UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SISTEMA DE VALIDACIÓN PARA EL DESARROLLO INCREMENTAL DE UN INTÉRPRETE DE R EN COQ

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

TOMÁS IGNACIO DÍAZ TRONCOSO

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
JOCELYN SIMMONDS WAGEMANN
FEDERICO OLMEDO BERÓN

SANTIAGO DE CHILE
2018

Resumen

El lenguaje de programación R es muy popular para desarrollar programas estadísticos y de análisis de datos, gracias a sus numerosas librerías y amplia comunidad, su sintaxis concisa y expresiva, así como soporte para su uso interactivo. Aún así, su semántica es bastante compleja y contiene numerosos casos de borde, y no está formalmente especificada, lo que hace difícil confiar plenamente en los programas desarrollados en R.

Esto tiene como consecuencia que, en el año 2017, se inicie el proyecto CoqR, que busca formalizar la semántica de R, definiendo para ello una semántica natural en la forma de un intérprete implementado en el asistente de pruebas Coq.

Como primera fuente de confianza, para validar la formalización, se utiliza una codificación monádica, que permite establecer una correspondencia visual directa entre el código en Coq y el intérprete de referencia implementado en C, GNU R, de tal forma que una o dos líneas de CoqR equivalen a una o dos de GNU R.

El sistema desarrollado en esta memoria sirve como infraestructura para ejecutar bancos de pruebas y poder comparar, de manera sistemática, el comportamiento de CoqR y GNU R, proporcionando una segunda fuente de confianza para validar la formalización. Además, proporciona herramientas para apoyar el proceso iterativo de desarrollo, permitiendo identificar y detectar tanto las funcionalidades más relevantes a implementar, como los errores que puedan surgir durante esta fase. De esta manera, el sistema se integra directamente al proceso de desarrollo y a la validación de CoqR, logrando que, en su versión actual, cubra cerca del 30 % de los casos de pruebas, tanto de GNU R como de otro proyecto relevante, FastR.

Para mis padres.

Agradecimientos

A mis padres, por su apoyo incondicional y gracias a quienes me encuentro aquí en este momento.

A la gente del “Culto”, en especial a Ferrada, Pancho Madrid y Tito, por sus diversas ayudas. También a Felipe Rodríguez por discutir conmigo temas de diseño o ayudarme a aclarar ideas.

A la Marle, Bella, Fraky y Casel, los perros más hermosos del mundo.

A Paula, por acompañarme en todo momento, alegrar mis días y ayudarme siempre que lo necesitara.

Tabla de contenido

Índice de ilustraciones

Índice de tablas

Capítulo 1

Introducción

El lenguaje de programación R [13, 7, 10] se ha vuelto muy popular en los últimos años, dado su enfoque al manejo de datos y estadísticas. Sus usuarios se sitúan en áreas desde la biología, pasando por matemáticas, hasta ciencias sociales. Parte de su encanto radica en que su sintaxis es simple y expresiva. Sin embargo, esta expresividad trae consigo ciertas sutilezas con respecto a su semántica. Lamentablemente, R es uno de tantos lenguajes que no posee una especificación formal, rigurosa, de su semántica, si no que se sustenta en especificaciones en prosa, actualmente borradores [14], y en cómo su interprete de referencia, GNU R [10], se comporta. En consecuencia, no es posible llevar a cabo estudios más formales sobre el lenguaje y su comportamiento.

A partir de esto, desde el Centro de Modelamiento Matemático (CMM) y el Departamento de Ciencias de la Computación (DCC), de la Universidad de Chile, se inicia un proyecto a mediados del año 2017 que busca realizar una formalización de R. Este proyecto, que recibe el nombre de *CoqR*, es liderado por Martin Bodin, quien trabajó anteriormente en el proyecto JSCert [3, 2]; un proyecto de formalización del lenguaje de programación Javascript, donde también se cuenta únicamente con una especificación en prosa y un intérprete de referencia. CoqR forma parte de su trabajo de postdoctorado, guiado por Éric Tanter.

La formalización se decide llevar a cabo en Coq [5], un asistente de pruebas, siguiendo metodologías similares a las del proyecto de JSCert. En particular, se decide realizar la traducción basándose en el intérprete de referencia, GNU R, y no a su especificación en prosa, por estar muy incompleta. Para validar la formalización se plantean dos fuentes de confianza: una dada por una correspondencia visual entre el código implementado y el de referencia, y otra dada por la ejecución de baterías de pruebas. Dado que la semántica es la base sobre la cual construir lo demás, esta validación busca entregar la confianza necesaria para establecer la equivalencia con respecto al lenguaje objetivo.

Al momento de comenzar este trabajo de memoria, CoqR ya cuenta con una versión inicial, en la forma de un intérprete, capaz de ejecutar un conjunto básico de funcionalidades de R [1]. El conjunto de funciones es bastante limitado y no cuenta con una validación dada por la segunda fuente de confianza, mencionada en el párrafo anterior.

1.1. Motivación

Cuando se habla de verificar formalmente un lenguaje se refiere, en simples palabras, a comprobar que éste se comporta como uno espera que lo haga. Esto significa demostrar la existencia o falta de ciertas propiedades del lenguaje; de demostrar o refutar su corrección. Para llevarlo a cabo, es necesario hacerlo con estricto rigor, basándose en reglas de inferencia, axiomas, lógica, etc., es decir, no son simples demostraciones antojadizas. Sin embargo, ya sea que se quiera demostrar propiedades del lenguaje o bien propiedades de programas particulares, es necesario tener definida su semántica. Sólo una vez que ésta haya sido definida es que se puede empezar a abordar la otra problemática.

En este contexto, CoqR busca traducir la semántica de R a Coq, pero como se mencionó anteriormente, no existe una especificación formal de ésta, si no que especificaciones informales en prosa o a través de su intérprete. La dificultad que esto plantea es decidir qué tan confiable es la traducción que se lleve a cabo; cómo saber que efectivamente la semántica descrita en Coq es equivalente a la de R y, por lo tanto, qué tan confiables serán las demostraciones posteriores que se puedan realizar en base a ésta.

Es, por lo tanto, imperativo comprobar de alguna forma que tanto GNU R y CoqR sean equivalentes. Mostrar esta equivalencia no es directo, es necesario desarrollar herramientas y metodologías que se integren al proceso de desarrollo actual y que permitan decir, con mayor grado de certeza, que se tiene lo que se buscaba.

Por una parte, se necesita herramientas que lleven a cabo la comparación entre los intérpretes y, por otro, herramientas que permitan identificar claramente las piezas que faltan, los errores presentes o el nivel de similitud, así como mantener registro de los resultados obtenidos, entre otras cosas. Estas mismas herramientas pueden permitir, a su vez, clarificar las distintas etapas de desarrollo, así como permitir elaborar metodologías y estrategias para el avance del proyecto.

1.2. Objetivos

El objetivo general del trabajo es validar la equivalencia entre CoqR y GNU R, a través de la ejecución de bancos de pruebas. Para este fin se plantea llevar a cabo la implementación de herramientas y metodologías que se integren al proceso actual de desarrollo, con los cuales poder ejecutar las pruebas, comparar los resultados de ambos intérpretes y entregar reportes de éstos. Este objetivo se plantea en pos de obtener una especificación del lenguaje, con la cual permitir que se realicen certificaciones del comportamiento de GNU R y programas en R.

A pesar de que se propone en particular para el caso de GNU R, las herramientas y metodologías se plantean con una perspectiva más amplia, a ser aplicable a otros lenguajes sin especificación formal y con intérpretes de referencia.

Para llevar a cabo este objetivo se plantean los siguientes objetivos específicos:

1. Desarrollo de un sistema que ejecute las pruebas y compare los resultados.
2. Desarrollo de un sistema de almacenamiento de resultados.
3. Desarrollo de visualizaciones de diferentes métricas sobre los resultados.
4. Integración con el proceso de desarrollo de CoqR.
5. Implementación de funcionalidades de GNU R en CoqR.

1.3. Estructura del documento

El capítulo 2 introduce los conceptos y herramientas involucradas en este trabajo de memoria, avanzando progresivamente desde el lenguaje de programación R y Coq, para luego repasar aspectos importantes de las formalizaciones mecanizadas y finalizando con una descripción de CoqR, el proyecto de formalización mecanizada de R, sobre el cual se enmarca el presente trabajo. En seguida, en el capítulo 3 se explica el sistema desarrollado, detallando el funcionamiento e interacción de sus distintos módulos, para luego continuar en el capítulo 4 con la validación de este sistema, repasando el funcionamiento y la colaboración con CoqR, permitiendo guiar el proceso de desarrollo de este último y presentando los resultados finales obtenidos. Por último, en el capítulo 5 se concluye sobre el trabajo realizado, respecto a los objetivos planteados, además de presentar líneas de trabajo a futuro.

Capítulo 2

Marco teórico

En este capítulo se introducen los conceptos y herramientas involucrados en el trabajo de memoria realizado. Primero se introduce el lenguaje de programación R, sobre el cual se lleva a cabo una formalización mecanizada de su semántica, y el asistente de pruebas (*proof assistant*) Coq, utilizado para poder realizarla. En seguida, se introduce el concepto de formalización mecanizada de lenguajes, donde se abarcan conceptos básicos sobre especificación de lenguajes, así como ejemplos reales de formalizaciones y problemáticas que surgen durante este tipo de proyectos, además de formas de abordarlas y cómo validar la formalización llevada a cabo. Esta sección tiene un enfoque orientado a mejorar la comprensión de la especificación que se realiza para R. Finalmente, el último segmento trata sobre CoqR, correspondiente al proyecto de formalización mecanizada de R y sobre el cual se enmarca, en particular, el trabajo realizado en esta memoria.

2.1. Lenguaje de programación R

R es un lenguaje y ambiente de programación, diseñado inicialmente en 1993 por Ross Ihaka y Robert Gentleman, orientado a la computación y análisis estadístico. Es similar al lenguaje de programación S[1] desarrollado por John Chambers y colegas, pudiendo llegar a considerarse como una implementación de este (varias funcionalidades del lenguaje son escritas directamente en S). Sin embargo, existen ciertas diferencias entre ambos que no los hacen del todo compatibles. El lenguaje está disponible gratuitamente bajo los términos de la licencia *GNU*[2] de la *Free Software Foundation*[3].

R está fuertemente apoyado y guiado por las comunidades que lo utilizan, lo que se refleja en sus diversas formas y estilos de programación, en las distintas áreas que abarca, desde biología a ciencias sociales [12], además de contar con más de 12.000 librerías [11]. Parte de su atractivo se debe a todas las herramientas que posee, permitiendo un fácil manejo de

[1] https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
[2] https://www.gnu.org/licenses/gpl-3.0.en.html
[3] https://www.gnu.org/

```r
f <- function (abc, ab, de) { c (abc, ab, de) }
f(1, 2, 3)           # Por posición
f(de=3, abc=1, ab=2) # Por nombre
f(1, d=3, 2)         # Mezclado
f(3, a=1, ab=2)      # 'a' se asocia a 'abc'
f(a=3, 1, 2)         # error: múltiples casos
```

Figura 2.1: Distintas formas de pasar argumentos a funciones en R.

```r
"(" <- function (x) x * x
(3) # Devuelve 9
```

(a) Redefiniendo el paréntesis

```r
if (TRUE) x <- 1 else return ()
"if" (TRUE, "<-" (x, 1), "return" ())
```

(b) Distintos estilos con mismo significado

Figura 2.2: Más sutilezas del lenguaje de programación R.

datos, cálculos vectoriales y matriciales, generación de gráficas, entre otros, pero además, a la simpleza y expresividad al momento de escribir código. Sin embargo, el lenguaje posee numerosos casos de borde, con pequeñas sutilezas en su semántica [4], lo que dificulta el razonamiento sobre el lenguaje en sí y los diferentes programas que se puedan desarrollar.

Estas sutilezas se pueden ver reflejadas en los ejemplos descritos a continuación. En primera instancia, en la figura 2.1, se muestran dos maneras de pasar argumentos a funciones: por posición o nombre. Aunque esto es algo habitual en un lenguaje de programación, la forma en que R procesa los argumentos no lo es, en particular, porque permite parear nombres usando prefijos. Por ejemplo, la primera llamada es una por posición, mientras la segunda es por nombre. Por otro lado, la tercera es una mezcla, donde `d` se asocia al argumento con nombre `de` por ser el único que calza. El cuarto caso es un poco más complicado, ya que podrían haber múltiples casos que calzan con los prefijos. Lo que se hace en esta situación es intentar calzar exactamente los nombres dados con los argumentos y los demás simplemente por prefijo. Así, en este caso `ab` calza justo con el segundo argumento de la función, mientras que `a` calza parcialmente con el argumento `abc`. Finalmente, en el quinto caso, hay más de un argumento con el mismo prefijo y no son exactos, por lo que se produce un error.

Otros ejemplos más simples, pero aún así dentro de las sutilezas de lo que es posible hacer en R, son los casos de las figuras 2.2a y 2.2b. La primera muestra que es posible redefinir el significado de un paréntesis, mientras que el segundo muestra que dos expresiones en estilos distintos, uno con sintaxis concreta y otro con cadenas de caracteres, son expresiones válidas y equivalentes dentro del lenguaje.

Para finalizar esta sección sobre R, vale mencionar las alternativas disponibles para entender la semántica del lenguaje y desambiguar los casos de borde que se puedan encontrar.

Por un lado, y como se mencionó anteriormente, R posee un documento donde se describe la semántica del lenguaje [14] y que los usuarios pueden consultar. Sin embargo, este documento se encuentra escrito en prosa, en inglés, y aún está en una fase de borrador. Esto significa que hay aspectos incompletos y que pueden ser ambiguos, dados por el lenguaje natural que utiliza.

```
1 Inductive nat : Type :=
2   | O : nat
3   | S : nat -> nat.
```

Figura 2.3: Definición inductiva de los números naturales en Coq.

Por otro lado, R también posee un intérprete de referencia, denominado GNU R, el cual puede servir como especificación para entender la semántica del lenguaje y resolver los casos de borde. Un desarrollador puede ejecutar el intérprete sobre distintas expresiones y así resolver dudas sobre el comportamiento del lenguaje. Además, el código de GNU R está escrito en el lenguaje de programación C y es abierto a la comunidad, por lo que es posible inspeccionarlo para entender más aún cómo se resuelven las distintas expresiones del lenguaje.

Estas dos especificaciones están disponibles para la comunidad, pudiendo elegir una u otra dependiendo de la situación. Aún así, es importante recalcar que el documento en prosa se encuentra aún en un estado de borrador y que hay inconsistencias entre las semánticas descritas por el documento y por el intérprete. Por ejemplo, la expresión `if ("TRUE") 42` en R, debiera fallar según el documento, mientras que GNU R la ejecuta correctamente y entrega el valor `42`.

2.2. Coq

Coq es un sistema computacional diseñado para asistir al desarrollo de pruebas formales, iniciado en 1984 por Thierry Coquand y Gérard Huet, como parte de la implementación del *Cálculo de Construcciones* [5]. Coq provee un ambiente para el desarrollo de pruebas formales, desde un lenguaje de programación (llamado *Gallina*) a herramientas para el desarrollo interactivo de demostraciones. Es posible, además, extraer el código escrito en Coq a otros lenguajes de programación como OCaml, Haskell y Scheme.

Por un lado, *Gallina* es un lenguaje que combina aspectos de lógica de orden superior junto a un lenguaje funcional fuertemente tipado, de tal manera que permite escribir definiciones matemáticas, algoritmos ejecutables y teoremas, entre otras cosas. Estas características permiten llevar a cabo formalizaciones y demostraciones que son verificadas por la máquina, de manera automática o semi-automática, durante el proceso de desarrollo.

Por otro lado, el sistema interactivo permite ejecutar parcialmente demostraciones, proveyendo visualizaciones de los pasos y metas restantes, así como información del estado de la demostración. Por último, puede dar mensajes de error en caso de que no sea válido realizar ciertos pasos lógicos, entre otras cosas.

A modo de ejemplo, es posible definir estructuras inductivas, como se ve en la figura 2.3, donde se definen los números naturales. Es posible definir funciones y demostrar teoremas sobre estas estructuras, como se ve en la figura 2.4, donde se define la operación de suma en el lado izquierdo, mientras que en el derecho se realiza la demostración de que un número sumado a cero, da efectivamente el mismo valor. Se puede ver en la línea 4 de la figura 2.4b que

```
1 Fixpoint plus (n m : nat) : nat :=
2   match n with
3   | O => m
4   | S n' => S (plus n' m)
5 end.
```

(a) Definición de función

```
1 Theorem plus_n_O :
2   forall n : nat, n = n + 0.
3 Proof.
4   intros n. induction n as [| n' IHn'].
5   - reflexivity. (* n = 0 *)
6   - simpl. rewrite <- IHn'.
    ↪  reflexivity. (* n = S n' *)
7 Qed.
```

(b) Definición de teorema

Figura 2.4: Definiciones de funciones y teoremas en Coq.

Figura 2.5: Interfaz provista por Emacs para trabajar interactivamente con demostraciones en Coq.

es posible trabajar la demostración de manera inductiva sobre la estructura de los naturales; las líneas 5 y 6 corresponden a la demostración para cada constructor de la estructura. Note el uso de la hipótesis de inducción en la línea 6.

Por otro lado, se puede ver en la figura 2.5 la interfaz durante el proceso de demostración de un teorema. En este ejemplo en particular, se está probando que el largo de una lista es igual al largo de la misma lista invertida[4]. En la esquina superior derecha puede verse información sobre las hipótesis, hipótesis inductivas, próximo paso en la demostración, etc. Además, la zona coloreada a mano izquierda indica secciones ya evaluadas y verificadas, como lo es el caso base, de lista vacía, para la demostración inductiva.

2.3. Semántica formal mecanizada de lenguajes

En esta sección se revisa el concepto de mecanizar la semántica de un lenguaje de programación, dado que este trabajo de memoria se enmarca dentro de un proceso de formalización, en particular del lenguaje R.

[4]Ejemplo obtenido del volumen 1 de *Software Foundations* (https://softwarefoundations.cis.upenn.edu/lf-current/Lists.html).

Primero que todo, una especificación formal de la semántica de un lenguaje se refiere a una descripción matemática del comportamiento de éste. Luego, una formalización "mecanizada" hace referencia a que la formalización se lleve a cabo utilizando un sistema computacional, tal como Coq, Agda[5] o Isabelle[6], entre otros. La mecanización viene del hecho de que estos sistemas permiten automatizar distintos aspectos del proceso de desarrollo, como lo es la verificación de tipos, o proveyendo sistemas interactivos para ejecutar demostraciones, entre otros factores.

El objetivo principal de llevar a cabo estas formalizaciones mecanizadas es que, una vez realizada, es posible verificar formalmente comportamientos del lenguaje y de programas. En palabras simples, certificar que el lenguaje o un programa se comporta como uno espera que lo haga. Por ejemplo, que ciertas operaciones de punteros sean correctas y no accedan a espacios de memoria no asignados, o bien, que el lenguaje o ciertas expresiones sean deterministas.

Estas demostraciones pueden realizarse perfectamente con lápiz y papel, pero pueden ser tediosas, además de que es posible introducir errores durante el proceso mismo de demostración. Es aquí donde entran en juego las herramientas provistas por estos sistemas computacionales; a través del sistema de tipos es posible verificar que los pasos llevados a cabo sean correctos y que no se han introducido errores, o bien, con un sistema interactivo, es posible analizar los estados paso a paso de las demostraciones, entre otras cosas. Una relación fundamental, que permite a estas herramientas realizar esto, es la *correspondencia de Curry-Howard*, la cual exhibe una relación directa entre sistemas formales de lógica y programas computacionales.

Algunos casos ejemplares del uso de estas herramientas en la formalización y verificación de la semántica de lenguajes incluye: JSCert, una formalización de Javascript hecha en Coq, CompCert[7], un compilador formalmente verificado en Coq para una gran porción de C99, sel4[8], un microkernel que utiliza una formalización de C99, en Isabelle, o Standard ML, completamente formalizado y con una especificación mecanizada hecha en Twelf[9].

Llevar a cabo estas formalizaciones mecanizadas no está exento de problemas. En lo que resta de esta sección se abordarán algunas problemáticas que se presentan al llevar a cabo una labor de este tipo, así como formas de abordarlas.

2.3.1. Especificación de lenguajes

La especificación de un lenguaje describe los programas que son válidos y cómo estos se ejecutan. Hay distintas maneras de especificar un lenguaje: a través de un intérprete, mediante un documento en prosa (como lo es el caso de C) o con una especificación formal, como se describió anteriormente. Estas distintas formas de especificar el comportamiento del lenguaje no son excluyentes, por ejemplo, Javascript tiene un intérprete de referencia, pero

[5]https://github.com/agda/agda
[6]https://isabelle.in.tum.de/
[7]http://compcert.inria.fr/
[8]https://sel4.systems/
[9]http://twelf.org

además una especificación en prosa, dada por ECMAScript[10]. Esto abre la posibilidad a que se consideren distintas especificaciones, según el propósito que se busque; un desarrollador puede estar más interesado en cómo se comporta el intérprete, siendo éste más tangible.

Cabe mencionar que son pocos los lenguajes que tiene una semántica formalmente especificada, lo que introduce un nivel de dificultad adicional al intentar mecanizarla, ya que, por un lado, un documento en prosa no es completamente riguroso y puede tener ambigüedades dadas por el lenguaje natural que se use. Mientras que una especificación dada por un intérprete de referencia puede llegar a ser de muy bajo nivel y, en consecuencia, muy compleja.

Por otro lado, la posibilidad de múltiples especificaciones introduce un problema al momento de certificar que el lenguaje se comporta correctamente. El problema es el de definir con respecto a qué es correcto el comportamiento; correcto con respecto al intérprete de referencia, al documento de especificación o a su formalización matemática. Parte de la importancia de este aspecto es qué tan válido y aceptado es para las distintas comunidades que utilicen el lenguaje.

A modo de ejemplo se puede considerar el caso del proyecto JSCert [3, 2], que busca formalizar la semántica de Javascript. La especificación de este lenguaje viene dada por dos fuentes: una especificación en inglés y un intérprete de referencia. La estrategia que se adopta en JSCert para poder formalizar Javascript y validar que es correcto, es la de aprovechar estas dos especificaciones. Por un lado, se intenta seguir el formato y especificación dado por ECMAScript, pero además, se desarrolla un intérprete en Coq (bajo el nombre de *JSRef*), capaz de ejecutar código en Javascript. Este intérprete adhiere a la semántica especificada por JSCert y, por lo tanto, a la de ECMAScript. Luego, basta con comparar *JSRef* con el intérprete de Javascript para comprobar que se tenga el comportamiento esperado.

2.3.2. Validez de las formalizaciones

Dado que la formalización de la semántica de un lenguaje es el punto de partida para trabajo posterior, es necesario poder confiar que sea correcta. El grado de confianza que se tenga es finalmente lo que le da la validez necesaria para llevar a cabo certificación sobre el comportamiento del lenguaje y de programas escritos en este.

Recordando lo mencionado en la sección anterior, la especificación del lenguaje puede ser dada de distintas maneras, con distintos usuarios y comunidades adoptando distintas especificaciones. Por lo tanto, la confianza que se obtenga de la formalización puede apuntar a distintos grupos.

Si se considera, nuevamente, el caso de JSCert descrito en la sección anterior, la validez de esta formalización viene dada por dos fuentes: en primer lugar, una correspondencia visual entre el código en Coq y la especificación dada por ECMAScript, y en segundo, la ejecución de baterías de pruebas para comparar el comportamiento de ambos intérpretes. La correspondencia visual se refiere netamente a poder leer ambos textos y verificar que estos

[10]https://www.ecma-international.org/publications/standards/Ecma-262.htm

sean similares, mientras que la ejecución de pruebas entrega resultados más concretos con los cuales comparar a los intérpretes. En este caso, diferencias de resultados podrían indicar que efectivamente Javascript no sigue la semántica especificada por ECMAScript. A través de estos dos mecanismos se espera un grado de confianza mayor, en todo aspecto.

2.3.3. Lenguajes imperativos con efectos

Un problema que surge cuando se intenta formalizar un lenguaje imperativo con efectos, tales como Java y C, o en el caso de R, a través de su intérprete de referencia escrito en C, es que los lenguajes de los asistentes de pruebas, como *Gallina* en Coq, son lenguajes funcionales puros, por lo que es necesario llevar a cabo abstracciones sobre ciertos elementos del lenguaje, como manejo de memoria, cambios de estado, manejo de punteros, etc. Una manera de modelar efectos en un lenguaje puro es a través de mónadas [16, 17].

Mónadas

El patrón monádico es un patrón de diseño para tipos, útil para modelar efectos en lenguajes funcionales puros, donde una mónada corresponde un tipo en particular que sigue este patrón. De manera simplificada, es posible ver una mónada como un tipo genérico donde, dado un elemento de un tipo particular, ésta aumenta el poder representativo del elemento, aunque preservando sus propiedades iniciales.

A modo ilustrativo, la mónada *Maybe* sirve para modelar el efecto de que una expresión falle sin retornar un valor, como puede ocurrir cuando se lanza una excepción en un lenguaje como C. Para lograr esto, la mónada toma un elemento de un tipo dado y extiende su poder representativo, de tal manera que ahora el elemento monádico resultante representa, o el mismo elemento inicial, o un elemento inexistente. Así, si en cualquier otro punto del cómputo, éste elemento monádico toma el valor inexistente, se puede omitir la ejecución completamente y simplemente retornar un valor vacío, efectivamente representando el efecto de que falló el programa.

Es importante notar que el patrón monádico es más que tipos genéricos, incluyendo un conjunto de reglas adicionales. Una de éstas es la existencia de dos funciones auxiliares:

- Constructor : Una función para crear elementos del tipo monádico, a partir de un elemento de otro tipo. Tradicionalmente se le llama *unit*.
- Operación de unión : Una función que toma dos argumentos, un valor monádico y otra función, y devuelve otro valor monádico, que es el resultado de aplicar el segundo argumento al valor subyacente del primer argumento. Si consideramos la mónada *Maybe*, la firma de esta función puede escribirse como: `Maybe[A] ->(A ->Maybe[B]) ->Maybe[B]`, donde la mónada está parametrizada por los tipos `A` y `B`. Básicamente permite desenvolver el valor contenido en la mónada, aplicar la función sobre este elemento y volver a envolverlo en una mónada. Esta función es la que describe de alguna forma el tipo de mónada que se quiere definir, ya que establece cómo desenvolver valores y

```
1 Inductive Maybe (A : Type) : Type :=
2   | Some : A -> Maybe A
3   | Nothing : Maybe A.
```

(a) Definición de la mónada Maybe

```
1 Definition unit (A : Type) (val : A) : Maybe A :=
2   Some val.
```

(b) Definición de la función *unit*

```
1 Definition bind (A B : Type) (val : Maybe A) (function : A -> Maybe B) : Maybe B
  ↪  :=
2   match val with
3   | Some v => function (v)
4   | Nothing _ => Nothing B
5   end.
```

(c) Definición de la función *bind*

Figura 2.6: Definición de la mónada Maybe en Coq, junto a sus funciones *unit* y *bind* asociadas.

combinarlos para crear otros elementos monádicos. Tradicionalmente recibe el nombre de *bind*.

En la figura 2.6 se puede ver la definición de la mónada Maybe y sus funciones *unit* y *bind*, escritas en Coq. En la figura 2.6a se pueden ver los constructores para esta mónada, `Some` y `Nothing`, que permiten extender un valor de tipo `A`; ya sea manteniendo su valor a través del constructor `Some` o describiéndolo como un valor inexistente, a través del constructor `Nothing`, que no recibe parámetros. Por otro lado, la figura 2.6b muestra la función *unit* que, dado un elemento de tipo `A`, retorna un valor monádico de tipo `Maybe A`. Por último, en la figura 2.6c se ve la definición de *bind*, donde se aplica la función entregada como parámetro al valor encapsulado dentro del elemento monádico `val`. En la línea 2 se puede ver que se realiza *pattern-matching* sobre el valor monádico entregado, para ver si el valor existe o no; si existe se aplica la función directamente (línea 3) y, si no, se propaga el valor inexistente (línea 4). Esto último permite simular que se propaga un resultado erróneo (excepción) que alguna función genere.

Más leyes monádicas se definen sobre estas funciones, como por ejemplo, que aplicar una función (a través de *bind*) sobre un elemento monádico sea equivalente a aplicar la función directamente sobre el valor envuelto por la mónada, entre otras cosas.

Con la apropiada definición de mónadas, es posible replicar efectos de lenguajes imperativos en uno funcional. Por ejemplo, operaciones de entrada y salida, modificaciones al estado de ejecución, lectura a un ambiente global compartido, no determinismo o excepciones, como fue mencionado. Además, es posible definir mónadas como composición de otras (por ejemplo, a través de transformadores monádicos), de tal manera que se pueda modelar un conjunto de efectos y replicar comportamientos imperativos en un lenguaje funcional puro.

2.4. CoqR

En esta sección se describe CoqR, una formalización mecanizada de GNU R, recordando que el trabajo de esta memoria se enmarca dentro del proceso de desarrollo de esta formalización, tanto en su validación y como apoyo a la implementación.

En primera instancia, se describen aspectos generales de CoqR, sobre las dificultades que surgen por trabajar en Coq y C, además de algunas decisiones llevadas a cabo al respecto. En seguida se continúa con la descripción de una estructura básica necesaria para capturar los efectos principales de R, como se mencionó en 2.3.3, así como para obtener una correspondencia visual entre CoqR y GNU R. Esta estructura recibe el nombre de *mónada de resultado*.

Luego, se retoma el aspecto de la correspondencia visual, con la cual se busca validar que CoqR sea equivalente a GNU R, y finalmente se continúa con una sección sobre el proceso mismo de desarrollo de CoqR, al momento de iniciar este trabajo de memoria, y cómo este está intimamente ligado a la estructura del código en C de GNU R. Estos dos últimos puntos se relacionan directamente con los objetivos del trabajo desarrollado en esta memoria; por el lado de la validación, ya que se plantea incluir un nuevo mecanismo para validar la formalización, mientras que el aspecto del proceso mismo de desarrollo es relevante, pues el sistema y herramientas planteados utilizan elementos, descritos en esta sección, para apoyar y guiar los esfuerzos de implementación de manera incremental.

2.4.1. Aspectos generales

CoqR es una formalización en la forma de un intérprete, definiendo una semántica natural (*natural semantic* o *big-step semantic*). La implementación de ésta se basa en el intérprete en C de R, descrito en la sección 2.1, y es extraída a código en OCaml, como se mencionó en la sección 2.2, de tal manera que pueda ser ejecutada.

Al momento en que se inicia este trabajo de memoria, el intérprete de CoqR cuenta con un conjunto de funcionalidades básicas y con una aplicación de consola, lo que permite trabajar en un ambiente interactivo, donde se pueden introducir expresiones a evaluar, o redirigiendo las salidas de otros comandos en consola hacia el ejecutable, a través de *pipes*. El intérprete está limitado a estas opciones, no siendo posible dar directamente el nombre de un archivo y que lo lea, procese e interprete. Por otro lado, la aplicación permite guardar el ambiente de ejecución en un archivo y cargarlo al momento de lanzar el programa.

La figura 2.7 muestra las distintas formas en las que se puede evaluar una misma expresión, en este caso una suma, en el intérprete de Coq. En las tres figuras se puede ver, en la primera línea, que se llama directamente al ejecutable y que se le pasan opciones, en particular `-initial-state` que permite darle un ambiente de ejecución inicial. En la figura 2.7a se puede ver el ambiente interactivo en el que se ejecuta la expresión, obteniendo su resultado, y luego se utiliza el comando `#quit` que permite terminar la sesión. Por otro lado, las figuras 2.7b y 2.7c muestran la redirección de las salidas de los comandos `echo` y `cat`, respectivamente,

```
$ ./src/runR.native -initial-state src/initial.state
> 1 + 1
[1] 2
> # quit
```

(a) CoqR en modo interactivo

```
$ echo "1 + 1" | ./src/runR.native -initial-state src/initial.state
[1] 2
```

(b) Redirigiendo `echo` a CoqR

```
$ cat ./file.R | ./src/runR.native -initial-state src/initial.state
[1] 2
```

(c) Redirigiendo `cat`, aplicado a un archivo, hacia CoqR

Figura 2.7: Distintas formas de ejecutar una misma expresión en CoqR.

hacia el intérprete, de manera de evaluar la misma expresión que en la primera figura.

De esta manera, el estado en el que se encuentra CoqR al momento de iniciar este trabajo de memoria, permite chequear que el intérprete en Coq sea equivalente a GNU R, ya sea a través del modo interactivo, ingresando expresiones a la consola, o redirigiendo el contenido de un archivo hacia el ejecutable, para luego comparar manualmente si las salidas de ambos intérpretes se asemejan.

En otro ámbito, el intérprete de CoqR interactúa también con código adicional, escrito en OCaml, que no se encuentra formalizado. Es habitual que programas verificados deban interactuar con código adicional no verificado, como lo son librerías que manejan llamadas de entrada y salida al sistema operativo. En el caso de CoqR, esto corresponde al código que se encarga de procesar las expresiones (*parsing*) y al código que se encarga de imprimir los resultados generados por el intérprete. La importancia de este aspecto se debe a que es posible que se introduzcan errores en la ejecución de los programas debido a este código no verificado o discrepancias en los resultados entre GNU R y CoqR.

Por último, la implementación del intérprete en Coq busca una correspondencia visual con el código fuente de GNU R. Este principio se refiere a que sea posible establecer una equivalencia entre ambos intérpretes, dada por una similitud visual entre el código de CoqR y GNU R. Esta semejanza no permite concluir que efectivamente ambos intérpretes se comporten igual, pero sirve como una primera fuente de confianza de que sí son equivalentes.

Obtener esta similitud es difícil, considerando los tipos de lenguajes con los que se trabaja. Recordando lo descrito en la sección 2.3.3, existe una dificultad al momento de querer replicar comportamientos de lenguajes imperativos con efectos, como lo es C para el caso de GNU R, en lenguajes funcionales puros, como es el caso de *Gallina* en Coq, donde no existen estos efectos secundarios. Parte de estas diferencias y dificultades se pueden apreciar en que, por ejemplo, Coq no permite tener funciones cuyo comportamiento no esté completamente definido (por ejemplo, cuando se hace búsqueda de patrones se deben definir todos los casos posibles), mientras que en C es habitual tener comportamientos no definidos. Tampoco existe

la noción de variables y mutación en Coq, ni tampoco es posible definir una función que no termine, mientras que esto en C es perfectamente realizable, entre otras cosas.

En la sección que sigue, se describe la mónada de resultado, una estructura que permite modelar estos efectos y palear algunas de las diferencias entre ambos lenguajes.

2.4.2. Mónada de resultado

La mónada de resultado permite capturar los principales efectos de R, como se describe en la sección 2.3.3, así como atacar la problemática de obtener la correspondencia visual entre CoqR y GNU R. Al momento de iniciar este trabajo de memoria, esta estructura ya se encuentra implementada en CoqR.

Con respecto a los efectos que esta mónada permite capturar, es posible propagar los resultados de ejecución, junto al estado actualizado tras cada expresión evaluada, y además capturar ejecuciones erróneas, propagándose en la pila de ejecución. También permite simular el comportamiento de no terminación de una función (posible de realizar en C, pero no en Coq), para lo cual la estrategia que se adopta es la de agregar un parámetro adicional, el cual va decreciendo con cada paso de ejecución, de tal manera que si llega a un caso base se termina la ejecución. Es decir, define una cantidad máxima de pasos que se pueden realizar antes de que se detenga la computación.

La figura 2.8 muestra la definición de esta mónada en Coq, donde se puede apreciar los distintos constructores monádicos, los cuales son:

- `result_success`: Representa una ejecución exitosa. Además de guardar el valor resultante, de tipo `A`, permite propagar el estado global tras finalizar la computación (de tipo `state`).
- `result_error`: Representa errores lanzados por GNU R, por ejemplo, un error de tipo durante la ejecución. Estos errores no son capturables y detienen inmediatamente el proceso de ejecución. Se incluye una cadena de caracteres para ayudar al proceso de depuración.
- `result_longjump`: Representa una llamada a la función `longjmp` en C. Aparece únicamente en elementos con saltos no locales, tales como `break` o `return`.
- `result_impossible`: Representa comportamientos no especificados en C, por ejemplo, dereferenciar un puntero inválido. Este resultado detiene completamente el proceso de ejecución. De estar CoqR libre de errores, obtener este resultado representaría un error en GNU R.
- `result_not_implemented`: Representa una sección de código no implementada. La importancia de este tipo de resultado es que permite que CoqR pueda ser ejecutado a pesar de no estar completamente desarrollado. Este tipo de resultado es particularmente interesante como ayuda durante el proceso de desarrollo de CoqR.
- `result_bottom`: Representa un cómputo que no terminó después de haber ejecutado una cantidad máxima de pasos de ejecución permitidos. Este constructor se asemeja a la mónada *Maybe* descrita en la sección 2.3.3, en el sentido que permite simular el

```
1 Inductive result (A : Type) :=
2   | result_success : state -> A -> result A
3   | result_error : state -> string -> result A
4   | result_longjump : state -> context -> result A
5   | result_impossible : state -> string -> result A
6   | result_not_implemented : string -> result A
7   | result_bottom : state -> result A.
```

Figura 2.8: La mónada de resultado implementada en Coq.

efecto de una ejecución fallida.

Además, se definen *binders* monádicos que permiten aplicar funciones y propagar los resultados y estados, de una manera transparente. Por ejemplo, el *binder* `let%success` permite evaluar una expresión y en caso de obtener `result_success`, se asocia el valor a un nombre, utilizable posteriormente, así como actualizar el estado de ser necesario. Cualquier otro resultado es propagado hacia arriba invisiblemente. Por lo tanto, este *binder* se puede ver como una asignación cuando en el lado derecho se tiene una llamada a una función.

2.4.3. Correspondencia visual

La correspondencia visual se refiere a poder, visualmente, reconocer código semejante entre GNU R y CoqR, como se mencionó en la sección 2.4.1, y establecer una primera fuente de confianza sobre la equivalencia de ambos intérpretes. Esta confianza es relevante para validar la formalización realizada.

Es importante notar que, aunque la principal razón para tener esta correspondencia visual es aumentar la confianza y validar la formalización, también es posible utilizarla durante el proceso de desarrollo como herramienta de depuración. Si se obtiene una similitud entre el código de GNU R y CoqR, es posible llevar a cabo inspecciones y comparaciones directas entre ambos intérpretes, en búsqueda de errores simples de resolver.

Las figuras 2.9a y 2.9b muestran fragmentos de la función `do_logic2`, en su versión original en GNU R y su implementación en Coq, respectivamente. Esta función es la responsable de poder realizar las operaciones de conjunción y disyunción lógicas, habitualmente reconocidas a través de los símbolos `&&` y `||`. El ejemplo que ilustran estas figuras sirve para ver la correspondencia visual entre GNU R y CoqR. Esta se puede obtener gracias a la mónada de resultado.

Primero que todo, las líneas 4, 5 y 6 en código C se omiten en Coq, dado que corresponden a declaraciones de variables sin asignar valores iniciales. Esto no se puede realizar en Coq, por lo que se posterga la declaración de estas variables al momento donde corresponda utilizarlas. En seguida, el condicional que abarca las líneas 8, 9 y 10 en C es equivalente a las líneas 3, 4 y 5 en Coq; la diferencia de nombre que se tiene con `R_length` se debe a que Coq ya tiene una función llamada `length`, por lo que puede haber una colisión de nombres. Por otra parte, en Coq se tiene una asignación, mientras que en C se llama y usa el valor de retorno inmediatamente, sin embargo, es evidente que en ambos casos se obtiene lo mismo.

A continuación, las líneas 12 y 13, en C, son equivalentes a los pares de líneas 7-8 y 9-10 en Coq, respectivamente. Esta diferencia en notación se debe a que los constructos `CAR` y `CADR`, junto a otros como `TAG`, son muy habituales en GNU R y permiten acceder a distintos campos de listas, por lo que en Coq se define una notación particular para acceder a ellos, dado por `read%list`. Por esta razón, en Coq, en la línea 7 primero se debe leer la lista de argumentos, `args`, con lo que se puede realizar la asignación a `s1` en la línea 8, y luego se realiza nuevamente la lectura de la cola de la lista y así llevar a cabo la asignación de `s2`.

La línea 14 en C es directamente equivalente a la línea 11 en Coq, para luego seguir con un nuevo condicional, donde se sigue el mismo patrón anterior. Por último, en ambos casos se termina con la llamada a `ScalarLogical` y, dado que esta es la última acción, es posible hacer el llamado directamente, sin hacer una asignación como en las expresiones anteriores.

Como se puede apreciar, pese a que la notación no es exactamente la misma, se puede efectivamente relacionar, visualmente, cada línea de C a una o dos líneas en Coq. Existen algunas diferencias, como el uso de `globals`, `runs`, pero que tienen relación con el uso de variables globales o no terminación, mencionados anteriormente.

Por último, hay que recordar que las distintas notaciones y estructuras que permiten tener esta similitud de código ya se encuentran definidas en CoqR al momento de iniciar este trabajo de memoria.

2.4.4. Desarrollo incremental

El proyecto de CoqR es grande, abarcando más de 18.000 líneas de código en Coq y aún creciendo, por lo que se adopta una metodología que permite un desarrollo incremental del intérprete. Para este efecto, se aprovecha el hecho de que GNU R está estructurado alrededor de una tabla de símbolos, la cual contiene todas las funciones presentes en el ambiente inicial de R. Como se mencionó en la sección 2.1, el código de GNU R está escrito en C.

La figura 2.10 muestra una porción de esta tabla. Este arreglo vincula nombres de funciones, tales como `if`, `+` o `while`, a su correspondiente función implementada en C, además de información sobre su aridad, régimen de evaluación, entre otras cosas. En total, se contabilizan más de 700 entradas en esta tabla, de las cuales alrededor de 100 ya están implementadas en CoqR al momento de empezar este trabajo de memoria. Durante el periodo de este proyecto se llegan a implementar cerca de 20 entradas adicionales, lo cual es descrito en la sección 4.2.

Gracias a esta estructura, es posible llevar a cabo un desarrollo incremental de CoqR, ya que cada una de estas entradas es independiente de las demás. Esto posibilita que se dejen secciones sin implementar y solo trabajar con aquellas que son más importantes. En consecuencia, en CoqR se define un núcleo, que corresponde a todas las funciones necesarias para llevar a cabo evaluaciones básicas, definir ambientes, clausuras, etc., mientras que las entradas de esta tabla de símbolos se consideran como funcionalidades adicionales.

```c
SEXP attribute_hidden do_logic2(SEXP call, SEXP op, SEXP args, SEXP env)
{
/* && and || */
    SEXP s1, s2;
    int x1, x2;
    int ans = FALSE;

    if (length(args) != 2)
        error(_("'%s' operator requires 2 arguments"),
              PRIMVAL(op) == 1 ? "&&" : "||");

    s1 = CAR(args);
    s2 = CADR(args);
    s1 = eval(s1, env);
    if (!isNumber(s1))
        errorcall(call, _("invalid 'x' type in 'x %s y'"),
                  PRIMVAL(op) == 1 ? "&&" : "||");
    /* ... */

    return ScalarLogical(ans);
}
```

(a) Función original en GNU R

```
Definition do_logic2 S (call op args env : SEXP)  :=
  add%stack "do_logic2" in
    let%success args_length := R_length globals runs S args using S in
    ifb args_length <> 2 then
        result_error S "operator requires 2 arguments"
    else
        read%list args_car, args_cdr, _ := args using S in
        let s1 := args_car in
        read%list args_cdr_car, _, _ := args_cdr using S in
        let s2 := args_cdr_car in
        let%success s1 := eval globals runs S s1 env using S in
        let%success s1_isNumber := isNumber globals runs S s1 using S in
        if negb s1_isNumber then
            result_error S "invalid type for first argument x."
    (* ... *)
    result_success S (ScalarLogical globals ans).
```

(b) Traducción a Coq

Figura 2.9: Función original en GNU R y traducción a Coq de `do_logic2`.

```
FUNTAB R_FunTab[] = {
  {"if",         do_if,       2,  true,  false},
  {"while",      do_while,    2,  true,  false},
  {"break",      do_break,    0,  true,  false},
  {"return",     do_return,   1,  true,  false},
  {"function",   do_function, -1, true,  false},
  {"<-",         do_set,      2,  true,  false},
  {"(",          do_paren,    1,  true,  true},
  {".Internal",  do_internal, 1,  true,  false},
  {"which",      do_which,    1,  false, true},
  {"+",          do_arith1,   2,  true,  true},
  {"-",          do_arith2,   2,  true,  true},
  {"cos",        do_math20,   1,  true,  true},
  {"sin",        do_math21,   1,  true,  true},
  /* ... */ }
```

Figura 2.10: Fragmento de la tabla de símbolos de GNU R.

2.5. Resumen

En este capítulo se describen los conceptos y herramientas principales involucrados en este trabajo de memoria. En primera instancia se describe el lenguaje de programación R y el asistente de pruebas Coq. En seguida, se prosigue con una descripción de qué es una formalización mecanizada de un lenguaje de programación, mencionando algunas problemáticas que surgen durante un proyecto de este tipo y formas de abordarlas; desde formalizar lenguajes imperativos con efectos en lenguajes funcionales puros, utilizando mónadas para modelar estos efectos, a cómo validar las formalizaciones de lenguajes que no tienen especificaciones rigurosas de sus comportamientos.

En la última sección de este capítulo se describe el proyecto CoqR, que reúne todos los aspectos mencionados en el párrafo anterior, buscando llevar a cabo una formalización mecanizada, en Coq, del lenguaje de programación R. Se describen componentes generales y de uso, así como la mónada de resultado, que es primordial para modelar los principales efectos de R. Se explica también la correspondencia visual directa entre el código de Coq y el código en C del intérprete de R, GNU R, lo que sirve como primera fuente de validación de la formalización llevada a cabo.

Finalmente, pese a que CoqR cuenta con un primer sistema para validarlo con respecto a GNU R, esto no es suficiente; es necesario chequear que efectivamente ambos intérpretes se comporten de una forma equivalente, más allá de su similitud en código. De esta manera, este proyecto de memoria se enmarca dentro del proyecto CoqR, de su validación en un sentido más amplio e integrándose a su proceso de desarrollo iterativo, lo cual es descrito en el capítulo siguiente.

Capítulo 3

Sistema de validación y desarrollo para CoqR

Este capítulo trata sobre el sistema desarrollado en esta memoria, el cual busca validar el intérprete R desarrollado en Coq, para formalizar mecánicamente el lenguaje de programación R. Esta validación sirve como una segunda fuente de confianza, además de la correspondencia visual descrita en la sección 2.4.3, la cual se obtiene a través de la ejecución de bancos de pruebas y comparación del comportamiento entre el intérprete de GNU R y CoqR.

A la vez, este sistema está diseñado para apoyar el desarrollo de CoqR, orientando los esfuerzos hacia los aspectos más apremiantes, a través de reportes y gráficas de resultados.

Durante todo el proceso de esta memoria se utilizó una metodología iterativa e incremental de trabajo, donde periódicamente se validaba y orientaba el trabajo hacia nuevos requisitos emergentes o cambios en requisitos previos.

En las secciones que continúan, se describe en más detalle cada uno de los elementos que conforman el sistema. Primeramente se expone, de manera general, la arquitectura del sistema desarrollado, con sus módulos y la comunicación entre ellos. Luego, se explican los bancos de pruebas utilizados para validar CoqR y el proceso llevado a cabo para ejecutarlas y comparar los valores de salida, generados por el intérprete de referencia en C y el implementado en Coq. A continuación, se describe el sistema de almacenamiento, el modelo utilizado y la interfaz definida para acceder a los datos, así como crear nuevas entradas. Por último, se incluye una descripción de la aplicación web, con sus interfaces y comunicación con el módulo de datos.

3.1. Arquitectura del sistema

En esta sección se describe, a grandes rasgos, el diseño del sistema desarrollado, con sus distintos módulos y la comunicación entre ellos.

La arquitectura del sistema, los distintos módulos y su comunicación, se muestra en la

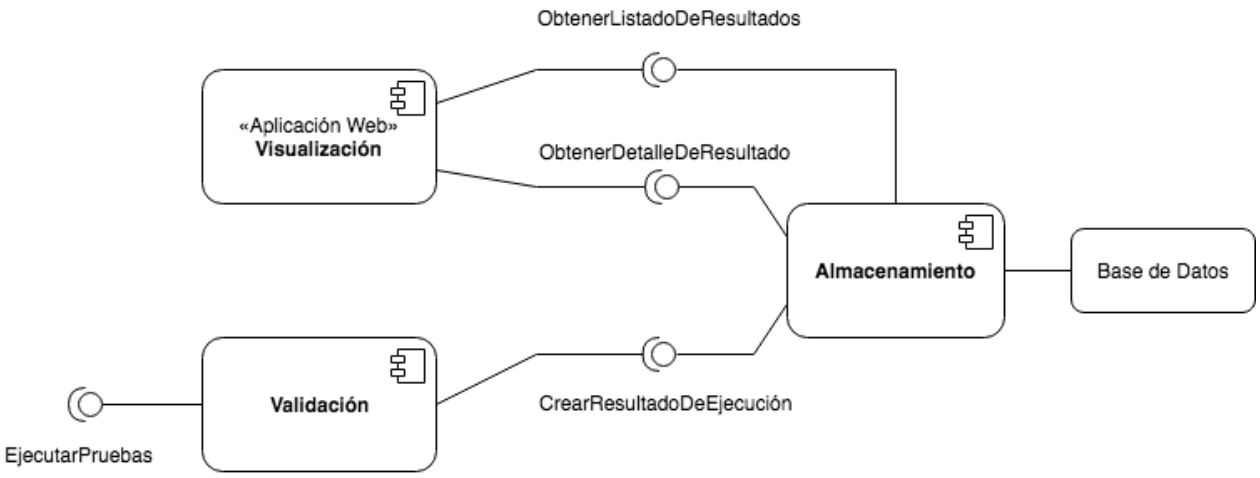

Figura 3.1: Módulos que comprenden el sistema de validación y desarrollo.

figura 3.1. Se puede apreciar que el sistema está compuesto de tres módulos principales: un módulo de validación, encargado de ejecutar las pruebas y comparar los intérpretes, un módulo encargado de las visualizaciones y que corresponde a una aplicación web, y por último un módulo de almacenamiento, que se conecta con la base de datos. Se indican de manera simplificada las interfaces con las cuales se comunican los módulos; la comunicación entre los módulos de validación y visualización con el de almacenamiento se hace a través de solicitudes HTTP, mientras que la interfaz expuesta por el módulo de validación se refiere a que es posible ejecutarla por un usuario o incluirla en otro módulo.

La separación del sistema en estos tres módulos responde a distintos motivos, los cuales en general se refieren al hecho de abstraer las distintas responsabilidades. Por un lado, todo lo que hace referencia a la ejecución de las pruebas y comparación de los intérpretes, de la cual se hace cargo el módulo de validación, mientras que por otro se tiene la capa responsable de los datos, correspondiente al módulo de almacenamiento, y la tercera responsable de la presentación, dada por el módulo de visualizaciones. La idea principal es desacoplar cada módulo, definiendo interfaces con las cuales se comuniquen, de manera que el desarrollo pueda ser realizado de manera independiente para cada uno, siempre que se respeten las interfaces definidas. De esta forma, los cambios que puedan ocurrir en un nivel, no afectan a otro en un nivel separado.

El módulo de validación es el que el usuario puede utilizar para ejecutar los bancos de pruebas, con GNU R y CoqR, y es donde se realizan las comparaciones y se generan reportes de los resultados entregados por ambos intérpretes. Para persistir estos datos, este sistema se conecta con el módulo de almacenamiento que se encuentra corriendo en un servidor del DCC. La figura 3.2 muestra los distintos componentes de este módulo. Por un lado el sistema validador, que se comunica con los intérpretes y que tiene acceso a las pruebas, exponiendo su interfaz para ser ejecutado, además de requerir un componente que reciba los resultados de las pruebas.

De la misma manera, la aplicación web, encargada de mostrar los resultados obtenidos, se comunica con el módulo de almacenamiento, quien provee los datos necesarios para generar gráficas y tablas que el usuario puede utilizar.

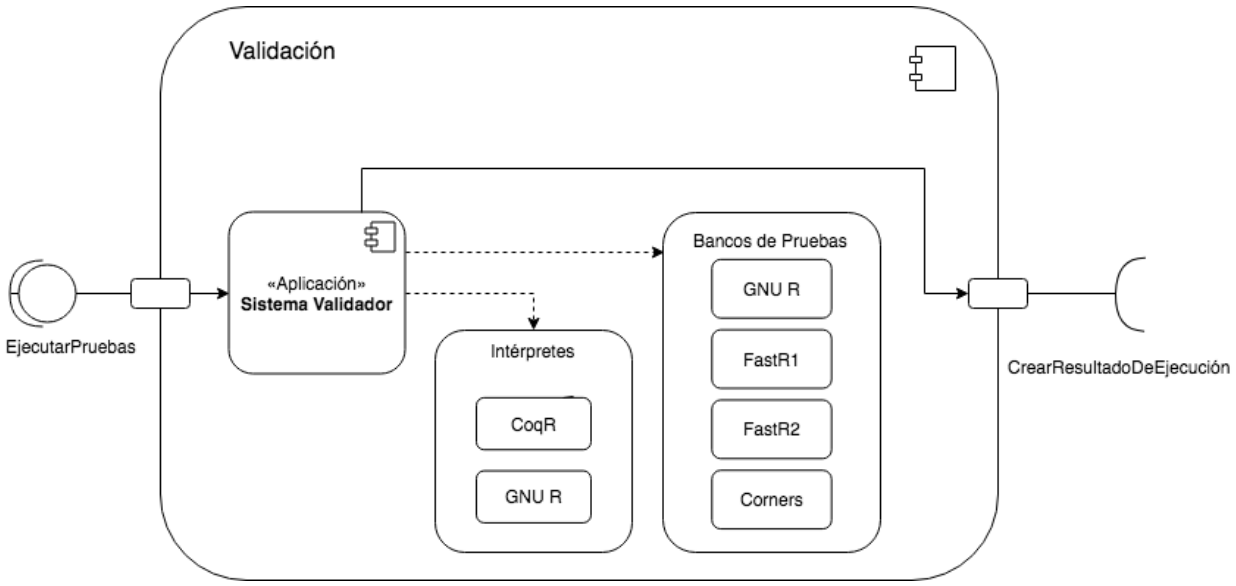

Figura 3.2: Módulo de validación.

En las secciones que continuan se describe con más detalle los distintos elementos que constituyen estos módulos y su funcionamiento, iniciando con los bancos de pruebas.

3.2. Bancos de pruebas

En esta sección se describen los conjuntos de pruebas utilizados para evaluar CoqR con respecto a GNU R.

Se utilizan distintos bancos de pruebas existentes: el banco de prueba de GNU R[1] y dos bancos de pruebas del proyecto FastR [8], en adelante denominados FastR1[2] y FastR2[3]. Para el caso de FastR2 se consideran los archivos de prueba bajo la carpeta *builtins*. Estos dos últimos tienen sus pruebas en archivos Java, por lo que es necesario extraer el código R que se encuentra contenido en cada uno. En total, estos archivos suman más de 17.000 expresiones a ser evaluadas: 1.706 para GNU R, 2.776 para FastR1 y 13.474 para el caso de FastR2.

Antes de utilizar estos bancos de pruebas, se lleva a cabo una fase de filtración, durante la cual se eliminan expresiones que hacen llamadas a sistema o utilizan recursos externos, tales como internet, lectura y escritura de archivos, generar gráficos, entre otros. Además, se excluyen pruebas que requieren paquetes particulares para ejecutarse (por ejemplo, el paquete JIT) o pruebas que se generan a través de funciones y plantillas en Java. La razón principal para excluir estos casos de prueba es la de priorizar los esfuerzos de desarrollo.

Pese a que estas funcionalidades son importantes para los usuarios de R, se considera que escapan el alcance inicial de CoqR, debido a la complejidad que deriva de comparar la

[1] https://github.com/wch/r-source/
[2] https://github.com/h2oai/fastr/tree/master/test/r
[3] https://github.com/oracle/fastr/tree/master/com.oracle.truffle.r.test/src/com/oracle/truffle/r/test

equivalencia de resultados entre ambos intérpretes. Por ejemplo, para las pruebas que generan archivos en formato PDF o que generan gráficos surge el problema de cómo comparar los resultados; se compara directamente el contenido de los archivos PDF o importa más el formato, y similarmente en gráficos, se compara pixel a pixel o de otra forma. Para pruebas que requieren conexión a internet, los resultados enviados desde el servidor pueden variar y surge la duda de si realmente se está probando el servidor o CoqR, además de que los tiempos de ejecución se pueden ver afectados. De esta manera, se prefiere dejar de lado estos casos, por el momento, para poder enfocar los esfuerzos en implementar funcionalidades básicas y transversales.

Tras esta fase de filtración, el banco de pruebas de GNU R es el que más se reduce, pasando de 50 archivos a 26, lo que significa pasar de tener más de 2.600 expresiones al valor actual de 1.700. Por otro lado, FastR2 reduce el número de archivos de 520 a 485, disminuyendo el número de expresiones en no más de 500. Finalmente, FastR1 no se ve modificado por estas filtraciones.

Por otro lado, se desarrolla un conjunto de pruebas propio, denominado Corners, diseñada particularmente para probar casos de borde del lenguaje R o bien, para evaluar expresiones particulares de forma más controlada. Está pensada además para evaluar más rápidamente las funcionalidades que se van implementando en CoqR, durante el proceso de desarrollo. Este banco de pruebas incluye 2.962 expresiones a ser evaluadas.

Es importante notar que las pruebas toman distintas formas: algunos archivos proveen expresiones que deben evaluar a `TRUE`, otros basta con que no arrojen errores, mientras que otros imprimen en salida estándar si es que pasaron o no. Para poder procesar cada prueba de manera uniforme, se evalúa y compara cada paso de ejecución de las pruebas (es decir, cada expresión), tanto para GNU R como CoqR. Visto de esta manera, cada expresión se convierte en sí en una prueba de la validez de CoqR, independiente de la prueba inicial que la contenía. Por lo tanto, en adelante cuando se haga referencia a una prueba, esta puede denotar ya sea un conjunto de expresiones o una expresión sola, según el contexto en el que se discuta. En caso de poder haber alguna ambigüedad, se explicitará a qué caso se refiere.

Por último, vale mencionar que se considera utilizar las herramientas de prueba *TestR* [9, 15] y *Genthat* [6] (correspondiente a la evolución de la primera), las cuales generan pruebas unitarias para funciones en R. Ambas librerías funcionan anotando y ejecutando las funciones, guardando la traza de ejecución con la cual generan pruebas unitarias. Esto sirve para tener pruebas que aseguren que cambios posteriores no rompan funcionalidades existentes. Sin embargo, estas pruebas generadas se basan en comportamientos correctos de las funciones, por lo que se considera inadecuado el uso de estas herramientas para este proyecto en particular.

3.3. Sistema validador

En esta sección se describe el sistema validador, descrito en la sección 3.1, encargado de comunicarse con GNU R y CoqR para ejecutar los bancos de pruebas y realizar la comparación del comportamiento de ambos intérpretes, generando reportes sobre estos resultados.

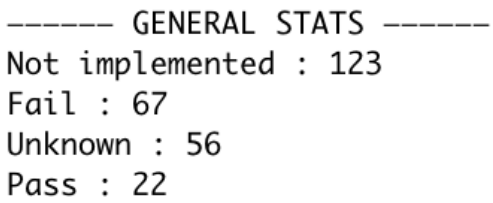

Figura 3.3: Mensaje generado por la aplicación de consola, tras ejecutar un archivo de pruebas.

Primero se describen algunos aspectos generales de la herramienta, para luego continuar con las distintas fases que realiza el módulo para validar CoqR.

3.3.1. Aspectos generales

Este sistema se encuentra desarrollado en Python como una aplicación de consola y es posible utilizarlo sobre un archivo de pruebas individual, permitiendo llevar a cabo ejecuciones y comparaciones rápidas de pruebas, para validación temprana de implementaciones que se realicen en CoqR, o utilizarlo sobre bancos de pruebas conteniendo múltiples archivos, para tener una perspectiva más general del estado del intérprete en Coq. Es posible dar opciones adicionales, entre las cuales vale destacar la opción que permite enviar los resultados a un servidor o generar archivos de texto con los reportes en formato JSON.

En la figura 3.3 se muestra el mensaje final que genera la aplicación de consola tras ejecutar un archivo o un banco de pruebas. Este mensaje contiene información general sobre el resultado de ejecutar las expresiones y comparar los valores generados por GNU R y CoqR. La descripción detallada de cada uno de estos resultados se describe en la sección 3.3.4, más adelante.

Para obtener información más precisa de cada caso, el sistema proporciona una herramienta que permite imprimir en pantalla reportes de estos resultados. La figura 3.4 muestra cómo se ven estos reportes más detallados, donde se aprecia información como el archivo de prueba, la línea que se ejecutó, para poder identificarla si se revisa el archivo, así como la expresión evaluada y los valores de salida generados por cada intérprete. Inicialmente, se utiliza esta herramienta para depurar o analizar el comportamiento de nuevas funcionalidades implementadas. Sin embargo, rápidamente se vuelve evidente la necesidad de desarrollar herramientas y visualizaciones más apropiadas, que permitan analizar y dimensionar correctamente los resultados obtenidos, para poder tomar mejores decisiones respecto a qué funcionalidades implementar y facilitar el desarrollo general de CoqR.

Estas visualizaciones recién descritas aún se mantienen en el sistema, pero más bien por motivos históricos, siendo las primeras formas desarrolladas durante la memoria para visualizar los resultados de las pruebas. Estas motivaron el desarrollo apropiado de otras formas de presentar la información, lo cual culmina en la implementación de una aplicación web con distintas gráficas y tablas, descrita en la sección 3.5.

A continuación se describen las distintas fases que se llevan a cabo para ejecutar los bancos

```
[
  {
    "filename": "TestSimpleArrays.R",
    "line": 1,
    "expression": "a = array()",
    "R": "[1] NA\n",
    "CoqR": "Not implemented: [do_array] Function not implemented: do_array.
     ↪  (Execution stack: eval_global, eval, evalList, eval, do_set, eval,
     ↪  applyClosure, R_execClosure, eval, do_begin, eval, do_if, eval,
     ↪  do_return, eval, do_internal, do_array)\nAn error lead to an undefined
     ↪  state. Continuing using the old one.\nAn error lead to an undefined
     ↪  result.\n",
  },
  {
    "filename": "TestSimpleArrays.R",
    "line": 2,
    "expression": "a = array()",
    "R": "[1] NA\n",
    "CoqR": "Not implemented: [do_array] Function not implemented: do_array.
     ↪  (Execution stack: eval_global, eval, evalList, eval, do_set, eval,
     ↪  applyClosure, R_execClosure, eval, do_begin, eval, do_if, eval,
     ↪  do_return, eval, do_internal, do_array)\nAn error lead to an undefined
     ↪  state. Continuing using the old one.\nAn error lead to an undefined
     ↪  result.\n",
  }
]
```


Figura 3.4: Fragmento del mensaje generado por el módulo, para visualizar los resultados en detalle.

de pruebas y la comparación entre GNU R y CoqR.

3.3.2. Interpretación

Esta es la primera fase en el proceso de ejecución y comparación de los bancos de pruebas, llevado a cabo en el módulo de validación. En esta etapa se realiza la ejecución de los archivos de prueba y se generan los resultados iniciales para cada intérprete. La ejecución de los archivos se realiza de manera secuencial, es decir, primero se ejecutan todas las expresiones con GNU R y, una vez que éste haya terminado, se ejecuta con CoqR. Ambos intérpretes realizan el mismo proceso iterativo, para cada archivo de un banco de pruebas, con lo cual obtienen resultados en crudo. En seguida se describen las distintas subetapas que conforman este proceso iterativo.

Identificación del formato del archivo de pruebas

Las pruebas contenidas dentro de un archivo pueden venir en distintos formatos, por lo que inicialmente el intérprete, sea GNU R o CoqR, debe identificar el formato particular que adopten éstas. Identificar este formato importa al momento de interpretar, ya que afecta al ambiente de ejecución en el que se evalúan las expresiones, como se describe en una sección posterior. Los dos formatos posibles son: que las pruebas abarquen múltiples líneas y expresiones dentro del archivo, en adelante *formato multilínea*, o que cada prueba utilice una única línea dentro de éste, pudiendo contener una o más expresiones, en adelante *formato línea a línea*. Cabe notar que el único banco de pruebas que utiliza el formato multilínea es el de GNU R, mientras que FastR1, FastR2 y Corners tienen archivos que utilizan el formato línea a línea.

En la figura 3.5 se pueden ver extractos de dos archivos, uno con el formato multilínea y otro con el formato línea a línea. En el caso multilínea (figura 3.5a) se puede ver que efectivamente hay expresiones que requieren múltiples líneas de código o que hacen referencia a expresiones definidas en líneas anteriores. Por ejemplo, las líneas 8 hasta la 14 definen una única expresión, correspondiente a la asignación de la variable `rErr`, y las líneas 17 y 19 contienen expresiones que utilizan la variable `Meps`, definida en la línea 3.

En cambio, en un archivo con formato línea a línea (figura 3.5b), cada prueba se define en una única línea, conteniendo toda la información necesaria para ser ejecutada y sin necesidad de hacer referencia a variables o funciones definidas en líneas anteriores. Por ejemplo, la línea 9 contiene cuatro expresiones que se ejecutan secuencialmente (de izquierda a derecha), y cada una depende únicamente de las expresiones que la preceden en esa misma línea, pero no dependen de otras en líneas anteriores. Por ejemplo, si la función `f` no estuviese definida, ambas llamadas a la función `g` fallarían, pese a que en la línea 7 también se define una función `f`.

Por defecto, el sistema desarrollado en esta memoria considera que las pruebas vienen en el primer formato indicado. Para identificarlo como el segundo caso, se puede agregar la

```
.proctime00 <- proc.time()
opt.conformance <- 0
Meps <- .Machine $ double.eps

set.seed(1)

options(rErr.eps = 1e-30)
rErr <- function(approx, true, eps = .Options$rErr.eps)
{
  if(is.null(eps)) { eps <- 1e-30; options(rErr.eps = eps) };
  ifelse(Mod(true) >= eps,
   1 - approx / true,
   true - approx)
}


abs(1- .Machine$double.xmin * 10^(-.Machine$double.min.exp*log10(2)))/Meps < 1e3
if(opt.conformance)
  abs(1- .Machine$double.xmax * 10^(-.Machine$double.max.exp*log10(2)))/Meps <
   ↪  1e3
```

(a) Archivo en formato multilínea

```
f <- function (...) missing ("x") ; g <- function (x) f (x) ; g (1)

f <- function (x) function (y) missing (x) ; f () ; f ("y") ()

f <- function (m, x) m (x) ; f (missing) ; f (missing, NULL) ; f (missing, f ())

f <- function (x, y, z) x ; g <- function (x, ...) f (..., x) ; g (1) ; g (1, 2)

f <- function (x) x ; g <- function (...) f (...) ; g (1) ; g ()

f <- function (x, y, z) y ; g <- function (...) f (...) ; g (1, 2, 3) ; g (1, 2)

f <- function (...) ... ; f (1)

head <- function (x, ...) x ; head (1, 2, 3) ; head ()

f <- function (x, ...) if (x) TRUE else f (TRUE, x, ...) ; f (FALSE)

f <- function (x, ...) if (x) TRUE else f (..., x) ; f (FALSE, FALSE, TRUE)
```

(b) Archivo en formato línea a línea

Figura 3.5: Fragmentos de los dos tipos de archivos de pruebas.

etiqueta `#@line` al comienzo del archivo o se puede definir un nuevo archivo, bajo el nombre de *RSettings*, cuyo contenido debe ser esta misma etiqueta. Este nuevo archivo se debe crear en la misma carpeta que los archivos de pruebas a ejecutar y permite no tener que manualmente modificar cada uno, para agregar la etiqueta recién mencionada.

Luego de identificado el formato del archivo, se puede continuar con el procesamiento de los archivos y la interpretación de las expresiones.

Procesamiento del archivo de pruebas

En esta etapa se lee y procesa el archivo de pruebas, para obtener las expresiones a ser evaluadas por los intérpretes. Los resultados generados tras procesar el archivo es levemente distinta según el formato del archivo. Para el formato multilínea se procesan todas las expresiones contenidas en el archivo y se genera una única lista de expresiones a ser evaluadas, mientras que, para el formato línea a línea, se genera una lista con listas anidadas, donde cada una de éstas representa una línea del archivo y sus expresiones. Si se considera nuevamente las figuras 3.5a y 3.5b, la primera da una lista con 8 expresiones, mientras que la segunda da una lista con 10 listas anidadas; la primera y segunda lista anidada contienen 3 expresiones, la tercera 4 expresiones, etc.

Con la lista de expresiones ya generadas, es posible continuar con la fase de interpretación.

Interpretación de expresiones

En esta fase se lleva a cabo la interpretación de las expresiones contenidas en el archivo de pruebas. A continuación se detallan algunos aspectos importantes sobre este proceso, desde el manejo del ambiente de ejecución, al orden y forma de evaluar las expresiones, entre otras cosas.

En primer lugar, es importante mencionar el orden y la forma en que se realiza la interpretación de las expresiones. Una forma de ejecutar una secuencia de expresiones es de interpretarlas todas de una sola vez, dado que una secuencia de expresiones es en sí una expresión, por lo que es válido hacerlo. Sin embargo, este enfoque presenta ciertos inconvenientes. Por un lado, GNU R invisibiliza los valores generados al evaluar ciertas expresiones, como la asignación, donde el valor sin invisibilizar que genera es el mismo valor que se está asignando. Esto puede traer problemas, ya que puede ser difícil asociar correctamente los valores de salida con la expresión correspondiente que lo generó.

Esto se puede apreciar en la figura 3.6, en la que se tiene tres expresiones separadas por el símbolo `;`, y en la que se puede ver un único valor de salida, correspondiente a un mensaje de error. La primera expresión corresponde a una asignación, identificada por el símbolo `<-`, donde se asigna al identificador `f` una función. Sin invisibilizar, esta expresión retorna el código de la función asignada. La segunda es una aplicación de la función `f` definida justo antes, la cual asigna el valor del parámetro dado a una variable y retorna este valor. Sin embargo, como es una asignación, nuevamente se invisibiliza el valor asignado (en este caso,

```
> f <- function (x) y <- x ; f(1) ; y
Error: object 'y' not found
```

Figura 3.6: Evaluación de una secuencia con valores invisibilizados.

```
> f <- function (x) x ; g() ; f(1)
Error in g(): could not find function "g"
```

(a) Secuencia con error en GNU R

```
> f <- function (x) x ; g() ; f(1)
(closure)
Error: Could not find function "g". (Execution stack: eval_global, eval, findFun3)
An error lead to an undefined result.
[1] 1
```

(b) Secuencia con error en CoqR

Figura 3.7: Evaluación de una secuencia de expresiones.

el valor numérico 1). La tercera expresión evalúa un identificador `y`, que no existe en el ambiente, lo que genera un mensaje de error.

Otro inconveniente que presenta ejecutar toda una secuencia de expresiones de una sola vez, es que GNU R detiene la ejecución de una secuencia de expresiones si es que una de ellas resulta en un error, mientras que CoqR las evalúa todas, independiente de si se produce un error entremedio. Una solución directa a esto es dejar los casos que producen errores al final de la secuencia, pero esto significa un esfuerzo adicional de parte del usuario que define los casos de prueba, además que da espacio a que hayan equivocaciones y no se realice apropiadamente este paso, pudiendo dejar casos erróneos en otro orden. La figura 3.7 muestra los resultados que genera GNU R y CoqR al evaluar una secuencia de expresiones con un error entremedio. La segunda expresión es la que provoca que se lance un error, lo que provoca que GNU R se detenga y no evalúe la tercera expresión, la cual debería retornar un valor numérico. En cambio, CoqR sí interpreta cada expresión, independiente de qué ocurra en cada paso.

Otro aspecto relevante a tener en cuenta, sobre la interpretación de las expresiones, es el ambiente de ejecución bajo el cual estas se ejecutan. La importancia radica en que, para el formato línea a línea, cada línea contiene toda la información necesaria para su ejecución, por lo que se debe trabajar cada caso con un ambiente nuevo. En cambio, para el caso multilínea no es así, dado que hay expresiones que pueden hacer referencia a variables o funciones definidas en líneas anteriores, por lo que *a priori* lo adecuado es manejar un único ambiente a lo largo de toda la interpretación.

Habiendo dicho esto, el sistema desarrollado lleva a cabo esta fase de interpretación de la siguiente manera. En primer lugar, las expresiones de la lista de expresiones (sea anidada o no) que se obtiene del paso descrito en la sección anterior, se evalúan de manera secuencial, pero cada una de forma individual, como si no existiese dentro de una secuencia. Utilizando el ambiente de ejecución se puede persistir la información generada en cada paso, de manera

```
> (f <- function (x) y <- x) ; (f(1)) ; (y)
function (x) y <- x
[1] 1
Error: object 'y' not found
```

Figura 3.8: Evaluacion de una secuencia de expresiones, cada una rodeada de paréntesis.

que no hayan expresiones que fallen por definiciones faltantes o porque no se persistió una modificación a una variable, entre otras cosas. Esto permite que, por un lado, no se detenga completamente la ejecución de GNU R si es que hay una expresión que lanza un error y, por otro, que se pueda forzar la visibilidad del resultado generado por la expresión. Esto último se obtiene rodeando a la expresión con paréntesis, lo cual fuerza a GNU R para que retorne el resultado generado.

En la figura 3.8 se puede apreciar la misma secuencia de expresiones que en la figura 3.6, salvo que cada una de las tres expresiones ha sido rodeada de paréntesis. Se puede ver que, efectivamente, la secuencia de expresiones con paréntesis entrega más resultados que la secuencia sin paréntesis, donde solo se tenía un mensaje de error. Los resultados que se pueden apreciar son: el código de una función, correspondiente a la primera asignación, un valor numérico asociado a la aplicación de la función y, finalmente, un mensaje de error dado por evaluar el identificador `y` sin haberlo definido previamente.

Respecto al manejo del ambiente durante la interpretación, tanto GNU R como CoqR permiten generar archivos donde almacenar esta información del ambiente. De esta forma, el sistema lee estos archivos antes de interpretar una expresión y lo actualiza luego de que esta se ejecuta, permitiendo persistir esta información a lo largo de todas las expresiones. Cabe mencionar que para los archivos con formato multilínea, se define un único ambiente de ejecución utilizado a lo largo de todas las expresiones, mientras que para el formato línea a línea, se crean nuevos ambientes para cada línea, lo que permite que éstas sean independientes las unas con las otras.

3.3.3. Procesamiento de valores de salida

Una vez interpretado el código en R, obtenemos los resultados de cada intérprete, GNU R y CoqR, en formato de cadenas de caracteres. Estas cadenas de caracteres no pueden ser comparadas directamente, ya que hay muchos detalles sintácticos que los hacen superficialmente diferentes. Esto viene dado por la parte de CoqR encargada de imprimir los resultados, la cual pertenece a la sección no formalizada del lenguaje. Esta herramienta no sigue una correspondencia visual con la de GNU R, que de por sí es bastante compleja, lo que introduce pequeñas discrepancias en los resultados generados. Por ejemplo, al imprimir vectores con GNU R, algunos tienen espaciado a la izquierda, otros a la derecha, etc. Otra fuente de discordancia son los espacios o saltos de línea. Por lo tanto, en esta fase se procesan las cadenas de caracteres producidas por ambos intérpretes, a un conjunto de valores en común que puedan ser comparados correctamente.

Este paso es llevado a cabo por *procesadores de resultados* (implementados en Python),

```
> 1:100
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23
 [24]  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46
 [47]  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
 [70]  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92
 [93]  93  94  95  96  97  98  99 100
```

(a) Resultado vectorial numérico de GNU R

```
> 1:100
[1] 1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23
24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
48  49  5051  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
97  98  99  100
```

(b) Resultado vectorial numérico de CoqR

Figura 3.9: Valores de salida de vectores numéricos.

uno que se encarga de las salidas de GNU R y el otro de las salidas de CoqR. Cada procesador hace calzar la cadena de caracteres de salida que reciba, con una lista de expresiones regulares. Este emparejamiento se hace en orden y apenas la salida calza con una expresión regular, se crea un objeto de tipo `Result`, deteniendo el proceso de emparejamiento. Hay nueve subclases de `Result`: `NotImplementedResult`, `ImpossibleResult`, `ErrorResult`, `NullResult`, `FunctionResult`, `UnknownResult`, `InvisibleResult`, `ListResult` y `VectorResult`, que además tiene las subclases para resultados de vectores booleanos, numericos y de cadenas de caracteres.

La diferencia entre los procesadores para GNU R y CoqR reside entonces en las expresiones regulares que utiliza para reconocer las cadenas de caracteres de salida generados por los intérpretes y, por lo tanto, en las instancias particulares de `Result` que produce.

Por último, cuando se crean los objetos de los resultados, es posible que los procesadores incluyan información adicional que es necesaria posteriormente para la comparación. Por ejemplo, si calza la expresión regular para vectores numéricos, el procesador extrae los valores reales contenidos dentro del valor de salida y los almacena en el correspondiente objeto `NumericVectorResult`.

A continuación se muestran algunos ejemplos que ilustran estas diferencias entre los valores generados por cada intérprete. En primer lugar, en la figura 3.9 se puede apreciar resultados numéricos generados por ambos intérpretes; la figura 3.9a muestra el resultado de GNU R y la figura 3.9b la de CoqR. Ambos resultados son equivalentes en cuanto a contenido numérico, sin embargo, como cadena de caracteres no lo son. En el caso de GNU R se tiene cinco líneas que comienzan con un número entre corchetes, por ejemplo `[[1]]` ó `[[24]]`, que indican el índice del primer elemento en la nueva línea, mientras que en CoqR se tiene una única línea, sin detalles adicionales. Ambos resultados son procesados y se crean instancias de `NumericResult`, cada uno conteniendo una lista de números, del 1 al 100.

De manera similar, tanto la figura 3.10a como la figura 3.10b muestran resultados erróneos obtenidos con ambos intérpretes. Si se compara ambos resultados directamente no tendría

```
> g()
Error in g() : could not find function "g"
```

(a) Resultado erróneo en GNU R

```
> g()
Error: Could not find function “g”. (Execution stack: eval_global, eval, findFun3)
An error lead to an undefined result.
```

(b) Resultado erróneo en CoqR

Figura 3.10: Valores de salida para casos de error.

sentido, porque evidentemente no son iguales, pero aún así, representan lo mismo. Por lo mismo, a través de los procesadores descritos en esta sección, se obtienen resultados de tipo `ErrorResult` para cada valor de salida y, así, son comparables como tal.

Por último, las figuras 3.11a y 3.11b muestran un caso más complejo, que corresponde al resultado de listas generados por GNU R y CoqR, respectivamente. Este caso es más complejo que los anteriores, dado que una lista puede contener diferentes tipos de objetos, incluidas listas anidadas, como es posible ver en la expresión en la primera línea de las imágenes mencionadas. Además, es posible dar nombres a los distintos índices, como se puede ver en la expresión `list(.name=3, ...)`, donde `.name` corresponde al nombre que se le da a la primera casilla. La dificultad de esto recae en cómo se imprime el resultado y cómo verificar que sean equivalentes. Por un lado, se puede ver la notación `$nombre` en la salida de GNU R, para identificar casillas con nombre, mientras que para CoqR, esto se ve reflejado en las dos últimas líneas, donde se describe el atributo `names`, con los nombres de cada casilla. Por otro lado, se tiene la notación `[[número]]` para indicar el índice de una casilla, o bien, el índice de una lista anidada, mediante una cadena de estos bloques. Por lo tanto, para procesar este tipo de salida, se debe trabajar tanto el tema de índices como el valor de cada objeto en cada casilla.

3.3.4. Comparación de resultados

Después de procesar las salidas de cada intérprete y obtener los objetos de resultado, simplemente se procede a utilizar la interfaz que estos proveen y con ello realizar la comparación. Esta comparación arroja un código que indica el resultado final de la prueba. De esta manera se puede categorizar cada paso de ejecución en una de siete categorías, denotando distintos escenarios posibles.

Tipos de resultados obtenidos de la comparación

Las siete categorías mencionadas anteriormente son:

1. Pasó: Corresponde a una ejecución exitosa de una prueba, es decir, los resultados de

```
> list(.name=3, list(list(a=TRUE, NULL)), pi)
$.name
[1] 3

[[2]]
[[2]][[1]]
[[2]][[1]]$a
[1] TRUE

[[2]][[1]][[2]]
NULL


[[3]]
[1] 3.141593
```

(a) Resultado de lista en GNU R

```
> list(.name=3, list(list(a=TRUE, NULL)), pi)
[[1]]
[1] 3

[[2]]
[[2]][[1]]
[[2]][[1]][[1]]
[1] TRUE

[[2]][[1]][[2]]
NULL

attr(,"names")
[1] "a" ""


[[3]]
[1] 3.14159

attr(,"names")
[1] ".name" ""       ""
```

(b) Resultado de lista en CoqR

Figura 3.11: Valores de salida para expresiones de listas.

GNU R y de CoqR son equivalentes.

2. Falló: Corresponde a una ejecución fallida de una prueba, es decir, los resultados de GNU R y CoqR no son equivalentes.
3. No implementado: Corresponde a una ejecución donde se alcanzó el constructor para resultado no implementado en CoqR, `result_not_implemented`, descrito en la sección 2.4.2. Esta categoría señala la necesidad de implementar funcionalidades pendientes en el intérprete.
4. Imposible: Corresponde a una ejecución donde se alcanzó el constructor para resultado imposible de CoqR, `result_impossible`, descrito en la sección 2.4.2. Como se mencionó previamente, esta categoría señala la presencia de un bug en CoqR o en GNU R.
5. Desconocido: Corresponde a una ejecución donde no es posible determinar el tipo de resultado obtenido, ya sea en GNU R o en CoqR; no es posible decidir si es un resultado correcto o no. Esto ocurre cuando el procesamiento de las cadenas de caracteres falla en detectar a cuál caso corresponde. Esto puede ocurrir, por ejemplo, debido a atributos especiales que modifican la forma en que se imprimen los vectores, de manera que los procesadores no son capaces de detectar estos cambios sutiles.
6. Potencialmente pasó/falló: Corresponde a ejecuciones que fueron exitosas o fallaron, luego de que fallara un paso de ejecución previo. Este tipo de resultado se da en archivos de pruebas con formato línea a línea, en el que si un paso de ejecución falla o da un resultado no implementado, no es posible hacer alguna aseveración sobre los pasos siguientes; los resultados siguientes pueden ser meramente coincidencia.

Como se puede notar, más allá del caso *Pasó*, todas las otras categorías existen para ayudar en el desarrollo de CoqR, permitiendo refinar los motivos de porqué una prueba podría haber fallado.

Retomando los ejemplos dados en la sección anterior, los resultados obtenidos al momento de comparar estos resultados serían: tanto para el caso de las figuras 3.9a y 3.9b, como el caso de las figuras 3.10a y 3.10b, se consideran equivalentes y pasan. Sin embargo, para las figuras 3.11a y 3.11b, el resultado final es fallido, ya que se puede apreciar que el último valor, correspondiente a `pi` no es el mismo en ambos casos; hay una diferencia dada por la precisión de números de punto flotante. Por el momento no se ha implementado una estrategia que permita identificar casos numéricos equivalentes, dentro de una cierta vecindad dada por un valor *épsilon*, principalmente porque se ha priorizado el trabajo en la detección de resultados de otro tipo y no en determinar valores apropiados para este *épsilon*. Además, cabe recordar que R es muy utilizado en el área de datos, por lo que los casos de uso pueden requerir precisiones más o menos finas, según el caso, lo que significa un análisis más profundo sobre cómo identificar apropiadamente estos resultados numéricos.

3.4. Módulo de almacenamiento de datos

Esta sección describe el módulo encargado de almacenar los datos y proveer los datos, al módulo de visualizaciones, para generar las gráficas y tablas pertinentes.

En el servidor Quetrahue del DCC se encuentra corriendo una aplicación desarrollada en Django[4], que se conecta a una base de datos en PostgreSQL[5]. Esta aplicación habilita una interfaz, a través de la cual el módulo de validación puede enviar los resultados finales obtenidos, luego de ejecutar los bancos de pruebas. Esta interfaz también permite al módulo de visualizaciones acceder a estos datos.

En las secciones siguientes se describe el modelo de datos utilizado y la interfaz implementada.

3.4.1. Modelo de datos

El modelo de datos incluye dos entidades principales, `Test` y `ExpressionReport`, y una menor, correspondiente a los usuarios registrados en el sistema. En la figura 3.12 se puede ver un diagrama resumiendo estas entidades y sus relaciones.

La entidad `Test` representa una ejecución de un conjunto de pruebas o, dicho de otra manera, a la ejecución de un conjunto de expresiones. Esto puede llegar a ser desde un banco de pruebas ejecutado, a múltiples bancos de pruebas ejecutados en conjunto, o bien, un único archivo con una sola expresión. Esta entidad contiene información general sobre la ejecución; información del sistema donde se ejecutó, las versiones de los intérpretes, la fecha, entre otros. Además, ésta mantiene una relación uno a muchos con la entidad `ExpressionReport`.

Como se indicó previamente, la comparación se hace para cada expresión que se encuentre

[4]https://www.djangoproject.com/
[5]https://www.postgresql.org/

en los archivos. De esta manera, la entidad `ExpressionReport` contiene toda la información relativa a una expresión individual interpretada; tiempos de ejecución para cada intérprete, valores de salida iniciales y procesados, el archivo al que pertenece y el resultado de la comparación, entre otros.

3.4.2. API

Se desarrolla una interfaz que permite acceder a los resultados de las pruebas, así como crear nuevas entradas en la base de datos. Se utiliza un sistema de autenticación basado en fichas, generadas para cada usuario registrado en el sistema. La idea principal de este sistema de autenticación es que, a través de un nombre de usuario y contraseña, se obtiene una ficha, que corresponde a un pedazo de información (por ejemplo, una cadena de caracteres), el cual se utiliza en posteriores solicitudes para certificar que corresponde a un usuario válido y poder acceder a recursos. En este caso, se utiliza un sistema de fichas desarrollado por Django mismo.

La tecnología utilizada es Django REST framework[6] y habilita los siguientes puntos de acceso.

GET

Todos estos puntos son de acceso libre, es decir, no requieren una ficha de autenticación.

- `/api/` : Devuelve la lista de todos los puntos de acceso habilitados.
- `/api/tests/` : Devuelve la lista de todas las ejecuciones de pruebas almacenados en el sistema, como se describió en la sección 3.4.1. La información devuelta corresponde al usuario que ejecutó el conjunto de pruebas, la fecha y el título que se le dio.
- `/api/tests/<número>/` : Devuelve los resultados obtenidos para la ejecución de pruebas indicada por el parámetro dado.

POST

- `/api/token-auth/` : Dado un nombre de usuario y contraseña, devuelve la ficha de autenticación necesario para poder enviar nuevos resultados de pruebas.
- `/api/tests/` : Permite el envío de datos sobre una nueva ejecución de pruebas. Requiere una ficha de autenticación.

[6] http://www.django-rest-framework.org/

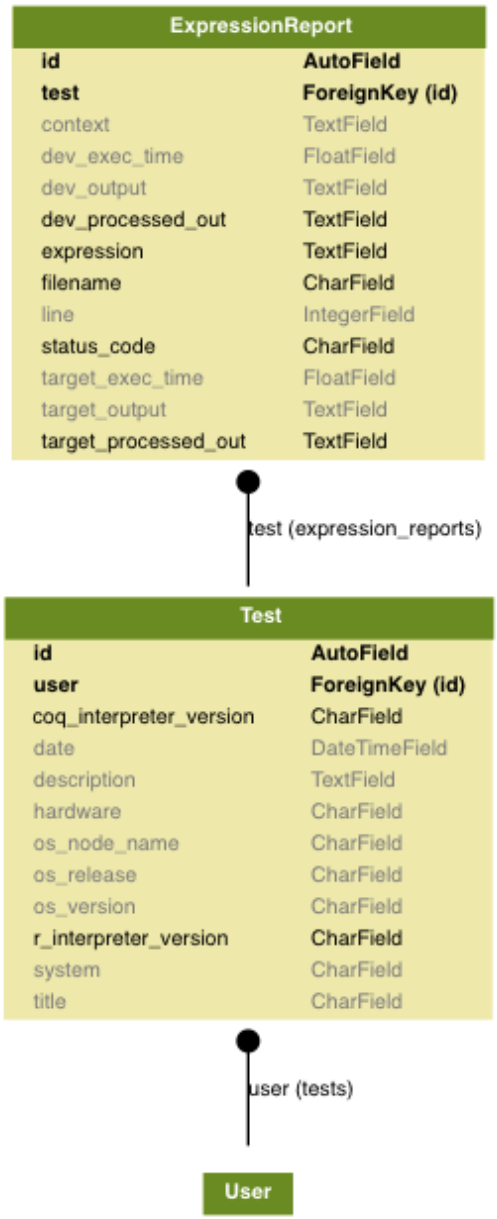

Figura 3.12: Modelo de datos utilizado para almacenar los resultados de las pruebas.

3.5. Módulo de visualizaciones

Esta sección trata sobre el módulo encargado de generar visualizaciones sobre los resultados generados tras ejecutar los bancos de pruebas.

Se dispone de una aplicación corriendo en el servidor Quetrahue del DCC[7], donde es posible ver gráficas y figuras sobre los resultados obtenidos de las pruebas. Esta aplicación accede a los datos almacenados en la base de datos previamente mencionada, utilizando para ello la API implementada.

Las tecnologías utilizadas para el desarrollo de esta aplicación son CoreUI[8], una plantilla sobre la cual construir la aplicación, y ReactJS[9].

La aplicación consiste de dos vistas; la primera permite visualizar el listado general de las ejecuciones de pruebas, mientras que la segunda permite visualizar en detalle los resultados de una ejecución de pruebas. En las secciones siguientes se ahonda más en estas interfaces.

3.5.1. Vista de listado de ejecuciones de pruebas

Esta es la vista inicial de la aplicación, proveyendo una tabla con el listado de ejecuciones de pruebas almacenados en el sistema, como se puede ver en la figura 3.13. La aplicación obtiene los datos a través de una solicitud GET al punto de acceso `/api/tests/` descrito anteriormente.

El propósito de esta vista es servir como punto inicial de acceso para todos los casos de prueba realizados. Es posible ordenar la tabla por fecha, usuario o título, además de permitir filtrar según cada uno de estos campos. Por último, al hacer click en una de las filas de la tabla se redirige al usuario a la vista de detalle, descrita a continuación.

3.5.2. Vista de detalle de resultados

Esta vista se encarga de obtener los resultados particulares de ejecución de una serie de pruebas, solicitándolos al servidor a través del punto de acceso `/api/tests/<número>`, mencionado anteriormente en la sección 3.4.2, y generar visualizaciones a partir de estos. Su propósito general es el de permitir analizar los resultados obtenidos, a través de una serie de figuras y tablas.

En primera instancia, se tiene una sección con información general de las expresiones ejecutadas, como se ve en la figura 3.14. Esta información incluye datos del sistema donde se ejecutaron las pruebas, tiempos de ejecución, así como información de las versiones de los intérpretes o fecha de ejecución. En particular esta sección es importante si se quiere saber la

[7] https://coqr.dcc.uchile.cl/
[8] https://coreui.io/
[9] https://reactjs.org/

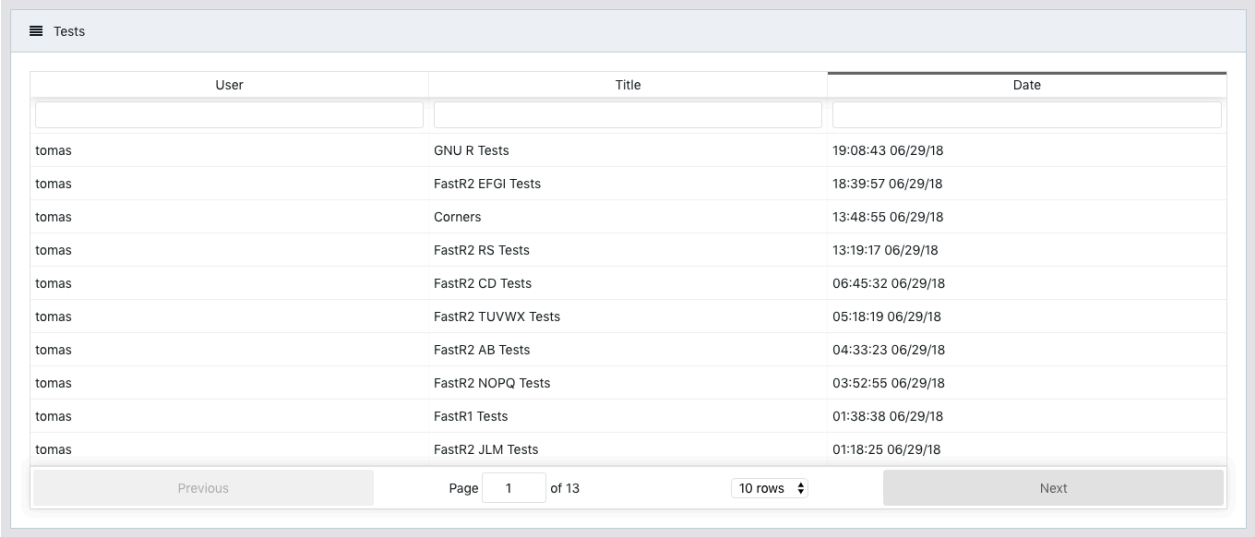

Figura 3.13: Vista inicial mostrando la lista de ejecuciones de pruebas.

Figura 3.14: Sección que despliega información general de la ejecución de un conjunto de expresiones.

versión de CoqR sobre la cual se llevaron a cabo las pruebas, de manera de poder actuar en base a esta información: en casos de depuración de código, para analizar cuál fue la última funcionalidad implementada, etc.

Seguidamente, vienen dos secciones con información sobre los resultados obtenidos. En la figura 3.15 se puede apreciar, en la parte superior, la primera sección, cuyo propósito es desplegar un resumen de los resultados obtenidos y las categorías en la que estos caen. Por lo tanto, sirve como inspección rápida para evaluar los casos de prueba ejecutados. Por ejemplo, en esta sección se puede ver rápidamente si hubo un caso con resultado imposible. Bajo esta sección viene una complementaria, donde se dispone de tablas con información detallada para cada categoría y donde cada fila corresponde a una expresión evaluada tanto por GNU R como CoqR. La información que se muestra incluye: nombre de archivo y línea en éste, de manera de poder identificar el punto exacto y analizarlo en detalle, la expresión en particular

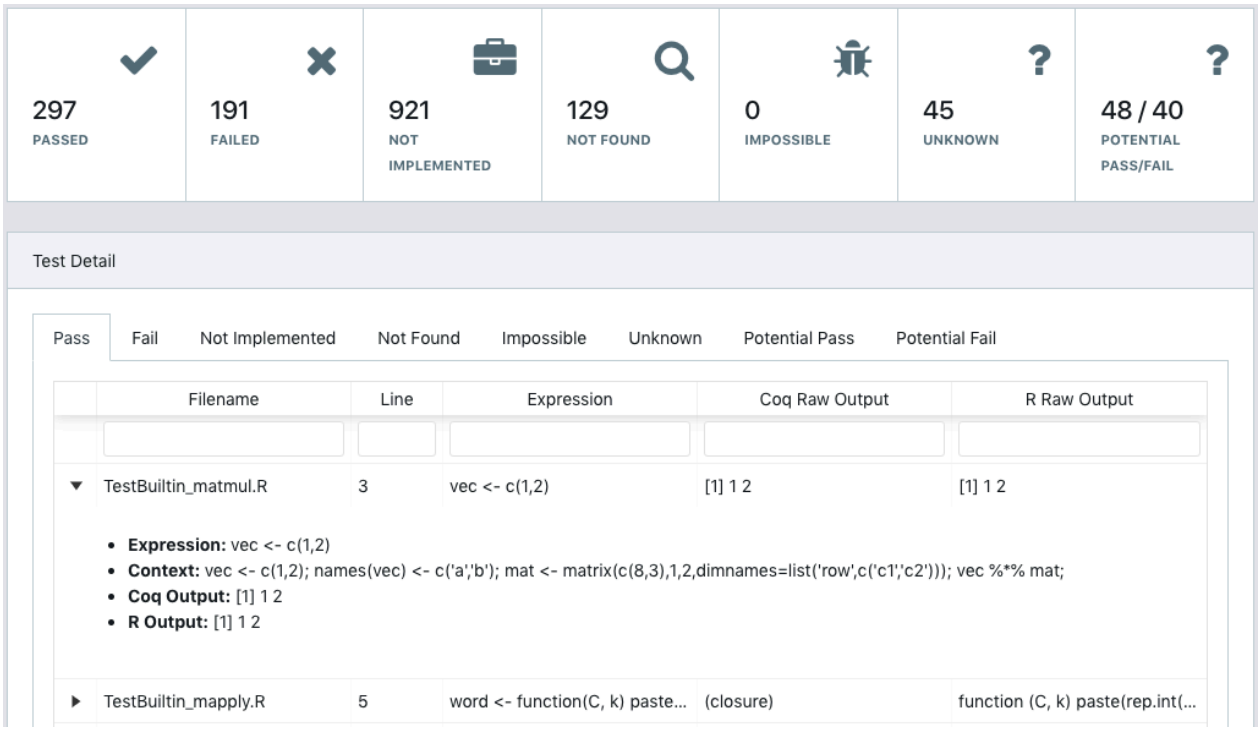

Figura 3.15: Sección que despliega información sobre los resultados de las pruebas y comparación. Arriba un resumen de los resultados y abajo tablas con información detallada para cada categoría.

que fue evaluada y los valores de salida para cada intérprete. Cada fila puede ser cliqueada, lo que produce que se expanda y muestre información adicional, como el contexto en el que se encuentra la expresión, así como permitir visualizar otros datos de manera más cómoda. Estas tablas también cuentan con entradas de texto, bajo el título de cada columna, con los que se pueden filtrar las filas desplegadas.

Por último, esta vista incluye dos gráficas con el mismo formato, como el que se ve en la figura 3.16. La primera representa las funcionalidades más comunes sin implementar en CoqR, mientras que la segunda indica las diez definiciones de funciones más comunes que faltan en el ambiente de ejecución de CoqR, según los casos de prueba. En el eje "x" se indica el nombre de la funcionalidad no implementada o de la función no encontrada, mientras que en el eje "y" se indica la cantidad de casos en los que falló CoqR debido a no tener esa funcionalidad implementada o en el ambiente. El propósito de ambas gráficas es permitir visualizar e identificar rápidamente los casos más comunes por los cuales falla la ejecución de CoqR, con el fin de tomar medidas más apropiadas al respecto y guiando los esfuerzos de desarrollo donde sea más apremiante.

Es muy importante recalcar la importancia y utilidad que tienen todos estos gráficos y reportes durante la evaluación e implementación de CoqR. Esto se puede apreciar claramente en la figura 3.16, recién mencionada, donde se puede ver que la funcionalidad `do_subset2` fue causante de más de 850 resultados fallidos, de la categoría *no implementado*, seguida en segundo lugar por `do_docall` con una cantidad apenas superior a los 50 casos. Gracias a visualizaciones simples y claras, se facilita enormemente el análisis de los resultados y, para

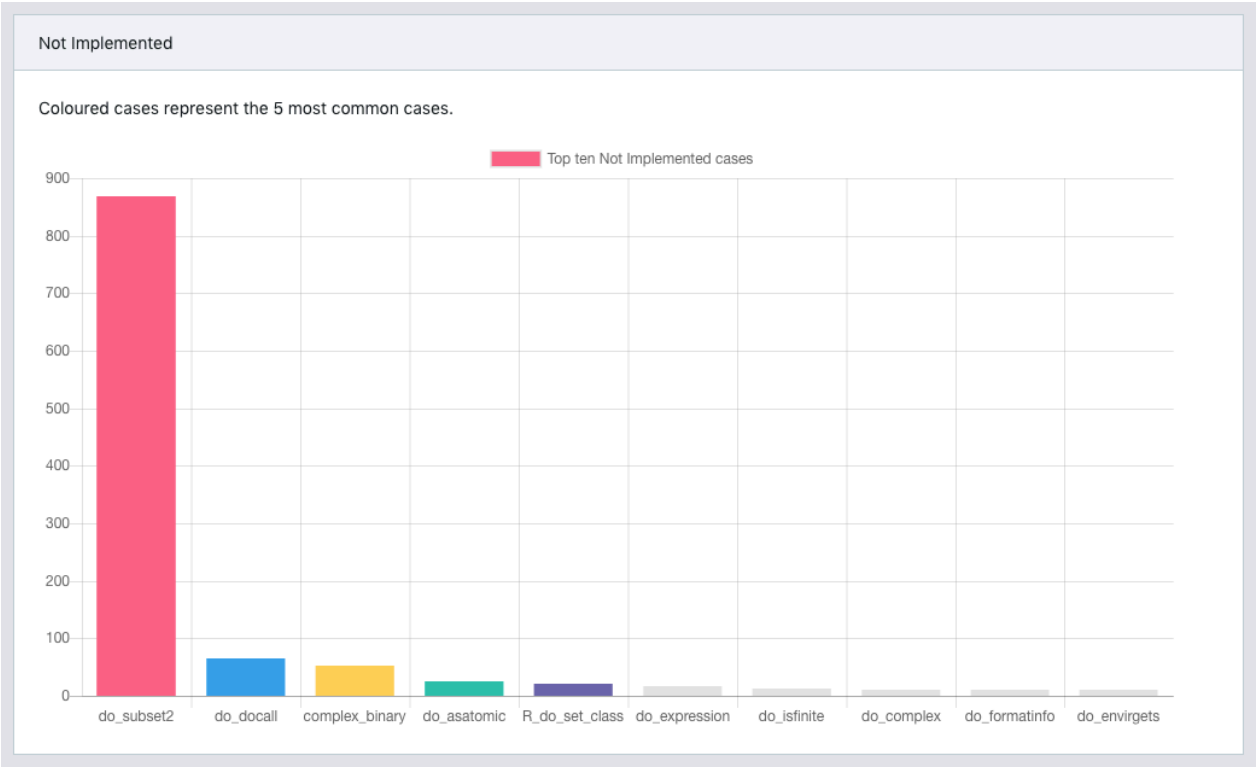

Figura 3.16: Gráfico mostrando los 10 casos más comunes de funcionalidades no implementadas.

este ejemplo, resulta evidente que lo más apropiado es enfocar los esfuerzos en implementar `do_subset2` antes que cualquier otra funcionalidad de la tabla de símbolos.

3.6. Resumen

En este capítulo se revisa el sistema desarrollado en esta memoria, cuyo propósito principal es brindar una segunda fuente de validación a la formalización del lenguaje de programación R que se lleva a cabo en el proyecto CoqR. Esta validación se basa en la ejecución de bancos de pruebas, con el intérprete de CoqR y con el intérprete de referencia de R, GNU R, para luego realizar una comparación de los valores de salida generados por ambos. De esta comparación se puede definir qué tan semejantes son las semánticas de GNU R y CoqR.

También se describe cómo el sistema implementado tiene un segundo propósito, el cual corresponde a colaborar en el proceso de desarrollo incremental, proveyendo herramientas que permitan visualizar y analizar los resultados obtenidos. A través del almacenamiento apropiado de los resultados y la generación de gráficas y tablas, es posible identificar las funcionalidades más importantes a ser implementadas en CoqR, efectivamente definiendo rutas de trabajo a seguir.

Para abordar estos aspectos, se describen los distintos módulos que componen el sistema desarrollado, iniciando con el módulo de validación, encargado de ejecutar las pruebas

y comparar los intérpretes. Se explica el proceso necesario para realizar esta tarea, el cual se divide en tres fases principales: interpretación de las expresiones, procesamiento de los valores de salida y comparación. Tras la comparación, se generan reportes que son almacenados en el módulo de datos y a los que, posteriormente, es posible acceder a través del módulo de visualizaciones. Gracias a esta separación de responsabilidades y una definición clara de las interfaces con las que se comunican, es posible trabajar en cada uno de manera casi completamente independiente, facilitando la extensión de funcionalidades, así como la implementación de gráficas y reportes.

Por supuesto, estas distintas herramientas no tienen sentido si es que no cumplen con su propósito, por lo que es necesario llevar a cabo una validación de estas. Por un lado, confirmar que permiten probar el comportamiento de CoqR con respecto a GNU R y, por otro, que sirven para mejorar el proceso general de desarrollo. Ambos aspectos son abordados en el capítulo siguiente.

Capítulo 4

Validación del sistema

Este capítulo versa sobre la validación del sistema en sí, para llevar a cabo la validación de CoqR, además de actuar como herramienta para el desarrollo incremental de éste. Se presentan los resultados finales obtenidos, gracias al sistema desarrollado, junto con vincularlos a los objetivos planteados.

Primeramente, se aborda el aspecto del desarrollo incremental y cómo el sistema elaborado en esta memoria permite guiar los esfuerzos hacia los aspectos más apremiantes. Se hace mención a una situación particular donde las herramientas y gráficas desarrolladas permitieron la identificación de un conjunto importante de definiciones faltantes en el ambiente de ejecución de CoqR, correspondiente a la librería base de GNU R, tras lo cual es posible obtener mejores y más claros resultados respecto a la equivalencia entre ambos intérpretes. A continuación, se revisa el objetivo de implementar funcionalidades de GNU R en CoqR, resumiendo aquellas que fueron implementadas durante esta memoria y cuya selección se realiza en base a los resultados obtenidos tras ejecutar las pruebas. Por último, se concluye con un análisis respecto al objetivo principal de esta memoria, relativo a la validación de CoqR. Para ello se hace un resumen y análisis de los resultados finales obtenidos a través del sistema y generados en base a los bancos de pruebas.

4.1. Guiando el proceso de desarrollo

En esta sección se analiza el sistema con respecto a su integración con el proceso de desarrollo de CoqR, cómo los distintos componentes apoyan al trabajo de implementación y cómo guían los esfuerzos hacia los aspectos más relevantes.

Antes que todo, vale mencionar que no parece evidente la forma de cuantificar el aporte o cómo validar las herramientas desarrolladas *a posteriori*, con respecto al objetivo de mejorar el proceso de desarrollo de CoqR. Esto se debe a que, como se indicó al comienzo del capítulo anterior, la metodología de trabajo iterativa e incremental que se adopta, conlleva una constante retroalimentación y validación de las distintas herramientas desarrolladas. A medida que se tiene mayor entendimiento de GNU R, de su funcionamiento y sutilezas, así como

de los resultados iniciales que se obtienen a través de las pruebas, los requisitos del sistema van cambiando. Esto conlleva a realizar modificaciones, de tal manera que efectivamente se satisfagan las demandas requeridas.

En este sentido, efectivamente los distintos componentes descritos en el capítulo anterior responden a las distintas necesidades, del equipo de desarrollo, que surgen durante el periodo de esta memoria y pueden considerarse como apoyo al proceso de trabajo. Algunos aspectos fundamentales sobre los cuales las herramientas tienen utilidad son el poder identificar los casos fallidos, poder ver en detalle las expresiones que fallan y los valores de salida generados por ambos intérpretes, permitiendo depurar el sistema. También el poder diferenciar los distintos casos de error, para saber sobre qué trabajar en una siguiente fase.

Aún así, no es claro si se valida con respecto a cuántas funcionalidades se implementaron gracias a las herramientas, o si por casos erróneos detectados u otros factores. Por ejemplo, durante el trabajo de memoria, se lograron detectar más de 250 casos imposibles, correspondientes a *bugs* en la implementación de CoqR. Este número de por sí indica que la herramienta sirve para el desarrollo del intérprete, permitiendo arreglar problemas de implementación, pero no es evidente dimensionar su impacto general. De esta manera, en lo que continúa de esta sección se describen distintos aspectos que el sistema desarrollado en esta memoria permite abordar apropiadamente, indicando ciertos momentos anecdóticos que permiten reforzar las ideas presentadas.

4.1.1. Identificando distintos tipos de resultados

Tener diferentes categorías para los resultados de las ejecuciones es crucial en un proyecto de la magnitud de CoqR, de manera de ser posible comprender la tarea por delante. En particular, poder diferenciar entre verdaderos resultados fallidos, es decir, casos donde el comportamiento de GNU R y CoqR es realmente diferente, y resultados fallidos debido a usos internos de `result_not_implemented` u otros debidos a definiciones faltantes en el ambiente de ejecución de CoqR.

Cabe recordar que una funcionalidad no implementada corresponde habitualmente a una entrada en la tabla de símbolos, descrita en la sección 2.4.4, que aún no ha sido traducida a CoqR. También es posible que corresponda a una sección del código, como una rama dentro de un `switch` grande, donde se decide posponer la implementación por algún motivo. Por otro lado, una definición faltante corresponde al uso de una función en R no definida en el ambiente durante la ejecución; esto típicamente corresponde a variables que no han sido definidas en código R y que no se encuentran en el ambiente de CoqR.

De esta manera, las gráficas mencionadas en la sección 3.5.2, permiten visualizar los casos más frecuentes de funciones faltantes o funcionalidades no implementadas. Ambas gráficas tienen el propósito de ayudar a identificar las oportunidades más inmediatas (*low-hanging fruit*) para implementar o agregar al intérprete, aportando el mayor progreso posible. Por ejemplo, en los inicios del proyecto, la librería base aparecía predominantemente en la lista de funciones faltantes, como se discute en secciones posteriores.

Figura 4.1: Casos no implementados obtenidos tras implementar `do_subset2` y `do_asatomic`.

Además, la gráfica de resultados no implementados se utiliza para decidir e implementar, de manera iterativa, cuáles funcionalidades de la tabla de símbolos se deben implementar primero, logrando llegar a las 128 funcionalidades implementadas actualmente.

La mejor forma de ver esto es a través de un ejemplo anecdótico. Al momento de ejecutar FastR2 se pudo apreciar, de los resultados y gráficas, que la próxima funcionalidad a implementar debía ser `do_subset2`. Esto, dado que los resultados tenían todos una forma similar a la mostrada en la figura 3.16, donde era evidente la diferencia de casos fallidos debido a que no estaba implementada esta funcionalidad. Por otro lado, para FastR1, los resultados indicaban que `do_asatomic` debía ser implementada.

Tras implementar ambas funcionalidades, se puede observar inmediatamente cómo cambian los resultados para los casos de prueba de FastR2. La figura 4.1 corresponde a los casos no implementados, para el mismo conjunto de pruebas cuyo resultado se indicó en el párrafo anterior. En primer lugar, la diferencia entre las distintas funcionalidades es menos pronunciada, pero más importante aún, aparecen funcionalidades como `do_identical` en primer lugar, la cual ni siquiera aparecía mencionada anteriormente. Esto se da como consecuencia de que CoqR es capaz de dar más pasos en la evaluación de las expresiones, antes de encontrarse con algo que arroja un resultado no implementado.

Así, efectivamente el sistema provee un mecanismo para identificar claramente los distintos tipos de resultados, en particular discernir los casos erróneos debido a código sin implementar o funciones faltantes. Si se desea visualizar el detalle de un resultado, por motivos de depuración por ejemplo, también se provee de visualizaciones específicas, como la indicada

en la figura 3.15.

4.1.2. Detección de librería base faltante

En esta sección se describe un caso particular, en donde se da cuenta cómo las herramientas desarrolladas permiten la detección de errores y oportunidades para trabajar en pos de una validación apropiada de CoqR con respecto a GNU R. Este caso corresponde a la detección de definiciones básicas faltantes en el ambiente de ejecución de CoqR, dada por la carencia de una librería escrita en código R.

La librería base consiste de alrededor de 160 archivos de código R, totalizando más de 19.000 líneas de código, los cuales son ejecutados por GNU R previo a cualquier otra expresión. Esta librería incluye definiciones de diversas funciones y objetos, tales como `mean`, `matrix` o `pi`.

Varias de estas funciones son envoltorios (*wrappers*) de otras funciones internas, que agregan chequeos de los valores de los argumentos o proveen valores por defecto. Un ejemplo es la función `eval`, que define valores por defecto a los argumentos utilizados por la función interna `eval`:

```
1 eval <- function(expr, envir = parent.frame(),
2     enclos = if(is.list(envir) || is.pairlist(envir))
3               parent.frame() else baseenv())
4     .Internal(eval(expr, envir, enclos))
```

Para poder llevar a cabo la mayoría de estas definiciones, no se requiere tener muchas funcionalidades implementadas: basta con que la palabra clave `function` y la asignación `<-` estén implementadas. Tomando el ejemplo de `eval` puede parecer que es necesario definir más funciones, como `parent.frame` o `is.list`, pero esto no es así, ya que estas funciones solo son evaluadas al momento que se aplique `eval` con argumentos concretos. Existen, sin embargo, casos particulares de definiciones donde sí es necesario implementar funciones adicionales, como en el archivo `constants.R` de la librería base, en la que se aprecia la definición `pi <- 4 * atan (1)`, la cual requiere que se implemente `atan` para poder definir el valor `pi`.

Por supuesto, ser capaz de ejecutar las definiciones e incluirlas al ambiente de ejecución de CoqR no significa que estas definiciones sean utilizables: la función interna llamada por la función de la librería puede que aún no haya sido implementada en CoqR, por lo que si una expresión la ejecuta, se obtendrá un resultado en la categoría de no implementado. Esta es la razón de porqué es tan crucial ser capaces de distinguir entre verdaderos resultados fallidos, donde hay discrepancia entre GNU R y CoqR, y los casos no implementados o de funciones faltantes. Es de notar que varias funciones pueden depender de la misma funcionalidad interna (no implementada), lo que da aún mayor importancia a los reportes y gráficas de los resultados no implementados más comunes.

De esta manera, el sistema desarrollado permite detectar estas falencias de CoqR y tomar las acciones apropiadas. En este caso, incluir una primera fase donde se ejecutan y se agregan

al ambiente las definiciones de la librería base, tras lo cual es posible ejecutar los bancos de pruebas y obtener resultados con información más apropiada, que muestren con mayor certitud el grado de equivalencia entre CoqR y GNU R.

A modo anecdótico, la detección de esta carencia de la librería base en el ambiente de ejecución de CoqR, se da tras ejecutar el banco de pruebas de GNU R, teniendo implementadas 105 funcionalidades de la tabla de símbolos, donde se obtienen 1.200 resultados fallidos. De estos 1.200 resultados, alrededor de 1.000 corresponden a casos donde faltan definiciones básicas.

Luego de extender CoqR para ser capaz de cargar este conjunto base de definiciones, el número total de resultados fallidos cae a 700, con apenas 300 debido a funciones faltantes, correspondientes a otras librerías básicas, predominantemente una librería de estadísticas. De manera similar a lo descrito en la sección anterior, también se puede apreciar un cambio en las gráficas de resultados no implementados.

4.2. Implementación de funcionalidades

En esta sección se describen las funcionalidades de la tabla de símbolos de GNU R, descrita en la sección 2.4.4, que se implementan en CoqR, como parte de los objetivos planteados en un comienzo de esta memoria. La selección de funcionalidades a implementar se basa en los resultados obtenidos tras ejecutar los bancos de pruebas, y evidenciados con los reportes y gráficos mencionados en 3.5.

En este trabajo, se implementan 16 nuevas funcionalidades, en particular: `do_matrix`, `do_subset2`, `do_subset2_dflt`, `do_subset3`, `do_asatomic`, `do_subassign`, `do_logic`, `do_logic2`, `do_substitute`, `do_is`, `do_for`, `do_asCharacterFactor`, `do_makevector`, `do_body`, `do_length` y `do_quote`.

La implementación de estas funcionalidades puede ser relativamente simple, como es el caso de `do_body` que, dada una función, permite obtener o redefinir su cuerpo. Esta funcionalidad requiere no más de 20 líneas de código, entre la definición de ésta y dos funciones auxiliares. La figura 4.2 muestra el código en C y en Coq necesaria para implementar esta funcionalidad. Vale notar, nuevamente, la similitud entre ambos códigos, donde una o dos líneas de código en CoqR corresponden a una de GNU R. Similar al caso descrito en la sección 2.4.3, se tienen líneas que corresponden directamente, como la línea 3 en C y en Coq donde se hace una llamada sin asignación, y otras, como la línea 4 en C que equivale a las líneas 4, 5 y 6 en Coq, donde se sigue el mismo patrón para acceder a la cabeza de la lista y donde se deben realizar asignaciones previo a poder evaluarlas en el `if`.

Existen algunos puntos interesantes a notar, como son la línea 6 en C y el bloque que comprende las líneas 9, 10 y 11 en C también. En el primer caso, se tiene el uso de la macro `RAISE_NAMED`, la cual se tradujo con una notación distinta en Coq, correspondiente a las líneas 8 y 9, pero que son fácilmente reconocibles como equivalentes. Para el segundo caso, se puede ver que corresponde a un mensaje de precaución que incluye GNU R al resultado

```
1  SEXP attribute_hidden do_body(SEXP call, SEXP op, SEXP args, SEXP rho)
2  {
3      checkArity(op, args);
4      if (TYPEOF(CAR(args)) == CLOSXP) {
5          SEXP b = BODY_EXPR(CAR(args));
6          RAISE_NAMED(b, NAMED(CAR(args)));
7          return b;
8      } else {
9          if(!(TYPEOF(CAR(args)) == BUILTINSXP ||
10              TYPEOF(CAR(args)) == SPECIALSXP))
11             warningcall(call, _("argument is not a function"));
12         return R_NilValue;
13     }
14 }
```

(a) Función original en C

```
1  Definition do_body S (call op args rho : SEXP) : result SEXP :=
2    add%stack "do_body" in
3      run%success Rf_checkArityCall globals runs S op args call using S in
4      read%list args_car, _, _ := args using S in
5      let%success args_car_type := TYPEOF S args_car using S in
6      ifb args_car_type = CloSxp then
7          let%success b := BODY_EXPR globals S args_car using S in
8          let%success args_car_named := NAMED S args_car using S in
9          set%named b := args_car_named using S in
10         result_success S b
11     else
12         (** A warning message has been left out **)
13         result_success S (R_NilValue : SEXP).
```

(b) Función traducida a Coq

Figura 4.2: Implementación de `do_body` en CoqR.

generado, indicando que el argumento entregado no es una función. En Coq se decide ignorar este mensaje, visible en la línea 12, dado que estos mensajes de precaución son simplemente notificaciones para el usuario, impresas en pantalla, pero no alteran los resultados generados.

Esto no siempre es el caso, como se puede apreciar para otras funcionalidades como `do_matrix` y `do_for`. En el primer caso se requieren casi 300 líneas de código, entre la función en sí más dos funciones auxiliares, mientras que en el segundo se requieren casi 150 líneas, lo que incluye 7 funciones adicionales. En casos como estos, donde se tienen funciones de gran envergadura, es interesante notar el uso que se le puede dar al constructor `result_not_implemented` para dejar secciones del código incompletas y poder trabajar sobre los aspectos que parecen más relevantes. La figura 4.3 muestra el código de la función `unimplemented_function` que simplemente retorna un resultado no implementado y que puede servir en secciones de código donde se desea dejar pendiente la implementación de una función. Agrega información, al mensaje de error, sobre la función que no se implementa, así como información a la traza de la pila de ejecución.

Para ilustrar el uso del constructor `result_not_implemented` y de la función recién mencionada, se puede apreciar la figura 4.4 que muestra un fragmento del código, en Coq, de

```
Definition unimplemented_function (A : Type) fname : result A :=
  add%stack fname in
    result_not_implemented ("Function not implemented: " ++ fname ++ ".").
```

Figura 4.3: Código de `unimplemented_function` escrita en Coq.

```
[...]
PROTECT(ans = allocMatrix(TYPEOF(vals), nr, nc));
if(lendat) {
    if (isVector(vals))
        copyMatrix(ans, vals, byrow);
    else
        copyListMatrix(ans, vals, byrow);
} else if (isVector(vals)) { /* fill with NAs */
[...]
```

(a) Fragmento original en C

```
[...]
let%success ans := allocMatrix globals runs S vals_type (Z.to_nat nr) (Z.to_nat
 ↪  nc) using S in
  run%success
  let%success vals_isVector := isVector S vals using S in
  ifb lendat <> 0 then
      if vals_isVector then
          copyMatrix S ans vals (Z.to_nat byrow)
      else
          unimplemented_function "copyListMatrix"
  else if vals_isVector then (* fill with NAs *)
[...]
```

(b) Fragmento traducido a Coq

Figura 4.4: Fragmento del código de `do_matrix`, mostrando su implementación parcial.

la implementación de `do_matrix`. La figura 4.4a muestra el código original en C, donde se realiza la llamada a la función `copyListMatrix`, en la línea número 7, mientras que en la figura 4.4b se puede ver que esta misma llamada, en la línea número 9, se ha reemplazado por la llamada a `unimplemented_function`. De esta forma, se puede acelerar el proceso de desarrollo, dejando pendientes ciertas secciones de código para las cuales se espera que su uso sea más bien acotado e implementarlas cuando sea estrictamente necesario.

4.3. Resultados finales

La figura 4.1 presenta el estado actual de ejecutar los cuatro bancos de pruebas descritos en la sección 3.2. En un procesador de 2 núcleos Intel Xeon 2.20GHz con 4 GB de RAM, los tiempos de ejecución varían entre 1.5 hrs. para GNU R a más de 12 hrs. para FastR2.

Suite	P	F	NI	NE	I	D	PP	PF
Corners	2.613	7	48	119	0	149	20	6
GNU R	243	31	739	723	1	27	0	0
FastR1	1.103	25	987	115	0	161	59	326
FastR2	2.411	1.128	6.888	493	0	1.914	297	343
Total	6.370	1.191	8.662	1.450	1	2.251	376	675

Número total de pruebas: 20.976

(P = Pasó, F = Falló, NI = No implementado, NE = No Encontrado, I = Imposible, D = Desconocido, PP = Potencialmente pasó, PF = Potencialmente falló)

Tabla 4.1: Resultados actuales al ejecutar los distintos bancos de pruebas.

En términos generales, CoqR cubre exitosamente el 30 % de las pruebas, tras implementar las funcionalidades descritas en la sección anterior. Para el caso de Corners, se tiene casi cobertura completa, mientras que para FastR2, integrado recientemente al proceso de desarrollo, se tiene un 18 % de casos exitosos, quedando atrás con respecto a FastR1, que alcanza un 40 % de casos exitosos.

Sin sorpresa, 50 % de los resultados negativos son, o bien "no implementado" o "no encontrado". Esto es de esperar, considerando que actualmente hay 128 funcionalidades implementadas de la tabla de símbolos, de un total de 700 entradas. Además, esto se debe a que la librería base se va incluyendo parcialmente e iterativamente, según lo requerido por los bancos de pruebas, permitiendo reducir los tiempos de ejecución.

En el caso de los resultados fallidos (5,7 %), a través de un análisis manual se observa que alrededor del 40 % se debe a discrepancias dadas por la precisión de números de punto flotante, como se mostró en la sección 3.3.4. Por el momento se está estudiando cómo eliminar estas diferencias, así como identificar las causas de los otros casos fallidos.

Por otro lado, se puede ver que 10,7 % de los casos son clasificados como "desconocido", mientras que 5 % corresponde a casos potencialmente válidos o fallidos. Como se explicó previamente, los resultados desconocidos se dan debido a las diferencias en cómo se imprimen ciertos valores entre CoqR y GNU R. En el estado actual del sistema, arreglar estas diferencias puede ser agotador, pero no desafiante, de un punto de vista conceptual.

Por último, cabe destacar que sólo hay un caso "imposible", correspondiente a un error en la implementación de CoqR. Los resultados indican que puede tratarse de la función `do_missing` de la tabla de funcionalidades. Este caso está actualmente en revisión.

4.4. Resumen

Este capítulo trata sobre la validación del sistema desarrollado en esta memoria y descrito en el capítulo anterior. La validación de este sistema se describe en términos de su utilidad como infraestructura para, a su vez, validar la formalización llevada a cabo en el proyecto CoqR. También se considera su utilidad con respecto a ser un sistema de apoyo para el

desarrollo iterativo del intérprete en Coq.

En cuanto a infraestructura de apoyo al trabajo, se describe cómo las herramientas y módulos del sistema son indispensables para guiar el proceso de desarrollo, permitiendo detectar tempranamente errores en la implementación de funcionalidades e identificando los aspectos más relevantes sobre los cuales trabajar. Se ilustra esto a través de la detección de funcionalidades faltantes en el ambiente de ejecución de CoqR, como lo es la librería base de R, y al uso que se le puede dar a los gráficos y reportes generados por el sistema.

Finaliza el capítulo con una evaluación de los resultados finales obtenidos, tras ejecutar los distintos bancos de pruebas, donde se puede apreciar que se alcanza una cobertura de un 30 %. Esta cantidad refleja que se tiene un conjunto considerable de funcionalidades del lenguaje, las cuales no son triviales de implementar. Además, se puede ver de lo descrito en las secciones de este capítulo, que tanto CoqR como el sistema desarrollado en esta memoria se complementan de tal manera que es posible continuar con un desarrollo iterativo e incremental, empujando este porcentaje aún más, siempre que se tenga poder ingenieril para hacerlo.

En base a estos distintos aspectos, es posible dar paso a un análisis y concluir respecto a los objetivos planteados inicialmente en esta memoria, sobre su capacidad de validar CoqR y colaborar en el proceso de desarrollo.

Capítulo 5

Conclusiones y trabajo a futuro

Respecto a los objetivos planteados inicialmente, se considera que el trabajo desarrollado los satisface plenamente. Por un lado, las herramientas implementadas permiten llevar a cabo la comparación y verificación del comportamiento entre CoqR y GNU R, permitiendo la ejecución de los bancos de pruebas, la comparación de resultados y la generación de reportes con información detallada y pertinente al respecto. La integración de estas herramientas al proceso de desarrollo resulta fundamental, permitiendo la temprana detección de errores en la implementación de CoqR que, de otra manera, sería complicada.

En esta misma línea, la metodología iterativa que se adopta para desarrollar estas herramientas permite una integración fluida y transparente con el proyecto de CoqR. Además, posibilita una colaboración y potencia el impacto en el proceso de desarrollo general, al permitir definir sendas de trabajo a seguir, en adición a corregir problemas que se presenten durante las fases de implementación. Actualmente esta metodología es adoptada por los participantes del proyecto de CoqR para avanzar y colaborar armoniosamente entre ellos.

Por otro lado, el desarrollo de las interfaces gráficas resulta ser primordial al momento de analizar los resultados obtenidos y, nuevamente, colaborar a todo el proceso de desarrollo. Si se recuerda lo descrito en la sección 2.4.1, inicialmente se tiene únicamente la interfaz de la consola de trabajo para ejecutar expresiones y visualizar los resultados, pero esto es claramente insuficiente para comprender el alcance de los resultados obtenidos y no permite un análisis apropiado. Una primera aproximación, como se muestra en la sección 3.3.1, no es suficiente, lo que motiva a la implementación de un módulo dedicado exclusivamente a proveer visualizaciones claras y fáciles de manejar. Con gráficas simples y directas se pueden detectar las próximas funcionalidades a implementar, de manera fácil y rápida, permitiendo dirigir correctamente los esfuerzos de desarrollo.

Por supuesto, es posible continuar el trabajo, desde mejorar las funcionalidades existentes a agregar nuevas. A continuación se detallan posibles áreas de trabajo a futuro.

Extender a nuevos resultados

Debido a que GNU R tiene muchas formas de imprimir resultados, así como opciones que alteran el formato de impresión, es necesario tener un mecanismo capaz de manejar estas sutilezas, que escale y sea de fácil uso. Además, se debe considerar que CoqR también posee su propia forma de imprimir resultados.

Una posible línea de trabajo aborda este aspecto desde CoqR, modificando el código en OCaml que se encarga de la impresión, descrito en la sección 2.4. De esta forma, es posible modificar el formato de las cadenas de caracteres generadas por CoqR para que se asemejen más a las generadas por GNU R, permitiendo que se utilice un mismo sistema para procesar los resultados de ambos intérpretes, a diferencia de lo que se realiza actualmente, donde se tienen procesadores distintos para cada uno.

Por otro lado, inevitablemente se debe trabajar sobre cómo procesar los resultados de ambos intérpretes. Una opción es crear interfaces más cercanas a los intérpretes, de tal forma de poder manejar los resultados generados más directamente y no a través de sus salidas como cadenas de caracteres, lo que permitiría evitar las sutilezas dadas por los formatos particulares que puedan tomar estas cadenas. Sin embargo, con esta opción se debe tener cuidado con que se pierdan resultados generados por efectos secundarios y que sea deseables capturar, o bien, que sea importante para GNU R las opciones y formatos que se le dan a los valores de salida. Para el caso de GNU R, es posible utilizar `rpy2`[1], una librería escrita en Python que embebe GNU R al proceso donde ésta sea importada y que se comunica directamente con el intérprete en C. Esto permite menores tiempos de ejecución y que se trabaje sobre las estructuras generadas por GNU R de forma más transparente. Es importante, nuevamente, tener cuidado de capturar todos los resultados pertinentes. Mientras que para CoqR, significaría un trabajo más profundo, que involucraría desarrollar nuevas herramientas e interfaces que se conecten con el intérprete extraído a OCaml, mencionado en la sección 2.4.1.

Una segunda opción concierne a los procesadores de resultados, así como a los objetos de tipo `Result` descritos en la sección 3.3.3, sobre cómo se definen y cómo se comparan. El problema actual que estos presentan es que, para agregar nuevos resultados es necesario modificar código existente, lo cual afecta la extensibilidad del sistema. Una opción a considerar es que el usuario defina archivos de configuración[2], definiendo los distintos procesadores y sus correspondientes expresiones regulares. En este mismo archivo se podrían definir los distintos tipos de resultados, así como las formas de compararse entre ellos. A partir de estos archivos de configuración, el sistema podría generar las clases y objetos pertinentes, lo que permitiría que el sistema sea más extensible, sin intromisión en el código existente y que sea aplicable a otros proyectos de características similares.

[1] https://rpy2.bitbucket.io/

[2] En formato JSON, por ejemplo

Reducir tiempos de ejecución

Uno de los principales problemas con la ejecución de los bancos de pruebas y la comparación de resultados, es el tiempo que demora en realizar la fase de interpretación, que puede llegar a ser de más de 10 horas. A continuación se describen ideas de cómo atacar esta problemática.

Primero que todo, es importante mencionar que mejorar el tiempo de ejecución de GNU R no tiene tanta relevancia como el de CoqR, considerando que el cuello de botella se encuentra en este último (entre 10 y 20 veces más lento). De todas maneras, una posible forma de mejorar los tiempos de ejecución para GNU R es utilizando la librería `rpy2`, mencionada en la subsección anterior, ya que se comunica más directamente con R.

En cuanto a CoqR, una propuesta es dividir la ejecución de los archivos a distintos hilos. Actualmente el proceso es muy secuencial; primero se interpreta con GNU R todos los archivos, luego se interpreta con CoqR todos los archivos, en seguida se procesan los resultados y finalmente se comparan. La interpretación podría hacerse por secciones, interpretando grupos de archivos en distintos hilos de ejecución y recolectando los resultados finales en un único reporte. Una posible herramienta a utilizar es Celery[3], que permite encolar y ejecutar tareas en paralelo.

Extender la aplicación web

Aún cuando la herramienta actual provee vistas y herramientas que permiten el desarrollo incremental de CoqR, es posible imaginar nuevos elementos que implementar. En general, esta área es un campo abierto, pudiendo haber muchos posibles cambios, por lo que aquí simplemente se mencionan algunas ideas.

Una posibilidad incluye agregar un área que permita ejecutar, en la aplicación web, el intérprete de CoqR, de manera de probar distintas expresiones y chequear ciertos resultados obtenidos. En la misma línea, podría habilitarse una interfaz que permita lanzar ejecuciones de los bancos de pruebas y hacer un seguimiento *online* de la ejecución; visualizar en qué fase se encuentra, archivos evaluados o incluso resultados parciales. Estas opciones pueden no ser tan fáciles *a priori*, ya que pueden requerir modificar el sistema desarrollado para que se pueda consultar en qué estado de su ejecución se encuentra o, bien, que periódicamente envíe señales de su estado. Además, respecto a ejecutar expresiones en CoqR, habría que evaluar si éstas se ejecutan en el servidor o del lado del cliente. Para el primer caso se deben habilitar puntos de acceso donde enviar las expresiones y evaluar si los tiempos de ejecución son apropiados, en términos de usabilidad, considerando la comunicación con el servidor y cuánto demore CoqR en evaluar las expresiones. Por otro lado, si se ejecutan del lado del cliente, se debe evaluar cómo enviar el intérprete ejecutable para que pueda ser utilizado directamente en el navegador.

Otra posibilidad es integrar una vista que permita hacer comparaciones entre distintas

[3]http://www.celeryproject.org/

ejecuciones de un banco de pruebas. Esta vista permitiría evaluar si, por ejemplo, de una ejecución a la siguiente hubo resultados que cambiaron de estado, pasando de ser resultados válidos a resultados fallidos, entre otras cosas. Esto apoyaría el desarrollo, al poder visualizar cambios inesperados en los resultados, que podrían indicar errores en la implementación.

Integración continua

Actualmente se tiene Jenkins[4] corriendo en el servidor Quetrahue del DCC, pero no se está utilizando. Jenkins es un programa que permite llevar a cabo integración continua, permitiendo automatizar tareas como, por ejemplo, ejecutar baterías de pruebas, compilar sistemas, construir paquetes listos para ser usados, entre otras cosas. El principio de integración continua, resumidamente, plantea que el sistema en el que se trabaja sea construido y probado a menudo, de manera de detectar errores tempranamente. En esta línea, Jenkins permite conectarse a repositorios en Github, de tal manera que se puedan ejecutar tareas en base a *commits* enviados al repositorio asociado y, a la vez, enviar datos sobre la ejecución de estas tareas de vuelta a Github, quien puede desplegar esta información en su página.

Una de las razones por las que no se está utilizando la aplicación de Jenkins corriendo en el DCC, es que el tiempo de ejecución de los bancos de pruebas es muy grande, de manera que llevar a cabo chequeo continuo y automático para cada *commit* realizado no es realista. En general, resulta más cómodo y apropiado realizar estos chequeos de forma manual.

Otra de las razones es que dado el desarrollo iterativo que se adopta, varios puntos no quedan completamente definidos, por ejemplo, las secciones de la librería base que incluir, qué bancos de pruebas ejecutar, etc. Este factor, sumado a que se debe ejecutar manualmente, conllevan a que se prefiera simplemente proceder de otra manera, sin utilizar Jenkins.

Es posible retomar este aspecto, ya sea considerando otras herramientas de integración continua o continuar con lo ya planteado, corrigiendo los aspectos deficientes. En particular, un aspecto importante puede ser el de corregir la conexión existente con Github[5], en particular los *webhooks*[6], a través de los cuales se envían notificaciones hacia el sistema que corre en el servidor del DCC.

Desarrollo de aplicación de escritorio

Una última línea de desarrollo puede ser la de incorporar una aplicación de escritorio, con una interfaz más amigable que la que provee una consola. Esta interfaz podría imitar a la aplicación web ya implementada, o permitir seleccionar las distintas alternativas actuales del sistema, como decidir si enviar los resultados al servidor, generar archivos de depuración, entre otros.

En cuanto a las interfaces, podría permitir un análisis más directo de los resultados ob-

[4]https://jenkins.io/
[5]https://github.com/
[6]https://developer.github.com/webhooks/

tenidos, sin necesidad de enviar al servidor en caso que sean pruebas simples. Esto va de la mano con incluir una conexión a una base de datos local donde poder almacenar los resultados obtenidos, para lo cual existen múltiples librerías en Python. También, podría incluir vistas que indiquen mejor el estado actual de ejecución de la herramienta, indicando tiempos estimados de término, archivos ya ejecutados, etc.

Una posible opción, considerando que ya existe una aplicación web, es la de utilizar un ambiente de desarrollo como Electron[7], el cual permite implementar aplicaciones de escritorio, utilizando tecnología web. Esto permitiría trasladar lo ya implementado de manera rápida y correcta, pero además acceder al sistema, pudiendo incorporar CoqR y el sistema de validación desarrollado a su funcionamiento.

[7]https://electronjs.org/

Bibliografía

[1] Bodin, Martin: *A Coq Formalisation of a Core of R*. En *CoqPl*, 2018.

[2] Bodin, Martin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt y Gareth Smith: *A Trusted Mechanised JavaScript Specification*. En *POPL '14, USA*, páginas 87–100, 2014. `http://doi.acm.org/10.1145/2535838.2535876`.

[3] Bodin, Martin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudinien, Alan Schmitt y Gareth Smith: *JSCert: Certified JavaScript*, 2012. `http://jscert.org/`.

[4] Burns, Patrick: *The R Inferno*. 2011.

[5] Coquand, Thierry, Gérard Huet, Christine Paulin *y cols.*: *the Coq Proof Assistant*, 1984. `https://coq.inria.fr/`.

[6] Ghibellini, Filippo: *Dynamic test generation for R packages*. Bachelor's Thesis, 2017.

[7] Ihaka, Ross y Robert Gentleman: *R: a Language for Data Analysis and Graphics*. Journal of Computational and Graphical Statistics, 5(3):299–314, 1996.

[8] Kalibera, Tomas, Petr Maj, Floréal Morandat y Jan Vitek: *A Fast Abstract Syntax Tree Interpreter for R*. En *VEE '14, USA, 2014*, páginas 89–102, 2014. `http://doi.acm.org/10.1145/2576195.2576205`.

[9] Maj, Petr, Tomas Kalibera y Jan Vitek: *TestR: R Language Test Driven Specification*. En *The R User Conference, UseR!*, volumen 10, página 149, 2013.

[10] R Core Team: *The Comprehensive R Archive Network*. `https://cran.r-project.org/`, visitado el 2017.

[11] R Core Team: *CRAN Contributed Packages*. `https://cran.r-project.org/web/packages/`, visitado el 2018.

[12] R Core Team: *CRAN Task Views*. `https://cran.r-project.org/web/views/`, visitado el 2018.

[13] R Core Team: *R: A Language and Environment for Statistical Computing*. R Foundation

for Statistical Computing, Vienna, Austria, 2015. https://www.R-project.org/.

[14] R Core Team: *R Language Definition*. R Foundation for Statistical Computing, Vienna, Austria, 2018. https://cran.r-project.org/doc/manuals/r-release/R-lang.pdf.

[15] Tsegelskyi, Roman y Jan Vitek: *TestR: Generating Unit Tests for R internals*. En *The R User Conference, UseR!*, 2014.

[16] Wadler, Philip: *Comprehending Monads*. Mathematical Structures in Computer Science, 2(4):461–493, 1992. https://doi.org/10.1017/S0960129500001560.

[17] Wadler, Philip: *Monads for Functional Programming*. En *Advanced Functional Programming*, páginas 24–52, 1995. https://doi.org/10.1007/3-540-59451-5_2.