UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA

Estudio del lenguaje funcional PureScript

Informe de Proyecto de Grado presentado por

Martín Feldman y Santiago Puppo

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Alberto Pardo
Marcos Viera, Juan Pablo Gracia

Montevideo, 18 de octubre de 2024

Estudio del lenguaje funcional PureScript por Martín Feldman y Santiago Puppo tiene licencia CC Atribución 4.0.

Agradecimientos

Agradecemos a todas las personas que nos dieron una mano y nos alentaron mientras llevábamos adelante este proyecto. En especial, a nuestras familias, amigos y parejas por su comprensión y apoyo. A las empresas en cuales trabajamos (Streaver y Qualabs) por la flexibilidad y facilidades brindadas durante este tiempo. También agradecemos a Marcos, Alberto y Juan Pablo, nuestros tutores, por estar siempre a nuestra disposición ante cualquier consulta, la buena onda y las horas que han dedicado a este proyecto. Finalmente, un agradecimiento a la Universidad de la República y a todos los docentes que tuvimos en estos años por darnos la oportunidad de aprender y llegar a recibirnos. Gracias a todos ellos por lo aprendido y por ayudarnos a cumplir este objetivo.

Resumen

Este proyecto de grado tiene como objetivo realizar un estudio profundo sobre el lenguaje funcional PureScript, un lenguaje de programación puramente funcional, fuertemente tipado, que compila a JavaScript y facilita la interoperabilidad con dicho lenguaje a través de una Interfaz Foránea (FFI). El enfoque del proyecto radica en analizar las características distintivas de PureScript, comparándolo con lenguajes relacionados como Haskell, JavaScript y TypeScript, evaluando tanto sus ventajas como sus desventajas en el contexto del desarrollo web.

Se ha realizado un análisis conceptual detallado de PureScript, explorando sus tipos de datos, módulos, sistema de tipos, y características clave como la inmutabilidad, la evaluación estricta y la interoperabilidad con JavaScript. Además, se implementaron tres casos de estudio para evaluar el uso práctico del lenguaje. El primer caso consistió en la creación de un servicio web que interactúa con una base de datos, el segundo abordó la implementación de un CRDT (*Conflict-Free Replicated Data Type*) y el tercero consistió en desarrollar una aplicación frontend utilizando React y PeerJS mediante la interfaz foránea de PureScript. Estos casos permitieron explorar el uso práctico de PureScript y su integración con tecnologías ampliamente utilizadas en la web moderna.

Los resultados obtenidos indican que PureScript presenta grandes beneficios en cuanto a seguridad y mantenibilidad del código, principalmente debido a su tipado estático fuerte y su capacidad de aprovechar bibliotecas de JavaScript. Sin embargo, también se identificaron desafíos, especialmente la reducida comunidad y la escasez de recursos los cuales impactan la experiencia de desarrollo. PureScript se presenta como una opción robusta para desarrolladores que buscan aplicar los principios de la programación funcional en el desarrollo web, combinando el poder del tipado estático y la pureza funcional con la flexibilidad del ecosistema JavaScript.

Palabras clave: Programación funcional, PureScript, Desarrollo Web, Haskell, JavaScript

Índice general

1. Introducción **1**

2. Conceptos Fundamentales en Programación Funcional y Desarrollo Web **3**
2.1. Introducción a la programación funcional 3
2.1.1. Contexto histórico y evolución 3
2.1.2. Conceptos fundamentales de la programación funcional . 4
2.1.3. Ventajas y desafíos 8
2.2. JavaScript 9

3. Análisis conceptual del lenguaje PureScript **15**
3.1. Tipos de datos básicos 16
3.1.1. Registros 16
3.1.2. Arreglos 17
3.1.3. Funciones 17
3.2. Tipos de datos y Clases 18
3.3. Trabajar con registros 19
3.3.1. Trabajo con Registros a nivel de tipos 21
3.4. Módulos 22
3.4.1. Manejo de Efectos Secundarios - Mónadas *Effect* y *Aff* . . 22
3.5. Interoperabilidad con JavaScript - Interfaz Foránea 24
3.6. Compilación a JavaScript 26
3.6.1. Extensiones de compilador 28
3.7. Evaluación Perezosa 28
3.8. Comparación de PureScript con TypeScript 29

4. Análisis Práctico del lenguaje PureScript - Casos de Estudio **35**
4.1. Caso de estudio 1 - REST *Endpoint* y Base de datos 36
4.1.1. Módulos implementados 37
4.1.2. Proceso de desarrollo 40
4.1.3. Conclusiones y aprendizajes 42
4.2. Caso de estudio 2 - CRDT 43
4.2.1. Módulos implementados 45
4.2.2. Conclusiones y aprendizajes 48

4.3. Caso de estudio 3 - FFI PeerJs y *Frontend* basado en React . . . 48
4.3.1. Marco Teórico . . . 49
4.3.2. Diseño . . . 50
4.3.3. Módulos implementados . . . 52
4.3.4. Conclusiones y Aprendizajes . . . 63

5. Conclusiones **65**
5.1. Dificultades encontradas . . . 65
5.2. Resumen . . . 66
5.2.1. Perspectiva desarrollador JavaScript . . . 67
5.2.2. Perspectiva desarrollador Haskell . . . 68
5.3. Reflexiones finales . . . 69
5.4. Posibles Extensiones al Trabajo . . . 71

Referencias **73**

A. Caso de estudio 0 - *Getting Started* **77**
A.1. Creación y configuración de un proyecto . . . 77
A.2. Instalación de paquetes . . . 78
A.3. Uso del intérprete . . . 78
A.4. Ejemplo de aplicación . . . 78
A.5. Configuración avanzada - Recomendaciones . . . 79

Capítulo 1

Introducción

La ingeniería de software y el desarrollo web han evolucionado significativamente durante las últimas décadas, lo cual se hace evidente al analizar la cantidad y diversidad de los lenguajes de programación creados en ese tiempo. Esto ha sido impulsado por la creciente necesidad de soluciones de software más eficientes, robustas y mantenibles. Entre este conjunto de lenguajes creados se encuentra PureScript [1], un lenguaje puramente funcional y fuertemente tipado, cuya sintaxis está inspirada en Haskell [2], diseñado para interactuar con JavaScript. Este tiene varias características interesantes que lo hacen destacar entre otros lenguajes de programación funcional. La más importante es que compila a código JavaScript y que provee una Interfaz Foránea (*Foreign Function Interface*, FFI), la cual permite interactuar con dicho lenguaje. Ya que la mayoría de las páginas web funcionan con JavaScript, esto permite llevar el poder de la programación funcional al dominio web, además de poder hacer uso de un inmenso número de bibliotecas desarrolladas en dicho lenguaje.

Desde hace algunos años se observan bibliotecas que utilizan técnicas de programación funcional en el mundo JavaScript (Phil Freeman, 2023), tales como Immutable.js, RxJS y Underscore.js. No obstante, esto presenta ciertas desventajas debido a que JavaScript no tiene tipado estático y carece de formas poderosas de abstracción.

Phil Freeman, el autor de PureScript, destaca que comenzó a trabajar en PureScript en respuesta a su experiencia trabajando con JavaScript. Debido a la creciente popularidad de la programación funcional en JavaScript y la falta de un ambiente apto para desarrollo de aplicaciones de gran escala, el objetivo de PureScript es llevar la potencia de la programación funcional y de los lenguajes fuertemente tipados al ámbito de desarrollo en JavaScript.

PureScript es un lenguaje que se desarrolló con el objetivo de mantener la semántica de JavaScript y, al mismo tiempo, aprovechar muchas de las características de Haskell, como su sintaxis y sistema de tipos. PureScript busca ofrecer

[1] https://www.purescript.org/
[2] https://www.haskell.org/

un balance entre el poder de la programación puramente funcional y el estilo rápido y fácil de programar de JavaScript.

El objetivo principal de este proyecto de grado es investigar, analizar y aprender acerca de PureScript. Se buscará identificar las características distintivas del lenguaje, con un enfoque particular en su interoperabilidad con JavaScript. Además, se realizará una comparación detallada con Haskell, un lenguaje funcional ampliamente utilizado, para resaltar las similitudes y diferencias. Asimismo, se analizará la relación de PureScript con TypeScript [3], otro lenguaje relacionado con JavaScript, que intenta solucionar algunos de los problemas que surgen del tipado dinámico de JavaScript. Se espera lograr un entendimiento de los fundamentos del lenguaje y entender su pragmática mediante casos de uso.

Estructura del documento

El documento está estructurado en 5 capítulos. En el Capítulo 2 se presentan algunos de los conceptos fundamentales para entender el trabajo desarrollado donde se exponen una visión general de la programación funcional, del lenguaje JavaScript y de su importancia en el mundo del desarrollo web. En el Capítulo 3 se realizará un análisis conceptual de PureScript explorando algunas de sus características, comparándolas ocasionalmente con las de Haskell o JavaScript. Luego, en el Capítulo 4 se llevará a cabo una serie de casos de estudio para realizar un análisis práctico del lenguaje, expandiendo en las características exploradas en el capítulo anterior. Finalmente, en el Capítulo 5 se presentan las conclusiones del proyecto de grado.

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

Capítulo 2

Conceptos Fundamentales en Programación Funcional y Desarrollo Web

2.1. Introducción a la programación funcional

La programación funcional es un paradigma de programación con un enfoque que se centra en la evaluación de funciones y expresiones, con un fuerte énfasis en la inmutabilidad de los datos. El código se escribe pensando en qué hacer en lugar de cómo hacerlo, lo que la categoriza bajo el paradigma más amplio de la programación declarativa.

2.1.1. Contexto histórico y evolución

El paradigma funcional tiene sus raíces en el cálculo lambda, introducido por Alonzo Church en la década de 1930 como un sistema formal para caracterizar la noción de función computable (Barendregt, 1984). El cálculo lambda proporciona una sólida base teórica matemática la cual es fundamental para dicho paradigma.

A finales de la década de 1950 John McCarthy creó Lisp (McCarthy, 1960), el primer lenguaje de programación enfocado en el paradigma funcional, dando una gran visibilidad al mismo. En su diseño se reflejan los principios del cálculo lambda e incluye varios de los conceptos fundamentales de la programación funcional de la actualidad, como por ejemplo el concepto de función de alto orden.

Con el paso del tiempo, surgieron lenguajes que alcanzaron una gran popularidad, aunque no todos siguen siendo utilizados en la actualidad. Uno de ellos fue ML, lanzado en la década de 1970 (Milner, Tofte, y Harper, 1990), que fue pionero en la introducción de conceptos como el polimorfismo (Milner, 1978) y la inferencia de tipos.

Otro lenguaje que tuvo un gran impacto académico e influencia fue Haskell, desarrollado en 1990 (Thompson, 2011). Es un lenguaje fuertemente tipado, con evaluación perezosa, con tipado estático y polimorfismo. El mismo surgió por la necesidad de la comunidad de programación funcional de tener un lenguaje común. En ese momento coexistían muchos lenguajes de programación funcional con características similares, como la de ser funcionalmente puros y no estrictos, y se creía que esta diversidad dificultaba el uso más extendido de este tipo de lenguajes. Fue por esto que en el año 1987 se decidió formar un comité con el objetivo de desarrollar Haskell. Entre los objetivos del lenguaje se buscó que fuera adecuado para la enseñanza, la investigación y las aplicaciones, incluyendo el desarrollo de sistemas de gran porte.

Haskell se inspiró en Miranda (Turner, 1985), que dentro del conjunto de lenguajes funcionales, era el más maduro, mejor diseñado y más difundido; razones más que suficientes para que el diseño de Haskell estuviera fuertemente influido por él. Con el paso del tiempo, Haskell fue desplazando en gran medida a Miranda, debido principalmente a su naturaleza de libre acceso y distribución, contrario a Miranda, que estaba bajo licencia.

En la actualidad, se destacan varios lenguajes de programación funcional. Por ejemplo Clojure (2007) [1] y Scheme (1975) [2], ambos de la familia Lisp. Clojure destaca por su interoperabilidad con Java y C#. Racket [3], derivado de Scheme, fomenta la creación de lenguajes específicos de dominio (*Domain Specific Languages*, DSL). Elixir (2011) [4], basado en Erlang [5], es reconocido por su escalabilidad y manejo de concurrencia. Dentro del ecosistema de .NET, está F# (2005) [6] que combina el paradigma de la programación funcional con el paradigma de programación orientada a objetos, similar a Scala (2003) [7] y OCaml (1996) [8] que también integran estos paradigmas, siendo OCaml parte de la familia ML y caracterizándose por su evaluación estricta. Elm (2012) [9] destaca en el desarrollo web *frontend* por su arquitectura que reduce los errores en tiempo de ejecución y mejora la experiencia de desarrollo.

2.1.2. Conceptos fundamentales de la programación funcional

Las siguientes definiciones son tomadas de (Mena, 2019)

[1] https://clojure.org/
[2] https://www.scheme.org/
[3] https://racket-lang.org/
[4] https://elixir-lang.org/
[5] https://www.erlang.org/
[6] https://fsharp.org/
[7] https://www.scala-lang.org/
[8] https://ocaml.org/about
[9] https://elm-lang.org/

Funciones de Alto Orden

El concepto de Funciones de Alto Orden (*Higher-Order Functions*) es un concepto que refiere a funciones que pueden tomar como argumentos a otras funciones e incluso pueden retornar una función. En los lenguajes que soportan esta característica las funciones pasan a ser ciudadanas de primera clase (*First-Class Citizens*), dado que al igual que los valores de los restantes tipos, estas pueden ser argumentos o resultados de otras funciones. La habilidad de manipular funciones de esta manera es un pilar fundamental del paradigma funcional.

Efectos Secundarios, Funciones Puras y Transparencia Referencial

En la programación imperativa los programas están formados por una serie de instrucciones que realizan cambios a un estado. A estos cambios se los denominan Efectos Secundarios (*Side Effects*).

Se denomina Función Pura (*Pure Function*) a una función que dados los mismos argumentos, siempre devuelve el mismo resultado y no produce efectos secundarios. Esta definición corresponde con la definición matemática de función, en contraste a las funciones impuras donde no se da esta correspondencia. Esto es relevante de destacar, ya que en la mayoría de los lenguajes de programación dentro de los paradigmas no declarativos se utilizan funciones impuras, lo que puede generar confusión. En varios lenguajes funcionales puros, como es el caso de Haskell, se suele distinguir las expresiones puras de las impuras.

Una de las ventajas de trabajar con funciones puras es que el programador puede asegurarse que utilizando los mismos valores como argumentos, la evaluación de la función siempre va a terminar en el mismo resultado. A este concepto se le llama Transparencia Referencial (*Referential Transparency*) y permite entre otras cosas, demostrar de forma simple la correctitud de código o realizar optimizaciones a su ejecución.

Sistemas de Tipos

Un Sistema de Tipos (*Type System*) es un mecanismo que categoriza los valores que pueden surgir en tiempo de ejecución, a un determinado tipo. Normalmente se restringe las acciones posibles que se pueden realizar sobre los valores de un cierto tipo ayudando a prevenir errores. Entre las funciones clave de un sistema de tipos se encuentran las siguientes:

- **Verificación de tipos** Asegura que las operaciones se realicen entre tipos compatibles. Por ejemplo, evitar sumar un número con un *string*

- **Inferencia de tipos** La Inferencia de Tipos se refiere a la capacidad de un lenguaje de programación de deducir automáticamente el tipo de una variable o función basándose en el contexto. Esto reduce la necesidad de explicitar los tipos, produciendo un código menos verboso.

- **Polimorfismo Paramétrico** El Polimorfismo Paramétrico (*Parametric Polymorphism*) es una propiedad que tienen las funciones y tipos de datos genéricos. Permite definir variables de tipos genéricas para poder reutilizar una función en distintos contextos.
- **Seguridad de tipos** Ayuda a evitar errores en tiempo de ejecución relacionados con operaciones de tipos incompatibles, garantizando que las operaciones sean seguras en cuanto a tipos.
- **Abstracción de datos** La Abstracción de datos (*Data abstraction*) ofrece mecanismos para definir tipos personalizados y complejos a partir de tipos más simples, facilitando el modelado de estructuras de datos y la encapsulación de detalles de implementación.

Tipos de Datos Algebraicos

Los Tipos de Datos Algebraicos (*Algebraic Data Types*) son un mecanismo que permite definir estructuras de datos complejas de una manera concisa.

Un clásico ejemplo es el de las listas `data List a = Nil | Cons a (List a)`. Esta definición provee dos constructores, `Cons` y `Nil`, que permiten construir un valor de tipo `List` paramétrico en un tipo `a` que puede ser inferido en base al valor que se le pasa al constructor o definido de manera explícita.

Clases de Tipos

Las clases de tipos (*Type Classes*) definen un comportamiento que debe ser implementado por cualquier tipo que instancie a dicha clase. Esto es similar a interfaces en otros lenguajes de programación orientados a objetos, pero con un enfoque en funciones y tipos.

Por ejemplo la clase `Show a`, que se usa para definir una función `show :: a -> String`, que permite transformar un valor de tipo `a` a un `string`.

Las clases de tipos permiten definir restricciones (*Constraints*) sobre los tipos de datos a utilizar. Por ejemplo, una función con una declaración de tipo `ejemplo :: forall a . Show a => a -> a` define la restricción `Show` sobre la variable de tipo (*Type Variable*) `a` la cual implica que sólo podrá sustituirse la variable por un tipo que disponga de una instancia para esta clase, en este caso implementando la función `show`.

Kinds

Se define a un *Kind* como el tipo de un tipo. Es una manera para expresar la aridad de un tipo, si se piensa a este tipo como una función. Es decir, cuantos tipos necesita su constructor para generar un tipo concreto. Por ejemplo, el constructor del tipo de datos `data Ejemplo = ConstrEjemplo` es de *Kind* "`Type` " (o "`*`" en Haskell) y es un constructor nulario, mientras que un constructor como `Either a b = Left a | Right b` es de *Kind* `Type -> Type -> Type`, ya que necesita dos tipos adicionales para generar un tipo concreto.

Tipado Estático y Dinámico

La diferencia entre el Tipado Estático (*Static Type Checking*) y el Tipado Dinámico (*Dynamic Type Checking*) se encuentra en cuándo se valida el uso del sistema de tipos.

Cuando el chequeo de tipos se hace en tiempo de compilación, se lo refiere como estático. Significa que el compilador debe validar que el uso del sistema de tipos en el programa sea válido antes de generar la salida. Lenguajes tales como Haskell realizan todo el chequeo de tipos en tiempo de compilación, lo cual les permite asegurarse que el programa generado va a ser correcto en cuanto a tipado y que va a realizar menos esfuerzo de cómputo en tiempo de ejecución.

Por otro lado, el tipado dinámico refiere a realizar la validación de tipos en tiempo de ejecución. Generalmente se da en lenguajes que permiten tipado más relajado y conversiones implícitas entre distintos tipos.

Algunos ejemplos de lenguajes que usan tipado dinámico son JavaScript, Python, Ruby y Clojure. Por otro lado, lenguajes como PureScript, Haskell, TypeScript y Java utilizan tipado estático.

Evaluación Perezosa

El concepto de Evaluación Perezosa (*Lazy Evaluation*) es una estrategia de evaluación en la que la evaluación de toda subexpresión no es realizada hasta que su valor no sea demandado. También se conoce como evaluación no estricta o por necesidad.

Currying

Currificar una función implica transformar una función que toma una tupla de n argumentos en una que toma un solo argumento y retorna una función que tomará el siguiente argumento de la tupla y repetirá este proceso hasta retornar el valor final de la función.

Inmutabilidad

En la programación funcional, se utilizan valores inmutables. Es decir, una vez asignados, no cambian.

Pattern Matching

El *Pattern Matching* es un mecanismo de programación muy utilizado en los lenguajes funcionales que permite chequear valores contra patrones y acceder a sus componentes. Esto permite al desarrollador escribir funciones compactas, que expresan ideas potencialmente complejas, al descomponer su implementación en múltiples casos.

2.1.3. Ventajas y desafíos

Hughes plantea que una gran ventaja de la programación funcional viene de la capacidad de modularización que brindan estos lenguajes. Las características principales que destaca son las funciones de alto orden, inmutabilidad de los datos y la evaluación perezosa. Estas permiten descomponer los programas en módulos más pequeños y generales, lo cual los hace más reutilizables. (Hughes, 1989)

Thompson usa el ejemplo de la función `map :: (a -> b)-> [a] -> [b]` para demostrar como se puede combinar el poder de usar funciones de alto orden y polimorfismo para generar código altamente reutilizable. De hecho, esta función puede generalizarse aún más si consideramos la clase `Functor f` que define la función `fmap :: Functor f => (a -> b)-> f a -> f b` . De esta manera se puede utilizar no solo en listas, sino que en cualquier tipo de datos `f` que instancie la clase `Functor`. (Thompson, 2011)

Además, la transparencia referencial asegura que el código no dependa del contexto y se comporte de manera más explícita y predecible. Esto mejora significativamente la capacidad de razonar sobre el código de manera local. Además, la separación explícita entre funciones puras y funciones con efectos secundarios no solo refuerza esta comprensión y previsibilidad, sino que también aumenta la reusabilidad y mantenibilidad del código. Al facilitar la identificación clara de las operaciones que pueden causar cambios de estado externos, se simplifica el proceso de diseño, prueba y mantenimiento de sistemas complejos.

La inmutabilidad tiene varias ventajas. Al asegurar que los datos no cambien una vez creados, elimina efectos secundarios y aumenta la predecibilidad, facilitando el razonamiento sobre el comportamiento del programa. Este enfoque no solo promueve la pureza funcional, al garantizar resultados consistentes sin alterar el estado global, sino que también simplifica la gestión de concurrencia al evitar condiciones de carrera cuando múltiples procesos intentan modificar un mismo dato, haciéndolo ideal para operaciones multi hilo (Marlow, 2013). Además, la inmutabilidad posibilita optimizaciones de rendimiento mediante técnicas como la *memoización* [10] y evaluación perezosa. Finalmente, facilita el seguimiento de cambios en el estado del programa, vital para deshacer acciones y sincronizar estados. (Gordon, Parkinson, Parsons, Bromfield, y Duffy, 2012)

Dentro del contexto de tipado estático, se dice que un lenguaje es Fuertemente Tipado (*Strongly Typed Language*) cuando el tipo de toda entidad (variables, constantes, expresiones, funciones, etc) está fijado y su cumplimiento es chequeado por sus reglas de tipado. Esto busca captar una mayor cantidad de invariantes al momento de compilación y así evitar posibles *bugs* o errores en el desarrollo. Esto, junto al tipado estático, asegura que no se produzcan errores de tipos en tiempo de ejecución. (Thompson, 1991)

Los lenguajes fuertemente tipados facilitan el desarrollo utilizando Programación orientada a Tipos (*Type Oriented Programming*) donde los desarrolladores pueden diseñar su función pensando primero en sus tipos y luego rellenar con la implementación necesaria.

[10] https://es.wikipedia.org/wiki/Memoización

Por otro lado, la evaluación perezosa, añade otras ventajas y desafíos. Una ventaja es la eficiencia en términos de memoria y rendimiento. Como la evaluación perezosa solo evalúa las expresiones cuando son estrictamente necesarias, evita así asignar espacio en memoria tanto como evaluar expresiones hasta el momento preciso de su necesidad. Otra ventaja es la capacidad de trabajar con estructuras de datos infinitas, ya que las porciones de la estructura que no se están utilizando no son evaluadas, lo que permite diferentes posibilidades en la manipulación de datos y algoritmos. Por otra parte, la evaluación perezosa también tiene asociadas ciertas desventajas, principalmente debido a la complejidad introducida. Puede complejizar tanto el entendimiento, mantenibilidad, legibilidad y depuración del código (Nilsson y Fritzson, 1994). Además, la incertidumbre de los tiempos de evaluación puede ocasionar demoras en la ejecución del programa que son difíciles de anticipar y optimizar. (Marlow, 2013)

En relación con los desafíos de la programación funcional, un tema recurrente es la percepción de que este paradigma tiene una curva de aprendizaje pronunciada, especialmente para los programadores con experiencia en el paradigma imperativo. Esto se debe, por ejemplo, a conceptos avanzados del álgebra incorporados en la programación funcional como Monoides, Mónadas y Functores, que suelen resultar desconocidos y complejos de incorporar para aquellos que no estén familiarizados con estos.

Además, este paradigma cambia significativamente tanto la manera de diseñar y escribir el código, como de depurarlo, lo que puede resultar poco intuitivo para programadores acostumbrados al paradigma imperativo. No obstante, estos cambios aportan algunos beneficios. Ciertas responsabilidades que debería asumir el programador, como la gestión de memoria o la verificación de tipos, son asumidas por el lenguaje en distintos niveles.

2.2. JavaScript

Como se detalla en (Wirfs-Brock y Eich, 2020), JavaScript fue lanzado en 1995 por Netscape como un lenguaje de *scripting* de objetos cuyo objetivo era modificar las propiedades y comportamientos de objetos Java, por lo que fue pensado como un complemento de este lenguaje. Su primera versión, llamada Mocha, fue creada en apenas 10 días. Hoy en día afecta a casi todo ser humano, ya que todas las aplicaciones web utilizan JavaScript para darle interactividad al usuario, y que gracias a NodeJS [11], también es posible escribir código JavaScript para el *backend*.

Es por esto que JavaScript se ha mantenido como el lenguaje de programación más popular en la última década (Daigle y Staff, 2023), lo que obviamente conlleva una inmensa comunidad detrás donde encontrar recursos, ayuda, oportunidades laborales, etc. También, debido a su popularidad, cuenta con el registro de paquetes más grande del mundo, *npm*, el cual dispone de más de dos millones de bibliotecas disponibles [12].

[11] https://nodejs.org/en
[12] https://tinyurl.com/bibliotecas-npm

A pesar de su popularidad, JavaScript tiene algunos problemas y limitantes por las cuales tuvo varios detractores. Como se mencionó anteriormente, JavaScript implementa tipado dinámico (MDN Web Docs Contributors, 2023). Este tipado se realiza en tiempo de ejecución, lo cual tiene varias desventajas, por ejemplo:

- Se es más propenso a escribir código inconsistente y que tenga errores en tiempo de ejecución. Esto entre otras cosas se debe a la conversión implícita de tipos; de usar tipado estático podrían ser identificados en tiempo de compilación.

- La falta de tipos explícitos dificulta el entendimiento del código si es que no tiene documentación; al usar tipado estático, es claro que datos se esperan y se devuelven.

- Falta de auto completado y herramientas de desarrollo; los editores de código e IDE (*Interactive Develompent Environment*) no pueden recomendar métodos o funciones adecuadas para las variables o funciones ya que estas carecen de tipos en esa etapa.

- Los programas pueden tener un rendimiento menor ya que requieren de más comprobaciones en tiempo de ejecución. Además, a diferencia de los programas escritos en lenguajes con tipado estático, hay menos posibilidades de optimización de código en el momento de la compilación.

Debido a esto, muchos desarrolladores están de acuerdo que tener tipado estático en JavaScript mejoraría tanto la experiencia de desarrollo como la calidad de software construido.

Es por esto que con el correr de los años se han realizado distintos esfuerzos para subsanar el tipado dinámico de JavaScript y otros problemas. Ejemplo de ello es el trabajo realizado en (Jensen, Møller, y Thiemann, 2009), donde se diseña e implementa un analizador de tipos para JavaScript que podría ser integrado a un IDE. En 2014 Facebook lanzó Flow [13], otra herramienta de análisis estático similar cuyo objetivo también es añadir tipado estático.

Además de herramientas también surgieron otros lenguajes de programación que intentaban solucionar los problemas de JavaScript. Dos de ellos son CoffeeScript [14], lanzado en 2009 y TypeScript [15] desarrollado por Microsoft en 2012. El objetivo del primero fue mejorar la legibilidad y sintaxis de JavaScript, pero lo interesante a destacar es que fue el primer lenguaje en compilar a JavaScript, haciendo que esto sea moneda corriente en los siguientes años.

En cambio, TypeScript es un *superset* de JavaScript, lo que significa que extiende las capacidades del mismo. Este también funciona como un compilador a JavaScript, pero su objetivo principal es proporcionar tipado estático opcional lo cual facilita el desarrollo y mantenimiento de proyectos de gran porte. Es

[13] https://flow.org/
[14] https://coffeescript.org/
[15] https://www.typescriptlang.org/

actualmente la opción más popular para usar tipado estático en JavaScript, muestra de esto es que en 2023 fue el tercer lenguaje más utilizado en GitHub (Daigle y Staff, 2023).

A pesar de que JavaScript no tiene un enfoque puramente funcional, es posible escribir código funcional en él. El lenguaje provee algunos elementos clave como funciones consideradas ciudadanas de primera clase, permite implementar la inmutabilidad de datos, escribir funciones puras tanto como componerlas.

Ejemplo de esto es el trabajo realizado en (Anton Ioffe, 2023) donde se explica como implementar técnicas de evaluación perezosa, presentadas en 2.1.2, en JavaScript y sus ventajas, en determinados casos, en cuanto a eficiencia de memoria y rendimiento frente a la evaluación estricta, utilizada por defecto en el lenguaje. Lo hace tanto con bibliotecas existentes como Lazy.js [16] como con las funciones generadoras [17] provistas por JavaScript.

Con el paso del tiempo el ecosistema de JavaScript ha evolucionado hacia el lado de la programación funcional. Bibliotecas tales como Immutable.js, RxJS y Underscore.js han surgido, introduciendo conceptos clásicos de la programación funcional. Immutable.js [18], creado también por Facebook, introduce estructuras de datos inmutables que previenen cambios accidentales en el estado de la aplicación. RxJS [19] aporta un modelo de programación reactiva basado en Observables, lo que permite a los desarrolladores gestionar secuencias de eventos sincrónicos y flujos de datos con operadores funcionales de manera declarativa y eficiente. Por otro lado, Underscore.js [20] permite a los desarrolladores hacer uso de funciones clásicas de la programación funcional como *map*, *filter* y *foldr* (llamada *reduce* en la biblioteca).

A su vez, JavaScript ha evolucionado y en las nuevas versiones se han realizado mejoras que permiten la aplicación de patrones o técnicas de programación funcional, como agregar las funciones mencionadas (*map*, *filter* y *foldr*), la incorporación del operador de dispersión (*spread operator* [21]), que permite expandir elementos de arrays o cadenas en lugares donde se esperan múltiples argumentos o elementos; *arrow functions* [22], una sintaxis concisa para definir funciones anónimas; promesas [23], que representan resultados futuros de operaciones asíncronas; y el uso de *callbacks* [24], funciones que se pasan como argumentos y se ejecutan después de que una operación ha concluido.

A partir de 2010 surgieron varios *frameworks* y bibliotecas de JavaScript para facilitar el desarrollo de interfaces de usuario y SPA (*Single Page Applications*) que alcanzaron una notable popularidad desde su lanzamiento. Entre

[16] https://github.com/dtao/lazy.js
[17] https://developer.mozilla.orgdocs/Web/JavaScript/Reference/Statements/function*
[18] https://immutable-js.com/
[19] https://rxjs.dev/
[20] https://underscorejs.org/
[21] https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Spread_syntax
[22] https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[23] https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Promise
[24] https://developer.mozilla.org/docs/Glossary/Callback_function

ellos se destaca React [25], que con el transcurso del tiempo tomó un enfoque más funcional siendo este uno de los aspectos más distintivos y atractivos entre sus competidores. Ejemplo de esto es el paso de uso de componentes de clase a componentes funcionales. Esto fue posible gracias al desarrollo de *Hooks* en React 16.8 [26], los cuales permitieron a los componentes funcionales "engancharse" a características fundamentales de React como lo es el estado de la aplicación y el ciclo de vida de la misma.

React hace un énfasis especial sobre la programación funcional por encima de la programación orientada a objetos. Cuando uno comienza a estudiar la documentación de React, es claro que el tema de programación funcional es recurrente. En la misma se promueve la inmutabilidad del estado y la composición de funciones (componentes) pequeños para construir aplicaciones más grandes y complejas. A su vez muchas bibliotecas de React también usan patrones de programación funcional, ejemplos de esto son Redux [27] y Recoil [28], dos de las más utilizadas para el manejo de estado en proyectos de React. (Banks y Porcello, 2020)

Como se puede ver, existe un impulso o entusiasmo en torno a utilizar la programación funcional en JavaScript. A pesar de que no es un lenguaje estrictamente funcional, se dieron cambios a nivel de lenguaje y un énfasis funcional en importantes bibliotecas como React que apoyan este giro más funcional del lenguaje.

De todas formas, tomar un enfoque puramente funcional en JavaScript tiene desventajas, algunas ya fueron mencionadas, como su tipado dinámico (Phil Freeman, 2023). Otra es que su sintaxis es verbosa, es decir, requiere de mucho texto o líneas de código para lograr algo y también carece de formas poderosas de abstracción. Asimismo, a pesar de que puedan implementarse patrones de programación funcional como inmutabilidad, funciones puras o usar recursión en lugar de bucles de iteración tradicional, el desarrollador va a ser el responsable de aplicarlos correctamente ya que el lenguaje no le brindara advertencias en caso de no hacerlo de manera nativa. Sin embargo, esta falta de soporte nativo puede ser mitigada mediante herramientas como *linters*, que recomienden o impongan el cumplimiento de un estilo de programación funcional a través de reglas. Un ejemplo destacado es el *plugin* de ESLint `eslint-plugin-functional` [29] el cual ofrece algunas configuraciones preestablecidos como:

- **Estricta**: aplica reglas diseñadas para imponer estrictamente la programación funcional.
- **Recomendada**: más flexible que la configuración estricta, permitiendo una mejor integración con bibliotecas de terceros que no son funcionales.
- **Ligera**: adecuada para desarrolladores que se inician en la programación

[25] https://react.dev/
[26] https://github.com/facebook/react/releases/tag/v16.8.0
[27] https://react-redux.js.org/
[28] https://recoiljs.org/
[29] https://github.com/eslint-functional/eslint-plugin-functional

funcional o para quienes estén migrando una base de código de gran tamaño.

Aún con estas soluciones, se han buscado alternativas para abordar los problemas de JavaScript, lo que ha motivado la creación de lenguajes como PureScript.

Capítulo 3

Análisis conceptual del lenguaje PureScript

PureScript es un lenguaje de programación diseñado para proporcionar una alternativa funcional, robusta y segura para compilar código JavaScript. Inspirado en Haskell, el lenguaje ofrece una sintaxis similar y tipado estático, lo que permite a los desarrolladores escribir código seguro y fácil de mantener, a su vez aprovechando el amplio ecosistema de desarrollo de JavaScript y los beneficios de un entorno funcional puro.

Freeman lo describe como un lenguaje pequeño, estáticamente y fuertemente tipado, con tipos expresivos, inspirado en Haskell (Phil Freeman, 2023). El término "pequeño" refiere a su diseño compacto, es decir, ofrece un conjunto limitado de funcionalidades base. Esta simplicidad facilita su aprendizaje en comparación a un lenguaje más complejo.

El lenguaje incluye funcionalidades y restricciones características de los lenguajes funcionales enfocadas a la correctitud del código producido y a la productividad en el desarrollo. Entre estas características se encuentran la inmutabilidad de datos, el *pattern matching*, las funciones *currificadas*, las clases de tipos y la notación *do*. Estas características promueven un código más predecible sobre el cual es fácil razonar.

PureScript ofrece compatibilidad con JavaScript de dos maneras. La primera siendo que el código que se genera contiene funciones que pueden ser invocadas normalmente desde un entorno de JavaScript, permitiendo incorporar un módulo desarrollado en PureScript a un sistema ya existente, lo cual facilita una migración parcial. La otra manera es mediante el uso de una interfaz foránea, con la cual se definen funciones en PureScript, especificando sus tipos, y se importan sus implementaciones desde JavaScript, permitiendo así hacer uso de las bibliotecas de JavaScript en el ecosistema de PureScript.

En este capítulo, se explorarán algunas características del lenguaje PureScript, basándose en (Phil Freeman, 2023) y (Martinez, 2024); y en Pursuit [1],

[1] https://pursuit.purescript.org/

el repositorio de documentación para las bibliotecas de PureScript. En algunos casos, se destacarán similitudes o diferencias con características de Haskell o de JavaScript. Además, se analizarán algunas bibliotecas que se consideraron centrales. Se espera que el lector esté familiarizado con Haskell para facilitar la comprensión de los conceptos presentados.

3.1. Tipos de datos básicos

PureScript define tres tipos de datos básicos, `Number`, `String` y `Boolean`, que corresponden a los tipos primitivos de JavaScript `number`, `string` y `boolean`, respectivamente.

PureScript también define otros tipos básicos, propios del lenguaje, para representar números enteros (`Int`), registros (`Record`), arreglos (`Array`) y funciones (`Function`).

Se destaca que no hay una relación jerárquica entre `Number` e `Int`, pero sí existen funciones para transformar valores de un tipo al otro y vice versa, principalmente en la biblioteca de `purescript-integers` [2].

3.1.1. Registros

En PureScript, los registros, o *records*, son estructuras que contienen pares clave-valor, similares a los objetos de JavaScript. Permiten definir tipos con campos específicos y acceder a estos campos de manera segura y tipada. PureScript ofrece una sintaxis similar a la de Haskell para construirlos.

En Haskell la siguiente expresión:

```
data Point = Point { x :: Number, y :: Number }
```

Introduce las siguientes funciones al contexto

```
Point :: Number -> Number -> Point
x :: Point -> Number
y :: Point -> Number
```

En cambio, PureScript no requiere usar una definición de `data` y permite crear un registro y acceder a sus atributos de manera similar a la de un objeto en JavaScript:

```
type PointRec = { x :: Number, y :: Number }
origin :: PointRec
origin = { x: 0.0 , y: 0.0 }

originX :: Number
originX = origin.x
```

La sintaxis para actualizar un elemento del registro también es similar a la de Haskell

[2] https://github.com/purescript/purescript-integers

```
setX :: Number -> PointRec -> PointRec
setX val point = point { x = val }
```

Se profundiza sobre el trabajo con registros en una sección posterior de este capítulo (3.3).

3.1.2. Arreglos

El módulo de arreglos, `Data.Array`, se utiliza para trabajar con arreglos inmutables de manera similar a su homónimo en Haskell, pero con la particularidad que interactúa con arreglos nativos de JavaScript.

- Se mantiene la sintaxis de JavaScript para crear arreglos. Por lo tanto `[1,2,3]` representa un arreglo de enteros (de tipo `Array Int`), mientras que en Haskell, el mismo código tipa a una lista de enteros (`[Int]` en Haskell).
- Debido a la inmutabilidad de los arreglos, las operaciones de `cons` y `uncons` tienen complejidad *O(n)*, a diferencia de las listas de `Data.List` que tienen *O(1)*. Esto lleva a que no se pueda realizar *pattern matching* sobre arreglos de la misma manera que se realizaría sobre listas en Haskell. Dependiendo en el caso de uso se recomienda usar `Data.List` o `Data.Sequence`.
- El *pattern matching* de arreglos solo se aplica para arreglos de tamaño fijo. Por ejemplo:

  ```
  f [a, b, c] =  a + b - c.
  ```

- No se provee sintaxis especial para crear arreglos mediante *Array Comprehension* [3]. Se recomienda usar *do notation* junto con la función *guard* de `Control.Alternative`.

3.1.3. Funciones

Funciones infijas

En PureScript, todas las funciones utilizadas como operadores infijos, son asociativas por izquierda y tienen la mayor precedencia. Por ejemplo, si se toma la expresión:

```
2 * 3 `mod` 5 `div` 2
```

- En Haskell esta expresión será evaluada como:

  ```
  ((2 * 3) `mod` 5) `div` 2 = 0
  ```

- En PureScript, evaluará a:

  ```
  2 * ((3 `mod` 5) `div` 2) = 2
  ```

[3] https://github.com/purescript/documentation/blob/master/language/Differences-from-Haskell.md#array-comprehensions

Composición de funciones

En PureScript se reemplaza el operador . de Haskell por dos operadores (<<<) y (>>>) para usar composición hacia atrás y adelante respectivamente. El operador (.) se usa para acceder a atributos de los registros.

3.2. Tipos de datos y Clases

En PureScript las clases se definen con la palabra reservada `class`, el nombre de la clase, los parámetros de tipo y las funciones que deben ser implementadas por cualquier instancia de la clase. En el siguiente ejemplo se puede ver la definición de la clase `Show`:

```
class Show a where
  show :: a -> String
```

Luego de definir la clase, se pueden crear instancias de la misma para un tipo específico. A continuación se muestran algunas instancias para la clase `Show`. Los nombres de las instancias no son obligatorios pero se recomiendan para facilitar la lectura del código JavaScript generado:

```
instance showString :: Show String where
    show s = s

data Person = Person { name :: String, age :: Int }
instance showPerson :: Show Person where
  show (Person person) = "Name: " <> person.name <> ", age: "
      <> show person.age

data Color = Red | Green | Blue
instance showColor :: Show Color where
    show Red = "Red"
    show Green = "Green"
    show Blue = "Blue"
```

También se destaca que para las restricciones a nivel de clase utilizan una notación levemente distinta, cambiando el sentido de la flecha. Por ejemplo, una posible clasica para definir una función de comparación:

```
data Orden = LT | EQ | GT
class Eq a <= Ord a where
    compare :: a -> a -> Orden
```

Variables de tipo y polimorfismo

En PureScript se debe usar explícitamente la palabra clave `forall` para declarar una variable de tipo. Por ejemplo:

```
head :: forall a . List a -> a
```

También permite declarar varias variables en un solo `forall` y definir utilizar la palabra `forall` no solo en el comienzo de la declaración de tipo. Por ejemplo:

```
ejemplo :: forall a b. a -> b -> (forall c. c -> String)
    -> String
```

3.3. Trabajar con registros

Filas de Tipos

En PureScript, las Filas o Filas de Tipos (*Rows* o *Row Types*) son estructuras que representan colecciones desordenadas de tipos nombrados, usadas principalmente en registros. Las filas no son de *kind* `Type`; en su lugar, tienen el *kind* `Row k` para algún *kind* `k`. Esto implica que las filas no pueden existir como valores por sí mismas. En cambio, son útiles en las firmas de tipos para definir tipos de registros u otros tipos donde resulten beneficiosos los tipos etiquetados y no ordenados. (PureScript Contributors, 2023)

Las filas, además, pueden tener etiquetas duplicadas, en cuyo caso sus tipos serán reunidos en una `NonEmptyList` permitiendo múltiples campos del mismo nombre pero con diferentes tipos.

La distinción entre filas abiertas y cerradas es crucial para el sistema de tipos de PureScript:

- **Filas cerradas**: representan un conjunto definido y fijo de campos. Para definir una fila cerrada, se deben enumerar los campos, separados por comas, donde cada campo está separado de su tipo mediante dos puntos dobles. Por ejemplo:

  ```
  ( nombre :: String, edad :: Number)
  ```

- **Filas abiertas**: indican que un conjunto de campos puede ser extendido, es decir, que puede unificarse con otra fila para añadir nuevos campos. Para definir una fila abierta se debe agregar una variable de fila que este separada de los términos especificados por un *pipe*:

  ```
  ( nombre :: String, edad :: Number | r )
  ```

Polimorfismo de filas

El polimorfismo de filas (*Row Polymorphism*) permite definir funciones que pueden operar sobre registros con diferentes combinaciones de campos, ofreciendo flexibilidad y reutilización del código. En el siguiente fragmento de código se muestra un ejemplo de este concepto. La función `saludarPersona` extrae el nombre de la persona de un registro pasado como parámetro y retorna un `String` que incluye dicho nombre.

```
saludarPersona :: forall r. { nombre :: String | r }
    -> String
saludarPersona { nombre } = "Hola " <> nombre
```

El parámetro r indica que la función puede operar no solo con registros que contengan exclusivamente un campo *nombre*, sino con cualquier registro que incluya un campo *nombre* junto con una cantidad indeterminada de campos adicionales. Este parámetro representa el conjunto de campos adicionales que se pueden incluir en el registro y es implícitamente de *kind* `Row Type`, lo que significa que representa un registro de tipos que puede ser extendido con cualquier combinación de campos nombrados (Phil Freeman, 2023).

Para definir una función que acepte exclusivamente un registro con los campos especificados y sin permitir campos adicionales, se debe proporcionar una declaración de tipo sin el parámetro de filas:

```
saludarPersonaExacto :: { nombre :: String } -> String
saludarPersonaExacto { nombre } = "Hola " <> nombre
```

En este caso, la función `saludarPersonaExacto` está restringida a aceptar sólo registros con exactamente el campo *nombre* y ningún otro.

Por otro lado, si se define la función sin una declaración de tipos explícita y se utiliza el intérprete para ver la firma inferida de la función, se observa que aparece el parámetro de filas. Esto ocurre porque tanto el intérprete como el compilador infieren el tipo más general posible para las funciones y expresiones, gracias al sistema de tipos de PureScript.

```
> saludarPersonaSinTipo { nombre } = "Hola " <> nombre

> :type saludarPersonaSinTipo
forall r. { nombre :: String | r } -> String
```

Los beneficios de aplicar el polimorfismo de fila incluyen:

- **Flexibilidad e interoperabilidad**: permite manejar registros con diferentes campos, adaptándose a diversas estructuras de datos y formatos. Esto es crucial para la integración con múltiples fuentes de datos, como APIs y formatos de serialización/deserialización (JSON, XML, CSV), facilitando el desarrollo de soluciones robustas y versátiles.
- **Reutilización de código**: fomenta la creación de funciones genéricas que son aplicables a múltiples tipos de datos, optimizando así el mantenimiento y la extensión del código.
- **Seguridad de tipos**: mantiene una verificación rigurosa de tipos, previniendo errores comunes durante la compilación, sin sacrificar la flexibilidad.

Algunos casos de uso del polimorfismo de filas son:

- **Serialización y deserialización de datos**: adaptabilidad a diferentes formatos de datos, facilitando la reutilización de funciones genéricas.
- **Validación de formularios**: manejo eficiente de formularios con campos variables.
- **Construcción de APIs**: especificación flexible de dependencias, aceptando parámetros adicionales cuando sea necesario.
- **Interacción con APIs externas**: flexibilidad para trabajar con datos de APIs sobre las cuales no se tiene control directo.

Algunos de estos casos de uso serán explorados en el siguiente capítulo.

3.3.1. Trabajo con Registros a nivel de tipos

El módulo `Prim.Row` se encuentra incluido en el compilador. Este ofrece distintas clases que pueden ser utilizadas como restricciones para trabajar con filas a nivel de tipos.

- `Union`: permite combinar dos filas de tipos en una sola, uniendo sus campos.
- `Cons`: agrega un nuevo campo a una fila existente, creando una nueva fila extendida.
- `Nub`: quita los campos repetidos de una fila.
- `Lacks`: verifica que un campo específico no esté presente en una fila. Esto es útil para garantizar que ciertos campos no se agreguen inadvertidamente.

Un ejemplo del uso de estas restricciones es el caso de la biblioteca `purescript-react-basic-dom` [4], que utiliza la restricción de `Union` para facilitar el uso de atributos para elementos del Modelo de Objetos del Documento HTML (DOM). Para el caso de los elementos del DOM, estos pueden contener cientos de atributos de los cuales generalmente se trabaja con unos pocos. La siguiente definición permite al desarrollador definir solo los atributos necesarios (`attrs`), pero aún así limitar las opciones posibles (`Props_div`).

```
div  :: forall attrs attrs_
     . Union attrs attrs_ Props_div
    => Record attrs
    -> JSX
```

[4] https://github.com/purescript-react/purescript-react-basic-dom

3.4. Módulos

Todos los módulos importados con un alias se toman como *qualified imports*. Ejemplo:

```
import (find) as M from Data.Map
```

Las clases deben ser importadas con la palabra clave `class`. Ejemplo:

```
import (class Ejemplo) from Ejemplo.Class
```

Una diferencia particular con Haskell es que el módulo `Prelude` no es importado implícitamente; el único módulo que se importa de manera implícita es `Prim` [5] que incluye los tipos básicos de PureScript.

Además, ninguna biblioteca es distribuida junto al compilador; se deben instalar utilizando la configuración de *spago*. Existen maneras alternativas utilizando una herramienta de manejo de dependencias de JavaScript, como *bower* que puede ser preferible al momento de desarrollar bibliotecas, pero en el caso general se recomienda usar *spago*.

Listas

Al igual que Haskell, PureScript también ofrece el módulo `Data.List`, pero con algunas modificaciones.

- El azúcar sintáctico para la construcción de listas `[]` no existe, ya que se usa para `Data.Array`.
- El operador de concatenación es `<>` en lugar de `++`.
- Algunas funciones que en Haskell se ofrecen en el módulo `Data.List`, en PureScript se ofrecen en `Data.Foldable` y `Data.Traversable`.

3.4.1. Manejo de Efectos Secundarios - Mónadas *Effect* y *Aff*

PureScript no ofrece la mónada `IO` de Haskell, en cambio ofrece una mónada equivalente llamada `Effect`. En (Phil Freeman, 2023) se describe a esta mónada como encargada de manejar los efectos secundarios nativos. Aquí se muestra una comparativa entre la clásica función `main` de Haskell y PureScript.

```
-- Haskell
main :: IO ()
main = do ...

-- PureScript
main :: Effect Unit
main = do ...
```

[5] https://pursuit.purescript.org/builtins/docs/Prim

El módulo `Effect` ofrece distintos submódulos para trabajar con distintos tipos de efectos. Por ejemplo, en Haskell `System.Random` ofrece instancias para trabajar con generadores de números aleatorios, pero en PureScript estas herramientas se pueden encontrar en `Effect.Random`.

Por otro lado, PureScript también ofrece algunas de las mónadas clásicas y sus transformadores del paquete `Control.Monad` de Haskell como `Except`, `Maybe`, `Reader`, `State`, `Writer`, entre otras. Estas mantienen semejanzas con su contraparte en Haskell, tanto en su uso como en la manera en que se importan y se definen, incluyendo el uso de clases de tipos y de transformadores. Por ejemplo, existen instancias para `MonadEffect` como para `MonadIO` en Haskell:

```
(Monoid w, MonadEffect m) => MonadEffect (WriterT w m)
```

En el caso de la biblioteca `Aff`, esta se usa para trabajar con efectos secundarios asíncronos. Provee herramientas para encadenar computaciones asíncronas, trabajar con hilos, agregar *callbacks*, y funciones para manejar casos de error y eventuales cancelaciones de ejecución.

Permite representar la noción de promesas [6] que define JavaScript, las cuales también son utilizadas en dicho lenguaje para representar la ejecución eventual de una operación asíncrona. Las promesas se pueden usar para encadenar funciones asíncronas mediante el uso de *callbacks*, al igual que funciones que ejecuten en caso de error o al finalizar la promesa. Las promesas en JavaScript proveen el azúcar sintáctico `async` y `await` para trabajar con promesas; el código escrito usando `Aff` permite una estructura similar.

El siguiente fragmento de código en JavaScript utilizando promesas:

```
async function promiseExample() {
    console.log("Comienzo");
    const val = await algunProcesoAsync();
    console.log("Resultado: " + val);
}
```

Se puede escribir en PureScript de la siguiente manera:

```
promiseExample :: Aff Unit
promiseExample = do
  alog "Comienzo"
  val <- algunProcesoAsync
  alog $ "Resultado: " <> (show val)
  where
    alog = liftEffect <<< log
```

La biblioteca también permite asignar funciones que se encarguen de manejar casos de errores al igual que funciones de finalización similar a las funciones de promesas, `catch` y `finally`.

[6] https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise

Se adjunta un ejemplo asíncrono en JavaScript con bloques `try`, `catch` y `finally`.

```
async function tryCatchPromise() {
    try {
        await promiseExample()
    } catch (e) {
        console.log("Error " + e.toString())
    } finally {
        console.log("Fin")
    }
}
```

El mismo ejemplo puede escribirse en PureScript de la siguiente manera:

```
tryCatchPromise :: Aff Unit
tryCatchPromise = do
  finally (aLog $ "Fin") $
    cancelWith promiseExample canceller
  where
    canceller = Canceler \e ->
        aLog $ "Error " <> (show e)
    alog = liftEffect <<< log
```

Se destaca como utilizando las funciones estándar, el orden en el que se lee la función cambia comparado a JavaScript, con el finalizador al principio, la lógica central en el medio y terminando con el manejo de errores, siguiendo una estructura más similar a la del manejo de errores de Haskell.

3.5. Interoperabilidad con JavaScript - Interfaz Foránea

El lenguaje ofrece una Interfaz Foránea (FFI) que permite importar funciones de módulos de JavaScript. Esto le permite operar con la gran variedad de bibliotecas que ofrece JavaScript. La FFI está diseñada para ser muy flexible. En la práctica esto significa que se puede elegir qué tipos darle a las funciones importadas. Se puede optar por tipos muy simples o más complejos para representar posibles retornos de las funciones importadas y evitar usos erróneos del código importado. En las bibliotecas estándar se suele seguir el segundo enfoque. (Phil Freeman, 2023)

El concepto de FFI no es originario de PureScript. Existen otros lenguajes que también utilizan este concepto. Por ejemplo, Haskell y Java ambos ofrecen FFIs con C/C++.

Para definir una FFI se necesita un archivo JavaScript que defina las funciones que se quieren utilizar, y un archivo PureScript con el mismo nombre. En el archivo de PureScript se debe escribir para cada función una expresión de `foreign import` que defina su tipo y en el archivo de JavaScript se debe incluir todas las implementaciones de las funciones especificadas en el primero.

Por ejemplo, si se tiene una función de JavaScript en el archivo `Utilidades.js` que dado un número lo convierte a un `String`

```
export const numeroAString = (num) => num.toString();
```

Y se desea utilizarla en PureScript, se debe crear un archivo `Utilidades.purs` donde se importa la función foránea con las posibilidad de agregarle tipado:

```
foreign import numeroAString :: Int -> String
```

Uno de los problemas mencionados al importar código JavaScript en PureScript es que las funciones en JavaScript pueden retornar diferentes tipos de valores. Por ejemplo, la siguiente función puede retornar un elemento o `null`, dependiendo de si encuentra o no un elemento con el *id* proporcionado:

```
export const unsafeGetElementById = (id) => document.
    getElementById(id)
```

Si se decide importar la función en PureScript sin considerar la posibilidad de un valor nulo, es decir, asumiendo incorrectamente que la función siempre encontrará el elemento, se posibilita a procesar un valor de tipo inesperado, generando un error de tipos en tiempo de ejecución:

```
foreign import unsafeGetElementById :: String
    -> Effect Element
```

Este enfoque es peligroso porque el compilador de PureScript no puede garantizar la seguridad en tiempo de ejecución respecto a los valores nulos provenientes del código JavaScript.

```
main :: Effect Unit
main = do
    element <- unsafeGetElementById "idQueNoExiste"
    {- Esta línea puede tener errores en tiempo de ejecución -}
```

Tomando como ejemplo la implementación de `getElementById` de la biblioteca `purescript-web-dom` [7], una manera recomendada para trabajar con estos casos es utilizando la biblioteca `purescipt-nullable` que resuelve facilmente este problema, permitiendo transformar un posible valor `null` a algo de tipo `Maybe a` retornando `Nothing` en caso de que no exista el elemento.

Una implementación correcta sería la siguiente:

[7] https://github.com/purescript-web/purescript-web-dom

```
foreign import _getElementById :: String
    -> Effect (Nullable Element)

getElementById :: String -> Effect (Maybe Element)
getElementById = map toMaybe <<< _getElementById
```

Es responsabilidad de quien implementa la FFI de controlar que todo valor retornado corresponda al tipo especificado.

3.6. Compilación a JavaScript

A continuación, se presentará un breve ejemplo de un extracto de código, en el cual se podrán apreciar algunas similitudes claras con Haskell. Posteriormente, dicho código será compilado y se analizará el resultado generado. Se utilizará el ejemplo clásico del programa 'Hola Mundo' tomado de la página principal de PureScript [8].

```
import Prelude
import Effect.Console (log)

greet :: String -> String
greet name = "Hello, " <> name <> "!"

main :: Effect Unit
main = log (greet "World")
```

Listing 3.1: Programa Hola Mundo en PureScript

Al observar este código, se puede apreciar a primera vista una similitud sintáctica con Haskell. Se destacan algunas diferencias menores, como la elección de los operadores de concatenación (`<>`), el tipo de la función `main` es `Effect Unit`, en lugar de `IO ()`, el uso de `Console.log`, en lugar de `IO.putStrLn` y la importación explícita del Preludio.

Al compilar el programa, se obtiene el siguiente resultado:

```
import * as Effect_Console from "../Effect.Console/index.js";
var greet = function (name) {
    return "Hello, " + (name + "!");
};
var main = /* #__PURE__ */ Effect_Console.log(/* #__PURE__ */
    greet("World"));
export {
    greet,
    main
};

// Funcion log importada de Effect.Console como
```

[8] https://www.purescript.org/

```
export const log = function (s) {
  return function () {
    console.log(s);
  };
};
```

Con el propósito de este análisis, se simplificará este código y se añadirán comentarios para hacer explícito el tipo de cada función:

```
// greet :: String -> String
var greet = function (name) {
    return "Hello, " + (name + "!");
};

// log ::  String -> Unit -> Unit
const log = function (s) {
  // Función anónima :: Unit -> Unit
  return function () {
    console.log(s);
  };
};

// main :: Unit -> Unit
var main = log(greet("world"));
```

De este resultado se puede observar la aplicación de algunos conceptos de los listados en la sección de conceptos fundamentales 2.1.2.

- La función `greet` generada es pura. Se puede ver que la ejecución retorna un resultado sin interactuar con ningún estado externo.

- La función `log` está *currificada*. Toma un valor *s* y retorna una función anónima que no toma argumentos y retorna implícitamente el valor `undefined`. `undefined` es un tipo de JavaScript, con un valor único con el mismo nombre, distinto de `null`. Es representado en PureScript por el tipo `Unit` y la expresión `unit :: Unit` que reemplaza a la representación con el símbolo '`()`' en Haskell.

- La aplicación de un `String s` a la función `log` es un ejemplo del uso de los efectos secundarios. En PureScript el tipo de `log s` es `Effect Unit`. En JavaScript es representado por la función anónima interior, mencionada previamente, de tipo `Unit -> Unit`. Esta es una función que toma un valor `undefined` como entrada, ejecuta un efecto secundario `console.log(s)`, modificando un estado externo (al escribir en consola) y retorna `undefined`.

3.6.1. Extensiones de compilador

El compilador de PureScript no permite agregar extensiones como el compilador de Haskell GHC. Sin embargo, incluye los comportamientos de algunas de las disponibles en GHC [9]. Un ejemplo es `MultiParamTypeClasses` que permite definir clases de tipo con más de un parámetro.

3.7. Evaluación Perezosa

Uno de los conceptos fundamentales que diferencia a PureScript de Haskell es que PureScript no realiza evaluación perezosa y en su lugar lo hace de forma estricta. Esto de alguna manera simplifica el código JavaScript generado, ya que este lenguaje también utiliza evaluación estricta.

Por ejemplo, un *idiom* común de Haskell es definir una lista infinita y utilizar evaluación perezosa para generar la lista a demanda a medida que se necesita evaluar los valores. Como PureScript no utiliza evaluación perezosa, esto no se puede hacer simplemente con `Data.List`.

Primero, se verá una solución inspirada en una técnica conocida para modelar listas infinitas en un lenguaje estricto presentada en (Paulson, 1996), y luego se presenta una biblioteca de PureScript que utiliza una técnica similar.

Considerando, por ejemplo, un caso en el que se quiere sumar los primeros n elementos de la secuencia de números de Fibonacci. Se define la función `fibs` que genera la lista infinita de números de Fibonacci y la función `sumaFibs` que evalúa la suma de los primeros n elementos.

Se comienza escribiendo el código en PureScript como se haría normalmente en Haskell. Podría implementarse de la siguiente manera:

```
fibs :: List Int
fibs = fibs' 0 1
  where
  fibs' :: Int -> Int -> List Int
  fibs' a b = Cons a $ fibs' b (a + b)

sumFibs :: Int -> Int
sumFibs n = sum $ take n fibs
```

El problema con este ejemplo es que al ser un lenguaje estricto, al cargar este módulo a memoria, se producirá un error en ejecución al intentar evaluar la expresión `fibs`.

Se define entonces el tipo de datos auxiliar `LazyList`, el cual permitirá tratar con listas infinitas generadas a demanda.

```
data LazyList a = LazyCons a (Unit -> LazyList a)
    | LazyNil
```

[9] https://github.com/purescript/documentation/blob/master/language/Differences-from-Haskell.md#extensions

Luego, se procede a construir `lazyFibs` utilizando este nuevo tipo de datos.

```
lazyFibs :: LazyList Int
lazyFibs = lazyFibs' 0 1
  where
  lazyFibs' :: Int -> Int -> LazyList Int
  lazyFibs' a b = LazyCons a \_ -> lazyFibs' b (a + b)
```

Ahora es necesario implementar la función `take` que permita trabajar con esta lista

```
take :: forall a. Int -> LazyList a -> LazyList a
take 0 _ = LazyNil
take n (LazyCons x xs) =
    LazyCons x (\_ -> take (n - 1) (xs unit))
take _ LazyNil = LazyNil

sumFibs :: Int -> Int
sumFibs n = sum $ take n lazyFibs
```

Finalmente, se puede invocar `sumFibs 8` y evaluar únicamente los primeros 8 elementos de la secuencia de Fibonacci.

El módulo `Data.Lazy` provee una abstracción de este proceso. Define la clase `Lazy` que representa valores computados de manera perezosa y dos funciones `defer` y `force` para aplazar y forzar la evaluación de estas computaciones, respectivamente:

```
foreign import defer :: forall a. (Unit -> a) -> Lazy a
foreign import force :: forall a. Lazy a -> a
```

Existe también, en la biblioteca de listas, un submódulo `Data.List.Lazy` que define las mismas funciones que `Data.List`, pero para trabajar con listas evaluadas de manera perezosa.

3.8. Comparación de PureScript con TypeScript

En esta sección, se realiza una comparación entre PureScript y TypeScript, destacando algunas de sus diferencias y características particulares.

TypeScript disfruta de varias ventajas en comparación con PureScript gracias a su elevada popularidad, el soporte brindado por Microsoft y su amplia comunidad. Muchas de las bibliotecas de JavaScript cuentan con definiciones de tipos específicas para TypeScript, lo cual facilita su integración en proyectos existentes. Además, TypeScript dispone de una documentación más extensa y detallada, actualizaciones más frecuentes, y un equipo de ingeniería considerablemente más grande dedicado a su desarrollo y mantenimiento. Asimismo, su extensa comunidad de usuarios facilita la búsqueda de soluciones y recursos en linea.

En cuanto al sistema de tipos de TypeScript, este está diseñado para ser pragmático y se centra en la adopción incremental, facilitando su integración en

proyectos JavaScript existentes. Utiliza un sistema de tipado opcional y estático que se superpone sobre JavaScript, permitiendo utilizar todas las características de JavaScript mientras se añade una capa de seguridad a través del tipado. TypeScript también ofrece varias operaciones para la manipulación y composición de tipos como `union`, `intersection`, `pick`, `omit`, `partial`, `required`, `readonly`, y tipos condicionales.

Uno de los principios básicos del sistema de tipos de TypeScript es que la comprobación de tipos se centra en la estructura que tienen los valores. Esto se conoce como tipado estructural. En un sistema de tipos estructural, si dos objetos tienen la misma estructura, se considera que son del mismo tipo. Además, la comprobación de la estructura no requiere que los campos de un objeto sean exactamente iguales, puede haber campos adicionales.

Por ejemplo, supongamos que se cuenta con una interfaz `Point` y una función `logPoint` para imprimir objetos de ese tipo.

```
interface Point {
  x: number;
  y: number;
}

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}
```

Como se observa, las variables `point`, `point3`, `rect` y `color` no tienen una declaración de tipo asociada. Sin embargo, al invocar la función `logPoint` para cada una de ellas, TypeScript compara sus estructuras con la de `Point` durante la comprobación de tipos, determinando que las tres primeras cumplen con dicha interfaz (TypeScript Documentation, 2024)

```
const point = { x: 12, y: 26 };
logPoint(point); // logs "12, 26"

const point3 = { x: 12, y: 26, z: 89 };
logPoint(point3); // logs "12, 26"

const rect = { x: 33, y: 3, width: 30, height: 80 };
logPoint(rect); // logs "33, 3"

const color = { hex: "#187ABF" };
logPoint(color);
// Argument of type '{ hex: string; }' is not assignable to
    parameter of type 'Point'.
//     Type '{ hex: string; }' is missing the following
    properties from type 'Point': x, y
```

Por otro lado, PureScript es un lenguaje puramente funcional con un sistema de tipos más avanzado, rígido y explícito, inspirado en Haskell y basado en el

sistema de tipos Hindley–Milner. PureScript enfatiza la inmutabilidad, funciones puras, y tipado estático fuerte, lo cual puede resultar en código más robusto y menos propenso a errores en tiempo de ejecución. La naturaleza funcional de PureScript permite una mayor garantía de corrección y facilita el razonamiento sobre el comportamiento del programa.

A diferencia del sistema de tipos estructural de TypeScript, el utilizado por PureScript es del tipo nominal. Esto significa que la compatibilidad y la equivalencia de tipos se determinan en función de sus nombres explícitos y declaraciones, en lugar de su estructura o contenido.

A modo de comparar la robustez de ambos sistemas de tipos, en (Trotter, 2024) se presenta un ejemplo de un componente de React donde se demuestra cómo el sistema de tipos de PureScript puede evitar errores que TypeScript no puede detectar nativamente. En el ejemplo, se muestra cómo un componente React escrito en TypeScript puede fallar en tiempo de ejecución debido al uso incorrecto de *hooks*. Por el contrario, escribiendo el mismo componente en PureScript, este no va a compilar. Entonces mientras TypeScript depende de herramientas como *eslint* [10] para advertir sobre estos errores, PureScript los previene de forma nativa gracias a su sistema de tipos.

Respecto a la sintaxis, TypeScript permite entrelazar valores, palabras claves e información de tipos en una misma expresión. En cambio la sintaxis de PureScript, similar a la de Haskell, requiere que la información de tipos se coloque en lineas separadas utilizando `::`.

En relación a los tipos primitivos, en TypeScript no se distinguen entre números de punto flotante y enteros, simplemente se usa el tipo primitivo `number` . Por el contrario, PureScript sí distingue entre ellos con los tipos `Number` e `Int` respectivamente.

A su vez, hay diferencias importantes respecto al código generado. Estas se explorarán mediante el siguiente ejemplo obtenido de (Rajchert, 2019), donde se comparan el código fuente y el JavaScript generado por TypeScript y PureScript para un simple módulo de 'Hola Mundo'.

Por ejemplo, el siguiente extracto de código escrito en TypeScript:

```
function exclaim(str: string) {
  return str + "!!!";
}

export function main() {
  console.log('Hello');
  console.log(exclaim('World'));
}
```

Resulta en el siguiente código, al ser compilado a JavaScript:

```
"use strict";
```

[10] https://eslint.org/

```
exports.__esModule = true;
function exclaim(str) {
  return str + "!!!";
}
function main() {
  console.log('Hello');
  console.log(exclaim('World'));
}
exports.main = main;
```

TypeScript produce un código que se parece mucho al código fuente original, manteniendo la estructura y lógica de JavaScript y eliminando solo las características específicas de TypeScript. Esto hace que el código resultante sea más legible y mantenible para los desarrolladores de JavaScript. A su vez, TypeScript tiene varias opciones para configurar su compilador de acuerdo a las necesidades especificas de un proyecto [11]. Por ejemplo la opción `target` permite especificar la versión de JavaScript que se desea generar y así obtener las características ofrecidas en dicha versión, o utilizar una más antigua para maximizar la compatibilidad del código en diferentes navegadores.

En cambio el mismo programa escrito en PureScript:

```
module Main (main) where

import Prelude

import Effect (Effect)
import Effect.Console (log)

exclaim :: String -> String
exclaim str = str <> "!!!"

main :: Effect Unit
main = do
  log "Hello"
  log $ exclaim "World"
```

Al ser compilado a JavaScript, produce el siguiente código:

```
"use strict";
var Control_Bind = require("../Control.Bind/index.js");
var Data_Function = require("../Data.Function/index.js");
var Data_Semigroup = require("../Data.Semigroup/index.js");
var Effect = require("../Effect/index.js");
var Effect_Console = require("../Effect.Console/index.js");
var Prelude = require("../Prelude/index.js");
```

[11] https://www.typescriptlang.org/docs/handbook/compiler-options.html

```
var exclaim = function (str) {
    return str + "!!!!";
};

var main = function __do() {
    Effect_Console.log("Hello")();
    return Effect_Console.log(exclaim("World"))();
};

module.exports = {
    main: main
};
```

La pureza funcional de PureScript hace que el código generado en JavaScript sea significativamente diferente y más difícil de seguir para los desarrolladores acostumbrados a JavaScript. El código JavaScript resultante incluye muchas funciones auxiliares y abstracciones funcionales, lo que puede aumentar su complejidad. Esta particularidad es especialmente relevante al momento de depurar código *frontend* utilizando las herramientas de desarrollo provistas por el navegador. (PureScript Contributors, 2024)

Se puede ver que tiene varios módulos importados los cuales gestionan diferentes aspectos de la funcionalidad, como la manipulación de efectos (`Effect`, `Effect.Console`), la concatenación de *strings* (`Data.Semigroup`), etc. Dichos módulos proporcionan las herramientas necesarias para mantener la pureza funcional de PureScript. Esto resulta en un código JavaScript que conserva las propiedades funcionales que se espera en PureScript, tales como la inmutabilidad y la ausencia de efectos secundarios inesperados.

Además, dentro de la función `main` primero se crea la computación `Effect_Console.log("Hello")()` la cual luego se ejecuta inmediatamente aplicándola a `()`. Esto es una consecuencia de la pureza funcional exigida por PureScript.

En ambos casos se está creando un módulo que exporta la función `main`, si se quisiera correr el código habría que llamar a dicha función. En TypeScript se podría haber agregado la invocación de `main()` en el mismo archivo, pero PureScript lo prohíbe, asegurando que importar un módulo no resulte en efectos secundarios inesperados.

En cuanto a la interoperabilidad entre TypeScript y PureScript, se puede utilizar la FFI de PureScript para importar el código TypeScript o realizar una compilación cruzada. Además existen bibliotecas para facilitar esta interacción. Por ejemplo, `purescript-oh-yes` [12] y `purescript-ts-bridge` [13] que permiten generar declaraciones de tipos de TypeScript desde PureScript, asegurando una integración segura y eficiente entre ambos lenguajes.

En resumen, elegir entre TypeScript y PureScript depende de las necesidades especificas del proyecto y del equipo. TypeScript es flexible y permite escribir en

[12] https://github.com/justinwoo/purescript-ohyes
[13] https://pursuit.purescript.org/packages/purescript-ts-bridge

varios paradigmas con un sistema de tipos menos estricto. PureScript, en cambio, tiene un sistema de tipos más riguroso, ideal para proyectos que requieren un enfoque funcional estricto.

Mientras TypeScript ofrece flexibilidad y facilidad de adopción desde JavaScript, PureScript exige un aprendizaje más profundo y un enfoque funcional desde el inicio. TypeScript se integra bien con el ecosistema de JavaScript, mientras que PureScript se asemeja más a Haskell. El código en TypeScript es similar a JavaScript, mientras que PureScript sigue una estructura y sintaxis más alejadas.

Capítulo 4

Análisis Práctico del lenguaje PureScript - Casos de Estudio

Para el desarrollo práctico de este proyecto de grado, se decidió implementar una serie de casos de estudio con el objetivo de evaluar la facilidad de desarrollo de diversos problemas utilizando PureScript. Dichas implementaciones pueden encontrarse en el siguiente repositorio [1]. Inicialmente, se desarrolló un caso de estudio cero enfocado en configurar el entorno de desarrollo adecuado para proyectos en PureScript, evitando la redundancia de explicar este procedimiento en los casos de estudio subsiguientes. El mismo se encuentra en el Anexo A.

El objetivo general fue abordar casos que permitan trabajar con las distintas características de PureScript. Se buscó explorar aspectos como el desarrollo de un sistema *backend* y un sistema *frontend* utilizando *frameworks* comunes del ecosistema, el desarrollo de una FFI para interactuar con una biblioteca ya existente de JavaScript, el trabajo con registros y polimorfismo de filas, el desarrollo con las mónadas `Effect` y `Aff` propias de PureScript. Además, se buscó explorar la implementación de un *Domain Specific Language* (DSL) y el trabajo con clases de tipos e instancias dentro de PureScript.

El primer caso de estudio comienza con el desarrollo de un sistema *backend* para la interacción, a través de servicios REST, con una base de datos que almacena información de usuarios. Se trabaja con tecnologías y herramientas típicas del ecosistema de JavaScript. Se utiliza Node.js [2], una plataforma común para ejecutar código JavaScript para aplicaciones *backend*, que ya fue mencionada en capítulos anteriores, y Express.js [3], un *framework* de aplicaciones comúnmente utilizado para construir aplicaciones web y APIs sobre esta plataforma. Este caso fue elegido en búsqueda de un adentramiento más fácil al lenguaje,

[1] https://gitlab.fing.edu.uy/santiago.puppo/purescript-application
[2] https://nodejs.org/en
[3] https://expressjs.com/

priorizando la simplicidad mediante la implementación de una solución común.

El segundo caso de estudio se enfoca exclusivamente en la implementación de un DSL. Involucra la implementación de una estructura de datos llamada CRDT (*Conflict-Free Replicated Data Types* o Tipos de Datos Replicados sin Conflictos) [4], diseñada para sincronizar datos en sistemas distribuidos. Se optó por trabajar con esta estructura debido a la falta de casos similares previamente desarrollados en el lenguaje, y se creó una biblioteca que permita utilizar este DSL.

Finalmente, el tercer caso consta de una aplicación *frontend* que se usa para gestionar el stock de distintas tiendas de manera distribuida. Cada tienda puede mantener un control de su stock de manera local, y además conocer el stock de otras tiendas de manera remota, al igual que puede conocer un contador global de stock para cada producto. Para la gestión de los contadores de stock se hace uso de la biblioteca implementada en el caso de estudio anterior, representando los contadores con CRDTs. Para la comunicación entre tiendas se utiliza la tecnología WebRTC [5], una tecnología comúnmente empleada para la comunicación entre pares desde navegadores web. Se accede a través de una API de JavaScript, implementada por la mayoría de los navegadores. En este caso, se implementó una FFI para adaptar PeerJS [6], una biblioteca que abstrae WebRTC y facilita su uso. Se eligen estas tecnologías, particularmente, porque se consideraron áreas poco exploradas dentro del ecosistema de PureScript. Por último, se crea una interfaz gráfica utilizando React, una biblioteca utilizada típicamente en la industria para desarrollo *frontend* que ya fue mencionada en capítulos anteriores. Con este caso se busca explorar aún más características del lenguaje, trabajando con una solución más compleja, utilizando más tecnologías propias de JavaScript.

4.1. Caso de estudio 1 - REST *Endpoint* y Base de datos

El objetivo principal del primer caso de estudio fue el de familiarizarse con el lenguaje. Se buscó seguir un ejemplo de un *backend* sencillo, que sea similar a algo que el equipo ya hubiera desarrollado en otro lenguaje. Para ello se siguió la guía en (Sarkar, 2020) que presenta un ejemplo de como implementar una API REST para interactuar con una base de datos con usuarios.

El caso plantea un escenario donde se quiere almacenar datos de usuarios. Para cada usuario se desea asociarle un nombre y un identificador. Se define una biblioteca sencilla para conectarse a la base de datos para leer y modificar los datos de los usuarios mediante funciones implementadas en PureScript. Además, se define un servicio REST que permite acceder a la información de los usuarios a través de mensajes HTTP.

[4] https://crdt.tech/implementations
[5] https://webrtc.org/
[6] https://peerjs.com/

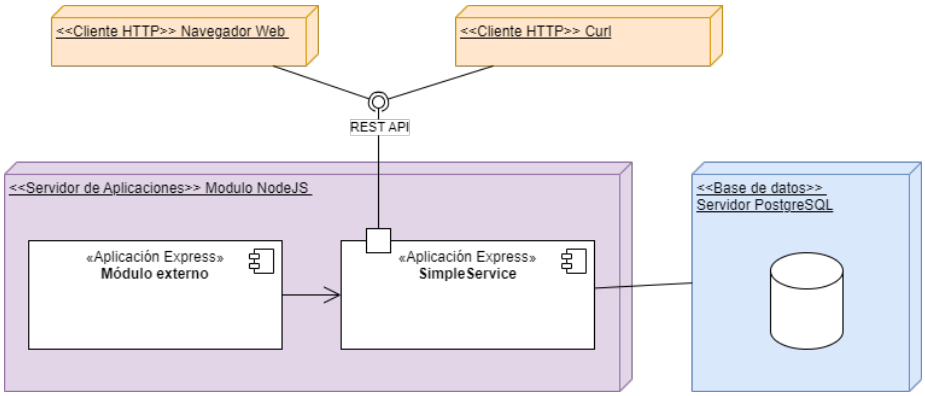

Figura 4.1: Arquitectura Caso 1

La arquitectura del caso (Figura 4.1) consiste en un servidor NodeJS que se comunica con una base de datos PostgreSQL y expone sus servicios mediante una interfaz REST. Como *framework* para construir la aplicación web, se utilizó la biblioteca `purescript-express`, un adaptador para Express. Para la interacción con la base de datos se utilizó la biblioteca `pursecript-postgresql-client` que adapta la biblioteca `node-postgres`.

4.1.1. Módulos implementados

A continuación una breve descripción de los módulos implementados:

SimpleService.Types: Define los tipos de nuestro servicio. En este caso, principalmente el tipo de Usuario como:

```
newtype User = User {id :: UserId, name :: String}
```

También define instancias para las clases `FromSQLRow` y `ToSQLRow` para hacer la transformación de una entrada de una tabla en una base de datos a nuestro tipo Usuario y vice versa.

SimpleService.Persistance: Define las distintas operaciones que se pueden realizar sobre la tabla de usuarios en la base de datos. Las operaciones que se definieron son para insertar, encontrar, actualizar y borrar un usuario y para obtener una lista general de usuarios.

SimpleService.Handler: Define los métodos para gestionar la interacción de la REST API con el módulo de persistencia. Este se encarga de interactuar con el módulo y retornar la información y estado HTTP correspondientes de acuerdo a la petición y el resultado de la operación. En este caso solo se implementó la

operación de obtener un usuario por *id* y obtener la lista de usuarios. Estas funciones retornan resultados de tipo `Handler` de la biblioteca de Express, el cual es un alias para `HandlerM Unit`. Un *Handler* en Express es un elemento encargado de procesar en base a una petición HTTP su respectiva respuesta; en la biblioteca *wrapper* esto se hace mediante la mónada `HandlerM`.

SimpleService.Server: Contiene la lógica necesaria para levantar el servidor de NodeJS e iniciar la aplicación Express. Esto se hace asociando a un puerto pasado por parámetro y redirigiendo los mensajes HTTP a una aplicación Express. La aplicación Express se encarga de asociar las rutas al *Handler* correspondiente.

SimpleService.Config.Database: Contiene la lógica para leer un archivo de configuración (`.env`) y cargar los distintos parámetros para poder conectarse con la base de datos.

Main: Por último, el módulo Main esta encargado de iniciar el proceso, cargando la configuración y pasándola al módulo de Servidor

A continuación, se presenta un breve resumen de la implementación del caso de uso de listar usuarios, para ilustrar la interacción entre los módulos y explicar algunos extractos del código.

Se comienza presentando un extracto del módulo `SimpleService.Server`:

```
runServer :: Int
          -> Effect Unit
runServer port = void $ runAff endCallback $ do
  -- Cargar configuración de la base de datos
  databaseConfig <- SimpleService.Config.Database.load
  liftEffect do
    -- Crear un pool de conexiones
    pool <- PG.new databaseConfig
    -- Asosciar app al puerto
    let app' = app pool
    listenHttp app' port listenCallback

app :: PG.Pool -> Express.App
app pool = do
  get "/v1/user/" $ getUsers pool
  get "/v1/user/:id" $ getUser pool
```

Este código esta encargado de iniciar el servidor. Para ello debe hacer tres cosas: cargar la configuración de la base de datos, crear un *pool* de conexiones, y asociar una `Express.App` al puerto determinado. Se debe usar una combinación de código síncrono y asíncrono, en particular debido a que `SimpleService.Config.Database.load` es de tipo `Aff Configuration` (asíncrono). Esta combinación se

logra mediante el uso de las bibliotecas `Aff` y `Effect` y las funciones `runAff` y `liftEffect`, donde:

```
runAff :: forall a. (Either Error a -> Effect Unit) -> Aff a ->
     Effect (Fiber Unit)

liftEffect :: forall (m :: Type -> Type) (a :: Type).
    MonadEffect m => Effect a -> m a
```

La primera función ejecuta una computación asíncrona seguida por un *callback* pasado como primer parámetro que procesa el resultado de la computación. La segunda función simplemente eleva la computación de `Effect`, en este caso al contexto de `Aff`. Lo principal a destacar aquí es que al querer combinar computaciones asíncronas y síncronas, se debe utilizar el contexto de `Aff`, lo cual implica utilizar una función del estilo de `runAff` que utilice un *callback* para evaluar el resultado.

Una vez teniendo inicializado el servidor, el caso comienza cuando un cliente HTTP realiza una petición al servicio correspondiente. Por ejemplo:

```
curl http://localhost:4000/v1/user
```

Siguiendo la asignación de rutas especificadas por la aplicación, la función que procesará la petición será `SimpleService.Handler.getUsers`:

```
getUsers :: PG.Pool -> Express.Handler
getUsers pool = do
  dbRes <- liftAff $ runExceptT ( P.listUsers (PG.fromPool pool
      ) )
  case  dbRes of
    Left e -> respond 500  { error: e }
    Right users -> respond 200 (users)
```

donde `P.listUsers` está definida como

```
listUsers :: PG.Connection -> DBM (Array User)
listUsers conn =
  query conn (PG.Query "select id, name from users") Row0
```

La última ejecutará la búsqueda especificada, retornando todos los usuarios de la tabla *users*, los cuales serán transformados al tipo `User` utilizando la instancia de `FromSQLRow User` definida en `SimpleService.Types`.

```
instance userFromSQLRow :: FromSQLRow User where
  fromSQLRow [id, name] =
```

```
  User <$> ({ id: _, name: _} <$> fromSQLValue id <*>
      fromSQLValue name)
fromSQLRow xs = Left $ "La fila tiene " <> show n <> "
    atributos, se esperaban 2."
  where n = Array.length xs
```

4.1.2. Proceso de desarrollo

Durante el desarrollo del caso, se siguió mayormente el enfoque planteado en la guía. Sin embargo, dado que la misma se creó con una versión de PureScript anterior a la que se utilizó en este trabajo, resultó más difícil de lo esperado adaptar la guía.

En primer lugar, la guía utiliza Bower y Pulp, en lugar de Spago, como herramientas gestoras de paquetes y de proyecto. Bower es una herramienta que anunció un proceso de descontinuación de manera oficial en 2017, unos meses después de la publicación de la guía [7]. Pulp es una herramienta que interactúa con el compilador de PureScript para compilar el proyecto, se utiliza principalmente junto con Bower en el desarrollo de bibliotecas de PureScript. Para este caso de estudio, a fines de utilizar herramientas actualizadas, se optó por utilizar Spago para el manejo de bibliotecas y de proyecto, realizando los cambios necesarios para instalar los paquetes.

Además de esto, en la actualización de PureScript 0.15, el lenguaje dejó de ser compatible con la sintaxis de JavaScript CJS (CommonJS), pasando a soportar únicamente ESM (ECMAScript Modules), como se puede ver en la guía de migración [8]. Esto provocó que al momento de compilar nuestro proyecto con dependencias creadas en una versión anterior de PureScript, se obtuvieran errores de compilación dentro de las mismas, ya que estas usaban términos de CJS tales como `nombreDemódulo = require('nombreDemódulo')` o `exports.miFunc = ...` [9].

En particular, la dependencia a `purescript-express` en su última versión oficial al momento, soportaba hasta PureScript v0.14, provocando una incompatibilidad con la versión de PureScript elegida. Se logró solucionar el problema modificando la versión de la dependencia para que utilice la rama *master* del repositorio ya que esta, al momento de realizar el caso, contiene los cambios necesarios para soportar la versión v0.15. Esta solución no es segura para un ambiente de producción, pero para los fines de este caso se consideró suficiente.

Por último, otro problema que afectó el desarrollo fue el cambio de sintaxis y *namespaces* entre algunas dependencias incluidas en la guía y sus versiones compatibles con la versión de PureScript elegida. Ejemplos de esto son:

- El cambio de las mónadas `Effect` y `Aff`: que provocaron cambios en las declaraciones de tipos de algunas funciones. Por ejemplo, en la guía, la función `main` tiene la siguiente firma:

[7] https://devblogs.microsoft.com/dotnet/what-happened-to-bower/
[8] https://github.com/purescript/documentation/blob/master/migration-guides/0.15-Migration-Guide.md
[9] https://wanago.io/2024/03/18/ecmascript-modules-esm-commonjs/

```
main :: forall eff. Eff ( console :: CONSOLE
                        , express :: EXPRESS
                        , postgreSQL :: PG.POSTGRESQL
                        | eff) Unit

-- definido en Control.Monad.Eff.Console
foreign import data CONSOLE :: Effect
-- definido en Node.Express.Types
foreign import data EXPRESS :: Effect
-- definido en Database.PostgreSQL
foreign import data POSTGRESQL :: Effect
```

En el renombre de `Effect` los implementadores decidieron dejar de utilizar la sintaxis con filas buscando simplificar el código. El mismo código mencionado arriba pasa a ser simplemente:

```
main :: Effect Unit
```

Esto puede llevar a causar confusión si se trabaja con documentación creada antes de este cambio.

- También hubieron cambios en dependencias no tan centrales al lenguaje como en `purescript-postgresql-client`.

  - La biblioteca también paso por una reorganización de los módulos ofrecidos, que implicó cambios mínimos de código al momento de importarlos.
  - Las firmas de algunas funciones cambiaron entre las versiones, por un lado, por la actualización de las mónadas `Aff` y `Effect` mencionada previamente.

    Se presenta el ejemplo de la función `query`. En la versión de la biblioteca utilizada en la guía (2.2.0) la firma de la función era:

    ```
    query :: forall eff o i.
        ToSQLRow i =>
        FromSQLRow o =>
        Connection -> Query i o -> i -> Aff (postgreSQL ::
            POSTGRESQL | eff) (Array o)
    ```

    Mientras que en su versión más reciente (3.0.2) su firma pasa a ser:

    ```
    query :: forall m o i.
        ToSQLRow i =>
        FromSQLRow o =>
        MonadError PGError m =>
    ```

```
MonadAff m =>
Connection -> Query i o -> i -> m (Array o)
```

> Se destaca que se pasa de utilizar `(postgreSQL :: POSTGRESQL | eff)` para restringir la mónada de retorno a usar restricciones de clases de tipo `MonadError PGError m => MonadAff m`. Fue por esto que para el caso de estudio se resolvió por definir la mónada auxiliar `DMB a = ExceptT DB.PGError Aff a` que instancia ambas clases.

Estos fueron cambios menores a los cuales se logró adaptar una vez identificados, pero dada la inexperiencia en el lenguaje resultó confuso al comienzo del desarrollo.

4.1.3. Conclusiones y aprendizajes

A partir de este caso se logró una familiarización inicial con el lenguaje. Se obtuvo una primera vista de la interacción de PureScript con bibliotecas de JavaScript, en este caso Express y Node para levantar un servidor *backend* rápidamente. También se vio una posible arquitectura o solución para un caso de uso muy común en la industria, la cual a nivel de diseño no es muy compleja.

En primer lugar, algo a destacar sobre el diseño de la solución final es que comparte cierta similitud con una solución esperada en JavaScript. Además, la estructura de código de la solución lograda mantiene cierta similitud a una del paradigma imperativo, haciendo amplio uso de efectos secundarios mediante la mónada `Effect`. Desde este punto de vista no se aprovecha tanto (de manera explícita) algunas características funcionales como la pureza o la inmutabilidad de datos. De todas formas, sí se hace énfasis en la expresividad del sistema de tipos de PureScript, permitiendo diferenciar, por ejemplo, bloques de código asíncrono del síncrono. De forma similar, el tipado estático, por más que fue difícil de trabajar, ayudó a detectar errores de forma temprana. También, el trabajo con clases de tipo e instancias, por más que no sea específico de PureScript, mostró características deseables como la abstracción y reusabilidad de datos, en particular en las bibliotecas utilizadas.

Dado el tamaño de la solución se puede argumentar que resultaría más beneficioso una implementación en JavaScript, desde un punto de vista que permite una implementación más veloz evitando algunas complejidades en el desarrollo. Sin embargo, al momento de escalar el sistema es claro que características como el tipado estático y un sistema de tipos expresivos darán una ventaja mayor que la velocidad de desarrollo a largo plazo.

Aprender a trabajar con la mónada `Aff` tuvo un costo inicial extra, pero una vez que se logró entender como utilizarla, su uso resultó intuitivo, permitiendo encadenar distintos efectos asíncronos de manera sencilla y permitiendo un código limpio sin el uso excesivo de *callbacks*.

Más allá de los problemas técnicos relacionados a la gestión de versiones, se destaca las similitudes de interactuar con las bibliotecas originales de JavaScript y sus adaptadores a PureScript. Dejando de lado la dificultad extra que

llevó entender como utilizar la biblioteca `purescript-aff`, no fue difícil entender como trabajar con estas bibliotecas ni tampoco asociarlo a su equivalente en un ambiente de JavaScript.

En cuanto a la practicidad del proceso de desarrollo, este fue más lento de lo que se esperó inicialmente, en principio debido a la elección de la guía a seguir. Haber trabajado con documentación desactualizada implicó investigar, analizar y realizar cambios para lograr llegar al resultado esperado. Además, PureScript, con el paso del tiempo tuvo cambios significativos provocando una baja compatibilidad entre sus versiones. El tener que solucionar este tipo de problemas entre bibliotecas en un momento tan temprano en el aprendizaje del lenguaje fue un obstáculo para lograr la introducción esperada.

A partir del problema atravesado con las versiones de las herramientas, se decidió para los siguientes casos, pasar a utilizar material más reciente como base de los proyectos y apegarse a las prácticas sugeridas del lenguaje, por ejemplo, trabajando con *package sets* pre-armados y gestionando las dependencias externas mediante la funcionalidad que ofrece *npm* con el archivo `package.json`. Se destaca la importancia de buscar una comunidad activa al momento de trabajar con un lenguaje nuevo.

De todas maneras, por más que se llegó a una solución válida, no se considera que utilizar PureScript para este tipo de soluciones sea particularmente beneficioso. Se puede lograr implementar una solución similar manteniendo el beneficio del tipado estático utilizando un lenguaje como TypeScript. Por supuesto, en un caso real siempre se debe considerar aspectos como experiencia en el lenguaje, tiempo esperado de desarrollo y soporte para tomar esta decisión, además de los beneficios específicos al lenguaje como son la alta reusabilidad de tipos, el sistema de tipos expresivo, entre otras.

4.2. Caso de estudio 2 - CRDT

En el segundo caso de estudio se optó por implementar una biblioteca en PureScript, enfocándose en una área aún no explorada en este lenguaje: los CRDT. Estas estructuras de datos están diseñadas para facilitar la replicación de datos en sistemas distribuidos, garantizando eventualmente un estado de consistencia entre los nodos.

Los CRDTs aseguran que todas las réplicas de un dato converjan a un estado común correcto a pesar de las actualizaciones concurrentes y fallos en el sistema, basándose en propiedades matemáticas simples como la conmutatividad, idempotencia y asociatividad. En otras palabras, un CRDT es una estructura de datos replicada a través de múltiples computadoras en una red, con las siguientes características (Kauffman, 2023):

- **Sin necesidad de coordinación**: la aplicación puede actualizar cualquier réplica de manera independiente y concurrente, sin necesidad de coordinar con otras réplicas.

- **Auto-sanación**: un algoritmo (parte del propio tipo de dato) resuelve automáticamente cualquier inconsistencia que pueda ocurrir.
- **Convergencia**: aunque las réplicas puedan tener estados diferentes en un momento dado, se garantiza que eventualmente converjan.

Estas estructuras son útiles para desarrollar aplicaciones distribuidas a gran escala y ofrecen una solución práctica y teóricamente sólida al problema de mantener la consistencia de datos en sistemas distribuidos (Shapiro, Preguiça, Baquero, y Zawirski, 2011). El tipo de aplicaciones distribuidas que se pueden construir utilizando CRDT varia en tipos y complejidad, desde editores de texto colaborativos, como Google Docs, hasta aplicaciones de mensajería en tiempo real, videojuegos en línea, redes sociales descentralizadas, etc.

Ya que el foco de la tesis no está en el diseño de aplicaciones distribuidas, se eligió un caso de uso más sencillo para la aplicación de CRDTs: un contador distribuido. Esta elección se inspiró en la siguiente serie de *blogs* donde se implementa un CRDT en JavaScript (Kauffman, 2023), y también se utilizó como referencia una implementación funcional de CRDT en Scala [10].

Lo primero a realizar fue construir un GCounter (*Grow-only Counter*), que es un contador que soporta únicamente incrementos. Su estructura básica requiere que cada nodo en el sistema tenga:

- un identificador único (UUID), dado la naturaleza distribuida del sistema.
- un mapa que asocie UUIDs y contadores para cada nodo en el sistema.
- funciones para obtener el valor del contador y para sincronizarlo con otros nodos.

Para ejemplificar el uso de este contador, supóngase que hay tres nodos en el sistema: `A`, `B` y `C`. Cada nodo tiene un mapa de tres elementos que asocia UUIDs a contadores. Inicialmente, todos los contadores están en 0. Cuando un nodo incrementa su contador, solo incrementa su propio valor en el mapa. Por ejemplo:

- Nodo `A` incrementa su contador: el mapa de `A` sería `{A: 1, B: 0, C: 0}`
- Nodo `C` incrementa su contador: el mapa de `C` sería `{A: 0, B: 0, C: 1}`

Para sincronizar el estado entre nodos, se requiere una regla para decidir quién tiene la información más reciente. La regla se basa en dos propiedades de este tipo de contadores: un contador solo puede ser incrementado y cada nodo solo puede incrementar su propio contador, es decir el nodo `A` solo incrementa el contador asociado a `A`. A partir de estos dos puntos, la regla para tomar dicha decisión es mantener como información más reciente al conteo más alto. Así, cuando dos nodos se sincronizan, combinan sus mapas tomando el valor máximo de cada posición, para asegurar que no se pierdan incrementos. Por

[10] https://github.com/schrepfler/crdt-kit

ejemplo, si el mapa de `A` es `{A: 5, B: 3, C: 2}` y el de `B` es `{A: 4, B: 4, C: 1}`, después de sincronizar, ambos nodos tendrán el mapa `{A: 5, B: 4, C: 2}`, que representa el valor máximo de cada contador.

Finalmente, el valor total del contador se obtiene sumando todos los elementos del mapa. Para el mapa `{A: 5, B: 4, C: 2}` el valor total sería 11.

Luego, se desarrolló un PNCounter (*Positive/Negative-Counter*), capaz de registrar tanto incrementos como decrementos. Este contador se implementó utilizando dos GCounter: uno para los incrementos y otro para los decrementos. La lógica de sincronización se adaptó para manejar estas dos dimensiones de manera independiente, aplicando la función de sincronización de GCounter para ambos contadores respectivamente. Luego, el valor total del contador se calcula como la diferencia entre el total de incrementos y decrementos.

Una vez construidos dichos contadores, se realizó un esfuerzo por abstraer la lógica detrás de CRDT para que puedan ser utilizada con otros tipos de datos utilizando una variable de tipos.

4.2.1. Módulos implementados

Finalmente, el código de la biblioteca quedó separado en los siguientes tres módulos, cada uno con sus respectivos contenidos:

CRDT:

- Tipo `UUID`: identificador único universal, representado aquí como una cadena de caracteres (`String`).

  ```
  type UUID = String
  ```

- Tipo `CounterType`: Define la estructura de un contador genérico, que incluye un `shardId` de tipo `UUID` y un `payload`, que es un mapa (`Map`) de identificadores únicos a valores de algún tipo v.

  ```
  type CounterType v = {
      shardId :: UUID,
      payload :: Map UUID v
  }
  ```

- Clase `StateBased`: Esta clase define una interfaz común para CRDTs basados en estado, con operaciones para obtener el valor actual del CRDT (`value`) y para sincronizar dos nodos (`merge`). Además, se utilizó una dependencia funcional en la clase para vincular el tipo de datos `m` con el tipo `r` asociado (`m -> r`). Esto garantizó que cualquier instancia de `StateBased` que implementara sus operaciones lo hiciera de acuerdo con su tipo de datos correspondiente.

```
class StateBased m r | m -> r where
    value :: m -> r
    merge :: m -> m -> m
```

GCounter:

- GCounter: modela el contador GCounter como un tipo algebraico de datos que contiene una instancia de `CounterType Int`, donde el mapa asocia identificadores de nodos con enteros que representan el estado del contador que tiene dicho nodo.

```
data GCounter = GCounter (CounterType Int)
```

- La clase `GCounterClass` extiende `StateBased` y añade la operación `increment` , que incrementa el valor del contador en una cantidad dada.

```
class (StateBased gcnt t) <= GCounterClass gcnt t where
    increment :: gcnt -> t -> gcnt
```

- La implementación de `increment` busca el valor actual asociado con `shardId` en el `payload` y lo incrementa por `amt`, o lo inicializa a `amt` si no está presente.

```
instance gCounterIntInst :: GCounterClass GCounter Int
    where
  increment (GCounter gcnt) amt =
    if amt > 0 then
      case lookup gcnt.shardId gcnt.payload of
        Just value ->
            GCounter
            { shardId: gcnt.shardId,
            payload: insert
                gcnt.shardId
                (value + amt)
                gcnt.payload }
        Nothing ->
            GCounter
            { shardId: gcnt.shardId,
            payload: insert
                gcnt.shardId
                amt
                gcnt.payload }
    else
       GCounter gcnt
```

- La instancia `stateIntInst` implementa las operaciones `value` (que suma todos los valores en el `payload`) y `merge` (que sincroniza dos `GCounters` tomando el máximo de los valores asociados para cada `shardId`).

```
instance stateIntInst :: StateBased GCounter Int where
  value (GCounter gcnt) =
    foldr (\val acc -> val + acc) 0 (values gcnt.payload)
  merge (GCounter gcnt1)  (GCounter gcnt2) =
    GCounter
        { shardId: gcnt1.shardId,
        payload: merge' gcnt1.payload gcnt2.payload }

merge' :: forall a b. (a :: Type) (b :: Type). Ord a => Ord
     b => Map a b -> Map a b -> Map a b
merge' p1 p2 = unionWith (\a b-> max a b) p1 p2
```

PNCounter:

- PNCounter: tipo algebraico de datos que combina dos GCounters, uno para las operaciones de incremento y otro para las de decremento.

```
data PNCounter = PNCounter G.GCounter G.GCounter
```

- La clase `PNCounterClass` extiende esta idea con operaciones de `increment` y `decrement`, implementadas como incrementos en cada contador.

```
class PNCounterClass gcnt t | gcnt -> t where
   increment :: gcnt -> t -> gcnt
   decrement :: gcnt -> t -> gcnt

instance pnCounterIntInst :: PNCounterClass PNCounter Int
    where
  increment (PNCounter pos neg) amt =
    (PNCounter (G.increment pos amt) neg)
  decrement (PNCounter pos neg) amt =
    (PNCounter pos (G.increment neg amt))
```

- La instancia `pnStateInst` para `StateBased` define cómo obtener el valor de un `PNCounter` (la diferencia entre los valores de los incrementos y decrementos) y cómo fusionar dos `PNCounters` (fusionando sus contadores de incrementos y decrementos respectivamente).

```
instance pnStateInst :: CRDT.StateBased PNCounter Int where
  value (PNCounter pos neg) = (CRDT.value pos) - (CRDT.
      value neg)
  merge (PNCounter pos neg) (PNCounter pos' neg') = (
      PNCounter (CRDT.merge pos pos') (CRDT.merge neg neg')
      )
```

4.2.2. Conclusiones y aprendizajes

En este caso de estudio, se exploraron y aplicaron varias características avanzadas de PureScript para implementar y trabajar con CRDTs.

Se desarrolló un DSL (*Domain Specific Language*) [11] específico para manejar las operaciones de CRDTs. Este DSL facilitó la abstracción y reutilización del código, permitiendo la creación de los contadores GCounter y PNCounter con su lógica de sincronización y actualización definida de manera clara.

Durante la implementación, se utilizaron registros, clases e instancias para estructurar los datos, abstrayendo y reutilizando el código de manera eficiente. Se definieron clases como `StateBased` y `GCounterClass` para proporcionar interfaces comunes y polimórficas para las operaciones de CRDTs, lo cual permitió trabajar con diferentes tipos de contadores.

Además, se utilizaron dependencias funcionales para asegurar una correcta asociación entre tipos y sus correspondientes operaciones. Estas agregan restricciones de tipos mejorando así tanto la robustez como la mantenibilidad del sistema.

En PureScript, las dependencias funcionales están disponibles de manera nativa, lo que simplifica su uso y evita configuraciones adicionales. Por otro lado, en Haskell, estas dependencias no son soportadas de manera nativa y deben ser activadas a través de extensiones que implementa el compilador GHC [12].

4.3. Caso de estudio 3 - FFI PeerJs y *Frontend* basado en React

El caso de estudio final busca explorar dos características de PureScript aún no abordadas: la implementación de una FFI y la implementación de un *frontend* utilizando un *framework* común en JavaScript. También se busca incorporar el DSL implementado en el caso anterior y explorar el trabajo con registros en PureScript.

El caso de estudio consiste en un grupo de tiendas que desea mantener un stock global de productos de manera sincronizada. Para cada uno de sus productos se debe tener un contador distinto, que permita conocer el stock

[11] https://en.wikipedia.org/wiki/Domain-specific_language
[12] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/functional_dependencies.html

local de cada una de estas tiendas y el stock global para cada producto, definido como la suma de el stock local para dicho producto de cada tienda. Las tiendas además deben conocer y actualizar un stock global en base a las ventas y restock que realicen.

4.3.1. Marco Teórico

A continuación se explican brevemente algunas tecnologías a utilizar en este caso de estudio a modo de contextualizar al lector.

FFI - Comunicación entre pares

Para la implementación de la FFI, se decidió adaptar la biblioteca PeerJS [13]. Esta es una biblioteca de JavaScript que simplifica el proceso de establecimiento de conexiones *peer-to-peer*, permitiendo comunicaciones directas entre navegadores para compartir archivos, realizar videollamadas y más, en tiempo real y sin servidores intermedios. Para esto utiliza WebRTC (*Web Real-Time Communication*) [14], una tecnología de código abierto, implementada por la mayoría de los navegadores modernos, que proporciona una API sencilla para manejar la conectividad entre pares y el envío y captura de datos.

Por sí solo, WebRTC es un estándar complejo que ofrece una API extensa para la comunicación en tiempo real entre pares, principalmente se utiliza para casos de audio y video, pero también permite comunicar datos. PeerJS ofrece una interfaz configurable simplificada que permite un uso más fácil de la tecnología. WebRTC no especifica un mecanismo para señalizar las conexiones; respecto a esto, PeerJS ofrece una solución simple para levantar un servidor gestor de conexiones llamado PeerServer [15]. Este servidor únicamente se encarga de conectar pares entre sí, pero no es un intermediario para la comunicación; es decir, los mensajes que envía un par a otro no pasan por este servidor una vez conectados.

Frontend - Desarrollo GUI

Un aspecto relevante para el equipo fue el desarrollo de una interfaz de usuario. Para ello, se consideraron dos opciones populares, en principio porque estas contaban con soporte directo a través de canales dedicados en el servidor de Discord de PureScript [16], lo que garantizaba que fueran bibliotecas activas y respaldadas por la comunidad. La primera opción fue Halogen [17], una biblioteca que permite construir interfaces de usuario sin abandonar el entorno de PureScript. La segunda opción fue `purescript-react` [18], que ofrece una abstracción sobre React y permite interoperar con dicha biblioteca, aprovechando su amplio

[13] https://peerjs.com/
[14] https://webrtc.org/
[15] https://github.com/peers/peerjs-server
[16] https://discord.gg/Dhab6Vmm3H
[17] https://github.com/purescript-halogen/purescript-halogen
[18] https://github.com/purescript-react

ecosistema. Dado que React tiene una popularidad considerable en el desarrollo moderno, se consideró más conveniente optar por esta opción.

React es una biblioteca de JavaScript de código abierto desarrollada por Facebook para construir interfaces de usuario interactivas. Lanzada en 2013, Se ha destacado por su enfoque declarativo y su capacidad para crear aplicaciones web rápidas y dinámicas mediante la construcción de interfaces con componentes reutilizables.

Una característica clave de React es su adopción del enfoque funcional en el desarrollo de interfaces, manifestado en varias formas:

- **Componentes funcionales**: React favorece el uso de componentes como funciones de JavaScript, lo cual simplifica el código y mejora la mantenibilidad. Además, con la introducción de los *hooks*, funciones especiales que se "enganchan" al ciclo de vida del componente, es posible manejar el estado y los efectos secundarios de manera funcional. Esto permite mantener el componente como una función pura mientras se realizan operaciones como consultas HTTP, manejo de I/O o gestión del estado. Por convención los *hooks* se definen con el prefijo "use".

- **Declarativo y reactivo**: Permite describir la interfaz de usuario de manera declarativa, enfocándose en el resultado deseado sin preocuparse por las actualizaciones del DOM, alineándose con los principios funcionales.

- **Inmutabilidad**: Fomenta el uso de datos inmutables, lo que facilita la comparación eficiente del Virtual DOM y evita efectos secundarios inesperados.

- **Composición de componentes**: La composición de componentes pequeños promueve la modularidad y reutilización del código, en línea con la programación funcional.

4.3.2. Diseño

En cuanto a la arquitectura de la aplicación, se decidió seguir una arquitectura *"Peer to Peer"* donde se modelan las tiendas como distintos nodos, o *peers*, conectados mediante el protocolo WebRTC. Cada nodo conoce a las demás tiendas y se encarga de enviar mensajes sobre eventos de venta o restock, así como de recibir y manejar los mensajes de otros nodos. Se utilizó la biblioteca PeerJS para simplificar el uso de WebRTC, incorporada a través de una FFI. También se utilizó una implementación de un servidor de PeerJS para intermediar las conexiones. Esta última no se implementó en PureScript. Se adjunta un ejemplo en la Figura 4.2

Para el *frontend* se decidió usar el framework de React utilizando la biblioteca `purescript-react`. Esta adapta los conceptos de React como Componentes y *Hooks* para ser utilizados desde PureScript, proporcionando así una estructura de código similar.

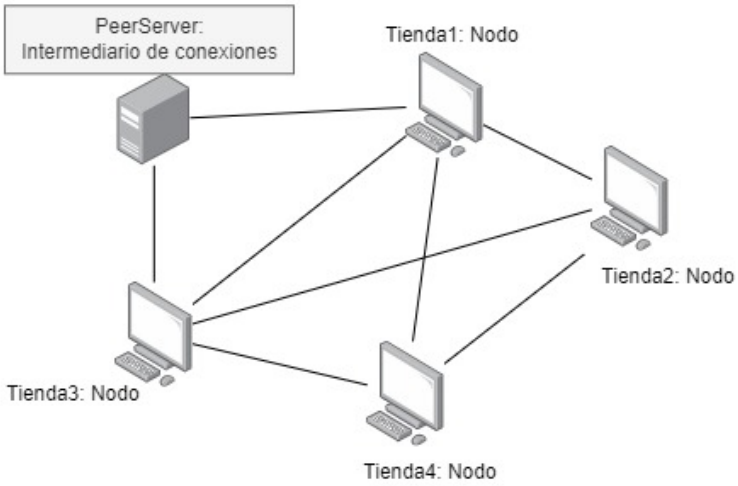

Figura 4.2: Caso 3 - Ejemplo sistema P2P

En cuanto al modelo general de la aplicación, se representaron los contadores de los productos mediante CRDTs, utilizando la biblioteca implementada en el caso de estudio anterior. Como el trabajo con CRDT es independiente a la capa de comunicación, se debió decidir cómo y con qué regularidad se va a comunicar las actualizaciones de estado.

Debido a que no se esperaba trabajar con un número grande de nodos, se decidió actualizar al resto de tiendas cada vez que se realice un cambio de estado. Para facilitar el intercambio de mensajes, se envían únicamente mensajes conteniendo el CRDT completo del producto modificado. Los nodos que reciben este mensaje están encargados de actualizar su estado local con la información recibida. Un punto a destacar sobre este método de actualización de estado es que no garantiza que en todo momento el estado va a estar actualizado, pero reduce la cantidad de mensajes enviados y es suficientemente simple como para no agregar complejidad innecesaria al alcance del caso.

Además, por simplicidad, se consideró que todos los nodos tendrán el mismo catálogo de productos. Se representa cada nodo con su *id* asignado por el PeerServer, el cual también servirá como *id* de la tienda.

Se decidió utilizar como base del proyecto una implementación de un clon del sitio web medium.com que emplea PureScript y React. Esta implementación forma parte de un proyecto comunitario que ofrece distintas versiones del mencionado sitio web con diversas tecnologías de *frontend* y *backend*. Esta elección permitió al equipo contar con una estructura base sólida, asegurar la compatibilidad entre versiones y disponer de comandos de construcción preconfigurados, lo que facilitó centrar los esfuerzos en la construcción del *frontend*. Se utilizó

Yarn [19] como gestor de dependencias y algunos *scripts* utilizando Parcel [20] para generar un ambiente de desarrollo cómodo de utilizar, permitiendo un flujo de compilación y prueba más fluido.

4.3.3. Módulos implementados

Modelo

Los submódulos de Modelo definen los tipos de datos utilizados como parte del modelo dentro de la aplicación. Todos estos módulos implementan funciones auxiliares para serializar y deserializar sus datos haciendo uso de la biblioteca `purescript-yoga-json` [21].

Model.Stock Este módulo ofrece un renombre para el tipo `PNCounter` y crea la fila de tipos `PStock` que representa el stock de un producto.

```
type Contador = PN.PNCounter
type PStock extra = (stock :: Contador | extra)
```

Además define métodos para interactuar con la fila `PStock` aprovechando del polimorfismo de filas. Por ejemplo:

```
vender :: forall p. Record (PStock p) -> Int -> Record (PStock
    p)
vender prod cant = prod { stock = decrement prod.stock cant }
```

El tipo de esta función puede reescribirse de la siguiente manera:

```
vender :: forall (extra :: Row Type)
     . { stock :: Contador | extra }
    -> Int
    -> { stock :: Contador | extra }
```

Es decir, recibe como primer parámetro cualquier registro que contenga al menos el atributo stock de tipo `Contador` y una fila de tipos extra y retorna un registro con los mismos atributos.

Model.Producto Define el tipo Producto como un registro con un identificador y un contador, representado por `PStock`,

[19] https://yarnpkg.com/
[20] https://parceljs.org/
[21] https://pursuit.purescript.org/packages/purescript-yoga-json/5.1.0

```
import Model.Stock (PStock)

type ProductoFields extra =
    (id :: String | PStock extra )
type Producto p = Record (ProductoFields p)
```

En este caso la sintaxis de la fila de tipos `ProductoFields` está compuesta por un id de tipo `String` y extiende los atributos definidos en la fila de tipos `PStock` pasándole la variable de tipos extra para permitir que la fila de tipos Producto sea abierta. Esta declaración se puede simplificar a:

```
type ProductoFields extra = (id :: String, stock :: Contador |
    extra )
```

El módulo también ofrece una función auxiliar para crear un producto con los atributos necesarios. Esta no es necesaria para interactuar con los registros, ya que cualquier registro que contenga los atributos necesarios será un producto válido, pero es de utilidad para el desarrollo ya que facilita posibles cambios de nombres de atributos y reduce la verbosidad del código.

Model.Tienda: Define una tienda de la siguiente manera:

```
type TiendaFields t p = (id :: TiendaId, name :: String,
    products :: ProductMap p | t)
type Tienda t p = { | TiendaFields t p }
type ProductoMap p = Map ProductoId (Producto p)
```

Donde `TiendaFields` es la fila de tipos que define los atributos genéricos que se esperan en una tienda, permitiendo expandirla con la variable de tipos `t`, y el tipo `Tienda` es simplemente un alias para crear el registro correspondiente de manera menos verbosa.

Model.Mensaje Define el tipo `Mensaje` compuesto por un encabezado (*header*) y una carga (*payload*).

```
type Mensaje =
    { header :: HeaderMensaje,
    payload :: PayloadMensaje }
data HeaderMensaje = Merge | Ack
data PayloadMensaje =
    PayloadProduct (Producto ()) | EmptyPayload
```

El tipo `HeaderMensaje` provee dos constructores, el primero, `Merge` indica que el producto enviado en el *payload* es enviado a modo de actualización de estado; el segundo, `Ack`, se utilizó principalmente durante el desarrollo para realizar

pruebas de conexión. El módulo fue diseñado con la intención de agregar otros tipos de mensajes a modo de extender la aplicación. Por ejemplo, para realizar una solicitud de actualización de estado a otra tienda, o para iniciar un traspaso de stock de una tienda a otra, que no fueron implementados en este trabajo.

Interfaz foránea

Los módulos dentro de este grupo están encargados de implementar la FFI con la biblioteca PeerJS. La biblioteca consta con dos elementos principales, `Peer` (nodo par) y `Connection` (conexión a ese nodo), ambos con sus propiedades y distintos *callbacks* configurables para manejar distintos eventos. Se busca representar estos aspectos de la biblioteca limitando a las funciones necesarias para realizar el caso (conectar y desconectar dos nodos, enviar y recibir datos y manejar errores).

Se definen los tipos de datos que representan los elementos de `Peer` y `Connection`

```
foreign import data Peer :: Type
foreign import data Connection :: Type
```

En primera instancia, para obtener un valor de estos tipos se debe tener funciones foráneas que permitan retornar estos valores. Por ejemplo:

```
foreign import _new :: forall config. Record config -> Effect
    Peer
foreign import _connect :: Peer -> PeerId -> Effect Connection
```

Se debe tener sus respectivas implementaciones en JavaScript:

```
export const _new = (config) => () => {
  return new Peer(id, config);
}
export const _connect = (peer) => (peer
ID) => () => peer.connect(peerID);
```

Para acceder a una propiedad de uno de estos elementos, por ejemplo el *id* de un *peer*, se debe utilizar una función auxiliar. En este caso:

```
id :: Peer -> Effect (Maybe PeerId)
id = map toMaybe <<< _id

foreign import _id :: Peer -> Effect (Nullable PeerId)
```

Con su respectiva implementación en JavaScript, donde en este caso `peer.id` retorna `null` en caso de no existir un *id*.

```
export const _id = (peer) => () => peer.id;
```

Finalmente, para configurar los *callbacks*, como la biblioteca provee una única función que recibe distintos parámetros en base a que evento se desea configurar, se implementa una función foránea para cada *callback* que se vaya a soportar. Por ejemplo, para configurar el *callback* que ejecuta al abrirse un nodo:

```
onOpen :: forall m
     . MonadEffect m
    => Peer
    -> (PeerId -> Effect Unit)
    -> m Unit
onOpen peer callback = liftEffect $
    _onOpen peer (Effect.Uncurried.mkEffectFn1 callback)

foreign import _onOpen :: Peer
    -> EffectFn1 PeerId Unit
    -> Effect Unit
```

En JavaScript:

```
export const _onOpen = (peer) => (callback) => () => peer.on("
    open", callback);
```

En este caso el segundo parámetro de la función `_onOpen` es una función que recibe un `PeerId` y ejecuta un efecto. Esto es porque así esta definido el *callback* del evento `open` del objeto `Peer` por la biblioteca original. Se utiliza el módulo `Effect.Uncurried` de manera auxiliar para pasar la función como un único parámetro.

Foreign.PeerId y Foreign.ConnectionId Los tipos `PeerId` y `ConnectionId` son un alias del tipo `String`. Se definen en módulos separados únicamente para resolver dependencias circulares.

Foreign.Peer: Define el tipo `Peer`, que representa un nodo, y `PeerConfig`, que representa los atributos de configuración. Se definen de la siguiente manera:

```
foreign import data Peer :: type
type PeerConfig =
    (host :: String,
    port :: Int,
    path :: String,
    debug :: String)
```

Además, define funciones para crear, abrir, destruir o conectar un nodo (`new` , `awaitOpen`, `connect`, `destroy`, y `connect`, respectivamente) y para configurar los distintos tipos de *callbacks* (en particular, `onOpen`, `onClose`, `onConnection` y `onError`). Estas funciones son implementadas en JavaScript a través de la FFI.

De las antes mencionadas se destaca la función `new`, que utiliza una práctica de programación a nivel de tipos tomada de la biblioteca de `purescript-react-basic-dom`, mencionada en un capítulo anterior (3.3.1).

```
new :: forall m config config_
   . MonadEffect m
  => Union config config_ PeerConfig
  => Maybe PeerId
  -> Record config
  -> m Peer
```

Esta firma de función especifica que el segundo parámetro que recibe la función, de tipo `Record config`, puede tener cualquier subconjunto de los atributos que define la fila de tipos `PeerConfig`. Esto se logra utilizando la restricción `Union` con una variable de tipo (`config_`) la cual es inferida por el compilador con el resto de los atributos de la fila. Este mecanismo permite trabajar con registros con atributos opcionales, donde los valores por defecto los manejan las APIs, lo cual es muy común en bibliotecas de JavaScript.

Foreign.Connection: Este módulo define una representación para una conexión a un nodo. Una conexión contiene un identificador propio y el del nodo remoto asociado.

```
foreign data Connection : Type

id :: Connection -> Maybe ConnectionId
peer :: Connection -> Maybe PeerId
```

Además define funciones para enviar datos, cerrar una conexión (`send` y `close`, respectivamente) y configurar *callbacks* para distintos eventos, `onData`, `onOpen`, `onClose`, `onError` .

Servicios

Los módulos dentro de este grupo se crean con el propósito de agregar una capa de abstracción entre la UI y el modelo, trabajando con el mismo para implementar la lógica de negocio.

Services.HandlerMonad: Este módulo utiliza transformadores de mónadas para crear la mónada `HandlerMonad`. Esta es una combinación de la mónada `State`, `Except` y `Effect`. `State` se usa para guardar un estado local del nodo y un

mapa de tiendas; `Except` para manejar casos borde como pedir una tienda que no existe en el mapa; y, `Effect` para trabajar con efectos secundarios.

La definición de la mónada es la siguiente

```
data HandlerError = NotFound String
type HandlerState t p = { tiendasMap :: TiendasMap t p, peer ::
    Peer }
type TiendasMap t p = Map TiendaId (Tienda t p)
type HandlerMonad a t p =
  ExceptT HandlerError
    (StateT (HandlerState t p) Effect)
    a
```

También provee un método para ejecutar una computación de esta mónada.

```
runHandlerMonad
  :: forall a t p
   . HandlerMonad a t p
  -> TiendasMap t p
  -> Peer
  -> Effect (Either HandlerError (Tuple a (HandlerState t p)))
runHandlerMonad handlerM tiendas peer = do
  (result /\ newState) <- runStateT (runExceptT handlerM) {
      tiendasMap: tiendas, peer: peer }
  pure case result of
    Left err -> Left err
    (Right value) -> Right (value /\ newState)
```

Además define métodos para acceder a tiendas y productos del estado mediante sus identificadores.

```
getTienda :: forall t p. TiendaId -> HandlerMonad (Tienda t p)
    t p
getProduct :: forall t p. Tienda t p -> ProductoId ->
    HandlerMonad (Producto p) t p
```

Services.ConnectionsService: Este módulo junta las funciones relacionadas a la comunicación con el gestor de conexiones y con las tiendas remotas. Extiende el tipo `Tienda` agregando un atributo de conexión para representar el concepto de tiendas remotas y locales, y define funciones para notificar a dichas tiendas. Además, provee una función auxiliar para obtener la lista de nodos del servidor gestor de conexiones. Para simplificar el trabajo se restringe el tipo `Producto` aplicándole una fila vacía.

El tipo de tienda remota se define de la siguiente manera, donde la fila vacía `()` restringe el tipo de `Producto p`:

```
type TiendaRemota = Tienda (conn :: Maybe Connection) ()
```

A modo de ejemplo, para destacar el uso de polimorfismo de filas, se adjunta también las firmas de las funciones para notificar tiendas.

```
notifyTiendasLocales :: forall t. Producto () -> HandlerMonad
    Unit t ()
notifyTiendasRemotas :: forall t. Producto () -> HandlerMonad
    Unit (conn :: Maybe Connection | t) ()
```

Para el caso de tiendas locales se pasa una variable `t` a la `HandlerMonad` que permite extender el tipo de `Tienda t` con cualquier atributo. En el caso de tiendas remotas se pasa una fila abierta que contenga al menos un atributo de tipo `Maybe Connection` con nombre `conn`. Nuevamente, se restringe el tipo de `Producto p` con la fila vacía.

Services.TiendasService: Este módulo provee una abstracción del modelo de tiendas agregando lógica a las funciones que modificarán el stock para notificar mediante el módulo de conexiones, a otras tiendas al momento de alterar el estado. Estas mismas funciones permiten realizar estas modificaciones únicamente usando el id de la tienda y del producto.

```
vender :: forall t. Int -> ProductoId -> TiendaId ->
    HandlerMonad (Producto ()) (conn :: Maybe Connection | t) ()
restock :: forall t. Int -> ProductoId -> TiendaId ->
    HandlerMonad (Producto ()) (conn :: Maybe Connection | t) ()
```

Interfaz gráfica

Contiene todos los módulos relacionados a la implementación de la UI. Todos estos módulos utilizan conceptos de React. Principalmente componentes, *hooks* y contextos.

A continuación se listan algunos ejemplos de *hooks* utilizados en este caso:

- `useState` se utiliza para mantener un estado particular al componente. Recibe un valor inicial y retorna el estado actual y una función para actualizarlo. Actualizar el estado es una forma de volver a renderizar el componente.

- `useEffect` recibe como primer parámetro un valor de tipo polimórfico que representa las dependencias y una función que normalmente interactúa con un sistema externo y retorna una función de "limpieza"(*cleanup*) para deshacer los cambios necesarios. Luego de renderizar el componente que contiene este *hook*, se ejecuta la función pasada por parámetro. Después

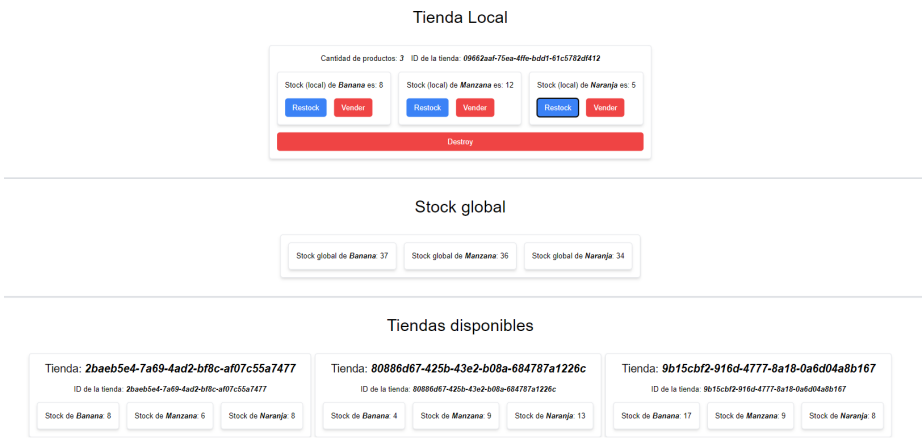

Figura 4.3: Interfaz gráfica

de este renderizado inicial, cada vez que se vuelve a renderizar el componente, se ejecuta primero la función de *cleanup* seguida por la función que representa el efecto, siempre y cuando se haya modificado el valor pasado como dependencia. Finalmente, al destruir el componente, se ejecuta por ultima vez la función de *cleanup*.

- `useMemo` recibe como primer parámetro que representa dependencias y como segundo parámetro una función que será evaluada y guardada hasta que el valor pasado como dependencia cambie.

- `useContext` se utiliza para compartir un conjunto de propiedades a través de distintos componentes anidados sin tener que agregar parámetros extra en cada uno. Se envuelve el componente más alto con un componente llamado *provider* que contiene el contexto inicial. Luego, los componentes hijos pueden acceder a este contexto mediante el *hook* `useContext as`.

UI.Contexts.RootContext Este módulo provee una abstracción para trabajar con el *hook* `Context` de React, un concepto similar al de la mónada `State`. Se elige `Context` sobre la mónada `State` en este caso por su compatibilidad con la biblioteca React, ya que esta es la que decide cuando se renderizan los componentes y por lo tanto cuando se ejecuta su lógica. El módulo define el valor del contexto y provee funciones para crear un componente proveedor de contexto, inicializar un contexto y define su propio *hook*, que a su vez utiliza `useContext`, para evaluar el contexto.

UI.Components Los módulos de este grupo implementan los distintos componentes de la UI.

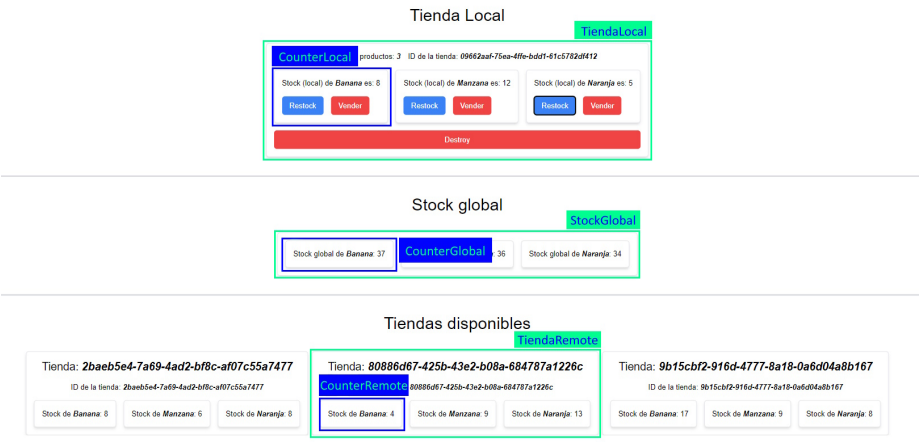

Figura 4.4: Interfaz gráfica - Componentes

En primer lugar se encuentran los módulos `UI.Components.CounterLocal`, `UI.Components.ConterRemote`, `UI.Components.CounterGlobal` que contienen tres tipos de contadores, señalizados en azul en la Figura 4.4. Estos representan los valores del stock de los productos locales, de tiendas remotas y globales, respectivamente. Cada producto en stock se representa con un contador. Para los productos remotos y para los globales se utilizan `CounterRemote` y `CounterGlobal` respectivamente, que despliegan un breve texto y el valor del stock. Para los productos locales se utiliza `CounterLocal`, que además contiene dos botones 'Vender' y 'Restock' para modificar el stock. Cada uno de estos tipos de contador usa un método distinto para acceder al valor del CRDT.

Luego, están los componentes de `TiendaLocal`, `TiendaRemote` y `StockGlobal`, señalizados en verde en la Figura 4.4, que cada uno utiliza uno de los contadores presentados previamente. Para representar la tienda local se utiliza `TiendaLocal`, que contiene información breve de la tienda, un `CounterLocal` por producto y un botón para destruir el nodo. Para representar las tiendas remotas se tiene por cada nodo remoto detectado un componente `TiendaRemota`, con un `CounterRemote` para cada uno de sus productos. Por último, para representar el stock global, se usa un componente `StockGlobal` que contiene un `CounterGlobal` por cada producto del sistema.

UI.Components.App Finalmente, el módulo `App` une a los componentes antes mencionados y además contiene la lógica para inicializar la aplicación. Además del estado de tiendas, que se accede a través del contexto `RootContext` previamente definido, este módulo mantiene un estado de mensajes recibidos.

Para inicializar la aplicación se realizan las siguientes acciones:

- Se inicializa y prepara al `Peer` por primera y única vez al iniciar la aplicación es decir, al cargar la página. Se debe configurar distintos *callbacks*

tanto para el nodo como para sus conexiones. Las conexiones se configuran para que al recibir un nuevo mensaje, este se agregue al estado de mensajes recibidos y para que al cerrarse una conexión, se actualice el estado de tiendas borrando la tienda correspondiente. Al nodo se lo configura para que al recibir una nueva conexión entrante, la configure de la misma manera que se mencionó previamente y que actualice el estado con una nueva tienda. Se utilizan algunos `callbacks` adicionales a modo de imprimir mensajes de información y de error en consola.

- Se obtiene del servidor gestor de conexiones la lista de identificadores conectados al sistema y se establecen conexiones con cada uno de ellos, agregando al estado una tienda para cada conexión. Estas conexiones también se configuran como se mencionó previamente.

- Se procesa la lista de mensajes, actualizando el estado de tiendas utilizando la función `notifyTiendasLocales` del módulo de `Services.ConnectionsService` . Este proceso consiste en actualizar el CRDT para el producto recibido para cada tienda del estado actual y luego actualizar el estado con el nuevo mapa de tiendas.

- Por último, existen algunos *callbacks* de eventos propios de JavaScript asociados a algunos botones de la aplicación. Se agregan a los botones de 'Restock' y 'Vender' de `CounterLocal` el *callback* necesario para representar dichas acciones. Estos *callbacks* se encargan de actualizar el estado local y luego enviar un mensaje con el nuevo estado del producto a cada nodo del sistema, utilizando la función `notifyTiendasRemotas` del módulo `Services.ConnectionsService`. De manera auxiliar también se agrega un botón para destruir al nodo, limpiando el estado.

Finalmente, se destaca que se utilizaron *hooks* para coordinar las distintas acciones. Se adjuntan algunos ejemplos simplificados para analizar:

- Se utilizó `useState` para definir un estado asociado al componente. Por ejemplo, para mantener la lista de mensajes recibidos, se utiliza:

```
messagesByPeer /\ setMessagesByPeer <- useState Nil
```

  El *hook* retorna el estado actual en el momento de ejecución y una función para actualizarlo. Usar esta función le indica a React que debe volver a procesar el componente, lo que permitió interactuar con el estado manteniendo la interfaz gráfica actualizada.

  El uso de este *hook* en JavaScript es muy similar, como se puede ver en el siguiente ejemplo:

```
const [messagesByPeer, setMessagesByPeer] = useState([]);
```

- Se utilizó `useEffect` para procesar la lista de mensajes, marcando como dependencia la propia lista.

```
useEffect (messagesByPeer) do
  if List.null messagesByPeer then log "No hay mensajes
      nuevos"
  else processMessages ctxVal setMessagesByPeer
      messagesByPeer
  pure mempty
```

Al evaluar el *hook*, en caso de que el valor `messagesByPeer` haya sido modificado, se ejecutará la función para procesar los mensajes. La misma se encarga de modificar el estado de mensajes utilizando la función `setMessagesByPeer` pasada por parámetro. De esta manera, la próxima vez que se evalúe el *hook*, se ejecuta la otra opción de la condición. En el caso de no recibir mensajes, como la dependencia no cambia, no se ejecuta el efecto. El efecto retorna la expresión `pure mempty` en este caso porque no hay *cleanup* necesario.

Nuevamente, se adjunta como se escribiría el mismo código en JavaScript:

```
useEffect(() => {
    if ( List.null(messagesByPeer) )
        console.log("No hay mensajes nuevos");
    else
        processMessages(ctxVal, setMessagesByPeer,
            messagesByPeer);
    return () => {}; // pure mempty
}, [messagesByPeer] );
```

Ignorando algunos detalles relacionados a la sintaxis se destaca cómo el uso de este *hook* es muy similar en ambos lenguajes.

- Se utiliza `useMemo` para *memoizar* los valores de `tiendaLocal` y `tiendasRemotas` a partir del mapa de tiendas del estado, tomándolo a este como dependencia. Ejemplo:

```
tiendaLocal <- useMemo (tiendas) \_ ->
  Map.lookup peer.id tiendas
tiendasRemotas <- useMemo (tiendas) \_ ->
  Map.filter (\t -> peer.id /= t.id) tiendas
```

Estos se utilizaron principalmente por motivos de optimización, para no tener que realizar la computación cada vez que se procesa el componente.

Se omite la comparación de código con JavaScript por su simplicidad, pero nuevamente se destaca la similitud.

4.3.4. Conclusiones y Aprendizajes

Se logró implementar un *frontend* que logre interactuar con el modelo de la aplicación utilizando herramientas del ecosistema de JavaScript. También se logró implementar la FFI para trabajar con la biblioteca PeerJS y comunicar distintos nodos.

Al trabajar siguiendo guías y herramientas más actualizadas, se logró un desarrollo más fluido. El uso de Yarn como principal herramienta gestora del proyecto permitió definir comandos útiles para trabajar con el ciclo de vida de la aplicación (compilación, *build* y *deploy*) y facilitó la integración con Parcel para facilitar el ciclo de desarrollo y prueba permitiendo levantar un servidor local y actualizar únicamente el código modificado.

Nuevamente, se destaca la similitud en la estructura de módulos en comparación con lo que se esperaría de un proyecto JavaScript usual. Además, se logró encontrar una similitud entre la estructura del código de los componentes de React y como se escribirían en JavaScript siguiendo las prácticas recomendadas para componentes funcionales. Un claro ejemplo es la similitud con el uso de *hooks* en JavaScript.

El trabajo con la FFI fue intuitivo, permitió integrar la biblioteca fácilmente. Se siguió la guía de buenas prácticas para FFI de PureScript [22]. Aún así, se debe tener especial cuidado al trabajar con una FFI, ya que da lugar a introducir posibles errores difíciles de detectar. Por ejemplo, inicialmente se implementó el tipo `Peer` como un registro con únicamente un atributo de *id* de tipo *String*. Al ser creado y modificado por la biblioteca puramente desde el ambiente de JavaScript, su valor puede ser modificado, en este caso su *id* inicia en un valor nulo y luego es asignado una vez se establece la conexión con el servidor. Esto resulta en que el registro `Peer` no sea un dato inmutable generando posibles comportamientos inesperados. Esto se solucionó usando un tipo foráneo, como es recomendado, y pasando a utilizar una función para acceder al atributo deseado. Se concluye que es responsabilidad del desarrollador de la FFI el asignar los tipos adecuados a las funciones.

El trabajo con registros resultó familiar y cómodo de utilizar en comparación al trabajo con objetos de JavaScript. Un punto en contra a considerar es que al agregar muchos atributos a un registro, este puede resultar muy verboso, lo cual puede volverse un problema al momento de analizar errores de tipos. El trabajo con polimorfismo de filas resultó práctico, permitiendo extender tipos (por ejemplo el tipo Tienda) fácilmente y definir funciones que trabajen con estos. Para el polimorfismo también aplica la verbosidad de la sintaxis. Para distintas combinaciones de atributos se debe optar entre definir aliases para combinaciones especificas o volver a escribir estas combinaciones en las firmas de las funciones.

[22] https://github.com/purescript/documentation/blob/master/guides/FFI-Tips.md

Esta vez se tomó un enfoque más proactivo para buscar soporte con el uso del lenguaje. Una vez más se destaca la ayuda de la comunidad, principalmente para el trabajo con la FFI y la biblioteca de React.

Capítulo 5

Conclusiones

En este trabajo de grado se lograron cumplir los objetivos planteados. El objetivo principal de investigar, analizar y aprender acerca de PureScript fue cumplido con éxito, habiendo alcanzando un entendimiento de las características fundamentales del lenguaje. Como parte de los resultados esperados, se realizó un relevamiento detallado de las principales características de PureScript, lo que permitió comprender sus capacidades y limitaciones. A lo largo de la investigación, se estudiaron las capacidades de interoperabilidad de PureScript con JavaScript, un aspecto crucial del estudio.

Además, la comparación con Haskell permitió resaltar diferencias y similitudes clave con dicho lenguaje. Por otro lado, la comparación con TypeScript mostró cómo ambos lenguajes abordan los desafíos del tipado en JavaScript y cómo el código TypeScript tiene una gran similitud a JavaScript, mientras que el código PureScript se asemeja mucho al de Haskell en su sintaxis.

Los casos de uso explorados fueron fundamentales para consolidar este entendimiento práctico, aplicando, por ejemplo, las primitivas de PureScript para la programación web, el trabajo con registros y polimorfismo de filas, y la interacción con JavaScript a través de la FFI, lo que permitió alcanzar una comprensión sólida tanto de los fundamentos del lenguaje como de su pragmática, logrando así otro de los resultados previstos.

5.1. Dificultades encontradas

En cuanto a las dificultades encontradas durante el uso de PureScript, uno de los problemas fue la baja precisión del *plugin* recomendado para Visual Studio Code al señalar el origen de los errores. En muchas ocasiones, el *plugin* marca la totalidad de una línea de código como errónea sin dar detalles específicos sobre el origen exacto del problema. Esta falta de información dificultaba la identificación del error y por lo tanto la velocidad para solucionarlo. Sumado a esto, la verbosidad del trabajo con registros dificultó la comprensión de los errores usando dicho *plugin*.

Por otro lado, el tamaño pequeño de la comunidad de PureScript representó un desafío significativo para el proyecto, especialmente en términos de acceso a recursos y soluciones documentadas. Las búsquedas tradicionales en plataformas como StackOverflow o Google no resultaron útiles, lo que obligó a recurrir a recursos más específicos. Se exploraron canales dedicados a PureScript, como Discourse [1] y un canal en Discord [2], siendo este último especialmente útil debido a su alta actividad.

Asimismo, la reducida comunidad también afecta la calidad y cantidad de documentación disponible, así como el mantenimiento de las bibliotecas existentes. En varias ocasiones, el equipo se encontró con bibliotecas que parecían muy útiles, pero al no tener documentación ni ejemplos claros de como utilizarlas, se requirió un esfuerzo extra considerable para usarlas. Otras estaban muy desactualizadas (la extensa mayoría no había recibido ninguna actualización en los últimos dos años), lo que generó complicaciones al intentar integrarlas con el resto de las dependencias del proyecto.

En resumen, la comunidad y la documentación de un lenguaje juegan un rol fundamental en la experiencia de aprendizaje y resolución de problemas. En el caso de PureScript, debido a su baja popularidad, estos factores presentaron un reto para el equipo, especialmente desde la perspectiva de quienes se están iniciando en el lenguaje, algo que no se había experimentado al aprender otros lenguajes.

5.2. Resumen

La compatibilidad con JavaScript es sin duda el aspecto más destacado de PureScript, diferenciándolo notablemente de otros lenguajes de programación funcional como Haskell. Esta característica ofrece varios beneficios prácticos que hacen que PureScript sea atractivo en el desarrollo moderno, entre ellos:

- **Solución al tipado dinámico de JavaScript**: Uno de los principales problemas de JavaScript es su sistema de tipado dinámico. PureScript, al ser un lenguaje funcional, con un sistema de tipos estático y seguro, resuelve este problema proporcionando una mayor seguridad en tiempo de compilación. Esto permite detectar errores comunes de tipado antes de ejecutar el código, mejorando la robustez de las aplicaciones.

- **Interoperabilidad con el ecosistema de JavaScript**: La capacidad de PureScript para interactuar con bibliotecas y módulos de JavaScript permite a los desarrolladores aprovechar el gran ecosistema existente de JavaScript. Es crucial realizar un buen trabajo tipando las funciones importadas de JavaScript para evitar problemas durante el desarrollo. Esto es sumamente necesario para asegurarse que el código mantenga la seguridad y robustez que ofrece PureScript, pero también puede añadir complejidad,

[1] https://discourse.purescript.org/
[2] https://purescript.org/chat

especialmente en proyectos grandes o cuando se trabaja con bibliotecas de JavaScript con tipos difíciles de definir.

- **Adopción incremental**: Otra ventaja importante es que PureScript permite una adopción gradual. No es necesario reescribir todo el código de un proyecto de una sola vez; se puede empezar a usar PureScript en módulos pequeños y coexistir con el código JavaScript existente. Esta posibilidad reduce los riesgos asociados con el cambio de tecnología y permite experimentar con la programación funcional sin comprometer la estabilidad del proyecto.
- **Aplicación tanto en frontend como en backend**: Gracias a la interoperabilidad con JavaScript, PureScript puede utilizarse tanto en el desarrollo *frontend* como *backend*, a través de Node.js, permitiendo la construcción de aplicaciones web completas utilizando un único lenguaje funcional. Además, PureScript cuenta con compiladores destinados a otros lenguajes de *backend*, como C [3] y Erlang [4]. Sin embargo, es importante destacar que dichos compiladores no han sido actualizados en los últimos años, lo que podría representar una limitación en términos de estabilidad y soporte para proyectos a largo plazo.
- **Un lenguaje puramente funcional para la web**: PureScript permite llevar la programación funcional al desarrollo web con todos los beneficios que este paradigma brinda.

Otra característica distintiva y atractiva de PureScript es su soporte para el polimorfismo de filas. Esta brinda una gran flexibilidad y facilita la reutilización de código, lo que resulta especialmente útil en contextos donde la adaptabilidad es crucial, como la serialización de datos, la validación de formularios y la construcción de APIs.

Aunque PureScript ofrece varias ventajas, su adopción masiva se enfrenta a ciertos desafíos. Tanto en la industria como en el desarrollo académico, prevalece mayormente el trabajo con lenguajes del paradigma imperativo sobre el funcional. En el contexto del desarrollo *frontend*, lenguajes como JavaScript, que a pesar de ser multiparadigma suele ser utilizado de forma imperativa, o lenguajes sintácticamente similares son predominantes. Esto lleva a que el cambio de paradigma tenga un mayor impacto en la curva de aprendizaje.

Dicho esto, se consideraran dos perspectivas a fin de evaluar la utilización de PureScript: la de un desarrollador familiarizado con JavaScript y la de un desarrollador familiarizado en Haskell.

5.2.1. Perspectiva desarrollador JavaScript

Los errores de tipado en PureScript pueden resultar particularmente difíciles de entender. En comparación con JavaScript o TypeScript, los mensajes de error

[3] https://github.com/pure-c/purec
[4] https://github.com/purerl/purerl

tienden a ser menos intuitivos, lo que añade una capa adicional de complejidad al proceso de desarrollo.

La falta de experiencia en lenguajes de programación funcional, sumada a las diferencias sintácticas y la dificultad para comprender los errores de tipado, son factores clave que contribuyen a la pronunciada curva de aprendizaje de los lenguajes funcionales. Como resultado, desde esta perspectiva, la velocidad del desarrollo al utilizar PureScript tiende a ser más lenta en comparación con lenguajes imperativos. Este es un factor importante al considerar la adopción de PureScript en contextos donde existan plazos de entrega.

A pesar de que PureScript aborda problemas clave de JavaScript, actualmente su adopción es muy limitada. Esto se debe, en parte, a que una gran parte de la comunidad de desarrolladores de JavaScript tiene escasa familiaridad con la programación funcional y encuentran desafiante la transición hacia la naturaleza estrictamente tipada y estructurada de PureScript, la cual contrasta con la flexibilidad de JavaScript.

Además, las ventajas de usar programación funcional en el contexto del desarrollo web no están bien investigadas ni documentadas, lo que podría llevar a que los desarrolladores no consideren este paradigma al iniciar un nuevo proyecto (Salmi, 2020).

Como consecuencia, PureScript no ha logrado una adopción generalizada dentro de la comunidad de programadores de JavaScript. Muchos desarrolladores que buscan resolver problemas relacionados con el tipado dinámico prefieren herramientas o lenguajes que requieran menos capacitación y cambios en su forma de pensar, como por ejemplo TypeScript. Este se ha consolidado como la opción preferida, ya que ofrece una transición más suave hacia un sistema de tipos más seguro sin exigir un cambio radical en la programación. Su flexibilidad, junto con el respaldo de grandes empresas, una comunidad considerablemente más amplia y la capacidad de implementar técnicas funcionales, ha relegado a PureScript a un nicho especializado y menos accesible para la mayoría de los desarrolladores

No obstante, PureScript es una opción muy atractiva para aquellos desarrolladores de JavaScript que quieran remediar algunos de los problemas clave de dicho lenguaje y estén abiertos a explorar un lenguaje funcional, especialmente debido a su buena interoperabilidad con JavaScript.

5.2.2. Perspectiva desarrollador Haskell

Trabajar dentro del paradigma funcional, utilizando una sintaxis similar a la de Haskell, ofrece una experiencia familiar para aquellos desarrolladores con conocimiento previo en dicho lenguaje. No obstante, existen algunas particularidades que requieren ser tenidas en cuenta. Algunas características, como el uso de evaluación estricta en lugar de evaluación perezosa, las pequeñas diferencias de sintaxis o las diferencias en *namespaces* y nombres de funciones dentro de las bibliotecas comunes, hacen que las soluciones no sean directamente trasladables de Haskell a PureScript. Dejando estas características de lado, el tipado estático, el trabajo con un sistema de tipos relativamente similar, tipos de datos, tipos

de clases, polimorfismo permite un fácil adentramiento al lenguaje sin impactar negativamente la velocidad de desarrollo.

Por otro lado, el trabajo con registros es más práctico teniendo el acceso a los atributos como operaciones de primera clase; el polimorfismo de filas también permite soluciones interesantes y reutilizables. Aún así, el trabajo con registros resulta verboso, lo cual puede ser considerado una desventaja. Se puede tomar como ejemplo el pasaje de Eff a Effect, donde se pasó de utilizar una fila de tipos para representar los tipos de efectos, a utilizar simplemente la mónada, principalmente optando por claridad en el código. Un ejemplo de esto es presentado en el primer caso de estudio 4.1. Otro aspecto donde se considera un inconveniente es en los mensajes de error relacionados a tipos que contienen registros, ya que estos resultan muy verbosos, complejizando su comprensión.

La introducción al ecosistema de JavaScript es un aspecto interesante que ofrece PureScript. Permite el acceso a un amplio conjunto de bibliotecas y además existen muchas herramientas que facilitan el desarrollo de un proyecto. Al ser compilado a JavaScript, el código generado resulta portable entre navegadores, permitiendo además fácil acceso a características nativas de los mismos como por ejemplo el trabajo con WebRTC o el Modelo de Objetos del Documento en HTML (DOM). También se debe tener en cuenta que el ecosistema de JavaScript es altamente cambiante, lo cual ha generado repercusiones en PureScript en el pasado, como lo fue la migración de módulos CJS a ESM.

5.3. Reflexiones finales

PureScript es una opción interesante para escribir código funcional e interactuar con el ecosistema JavaScript, por ejemplo construyendo aplicaciones web.

Desde la perspectiva de un programador Haskell, PureScript probablemente sea la mejor alternativa para quienes estén interesados en el desarrollo web, dado que ambos comparten una sintaxis muy similar. Más allá de la sintaxis, el lenguaje consta con cualidades similares, permitiendo una transición relativamente sencilla; logra mantener un alto nivel de abstracción, permitiendo reusabilidad en las bibliotecas y permite aplicar varios de los conceptos característicos, tanto simples como avanzados, de la programación funcional. El trabajo con registros como elementos nativos del lenguaje es práctico, y el uso de polimorfismo de filas permite llegar a soluciones interesantes. Además, el acercamiento al ecosistema de JavaScript abre las puertas a tecnologías especificas del mismo.

Por otro lado, para un programador JavaScript, creemos que hay opciones más atractivas.

Como se ha mencionado previamente, PureScript enfrenta desafíos importantes que dificultan su adopción en la industria, los cuales deben ser considerados al evaluar su viabilidad en proyectos de software. Algunos de estos problemas son la escasa documentación, el limitado mantenimiento de las bibliotecas y la reducida cantidad de profesionales con experiencia en el lenguaje, todos ellos muy vinculados al tamaño reducido de su comunidad de usuarios. Esta situación

genera un "círculo vicioso": la baja cantidad de desarrolladores activos dificulta la actualización y el mantenimiento de las bibliotecas y de la documentación, lo que, a su vez, desincentiva la adopción del lenguaje por parte de nuevos usuarios. Además, el equipo considera, basándose en su propia experiencia, que capacitar a desarrolladores en PureScript no es fácil con los recursos actuales.

Por lo tanto, al menos al momento de escribir esta tesis, opciones como TypeScript, tomando un enfoque funcional, son preferibles desde nuestro punto de vista. Se recomienda utilizar una configuración estricta de tipos, así como el *plugin* presentado la Sección 2.2, para adherirse a los principios de la programación funcional.

Cabe destacar que estos problemas no son exclusivos de PureScript. Por ejemplo, en una tesis que compara el desarrollo de aplicaciones web utilizando Elm, un lenguaje puramente funcional, y JavaScript aplicando técnicas de programación funcional, se identifican las mismas limitaciones en cuanto al acceso a documentación y al mantenimiento de las bibliotecas. En dicho estudio llegan a conclusiones muy similares a las nuestras: a pesar de que Elm es una buena alternativa para el desarrollo web, la cantidad de bibliotecas desactualizadas y la falta de marcos de trabajo adecuados limitan su viabilidad para proyectos de gran porte. Por lo tanto, también concluyen que Elm aún no está listo para un uso amplio en proyectos grandes debido a estas carencias, y sugieren que, por el momento, es preferible adoptar un enfoque funcional utilizando lenguajes multiparadigma como TypeScript, que ofrece tipado estático y un ecosistema más maduro y con mayor soporte. (Salmi, 2020).

Finalmente, el hecho de que PureScript haya permanecido en la versión 0.x.x desde su creación en 2013 es un indicio de que sus desarrolladores aún no lo consideran lo suficientemente maduro para una versión 1.0. Hace dos años, el propio creador del lenguaje mencionó que "PureScript continuará teniendo cambios disruptivos hasta alcanzar la versión 1.0, pero ese hito está cada vez más cerca. Las cosas se están estabilizando, y no se esperan muchos cambios importantes entre la versión 0.12.0 y 1.0". [5] Sin embargo, la actualización 0.15 de PureScript tuvo cambios importantes en su ecosistema. Al no soportar la sintaxis de JavaScript CJS, requirió que todas las bibliotecas se actualicen a ESM. Debido a que muchas de ellas no tienen mantenimiento esta adaptación no fue realizada y eso complica su uso.

Además, desde dicha publicación no hay noticias que sugieran que la versión 1.0 esté en camino, lo que genera incertidumbre para proyectos a largo plazo.

Es importante aclarar que la versión de un software no siempre refleja directamente su estabilidad o capacidad para ser usado en producción. No obstante, el hecho de que PureScript siga en una versión 0.x.x implica riesgos asociados a cambios futuros que podrían complicar el mantenimiento y la evolución de proyectos que dependan de este lenguaje.

A pesar de estos riesgos, PureScript es utilizado por diferentes organizaciones en entornos de producción.

[5] https://gist.github.com/paf31/adfd15fbb1ac8b99fc68be2c9aca8427#purescript-changes-too-quickly

5.4. Posibles Extensiones al Trabajo

- Una posible extensión de este trabajo podría consistir en desarrollar funcionalidades adicionales para las bibliotecas de CRDT y PeerJS creadas en los casos de estudio, con el objetivo de publicarlas como paquetes de PureScript y ponerlas a disposición de la comunidad. Sin embargo, se decidió no llevar a cabo esta publicación en la etapa actual, ya que se consideró agregar más funcionalidades a estas bibliotecas antes de compartirlas. Además, el equipo no estaba dispuesto a asumir el compromiso de mantener estas bibliotecas actualizadas, lo que probablemente habría resultado en que quedaran obsoletas en el corto plazo. Esta decisión se fundamenta en una de las dificultades identificadas durante el desarrollo del proyecto de grado: la gran cantidad de bibliotecas sin mantenimiento, por lo que al equipo no la pareció adecuado contribuir al problema.

- Un aspecto interesante para investigar a futuro es la migración de un proyecto de gran porte en React a PureScript. En (Honeyman, 2020) se detalla un enfoque que no solo demuestra cómo los componentes escritos en JavaScript o TypeScript pueden coexistir con aquellos desarrollados en PureScript, sino que también facilita una migración incremental y progresiva del código. Este enfoque podría ser de gran valor para la industria, ya que permite mejorar gradualmente la estabilidad y mantenibilidad de las aplicaciones sin la necesidad de reescribir todo el proyecto desde cero reduciendo los riesgos asociados a ello.

  El autor menciona haber participado en la realización exitosa de migraciones de este tipo en diversas empresas, las cuales experimentaron una disminución significativa de errores y una mayor mantenibilidad en sus aplicaciones.

- Otra posibilidad es explorar la otra alternativa que consideramos para construir la interfaz de usuario del caso de estudio tres, Halogen. Esta biblioteca esta implementada completamente en PureScript y adopta un enfoque declarativo y basado en componentes, similar a `purescript-react`. La extensión al trabajo consistiría en desarrollar una aplicación *frontend* utilizando Halogen, ya sea replicando la interfaz de usuario implementada en el caso de estudio mencionado o aplicando la biblioteca a un nuevo caso de estudio. Esto permitiría realizar una comparación de ambas bibliotecas, evaluando sus respectivas ventajas, simplicidad y limitaciones. Además, proporcionaría una visión más completa sobre las alternativas que ofrece PureScript para el desarrollo de aplicaciones *frontend*.

- Por otro lado, una iniciativa interesante sería el desarrollo de actividades de formación y difusión del lenguaje. Esto puede incluir la organización de cursos, talleres, charlas en *meetups* o eventos orientados a desarrolladores interesados en aprender sobre PureScript. Estas iniciativas ayudarían a aumentar la visibilidad del lenguaje, el acceso a recursos educativos y un fortalecimiento de la comunidad. Por lo tanto esto contribuiría en abordar,

en cierta medida, algunas de las dificultades identificadas en este estudio, particularmente en relación con la falta de recursos y la baja adopción del lenguaje.

- Otra idea es el desarrollo de una extensión para PureScript que permita incluir una sintaxis similar a JSX/TSX [6]. Esto facilitaría la escritura de etiquetas HTML dentro de archivos PureScript, lo que simplificaría el desarrollo de aplicaciones web utilizando PureScript y React. De esta forma se ofrecería una experiencia de desarrollo más accesible y familiar para desarrolladores JavaScript, la cual combinada con las garantías de tipado de PureScript, puede fomentar la adopción de PureScript en proyectos de *frontend*.

- Otra propuesta es implementar una aplicación de gran escala utilizando PureScript y estudiar como se comporta. Para esto se podría continuar el desarrollo de la aplicación construida en el caso de estudio tres, o bien implementar una nueva. Este enfoque permitiría profundizar en la implementación de una aplicación más compleja y de mayor escala, lo que ofrecería una perspectiva más amplia sobre el comportamiento de PureScript en proyectos reales.

- Otra posible extensión inspirada en (Salmi, 2020) consiste en el estudio de las diferencias de rendimiento y la cantidad de errores entre PureScript y otros lenguajes de programación, como JavaScript, TypeScript y Haskell. De manera similar a dicho trabajo, este estudio compararía aplicaciones equivalentes desarrolladas en los distintos lenguajes para evaluar no solo su rendimiento, sino también la facilidad de desarrollo, de testear y la mantenibilidad en el contexto de aplicaciones web modernas. Estos resultados serían muy útiles para quienes busquen elegir el mejor lenguaje para sus proyectos basados en datos empíricos.

[6] https://es.react.dev/learn/writing-markup-with-jsx

Referencias

Anton Ioffe. (2023). *Reader Mode.* https://borstch.com/blog/lazy-evaluation-techniques-in-javascript. (Accessed: 2024-02-06)

Banks, A., y Porcello, E. (2020). *Learning react: Modern patterns for developing react apps.* Sebastopol, CA: O'Reilly Media, Inc.

Barendregt, H. (1984). *The lambda calculus, its syntax and semantics.* North-Holland.

Daigle, K., y Staff, G. (2023). *Octoverse: The state of open source and rise of ai in 2023.* https://github.blog/2023-11-08-the-state-of-open-source-and-ai/. (Accessed: 2024-02-08)

Gordon, C. S., Parkinson, M. J., Parsons, J., Bromfield, A., y Duffy, J. (2012). Uniqueness and reference immutability for safe parallelism. En *Proceedings of the acm international conference on object oriented programming systems languages and applications* (p. 21–40). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2384616.2384619

Honeyman, T. (2020). *Replace react components with purescript.* https://thomashoneyman.com/articles/replace-react-components-with-purescript/. (Accessed: 2024-08-07)

Hughes, J. (1989, 01). Why Functional Programming Matters. *The Computer Journal*, *32*(2), 98-107. doi: 10.1093/comjnl/32.2.98

Jensen, S. H., Møller, A., y Thiemann, P. (2009). Type analysis for javascript. En J. Palsberg y Z. Su (Eds.), *Static analysis* (pp. 238–255). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kauffman, F. (2023). *Crdts and distributed consistency: Part 1 - the simple distributed counter.* Descargado de https://www.streaver.com/blog/posts/crdts-and-distributed-consistency-1-the-simple-distributed-counter (Accessed: 2024-02-04)

Marlow, S. (2013). *Parallel and concurrent programming in haskell: Techniques for multicore and multithreaded programming.* Sebastopol, CA: O'Reilly Media.

Martinez, J. (2024). *Jordan's reference site.* Descargado de https://jordanmartinez.github.io/purescript-jordans-reference-site/ (Accessed: 2024-08-08)

McCarthy, J. (1960, apr). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, *3*(4), 184–195. doi: 10.1145/367177.367199

MDN Web Docs Contributors. (2023). *Dynamic typing.* https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing. (Accessed: 2024-03-11)

Mena, A. (2019). *Practical haskell: A real world guide to programming.* Apress.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, *17*(3), 348–375.

Milner, R., Tofte, M., y Harper, R. (1990). *The definition of standard ml.* MIT Press.

Nilsson, H., y Fritzson, P. (1994). Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, *4*(3), 337–369. doi: 10.1017/S095679680000109X

Paulson, L. C. (1996). *Ml for the working programmer* (2.ª ed.). Cambridge University Press.

Phil Freeman. (2023). *Purescript by example.* https://book.purescript.org. (Accessed: 2024-01-06)

PureScript Contributors. (2023). *PureScript Language Documentation: Types.* https://github.com/purescript/documentation/blob/master/language/Types.md. (Accessed: 2024-03-29)

PureScript Contributors. (2024). *Getting started guide.* Descargado de https://github.com/purescript/documentation/blob/master/guides/Getting-Started.md (Accessed: 2024-07-14)

Rajchert, H. (2019). *Typescript vs. purescript.* https://blog.logrocket.com/typescript-vs-purescript-not-all-compilers-are-created-equal-c16dadaa7d3e/. (Accessed: 2024-05-27)

Salmi, E. (2020). *Comparing the use of a purely functional programming language to event-driven javascript in modern web application development* (Tesis Doctoral, Tampere University). Descargado de https://trepo.tuni.fi/handle/10024/121287

Sarkar, A. (2020). *Creating a simple rest service in purescript.* Descargado de https://abhinavsarkar.net/posts/ps-simple-rest-service/ (Accessed: 2024-07-24)

Shapiro, M., Preguiça, N., Baquero, C., y Zawirski, M. (2011). Conflict-free replicated data types. En X. Défago, F. Petit, y V. Villain (Eds.), *Stabilization, safety, and security of distributed systems* (pp. 386–400). Berlin, Heidelberg: Springer Berlin Heidelberg.

Thompson, S. (1991). *Type theory and functional programming.* Addison Wesley.

Thompson, S. (2011). *Haskell: The craft of functional programming* (3.ª ed.). Addison-Wesley.

Trotter, M. (2024). *Type system showdown - purescript and typescript.* Descargado de https://discourse.purescript.org/t/type-system-showdown-purescript-and-typescript/2084/4 (Accessed: 2024-05-31)

Turner, D. A. (1985). Miranda: A non-strict functional language with polymorphic types. En J.-P. Jouannaud (Ed.), *Functional programming languages and computer architecture* (Vol. 201, pp. 1–16). Berlin, Heidelberg: Springer. doi: 10.1007/3-540-15975-4_26

TypeScript Documentation. (2024). *Typescript in 5 minutes: Structural type system.* Descargado de https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html#structural-type-system (Accessed: 2024-05-31)

Wirfs-Brock, A., y Eich, B. (2020, jun). Javascript: the first 20 years. *Proc. ACM Program. Lang.*, *4*(HOPL). doi: 10.1145/3386327

Anexo A

Caso de estudio 0 - *Getting Started*

El caso de estudio cero se centra en explorar y describir el conjunto de herramientas necesarias para configurar un entorno de desarrollo adecuado para la realización de proyectos con PureScript, que permita tanto la compilación eficiente del código como la gestión conveniente de paquetes y dependencias. Este procura evitar la redundancia de tener que explicar este procedimiento inicial en los casos de estudio subsiguientes, optimizando así el flujo de trabajo y la comprensión de los mismos. Parte de este caso se basa en la guía proporcionada en (PureScript Contributors, 2024).

Las herramientas requeridas para trabajar con PureScript son las siguientes:

- **purs**: el compilador de PureScript, necesario para transformar el código fuente en JavaScript ejecutable.
- **spago**: el gestor de paquetes de PureScript, que facilita tanto la instalación de bibliotecas externas como la gestión de dependencias del proyecto.
- **PSCi**: el intérprete REPL de PureScript. Permite la ejecución de código PureScript de manera inmediata para pruebas rápidas o aprendizaje interactivo.

Como prerrequisito se requiere la instalación de Node.js y un gestor de paquetes como Yarn o Npm, los cuales son necesarios para la instalación de las herramientas antes mencionadas. A continuación, se detallan los pasos para la instalación y configuración inicial del entorno:

A.1. Creación y configuración de un proyecto

Para iniciar un proyecto, se usará un directorio especifico para el mismo y se iniciará como un proyecto Npm o Yarn habitual en el cual se instalarán

las herramientas Spago y Purs; para este ejemplo se utilizará Npm. Con el comando `npm init` se creará un archivo `package.json`, necesario para la gestión de paquetes JavaScript en el proyecto. Luego, se utilizará Spago para inicializar el proyecto PureScript. Esto va a generar la estructura de directorios estándar para proyectos de PureScript, la cual incluye la carpeta `.spago` donde se almacenarán los paquetes instalados.

```
mkdir getting-started
cd getting-started

npm init -y

npm install purescript spago

# Verificar que se hayan instalado correctamente
npm exec purs --version
npm exec spago --version

npm exec spago init
```

A.2. Instalación de paquetes

Teniendo un proyecto de Npm inicializado, se pueden instalar paquetes de JavaScript, como `purescript-foldable-traversable`, mediante el comando `npm install purescript-foldable-traversable --save`.

Luego, para instalar bibliotecas en el proyecto de PureScript, se utiliza Spago ejecutando el comando `spago install` seguido del nombre de la biblioteca, como se muestra en el siguiente ejemplo:

```
npm exec spago install foldable-traversable
```

A.3. Uso del intérprete

Para interactuar con el código PureScript de manera inmediata, se puede ejecutar el comando `spago repl`

Por defecto no se importa ningún módulo, pero se puede crear un archivo `.purs-repl` donde se definen instrucciones para correr al inicio del intérprete; donde normalmente se importan módulos.

A.4. Ejemplo de aplicación

Se presenta un ejemplo práctico donde se resuelve un problema utilizando PureScript. El objetivo es encontrar la suma de todos los múltiplos de 3 o 5 por debajo de 1000

```
module Euler where
```

```
import Prelude

import Data.List (range, filter)
import Data.Foldable (sum)

ns = range 0 999

multiples = filter (\n -> mod n 3 == 0 || mod n 5 == 0) ns

answer = sum multiples
```

Para probar la solución, lo primero que hay que hacer es compilar el proyecto. Esto se hace con el comando `spago build`, lo cual generará el directorio `output` con el código compilado y mostrará advertencias o errores de ser necesario, como puede ser en este ejemplo, por falta de tipos explícitos en las funciones.

Luego, se puede usar el intérprete (`spago repl`) para invocar funciones de manera interactiva. El uso del intérprete invoca al compilador implícitamente.

Una vez terminado el desarrollo, para facilitar la puesta en producción se puede utilizar la funcionalidad `spago bundle-app` para generar un único archivo JavaScript que contenga el código de nuestro proyecto. Este proceso, además, se encarga de remover funciones no utilizadas para minimizar el tamaño del código generado. Versiones más nuevas de spago ofrecen configuraciones para adaptarse a distintos casos, como crear módulos o optimizar el código generado para distintos ambientes de producción.

A.5. Configuración avanzada - Recomendaciones

Por último, se agregan algunos conceptos y configuraciones extra que resultaron útiles para desarrollar a lo largo de este proyecto.

Se trabajó principalmente desde el IDE Visual Studio Code [1] con los *plugins* PureScript IDE [2] y PureScript Language Support [3] que ofrecen soporte de lenguaje, detección y resaltado de errores mediante la interacción con spago y acciones automáticas para trabajar con dependencias.

Existe un repositorio [4] mantenido por los desarrolladores y la comunidad de PureScript que provee conjuntos de dependencias (*package sets*) compatibles entre sí en sus versiones y con determinadas versiones de PureScript. Trabajar con estos conjuntos de dependencias en altamente recomendable ya que provee fácil acceso a dependencias básicas comúnmente utilizadas (como lists, aff, etc.) además de otras más específicas (como mysql, postgresql, etc) y facilita la instalación de las mismas sin tener que buscar versiones compatibles entre ellas.

Para incluir un package-set en un proyecto se debe agregar al archivo `packages.dhall` generado por `spago init`.

[1] https://code.visualstudio.com/
[2] https://marketplace.visualstudio.com/items?itemName=nwolverson.ide-purescript
[3] https://marketplace.visualstudio.com/items?itemName=nwolverson.language-purescript
[4] https://github.com/purescript/package-sets