UNIVERSIDAD DE LA REPÚBLICA

Mejoras al intérprete MateFun

por

Nicolás Vázquez

nicolas.vazquez@fing.edu.uy

tutores:

Marcos Viera mviera@fing.edu.uy

Gonzalo Tejera gtejera@fing.edu.uy

Proyecto de Grado, Ingeniería en Computación

en la

Facultad de Ingeniería

Instituto de Computación

12 de diciembre de 2019

Resumen

MateFun es un programa matemático de software con fines académicos orientado a estudiantes de secundaria desarrollado previamente al comienzo de este trabajo. El componente principal de MateFun, su intérprete fue desarrollado por el Instituto de Computación. Luego, como parte de otro proyecto de grado se extendió el sistema añadiendo una capa de presentación y de lógica de negocios, interactuando con el intérprete.

El intérprete de MateFun interpreta y compila un lenguaje funcional simple, con una sintaxis muy similar al lenguaje matemático. La simplicidad del lenguaje funcional busca acompañar los conocimientos matemáticos de los estudiantes, específicamente reforzar los conceptos de funciones y conjuntos.

Actualmente el programa está en período de pruebas y es evaluado por profesores y estudiantes de secundaria.

El presente trabajo está enfocado enteramente en el intérprete de MateFun y las tareas realizadas en torno a él.

Al ser un programa existente se realizó una evaluación del estado actual del sistema y sus funcionalidades. En esta tarea se hizo enfoque especialmente en comprender su código fuente para poder realizar modificaciones y agregar nuevas funcionalidades al sistema.

Se agregó la funcionalidad de traducción al sistema, para brindar soporte a múltiples idiomas. Además, se planteó corregir un problema detectado en funciones definidas por partes, dado que no se realizaban chequeos en los subdominios definidos de estas funciones.

Además se introdujo integración continua al proceso de desarrollo del sistema, así como también una mejora en este proceso en la forma de contribuir al sistema.

Por último, se creó documentación sobre el sistema existente, las nuevas funcionalidades y procesos definidos.

Palabras clave: MateFun, Matemática, Haskell, Internacionalización, Integración Continua, Superposición de dominios.

Índice general

Resumen II

Índice de figuras VII

Índice de tablas VIII

1. Introducción **1**
1.1. MateFun . . . 1
1.2. Arquitectura del sistema MateFun . . . 2
1.3. Lenguaje MateFun . . . 4
1.3.1. Descripción del lenguaje . . . 4
1.3.2. Sintaxis . . . 6
1.4. Intérprete . . . 7
1.4.1. Análisis del código fuente . . . 7
1.4.2. Comandos del intérprete . . . 8
1.4.3. Funciones predefinidas . . . 10

2. Fundamentos teóricos **14**
2.1. Integración continua en el desarrollo de software . . . 14
2.1.1. Procesos de software . . . 14
2.1.2. Integracíon continua . . . 15
2.1.3. Prácticas recomendadas . . . 16
2.2. Internacionalización . . . 17
2.2.1. Conceptos preliminares . . . 17
2.2.2. GetText . . . 18
2.2.3. Internacionalización en Haskell . . . 20
2.3. Problema de detección de intersección de dominios en funciones . . . 21
2.3.1. Conceptos preliminares . . . 21
2.3.2. Enunciado del problema . . . 22
2.3.3. Resolución del problema . . . 23
2.3.3.1. Boolector . . . 24
2.3.3.2. SBV . . . 25

3. Integración Continua en MateFun **27**
3.1. GitLab CI/CD . . . 27
3.1.1. Runners . . . 28

3.1.2. Pipelines 28
3.2. Docker y contenedores de software 29
3.3. Aplicación en MateFun 30
3.3.1. Imagen Docker para ejecución de jobs 31
3.3.2. Definición de pipeline de integración continua 33

4. Proceso de contribución a MateFun 38
4.1. Motivación 38
4.2. Contribuciones 39
4.3. Reportar un problema encontrado mediantes Issues 39
4.4. Creacion de merge requests 40
4.5. Protección de la rama principal del repositorio 41

5. Internacionalización en MateFun 43
5.1. Preparación del código fuente 43
5.2. Aplicación en MateFun 44
5.2.1. Anotación de claves en MateFun 45
5.2.2. Directorio de internacionalización 46
5.2.3. Script de internacionalización 47
5.2.4. Selección de idiomas en MateFun 49

6. Detección de intersección de dominios en funciones por partes en MateFun 51
6.1. Funciones definidas por partes 51
6.2. Funciones por partes en MateFun 52
6.3. Método de detección de intersección de dominios en funciones definidas por partes 53
6.4. Aplicación en MateFun 54
6.4.1. Solución en alto nivel 54
6.4.2. Implementación 56
6.4.2.1. Evaluación estática de intersección de dominios 56
6.4.2.2. Chequeo dinámico de intersección de dominios 64

7. Conclusiones 68

Índice de figuras

1.1. Interfaz gráfica de MateFun 2
1.2. Diagrama de arquitectura de MateFun 3
1.3. Diagrama de módulos del intérprete MateFun 8
1.4. Lista de comandos de MateFun 9

2.1. Esquema clásico de integración continua 16
2.2. Esquema del proceso de internacionalización de GNU gettext 19
2.3. Arquitectura del solver Z3 24
2.4. Arquitectura del solver Boolector implementando lemas a demanda 24

3.1. Diagrama comparativo entre contenedores y máquinas virtuales 29
3.2. Versiones disponibles de la imagen matefun-env 31
3.3. Pipeline de integración continua de MateFun 36
3.4. Runners compartidos en MateFun 36

4.1. Menú lateral del repositorio MateFun en GitLab 40
4.2. Rama principal de MateFun protegida de push 41

5.1. Esquema de internacionalización en MateFun 44
5.2. Diagrama de módulos actualizado de MateFun 46
5.3. Directorio de internacionalización en MateFun 46
5.4. Script de internacionalización en MateFun 48
5.5. Selección de idioma en MateFun 49

Índice de tablas

1.1. Funciones aritméticas y trigonométricas predefinidas en MateFun 10
1.2. Funciones para el manejo de figuras predefinidas en MateFun 11
1.3. Funciones predefinidas para el manejo de secuencias en MateFun 12

2.1. Tabla comparativa de servidores de integración continua 17

Índice de códigos

1.1. Ejemplos de definiciones de conjuntos en MateFun 5
1.2. Ejemplos de definiciones de funciones en MateFun 5
3.1. Definición imagen Docker matefun-env en el archivo Dockerfile 32
3.2. Definición del pipeline de integración continua en MateFun en el archivo .gitlab-ci.yml 34
6.1. Carga de un programa MateFun 57
6.2. Ejemplo de programa MateFun con intersección en dominios 62
6.3. Evaluación dinámica de funciones 65

Capítulo 1

Introducción

Este trabajo está enfocado en una herramienta informática desarrollada para apoyar a estudiantes de matemáticas de educación media en Uruguay. Mediante esta herramienta informática se busca también acercar a los estudiantes a la programación. Esta herramienta es MateFun[1].

1.1. MateFun

MateFun[1] es un lenguaje funcional simple de programación, diseñado para facilitar el aprendizaje de las funciones matemáticas. El objetivo de MateFun es desarrollar el pensamiento matemático como también computacional en los estudiantes. En la currícula de educación media se empezó a incluir la programación debido a que es una herramienta necesaria para el individuo en el mundo actual. Sin embargo, la enseñanza de programación corre por carriles separados a la enseñanza de matemáticas, cuando ambas disciplinas están altamente relacionadas. Los resultados de matemáticas en la enseñanza media en Uruguay muestran que es la asignatura con menor nivel de aprobación entre las materias de Ciclo Básico[2].

Para atacar este problema y fomentar la relación entre programación y matemática en estudiantes, se han hecho actividades con estudiantes[2] y profesores. Estas actividades ponen énfasis en inculcar el pensamiento aplicado en la programación mediante el desarrollo de algoritmos y una posterior traducción a un lenguaje de programación.

[1] Disponible en: `https://matefun.math.psico.edu.uy`

[2] Monitor Educativo Licel de ANEP - 2015: `http://servicios.ces.edu.uy/monitorces/servlet/portada`

Como otra alternativa al enfoque en algoritmos, MateFun surge como un lenguaje funcional simple, que permita una mejor comprensión de las funciones, con una sintaxis muy similar al lenguaje matemático. La forma de interacción entre los estudiantes y el lenguaje MateFun es mediante una interfaz web. Esta interfaz web forma parte de la capa de presentación del sistema MateFun que describiremos a continuación.

1.2. Arquitectura del sistema MateFun

El sistema MateFun está formado por tres capas, que en un enfoque *top-down* se representan como:

- Capa de presentación: Esta capa se ejecuta en los navegadores web de los usuarios. El usuario interactúa con el sistema mediante la interfaz gráfica (Figura 1.1), y esta capa se comunica con la capa de lógica de negocios para delegar su ejecución.
- Capa de lógica de negocios: Esta capa está compuesta por las clases y entidades que representan los objetos a ser utilizados en el sistema. Contiene también la implementación de las funcionalidades con las que el usuario interactúa mediante la interfaz gráfica. La implementación de estas funcionalidades se logran mediante el intérprete de MateFun.
- Capa de persistencia: Esta capa realiza la interacción con la base de datos y brinda comunicación con la capa de negocios para poder persistir y consultar datos desde la base de datos.

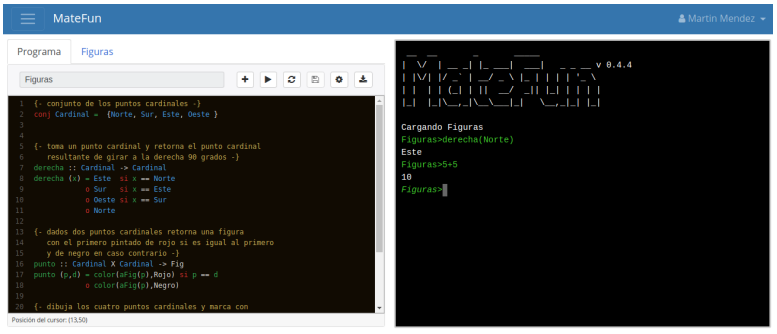

FIGURA 1.1: Interfaz gráfica de MateFun

En la Figura 1.2 se observa un diagrama de distribución incluyendo los componentes del sistema y las capas involucradas.

La comunicación entre los navegadores web y el servidor de aplicaciones se realiza mediante servicios web. Estos servicios web son expuestos por la Aplicación Web en el servidor de aplicaciones.

La Aplicación Web invoca al Intérprete mediante el runtime de Java, debido a que el intérprete se encuentra empaquetado como un binario. De esta forma se crea un nuevo proceso en el servidor de aplicaciones con la ejecución del intérprete. Las interacciones entre la Aplicación y el Intérprete se realizan mediante entrada y salida: la Aplicación Web envía la entrada al Intérprete y recolecta la salida resultante, para devolverla al usuario.

Por último, la Aplicación Web logra la comunicación con el servidor de base de datos mediante JPA (Java Persistence API).

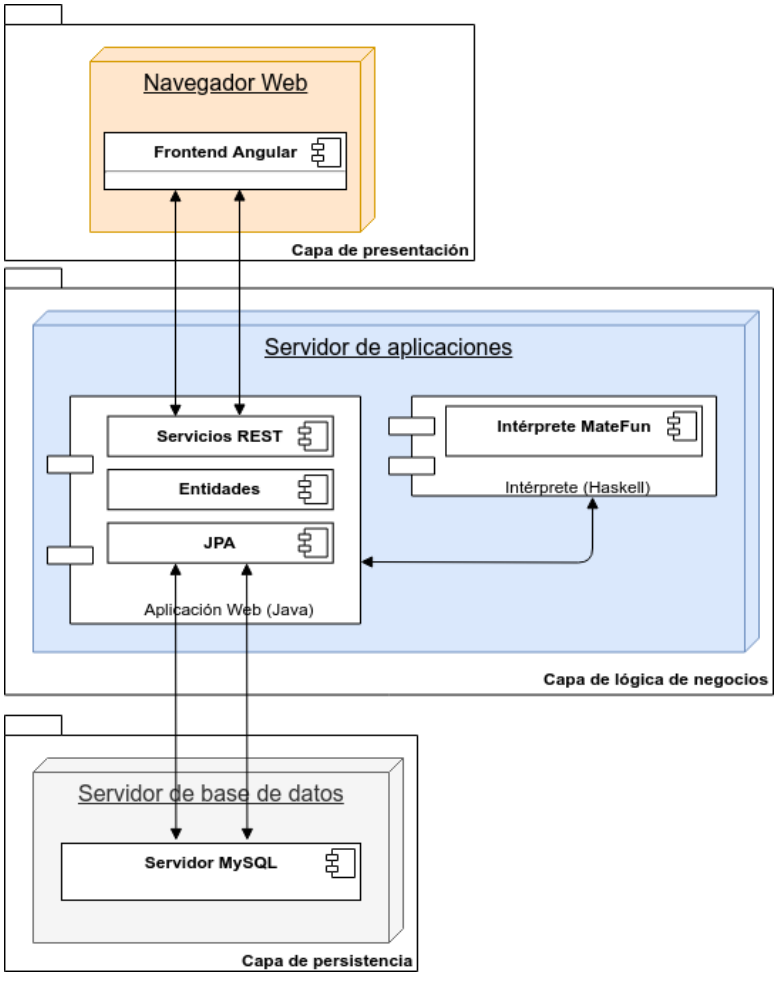

FIGURA 1.2: Diagrama de arquitectura de MateFun

1.3. Lenguaje MateFun

La programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas. Este paradigma es utilizado mayormente con fines académicos y de investigación y es diferente al paradigma imperativo, más comunmente utilizado en la industria debido a la posibilidad de contar con el concepto de estado facilitando su comprensión. En el paradigma imperativo pueden definirse funciones, pero la diferencia fundamental con el paradigma funcional es que se pueden obtener diferentes resultados de funciones en distintos momentos dada la misma entrada de parámetros. Con el paradigma funcional, los resultados de una función dependen exclusivamente de los parámetros de entrada, por lo tanto se obtiene el mismo resultado a iguales parámetros de entrada en cualquier momento de la ejecución.

Los lenguajes funcionales utilizan el paradigma funcional y por lo tanto permiten definir funciones matemáticas. Sin embargo, como en el caso de Haskell, se añade una gran complejidad para el estudiante al contar con conceptos como pattern matching, polimorfismo, tipos de datos, evaluación lazy, mónadas. Para combatir esta complejidad, se desarrolló el lenguaje MateFun, escrito en Haskell[3].

MateFun es un lenguaje de programación funcional muy simple, diseñado para introducir la programación a estudiantes liceales y a su vez fortalecer la apropiación del concepto de función matemática y tener un mejor acercamiento al álgebra. Se busca que el lenguaje pueda ser rapidamente asimilado por los usuarios debido a la cercana notación con la notación matemática.

1.3.1. Descripción del lenguaje

Un programa de MateFun está definido como una lista de conjuntos y/o funciones.

Un conjunto se define utilizando la palabra reservada *conj* seguido del nombre del conjunto. Se puede definir un conjunto por enumeración o por comprensión. El nombre del conjunto debe comenzar con una letra mayúscula, y acepta letras y números.

- Al definir un conjunto por enumeración, se deben declarar los elementos del conjunto enumerado separados por comas, cada uno comenzando con letra mayúscula.

[3]https://www.haskell.org/

- Al definir un conjunto por comprensión, se debe indicar un conjunto base (puede ser cualquier conjunto definido o un conjunto predeterminado) y una condición que deben cumplir los elementos del conjunto base para pertenecer al conjunto definido

En el fragmento de código 1.1 se presentan ejemplos de definiciones de conjuntos por enumeración y por comprensión.

```
1 {- Por Enumeracion -}
2 conj Mes = { Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio, Agosto,
3              Setiembre, Octubre, Noviembre, Diciembre }
4
5 {- Por Comprension -}
6 conj RealesPositivos = { x en R | x >= 0 }
```

Código 1.1: Ejemplos de definiciones de conjuntos en MateFun

Para definir una función se debe indicar su signatura y la ecuación que la define. La signatura se compone del nombre de la función, el conjunto dominio y el codominio. La ecuación se define dando el nombre de la función, los parámetros y el cuerpo de la función. Se pueden definir funciones con más de una variable independiente, para esto se utiliza el conjunto producto cartesiano (y n-tuplas).

Una función se puede definir por casos, estableciendo la condición de cada caso y por último un caso por defecto (si no se cumple ninguna condición), de manera que la función sea total sobre su dominio. Se presentan ejemplos de definiciones de funciones en el fragmento de código 1.2.

```
1 {- Funciones -}
2 area_triangulo :: R X R -> R
3 area_triangulo(base,altura) = base * altura / 2
4
5 {- Funcion por casos -}
6 valor_absoluto :: R -> R
7 valor_absoluto(x) =   x si x > 0
8                     o -x si x < 0
9                     o 0
```

Código 1.2: Ejemplos de definiciones de funciones en MateFun

Las expresiones, que definen el cuerpo de una función, se pueden componer de literales, variables, aplicación de operadores infijos, aplicación de operadores prefijos y aplicación de funciones.

Las condiciones soportadas por MateFun son relaciones binarias de igualdad y orden. Las condiciones se aplican a dos expresiones. Una condición se define como una relación entre dos expresiones o una lista de relaciones separadas por comas.

Además se permite definir secuencias de elementos de un mismo conjunto, permitiendo repetir elementos. Las secuencias son colecciones ordenadas de elementos de un mismo conjunto que admite elementos repetidos.

1.3.2. Sintaxis

Luego de la breve introducción en la sección anterior, en esta sección se introduce formalmente el lenguaje MateFun. Para describir su sintaxis utilizamos la notación BNF[4]:

```
1  programa :== { def_conjunto | def_funcion }
2
3  {- Conjuntos -}
4  def_conjunto :== "conj" nombre_conjunto "=" "{" (conj_enumeracion |
       conj_comprension) "}"
5  conj_enumeracion :== [ nombre_enumerado "," ] nombre_enumerado
6  conj_comprension :== variable "en" conjunto "|" condicion
7  conjunto :== "R" | "Fig" | "Color" | nombre_conjunto | { conjunto "X"}
       conjunto "X" conjunto | conjunto "*"
8  condicion :== relacion | "(" {relacion ","} relacion ")"
9  relacion :== expresion operador expresion
10 operador :== "==" | "/=" | "<" | ">" | "<=" | ">="
11
12 {- Expresiones -}
13 expresion :== literal | variable | operacion_infijo | operacion_prefijo |
        aplicacion_funcion | secuencia
14 literal :== real | nombre_enumerado | figura | figura3d | color
15 operacion_infijo :== expresion operador_binario expresion
```

[4]Backus-Naur Form

```
16 operacion_prefijo :== operador_prefijo expresion
17 operador_binario :== "+" | "-" | "*" | "/" | "^" | "!" | ":"
18 operador_prefijo :== "-"
19 secuencia :== "[" "]" | "[" literal "]" | literal ":" secuencia
20
21 {- Funciones -}
22 def_funcion :== signatura ecuacion
23 signatura :== nombre_funcion "::" conjunto "->" conjunto
24 ecuacion :== nombre_funcion "(" [{variable ","} variable] ")" "=" gexp
25 gexp :== {expresion "si" condicion "o"} expresion
```

1.4. Intérprete

El intérprete de MateFun[5] es un programa ejecutable por consola escrito en Haskell, desarrollado por el Instituto de Computación (InCo) de la Facultad de Ingeniería. El intérprete define un lenguaje funcional llamado MateFun y permite interpretar programas escritos en este lenguaje, cargarlos en el intérprete e interactuar con ellos.

1.4.1. Análisis del código fuente

En primer lugar es importante comprender el funcionamiento del programa con el cual se va a trabajar. MateFun es un programa escrito en Haskell de pequeño porte que cuenta con los siguientes módulos:

- Core: Se definen las estructuras y tipos necesarios durante la ejecución del programa
- CstProp: Este módulo define las funciones necesarias para realizar propagación de constantes en los programas que el usuario carga en el sistema
- Eval: Este módulo define las funciones necesarias para realizar evaluaciones de expresiones y condiciones en tiempo de ejecución
- GenCanvas: Este módulo se encarga de generar figuras en formato JSON para ser enviados al cliente web

[5]Repositorio: `https://gitlab.fing.edu.uy/matefun/MateFun`

- Parser: Este módulo declara las estructuras y funciones necesarias para parsear los programas del usuario y llevarlos a estructuras manejables por MateFun en tiempo de ejecución.
- RenderFun: Módulo para generar gráficas de funciones
- TypeCheck: Módulo para realizar chequeo de tipos en funciones a ser cargadas
- Warning: Módulo para realizar chequeo de advertencias en programas a ser cargados
- MateFun: Módulo principal del intérprete

En la Figura 1.3 se observan las dependencias entre los módulos[6] previamente presentados.

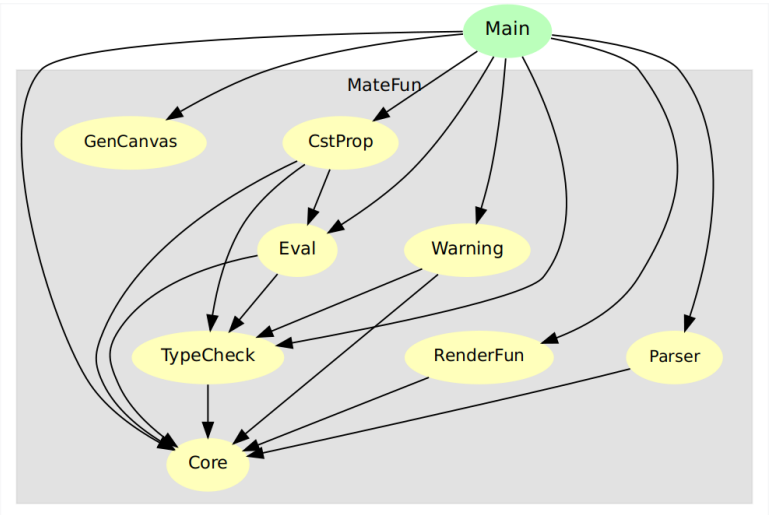

FIGURA 1.3: Diagrama de módulos del intérprete MateFun

1.4.2. Comandos del intérprete

Los comandos de MateFun son de dos tipos: descriptivos y de acciones. Para diferenciar los comandos de los nombre de funciones, se utilizan caracteres especiales antes del nombre del comando:

- '!' indica que el comando es un comando de acción
- '?' indica que el comando es un comando descriptivo

[6]Diagrama generado a partir de la biblioteca `http://hackage.haskell.org/package/graphmod` y visualizado con la herramienta `http://www.graphviz.org/`

La lista de comandos de MateFun se representa al usuario en español como se muestra en la Figura 1.4

FIGURA 1.4: Lista de comandos de MateFun

Los comandos de acción realizan un cambio en el contexto del sistema:

- cargar: Permite cargar un programa. Las funciones y conjuntos se cargan en el contexto del sistema siempre que el programa sea correcto. Una vez cargado el programa, el usuario puede utilizar las funciones y conjuntos definidos en el intérprete.
- recargar: Recargar el programa actual y el contexto
- salir: Terminar la ejecución

Los comandos descriptivos muestran información del contexto de la ejecución de MateFun:

- funs: Muestra las funciones cargadas en el contexto: predefinidas y definidas por el usuario.
- vars: Muestra las variables definidas por el usuario
- conjs: Muestra los conjuntos definidos por el usuario
- fun: Muestra información de una función
- var: Muestra información de una variable
- conj: Muestra información de un conjunto

- grafica: Devuelve una gráfica de una función
- ayuda: Despliega el mensaje de ayuda

1.4.3. Funciones predefinidas

El intérprete MateFun cuenta con funciones predefinidas para facilitar el uso del intérprete y las definiciones de conjuntos y funciones, incluyendo funciones de aritmética y trigonometría básicas y también funciones para el manejo y definición de figuras. A continuación se listan todas las funciones predefinidas en MateFun:

Función	Firma	Descripción
$-$	$\mathrm{R} \to \mathrm{R}$	Devuelve el opuesto del parámetro
red	$\mathrm{R} \to \mathrm{R}$	Función redondeo, devuelve el siguiente entero más cercano
sen	$\mathrm{R} \to \mathrm{R}$	Función seno
cos	$\mathrm{R} \to \mathrm{R}$	Función coseno
raizcuad	$\mathrm{R} \to \mathrm{R}$	Función raíz cuadrada

Tabla 1.1: Funciones aritméticas y trigonométricas predefinidas en MateFun

Función	Firma	Descripción
rgb	(R X R X R) → Color	Genera un color mediante una tupla de valores reales correspondientes a los valores de: rojo, verde y azul
rect	(R X R) → Fig	Genera una figura rectangular a partir de la tupla correspondientes al largo y ancho de la figura
circ	R → Fig	Genera una figura circular con radio correspondiente al parámetro
linea	((R X R) X (R X R)) → Fig	Genera una figura linear que pasa por los puntos pasados como parámetros mediante tuplas
poli	(R X R)* → Fig	Genera una figura con forma de polígono que conecta los puntos pasados como parámetro mediante la secuencia
juntar	(Fig X Fig) → Fig	Genera una nueva figura a partir de juntar dos figuras
color	(Fig X Color) → Fig	Aplica un color a una figura
mover	(Fig X (R X R)) → Fig	Mueve una figura a un punto en el plano
rotar	(Fig X R) → Fig	Dada una figura y un numero, rota la figura ciertos grados en sentido horario
escalar	(Fig X R) → Fig	Dada una figura y un numero, escala la figura
aFig	A → Fig	Genera una figura a partir de un elemento de un conjunto.

Tabla 1.2: Funciones para el manejo de figuras predefinidas en MateFun

De manera análoga a las funciones predefinidas para figuras en dos dimensiones, se tienen funciones predefinidas para el manejo de figuras 3D que no se incluyen en esta sección.

Por último, las funciones predefinidas para secuencias son las siguientes:

Función	Firma	Descripción
rango	(R X R X R) → R*	Genera una secuencia de números a partir de un número base, un número final y un paso
primero	A* → A	Devuelve el primer elemento de la secuencia
resto	A* → A*	Devuelve la secuencia sin su primer elemento

Tabla 1.3: Funciones predefinidas para el manejo de secuencias en MateFun

Capítulo 2

Fundamentos teóricos

En este capítulo se resaltan los aspectos más importantes desarrollados en el documento de Estado del Arte[1]. Los temas mencionados en esta sección y en el documento no están relacionados entre sí excepto por ser las líneas principales del proyecto de grado.

2.1. Integración continua en el desarrollo de software

2.1.1. Procesos de software

Un proceso de software[3] se define como una serie de actividades relacionadas que conduce a la elaboración de un producto de software. Se puede clasificar un proceso como: dirigido por un plan, en el que se mide el avance del proceso contra un plan previamente definido, o proceso ágil, en el cual los requerimientos cambiantes del cliente implican que el proceso se modifique y se adapte a estas necesidades.

Las actividades fundamentales de un proceso de software incluyen: especificación, diseño e implementación, validación y evolución. En la práctica es más común optar por procesos ágiles ya que es difícil que los requerimientos se mantengan incambiados durante el proceso. Los procesos ágiles de desarrollo de software tienen una gran ventaja sobre los dirigidos a un plan, debido a que estos últimos no están orientados al desarrollo rápido de software. Un cambio en los requerimientos en un modelo como el de cascada, implica una reelaboración de las primeras etapas. Por otra parte, los métodos ágiles aprovechan

[1] `https://gitlab.fing.edu.uy/matefun/MateFun/wikis/uploads/07b4ec42ceb999b1c9681936c0155376/MateFun_EstadoArte.pdf`

la retroalimentación obtenida a partir de los clientes para poder adaptarse a los requerimientos cambiantes. Sin embargo, los procesos ágiles tienen la desventaja de centrarse en la comunicación entre personas en vez de la documentación debido a los requerimientos cambiantes, lo que genera una dependencia hacia las personas involucradas en vez de apoyarse en la documentación.

2.1.2. Integracíon continua

La integración continua[4] es una práctica ágil de desarrollo de software en las que los miembros de un equipo de desarrollo integran su trabajo a un repositorio frecuentemente, al menos una vez al día, buscando múltiples integraciones al día. La finalidad de realizar integraciones en el código fuente frecuentemente es poder detectar errores de forma temprana en el proyecto y reducir largos períodos de integración de código fuente. La forma de detección de estos errores es mediante *builds* y pruebas automatizadas por cada contribución que se realiza al repositorio.

La idea central es contar con una versión estable del producto en cada instante en el repositorio. Una versión se considera estable si:

- El programa se puede compilar
- Las funcionalidades básicas del programa funcionan correctamente

La forma de asegurar la estabilidad del código fuente es mediante la automatización de las tareas de compilación y *build*, de ejecución de tests unitarios y de integración. Estas tareas al ser automatizadas y ejecutadas cada vez que se hace un commit al repositorio, brindan una forma sencilla de detectar errores potencialmente introducidos por los desarrolladores. Al conjunto de tareas automatizadas al realizar un commit sobre el repositorio se le conoce como pipeline de integración continua.

Un pipeline de integración continua puede contar con varias etapas, con múltiples tareas por cada etapa. La única restricción que se impone es que las etapas de un pipeline deben ejecutarse en orden, según el orden con el que se hayan definido. Una etapa fallida genera que el pipeline completo falle. Una etapa satisfactoria, genera que se ejecute la siguiente etapa del pipeline.

Un pipeline clásico de integración continua (ilustrado en la Figura 2.1) cuenta con dos etapas: una etapa inicial de compilación y ejecución de tests unitarios; y una

segunda etapa de ejecución de tests de integración

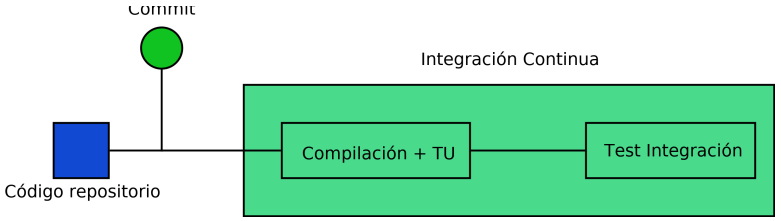

FIGURA 2.1: Esquema clásico de integración continua

2.1.3. Prácticas recomendadas

Para poder adoptar la integración continua se recomienda contar con un único repositorio para el código fuente que sea accesible a todos los miembros del equipo de desarrollo. Además se recomienda que las tareas automatizadas se mantengan testeables y que su ejecución se realice sobre un ambiente lo más parecido a producción posible. Es decisión de los administradores del proyecto organizar la infraestructura para que la ejecución se realice en servidores dedicados de su organización o también pueden optar por servidores alojados en la nube.

Existen diversos productos que ofrecen soluciones de integración continua como servicios[5] en la nube, o como aplicaciones instalables en servidores de las organizaciones. A continuación se muestra un cuadro comparativo de los productos más destacados para lograr la integración continua en los que se considera si el producto se ofrece como servicio, si ofrece una versión instalable, si es gratuito o requiere comprar una licencia y por último la forma de ejecutar los jobs (tareas) en los pipelines:

Producto	Instalable	Hosteado	Gratuito	Ejecución de jobs
Jenkins	Si	No	Si, ilimitado	Máquinas virtuales y contenedores
Bamboo	Si	No	No, prueba por 30 días	Máquinas virtuales y contenedores
Travis CI	No	Si	Si, solo proyectos públicos	Máquinas virtuales
GitLab CI	Si	Si	Si, limitado	Máquinas virtuales y contenedores
Circle CI	Si	Si	Si, limitado	Contenedores
Buddy	Si	Si	Si, limitado	Contenedores

Tabla 2.1: Tabla comparativa de servidores de integración continua

Debido a que la Facultad de Ingeniería cuenta con infraestructura dedicada a GitLab[2] y que el repositorio de MateFun se encuentra alojado allí, se hará uso de GitLab CI/CD incluido en la versión de GitLab alojada en los servidores de Facultad.

2.2. Internacionalización

2.2.1. Conceptos preliminares

Comenzamos introduciendo el concepto de configuración regional, también conocido como *locale* en inglés. La configuración regional es una lista de atributos que definen un lenguaje, por ejemplo: nombre del idioma, nombre de la región o país, formato de fechas, monedas, numeración utilizada en la región. Los *locales* se representan con un identificador que contiene como mínimo el identificador del idioma y el identificador de

[2] `https://gitlab.fing.edu.uy/`

la región[6]. Se utiliza el estándar ISO 15897, variante de BCP 47, para identificar locales en sistemas POSIX.

El proceso por el cual se busca adaptar un producto de software a una región o mercado determinado se llama localización (L10n). Las tareas de localización incluyen la traducción de textos mostrados al usuario en un producto de software, pero contienen además tareas más complejas que refieren a una adaptación cultural del producto de software[7].

El proceso de internacionalización (i18n) implica transformar un producto de software para que pueda soportar localización. La internacionalización requiere que se realicen cambios en el código fuente de un producto de software para que se permita la localización. En general, el proceso de internacionalización de software implica:

- Diseñar o modificar el diseño para que permita múltiples *locales*
- Preparar el código fuente para que permita múltiples parámetros de los *locales*
- Mantener por separado el código fuente de los elementos internacionalizables, para permitir que se carguen dinámicamente según la preferencia del usuario.

2.2.2. GetText

El estándar de internacionalización de programas de software se conoce como Gettext[8]. En este estándar se consideran diversos actores: desarrolladores, traductores y usuarios finales. Los desarrolladores deben encargarse que el código fuente de los programas a internacionalizar sea flexible y soporte textos que varíen según el idioma. Sin embargo, no es responsabilidad de los desarrolladores el encargarse de las traducciones, sino que le deben proporcionar los textos a traducir a los traductores. Por lo general, desarrolladores y traductores acuerdan en un idioma en común en el cual:

- Los desarrolladores anoten las claves a traducir en el código fuente
- Los traductores obtengan estas claves y le asignen un valor a cada una según el idioma de destino.
- Los usuarios finales deberán poder seleccionar el idioma al cual traducir el programa de software. Una vez que los pares clave-valor para cada lenguaje están definidos, se mostrarán de acuerdo al idioma seleccionado.

A más bajo nivel, el estándar define tres tipos de archivos para que el proceso descrito anteriormente pueda realizarse:

- Archivo POT: Este tipo de archivo es una plantilla para cada archivo de traducción de un lenguaje específico. Es decir, cada archivo de traducción de lenguaje es creado a partir de esta plantilla. Esto se debe a que la plantilla contiene todas las claves a ser traducidas para el programa de software.
- Archivos PO: Se tiene un archivo PO por cada lenguaje a traducir. Sobre estos archivos trabajan los traductores, generando los valores para cada clave a traducir.
- Archivos MO: Una vez terminadas las traducciones a los idiomas, los archivos PO se codifican a lenguaje de máquina. Son estos archivos los que Gettext utiliza para realizar las traducciones a los distintos idiomas

Una implementación de Gettext es GNU Gettext, el cual toma código fuente en lenguaje C para comenzar la etapa de extracción. El proceso seguido se ilustra en la Figura 2.2. La parte final del proceso de internacionalización consiste en setear las siguientes variables de entorno: *$LANG*, *$LC_ALL* y *$LANGUAGE*. *$LANG* es por lo general la variable de entorno utilizada para referenciar un *locale* a utilizar, pero el uso más común de gettext es setear la variable *$LANGUAGE* al idioma a utilizar, la cual tiene preferencia sobre las demás variables. Una vez configurada la variable *$LANGUAGE* se logra la traducción al idioma seleccionado.

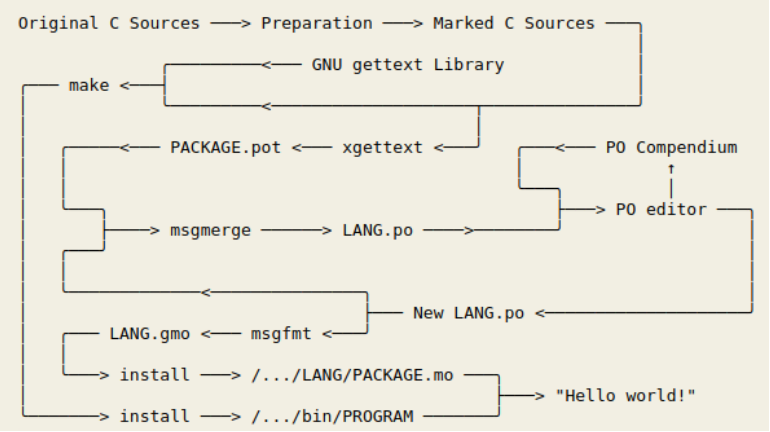

FIGURA 2.2: Esquema del proceso de internacionalización de GNU gettext, extraído de [9]

2.2.3. Internacionalización en Haskell

Existen distintas bibliotecas para Haskell con las cuales lograr la internacionalización:

El enfoque más simple de internacionalización en Haskell es mediante la biblioteca hgettext[8]. Esta biblioteca contiene:

- Una función especial para que los desarrolladores puedan marcar cuáles son los textos a traducirse.
- Generación de archivo POT a partir de los textos marcados mediante la función especial
- una interfaz de comunicación con el sistema GNU gettext, con el cual se le indica a gettext el directorio con los archivos necesarios para realizar las traducciones.

Los desarrolladores deben utilizar la función especial brindada por la librería en cada texto a traducir en el código fuente. Luego, se debe generar el archivo POT mediante la librería. El último paso es indicar también en el código fuente en qué directorio GNU gettext debe buscar los archivos necesarios para la traducción.

Al invocar directamente a GNU gettext, esta librería necesita que se setee la variable de entorno *$LANGUAGE* antes de la invocación al programa, es decir que la localización es externa al programa, de forma análoga a gettext.

Otra biblioteca de Haskell: i18n[10], presenta un enfoque más simplificado que hgettext y no requiere que se utilice GNU gettext. Por el contrario, obtiene las traducciones directamente de los archivos PO sin invocar a gettext. Esta librería no requiere el uso de variables de entorno para la localización, sin embargo su principal desventaja es que los tipos de datos deben contener también el idioma en el código fuente, además de no contar con una herramienta que permita crear archivos POT a partir del código fuente.

La biblioteca Shakespeare[11] utiliza un enfoque de internacionalización con tipos de datos nativos de Haskell. Cada texto a traducir debe agregarse como un nuevo constructor del tipo de datos definido para traducciones en el código fuente. Las traducciones se mantienen en archivos separados al igual que hgettext e i18n, sin embargo no utilizan el formato estándar de gettext sino que cada archivo en Shakespeare requiere que se provea el constructor del tipo de datos como clave a traducir. Esto dificulta enormente la tarea

de los traductores, quienes ya no cuentan con textos a traducir sino con un identificador de un constructor, por lo cual se requiere que se examine el contexto en el código fuente para lograr una traducción.

Presentadas las opciones de bibliotecas a utilizar, se consideraron varios aspectos en los cuales se priorizaron: la simplicidad de la solución, la facilidad de modificar traducciones existentes y de agregar nuevos idiomas y la refactorización del código fuente existente. En primer lugar, se descarta la biblioteca Shakespeare dado que las traducciones se encuentran estrechamente ligadas al código fuente y se desea manatener por separado las traducciones del código. Comparando las restantes bibliotecas se optó por utilizar hgettext para minimizar el impacto en la refactorización del código fuente existente manteniéndolo al mínimo y además aprovechando que MateFun se ejecuta en servidores Unix, lo cual asegura el correcto funcionamiento del estándar GetText.

2.3. Problema de detección de intersección de dominios en funciones

2.3.1. Conceptos preliminares

Una función es una relación entre dos conjuntos llamados dominio y codominio, en la que a cada punto del dominio le corresponde un único punto del codominio. Si podemos representar el conjunto dominio como la unión de conjuntos disjuntos, podríamos asignar un valor funcional a cada subconjunto, obteniendo una función definida por partes.

$$A = \bigcup_{i=1}^{n} A_i, \quad A_i \cap A_j = \emptyset, \quad i, j \in [1, .., n], \quad i \neq j \tag{2.1}$$

$$f(x_1, x_2, ..., x_n) = \begin{cases} f_1(x_1, x_2, ..., x_n) & \text{si } (\mathrm{x_1}, \mathrm{x_2}, ..., \mathrm{x_n}) \in \mathrm{A_1} \\ f_2(x_1, x_2, ..., x_n) & \text{si } (x_1, x_2, ..., x_n) \in A_2 \\ ... & \\ f_n(x_1, x_2, ..., x_n) & \text{si } (x_1, x_2, ..., x_n) \in A_n \end{cases} \tag{2.2}$$

La función f representada en la Ecuación 6.3 se llama función definida por partes.

El problema de detección de intersección de dominios en las definiciones de funciones por partes consiste en determinar si efectivamente la función que se intenta definir es una función. Es decir, se debe verificar que los conjuntos que componen la función sean disjuntos, y que su unión sea el conjunto dominio de la función (verificar que se cumpla la restricción 6.2).

Podemos modelar este problema como un problema conocido, el problema de las teorías de satisficibilidad módulo (SMT). Este problema es una extensión del problema de satisficibilidad booleana (SAT) para lógica de primer orden. Por lo tanto, dada una fórmula de primer orden, SMT consiste en encontrar una interpretación (o modelo) que haga la fórmula verdadera.

El problema de satisfactibilidad booleana (SAT) plantea determinar si existe una interpretación que haga que una fórmula proposicional sea satisfactible. Este problema fue el primero en ser demostrado como un problema NP-completo[12]. Al ser un problema NP-completo, no puede probarse si el problema general tiene siempre solución. Sin embargo, existen formas de encontrar soluciones aproximadas en tiempo exponencial con respecto a la entrada del problema.

El problema de las teorías de satisfactibilidad módulo (SMT) es una extensión del problema SAT para lógica de primer orden. SMT es un problema de decisión, consiste en decidir la satisfacibilidad de una fórmula de primer orden con respecto a una teoría de primer orden. Esto quiere decir, que consiste en encontrar una interpretación que haga a una fórmula de primer order tomar el valor $VERDADERO$. Esto es análogo a encontrar un modelo de la fórmula.

Por lo tanto, el problema a resolver se puede representar de la siguiente manera:

2.3.2. Enunciado del problema

Dada f una función definida por partes, tal que:

$$f : A \subseteq \mathbb{R}^n \to B \subseteq \mathbb{R}^m, \quad f(x_1, x_2, ..., x_n) = (y_1, y_2, ..., y_m) \tag{2.3}$$

$$f(x_1, x_2, ..., x_n) = \begin{cases} f_1(x_1, x_2, ..., x_n) & \text{si } (\mathrm{x_1}, \mathrm{x_2}, ..., \mathrm{x_n}) \in \mathrm{A_1} \\ f_2(x_1, x_2, ..., x_n) & \text{si } (x_1, x_2, ..., x_n) \in A_2 \\ ... & \\ f_n(x_1, x_2, ..., x_n) & \text{si } (x_1, x_2, ..., x_n) \in A_n \end{cases} \tag{2.4}$$

Necesitamos determinar si se cumple la siguiente restricción:

$$A = \bigcup_{i=1}^{n} A_i, \quad A_i \cap A_j = \emptyset, \quad i, j \in [1, .., n], \quad i \neq j \tag{2.5}$$

Es decir, si existe una interpretación en lógica de primer orden que haga verdadera a la siguiente fórmula:

$$\bigwedge_{i=1}^{n} (A_i \cap A_j = \emptyset), \quad i, j \in [1, .., n], \quad i \neq j \tag{2.6}$$

2.3.3. Resolución del problema

Afortunadamente existen los llamados solvers SMT, que son programas que pueden determinar si una fórmula es satisfactible e incluso (según el solver utilizado) encontrar una interpretación que haga la fórmula verdadera. A continuación se presentan algunas opciones de solvers SMT disponibles para Haskell:

Z3[13] es un software desarrollado por Microsoft Research escrito en C++. Es un proyecto de código abierto y posee APIs para distintos lenguajes, incluidos Haskell.

Z3 es un solver SMT eficiente[14] que recibe fórmulas y las procesa para simplificarlas. Una vez simplificada una fórmula pasa por un compilador que la convierte a una estrucutra interna útil. Una vez compilada una fórmula se somete a un núcleo de teorías que se integra a solvers de teorías y un solver SAT interno basado en DPLL para producir modelos. La arquitectura en alto nivel se puede observar en la Figura 2.3.

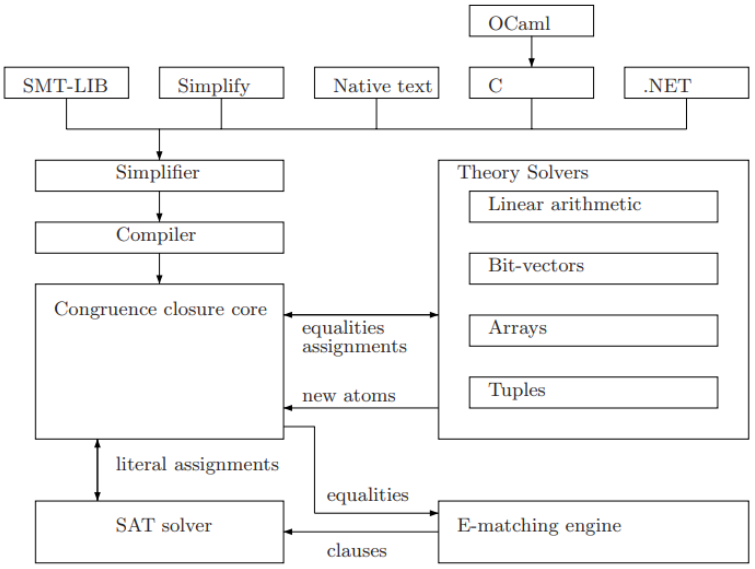

FIGURA 2.3: Arquitectura del solver Z3, extraído de [14]

2.3.3.1. Boolector

Boolector es un solver SMT que implementa una variante del enfoque lazy, llamado enfoque de lemas a demanda. La arquitectura de Boolector en alto nivel se muestra en la Figura 2.4.

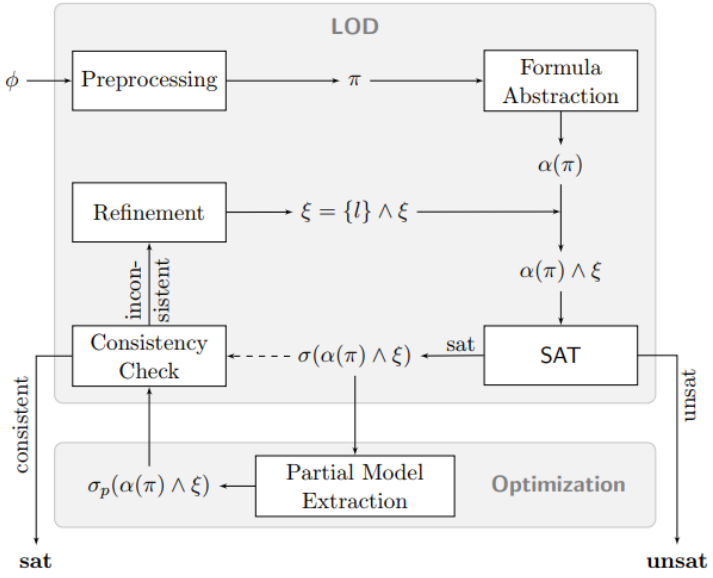

FIGURA 2.4: Arquitectura del solver Boolector implementando lemas a demanda, extraído de [15]

Por cada fórmula de entrada, se enumeran varios candidatos a modelo partiendo de asignaciones verdaderas que se van refinando hasta converger a lemas en forma normal conjuntiva. Una vez que se obtienen estos lemas se evaluan junto con la fórmula de entrada mediante el solver SAT.

2.3.3.2. SBV

Por último incluímos a la biblioteca SBV para Haskell. SBV provee una interfaz para interactuar con solvers SMT a través de Haskell. Si bien SBV no es un solver en sí, nos permite integrarlo a varios solvers que soporten el formato de la librería SMT-LIB.

Su gran ventaja es la posibilidad de poder ejecutar múltiples solvers a la vez, retornando el resultado del solver más rápido en cada instancia.

Los solvers SMT soportados por SBV son: ABC, Boolector, CVC4, MathSAT, Yices y Z3.

Debido a que SBV no representa un solver SMT en sí, se analizó la disponibilidad de bibliotecas para Haskell de Z3[16] y Boolector. Si bien la biblioteca de Haskell para Z3 se encuentra disponible desde 2012, Boolector contaba con una versión de prueba al momento de realizar el estudio comparativo, por lo tanto se optó por el solver Z3 y su biblioteca para Haskell para la detección de intersección de dominios en MateFun.

Capítulo 3

Integración Continua en MateFun

GitLab ofrece una versión en la nube y una versión instalable, cada una con distintos planes. Los planes más básicos de ambas versiones son gratuitos pero tienen la ventaja de que incluyen GitLab CI/CD (la solución de integración continua) integrada. La limitación del plan gratuito sobre integración continua en la versión alojada en la nube es que permite 2000 minutos mensuales de ejecución de pipelines. Esto significa aproximadamente una hora por día en promedio para ejecución de pipelines, lo cual sería suficiente dada la actividad actual del proyecto MateFun y la cantidad de colaboradores activos.

Sin embargo, el código fuente de MateFun se encuentra alojado en un repositorio privado[17] en GitLab de Facultad de Ingeniería[1]. Esto representa una gran ventaja dado que GitLab se encuentra instalado en los propios servidores de Facultad, por lo cual el tiempo de ejecución de pipelines no estará limitado.

3.1. GitLab CI/CD

GitLab CI/CD ofrece la funcionalidad de ejecutar pipelines automaticamente siempre que se definan para un proyecto cada vez que se hace push a un repositorio, sea cual sea la rama sobre la que se hace push. En general estos pipelines se utilizan para asegurar que la aplicación se construya de forma adecuada, y que las funcionalidades de la aplicación continuen funcionando correctamente con los nuevos cambios que se desean agregar en el push.

[1] `https://gitlab.fing.edu.uy/`

3.1.1. Runners

Los encargados de ejecutar los pipelines son aplicaciones llamadas runners[18]. Un runner de GitLab se encarga de ejecutar los distintos jobs que recibe de GitLab. Se distribuye como un binario escrito en Go que se instala en los servidores para ejecutar pipelines de integración continua y luego reportar los resultados a GitLab. Un runner puede ejecutarse como una máquina virtual, un contenedor Docker o incluso una máquina dedicada. Los runners permiten la ejecución de múltiples pipelines concurrentes y soportan incluso integración a Docker. El administrador del proyecto se debe encargar de registrar un runner en GitLab una vez instalado para poder ser utilizado.

El administrador de un proyecto puede registrar un runner como compartido o específico. Un runner compartido es utilizado para ejecutar pipelines de diferentes proyectos siempre que estos tengan los mismos requerimientos. Si se quiere asignar un runner a un proyecto en particular se debe registrar como específico. Su diferencia se encuentra en la forma en la que procesan los jobs a ejecutar. Un runner compartido utiliza un algoritmo de utilización justa (fair usage) mientras que en un runner específico se sigue el algoritmo FIFO (First In First Out).

3.1.2. Pipelines

Los pipelines deben definirse mediante un archivo de configuración en formato YAML en el directorio principal del proyecto. Los runners utilizaran la descripción brindada en el archivo de configuración para ejecutarlos.

Un pipeline está compuesto por:

- Jobs: Un job define qué es lo que se debe ejecutar
- Etapas: En cada etapa se define cuándo y cómo se deben ejecutar los jobs. Una etapa puede tener varios jobs

La ejecución de pipelines se realiza por etapas, y se cumple lo siguiente:

- Dentro de cada etapa se ejecutan los jobs concurrentemente
- Si todos los jobs de una etapa se ejecutan correctamente, se dice que la etapa es correcta y la ejecución continua en la siguiente etapa

- Si un job de una etapa falla, se dice que la etapa no es correcta y no se continua ejecutando el pipeline.

La forma más común de ejecución de pipelines es al hacer push al repositorio. Además, se pueden programar pipelines para que se ejecuten cada cierto tiempo definido por el administrador.

3.2. Docker y contenedores de software

Docker es una herramienta de código abierto que permite crear y ejecutar aplicaciones dentro de un ambiente aislado, independiente del sistema operativo, llamado contenedor. La idea del contenedor es empaquetar todas las dependencias necesarias para la ejecución de un programa en un solo lugar, permitiendo que pueda ser ejecutado independientemente del sistema operativo en el que se ejecute. De allí toma el nombre de contenedor, permitiendo que pueda ser trasladado como un contenedor físico de un lugar a otro sin perder su contenido. A diferencia de una máquina virtual, un contenedor es más liviano ya que no cuenta con la capa del sistema operativo. Solamente añade la abstracción de paquetes y bibliotecas necesarias (dependencias) para la correcta ejecución del programa.

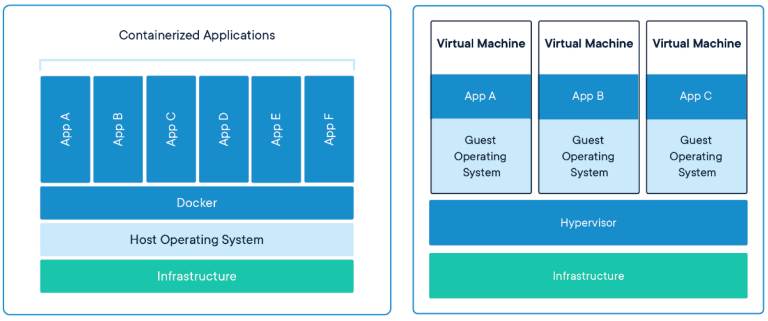

FIGURA 3.1: Comparación entre capas de abstracción entre contenedores y máquinas virtuales, extraído de: `https://www.docker.com/resources/what-container`

En la Figura 3.1 se observan los dos enfoques mencionados para la ejecución de un programa. Al utilizar máquinas virtuales hablamos de virtualización de plataforma, mientras que al utilizar contenedores se utiliza virtualización a nivel de sistema operativo. La diferencia primordial entre los dos tipos de virtualización que para la virtualización de plataforma se necesita un hipervisor que pueda ejecutar máquinas virtuales, cada una con su sistema operativo invitado. Al virtualizar varias máquinas virtuales el sistema

tiende a degradarse debido a la cantidad de recursos que el sistema debe dedicar a cada máquina virtual para poder virtualizar cada sistema operativo invitado, en donde debe ejecutarse el programa. Por otro lado, instalando Docker no necesitaremos un hipervisor, y no necesitaremos un sistema operativo por cada contenedor. De esta manera se pueden crear contenedores independientes sin necesidad de que cada uno se ejecute encima de un sistema operativo invitado, sino que se utiliza el sistema operativo huésped para su ejecución. Por esto último el enfoque de contenedores es claramente más liviano.

Dadas las grandes ventajas del uso de contenedores y la necesidad de agregar Integración Continua al repositorio principal de MateFun surge el uso de Docker para la Integración Continua en MateFun

3.3. Aplicación en MateFun

Se desea construir un pipeline de integración continua en el cual se pueda verificar rapidamente que todo código que se desea agregar al repositorio no genera regresión, es decir que no genera fallas en el sistema. Para lograr esto, es necesario definir primero un protocolo de contribución de código al repositorio[19]. Este protocolo es utilizado en la mayoría de los sistemas de código abierto:

- La rama (branch) principal del repositorio debe ser correcta en todo momento. No es aceptable una rama principal con errores.
- Todas las contribuciones de código fuente al repositorio deben crearse como pull requests a la rama principal, a partir de una rama separada. Una vez que este pull request sea correcto y no genere regresión al pasar correctamente por el pipeline de Integración Continua puede ser intregrado a la rama principal.
- Los administradores y algunos desarrolladores con ciertos permisos son los encargados de aceptar e integrar el código proveniente de pull requests a la rama principal del repositorio, mientras que los contribuidores no tienen permisos para realizar cambios a la rama principal.

Siguiendo este protocolo, cada contribución va a ejecutar un pipeline de integración continua con el cual se testea que el código del repositorio luego de incluir el nuevo código siga siendo correcto. Para ello es muy importante la correcta definición del pipeline.

3.3.1. Imagen Docker para ejecución de jobs

GitLab CI/CD soporta la ejecución de jobs con contenedores de Docker. Esto significa que al interpretar el pipeline a ejecutar, un runner carga un contenedor a partir de la imagen provista para realizar un job dentro del contenedor.

Resulta útil contar con una imagen Docker con un ambiente donde los requisitos para ejecutar MateFun ya se encuentren instalados. Cada job se ejecuta sobre un ambiente donde MateFun puede ejecutarse correctamente. Para ello se creó la imagen 'matefun-env' y se encuentra disponible en Docker Hub[20] para ser descargada.

La imagen tiene dos versiones (tags) disponibles, ambas basadas en una distribución Ubuntu Linux. En la Figura 3.2 se listan las distintas versiones. La versión 'latest' apunta por defecto a la última versión creada, en este caso 'ubuntu18.04'. Se tienen ambas versiones disponibles debido a que las versiones 16.04 y 18.04 son las últimas dos versiones con soporte a largo plazo (LTS: Long Term Support) de Ubuntu, teniendo soporte hasta los años 2021 y 2023 respectivamente[21].

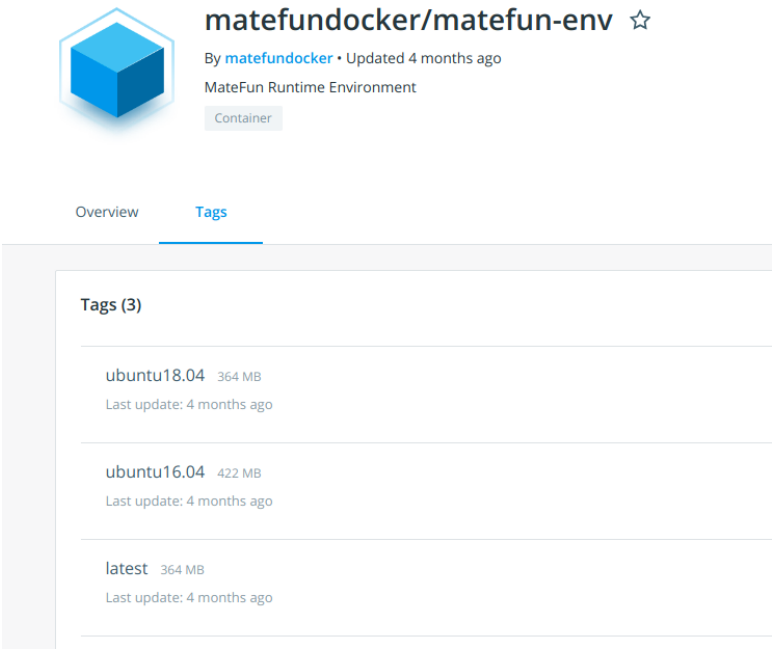

FIGURA 3.2: Versiones disponibles de la imagen matefun-env

La imagen matefun-env contiene el ambiente necesario para poder compilar, instalar y ejecutar MateFun. A continucación se presenta el archivo de definición de la imagen, creado a partir de una imagen Ubuntu. A partir de la imagen Ubuntu, se instala el compilador de Haskell GHC y el administrador de paquetes cabal para Haskell, así como

los paquetes necesarios para internacionalización y detección de intersección de dominios.

La definición de la imagen matefun-env versión ubuntu18.04 es la siguiente:

```
1  # MateFun environment Dockerfile
2  # Pull image from matefundocker/matefun-env
3  FROM ubuntu:18.04
4
5  RUN apt-get update && \
6      apt-get install -y ghc cabal-install gettext locales
7  ENV PATH=$PATH::/root/.cabal/bin
8
9  # Set up internationalization
10 RUN locale-gen es
11 RUN locale-gen en
12 ENV LANG=en_US.UTF-8
13
14 # Install Z3 4.6.0
15 # Releases on their GitHub repository: https://github.com/Z3Prover/z3/
       releases
16 RUN apt-get install -y wget unzip
17 RUN wget https://github.com/Z3Prover/z3/releases/download/z3-4.6.0/z3-4.6.0-
       x64-ubuntu-16.04.zip
18 RUN unzip z3-4.6.0-x64-ubuntu-16.04.zip
19 RUN cp z3-4.6.0-x64-ubuntu-16.04/bin/libz3.so /usr/lib/
20 RUN cp z3-4.6.0-x64-ubuntu-16.04/bin/z3 /usr/bin/
21 RUN cp z3-4.6.0-x64-ubuntu-16.04/include/* /usr/include/
22
23 # Update cabal and install dependencies
24 RUN cabal update && \
25     cabal install happy && \
26     cabal install hgettext && \
27     cabal install z3
```

Código 3.1: Definición imagen Docker matefun-env en el archivo Dockerfile

3.3.2. Definición de pipeline de integración continua

Un pipeline clásico de integración continua se compone de dos etapas principales: la etapa de compilación y la etapa de pruebas. Durante la etapa de compilación, la tarea en ejecución obtiene la versión del código fuente a partir del commit. Una vez obtenido el código fuente se intenta compilar el programa. Es usual que durante la compilación se ejecuten también tests unitarios que son pruebas estáticas, a nivel de código, con la cual se busca verificar que los métodos sean consistentes con el resultado esperado. Una vez generado el programa, se procede a la etapa de pruebas de integración, con la cual se necesita haber generado el programa previamente y se ejecutan pruebas tratando el programa como una caja negra.

Agregamos un pipeline de integración continua con estructura clásica basado en la imagen Docker 'matefun-env', con dos etapas:

- Etapa de compilación (build): en esta etapa se toma el código fuente y se intenta compilar
- Etapa de pruebas (test): en esta etapa se realiza un conjunto de pruebas predefinidas sobre el sistema

La etapa build está compuesta por un solo job mientras que la etapa de test está compuesta por dos jobs. Por cada job, el runner asignado a la ejecución inicializa un contenedor utilizando la imagen 'matefun-env' y ejecuta los comandos definidos en el job.

- El job 'build-matefun' en la etapa 'build' se encarga de compilar el código fuente. En caso de compilar correctamente, el job es exitoso y se procede a la siguiente etapa (debido a que es el único job de la etapa 'build'). En caso de error, el pipeline es incorrecto y se reporta el error.
- Los jobs de la etapa 'test' se ejecutan en paralelo si la cantidad de runners disponibles lo permite. Al contar con un solo runner se ejecuta uno después del otro sin un orden en particular.
    - El job 'test-internationalization' se encarga de agregar un nuevo idioma, generar traducciones para ciertas claves, ejecutar MateFun con ese idioma y

verificar que se realiza la traducción. En caso que se realice correctamente, el job es exitoso.

- El job 'test-runtests' invoca a un script que explora el directorio de pruebas por cada idioma e itera entre los archivos de prueba disponibles, en donde por cada prueba necesita:
  - Un archivo de entrada con definición de funciones MateFun
  - Un archivo de entrada de comandos, con los comandos a ejecutar en la instancia de MateFun
  - Un archivo de salida, para comparar la salida actual con la esperada

  El job es exitoso cuando la salida producida por MateFun dados los archivos de entrada es igual a la salida esperada para cada prueba.

```
1  image: matefundocker/matefun-env:ubuntu18.04
2
3  stages:
4    - build
5    - test
6
7  build-matefun:
8    stage: build
9    script:
10     # Build MateFun
11     - cabal install --only-dependencies
12     - cabal build
13
14 test-internationalization:
15   stage: test
16   script:
17     # Install MateFun
18     - cabal install
19
20     # Give execution permissions
21     - chmod +x translations.sh
```

```
22
23    # Generate latest .pot file with every key that needs translation
24    - sh translations.sh -g
25
26    # Add a new language. A .po file is created. Translations need to be
       added to the .po file
27    - ./translations.sh -a pt
28
29    # Translations into the .po file
30    - sed -i '/msgid "No File"/{n;s/msgstr ""/msgstr "Nao Arquivo"/;}'
       internationalization/pt.po
31    - sed -i '/msgid "exit"/{n;s/msgstr ""/msgstr "sair"/;}'
       internationalization/pt.po
32    - sed -i '/msgid "Bye!!"/{n;s/msgstr ""/msgstr "Tchau!!"/;}'
       internationalization/pt.po
33
34    # Generate .mo file from the translations .po file
35    - ./translations.sh -m pt
36
37    # Invoke MateFun using the new language and verify translations
38    - LANGUAGE=pt MateFun <<< $'!sair' | grep "Nao Arquivo"
39
40 test-runtests:
41   stage: test
42   script:
43    # Install MateFun, set execution permissions
44    - cabal install
45    - chmod +x runtests.sh
46
47    # Run tests
48    - ./runtests.sh
```

Código 3.2: Definición del pipeline de integración continua en MateFun en el archivo .gitlab-ci.yml

De forma gráfica, el pipeline definido se puede ver en la Figura 3.3

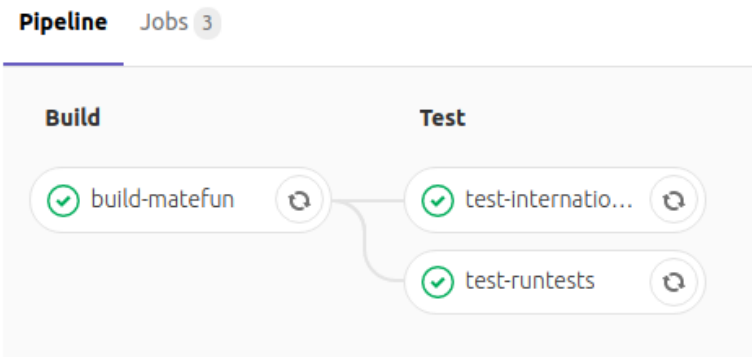

FIGURA 3.3: Pipeline de integración continua de MateFun

La ejecución de pipelines en MateFun está restringida a un runner compartido en la infraestructura de Facultad. Como muestra la Figura 3.4 se tienen dos runners compartidos registrados, de los cuales solo uno está activo.

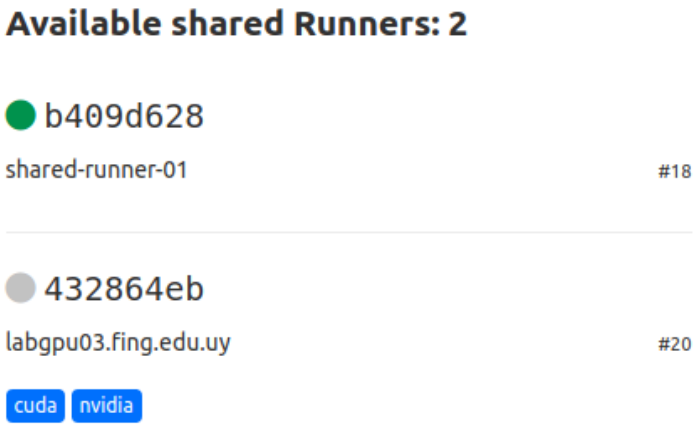

FIGURA 3.4: Runners compartidos en MateFun

Capítulo 4

Proceso de contribución a MateFun

Al no existir un proceso de contribución definido, las contribuciones al repositorio se hacían directamente a la rama principal, sin verificación ni pruebas. Esto implica que se puedan generar regresiones si el código a incluir no se testea completamente. Además, tener un registro histórico y navegable sobre los cambios y contribuciones hechas al repositorio se vuelve tedioso, debiendo indagar commit por commit hasta encontrar cuál de ellos fue el que introdujo el cambio.

Para poder mejorar el uso del repositorio de MateFun se define un proceso de contribución, estableciendo los protocolos esperados para la adecuada interacción con el repositorio.

4.1. Motivación

Un proceso de contribución a un repositorio establece uno o varios protocolos entre los contribuidores sobre la forma en que se espera interactuar. Se busca introducir un nuevo procedimiento buscando ser la base para futuras contribuciones, basada en proyectos de código abierto, lo que permite realizar verificaciones antes de incluir código a la rama principal del repositorio. Si bien MateFun no es un proyecto de código abierto, se obtendrían beneficios al aplicar algunas de las prácticas[19] de estos proyectos. Por ejemplo, el proyecto Apache Tomcat tiene definido un proceso específico y detallado en su repositorio[22], con el cual nuevos contribuidores pueden consultar y seguir los lineamientos esperados por toda la comunidad.

4.2. Contribuciones

No toda forma de contribución es mediante código fuente. Una buena fuente de contribución son los usuarios, en este caso los estudiantes que utilizarán la herramienta. Los mismos pueden aportar sugerencias y/o errores encontrados al utilizar la herramienta. Al estar MateFun alojado en GitLab, este nos provee los medios necesarios para reportar errores encontrados (Issues) y resolverlos agregándolos a la rama principal (Merge Requests - equivalente a pull requests mencionados en el capítulo anterior). Los usuarios reportan errores detallando la forma de reproducirlos, el comportamiento esperado y el comportamiento obtenido. De esta forma, los contribuidores desarrolladores pueden tomar los Issues abiertos y resolverlos. Una vez que se resuelven, crean un Merge Request con los cambios en el código.

De forma rápida, GitLab nos permite navegar por los distintos Issues o Merge Requests a través del menú lateral del repositorio como se muestra en la Figura 4.2.

4.3. Reportar un problema encontrado mediantes Issues

Al navegar a la sección Issues se listan los problemas existentes según su estado: abiertos si aún no se han resuelto o cerrados si ya se han resuelto. Al poder tener visibilidad de los problemas existentes se evita que se creen Issues duplicados. Por lo tanto la primer parte en el reporte de un problema encontrado es asegurarse que no haya sido reportado previamente. En caso que no haya sido reportado, se debe proceder a crear el reporte mediante el botón 'New Issue'.

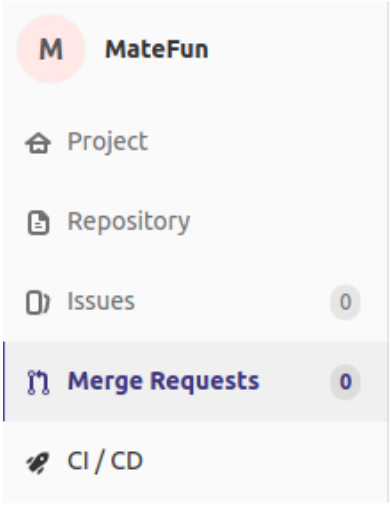

FIGURA 4.1: Menú lateral del repositorio MateFun en GitLab

Un issue debe ser creado con un título descriptivo, y una descripción detallada del problema indicando como mínimo la versión del programa y como reproducirlo.

Los usuarios tienen acceso a la lista de issues del repositorio. Un issue tiene dos estados: abierto (se encuentra activo, es decir, no ha sido resuelto) y cerrado (el problema ha sido resuelto o no es realmente un problema).

La lista de issues abiertos es de gran utilidad para los desarrolladores, porque les permite trabajar en ellos, resolviendo los problemas encontrados. Un desarrollador se debe asignar a un issue, y porteriormente crear un merge request para resolver el problema reportado.

4.4. Creacion de merge requests

Un desarrollador puede crear merge requests, con las cuales propone nuevo código para que ingrese al repositorio. Este código puede agregar una nueva funcionalidad, o resolver algún problema reportado, o no reportado.

La lista de merge requests activos puede encontrarse en el menú izquierdo del repositorio de MateFun con nombre 'Merge Requests'. A nivel de configuración del repositorio, los administradores pueden decidir si un pull request debe ser aprobado por al menos un administrador, o si no es necesario. Esto permite un control mayor sobre el nuevo código, además de testear de forma aislada cada pull request.

Una vez aprobado un merge request, se incluye en el repositorio del proyecto. Los pasos a seguir para crear un merge request son:

- Realizar una copia (fork) del repositorio principal al repositorio del usuario
- Obtener la última versión de la rama principal del repositorio
- Crear una rama específica para cada contribución (feature branch) a partir de la rama principal actualizada
- Hacer push de la rama específica al repositorio del usuario.
- Desde la web de GitLab, crear un merge request desde la rama específica del repositorio del usuario a la rama principal del repositorio principal.

Los comandos a utilizar para realizar estos pasos se detallan en la sección Contribución de código fuente[1] en la Wiki del proyecto MateFun.

4.5. Protección de la rama principal del repositorio

Para asegurar que el proceso de contribución se realice correctamente y prevenir que usuarios introduzcan código sin ser verificado, se añade un nivel más de protección a la rama principal del repositorio.

Este caso se le brinda la posibilidad de pushear directamente a la rama principal únicamente a los miembros del proyecto con nivel Maintainer o superior. Miembros del proyecto con nivel menor requieren seguir el protocolo definido previamente:

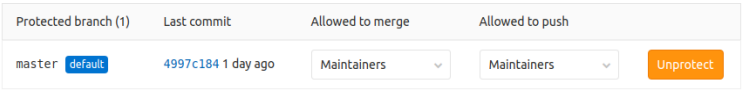

FIGURA 4.2: Rama principal de MateFun protegida de push

[1] `https://gitlab.fing.edu.uy/matefun/MateFun/wikis/contribucion/codigo`

Capítulo 5

Internacionalización en MateFun

En esta sección se describe el proceso con el cual brindar soporte a varios idiomas en MateFun y poder brindar al usuario una forma fácil de elegir el idioma al invocar el programa. En particular se requería incluir como mínimo los idiomas inglés y español, y dejar sentadas las bases para agregar nuevos idiomas de manera simple y sin reescribir el código fuente al querer agregar un nuevo idioma.

Nos dedicaremos a internacionalizar MateFun sin especificar en localización, es decir, daremos soporte a distintos idiomas sin configuraciones regionales. Por convención, los lenguajes siguen el formato ISO 639-1[23] en el cual se utilizan dos letras minúsculas para representar cada lenguaje. En el caso particular de MateFun incluiremos: 'es' y 'en' correspondientes a Español e Inglés respectivamente.

5.1. Preparación del código fuente

El primer paso en el camino a la internacionalización es la preparación del código fuente. De forma simple, esto consiste en detectar cuáles son los textos a traducirse dado un idioma soportado en la ejecución de MateFun. Una vez identificados los textos a traducir, se incluyen en inglés en el código fuente dado que el inglés será el idioma por defecto. Esto significa que las claves generadas para los archivos de traducción se encontrarán en inglés.

5.2. Aplicación en MateFun

La motivación era poder brindar soporte a múltiples idiomas a MateFun. Como hemos definido en las secciones anteriores, conocemos la herramienta de traducción gettext y su funcionamiento. Además sabemos de la existencia de un wrapper de la herramienta para Haskell, lenguaje en el cual está escrito MateFun. La solución al problema de internacionalizar MateFun se reduce a poder adaptarlo a una forma conocida para gettext, herramienta que ya sabemos nos garantiza la traducción.

La solución planteada puede verse en alto nivel en la Figura 5.1. En ella podemos observar en primera instancia una separación de niveles o capas según la responsabilidad de los actores involucrados.

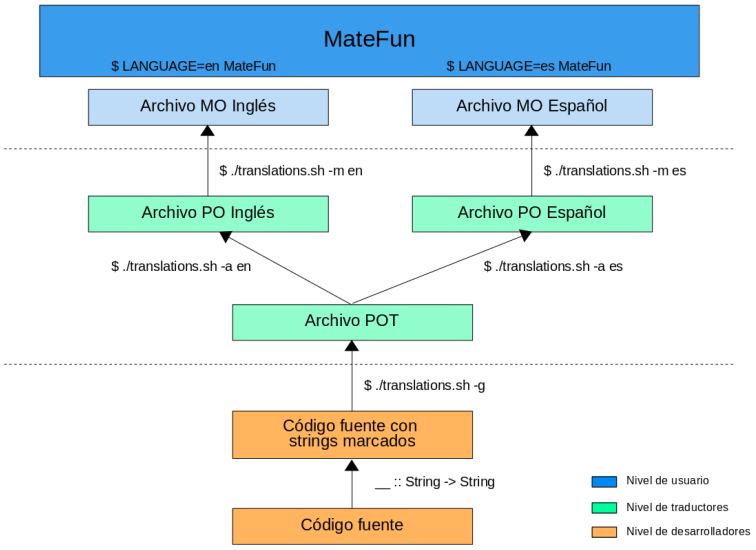

FIGURA 5.1: Esquema de internacionalización en MateFun

A mayor nivel de abstracción, la solución permite a los usuarios ser capaces de realizar invocaciones a MateFun seleccionando el idioma deseado, entre los idiomas disponibles. Como mencionamos anteriormente, los idiomas disponibles son Inglés y Español. La selección del idioma se logra al setear la variable de entorno *LANGUAGE* al idioma deseado.

El desafío se reduce a generar un buen proceso de anotación de claves en el código fuente de MateFun y de generar un proceso que permita unificar todos los niveles mencionados anteriormente para lograr la traducción de texto a los distintos idiomas.

5.2.1. Anotación de claves en MateFun

El primer paso de este proceso fue de investigación de hgettext. Este paquete es un wrapper de gettext, y nos permite econtrar una solución simple[24] para anotar claves en el código fuente en Haskell.

Es preciso notar que el objetivo de traducción del intérprete de MateFun no son solamente los mensajes mostrados al usuario, sino que se busca traducir también los comandos del intérprete, funciones predefinidas y palabras clave. Al ser el intérprete también un compilador del lenguaje MateFun, y dado que este lenjuage soporta distintos idiomas, se podría decir que MateFun es también un compilador de una familia de lenguajes de MateFun.

Agregando la dependencia a hgetttext en el código de MateFun, nos aseguramos de definir una función que dado un string a traducir devuelva su string traducido:

```
1 import Text.I18N.GetText
2 import System.IO.Unsafe
3
4 __ :: String -> String
5 __ = unsafePerformIO . getText
```

De esta forma, al aplicar la función `__` a cada string a traducir en el código fuente, completamos el proceso de anotación de claves. Esta función nos asegura mediante la librería hgettext poder hacer uso de gettext para devolver el string adecuado según el lenguaje seleccionado.

Creamos un nuevo módulo en el código fuente llamado 'InternationalizationHelper' en donde definimos la función `__`. Además definimos el módulo 'ReservedNames' con los nombres de los comandos y funciones predefinidas, para facilitar su visibilidad en el ćodigo fuente. En la Figura 5.2 se presenta el diagrama de módulos actualizado de MateFun luego de este trabajo. Se puede observar que los nuevos módulos 'InternationalizationHelper' y 'ReservedNames' son utilizados por la mayoría de los módulos originales, tomando un rol central en el diagrama.

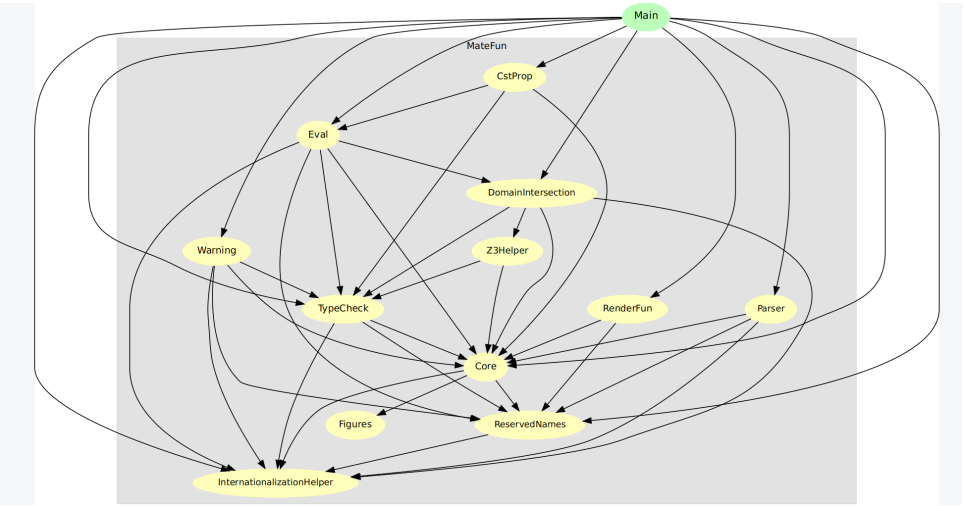

FIGURA 5.2: Diagrama de módulos actualizado de MateFun

5.2.2. Directorio de internacionalización

Por sí solo, el proceso de anotación de claves en el código fuente no nos asegura la traducción completamente. Gettext necesita conocer un directorio de internacionalización en donde poder buscar los archivos MO según el idioma que se requiera. Dentro de este directorio de internacionalización, gettext espera encontrar los archivos MO en la siguiente ruta:

```
1 DIR_INTERNACIONALIZACION/LANG/LC_MESSAGES/
```

Se creó un nuevo directorio llamado 'internationalization' dentro del directorio raíz de MateFun, siguiendo la estructura mostrada en la Figura 5.3

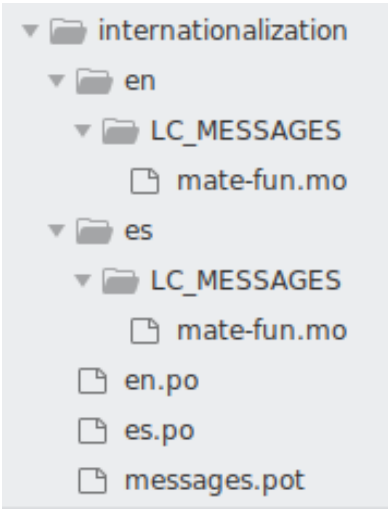

FIGURA 5.3: Directorio de internacionalización en MateFun

La plantilla de archivos PO (archivo POT) y los archivos PO para cada idioma se crean en este directorio. Por convención, los nombres de los archivos PO siguen el formato ISO 639-1[23]. Además se agrega un directorio por cada idioma, siguiendo el mismo formato de nombre, que contendrán los binarios necesarios para la traducción a los distintos idiomas.

Al contar con este directorio, solamente resta explicitar en el código fuente donde se encuentra ese directorio, por eso definimos las siguientes funciones en el módulo 'MateFun.InternationalizationHelper':

```
1 hgettext_locale_dir = "internationalization"
2 hgettext_locale_domain = "mate-fun"
```

Estas representan el nombre del directorio de internacionalización en MateFun, relativo a la raíz del repositorio y el nombre de los archivos MO esperado por gettext para cada idioma, ya que agregamos al main de MateFun lo siguiente:

```
1 setLocale LC_ALL (Just "")
2 bindTextDomain hgettext_locale_domain (Just hgettext_abs_dir)
3 textDomain (Just hgettext_locale_domain)
```

El último paso para asegurar la internacionalización según la solución planteada en la Figura 5.1 es tener una herramienta que actúe como un nexo entre todos los niveles, representado por las flechas en la figura, permitiéndo de forma automática realizar todos los pasos esperados para lograr la traducción. Implementamos esta herramienta como un script bash.

5.2.3. Script de internacionalización

En la raíz del directorio MateFun creamos el archivo 'translations.sh' con el cual es posible:

- Extraer las claves anotadas del código fuente y generar el archivo POT con todas las claves a traducir.
- Generar un archivo PO dado un idioma en las que se deben agregar las traducciones de cada clave.

- Generar el archivo MO con el nombre esperado por gettext, a partir del archivo PO correspondiente al idioma.

Estas tres tareas se realizan al ser invocadas como se explica en su mensaje de ayuda (Figura 5.4).

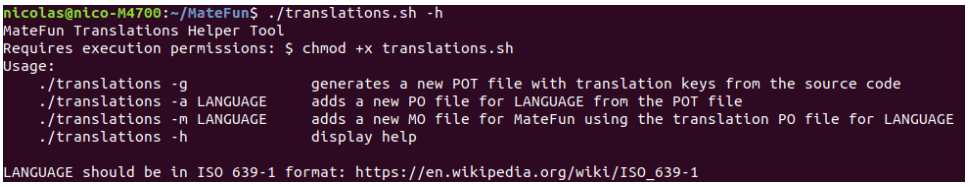

FIGURA 5.4: Script de internacionalización en MateFun

Una vez anotados las claves a traducir en el código fuente, se puede generar el archivo POT simplemente invocando al script con la opción g:

```
1 ./translations.sh -g
```

Se buscan todas las ocurrencias de '__' en el código fuente, específicamente dentro del directorio 'src' y por cada ocurrencia se agrega un par (clave, traducción) donde la traducción es vacía.

```
1 msgid "CLAVE1"
2 msgstr ""
3
4 msgid "CLAVE2"
5 msgstr ""
```

Cada vez que se invoca esta opción, el archivo messages.pot es regenerado. Por lo tanto luego de anotar una nueva clave en el código fuente, debe regenerarse el archivo POT.

Para generar un archivo PO para cierto lenguaje o regenerarlo (para traducciones ya existentes), se debe invocar al script de la siguiente manera:

```
1 ./translations.sh -a LANG
```

donde LANG es un idioma en formato ISO 639-1[23]. Inicialmente, los idiomas disponibles son inglés y español (en y es respectivamente).

Esta invocación genera un nuevo archivo LANG.po en el directorio internationalization si no existía previamente a partir del archivo POT. Si el archivo PO ya existia

previamente, las traducciones existentes no se reemplazan y solamente se quitan las claves que ya no estan presentes o se agregan las nuevas claves para que sean traducidas.

En este punto es cuando se deben generar las traducciones. Si se cuenta con traductores, su trabajo es especificamente traducir las claves anotadas y guardar los cambios en los archivos PO. En el caso de MateFun, al no contar con traductores dedicados, se realizaron todas las traducciones al idioma español. Por defecto MateFun ejecutará en inglés.

El último paso en el proceso de internacionalización es generar los archivos MO necesarios para que se interpreten por la máquina mientras ejecuta el software. Estos se generan invocando:

```
1 ./translations.sh -m LANG
```

Una vez generados los archivos binarios ya queda completo el ciclo de internacionalización, permitiendo que el usuario pueda seleccionar entre los idiomas soportados.

5.2.4. Selección de idiomas en MateFun

Actualmente los idiomas disponibles son Inglés y Español, siendo Inglés el idioma por defecto. Para poder seleccionar uno de los idiomas disponibles se debe invocar a MateFun seteando la variable 'LANGUAGE' a el idioma que se desee, ya sea inglés (en) o español (es). De esta manera, MateFun utilizará los archivos MO para poder mostrar las traducciones.

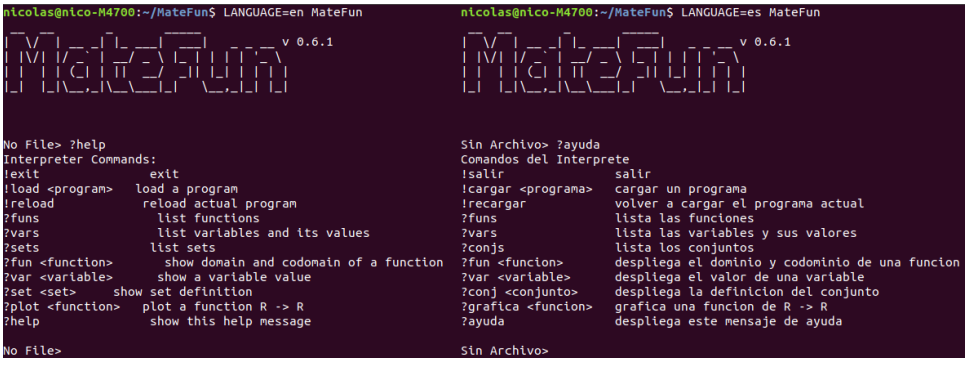

FIGURA 5.5: Selección de idioma en MateFun

Como muestra la Figura 5.5 no solo el texto mostrado al usuario es traducido, sino también los comandos esperados.

Capítulo 6

Detección de intersección de dominios en funciones por partes en MateFun

Este capítulo se centra en el estudio de intersección de dominios en funciones definidas por partes en MateFun. La definición y el uso correcto de funciones definidas por partes fue una limitación encontrada en el Estado del Arte. Se mostrarán diversas opciones para atacar el problema de detectar la intersección de dominios en las funciones definidas por partes, y la posterior elección de la biblioteca Z3 como solución al problema. Brindaremos detalles sobre el funcionamiento de la biblioteca, asi como sus limitaciones. Finalmente, se explicará como ha sido realizada la integración de la biblioteca elegida a MateFun.

6.1. Funciones definidas por partes

Comenzamos el capítulo con un breve repaso de las funciones matemáticas. Por definición, una función es una relación entre dos conjuntos llamados dominio y codominio, en la que a cada punto del dominio le corresponde un único punto del codominio. Considerando una función de nombre f, dominio A y codominio B, podemos representarla como:

$$f : A \subseteq \mathbb{R}^n \to B \subseteq \mathbb{R}^m, \quad f(x_1, x_2, ..., x_n) = (y_1, y_2, ..., y_m) \tag{6.1}$$

A cada punto $(x_1, x_2, ..., x_n)$ del dominio A le corresponde un único punto $(y_1, y_2, ..., y_m)$ del codominio B.

Respetando la restricción anterior, se puede dividir el dominio A en conjuntos disjuntos, asignando un valor funcional a cada uno de ellos. De esta manera se obtiene una función definida a trozos:

$$A = \bigcup_{i=1}^{n} A_i, \quad A_i \cap A_j = \emptyset, \quad i, j \in [1, .., n], \quad i \neq j \tag{6.2}$$

$$f(x_1, x_2, ..., x_n) = \begin{cases} f_1(x_1, x_2, ..., x_n) & \text{si } (\mathrm{x_1}, \mathrm{x_2}, ..., \mathrm{x_n}) \in \mathrm{A_1} \\ f_2(x_1, x_2, ..., x_n) & \text{si } (x_1, x_2, ..., x_n) \in A_2 \\ ... & \\ f_n(x_1, x_2, ..., x_n) & \text{si } (x_1, x_2, ..., x_n) \in A_n \end{cases} \tag{6.3}$$

La función f representada en la Ecuación 6.3 se llama función definida por partes.

6.2. Funciones por partes en MateFun

El intérprete MateFun permite al usuario cargar funciones definidas en archivos de texto. En estos archivos, los usuarios realizan las definiciones de las funciones a ser cargadas.

Las funciones definidas por partes están permitidas en MateFun, sin embargo, como se ha explicado en el estado del arte, tienen una limitación. Lo veremos con un ejemplo.

Si un usuario define archivo de texto MateFun:

```
1 part :: R X R -> R
2 part(x,y) = 1 si x > 0
3         o -1 si y > 0
4         o 0
```

MateFun no puede detectar que la función *part* no cumple la restricción 6.2, por lo cual no cumple la definición de función por partes (6.3). Sin embargo, realiza la carga de *part* como si fuera una función. Para los puntos del dominio que pertenecen a más de un subconjunto, devuelve siempre el valor funcional del primer conjunto al que encuentra que pertenece, siguiendo el orden definido en el archivo de texto. Es decir, que para MateFun, *part* es una función por partes (la llamaremos *partm*), $partm : \mathbb{R}^2 \to \mathbb{R}$ definida como:

$$partm(x,y) = \begin{cases} 1 & \text{si x} > 0 \\ -1 & \text{si y} > 0 \\ 0 & \text{sino} \end{cases} \tag{6.4}$$

Al no cumplirse la restriccion 6.2, los puntos del dominio pertenecientes al conjunto $X = \{(x,y) \in \mathbb{R}^2 / x > 0, y > 0\}$ harían que la función *partm* tome dos valores funcionales, ya que dos de los casos definidos se hacen verdaderos, violando la definición de función.

Debido a esta limitación en MateFun, es deseable realizar chequeos de los subconjuntos del dominio en las potenciales funciones que el usuario ingresa en los archivos de texto. Es decir, verificar que se cumplen las restricciones planteadas en la Ecuación 6.2 para funciones por partes. De esta manera, si se pudiera detectar que no se cumple la restricción, reportar el error y no cargarla como función, para que el usuario pueda corregir los errores en la definición.

6.3. Método de detección de intersección de dominios en funciones definidas por partes

Extendiendo el ejemplo anterior a un problema general, debemos considerar en primer lugar una función definida por partes como 6.3 sobre la cual se quiere estudiar si los dominios definidos son disjuntos, es decir, si cumplen la restricción 6.2.

Una forma análoga de representar la restricción 6.2 es:

$$\nexists x \in A / x \in A_i \wedge x \in A_j \quad \forall i, j \in [1, .., n], \quad i \neq j \tag{6.5}$$

Si definimos las siguientes variables booleanas:

$$c_i = x \in A_i \quad \forall i \in [1, .., n] \tag{6.6}$$

La ecuación 6.5 se puede representar como:

$$\nexists x \in A / c_i \wedge c_j \quad \forall i, j \in [1, .., n], \quad i \neq j \tag{6.7}$$

Lo que equivale a que el siguiente sistema no es satisfacible:

$$c_i \wedge c_j \qquad \forall i, j \in [1, .., n], \quad i \neq j \tag{6.8}$$

Esto significa que la condición para considerar una función definida por partes como válida es que no exista una expresión satisfacible en el sistema 6.8. Si existe al menos una expresión satisfacible entonces sabremos que hay intersección de dominios y la relación definida no es una función.

6.4. Aplicación en MateFun

6.4.1. Solución en alto nivel

Para poder determinar si hay intersección en los dominios de definición de las funciones se utiliza la biblioteca Z3 y su wrapper en Haskell también llamado z3. Z3 es un probador de teoremas de bajo nivel que permite determinar si un conjunto de condiciones es satisfactible.

El enfoque seguido en MateFun consiste en comparar las condiciones de cada función definida por partes de a pares. Es decir, se consideran las combinaciones de N elementos (condiciones) tomadas de a dos elementos y se comparan mediante la biblioteca Z3.

Se agrega una nueva flag para determinar si MateFun debe analizar si hay intersección de dominios en las funciones definidas por partes.

- Si la flag no se encuentra activa al momento de la invocación a MateFun, entonces no se realiza chequeo de intersección de dominios
- Si la flag se encuentra activa al invocar a MateFun, entonces MateFun realizará un chequeo estático al cargar el programa, y en caso de ser necesario un chequeo dinámico de intersección en dominios de funciones por partes.

El chequeo estático de intersección de dominios al cargar un programa consiste en analizar las condiciones de a pares como se explicó anteriormente, intentando determinar si hay una intersección entre alguna de las condiciones, es decir, si existen valores del dominio que cumplan más de una condición. Z3 se encarga de realizar el chequeo, y puede obtener tres resultados:

- SAT: Existe una variable que satisface las condiciones. Por lo tanto existe intersección de dominios y no debe cargarse la función ya que no cumple con la definición de función.
- UNSAT: Las condiciones no son satisfactibles, por lo tanto no hay intersección de dominios entre ellas y se debe seguir analizando las condiciones de la función.
- UNKNOWN: Z3 no puede determinar si las condiciones son satisfactibles o no con la información dada. En este caso, marcamos la función para posterior chequeo en tiempo de ejecución cada vez que se invoque. Las razones por las cuales Z3 podría no poder determinar un resultado pueden ser que termine la memoria en el programa, que la fórmula no sea determinable debido a la naturaleza del problema como en el caso de aritmética no lineal con enteros o también por falta de implementación en formas de resolver algunos problemas.

Al evaluar una función en tiempo de ejecución que ha sido marcada como UNKNOWN, MateFun realiza evaluaciones de cada una de las condiciones de una función definida por partes con el elemento del dominio que ingresa el usuario.

- Si existe más de una evaluación verdadera para un elemento del dominio, se excluye el punto del dominio de la función mostrando un error al usuario.
- Si existe solamente una evaluación verdadera para un elemento del dominio, MateFun procede a devolver su valor funcional.

6.4.2. Implementación

Como se explicó en la solución de alto nivel, se introducen dos nuevos chequeos:

- Al cargar un programa al intérprete: se realiza un chequeo estático de las funciones, verificando la satisfacibilidad mediante Z3.
- Al evaluar una función: se realiza un chequeo dinámico de la función en el punto a evaluar, cuando no se pudo determinar la satisfacibilidad de las condiciones en el chequeo estático. El chequeo dinámico consiste en verificar si solo una condición se hace verdadera al evaluar la función en el punto del dominio.

6.4.2.1. Evaluación estática de intersección de dominios

En el módulo principal, módulo MateFun, se encuentra una de las funciónes centrales del intérprete, la función 'evalInterp':

```
1 evalInterp :: Interp -> Ctx -> IO (Maybe (String, Ctx))
```

Básicamente, el intérprete está en loop a la espera de una entrada por parte del usuario. Al recibir una entrada, se invoca a la función 'evalInterp', se genera una salida y se vuelve a esperar por la siguiente entrada.

Del tipo de datos *Interp* nos interesa en particular el constructor *LoadString* en cual permite cargar el contenido de un archivo al intérprete.

Originalmente, 'evalInterp' aplicada al constructor Load realizaba lo siguiente:

- Paso 1: Parseo del archivo a cargar. En caso de error en el parseo, se reporta el error y no se cargan las funciones y/o conjuntos en el intérprete. Si no hay errores se crea un contexto con las funciones y conjuntos parseados
- Paso 2: Chequeo de tipos de las funciones y conjuntos parseados, para verificar si son correctos.
- Paso 3: Propagación de constantes. En este último paso se busca simplificar las ecuaciones definidas reemplazando los valores constantes conocidos por su valor, simplificando ecuaciones de funciones.

En esta sección, extenderemos la funcionalidad de 'evalInterp' al cargar un archivo, añadiendo un nuevo paso entre los pasos 2 y 3 mencionados arriba. Este nuevo paso es el de chequeo de intersección de dominios. Es importante notar la posición que ocupa este nuevo paso en el proceso realizado por 'evalInterp'. Es necesario primero realizar un chequeo de tipos y asegurarnos que estos son correctos antes de analizar si hay intersección en los dominios definidos en funciones por partes.

```
1  evalInterp (Load fn) ctx = do
2
3    -- Paso 1: Parseo
4    let pre   = snd $ ctxFile ctx
5    let carga = (__"Loading") ++ " " ++ fn ++ "\n"
6        errorMsg = "Error: {" ++ (__"file") ++ ": " ++ fn
7    prg <- parseFiles [fn] pre [] [] []
8
9    case prg of
10     Left err  -> just (customShow err , emptyCtx ctx)
11     Right ans -> do
12
13       -- Paso 2: Chequeo de tipos
14       case runCheck ans (initCtx fn pre (wrn ctx) (wrnA ctx)
15                          (ctxDomInt ctx) (ctxVerbose ctx) ans) of
16         (ctx,[]) -> do
17           let ctx' = propCtxLoadedFuns ctx
18           -- Paso intermedio: Chequeo de dominios (si flag -d)
19           if ctxDomInt ctx'
20           then case runDomainIntersection ctx' of
21             Left err -> just (carga ++ show err, emptyCtx ctx')
22             -- Paso 3: Propagacion de constantes
23             Right newfuns -> constantsPropagation ctx' carga newfuns
24           else constantsPropagation ctx' carga (loadedFuns . ctxE $ ctx')
25         (_,errs) -> just (carga ++ showErrors errs, emptyCtx ctx)
```

Código 6.1: Carga de un programa MateFun

En la línea 17 de 6.1 se crea un nuevo contexto a partir del contexto actual, en el cual se propagan las funciones declaradas a un nuevo formato definido, con el cual facilitar el chequeo estático como veremos más adelante.

```
1  data LoadedFunction = LoadedFunction {
2      ...
3      isEvalUnknown :: Bool
4  }
5
6  extrVarsTy :: Ty -> [Ty]
7  extrVarsTy (TyProd x) = x
8  extrVarsTy x = [x]
9
10 expConds :: ExpG -> [Cond]
11 expConds (ExpSi _ cond gelse _) = cond : expConds gelse
12 expConds (Else _ _) = []
13
14 toLoadedFuns :: [(NFun, ([NVar],ExpG))] -> [(NFun,(Ty,Ty))] -> [(NConj,
       TConj)] -> [LoadedFunction]
15 toLoadedFuns list1 list2 conjs
16     = [LoadedFunction {funName = name1,
17                        funVars = zip vars (extrVarsTy varsTy),
18                        funRetTy = retTy,
19                        funExpG = expg,
20                        funConds = expConds expg,
21                        funConjs = conjs,
22                        isEvalUnknown = False
23                       }
24         |   (name1, (vars, expg)) <- list1,
25             (name2, (varsTy, retTy)) <- list2,
26             name1 == name2
27       ]
28
29 propCtxLoadedFuns :: Ctx -> Ctx
```

```
30 propCtxLoadedFuns ctx = ctx {
31     ctxE = (ctxE ctx) {
32         funs = (funs . ctxE $ ctx),
33         loadedFuns = toLoadedFuns (funs . ctxE $ ctx) (ctxF ctx) (ctxC
       ctx)
34     }
35 }
```

El nuevo contexto contiene las LoadedFunction cargadas e inicializadas como evaluación Unknown en falso. Cuando la flag de chequeo de intersección de dominios se encuentra activa, entonces se realiza el nuevo chequeo (línea 24 de 6.1).

La función 'runDomainIntersection' se encuentra definida en un nuevo módulo llamado 'DomainIntersection'. En este módulo se definen todas las funciones necesarias para lograr plantear la resolución del problema presentado a la biblioteca Z3. Esta función chequea los dominios de cada función definida, analizando las condiciones (o casos) definidos en la función y tomando las combinaciones sin repetición tomadas de a dos de estas condiciones. Es importante notar que estas combinaciones se consideran para poder evaluar todas las condiciones una con cada una de forma mínima. De cada par de elementos de las combinaciones posibles, se analiza la satisfacibilidad de la teoría formada por ambas fórmulas. A continuación se muestran las funciones más relevantes en el proceso mencionado, primero generando las combinaciones sin repetición de todas las condiciones de la función y luego realizando evaluaciones de a pares mediante Z3:

```
1 data DomainIntersection = NoIntersection Cond Cond Ann Ann
2                         | Intersection  Cond Cond Ann Ann
3                         | UnknownIntersection Cond Cond Ann Ann
4
5 {-
6   Analisis de interseccion de dominio de una funcion, de a pares de
      condiciones, tomando las combinaciones de estas condiciones
7 -}
8 verifyDomainExpG :: LoadedFunction -> [LoadedFunction] -> Bool -> Either
      Error LoadedFunction
9 verifyDomainExpG fun functions verbose
```

```
10          | null evaluations = Right newfun
11          | intersection = Left err
12          | otherwise = Right newfun
13          where condsanns = getExpConds (funExpG fun)
14                combinationsWithoutRep = [(x,y) | (x:rest) <- tails
    condsanns , y <- rest ]
15                evaluations = [checkSat x y ann1 ann2 fun functions verbose
     | ((x,ann1),(y,ann2)) <- combinationsWithoutRep]
16                (intersection, err) = isIntersection fun evaluations
17                newfun = fun {isEvalUnknown = isUnknown evaluations}
18
19  {-
20      Se invoca a Z3 para analizar satisfacibilidad del par de condiciones
    para determinar el resultado
21  -}
22  checkSat :: Cond -> Cond -> Ann -> Ann -> LoadedFunction -> [
    LoadedFunction] -> Bool -> DomainIntersection
23  checkSat cond1 cond2 ann1 ann2 fun functions verbose =
24    case unsafePerformIO $ run cond1 cond2 fun functions verbose of
25      Unsat -> NoIntersection cond1 cond2 ann1 ann2
26      Sat -> Intersection cond1 cond2 ann1 ann2
27      Undef -> UnknownIntersection cond1 cond2 ann1 ann2
```

Se define también un nuevo módulo llamado Z3Helper. Este nuevo módulo permite la interacción entre MateFun y el solver Z3 mediante el wrapper que nos provee la librería z3 de Haskell. El objetivo de este módulo es generar una entrada válida para Z3 de manera que éste pueda chequear la satisfacibilidad de las fórmulas de primer orden planteadas de a pares de condiciones.

Por lo tanto, dado un par de condiciones, este módulo ayuda a generar un string de entrada a Z3 verificar la satisfacibilidad de cada par de condiciones de la función.

El formato esperado por Z3 es un string en el formato definido por el estándar

SMT-LIB 2.0[25]. Mediante el módulo Z3Helper, llevaremos el chequeo de un par de condiciones a la verificación de satisfacibilidad en el formato esperado por Z3.

```
1  run :: Cond -> Cond -> LoadedFunction -> [LoadedFunction] -> Bool-> IO (
       Result)
2  run cond1 cond2 fun functions verbose =
3    do
4      if verbose
5      then putStrLn $ "Evaluation on Z3 for conditions: " ++ show cond1 ++
       " and " ++ show cond2 ++
6             ":\n" ++ generateZ3Str cond1 cond2 fun functions ++ "\n"
7      else return ()
8      evalZ3 (script cond1 cond2 fun functions) >>= return
9
10 script :: Cond -> Cond -> LoadedFunction -> [LoadedFunction] -> Z3 Result
11 script cond1 cond2 fun functions = do
12     l <- parseSMTLib2String (generateZ3Str cond1 cond2 fun functions) []
       [] [] []
13     assert l
14     check
15
16 generateZ3Str :: Cond -> Cond -> LoadedFunction -> [LoadedFunction] ->
       String
17 generateZ3Str cond1 cond2 fun functions =
18         generateEnumDeclaration (funConjs fun) ++
19         generateVarsDeclaration (funVars fun) ++
20         generateFunctionsDeclarations cond1 cond2 functions ++
21         generateCondAssert cond1 fun functions ++
22         generateCondAssert cond2 fun functions ++
23         "(check-sat)"
```

La función 'generateZ3Str' toma un par de condiciones, la función actual y todas las funciones definidas para generar el string de entrada para Z3. El string generado está en el formato esperado por Z3[26] y consta de las siguientes declaraciones:

- Declaración de conjuntos enumerados: en primer lugar se definen todos los conjuntos definidos por enumeración en MateFun mediante la declaración 'declare-datatype' en los cuales se enumeran los elementos de cada conjunto.
- Declaración de variables de la función: En esta sección de definen los nombres de las variables ocurrentes en las condiciones de la función como constantes de Z3, mediante la directiva 'declare-const' donde se indica solamente el tipo de dato que puede ser entero o real, sin asignar ningún valor.
- Declaración de funciones: Debido a que las funciones en MateFun pueden invocar otras funciones, es necesario definir todas las funciones incluidas por el usuario.
- Declaración de las condiciones: Se incluyen las condiciones, basicamente relaciones entre variables, conjuntos y funciones.

Notar que en la línea 20 de 6.4.2.1 se incluyen las declaraciones de funciones definidas. Para ejemplificar su utilización, tomaremos el siguiente programa de entrada:

```
1 g :: R -> R
2 g(x) = 1 si x > 0
3        o 10 si x < 0
4        o 6
5
6 f :: R -> R
7 f(x) = 2 si g(x) > 0
8        o 3 si x > 0
9        o 0
```

Código 6.2: Ejemplo de programa MateFun con intersección en dominios

Como se puede observar, f no corresponde a una función en $\mathbb{R} \to \mathbb{R}$ debido a que hay puntos del dominio que pueden tomar más de un solo valor funcional. Por ejemplo $x = 1$ hace que las condiciones $g(x) > 0$ y $x > 0$ sean verdaderas.

Al querer cargar el programa 6.2 en el intérprete activando la detección de interesección de dominios y el modo detallado, observamos que se realizan los siguientes chequeos:

```
1 Evaluation on Z3 for conditions: x > 0 and x < 0:
```

```
2  (declare-const x Real)(assert (> x 0.00))(assert (< x 0.00))(check-sat)
3
4  Evaluation on Z3 for conditions: g x > 0 and x > 0:
5  (declare-const x Real)(define-fun g ((x Real)) Real (if (> x 0.00) (if (<
       x 0.00) 6.00 10.00) 1.00))(assert (> (g x) 0.00))(assert (> x 0.00))(
   check-sat)
6
7  Cargando Test2
8  {archivo: Test2 linea: 7 columna: 8}
9   Interseccion de dominios en la funcion f: interseccion entre las
   condiciones
10         g x > 0 (Ann {pos = "Test2" (line 7, column 8), typ = R})
11  y:
12         x > 0 (Ann {pos = "Test2" (line 8, column 10), typ = R})
```

En las líneas 2 y 5 se observan los strings generados por la función 'generateZ3Str' y luego enviados a Z3. Observar que luego de la línea 5 se reporta el error al usuario, detallando cuales condiciones son las conflictivas. Esta decisión es generada por el intérprete al recibir como resultado de la evaluación de Z3 que el sistema es satisfactible. Podemos verificar los resultados reportados por Z3 al pasar directamente estos strings como entrada del binario Z3:

```
1  $ z3 -in
2  (declare-const x Real)(assert (> x 0.00))(assert (< x 0.00))(check-sat)
3  unsat
4
5  $ z3 -in
6  (declare-const x Real)(define-fun g ((x Real)) Real (if (> x 0.00) (if (<
       x 0.00) 6.00 10.00) 1.00))(assert (> (g x) 0.00))(assert (> x 0.00))(
   check-sat)
7  sat
```

Sin embargo, hay casos en los que Z3 no es capaz de determinar la satisfacibilidad del sistema planteado. En estos casos, no puede determinar la satisfacibilidad con la

información recibida y devuelve 'unknown'. Tomemos por ejemplo el siguiente programa en MateFun:

```
1 f :: R X Z -> R
2 f(x,y) = 5 si (x > 1, y > 1)
3           o 3 si x * 2 * y > 2
4           o 9
```

Observamos que por ejemplo con el punto del dominio $(x, y) = (2, 2)$, las condiciones $(x > 1, y > 1)$ y $2xy > 2$ se hacen verdaderas, lo cual no corresponde con la definición de función. Sin embargo, Z3 no es capaz de determinar la satisfacibilidad de este sistema formado por ambas condiciones:

```
1 $ z3 -in
2 (declare-const x Real)(declare-const y Int)(assert (> x 1.00))(assert (>
     y 1.00))(assert (> (* (* x  2.00)  y) 2.00))(check-sat)
3 unknown
```

En estos casos, las funciones son cargadas al intérprete, sin embargo se va a realizar el chequeo de intersección de dominios en tiempo de ejecución.

6.4.2.2. Chequeo dinámico de intersección de dominios

El chequeo dinámico de intersección de dominios es ejecutado para aquellas funciones en la que el chequeo estático (realizado por Z3) no se pudo determinar (evaluación en 'Unknown').

De esta forma, cuando el usuario realice una evaluación de una función que no pudo ser determinada estáticamente, se procederá a verificar que solo una de las condiciones se hace verdadera para los valores provistos por el usuario. En el caso particular de las funciones que no están definidas por partes, el problema es trivial.

Para funciones definidas por partes, las evaluaciones en puntos del dominio resulta en lo siguiente.

Supongamos que tenemos una función f definida como en 6.3 y se quiere evaluar la función en un punto del dominio $(z_1, z_2, ..., z_n)$, es decir evaluar $f(z_1, z_2, ..., z_n)$. El intérprete itera en la definición de la función, evaluando cada una de las condiciones y

armando una lista de resultados booleanos de evaluaciones. De esta lista, se cuenta la cantidad de evaluaciones verdaderas y en caso de haber una única evaluación verdadera, se procede a evaluar la función. En caso contrario se muestra un error, ya que no se cumple la definición de función.

```
1  evalExpG :: ExpG -> NFun -> Exp -> Eval Val
2  evalExpG expg@(ExpSi exp cond gelse ann) fun expEv =
3    do
4      env <- ask
5      let loadedFunctions = loadedFuns env
6      let loadedFun = lFun loadedFunctions fun
7      let conds = funConds loadedFun
8      case isEvalUnknown loadedFun of
9        True ->
10         do
11           (VR evals) <- evalFunConds (funConds loadedFun)
12           if evals > 1
13             then (tell [FunIntersectEval (pos ann) fun expEv] >> exit)
14           else do
15             vcond <- evalCond cond
16             if vcond
17             then evalExp  exp
18             else evalExpG gelse fun expEv
19       False ->
20         do
21           vcond  <- evalCond cond
22           if vcond
23           then evalExp  exp
24           else evalExpG gelse fun expEv
25
26 evalExpG (Else exp ann) fun expEv = evalExp exp
```

Código 6.3: Evaluación dinámica de funciones

De esta forma se completa la detección de intersección de dominios en funciones por partes en MateFun. A modo de resumen:

- Se agregó un nuevo parámetro (-d) a MateFun con el cual el usuario indica si desea que se realice detección en la intersección de dominios de funciones por partes. Por defecto el chequeo se encuentra desactivado.
- Se agregó un nuevo parámetro (-v) a MateFun con el cual el usuario indica si desea activar el modo detallado. En el modo detallado se muestran los llamados realizados a Z3 para detección de intersección de dominios.
- Se agregó la detección de intersección de dominios en dos formas:
  - En tiempo de compilación: Se realiza un chequeo estático en los dominios de las funciones por partes a cargar en MateFun mediante invocación al solver SMT Z3. Las evaluaciones se realizan de a pares de condiciones por cada función por partes definida:
    - Si Z3 evalúa un par de condiciones como satisfacible, entonces hay intersección de dominios y se muestra un error.
    - Si Z3 evalúa un par de condiciones como insatisfacible, se continúa la evaluación con el siguiente par de condiciones
    - Si Z3 no puede determinar la satisfacibilidad de un par de condiciones, se marca la función para posterior chequeo en tiempo de ejecución al realizar una evaluación.
  - En tiempo de ejecución: Se realiza un chequeo dinámico en la evaluación de aquellas funciones por partes en las cuales Z3 no pudo determinar si había o no intersección en los dominios:
    - Si al evaluar la función en un punto hay más de una condición que se hace verdadera, entonces se muestra un error.
    - Si al evaluar la función en un punto hay solo una condición que se hace verdadera, entonces se devuelve el valor funcional

Capítulo 7

Conclusiones

Este trabajo aportó muchas mejoras al intérprete de MateFun. Estas mejoras han sido en distintas áreas: en funcionalidades, en documentación y en su proceso de contribución.

A nivel de código fuente, se agregaron nuevas funcionalidades al compilador e intérprete de MateFun. En primer lugar se brindó soporte a internacionalización. Se agregaron dos idiomas con los cuales el usuario puede interactuar. En segundo lugar, se agregó una funcionalidad para atacar una limitación del sistema, el chequeo de intersección de dominios en funciones definidas por partes.

El usuario es capaz de decidir si realizar o no un chequeo de intersecciones de dominios mediante un nuevo flag pasado como parámetro al binario. En caso de activar el chequeo, se va a utilizar un solver SMT para verificar la satisfacibilidad de cada par de condiciones definidas y determinar si existe o no intersección de dominios. Para los casos en que el solver no puede determinarlo estáticamente, se realiza la evaluación en tiempo de ejecución de las condiciones de la función, buscando determinar si se cumple con la restricción de que solamente una condición es verdadera al evaluar la función en un punto determinado.

Además, se creó documentación para un mejor entendimiento del sistema. Esta documentación era casi nula al inicio de este trabajo. Actualmente se encuentra disponible en la Wiki[1] del repositorio.

[1] `https://gitlab.fing.edu.uy/matefun/MateFun/wikis/home`

Se definió un proceso de contribución adecuado al repositorio para mejorar las contribuciones futuras. Se introdujo también integración continua al repositorio del proyecto.

Para finalizar, se plantean posibles mejoras y trabajos a futuro.

Sobre integración continua y el proceso de contribución se plantean dos temas a considerar a futuro. Dado que el repositorio de MateFun se encuentra dentro del dominio de Facultad de Ingeniería se podría analizar la capacidad de cómputo que se quiera dedicar a las tareas automatizadas definidas en integración continua. Esto brindaría mejoras en los tiempos de ejecución de pipelines al dedicar mayor capacidad de cómputo y poder paralelizar tareas. Por otra parte, dado que solamente los usuarios en el directorio de Facultad con un usuario en GitLab son quienes pueden realizar contribuciones (ya sean en reporte de errores o contribuciones al código fuente), sería deseable poder extender el alcance a más usuarios. Si el alcance fuera público, se permitiría que cualquier persona pueda realizar contribuciones (con los permisos restringidos según el nivel otorgado) obteniendo retroalimentación muy valiosa por parte de los usuarios y posibles mejoras o nuevas implementaciones por parte de otros desarrolladores.

Con respecto a la internacionalización de MateFun, sería interesante establecer un protocolo sobre el mantenimiento de las traducciones existentes para los idiomas soportados. Dada la separación de las traducciones del código fuente y la naturaleza de GetText, es necesario evaluar si el mantenimiento de estas traducciones puede ser llevado a cabo por una nueva tarea en el pipeline de integración continua que explore los archivos de traducción y reporte errores en caso de traducciones faltantes. Se podría analizar también si es necesario contar con traductores a los distintos idiomas soportados para asegurar el mantenimiento de cada idioma y además incluir las traducciones a nuevos idiomas.

Por último se plantea optimizar la interacción con el solver Z3 para la detección de intersección de dominios en funciones por partes. Actualmente, se realiza un llamado por cada par de condiciones definidas en las funciones a ser cargadas en un archivo. En un archivo a cargar con un número bastante alto de funciones definidas por partes, el número de llamados a Z3 puede ser muy alto. Sería deseable optimizar los llamados y reducirlos mediante la utilización del stack interno de Z3. También sería deseable poder interactuar con la interfaz definida para Z3 en vez de generar el string en formato SMT-LIB, ya que un cambio en la sintaxis puede generar regresiones. Además, dado que el soporte a conjuntos definidos por enumeración solo soporta la comparación por igualdad

sería bueno poder extender el soporte y realizar comparaciones de orden sobre conjuntos enumerados.

Bibliografía

[1] Alejandra Carboni, Victor Koleszar, Gonzalo Tejera, Marcos Viera, Javier Wagner. *MateFun: Functional Programming and Math with adolescents*. CLEI - LACLO 2018.

[2] Sylvia da Rosa, Gustavo Cirigliano. *Matemática y programación*. Proceedings of VII Congreso Iberoamericano de Educación Superior en Computación, PUCE (Pontificia Universidad Católica del Ecuador). 1998.

[3] Ian Sommerville. *Ingeniería de Software, Novena Edición*. Addison-Wesley, 2011.

[4] Martin Fowler. *Continuos Integration (original version)*. Disponible en: `https://www.martinfowler.com/articles/originalContinuousIntegration.html`. Última consulta: 20/08/2019.

[5] Microsoft Azure. *¿Qué es SaaS*. Disponible en: `https://azure.microsoft.com/es-es/overview/what-is-saas/`. Última consulta: 20/08/2019.

[6] *Tags for Identifying Languages*. Disponible en: `https://tools.ietf.org/html/bcp47`. Última consulta: 20/08/2019.

[7] GNU Project - Free Software Foundation (FSF). *GNU gettext utilities: Concepts*. `https://www.gnu.org/software/gettext/manual/html_node/Concepts.html#Concepts`. [Última consulta: 20/10/2019].

[8] *hgettext: Bindings to libintl.h (gettext, bindtextdomain)*. Disponible en: `http://hackage.haskell.org/package/hgettext`. Última consulta: 20/08/2019.

[9] GNU Project - Free Software Foundation (FSF). *Overview of GNU gettext*. `https://www.gnu.org/software/gettext/manual/html_node/Overview.html#Overview`. [Última consulta: 29/10/2019].

[10] *i18n*. Disponible en: `https://github.com/mcfilib/i18n`. Última consulta: 20/08/2019.

[11] *shakespeare: A toolkit for making compile-time interpolated templates*. Disponible en: `http://hackage.haskell.org/package/shakespeare`. Última consulta: 20/08/2019.

[12] Stephen Cook. *The Complexity of Theorem-Proving Procedures*. 1971.

[13] *The Z3 Theorem Prover*. Disponible en: `https://github.com/Z3Prover/z3`. Última consulta: 20/08/2019.

[14] Nikolaj Bjørner Leonardo de Moura. «Z3: An Efficient SMT Solver». En: *C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963* (2008).

[15] Aina Niemetz, Mathias Preiner y Armin Biere. «Boolector 2.0 system description». En: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), págs. 53-58.

[16] *hgettext: Bindings for the Z3 Theorem Prover*. Disponible en: `http://hackage.haskell.org/package/z3`. Última consulta: 20/08/2019.

[17] GitLab. *matefun/MateFun - GitLab*. `https://gitlab.fing.edu.uy/matefun/MateFun`. [Última consulta: 09/06/2019].

[18] GitLab. *GitLab Runner | GitLab*. `https://docs.gitlab.com/runner/`. [Última consulta: 09/06/2019].

[19] Open Source Guides. *Cómo Contribuir con el Código Abierto*. `https://opensource.guide/es/how-to-contribute/#cómo-enviar-una-contribución`. [Última consulta: 20/10/2019].

[20] Docker. *matefundocker/matefun-env ' Docker Hub*. `https://hub.docker.com/r/matefundocker/matefun-env`. [Última consulta: 09/06/2019].

[21] Ubuntu. *Releases - Ubuntu Wiki*. `https://wiki.ubuntu.com/Releases`. [Última consulta: 09/06/2019].

[22] Apache Tomcat. *Contributing to Apache Tomcat (ACS)*. `https://github.com/apache/tomcat/blob/master/CONTRIBUTING.md`. [Última consulta: 20/10/2019].

[23] Wikipedia. *ISO 639-1*. `https://en.wikipedia.org/wiki/ISO_639-1`. [Última consulta: 20/10/2019].

[24] HaskellWiki. *Internationalization of Haskell programs using gettext*. `https://wiki.haskell.org/Internationalization_of_Haskell_programs_using_gettext`. [Última consulta: 23/10/2019].

[25] David R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial*. `http://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf`. [Última consulta: 09/06/2019].

[26] *Z3 - Guide*. Disponible en: `https://rise4fun.com/z3/tutorialcontent/guide`. Última consulta: 20/08/2019.