos en expresiones.

5.2.3. Selección de términos

Como indica el requisito funcional **RF15** cualquier parte de la expresión con la que se esté trabajando debe poder ser seleccionada por el usuario. Además el **RF8** señala la necesidad de aplicar una manipulación a la selección sin que afecte al resto de la expresión. En la versión anterior de Chalkpy, para conseguir dicho control, cada lista contaba con un identificador único y una referencia al identificador de la lista padre. No obstante, en esta versión se ha preferido buscar una manera de señalizar, en la propia expresión, cual es el término seleccionado. De esta forma no es necesario guardar datos extras.

Al querer marcar la selección en la propia expresión, y teniendo en cuenta su representación interna, se ha decidido generar la selección como un operador primitivo especial. Dicho operador tendrá como único argumento el término seleccionado. De esto se deriva que pueda ser un argumento de cualquiera de los operadores con los que trabaje el motor cuando marque una subexpresión. Con esta lógica, cuando se quiera aplicar cualquier manipulación, solo será necesario buscar el operador selección.

Hay que tener en cuenta que este operador debe aparecer obligatoriamente. Para simplificar el trabajo de un usuario, si este no lo definiese, será el propio motor quien introduzca el operador seleccionando la expresión completa.

La Figura 5.5 muestra la expresión algebraica **8*2** con todos sus elementos seleccionados, y la ecuación **1+x** $= 7$ donde solo uno de sus miembros tiene tal selección. La selección se ha mostrado en verde.

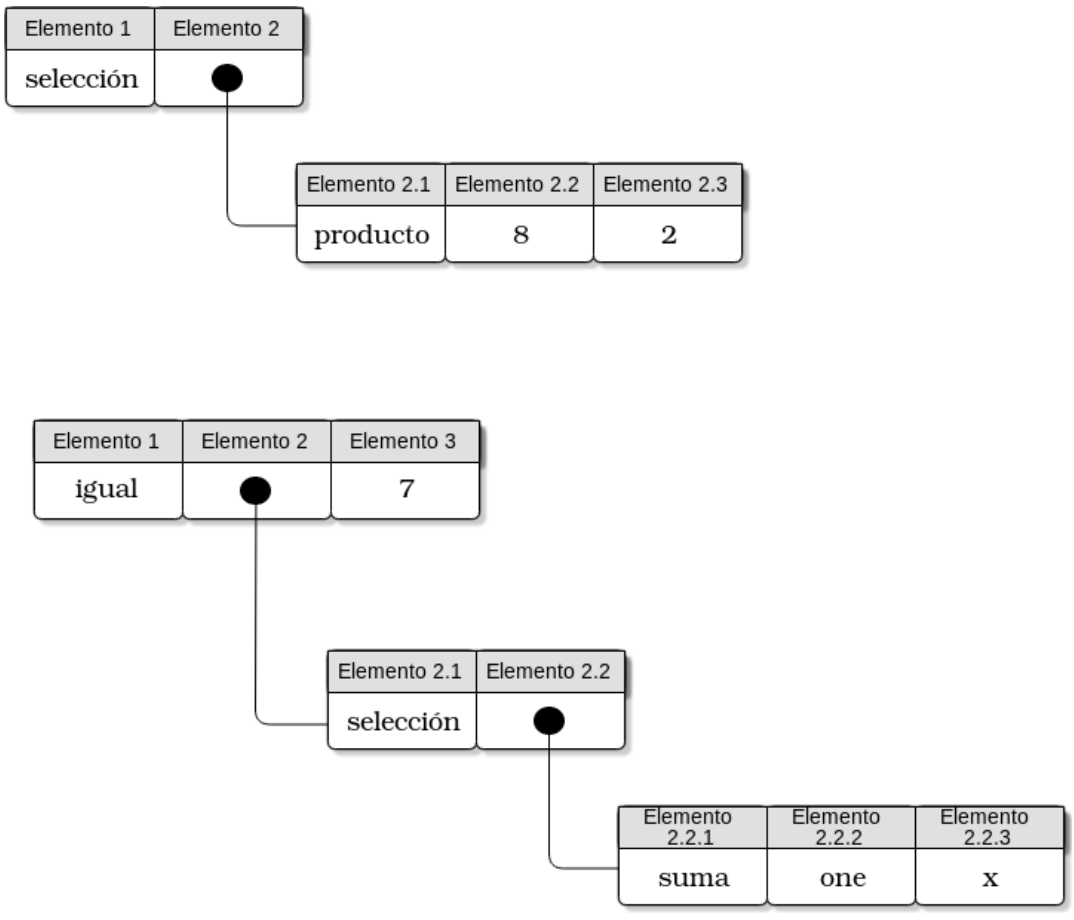

Figura 5.5: Ejemplos de expresión algebraica y ecuación con selección.

5.2.4. Definición de operadores

Los operadores deben definirse por un nombre, su simbología, el tipo de representación, sus propiedades y sus argumentos. Las tres primeras características son decisión del usuario y el sistema solo debe conocerlos, y utilizarlos, para crear e imprimir las expresiones y comprobar si una manipulación se puede aplicar sobre un operador. El motor contará con unos predicados para tal fin.

En los siguientes apartados se van a detallar todas las definiciones que se deben llevar a cabo para la creación de un operador, apoyándonos sobre los ejemplos de la suma. El ejemplo completo se puede ver en el Anexo A.

Nombre

La definición del nombre en el motor también marca el punto en que el CAS sabe de la existencia de este operador para su posible uso. Todos los nombres de los operadores se definen con el predicado *operator* cuyo parámetro será el nombre. Vemos el ejemplo de la definición de la suma en el Código 5.1.

```
operator(suma).
```

Código 5.1: Definición del operador suma.

Simbología

La simbología define como debe representar el motor los operadores para mostrárselos a un usuario. El motor ya incorpora la lógica de impresión genérica para mostrar las expresiones, por lo que el predicado que se usa para esta definición es el mismo que el usado para la lógica especifica de impresión de operadores. Recibe como argumento el nombre del operador y lleva asociada la lógica para imprimir por pantalla el símbolo. Para asignar el símbolo $+$ a la suma es necesario detallar el predicado mostrado en el Código 5.2

```
print_operator(suma) :-
    write(" + ").
```

Código 5.2: Definición del símbolo de la suma.

Tipo de representación

Se definen dos tipos de representación, o notación, para los operadores: infija y prefija. La notación infija muestra el operador entre los operandos mientras que en la prefija el operador los precede. Como ya se ha mencionado el motor cuenta con una lógica genérica para la impresión de expresiones, y como veremos en la subsección 5.2.7, existe un tipo de representación por defecto para todos los operadores. Si el usuario quisiera cambiarlo debería indicar un segundo parámetro al predicado *operator* con el valor *infijo* o *prefijo*, según el tipo que se desee. En el caso de la suma, donde se quiere una representación infija, se definiría como muestra el Código 5.3.

```
operator(suma,infijo).
```

Código 5.3: Definición del tipo de representación para la suma.

Argumentos

Los operadores básicos como la suma o el producto pueden tener un número arbitrario de argumentos; otros determinan un número exacto. Por ejemplo, la potencia tiene dos argumentos mientras que la exponencial solo uno. Hay que tener en cuenta también que hay ciertos operadores que quizás precisen, no solo de definir el número de argumentos, sino de colocarlos en una posición determinada o comprobar que cumplan ciertas condiciones, por ejemplo el sumatorio.

Para poder cubrir las necesidades de los distintos operadores, y facilitar el trabajo del usuario, se desarrollan dos métodos para la definición de los argumentos. En el primero de ellos el usuario contará con unos predicados, mostrados en la Tabla 5.1, para indicar si el operador es unario, binario o n-ario, además de las validaciones necesarias para comprobar que las expresiones cumplen la condición del número de argumentos definido.

Predicado	Significado
unary_operator(Op)	Indica que el operador *Op* tiene 1 argumento.
binary_operator(Op)	Indica que el operador *Op* tiene 2 argumentos.
n_ary_operator(Op)	Indica que el operador *Op* tiene un número arbitrario de argumentos.

Tabla 5.1: Predicados correspondientes a la definición de argumentos de un operador

Por ejemplo, el Código 5.4 muestra la definición de la suma como un operador n-ario.

```
n_ary_operator(suma).
```

Código 5.4: Definición de la suma como operador n-ario.

En el segundo caso nos situamos en la necesidad de que los argumentos cumplan ciertas condiciones y/o tengan un número determinado mayor que dos, debiendo ser el usuario quien defina los métodos. Para abordarlo, los usuarios cuentan con el predicado *valid_list* cuyo único parámetro es una lista con el formato que debe cumplir la expresión usada para el operador. Además podrá asociarle la lógica necesaria para las comprobaciones que quiera que se lleven a cabo. Por ejemplo, el Código 5.5 muestra la definición de un operador *opr* con tres argumentos numéricos, los cuales, deben cumplir que el segundo siempre sea el número 3 y el primero debe ser menor o igual que el tercero.

```
valid_list([opr,Arg1,3,Arg3]) :-
    number(Arg1),
    number(Arg3),
    =<(Arg1, Arg3).
```

Código 5.5: Definición de argumentos personalizados para el operador opr.

Propiedades del operador

Todas las propiedades matemáticas básicas (asociar, conmutar, etc.) para resolver una expresión vienen determinadas como primitivas. Esto hace que el usuario solo necesite indicar las propiedades que cumple un operador. Para ello se aportan los predicados definidos en la Tabla 5.2.

Predicado	Significado
commute(Op)	Indica que el operador *Op* cumple la propiedad conmutativa.
associate(Op)	Indica que el operador *Op* cumple la propiedad asociativa.
distribute(Op1,Op2)	Indica que el operador *Op1* cumple la propiedad distributiva respecto al operador *Op2*.

Tabla 5.2: Predicados para las propiedades básicas que cumple un operador.

Se considera también como propiedad de un operador la posibilidad de operarlo, siendo necesario para ello que el usuario defina el predicado *operate* cuyos parámetros serán la expresión sobre la que aplicar la operación y una variable para devolver el resultado. Para llevar a cabo el proceso de operar el motor implementa una lógica genérica, pero es el usuario quien debe detallarlo. Esto está relacionado, en parte, con el número de argumentos. Por ejemplo, los operadores con dos o más argumentos que cumplen la propiedad asociativa y conmutativa se pueden operar de manera recursiva. Este requisito lo cumple, entre otros, la suma, por lo que para operarla sería posible coger sus dos primeros elementos y sumarlos. A continuación, al resultado de este paso, se le podría sumar el tercer argumento, y seguir así hasta haber procesado todos los elementos de la suma obteniendo el resultado final. En caso de no cumplirlo el usuario debe indicar como se operan todos sus argumentos.

Similar a la solución encontrada para la definición de los argumentos, existen dos alternativas para definir el proceso de operar, aunque se usa el mismo predicado. En el caso de que el operador sea n-ario y cumpla la propiedad asociativa y conmutativa podrá operarse por pares. El usuario solo necesitará definir el comportamiento respecto a dos elementos y será el propio sistema el que aplique la recursividad con todos los argumentos. En caso contrario, será el usuario el que deba determinar cómo se operan todos y cada uno de sus argumentos.

Como se ve en el Código 5.6, el operador suma, definido como n-ario y con la propiedad asociativa y conmutativa, define la operación solo para dos elementos y con la condición de que ambos sean números.

```
operate(suma).

operate([suma,Elemento1,Elemento2],R) :-
    number(Elemento1),
    number(Elemento2),
    R is Elemento1+Elemento2.
```

Código 5.6: Definición del proceso de operar para la suma.

5.2.5. Operador evalúa

Para la sustitución de una variable por su valor en la resolución de ecuaciones, y para la transformación de las constantes simbólicas entre su valor numérico y su símbolo, es necesario que el motor cuente con un mecanismo que lleve a cabo estas transformaciones. Se ha decidido que este mecanismo sea un operador primitivo cuyos argumentos son el elemento a sustituir, el elemento sustitutivo y la lista donde llevar a cabo la sustitución. Su definición de operar debe buscar en la lista el elemento a sustituir y cambiarlo por el nuevo valor, devolviendo la lista modificada. En el Código 5.7 podemos ver la definición del operador.

```
operator(eval,prefijo).

operate(eval).

operate([eval,List,OldE,NewE],R) :-
    evaluate(OldE,NewE,List,R).

print_operator(eval) :-
    write("Eval").
```

Código 5.7: Definición del operador evalúa

Definirlo como operador primitivo permite que si el usuario crea un operador que necesite este mecanismo, bastará usar en su definición de operar una llamada al opera de evalúa. Para ejemplificar este punto se va a utilizar la definición de operar para el operador *igual*, al que se ha llamado *equal*, en el Código 5.8. Esta lógica ya se ha esbozado en la sección 5.2.2.

```
operate_equal([equal,A,B],NewOp,R) :-
    valid_list(NewOp),
    prepare_operation(NewOp,NewOpPrep),
    operate([eval,NewOpPrep,@,A], NewA),
    operate([eval,NewOpPrep,@,B], NewB),
    assign([equal,NewA,NewB],R).
```

Código 5.8: Definición de opera para el operador primitivo equal.

5.2.6. Definición de constantes simbólicas

Definir una constante simbólica es simple, solo se necesita detallar su nombre y la correspondencia entre dicho nombre y su valor numérico. Para tal fin el motor cuenta con dos predicados, uno para la definición y otro para la transformación entre ambos valores. El de definición, *symbolic_constant*, admite un parámetro que será el nombre con el que se utilice la constante en las expresiones. El usado para la transformación, *symbol_to_number* obtiene como parámetros el nombre y el valor, en dicho orden.

Para definir como constante simbólica el número 1 basta con definir los predicados como muestra el Código 5.9

```
symbolic_constant(one).
symbol_to_number(one,1).
```

Código 5.9: Definición del número uno como constante simbólica.

5.2.7. Operadores y constantes simbólicas predefinidos

Como se ha visto en los puntos anteriores el motor contará con unos operadores ya definidos. Del mismo modo, y según los requisitos, hay ciertos valores numéricos que deben tratarse como constantes simbólicas. En la Tabla 5.3 se detallan, en función de su simbología, tanto los operadores internos como las constantes.

Simbología	Significado
equal	Operador de igualdad para las ecuaciones
selection	Operador de selección
eval	Operador que sustituye unos valores por otros en una expresión
one	Constate simbólica para el número 1
zero	Constate simbólica para el número 0
minus_one	Constate simbólica para el número -1

Tabla 5.3: Operadores y constantes simbólicas predefinidos del motor.

5.2.8. Elemento neutro e inverso

El elemento neutro de una operación es aquel que hace que al operarlo con cualquier elemento *A*, el resultado sea ese mismo elemento *A*. Por ello, para definir este comportamiento en el motor, bastará con expresarlo como un caso particular del comportamiento de la propiedad operar. En el Código 5.10 se detalla el elemento neutro de la suma, el cero.

```
operate([suma,Elemento1,zero],Elemento1).
operate([suma,zero,Elemento2],Elemento2).
```

Código 5.10: Definición del elemento neutro del operador suma.

El elemento inverso u opuesto es aquel que al operarlo con un elemento *A* se obtiene el elemento neutro. De la misma forma que en el caso anterior, esta propiedad se expresará como un caso particular de la propiedad operar. No obstante, será necesario indicar la forma de dicho elemento que, generalmente, es una expresión que consta de un operador y sus elementos. Como detalla el Código 5.11 el producto del número -1 por un elemento *A* es el elemento inverso de *A* para la operación suma.

```
operate([suma,Elemento1,Elemento2],zero) :-
    inverse(suma,Elemento1,Elemento2).

inverse(suma,Elemento,[producto,minus_one,Elemento]).

inverse(suma,[producto,minus_one,Elemento],Elemento).
```

Código 5.11: Definición del elemento inverso del operador suma.

5.2.9. Valores por defecto

Es necesario que el sistema cuente con valores por defecto para todas aquellas características de un operador que debe definir el usuario. En la Tabla 5.4 se muestran dichos valores.

Características	**Valores por defecto**
Símbolo operador	*Nombre del operador*
Número de argumentos	n-ario
Representación para operadores unarios	prefijo
Representación para el resto de operadores	infijo
Aplica operar	false
Aplica conmutar	false
Aplica asociar	false
Aplica distribuir	false

Tabla 5.4: Valores por defecto para la características a definir por el usuario.

5.3. Interfaz de pruebas

Para facilitar el uso del motor en su desarrollo y sus pruebas manuales se ha creado una sencilla interfaz de pruebas. Permite que, mediante el uso del teclado, un usuario pueda crear expresiones y realizar manipulaciones. Para ello se utilizan herramientas de la shell de Linux y una conexión a Swi-Prolog, donde se lleva a cabo la ejecución de la lógica, presentándole al usuario el resultado por pantalla.

Para la interacción mediante el teclado se ha mapeado cada instrucción que se puede ejecutar en el motor con una tecla diferente. Dicha relación se puede ver en la Tabla 5.5, donde se separan las instrucciones de creación y las de manipulación.

Tecla	Instrucción	Tipo
o	Crear operación	Creación
e	Crear ecuación	
s	Crear sistema de ecuaciones	
a	Asociar	Manipulaciones
d	Disociar	
f	Factor común	
t	Distributiva	
c	Conmutar	
o	Operar expresión	
e	Operar ecuación	
s	Sustituir(Evalúa)	
q	Salir	Genérica

Tabla 5.5: Mapeo para la interfaz de pruebas entre las instrucciones y el teclado.

5.4. Funcionamiento interno del CAS

Para cerrar este capítulo se va a mostrar el funcionamiento interno del CAS de Chalkpy durante la ejecución de las manipulaciones necesarias para resolver una ecuación. Para poder llevar a cabo este ejemplo suponemos definido el fichero de usuario con el operador suma. En cada paso se detalla el predicado a aplicar y la expresión resultante. Además, en la Figura 5.6 se muestra la resolución de la ecuación mediante la interfaz de pruebas junto con el paso al que corresponde cada entrada entre corchetes. Para obtener más información sobre los predicados internos del motor de Chalkpy consultar el Anexo B.

Los pasos necesarios para resolver la ecuación $2+x=6$, agrupando las selecciones que se hagan seguidas en un único paso, son:

1. Crear la ecuación:

```
gb_create_equal([suma,2,x],6).    |    [selection,[equal,[suma,2,x],6]]
```

2. Introducir el -2 en ambos términos de la ecuación:

```
gb_operate_equal([suma,@,-2]). |  [selection,[equal,[suma,[suma,2,x],-2],[suma,6,-2]]]
```

3. Seleccionar el primer término de la ecuación:

```
        gb_sd.       |        [equal,[selection,[suma,[suma,2,x],-2]],[suma,6,-2]]
```

4. Disociar el primer término de la operación seleccionada:

```
     gb_dissociate(1).    |     [equal,[selection,[suma,2,x,-2]],[suma,6,-2]]
```

5. Conmutar el primer y segundo elemento del operador seleccionado:

```
     gb_commute(1,2).     |     [equal,[selection,[suma,x,2,-2]],[suma,6,-2]]
```

6. Asociar el segundo y tercer elemento del operador seleccionado:

```
     gb_associate(2,3).   |  [equal,[selection,[suma,x,[suma,2,-2]]],[suma,6,-2]]
```

7. Seleccionar el término $(2 + -2)$:

```
       gb_sd. , gb_sr.    |   [equal,[suma,x,[selection,[suma,2,-2]]],[suma,6,-2]]
```

8. Operar:

```
         gb_operate.      |      [equal,[suma,x,[selection,zero]],[suma,6,-2]]
```

9. Seleccionar el primer término de la ecuación:

```
           gb_su.         |       [equal,[selection,[suma,x,zero]],[suma,6,-2]]
```

10. Operar:

```
gb_operate.        |        [equal,[selection,x],[suma,6,-2]]
```

11. Seleccionar el segundo término de la ecuación:

```
gb_sr.             |        [equal,x,[selection,[suma,6,-2]]]
```

12. Operar:

```
gb_operate.        |        [equal,x,[selection,4]]
```

Finalmente se obtiene que la ecuación se cumple cuando la x toma el valor 4.

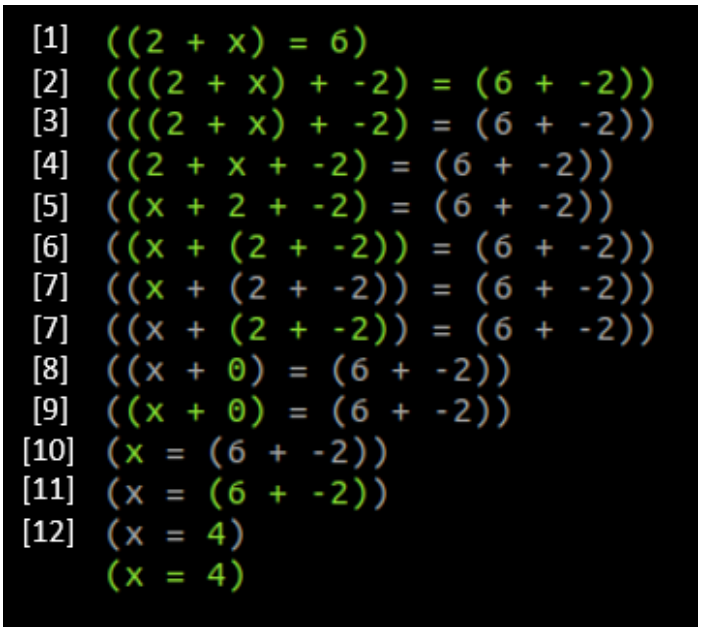

Figura 5.6: Resolución en la interfaz de la ecuación $2 + x = 6$

6

Pruebas

Para controlar los posibles errores que pueden surgir en las fases de desarrollo se han diseñado y ejecutado diversas pruebas con distintos objetivos. Gracias a la metodología utilizada, el código y las pruebas se han podido ir realizando casi simultáneamente, lo que ha hecho que los errores se pudieran solventar de manera ágil y no se propagasen por todo el código. En las siguientes secciones se explican dichas pruebas.

6.1. Inspección de código

Este tipo de prueba se ha llevado a cabo tras la finalización de un módulo o un incremento. Se ha examinado el código perteneciente al módulo o la iteración, así como cualquier otro que se hubiese modificado, principalmente en busca de errores. También se ha usado esta técnica para comprobar su estructura, calidad y complejidad, y llevar a cabo cualquier refactorización de código que fuese necesaria. De esta forma, este tipo de pruebas ha cumplido dos funcionalidades, la subsanación de errores y la mejora del código.

Un punto crítico donde fue muy útil este tipo de pruebas fue en la calidad del código. Ir desarrollando en incrementos hacía que en ocasiones se repitiesen bloques de códigos en diferentes predicados. Por lo que tras estas inspecciones se procedía a la refactorización para extraerlos en predicados usables por otros. Esto hizo que tras finalizar cada iteración no solo se inspeccionase el código creado o modificado en ella, sino el

generado en otras anteriores.

6.2. Caja blanca

Las pruebas de caja blanca se centran en el flujo del sistema en función de la lógica, es decir, en su funcionamiento interno. Se fundamenta en probar todos los posibles caminos que se pueden dar durante el uso del sistema buscando errores o probando los casos extremos.

Debido a que en cada incremento se añadían un número reducido de funciones estas pruebas se han llevado a cabo manualmente. Para ello se diseñó diferentes entradas y casos límites teniendo en cuenta sus flujos, y se ejecutaron desde el programa principal y desde la interfaz de pruebas. Para ayudar a seguir los flujos se introdujo la impresión de los predicados que se iban ejecutando.

6.3. Caja negra

Las pruebas de caja negra se enfocan en lo que el sistema debe hacer sin preocuparse del funcionamiento interno. Es decir, qué hace sin importar el cómo, dada una entrada comprobar que la salida generada sea la esperada. Para ello estas entradas y salidas deben estar perfectamente definidas con anterioridad a la ejecución de las pruebas.

Se encontró la necesidad de ejecutar estas pruebas varias veces a lo largo del desarrollo. Se pretendía comprobar que ninguna modificación había generado un error que pudiese comprometer el desarrollo de la nueva funcionalidad, y, por supuesto, durante la fase de pruebas. Por ello se tomó la decisión de automatizar la ejecución de estas pruebas.

6.3.1. Automatización

Para la automatización de las pruebas se ha decidido usar las herramientas de la shell de Linux junto al lenguaje AWK. Dado un fichero de entrada con instrucciones a ejecutar y las salidas esperadas, se procesa llevando a cabo la ejecución de los predicados en Swi-Prolog y comparando las salidas obtenidas con las salidas esperadas. A continuación se muestra el formato de los ficheros de entrada.

Entradas

Para la entrada se ha decidido usar un fichero en texto plano con un formato lo más simplificado posible. Cada línea representa una instrucción a ejecutar en el motor y, separado por un punto y coma, su salida esperada, precedida por la palabra *List:* , en el formato interno. Como norma general las pruebas se estructuraban en bloques, donde cada uno de ellos representaba una propiedad u operación. De esta forma se creaba una nueva expresión para cada bloque de instrucciones o cada combinación de elementos que se desease probar dentro de la misma funcionalidad.

En la Figura 6.1 podemos ver un ejemplo de uno de los ficheros utilizados para las pruebas del nuevo motor de Chalkpy, más concretamente, para la propiedad conmutativa y la operación de la suma.

```
#****************************************#
#                 CONMUTAR               #
#****************************************#
# Crear ecuacion
gb_create([suma,2,3,[suma,4,5]]);List: [selection,[suma,2,3,[suma,4,5]]]
#Conmutar elemento 1 y 2 de la seección
gb_commute(1,2);List: [selection,[suma,3,2,[suma,4,5]]]
#Conmutar con indice no existente
gb_commute(0,2);List: [selection,[suma,3,2,[suma,4,5]]]

#****************************************#
#                 OPERA SUMA             #
#****************************************#
# Crear ecuacion
gb_create([suma,10,5,2]);List: [selection,[suma,10,5,2]]
# Opera
gb_operate;List: [selection,17]
# Crear ecuacion
gb_create([suma,1,5]);List: [selection,[suma,one,5]]
# Opera
gb_operate;List: [selection,6]
```

Figura 6.1: Ejemplo de entrada para pruebas de caja negra

7

Resultados

Tras el desarrollo del Trabajo de Fin de Máster se ha obtenido un nuevo sistema de álgebra computacional que cumple con todos los objetivos marcados. Además, como se verá a continuación, se solventan los aspectos negativos que planteaba el primer CAS de Chalkpy: cuenta con una mayor extensibilidad, permite la definición de operaciones complejas y es posible aplicarlo en un ámbito diferente al de la matemática tradicional. También se ha conseguido definir una sencilla interfaz de pruebas, que ha sido un gran apoyo a la hora de interactuar con el motor.

Como prueba de concepto, para ilustrar las capacidades del motor algebraico, se han desarrollado ficheros de definiciones para cada uno de los siguientes elementos:

1. Las operaciones básicas de suma, resta y potencia. La resta se ha resuelto como la suma de números negativos y la división como una potencia elevada a -1.

2. Los operadores AND, OR y NOT del álgebra de Boole.

3. Los operadores logaritmo y exponencial.

4. Derivadas simples.

5. El operador sumatorio.

6. Sistemas lineales de dos ecuaciones con dos incógnitas.

En esta sección se van a mostrar los ficheros de definiciones junto a un sencillo ejemplo de resolución para los casos correspondientes al Álgebra de Boole, el sumatorio y los sistemas de ecuaciones. Nótese que el punto relativo a los sistemas de ecuaciones no se contemplaba en los objetivos iniciales, no obstante, a lo largo del desarrollo del motor se vio la posibilidad de incorporarlo como un operador, esto se verá en la sección 7.3.

Si se desea consultar todos los ficheros de usuario definidos se encuentran en el Anexo A.

7.1. Álgebra de Boole

Para poder extender el motor al Álgebra de Boole es necesario definir los operadores AND, OR y NOT, junto a su comportamiento básico. Estas definiciones se pueden consultar en el Código 7.1. Como se puede comprobar son muy similares a las definiciones de la operación suma mostradas a lo largo de la sección 5.2. Esto hizo que, al haber generalizado la lógica del motor, la extensión a este ámbito fuera casi instantánea.

```
/*
-- Definición de los operadores
*/
operator(oor).
operator(aand).
operator(nnot).

/*
-- Número de argumentos
*/
n_ary_operator(oor).
n_ary_operator(aand).
unary_operator(nnot).

/*
-- Predicados para indicar la conmutativa a los operadores.
*/
commute(oor).
commute(aand).

/*
-- Predicados para indicar la asociatividad de operadores.
*/
associate(oor).
associate(aand).
```

```
/*
-- Predicados para indicar la distributiva.
*/
distribute(aand,oor).
distribute(oor,aand).

/*
-- Predicados para indicar que se pueden operar los operadores.
*/
operate(oor).
operate(aand).
operate(nnot).

/*
-- Define la representación
*/
print_operator(oor) :-
    write(" || ").
print_operator(aand) :-
    write(" && ").
print_operator(nnot) :-
    write("!").

/*
-- Define las constantes simbólicas
*/
symbolic_constant(ttrue).
symbolic_constant(ffalse).

symbol_to_number(ttrue,t).
symbol_to_number(ffalse,f).

/*
-- Predicados para operar el NOT.
*/
operate([nnot,ttrue],ffalse).
operate([nnot,ffalse],ttrue).
operate([nnot,[nnot,Elemento]],Elemento).

/*
-- Predicados para operar el OR.
*/
operate([oor,_,ttrue],ttrue).
operate([oor,ttrue,_],ttrue).
operate([oor,Elemento1,ffalse],Elemento1).
operate([oor,ffalse,Elemento1],Elemento1).
```

```
/*
-- Predicados para operar el AND.
*/
operate([aand,Elemento1,ttrue],Elemento1).
operate([aand,ttrue,Elemento1],Elemento1).
operate([aand,ffalse,_],ffalse).
operate([aand,_,ffalse],ffalse).
```

Código 7.1: Definición del fichero de usuario para el Álgebra de Boole.

En la Figura 7.1 se muestra el proceso de la expresión $(T||F||!F)\&\&!!F$ durante los pasos que se dan para su resolución, usando para mostrarlo la interfaz de pruebas. Solo se han usado predicados de selección, asociación y operación.

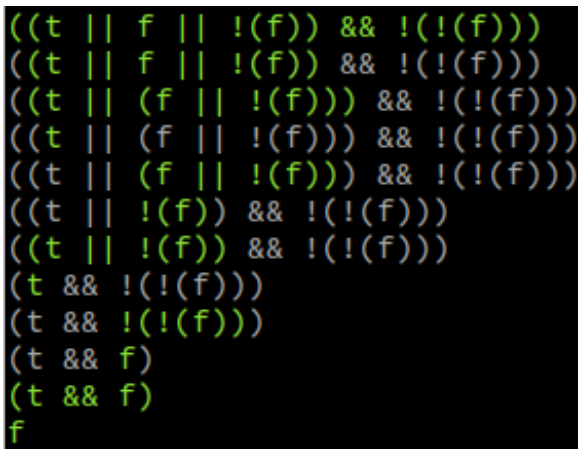

Figura 7.1: Resolución de una expresión perteneciente al Álgebra de Boole.

7.2. Sumatorio

Se ha definido el sumatorio como un operador con cuatro argumentos: el índice de la suma, el valor inicial de índice, el valor final del índice y la suma. Al tener este número fijo de argumentos, y necesitar que los valores inicial y final de índice cumplan ciertas condiciones, ha sido necesario definir una validación propia. Aunque lo más interesante de este operado es, quizás, su definición para operar. Esto se debe a que emula el desarrollo de un sumatorio paso a paso y, para ello, ha necesitado usar el operador evalúa. Para agilizar la resolución, también se ha definido la fórmula para obtener la suma de los primeros *N* números y la suma de sumatorios con índices iguales. Todo esto se puede observar en el Código 7.2.

```
operator(sumatory,prefijo).

valid_list([sumatory,Ind,Ini,Fin,Sumando]) :-
    atom(Ind),
    number(Ini),
    number(Fin),
    =<(Ini, Fin),
    valid_list(Sumando).

operate([suma,[sumatory,Ind,Ini,Fin,Sumando],[sumatory,Ind,Ini,Fin,Sumando2]],
        [sumatory,Ind,Ini,Fin,[suma,Sumando,Sumando2]]).

operate([sumatory,Ind,one,Fin,Ind],[producto,[producto,Fin,[suma,Fin,1]],[pow,2,minus_one]]).

operate([sumatory,Ind,Ini,Fin,Sumando], R) :-
    Ini==Fin,
    operate([eval,Sumando,Ind,Ini],R).

operate([sumatory,Ind,Ini,Fin,Sumando],[suma,R,[sumatory,Ind,NewInd,Fin,Sumando]]) :-
    operate([eval,Sumando,Ind,Ini],R),
    NewInd is Ini+1.

print_operator(sumatory) :-
    write("SUM").
```

Código 7.2: Definición del fichero de usuario para el sumatorio

Como ejemplo se puede observar cómo se ha resuelto el sumatorio $\sum_{i=4}^{6} 2^i$ en la Tabla 7.1. Muestra los predicados a aplicar, el resultado de aplicarlos y la selección en tono verde.

Predicado	Resultado
gb_create([sumatory,i,4,6,[pow,2,i]])	**SUM(i,4,6,(2^i))**
gb_operate	**((2^4) + SUM(i,5,6,(2^i)))**
gb_sd	((**2^4**) + SUM(i,5,6,(2^i)))
gb_operate	(**16** + SUM(i,5,6,(2^i)))
gb_sr	(16 + **SUM(i,5,6,(2^i))**)
gb_operate	(16 + **((2^5) + SUM(i,6,6,(2^i)))**)
gb_sd	(16 + (**(2^5)** + SUM(i,6,6,(2^i))))
gb_operate	(16 + (**32** + SUM(i,6,6,(2^i))))
gb_sr	(16 + (32 + **SUM(i,6,6,(2^i))**))
gb_operate	(16 + (32 + **(2^6)**))
gb_operate	(16 + (32 + **64**))
gb_su	(16 + **(32 + 64)**)
gb_operate	(16 + **96**)
gb_su	**(16 + 96)**
gb_operate	**112**

Tabla 7.1: Resolución del sumatorio $\sum_{i=4}^{6} 2^i$.

7.3. Sistemas de dos ecuaciones

Similar a la lógica que se siguió para crear el operador ecuación, podemos definir los sistemas como un operador, no operable, cuyos argumentos son las dos ecuaciones que lo forman. Por tanto, solo se necesita definir el nombre y simbología del operador, y llevar a cabo la validación de los argumentos como se muestra en el Código 7.3. El resto de la lógica pertenece a la operativa de ecuación y evaluar. No obstante, aunque teníamos definido el operador evaluar y su lógica nos permite hacer sustituciones en listas, no existía un método que lo aplicase a una expresión seleccionada. Por ello se definió el predicado *gb_evaluate(OldEl, NewEl)* que sustituye, en la lista seleccionada, el elemento *OldEl* por *NewEl*.

```
operator(sistema_eq,infijo).

valid_list([sistema_eq, [equal|Elemento1], [equal|Elemento2]]):-
    valid_list([equal|Elemento1]),
    valid_list([equal|Elemento2]).

print_operador(sistema_eq):-
    write(" ----- ").
```

Código 7.3: Definición del fichero de usuario para los sistemas lineales de dos ecuaciones.

A continuación se va a resolver paso a paso el sistema de ecuaciones:

$$\left.\begin{array}{rcl} 2x + 3y & = & 5 \\ x - y & = & 0 \end{array}\right\}$$

En casa paso se va a mostrar el predicado que se debe ejecutar, el resultado de su ejecución y la selección, en verde, en cada momento. Al igual que en ejemplos anteriores se agruparán en un único paso la aplicación de ciertos predicados.

1. Crear el sistema de ecuaciones.

```
gb_create_sistema([suma,[producto,2,x],[producto,3,y]],5,[suma,x,[producto,-1,y]],0)
```

((((2 * x) + (3 * y)) = 5) —— ((x + (-1 * y)) = 0))

2. Seleccionar la segunda ecuación.

```
gb_sd
gb_sr
```

((((2 * x) + (3 * y)) = 5) —— **((x + (-1 * y)) = 0)**)

3. Introducir la suma de y en ambos términos de la segunda ecuación.

```
gb_operate_equal([suma,@,y])
```

(((2 * x) + (3 * y)) = 5) —— **(((x + (-1 * y)) + y) = (0 + y))**)

4. Seleccionar el primer término de la segunda ecuación.

```
gb_sd
```

(((2 * x) + (3 * y)) = 5) —— (**((x + (-1 * y)) + y)** = (0 + y)))

5. Disociar el primer elemento de la selección.

```
gb_dissociate(1)
```

(((2 * x) + (3 * y)) = 5) —— (**(x + (-1 * y) + y)** = (0 + y)))

6. Asociar el segundo y tercer elemento de la selección.

```
gb_associate(2,3)
```

(((2 * x) + (3 * y)) = 5) —— (**(x + ((-1 * y) + y))** = (0 + y)))

7. Seleccionar el segundo miembro de la selección.

```
gb_sd
gb_sr
```

(((2 * x) + (3 * y)) = 5) —— ((x + **((-1 * y) + y)**) = (0 + y)))

8. Operar la selección.

```
gb_operate
```

(((2 * x) + (3 * y)) = 5) —— ((x + **0**) = (0 + y)))

9. Seleccionar y operar el primer término de la segunda ecuación

```
gb_su
gb_operate
```

((((2 * x) + (3 * y)) = 5) —— (**x** = (0 + y)))

10. Seleccionar y operar el segundo término de la segunda ecuación

```
gb_sr
gb_operate
```

((((2 * x) + (3 * y)) = 5) —— (x = **y**))

11. Seleccionar la primera ecuación y, conocido ya el valor de y, sustituir y por x.

```
gb_sl
gb_evaluate(y,x)
```

(**(((2 * x) + (3 * x)) = 5)** —— (x = y))

12. Seleccionar y sacar factor común al primer término de la primera ecuación. Pero, para ello, primero es necesario que el factor sea el primer elemento de cada término interno.

```
gb_sd
gb_sd
gb_sr
gb_commute(1,2)
```

((((2 * x) + **(x * 3)**) = 5) —— (x = y))

```
gb_sl
gb_commute(1,2)
```

(**(((x * 2)** + (x * 3)) = 5) —— (x = y))

13. Seleccionar el primer término de de la primera ecuación para sacar el factor común.

```
gb_su
gb_common_factor
```

$$(((\mathbf{x * (2 + 3)}) = 5) \longrightarrow (x = y))$$

14. Seleccionar y operar el segundo elemento seleccionado.

```
gb_sd
gb_sr
gb_operate
```

$$(((x * \mathbf{5}) = 5) \longrightarrow (x = y))$$

15. Seleccionar la primera ecuación y dividir por cinco, en ambos lados, usando la potencia.

```
gb_su
gb_su
gb_operate_equal([producto,@,[pow,5,minus_one]])
```

$$(\mathbf{(((x * 5) * (5^{-1})) = (5 * (5^{-1})))} \longrightarrow (x = y))$$

16. Asociar el cinco y su inverso del primer término.

```
gb_sd
gb_dissociate(1)
gb_associate(2,3)
```

$$(((x * \mathbf{(5 * (5^{-1}))}) = (5 * (5^{-1}))) \longrightarrow (x = y))$$

17. Operar el término seleccionado.

```
gb_operate
```

(((x * **1**) = (5 * (5^{-1}))) ⟶ (x = y))

18. Operar primer término de la primera ecuación.

```
gb_su
gb_operate
```

((**x** = (5 * (5^{-1}))) ⟶ (x = y))

19. Operar segundo término de la primera ecuación.

```
gb_sr
gb_operate
```

((x = **1**) ⟶ (x = y))

Finalmente hemos obtenido los valores para que se cumpla el sistema, las variables x e y deben tomar el valor 1.

8

Conclusiones y Trabajo Futuro

Tras la realización del Trabajo de Fin de Máster se ha obtenido un sistema de álgebra computacional para la versión 3.0 de Chalkpy. Permite la definición de operadores con diferentes niveles de complejidad, su uso en expresiones y ecuaciones y, ante todo, se ha conseguido una amplia extensibilidad del motor.

Se ha conseguido desarrollar la definición de expresiones algebraicas y de ecuaciones con variables, números y constantes simbólicas. En la definición de las propiedades matemáticas básicas se ha logrado una lógica generalizada que ha permitido su aplicación a cualquier operador e incluso a ámbitos diferenciados de la matemática tradicional. Estas propiedades y las definidas por el usuario han permitido manipular y transformar tanto las expresiones algebraicas como las ecuaciones.

Se han realizado pruebas con distintos ficheros de definiciones, generados en distintas etapas del proyecto conforme el motor iba creciendo: las operaciones básicas de suma, producto y potencia; el logaritmo y la exponencial; las derivadas básicas; los operadores AND, OR y NOT junto a sus constantes simbólicas True y False del Álgebra de Boole; el sumatorio; y sistemas ecuaciones. Cabe destacar que este último tipo de operador no se definía como un objetivo ni una prueba de concepto. Durante el desarrollo del motor, debido a las facilidades que planteaba éste para definir diversos operadores, se evaluó su posible definición consiguiendo poder resolver sistemas lineales de dos ecuaciones con dos incógnitas.

Se puede concluir que el proyecto ha sido exitoso al haber alcanzado todos los ob-

jetivos planteados, tanto parciales como globales. También se han superado las pruebas de concepto propuestas, llegando incluso ha introducir operaciones no planificadas y sin necesidad de modificar la lógica del motor.

A pesar de haber cumplido con los objetivos marcados, han surgido líneas de trabajo que permitirían la extensión del proyecto en un futuro.

La interfaz de pruebas es válida como apoyo para llevar a cabo pruebas de desarrollo de una manera rápida y más cómoda que por código. No obstante, sería necesario crear una interfaz donde un usuario final pudiese interactuar de una manera clara, cómoda y sencilla.

Como se ha señalado el nuevo motor nos permite resolver sistemas de dos ecuaciones. No obstante, la interacción con el motor para llevar a cabo las sustituciones puede resultar problemática debiendo introducir a mano el usuario valores que ya son términos del sistema. Se propone buscar una manera de realizar las sustituciones de valores pertenecientes a las ecuaciones que conforman el sistema de manera automática.

Se plantea finalmente la posibilidad de introducir aprendizaje automático para conseguir que el motor realice por cuenta propia las manipulaciones irrelevantes y asiduas en la resolución. Este podría ser un primer paso por si se decidiese plantear que Chalkpy contase con una resolución automatizada.

Bibliografía

[1] B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, *Computer Algebra: Symbolic and Algebraic Computation*. Springer-Verlag,, 2012.

[2] "Chalkpy: Una herramienta para mejorar la presentación de contenidos matemáticos en el aula." Convocatoria de Proyectos de Innovación Docente-UAM, 2014-2015.

[3] G. Sandoval, "Herramienta para mejorar la presentación de contenidos matemáticos en el aula," 2016.

[4] "Sympy." Available: `https://www.sympy.org/es/`, Último acceso: 2018.

[5] "Chalkpy2.0: Extensión de la herramienta chalkpy para la presentación de contenidos matemáticos mediante dispositivos móviles." Convocatoria de Proyectos de Innovación Docente-UAM, 2015-2016.

[6] L. Salcedo, "Desarrollo de un motor de matemática simbólica para la herramienta chalkpy," 2016.

[7] U. C. Merzbach and C. B. Boyer, *A History of Mathematics*. Wiley, 1991.

[8] D. J. Struik, *A Concise History of Mathematics*. Wiley, 1991.

[9] D. Hilbert and W. Ackermann, *Principles of Mathematical Logic*. AMS Chelsea Publishing, 1950.

[10] K. Gödel, "*Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I.*" Monatshefte für Mathematik und Physik, vol. 38, nº1, pp. 173-198, 1931.

[11] A. Church, "*A Set of Postulates for the Foundation of Logic*." Annals of Mathematics, vol. 33, nº2, pp. 346-366, 1932.

[12] A. M. Turing, "*On Computable Numbers, with an Application to the Entscheidungsproblem*." Proceeding of the London Mathematical Society, vol. 43, nº2, pp. 230-265, 1936.

[13] M. Veltman, "*Schoonschip a program for symbol handling*." Available: `http://inspirehep.net/record/1685521/files/schipman.pdf?version=1`, 1991.

[14] "Docon." Available: `https://github.com/athanclark/DoCon`, Último acceso: 2018.

[15] "Reduce." Available: `https://reduce-algebra.sourceforge.io/`, Último acceso: 2018.

[16] "Prolog equation solving system." Available: `https://github.com/maths/PRESS`, Último acceso: 2018.

[17] "Macsyma." Available: `http://maxima.sourceforge.net/`, Último acceso: 2018.

[18] "Maple." Available: `https://www.maplesoft.com/`, Último acceso: 2018.

[19] "Mathematica." Available: `https://www.wolfram.com/mathematica/`, Último acceso: 2018.

[20] "Matlab." Available: `https://www.mathworks.com/products/matlab.html`, Último acceso: 2018.

[21] "Wolframalpha." Available: `http://www.wolframalpha.com/`, Último acceso: 2018.

[22] "Yacas." Available: `http://www.yacas.org/`, Último acceso: 2018.

[23] I. Sommerville, *Ingeniería del Software*. Pearson Educación, 2005.

[24] "Prolog." Available: `https://es.wikipedia.org/wiki/Prolog`, Último acceso: 2018.

[25] "Awk." Available: `https://es.wikipedia.org/wiki/AWK`, Último acceso: 2018.

[26] "Latex." Available: `https://www.latex-project.org/`, Último acceso: 2018.

[27] "Vim." Available: `https://www.vim.org/`, Último acceso: 2018.

[28] "Swi-prolog." Available: `http://www.swi-prolog.org/`, Último acceso: 2018.

[29] "Bitbucket." Available: `https://bitbucket.org`, Último acceso: 2018.

[30] “Mercurial.” Available: https://www.mercurial-scm.org/, Último acceso: 2018.

[31] “Git.” Available: https://git-scm.com/, Último acceso: 2018.

Glosario

API La Interfaz de Programación de Aplicaciones, del inglés ***A****pplication* ***P****rogramming* ***I****nterface*, es un conjunto de funcionalidades que ofrece una biblioteca para ser utilizado como una capa de abstracción.

axioma Proposición tan clara y evidente que se admite sin demostración.

ideograma Imagen o símbolo que denota un concepto o idea.

refactorización de código Técnica de Ingeniería del Software para la reestructuración del código fuente que altera su estructura pero no su comportamiento.

shell Intérprete de comandos de los sistemas UNIX.

sistema de álgebra computacional Conocido como CAS, del inglés *Computer Algebra System*, es un programa de cálculo que facilita la resolución de ecuaciones y el trabajo con fórmulas simbólicas. Se basa en la manipulación de expresiones simbólicas y numéricas mediante el uso de reglas definidas.

teorema Proposición demostrable lógicamente partiendo de axiomas, postulados o de otras proposiciones demostradas.

término Producto de factores representado por números o variables literales.

Acrónimos

CAS Computer Algebra System.

MIT Massachusetts Institute of Tecnology.

YACAS Yet Another Computer Algebra System.

Anexos

A

Ficheros de usuarios

En este Anexo se van a mostrar los ficheros de definición de usuario que se han obtenido al desarrollar el proyecto.

Operadores básicos

El Código A.1 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para las operaciones básicas de suma y producto.

```
/*
-- Definición de operadores.
*/
operator(suma).
operator(producto).

/*
-- Definición argumentos
*/
n_ary_operator(suma).
n_ary_operator(producto).

/*
-- Predicados para imprimir operadores
*/
```

```
print_operator(suma) :-
    write(" + ").
print_operator(producto) :-
    write(" * ").

/*
-- Predicados para indicar que se pueden operar la suma y el producto.
*/
operate(suma).
operate(producto).

/*
-- Predicados para indicar la conmutatividad de operadores.
*/
commute(suma).
commute(producto).

/*
-- Predicados para indicar la asociatividad de operadores.
*/
associate(suma).
associate(producto).

/*
-- Predicados para indicar el factor común de dos operadores.
*/
distribute(producto,suma).

/*
-- Predicados para definir el inverso de la suma
*/
operate([suma,Elemento1,Elemento2],zero) :-
    inverse(suma,Elemento1,Elemento2).

inverse(suma,Elemento,[producto,minus_one,Elemento]).
inverse(suma,[producto,minus_one,Elemento],Elemento).

/*
-- Predicados para definir el inverso del producto.
-- Se complementa en el fichero de la potencia.
*/
operate([producto,Elemento1,Elemento2],one) :-
    inverse(producto,Elemento1,Elemento2).

/*
-- Predicados para definir como operar la suma.
*/
operate([suma,Elemento1,zero],Elemento1).
```

```
operate([suma,zero,Elemento2],Elemento2).

operate([suma,Elemento1,Elemento2],R) :-
    number(Elemento1),
    number(Elemento2),
    R is Elemento1+Elemento2.

/*
-- Predicados para definir como operar la suma con constante
*/
operate([suma,Elemento1,one],R) :-
    number(Elemento1),
    R is Elemento1+1.

operate([suma,one,Elemento2],R) :-
    number(Elemento2),
    R is Elemento2+1.

operate([suma,one,one],2).

operate([suma,Elemento1,minus_one],R) :-
    number(Elemento1),
    R is Elemento1-1.

operate([suma,minus_one,Elemento2],R) :-
    number(Elemento2),
    R is Elemento2-1.

operate([suma,one,minus_one],zero).

operate([suma,minus_one,one],zero).

/*
-- Predicados para definir como operar el producto.
*/

operate([producto,Elemento1,one],Elemento1).

operate([producto,one,Elemento2],Elemento2).

operate([producto,Elemento1,Elemento2],R) :-
    number(Elemento1),
    number(Elemento2),
    R is Elemento1*Elemento2.
```

```
/*
-- Predicados para definir como operar el producto con constante
*/
operate([producto,_,zero],zero).
operate([producto,zero,_],zero).

operate([producto,Ele,minus_one],R) :-
    number(Ele),
    R is Ele * -1.
```

Código A.1: Definición del fichero de usuario básico.

Potencia

El Código A.2 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para la potencia.

```
/*
-- Definición de operadores.
*/
operator(pow).

/*
-- Definición argumentos
*/
binary_operator(pow).

/*
-- Predicados para imprimir operadores
*/
print_operator(pow) :-
    write("^").

/*
-- Predicados para indicar que se pueden operar los operadores.
*/
operate(pow).

/*
-- Predicados para definir como operar la potencia.
*/
operate([pow,zero,_],zero).
```

```
operate([pow,Elemento1,minus_one],R) :-
    number(Elemento1),
    R is 1**Elemento1.

/*
-- Predicados para definir como operar la potencia con constantes.
*/
operate([pow,zero,_],zero).

operate([pow,Elemento1,minus_one],R) :-
    number(Elemento1),
    R is 1**Elemento1.

operate([pow,Elemento1,one],Elemento1).

operate([pow,Elemento1,Elemento2],R) :-
    number(Elemento1),
    number(Elemento2),
    R is Elemento1**Elemento2.

operate([pow,_,zero],one).

/*
-- Predicados para definir el inverso del producto
-- en función de la potencia.
*/

inverse(producto,Elemento1,[pow,Elemento1,minus_one]).

inverse(producto,[pow,Elemento1,minus_one],Elemento1).
```

Código A.2: Definición del fichero de usuario para la potencia.

Álgebra de Boole

El Código A.3 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para el Ágebra de Boole.

```
/*
-- Definición de los operadores
*/
operator(oor).
operator(aand).
operator(nnot).
```

```
/*
-- Lista de operadores válidos.
*/
n_ary_operator(oor).
n_ary_operator(aand).
unary_operator(nnot).

/*
-- Predicados para indicar la conmutatividad de operadores.
*/
commute(oor).
associate(aand).

/*
-- Predicados para indicar la asociatividad de operadores.
*/
associate(oor).
associate(aand).

/*
-- Predicados para indicar la distributiva y el factor común de dos operadores.
*/
distribute(aand,oor).
distribute(oor,aand).

/*
-- Predicados para indicar que se pueden operar dos operadores.
*/
operate(oor).
operate(aand).
operate(nnot).

/* Define la representación */
print_operator(oor) :-
    write(" || ").
print_operator(aand) :-
    write(" && ").
print_operator(nnot) :-
    write("!").

/* Define las constantes simbólicas */
symbolic_constant(ttrue).
symbolic_constant(ffalse).

symbol_to_number(ttrue,t).
symbol_to_number(ffalse,f).
```

```
/*
-- Predicados para operar el NOT.
*/
operate([nnot,ttrue],ffalse).
operate([nnot,ffalse],ttrue).
operate([nnot,[nnot,Elemento]],Elemento).

/*
-- Predicados para operar el OR.
*/
operate([oor,_,ttrue],ttrue).
operate([oor,ttrue,_],ttrue).
operate([oor,Elemento1,ffalse],Elemento1).
operate([oor,ffalse,Elemento1],Elemento1).

/*
-- Predicados para operar el AND.
*/
operate([aand,Elemento1,ttrue],Elemento1).
operate([aand,ttrue,Elemento1],Elemento1).
operate([aand,ffalse,_],ffalse).
operate([aand,_,ffalse],ffalse).
```

Código A.3: Definición del fichero de usuario para el Álgebra de Boole.

Logaritmo y exponencial

El Código A.4 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para el logaritmo y la exponencial.

```
/*
-- Predicado para definir el operador
*/
operator(log,prefijo).
operator(ln).
operator(exp).

/*
-- Lista de operadores válidos.
*/
unary_operator(ln).
unary_operator(exp).

/*
```

```
-- Predicados para indicar que se pueden operar.
*/
operate(log).
operate(exp).
operate(ln).

/*
-- Predicados para operar la exponencial.
*/
operate([exp,Elemento],R) :-
    number(Elemento),
    R is exp(Elemento).
operate([exp,Elemento],[exp,Elemento]).

/*
-- Predicados para operar el logaritmo.
*/
operate([log,_,one],zero).
operate([log,Base,Base],one).
operate([log,Base,[pow,Base,Elemento]],Elemento).
operate([log,Base,[producto,Elemento1,Elemento2]],
        [suma,[log,Base,Elemento1],[log,Base,Elemento2]]).
operate([suma,[log,Base,Elemento1],[log,Base,Elemento2]],
      [log,Base,[producto,Elemento1,Elemento2]]).
operate([log,Base,[pow,Elemento1,Elemento2]],[producto,Elemento2,[log,Base,Elemento1]]).
operate([producto,Elemento2,[log,Base,Elemento1]],[log,Base,[pow,Elemento1,Elemento2]]).
operate([log,Base,Elemento],[log,Base,Elemento]).

/*
-- Predicados para operar el logaritmo neperiano.
*/
operate([ln,[exp,Elemento]],Elemento).
operate([ln,one],zero).
operate([ln,Elemento],R) :-
    number(Elemento),
    R is log(Elemento).

operate([ln,[producto,Elemento1,Elemento2]],[suma,[ln,Elemento1],[ln,Elemento2]]).
operate([suma,[ln,Elemento1],[ln,Elemento2]],[ln,[producto,Elemento1,Elemento2]]).
operate([ln,[pow,Elemento1,Elemento2]],[producto,Elemento2,[ln,Elemento1]]).
operate([producto,Elemento2,[ln,Elemento1]],[ln,[pow,Elemento1,Elemento2]]).
operate([ln,Elemento],[ln,Elemento]).

/*
-- Predicados para imprimir operadores
*/
print_operator(ln) :-
    write("ln").
print_operator(log) :-
```

```
    write("log").
print_operator(exp) :-
    write("exp").
```

Código A.4: Definición del fichero de usuario para el logaritmo y la exponencial.

Derivadas

El Código A.5 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para las derivadas simples.

```
/*
-- Definición de operador
*/
operator(deriv,prefijo).

/*
-- Definición de argumentos
*/
binary_operator(deriv).

/*
-- Predicado para indicar que se puede operar
*/
operate(deriv).

/*
-- Predicados para operar las derivadas.
*/
operate([deriv, X, X], one).
operate([deriv, X, Y], zero):-
    atom(X) ; number(X).
operate([deriv, [suma, A, B], X], [suma, [deriv, A, X], [deriv, B, X]]).
operate([deriv, [producto, A, B], X],
        [suma, [producto, [deriv, A, X], B], [producto, A, [deriv, B, X]]]).
operate([deriv, [pow, X, B], X],
        [producto, B, [pow, X, [suma, B, minus_one]]]).
print_operator(deriv) :-
    write("DERIV").
```

Código A.5: Definición del fichero de usuario para las derivadas.

Sumatorio

El Código A.6 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para el sumatorio.

```
/*
-- Definición de operador.
*/
operator(sumatory,prefijo).

/*
-- Definición de argumentos.
*/
valid_list([sumatory,Ind,Ini,Fin,Sumando]) :-
    atom(Ind),
    number(Ini),
    number(Fin),
    =<(Ini, Fin),
    valid_list(Sumando).

/*
-- Predicados para indicar que se puede operar el sumatorio.
*/
operate(summatory).

/*
-- Predicados para operar el sumatorio.
*/
operate([suma,[sumatory,Ind,Ini,Fin,Sumando],[sumatory,Ind,Ini,Fin,Sumando2]],
        [sumatory,Ind,Ini,Fin,[suma,Sumando,Sumando2]]).

operate([sumatory,Ind,one,Fin,Ind],[producto,[producto,Fin,[suma,Fin,1]],[pow,2,minus_one]]).

operate([sumatory,Ind,Ini,Fin,Sumando], R) :-
    Ini==Fin,
    operate([eval,Sumando,Ind,Ini],R).
operate([sumatory,Ind,Ini,Fin,Sumando],[suma,R,[sumatory,Ind,NewInd,Fin,Sumando]]) :-
    operate([eval,Sumando,Ind,Ini],R),
    operate([suma,Ini,1],NewInd).

/*
-- Predicados para imprimir el operador.
*/
print_operator(sumatory) :-
    write("SUM").
```

Código A.6: Definición del fichero de usuario para el sumatorio

Sistemas de dos ecuaciones

El Código A.7 muestra el fichero de usuario donde se definen los operadores y propiedades necesarias para un sistema de dos ecuaciones.

```
/*
-- Definición del operador.
*/
operator(sistema_eq,infijo).

/*
-- Definición de argumentos.
*/
valid_list([sistema_eq, [equal|Elemento1], [equal|Elemento2]]):-
    valid_list([equal|Elemento1]),
    valid_list([equal|Elemento2]).

/*
-- Predicado para imprimir el operador..
*/
print_operator(sistema_eq):-
    write(" ----- ").
```

Código A.7: Definición del fichero de usuario para los sistemas de dos ecuaciones.

B

Principales predicados del motor de Chalkpy

Este Anexo se centra en dar una visión de la lógica interna del motor detallando los predicados más importantes, separados por los módulos que componen el proyecto. Sin embargo, no se mostrará ningún predicado del Módulo de usuario debido a que dicha lógica no viene predefinida, es el usuario quien debe detallarla. En el Anexo A se encuentran diversos ejemplos de los ficheros que podría contener este módulo.

Módulo de operación

En la Tabla B.1 se encuentran los predicados más importantes de este módulo junto a una breve descripción de su cometido.

Módulo de validación

En la Tabla B.2 se encuentran los predicados más importantes de este módulo junto a una breve descripción de su cometido.

Predicado	Acción
gb_create(List)	Valida la lista en función de la definición de argumentos de un operador. Si es correcto sustituye, para los que aplique, los valores numéricos por constantes simbólicas. Finalmente, guarda la expresión en una variable con el predicado de Prolog *nb_setval(Name,Value)*. Se seleccionará toda la expresión.
gb_create_selec(List)	Misma lógica que el predicado anterior permitiendo que la lista que se recibe contenga el operador *selec*.
gb_create_equation(LeftList,RightList)	Trata cada lista como el primer predicado. Antes de guardar la expresión resultante genera una nueva lista con el operador equal.
evaluate(OldE,NewE,List,Resultado)	Sustituye OldE por NewE en la lista L devolviendo la nueva lista en Resultado.
gb_sd()	Selecciona el primer elemento hijo de la selección actual, si es posible, en la expresión de trabajo.
gb_su()	Selecciona el elemento padre de la selección actual, si es posible, en la expresión de trabajo.
gb_sr()	Selecciona el elemento situado a la derecha de la selección actual, si es posible, en la expresión de trabajo.
gb_sl()	Selecciona el elemento situado a la izquierda de la selección actual, si es posible, en la expresión de trabajo.

Tabla B.1: Predicados principales del módulo de operación de Chalkpy.

Predicado	Acción
symbolic_constant(zero)	Define la constante simbólica para el 0.
symbol_to_number(zero,0)	Transformación del número 0 entre su valor simbólico y el numérico.
symbolic_constant(one)	Define la constante simbólica para el 1.
symbol_to_number(one,1)	Transformación del número 1 entre su valor simbólico y el numérico.
symbolic_constant(minus_one)	Define la constante simbólica para el -1.
symbol_to_number(minus_one,-1)	Transformación del número -1 entre su valor simbólico y el numérico.
unary_operator(selection)	Define el operador de seleccion.
valid_list(List)	Comprueba si el operador de la lista está definido, si cumple con el número de argumentos y que cada uno de los argumentos es válido. En caso de tener argumentos de tipo lista utiliza la recursión para validarlas.
valid_operate_pairs(Operator)	Comprueba que el operador Operator pueda operarse por pares. Es decir, sea n-ario, aplique la asociativa y aplique la conmutativa.

Tabla B.2: Predicados principales del módulo de validación de Chalkpy.

Módulo principal

En la Tabla B.3 se encuentran los predicados más importantes de este módulo junto a una breve descripción de su cometido. Como ya se ha dicho en el documento este módulo podría funciona como API en un futuro. Todos estos predicados ejecutan sus acciones sobre la operación seleccionada.

Módulo de aplicación

En la Tabla B.4 se encuentran los predicados más importantes de este módulo junto a una breve descripción de su cometido.

Predicado	Acción
gb_commute(Start,End)	Solicita aplicar la propiedad conmutativa desde el elemento en la posicion Start hasta el elemento en la posicion End.
gb_associative(Start,End)	Solicita aplicar la propiedad asociativa desde el elemento en la posición Start hasta el elemento en la posición End..
gb_dissociate(Position)	Solicita disociar el término en la posición Position..
gb_distribute(Position)	Solicita aplicar la propiedad distributiva sobre el elemento en la posición Position.
gb_commo_factor	Solicita aplicar el factor común.
gb_operate	Solicita aplicar la propiedad operar.
gb_operate_equal	Solitica aplicar la propiedad operar de una ecuación.
gb_evaluate	Solicita sustituir todas las apariciones de un elemento en una ecuación.

Tabla B.3: Predicados principales del módulo principal de Chalkpy.

Predicado	Acción
gb_apply_selection(Property)	Extrae la expresión de trabajo de la variable global, y hace la llamada a Property para que la ejecute sobre la selección. Guarda el resultado en la variable global.
gb_apply_start_end(Start,End,Property)	Extrae la expresión de trabajo de la variable global, y hace la llamada a Property para que la ejecute sobre los elementos entre el índice Star y el End de la selección. Guarda el resultado en la variable global.
gb_apply_position(Position,Property)	Extrae la expresión de trabajo de la variable global, y hace la llamada a Property para que la ejecute sobre el elemento en la posición Position de la selección. Guarda el resultado en la variable global.

Tabla B.4: Predicados principales del módulo de aplicación de Chalkpy.

Módulo de operativa

En la Tabla B.5 se encuentran los predicados más importantes de este módulo junto a una breve descripción de su cometido.

Predicado	Acción
operate(List)	Opera la lista por recursión.
operate_equal(Equation,NewOp,R)	Devuelve en R una nueva ecuación con añadiendo a los términos de Equation la operación con su elemento NewOp.
commute(List,Start,End,R)	Devuelve en R la conmutación de los elemento de la lista desde el elemento en el índice Start al elemento en el índice End.
associate(List,Start,End,R)	Devuelve en R la asociación de los elemento de la lista desde el elemento en el índice Start al elemento en el índice End.
dissociate(List,Position,R)	Devuelve en R el resultado de disociar el elemento en la posición Position.
distributive(List,Position,R)	Devuelve en R el resultado de distribuir el elemento en la posición Position.
common_factor(List,R)	Devuelve en R el resultado de aplicar el factor común a la lista List.

Tabla B.5: Predicados principales del módulo de operativa de Chalkpy.

Módulo de impresión

En la Tabla B.6 se encuentran los predicados más importantes de este módulo junto a una breve descripción de su cometido.

Predicado	Acción
gb_print	Imprime, de forma recursiva, la expresión de trabajo con la representación del usuario y la selección en color.

Tabla B.6: Predicados principales del módulo de impresión de Chalkpy.