UNIVERSIDAD
COMPLUTENSE
MADRID

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TRABAJO DE FIN DE GRADO

Una implementación en Haskell de un lenguaje funcional no determinista

Grado en Ingeniería Informática

Autor:
Manuel Velasco Suárez

Profesor director:
Francisco Javier López Fraguas

Índice

1. Resumen

En este trabajo se propone e implementa un lenguaje funcional que incorpora características habituales de este paradigma, como es el orden superior, ajuste de patrones o evaluación perezosa. Añade características no tan habituales como es el indeterminismo, expresado mediante funciones no deterministas, es decir, funciones que para unos argumentos dados pueden devolver más de un resultado. Este tipo de funciones existen en los llamados lenguajes lógico funcionales, que adoptan para el indeterminismo una semántica de compartición (call-time choice), en contraste con la semántica de no compartición (run-time choice) más típica de otros formalismos, como son los si temas de reescritura.

Lo específico del lenguaje nuevo que se va a proponer e implementar, *Sharade*, es que combinará ambos tipos de semánticas mediante primitivas que existirán en el propio lenguaje.

La implementación del lenguaje está realizada íntegramente en Haskell, en contraste con las implementaciones clásicas de los lenguajes lógico funcionales, habitualmente basadas en Prolog. Haskell es un lenguaje funcional puro con un sistema de tipos muy fuerte, siendo el lenguaje de referencia en el ámbito de la programación funcional con evaluación perezosa. Aparte de fases auxiliares como son el análisis sintáctico y la inferencia de tipos, lo esencial de la implementación consiste en un proceso de traducción de programas fuente Sharade en programas objeto Haskell, aprovechando así muchas de las características de este último como lenguaje funcional. Los programas objeto hacen uso intensivo de programación monádica, a través en particular de una biblioteca Haskell para la programación con indeterminismo con call-time choice y evaluación perezosa, biblioteca ya existente pero que ha debido ser adaptada por nosotros para actualizarla a las nuevas versiones de Haskell.

Palabras clave

Programación Funcional, No determinismo, Curry, Toy, Sharade, Compartición, Call-time Choice, Run-time Choice, Mónadas, Hindley-Milner

2. Abstract

This paper proposes and implements a functional language that incorporates common features of this paradigm, such as higher order functions, pattern matching or lazy evaluation. It adds a not so common feature, namely non-determinism, expressed through non-deterministic functions, that is, functions that for given arguments can return more than one result. This type of functions exist in the so-called functional logic languages, which adopt for indeterminism a semantics of sharing (call-time choice), in contrast with the semantics of not sharing (run-time choice) more typical of other formalisms, such as rewriting systems.

The specific thing about the new language that will be proposed and implemented, *Sharade*, is that it will combine both types of semantics through primitives that will exist in the language itself.

The implementation of the language is done entirely in Haskell, in contrast to the classic implementations of functional logic languages, usually based on Prolog. Haskel is a pure functional language with a very strong type system, being the reference language in the field of functional programming with lazy evaluation. Apart from auxiliary phases such as syntactic analysis and type inference, the essential part of the implementation consists of a process of translating Sharade source programs into Haskell object programs, thus enabling many of the latter's features as a functional language. The object programs make intensive use of monadic programming, in particular through a Haskell library for programming with indeterminism with call-time choice and lazy evaluation, library already existing but that has had to be adapted by us to update it to new versions of Haskell.

Keywords

Functional programming, non determinism, Curry, Toy, Sharade, Sharing, Call-time Choice, Run-time Choice, Monads, Hindley-Milner

3. Introducción

Este documento es parte del resultado de un año lectivo de trabajo para concluir el trabajo de fin de grado de la carrera de Ingeniería Informática en la Universidad Complutense de Madrid. En este documento abordo la propuesta e implementación de un pequeño y simple lenguaje funcional con características habituales en lenguajes funcionales como el orden superior y ajuste de patrones, a las que se añade otra no habitual en este paradigma, pero sí en una extensión de él, la denominada programación lógico funcional [1], que combina características de los paradigmas independientes de programación lógica y funcional. Del primero adopta en particular la posibilidad de cómputos indeterministas que se encuentran sustentados en los lenguajes más conocidos del paradigma como Curry [2] y Toy [3], en la noción de función indeterminista, es decir, función que para unos argumentos dados puede devolver más de un valor.

En trabajos clásicos [4] se distingue entre dos tipos fundamentales de semántica para el indeterminismo en programas funcionales o ecuacionales en general: semántica de no compartición (conocida habitualmente como 'run-time choice') o semánticas de compartición (call-time choice). En el apartado 3.1 explicamos la diferencia entre ambas.

La corriente estándar de la programación lógico funcional, cuyos fundamentos teóricos se encuentran en [5] ha considerado que la semántica más adecuada para la programación práctica es la de call-time choice, que es por ese motivo la adoptada en lenguajes como Curry o Toy.

Las implementaciones tradicionales de estos lenguajes han estado basadas en compilación a Prolog [6], es decir, los programas originales se traducen a programas Prolog objeto, de modo que la ejecución Prolog de estos ya expresa correctamente la semántica de los programas originales. Esta tendencia cambió en KiCS2 [7], una implementación de Curry basada en Haskell, en la que el modo en el que se captura la semántica de call-time choice resulta bastante oscuro y dependiente de aspectos de bajo nivel. Más adelante, S. Fischer realizó otra propuesta puramente funcional para la semántica de call-time choice con evaluación perezosa [8] que resulta mucho más clara y abstracta. Esta propuesta quedó plasmada en una biblioteca de Haskell, `explicit-sharing` que, con las adaptaciones necesarias, ha constituido una base importante de este trabajo.

De manera independiente, en otros trabajos [9] se desarrolla la tarea de combinar las dos semánticas citadas. Sin embargo, esas propuestas solo se implementaron como 'parches' muy poco abstractos en el sistema Toy cuya implementación está basada en Prolog.

La parte esencial de este trabajo del trabajo que presentamos aquí es crear un lenguaje puramente funcional donde se vea el potencial de la combinación de los dos tipos de semánticas y lo que ello puede suponer para la descripción

de algoritmos indeterministas. Hay ejemplos prácticos donde ninguna de las dos semánticas se adapta bien para resolver un problema pero sí una combinación de ambas. Intentaré plasmar la utilidad de esta combinación en ejemplos de uso a lo largo de este documento.

3.1. Programación con funciones no deterministas

La programación no determinista ha demostrado en varias ocasiones simplificar la forma en la que un algoritmo puede ser escrito, como por ejemplo en lenguajes como Prolog, Curry y Toy. En [1] se alega que esto es debido a que los resultados pueden ser vistos individualmente en vez de como elementos pertenecientes a un conjunto de posibles resultados. Un ejemplo recurrente para explicar algunos aspectos básicos de la programación no determinista suele ser el resultado indeterminista de una moneda lanzada al aire, cara o cruz, 0 o 1, para luego proceder a explicar las formas de las que se puede entender una expresión con este valor determinista, cómo funcionaría una suma de estos valores o cómo funcionaría una llamada a una función. Si esa pieza aparece múltiples veces ¿deben tener todas ellas el mismo valor o pueden tener distintos? Tradicionalmente en la programación no determinista existen dos tipos de semántica que nos dicen como interpretar estas piezas no deterministas. Estas dos semánticas son denominadas 'Call-time choice' y 'run-time choice'.

3.1.1. Call-time choice

Conceptualmente, la semántica 'call-time choice' consiste en elegir el valor no determinista de cada uno de los argumentos de una función antes de aplicarlos [8]. Uno de estos lenguajes es Curry. Veamos un ejemplo de este comportamiento:

```
choose x _ = x
choose _ y = y

coin = choose 0 1

double x = x + x
```

Primero vamos a comentar la definición de `choose`. Si escribimos esa definición en Haskell siempre escogerá la primera ecuación, pero en Curry se puede expresar el no determinismo escribiendo varias definiciones que solapen. De este modo `choose 0 1` podrá reducirse a un `0` o a un `1`, ¡esto es justo el no determinismo! Ahora que he explicado el concepto del no determinismo, veamos qué es el 'call-time choice'. En la declaración `double x = x + x`, `x` aparece como argumento. Esto significa que todas las apariciones de `x` corresponderán al mismo valor. Si

evaluamos `double coin` obtendremos solamente dos resultados, `0` y `2`. Si por el contrario simplemente evaluamos la expresión `coin + coin` en el intérprete obtendremos cuatro resultados, `0, 1, 1, 2`. Es decir, en la semántica de 'call-time choice' es como si los valores de los argumentos se eligen en la llamada de la función y lo 'comparten' en toda la expresión.

3.1.2. run-time choice

La semántica 'run-time choice' es mucho más simple. Los valores de las piezas no deterministas no se comparten. Cada aparición de una pieza no determinista generará todos los resultados que le corresponden. En un lenguaje con estas características la evaluación de `double coin` del ejemplo anterior sí producirá los cuatro resultados.

Es decir, en esta semántica hay que entender que cada pieza no determinista, así como las copias de esta que puedan aparecer por aplicación de las ecuaciones que definen las funciones, actuará como una fuente independiente de valores indeterministas, al contrario que en 'call-time choice' donde las apariciones de los argumentos de una función están ligadas a un mismo resultado a lo largo de toda la expresión.

3.1.3. Otras semánticas

Existen otras semánticas en el campo de la programación indeterminista, como la especificada en el trabajo [9], que se propone por la falta de concisión de la semántica 'run-time choice' cuando el ajuste de patrones entran en juego.

Otra semántica es precisamente la combinación de los dos tipos de semánticas comentadas anteriormente, considerada en [9] y que es el corazón de este trabajo. En nuestro trabajo la combinación se consigue con primitivas presentes en Sharade, que especifican cuando compartir el valor de una variable en todas sus apariciones a lo largo de una expresión. Posteriormente daremos ejemplos de uso de esta semántica.

3.2. Objetivos

Mi objetivo en este trabajo de fin de grado es crear un lenguaje funcional que combine las dos semánticas anteriores. Para ello, el lenguaje tendrá primitivas para especificar cuándo las piezas no deterministas de una expresión deben compartir el mismo valor y cuándo se tienen que comportar de la misma forma que en run-time choice. Este lenguaje tendrá características propias de un lenguaje funcional moderno, como orden superior, evaluación perezosa, expresiones lambda, etc. Todo ello combinado con el no determinismo, por ejemplo, una pieza en una expresión se puede corresponder con una función indeterminista. Todo estará implementado

en Haskell aprovechando características que ya tiene de serie además de una serie de bibliotecas. El lenguaje es simple por una cuestión de acotación y para mostrar el núcleo de este trabajo, la combinación de las dos semánticas. No se persigue la eficiencia, pero en el apartado 10.1 se comenta el rendimiento y algunas ideas para mejorarlo.

Para combinar los dos tipos de semánticas, Sharade tiene una estructura primitiva `choose ... in ...`. Es parecido a un `let ... in ...`, pero tiene un comportamiento en el que el identificador ligado que se crea en la parte de la izquierda compartirá el valor en todas las apariciones de ese identificador en la parte derecha. Es decir, tiene el mismo comportamiento que 'call-time choice'. Si no se especifica nada, mi lenguaje semánticamente se comporta igual que 'run-time choice'. Por ejemplo:

```
coin = 0 ? 1 ;

f x = x + x ;
f' x = choose x' = x in x' + x' ;
```

Si evaluamos `f coin` obtendremos los cuatro valores del ejemplo anterior, pero si evaluamos `f' coin` obtenemos dos valores, debido a que se ha creado una ligadura de `x'`. De esta forma, todas las apariciones de `x'` compartirán el mismo valor al evaluar la expresión. Para aclarar, el operador infijo `?` es una primitiva más de Sharade, significa lo mismo que la función `choose` del ejemplo anterior.

En resumen, Sharade combina los dos tipos de semántica, dejando al usuario especificar qué piezas se van a comportar como lo harían en 'call-time choice'. Esta combinación de las dos semánticas no lo hemos visto en la literatura ya existente, salvo en [10], pero allí no es tan clara ni explícita como la que yo propongo en Sharade, ni se propone una implementación con principios claros.

3.3. Plan de trabajo

Primeramente intentaremos expresar el indeterminismo en Haskell. Para ello nos hará falta una cierta soltura en programación monádica. El indeterminismo en Haskell por defecto se trata con la semántica de 'run-time choice'. Cada vez que se quiera 'extraer' un valor de una expresión indeterminista (monádica) surgirán todos los resultados. Este tipo de programación ya es de por sí bastante farragosa, a pesar de que se pueden usar abstracciones bastante elegantes como la notación `do`.

Después comprenderemos la biblioteca `explicit-sharing` de Fischer, que trae las características de 'call-time choice' a Haskell sin perder la evaluación perezosa. Esto lo consigue gracias a las mónadas con estado, con las que implementa la elección temprana de un valor no determinista y la evaluación tardía, que es

exactamente lo que permite esta evaluación perezosa. Es decir, las elecciones del valor a usar se hacen en el momento pero la evaluación de dicho valor solamente cuando es necesaria. Si la programación ya era farragosa antes, ahora mucho más, ya que se añade un nivel superior de mónadas, haciendo necesarias más extracciones de valores de las mónadas, más líneas de código.

De aquí sale la necesidad de crear un pequeño lenguaje donde se descargue toda la sintaxis, pero suficientemente amplio para expresar lo que nos atañe, la combinación de semánticas y su utilidad. Con un comportamiento de 'run-time choice' por defecto y a petición del programador, 'call-time choice' sobre una variable especificada.

Con todo esto en mente, es necesaria la creación de un compilador de programas Sharade a programas Haskell, con una función de traducción formando parte del corazón del trabajo. Esta función de traducción está comentada en el apartado 9.4.

3.4. Estructura del documento

Este documento está pensado para personas con un mínimo conocimiento de Haskell aunque no sepan qué es la programación no determinista. Para ello, en este punto de introducción explico de qué se trata este paradigma con lenguajes ya existentes como Curry. Después explicaré cómo Sharade aborda este paradigma con la combinación de los dos tipos de semántica que existen. El resto del documento trata sobre la implementación de Sharade, que debido a la complejidad subyacente en la implementación de este paradigma en Haskell, es necesario explicar bastantes conceptos de programación monádica en el punto 6. Después comentaré la implementación de Fischer que soluciona el problema de la compartición con evaluación perezosa en el apartado 7.1. En el punto 8 haré una breve explicación sobre la biblioteca `Parsec` que utilizo para realizar el análisis sintáctico de Sharade. Una vez el lector se sienta cómodo con los conceptos de programación monádica, indeterminismo y las dos semánticas, abordaré la implementación de Sharade, desde el análisis sintáctico, análisis de tipos y traducción. Por último habrá un apartado de conclusiones, donde evaluaré el cumplimiento de los objetivos del trabajo, el interés de la propuesta y algunos comentarios sobre Haskell. También recogeré algunos comentarios sobre posibles trabajos futuros, como mejorar el rendimiento de Sharade y la implementación del proyecto, cómo se pueden incorporar nuevas características al lenguaje, como tipos de usuario y otras formas de recolectar resultados.

4. Introduction

This document is part of the result of a academic year of work to complete the BSc Thesis for the Computer Science Engineering degree at the Complutense University of Madrid. In this document I tackle the proposal and implementation of a small and simple functional language with standard features in functional languages such as higher order and pattern matching, to which is added another one not usual in this paradigm, but in an extension of it, the so-called functional logic programming [1], which combines features of independent paradigms of logical and functional programming. From the first it adopts in particular the possibility of indeterministic computations that are sustained in the most well-known languages of the paradigm such as Curry [2] and Toy [3], of the notion of non-deterministic function, that is to say, a function that for given arguments can return more than one value.

In classical works such as [4] there is a relationship between two fundamental types of semantics for indeterminism in functional or equational programming in general: semantics of no sharing (commonly known as 'run-time choice') or semantics of sharing (call-time choice). In the section 3.1 we explain the difference between the two types.

The standard approach to functional logic programming, whose theoretical foundations are found in [5] has considered that the most suitable semantics for practical programming is call-time choice, which is why it is adopted in languages such as Curry or Toy.

The traditional implementations of these languages have been based on compilation to Prolog, that is, the original programs are translated to object Prolog programs, so that the Prolog execution of these programs already expresses correctly the semantics of the original programs. This trend changed in KiCS2 [7], a Curry implementation based on Haskell, in which the mode that captures the semantics of call-time choice is quite dark and dependent on low-level aspects. Later, S. Fischer made another purely functional proposal for call-time choice semantics with lazy evaluation [8] that is much clearer and more abstract. That proposal was embodied in a Haskell library which, with the necessary adaptations, has been an important basis for this work.

The task of combining these two semantics is developed independently in other works [9]. However, these proposals were only implemented as 'patches' in the Toy system whose implementation is based on Prolog.

The essential part of this work is to create a purely functional programming language where you can see the potential of the combination of the two types of semantics and what it can mean for the description of indeterministic algorithms. There are practical examples where neither of the two semantics adapts well to solve a problem but a combination of both ones. I will try to translate the utility

of this combination in the examples used throughout the document.

4.1. Programming with no-deterministic functions

Non-deterministic programming has repeatedly shown to simplify the way in which an algorithm can be written, such as in languages such as Prolog, Curry and Toy. People like Antony and Hanus think this is because the results can be seen individually rather than as elements belonging to a set of possible results [1]. A recurrent example to explain non-deterministic programming is usually the indeterministic result of a coin flipped, heads or cross, 0 or 1, and then proceed to explain the ways in which an expression with this deterministic value can be understood, how a sum of these values would work, or how a call to a function would work. If that piece appears multiple times, must all of them have the same value or can they have different values? Traditionally in non-deterministic programming there are two types of semantics that tell us how to interpret these non-deterministic pieces. These two semantics are the 'Call-time choice' and the 'run-time choice'.

4.1.1. Call-time choice

Conceptually, 'call-time choice' semantics consists of choosing the non deterministic value of each of the arguments of a function before applying them [8]. One of these languages is Curry. Let's see an example of this behavior:

```
choose x _ = x
choose _ y = y

coin = choose 0 1

double x = x + x
```

Let's first comment on the definition of `choose`. If we write that definition in Haskell it will always choose the first equation, but in Curry you can express non-determinism by writing several overlapping definitions. In this way, `choose 0 1` can be reduced to `0` or a `1`, this is exactly non-determinism! Now that I've explained the concept of non-determinism, let's see what the 'call-time choice' is. In the statement `double x = x + x`, `x` appears as an argument. This means that all appearances of `x` will correspond to the same value. If we evaluate `double coin` we will get only two results, `0` and `2`. If we simply evaluate the expression `coin + coin` in the interpreter we will get four results, `0, 1, 1, 2`. That is to say, in the semantics of 'call-time choice' it is as if the values of the arguments are chosen in the call of the function and 'share' it in all the expression.

4.1.2. run-time choice

The semantics 'run-time choice' is much simpler. The values of non deterministic parts are not shared. Each appearance of a non-deterministic piece will generate all the results that correspond to it. In a language with these characteristics the evaluation of `double coin` of the previous example will produce the four results.

That is to say, in this semantics it must be understood that each non deterministic piece will act as an independent source of indeterministic values, unlike in 'call-time choice' where the appearances of the arguments of a function are linked to the same result throughout the entire expression.

4.1.3. Other semantics

There are other semantics in the field of indeterministic programming, such as the one specified in the work *Singular and plural functions for functional logic programming* [9], which is proposed by the lack of concision of the semantics 'run-time choice' when pattern matching comes into play.

Another semantics is indeed the combination of the two types of semantics mentioned above, which is the core of this work. In our work the combination is achieved with primitives present in Sharade, which specify when to share the value of a variable in all its apparitions throughout an expression. Later we will give examples of the use of this semantics.

4.2. Objetives

My purpose in this BSc Thesis is to create a functional language that combines the two previous semantics. To do this, the language will have primitives to specify when nondeterministic pieces of an expression must share the same value and when they must behave in the same way as in run-time choice. This language will have characteristics of a modern functional language, such as higher order functions, lazy evaluation, lambda expressions, etc. All this combined with non-determinism, for example, a part in an expression can correspond to an indeterministic function. Everything will be implemented in Haskell taking advantage of features it already has as standard in addition to a series of libraries. The language is simple for a matter of dimensions of the work and to show the the core of it, the combination of the two semantics. Efficiency is not pursued, but in the section 10.1 the performance is discussed and some ideas are given for improve it.

To combine the two types of semantics, Sharade has a primitive structure `choose ... in ...`. It's like a `let ... in ...`, but it has a behavior in which the bound identifier that is created on the left side will share the value in all

occurrences of that identifier on the right side. If nothing is specified, my language semantically behaves the same as 'run-time choice'. For example:

```
coin = 0 ? 1 ;

f x = x + x ;
f' x = choose x' = x in x' + x' ;
```

If we evaluate `f coin` we get the four values from the previous example, but if we evaluate `f' coin` we get two values, because a binding for `x'` has been created. This way, all occurrences of `x'` will share the same value when evaluating the expression. To clarify, the operator infixed `?` is a primitive of Sharade, it means the same as the function `choose` of the previous example.

In conclusion, Sharade combines the two types of semantics, letting the user specify which pieces will behave in a 'call-time choice' regime. This combination of the two semantics has not been seen in existing literature. It is true that there is a similar approach in *A Flexible Framework for Programming with Non-deterministic Functions* [10], but it is not as clear or explicit as the one I propose in Sharade.

4.3. Workplan

First we will try to express the indeterminism in Haskell. For this we will need a certain fluency in monadic programming. The indeterminism in Haskell by default is dealt with the semantics of 'run-time choice'. Every time you want to 'extract' a value from an indeterministic (monadic) expression, all the results will appear. This type of programming is in itself quite tedious, although elegant abstractions such as notation `do` can be used.

Then we'll understand Fischer's explicit-sharing library, which brings the 'call-time choice' features to Haskell without losing lazy evaluation. He achieves this thanks to the state monads, with which he implements the early choice and late evaluation. In other words, the choices of the value to use are made at the moment but the evaluation of that value is only done when it is required. If the programming was already tedious before, now it is even more, since a higher level of monads is added, making necessary more extractions of values of the monads, more lines of code.

It turns out then the convenienve of creating e a small language where all the syntax is discharged, but wide enough to express what concerns us, the combination of semantics and their usefulness. With a behavior of 'run-time choice' by default and at the request of the programmer, 'call-time choice' on a specified variable.

With all this in mind, it is necessary to create a Sharade program compiler to Haskell programs, with a translation function being part of the heart of the work. This translation function is discussed in the section 9.4.

4.4. Document structure

This document is intended for people with minimal Haskell knowledge even if they don't know what non-deterministic programming is. To this end, at this introductory point I have explained what this paradigm is about with existing languages such as Curry. Later I will explain how Sharade approaches this paradigm with the combination of the two types of semantics that concern us. The rest of the paper deals with the implementation of Sharade. Due to the complexity underlying the implementation of this paradigm in Haskell, it is necessary to explain quite a few monadic programming concepts in section 6. Then I will comment on the implementation of Fischer that solves the problem of sharing with lazy evaluation in the section 7.1. In point 8 I will make a brief explanation about the library `Parsec` that I use to perform the Sharade syntax analysis. Once the reader is comfortable with the concepts of monadic programming, indeterminism and the two semantics, I will approach the implementation of Sharade, from parsing, type analysis and translation. Finally there will be a section of conclusions, where I will collect some comments on performance, how to improve the implementation of the project, how new features can be incorporated into the language, such as user types, other ways to collect results and possible future work.

5. Sharade

5.1. Introducción

El resultado de este trabajo de fin de grado es un lenguaje funcional con características habituales en el paradigma de la programación funcional como el orden superior, evaluación perezosa, ajuste de patrones, expresiones lambda, no determinismo, semánticas de compartición (call-time choice) y no compartición (run-time choice). Posee la inferencia de tipos de Hindley-Milner pero no se pueden especificar los tipos en la declaración de las funciones. No tiene clases de tipos ni permite crear tipos de usuario por cuestiones explicadas más adelante. La sintaxis está muy influenciada por Haskell, tiene aplicación parcial (excepto para operadores infijos) y notación currificada. El lenguaje carece de algunos azúcares sintácticos habituales, por ejemplo, las listas se tienen que escribir a base de las constructoras `Cons` y `Nil`, no existe ajuste de patrones en los lados izquierdos de la definición de funciones, estando delegados a una expresión `case` en las expresiones de los lados derechos ni tampoco existen listas intensionales. No posee ninguna biblioteca estándar ni preludio, todas las funciones básicas deben ser programadas por el usuario. En cuanto a los tipos primitivos disponibles (y únicos) están los enteros, los reales, los caracteres, las listas y las parejas. Todo esto es debido a que se trata de un trabajo de fin de grado donde el objetivo era mostrar de una forma simple y clara la combinación de los dos tipos de semánticas. Esta combinación puede ser muy útil, ya que hay situaciones prácticas donde ninguna de las dos semánticas son apropiadas pero sí una combinación de ellas [10], como veremos en los ejemplos más adelante. La combinación también permite describir algoritmos con mayor flexibilidad y otorgando más control al programador. Tal como hemos decidido definir la sintaxis, la combinación resulta muy sencilla y limpia, como veremos en el siguiente apartado.

Existen lenguajes más avanzados con más características y con mejor rendimiento que el mío, como pueden ser Curry y Toy. Sharade no persigue el rendimiento (aunque con algunas optimizaciones puede llegar a tener un rendimiento similar) ya que su objetivo es incorporar a la literatura de la comunidad lógico-funcional algo nuevo, la combinación de dos semánticas indeterministas, el 'call-time choice' y 'run-time choice'. Esto se consigue, desde el punto de vista de la estructura del lenguaje, asumiendo por defecto un comportamiento de semántica de 'run-time choice' y a la incorporación de una primitiva que veremos en el siguiente apartado mediante ejemplos.

En cuanto a la implementación, que comentaremos durante capítulos sucesivos, la basaremos en una traducción de Sharade a Haskell que hace un uso intensivo de la programación monádica para expresar el indeterminismo.

5.2. Un vistazo a Sharade

En los siguientes ejemplos voy a mostrar la sintaxis y características de Sharade y cómo se pueden combinar entre ellas. Además, también voy a mostrar dos programas algo más elaborados, como la implementación de un autómata finito no determinista y un tipo de ordenación de listas algo peculiar, la ordenación por permutación. Este último ejemplo se usa bastante en la literatura para comparar el rendimiento de los lenguajes no deterministas, ya que usa intensivamente el indeterminismo y tiene gran carga computacional. Se basa en escoger la permutación ordenada de una lista.

5.2.1. No determinismo

Un ejemplo simple donde aflora el no-determinismo podría ser el siguiente:

```
number = 1 ? 2 ? 3 ? 4 ;

duplicate x = x + x ;
```

El operador infijo '`?`' es el origen del indeterminismo en Sharade. Esta definición significa que `number` se puede reducir a uno de esos cuatro valores. La función `duplicate` tiene un argumento, que potencialmente puede ser no determinista. Es decir, puede operar con un valor determinista (por ejemplo, un simple `2`) o un valor no determinista como `number`. Tradicionalmente hay dos formas de entender este programa. Si entiende con la semántica de 'call-time choice', `duplicate number` solo tendrá 4 resultados. Sin embargo, con la semántica 'run-time choice' se obtienen 16 resultados, provenientes de todas las combinaciones posibles de los valores de `number`.

Sharade por defecto opera con la semántica de 'run-time choice', por lo tanto la expresión `duplicate number` tiene 16 resultados, de los cuales 12 se repiten por la conmutatividad de la suma. En el siguiente ejemplo vamos a ver como podemos operar con la otra semántica.

5.2.2. Compartición

Para usar el otro tipo de semántica hay que introducir la primitiva '`choose`'. Es una primitiva parecida a una expresión '`let`'. Crea un nuevo ámbito con una nueva variable que cumplirá las leyes del 'call-time choice' [11]. Es decir, todas sus apariciones en una expresión se evaluarán al mismo valor.

```
number = 1 ? 2 ? 3 ? 4 ;

duplicate x = choose a = x in a + a ;
```

En el ejemplo se está ligando la variable `a` a uno de los posibles valores de `x`. De esta forma, la expresión `duplicate number` solo tiene 4 posibles valores, igual que en Curry.

5.2.3. Combinación de semánticas

En el siguiente ejemplo se puede ver la combinación de los dos tipos de semántica, un salto que ha dado Sharade:

```
number = 1 ? 2 ? 3 ? 4 ;
coin = 0 ? 1 ;

f x y = choose a = x in a + a + y + y ;
```

Si se evalúa `duplicate number coin` existen 16 combinaciones. Esto es debido a que la variable 'y' está siguiendo las reglas del 'run-time choice' y la variable 'a' las del 'call-time choice'.

Se puede pensar que una variable sujeta a la semántica de 'call-time choice' es una variable normal de cualquier lenguaje, sus apariciones comparten el valor a lo largo de la expresión.

Veamos otro ejemplo práctico donde además se introducen listas infinitas e infinitos resultados indeterministas:

```
letter = 'a' ? 'b' ? 'c' ;

word = Nil ? Cons letter word ;

concatenate xs ys = case xs of
  Nil -> ys ;
  Cons x xs -> Cons x (concatenate xs ys) ;;

reverse' xs ys = case xs of
  Nil -> ys ;
  Cons x xs -> reverse' xs (Cons x ys) ;;

mreverse xs = reverse' xs Nil ;

palindrome = choose w = word in concatenate w (mreverse w) ;
```

Las únicas piezas no deterministas en este programa son las definiciones de `letter`, `word` y `palindrome`. Una letra puede tener cuatro posibles valores y una palabra puede ser la cadena vacía (`Nil`) o una letra seguida de una palabra. Así,

una letra tiene el tipo `Char` y una palabra el tipo `[Char]`. Un palíndromo es una palabra concatenada con su inversa. Por simplicidad en el ejemplo, no he considerado los palíndromos de longitud impar. Las funciones auxiliares son deterministas y no difieren en nada de como se programarían en un lenguaje funcional tradicional. En este ejemplo se puede ver la combinación de las dos semánticas, 'run-time choice' en `word` y 'call-time choice' en `palindrome`. Completemos el ejemplo:

```
mlength xs = case xs of
  Nil -> 0 ;
  Cons x xs -> 1 + mlength xs ;;

e1 = choose p = palindrome in case mlength p == 4 of
  True -> p ;;
```

La función `mlength` es clásica. Si evaluamos `e1` se calcularán todos los palíndromos de longitud cuatro pero desgraciadamente el cómputo nunca terminará. Esto es debido a una limitación de Haskell y se debe a la misma razón por la cual la expresión `filter (\a -> length a == 4) (iterate (1:) [])` no termina.

5.2.4. Orden superior

Sharade también tiene características de orden superior, indispensables en un lenguaje funcional. El orden superior se puede combinar con el no determinismo, con el operador '`?`' y con la primitiva `choose`.

```
f a b = a + b ;
g a b = a * b ;

h z a b = z a b ;
```

En el ejemplo se crea una función `f` idéntica a la función suma, una función `g` idéntica a la multiplicación y una función `h`, con tres argumentos. El primero es una función que se aplica al segundo y al tercero. De esta forma, se puede pasar como argumento una función no determinista e incluso aprovechar las expresiones lambda. También es posible combinar esta característica con la primitiva `choose`:

```
e1 = h (f ? g) 1 2 ;
e2 = choose f' = f ? g in f' (f' 1 2) (f' 1 2) ;
```

La definición de `e1` es muy simple y tendrá como resultado dos valores, `3` y `2`. Con `e2` se puede ver de nuevo en acción la primitiva `choose` ligando `f'`. De esta manera, los dos únicos valores posibles son `(1 + 2) + (1 + 2) = 6` y `(1 * 2) * (1 * 2) = 4`.

5.2.5. Ajuste de patrones

Haskell permite usar ajuste de patrones en la definición de las funciones, pero no es más que azúcar sintáctico para expresiones `case` muy tediosas. Por simplicidad, Sharade no cuenta con este azúcar sintáctico.

```
f a = case a of
  0 -> 1 ;
  1 -> 2 ;
  v -> v + v ;;

fact n = case n of
  1 -> 1 ;
  n -> n * fact (n - 1) ;;
```

El primer ejemplo es el interesante, el segundo es una implementación simple del cálculo del factorial, no emerge ninguna característica del no determinismo y menos de la compartición. Se deja ver que Sharade también soporta recursión, un añadido a la inferencia de tipos de Hindley-Milner. En el primer ejemplo, igual que en la primitiva `choose`, el último patrón crea un nuevo ámbito con la variable `v` disponible. En este caso también aplican las leyes del 'call-time choice', pero no con buenas propiedades, como la demanda tardía y la ley 'ignore', como veremos más adelante en la implementación. Para entenderlo rápidamente, se pierde la evaluación perezosa cuando es obligatorio evaluar `a`.

5.2.6. Autómata finito no determinista

Un primer ejemplo de la potencia que permite la programación no determinista podría ser la implementación de un autómata finito no determinista de una forma muy sencilla.

```
-- f :: State -> Char -> State
f s a = case s of
  0 -> case a of
    'a' -> 0 ;
    'b' -> 0 ? 1 ;;
  1 -> case a of
    'a' -> 2 ;;;

-- accept' :: (State -> Char -> State) -> State -> [Char] -> Bool
accept' f s as = case as of
  Nil       -> s == 2 ;
```

```
  Cons a as -> accept' f (f s a) as ;;

-- accept :: (State -> Char -> State) -> [Char] -> Bool
accept f as = accept' f 0 as ;
```

Aquí se ven en juego todas las características previamente mencionadas. El ajuste de patrones, orden superior y no determinismo. No se ve ningún ejemplo de semántica 'call-time choice', ya que en este ejemplo no tiene cabida, se quiere usar las características que proporciona la semántica 'run-time choice'.

Este AFN reconoce las cadenas de caracteres terminadas en 'ba'. La función `f` es la función de transición. La función de transición en un AFN es no determinista y en este ejemplo esto se puede apreciar en el primer estado cuando se lee una 'b'. En un lenguaje tradicional se tendría que trabajar con un conjunto de valores, pero en este paradigma eso se da por supuesto, permitiendo una potencia descriptiva del algoritmo bastante atractiva. La función `accept'` aplica recursivamente la función de transición, aceptando únicamente si se ha consumido toda la entrada y el autómata se encuentra en el estado 2.

5.2.7. Ordenación por permutación

Este ejemplo es el que usa Fischer para mostrar la biblioteca en la que me he basado para construir Sharade. Este tipo de ordenación es simplemente para un ejemplo y no tiene ningún uso práctico, ya que este algoritmo de ordenación es extremadamente ineficiente. Consiste en calcular todas las permutaciones de una lista y comprobar si alguna de ellas está ordenada. Con la evaluación perezosa quizás no es necesario calcular toda la permutación completa, ya que solo con mirar los dos primeros elementos de la lista puedes descartar todo el subárbol de permutaciones que se podrían generar. Fischer lo utiliza para mostrar que funciona la evaluación perezosa y por tanto se pueden descartar permutaciones sin necesidad de calcularlas enteras. Lo he querido poner como ejemplo para mostrar de nuevo la semántica 'call-time choice' de Sharade y posteriormente para analizar la pérdida de rendimiento de Sharade. Como el ejemplo puede resultar complejo la primera vez, he decidido separarlo por trozos para explicarlo mejor.

```
insert e ls = (Cons e ls) ? case ls of
  Cons x xs -> Cons x (insert e xs) ;;
```

Esta función 'insertar' recibe un elemento y una lista. Inserta el elemento en algún lugar de la lista. Por lo tanto, se puede poner en la cabeza de la lista o insertarlo en algún lugar de la cola, llamando recursivamente.

```
perm ls = case ls of
  Nil -> Nil ;
  Cons x xs -> insert x (perm xs) ;;
```

Esta función calcula todas las permutaciones de una lista dada, apoyándose en la función `insert`y llamándose recursivamente. Se calcula por tanto insertando la cabeza en todas las posiciones posibles de la permutación de la cola.

```
isSorted ls = case ls of
  Nil -> True ;
  Cons x xs -> case xs of
    Nil -> True ;
    Cons y ys -> x <= y && isSorted (Cons y ys) ;;;

sort ls = choose sls = perm ls in case isSorted sls of
  True -> sls ;;
```

La función `isSorted` es compleja de seguir debido a todo el ajuste de patrones que hay que hacer pero no difiere en nada de como se haría en un lenguaje funcional tradicional. Simplemente calcula si una lista dada está ordenada. La segunda función, `sort`, calcula una permutación y comprueba si está ordenada, en tal caso la devuelve. Para ello hay que usar la primitiva `choose`, ya que `sls` aparece dos veces en la expresión y debe compartir el valor porque tiene que pasar el filtro de si está ordenada.

En resumen, el ejemplo calcula todas las permutaciones y devuelve la que está ordenada. También se aprovecha de las características como la evaluación perezosa que Fischer consigue traer al no determinismo en Haskell.

5.2.8. Evaluación perezosa

Para terminar, me gustaría mostrar que Sharade evalúa perezosamente y por tanto puede tratar con estructuras de datos infinitas, resultados que podrían dar error o cómputos infinitos.

```
f x y = y ;
e0 = f (1/0 ? 2/0) 0 ;
```

Si se evalúa `e0` da como resultado `0`, el segundo argumento, no da ningún error por dividir por cero, tal como lo haría Haskell. Esto es debido a que la función no es estricta en el primer argumento. Esta propiedad es muy importante, Fischer la llama 'Ignore'. Explicaré en el aparatado 'Biblioteca explicit-sharing' cómo se consigue esto en la implementación. Veamos el siguiente ejemplo:

```
mrepeat x = Cons x (mrepeat x) ;
mtake n xs = case n of
  0 -> Nil ;
  n -> case xs of
    Nil -> Nil ;
    Cons y ys -> Cons y (mtake (n - 1) ys) ;;;

coin = 0 ? 1 ;
e1 = mtake 3 (mrepeat coin) ;
e2 = choose c = coin in mtake 3 (mrepeat c) ;
e3 = let c = coin in mtake 3 (mrepeat c) ;
```

Las funciones `mrepeat` y `mtake` son deterministas. La primera crea una lista infinita con el primer argumento en cada posición y la segunda recoge los primeros `n` elementos de una lista. La primera y la tercera expresión, `e1` y `e3`, son idénticas. El cómputo termina y da como resultado un conjunto de ocho resultados, ya que las dos expresiones son equivalentes a `[coin, coin, coin]` y como cada coin es independiente se generan los ocho resultados.

La segunda expresión, `e2`, también termina pero da como resultado dos valores. Esto es normal porque, como ya veremos, la expresión es equivalente a `choose c = coin in [c, c, c]`, por lo que el valor de todas las `c` se comparte.

En resumen, Sharade consigue mantener las buenas propiedades que Fischer consigue con su implementación en Haskell, en este ejemplo concreto, con la evaluación perezosa. Otra propiedad que mantiene, esta vez gracias a Haskell, es la compartición de los argumentos de una función, de manera que si uno de ellos tarda mucho en computarse, se puede reutilizar ese valor en todas las apariciones de dicho argumento.

5.3. Recolección de resultados

La implementación de Fischer provee dos formas de recolectar los resultados en Haskell. Sobre cómo usar el código generado hablaremos más adelante. Estas dos formas son con la función `results` y otra con `unsafeResults`. Para entender a qué se refiere Fischer con 'unsafe' hay que entender las siguientes leyes:

- Idempotencia: `a ? a = a`

- Conmutativa: `(a ? b) ? (c ? d) = (a ? c) ? (b ? d)`

Cualquier implementación que recolecte los resultados que no cumpla esas leyes es considerada 'insegura'. Con estas leyes, una recolección de resultados por anchura o por profundidad sería considerada insegura, ya que no cumpliría

la segunda ley. Además, también se tendría que omitir resultados, ya que no se pueden repetir.

La primera implementación de Fischer se basa en un conjunto ordenado de Haskell (`Set a`). Los resultados no se almacenan de acuerdo con la recolección de ellos. Otro detalle es que se tienen que recolectar todos los resultados, por lo que se carece de evaluación perezosa y en caso de que haya infinitos resultados, la ejecución nunca terminará. De hecho, ni si quiera se podrá obtener el primero de ellos.

La segunda implementación, la insegura, usa las listas estándar de Haskell. Al contrario que la anterior, posee evaluación perezosa, puede tratar con resultados infinitos y es equivalente a una búsqueda en profundidad por la izquierda.

5.4. Sintaxis abstracta

Los ejemplos anteriores han permitido introducir buena parte de los elementos sintácticos de Sharade, así como su sentido dentro de los objetivos de este trabajo. En este apartado mostramos una visión más completa y unificada de las diferentes construcciones sintácticas del lenguaje para posterior referencia, dejando los detalles más concretos de la sintaxis al apartado 9.1, al describir las distintas componentes de la implementación.

Un programa de Sharade consiste en una serie de definiciones de funciones de la forma `f x1 ... xn = e ;`, con $n >= 0$, siendo `f, x1 ... xn` identificadores que sirven tanto para variables como funciones (permitiendo orden superior) y `e` es una expresión. Cada nombre de función `f` dispone de una única ecuación definicional, por lo tanto el ajuste de patrones y el indeterminismo se concentran en la expresión `e`. Una expresión `e` adopta una de las siguientes estructuras:

- l ⇒ Un literal, que puede ser True, False y un número entero o real.
- x ⇒ Una variable, siempre empiezan con minúscula.
- C ⇒ Una constructora de datos, siempre empiezan con mayúscula.
- $e1\ e2$ ⇒ Una aplicación de funciones.
- $\backslash x \rightarrow e$ ⇒ Una función lambda.
- $e1\ ?\ e2$ ⇒ Una elección de dos valores, origen del indeterminismo.
- $(e1,\ e2)$ ⇒ Parejas polimórficas.
- $choose\ x = e1\ in\ e2$ ⇒ Compartición de la variable.
- $let\ x = e1\ in\ e2$ ⇒ Creación de una ligadura.

- $e1\ (infixOp)\ e2$ $\Rightarrow$ Una expresión con un operador infijo. Los únicos operadores infijos que existen son $(+)$, $(-)$, $(*)$, $(/)$, $(\&\&)$ y $(||)$.

- $case\ e\ of\ t1 \rightarrow e1; \ldots tn \rightarrow en;$ $\Rightarrow$ Donde `t` es un patrón, que tiene la estructura `C x1 ... xn` o `(x1, x2)`, siendo C una constructora de datos y el segundo patrón es el de las parejas. Las únicas constructoras de datos que existen son los números, True, False, Cons, Nil y Pair.

Para las aplicaciones asumimos las convenciones habituales de la notación currificada, en la que '$e1\ e2 \ldots en$' es azúcar sintáctico para '$(\ldots(e1\ e2)\ldots)\ en$'. Las únicas funciones que existen en el lenguaje son $dAdd$, $dSub$, $dMul$, $dDiv$, ..., $dNeq$ para el tipo Double. Los operadores infijos son únicamente para los números enteros y booleanos. Esto es así debido a que el lenguaje no tiene clases de tipos. Para la función `?` asumimos el tipo `(?) :: a -> a -> a`.

Pedimos a los programas que estén bien tipados con relación al sistema de tipos Hindley-Milner. Como tipos primitivos consideramos `Bool`, `Char`, `Integer`, `Double`, las parejas polimórficas (`(a, b)`) y las listas (`[a]`). Los tipos no se pueden especificar en la definición de las funciones.

5.5. Instalación y uso

Instalar la biblioteca Sharade y el compilador es relativamente sencillo. Para empezar, se necesita una versión especifica de una biblioteca que no existe en el repositorio de paquetes de Haskell, Hackage. Esta biblioteca es `explicit-sharing`. Su autor original, como ya he dicho en múltiples ocasiones, es Fischer, pero he tenido que hacer unos retoques para hacerla compatible con las nuevas versiones de Haskell. Para instalar esta biblioteca, se puede usar la siguiente ristra de comandos:

```
# Clonar el repositorio
git clone git@github.com:ManuelVs/explicit-sharing.git
# Actualizar la lista de paquetes de cabal
cabal update
# Generar la biblioteca
cabal sdist
# Instalar la biblioteca
cabal install dist/explicit-sharing*.tar.gz
# Limpiar los resultados intermedios (opcional)
cabal clean
```

Ahora hay que hacer algo parecido para instalar mi biblioteca y compilador:

```
# Clonar el repositorio
git clone git@github.com:ManuelVs/Sharade.git
# Actualizar la lista de paquetes de cabal
cabal update
# Generar la biblioteca
cabal sdist
# Instalar la biblioteca
cabal install dist/Sharade*.tar.gz
# Limpiar los resultados intermedios (opcional)
cabal clean
```

En la instalación `cabal` debe haber puesto el compilador de Sharade en alguna localización accesible para la línea de comandos, añadiéndolo al `PATH` o similar. En caso de que no lo haya hecho, se puede hacer a mano. En un sistema Linux debería poder encontrarse en `/home/username/.cabal/bin`. En un sistema Windows puede encontrarse en `C:\Users\username\.cabal\bin`.

Para compilar un fichero tan solo hay que pasar como argumento la localización de dicho fichero.

```
$ cat example.sa
mRepeat a = a ? mRepeat (a + 1) ;
$ Sharade example.sa
Done.
```

5.6. Uso en Haskell

Para usar la traducción hay que conocer un repertorio de funciones básicas:

- `results`: Recolecta los resultados en un conjunto ordenado de Haskell. Tiene el tipo `results :: Sharing s => s a -> Set a`. Esta función asegura las leyes de Fischer, explicadas anteriormente.

- `unsafeResults`: Recolecta los resultados en una lista estándar de Haskell. Tiene el tipo `unsafeResults :: Sharing s => s a -> [a]`.

- Operador infijo `<#>`: Necesario para hacer cualquier aplicación funcional con funciones traducidas de Sharade. Su existencia es explicada en un punto posterior de este documento. Por ejemplo, `mRepeat <#> (return 1)`.

- `convert`: Convierte tipos primitivos de Haskell a Sharade y viceversa, incluyendo listas y parejas. El tipo es complejo e incorpora nuevas clases de tipos cuya explicación distraería el objetivo de este apartado. Para transformar

los tipos de Haskell a los tipos de Sharade hay que usarla de la manera convencional, por ejemplo, `convert [1,2,3::Integer]`. Es necesario especificar los tipos, ya que Haskell en ocasiones no es capaz de inferirlos (por estas clases de tipos). Para transformar los tipos en el otro sentido hay que usar el operador `>>=`. Por ejemplo,
`sort <#> (convert [1..20::Integer]) >>= convert`.

De esta manera, para usar la función compilada en el ejemplo de `mRepeat` se puede usar la siguiente expresión Haskell:

```
e0 = unsafeResults $ mRepeat <#> (return 1) :: [Integer]
```

Para evaluar la función `sort` del ejemplo de ordenación por permutación se puede usar:

```
e1 = unsafeResults $ sort <#>
      (convert [1..20::Integer]) >>= convert :: [[Integer]]
```

6. Conocimientos previos

En este apartado explicaremos conceptos que son necesarios para un buen entendimiento de este trabajo, ya que hacemos uso de una programación monádica muy intensiva. Creemos que un programador aficionado no tiene por qué conocer los detalles ni tener soltura implementando instancias de las clases de tipos que veremos, por lo que intentaremos hacer una progresión de menor a mayor dificultad. Algún ejemplo puede resultar un reto entenderlo, sobre todo si nunca se ha tratado con este tipo de programación con anterioridad.

Muchas de las explicaciones de este punto están remotamente inspiradas en el libro *Learn You a Haskell for Great Good* [12]. El código de ejemplo y las explicaciones son de mi puño y letra, pero he tomado prestadas algunas comparaciones como 'caja' para poder hacer la explicación mas entendible para las personas que no están acostumbradas al paradigma de la programación funcional.

6.1. Clases de tipos

Las clases de tipos en Haskell son agrupaciones de tipos que tienen la característica común de que implementan una serie de funciones. Por ejemplo, `Int` o `Double` pertenecen a la clase de tipos `Num`, tienen en común que implementan las funciones suma, multiplicación, resta, valor absoluto...

Dicho de otra manera, las clases de tipos son una especie de interfaz que implementa algún tipo concreto, añadiéndole alguna funcionalidad. No hay que confundir las clases de tipo con las clases de Java o C++, no tiene nada que ver. Una comparación mas idónea sería decir que son como las interfaces de Java, pero en mi opinión las clases de tipos son más potentes.

6.2. Familias de tipos, *Kinds*

También es conveniente entender cómo funcionan los tipos en Haskell. Los tipos en Haskell pertenecen a familias de tipos. La familia de `Int` o de `Double` o de `[Int]` es `*`. Un asterisco quiere decir que es un tipo concreto. Por ejemplo, la familia de `[]`, las listas, es `* -> *`, es decir, necesita un tipo concreto para dar otro tipo concreto. De este modo, la familia de `Either` es `* -> * -> *`, necesita dos tipos concretos para dar otro. Es parecido al tipo de las funciones, donde `Int -> Int` representa un tipo funcional que espera un `Int` para dar un `Int`, solo que en este caso solo existe un tipo básico, `*`. Se puede ver que esta semántica no tiene la misma riqueza que la semántica de los tipos de Haskell, donde existen múltiples variables de tipos (`a`, `b`...) al contrario que esta, donde solo tenemos el asterisco.

Hay que remarcar que no se debe confundir las familias de tipos con las clases de tipos. Una clase de tipos es por ejemplo `Num`, `Fractional`. Rizando el rizo, las siguientes son clases de constructoras de tipos. Se llaman así porque, por ejemplo, se puede entender que `[]` es una constructora de tipos, igual que `Either`. Así, en vez de ser simples clases de tipos, se convierten en clases de constructoras de tipos.

Podríamos dar una vuelta de tuerca más y explicar que las familias de tipos también tienen su sistema de tipos especial, ya que esperan una familia de tipos concreta, pero no es el objetivo de este trabajo por lo que no creo que tengan mucho interés. Eso sin contar que, hasta donde llegan mis conocimientos de Haskell, no hay sistemas superiores de tipos que esperen como argumentos familias de tipos.

6.3. Funtores

Los funtores, `Functor` en inglés, es una clase de constructora de tipos de Haskell definida de la siguiente forma:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

Define una función `fmap` que necesita una función que mueva elementos del tipo `a` a elementos del tipo `b`, un valor de tipo `f a` y devuelve un `f b`. Como se puede ver, `f` es un tipo que pertenece a la familia `* -> *`, es decir, necesita un tipo para ser un tipo concreto. Hay muchas constructoras de tipos que pertenecen a esta clase, como la del tipo de las listas o la constructora de tipos `Maybe`.

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Efectivamente, la función `fmap` para las listas es exactamente la función `map`, que en este caso, el funtor `f` es `[]`. Conceptualmente, `map` hace lo que tiene que hacer `fmap`, mover valores dentro de un funtor de un tipo a otro, dicho de otra manera, transformar una lista de elementos de `a` a otra lista de elementos de `b`.

Se puede entender los tipos de esta clase de tipos como cajas que encierran un tipo. Por ejemplo, `[]` es una caja que encierra un tipo, `Maybe` también. Esta analogía de las cajas es bastante usada en la literatura, y la usaré más veces explicando las demás clases de tipos.

Otro ejemplo sería el caso de `Either a b`. No se puede conseguir meter este tipo en la clase `Functor` con los dos valores, ya que es obligado que pertenezca a la familia de tipos `* -> *`. Pero como también existe aplicación parcial en las constructoras de tipos, se puede meter el tipo `(Either a)`' en la clase `Functor`:

```
instance (Either a) where
  fmap _ (Left a)  = Left a
  fmap f (Right b) = Right (f b)
```

6.4. Funtores aplicativos

Podemos 'bajar' de alguna manera en la jerarquía de las clases de tipos, encontrándonos con los funtores aplicativos. Se llaman aplicativos porque es un funtor que se puede aplicar, es decir, la función se recibe dentro de una 'caja', ¡dentro de un funtor! El valor también se recibe dentro de una caja. Veamos cómo está declarada esta clase:

```
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Antes de estar en la clase de los funtores aplicativos, un requisito es ser un funtor. Para terminar de serlo, hay que tener implementadas dos funciones, `pure` y el operador infijo `<*>`. `pure` es simplemente una función que recibe un valor tradicional y lo devuelve metido en una caja. El operador `(<*>)`, primo hermano de `fmap`, recibe una función metida en una caja, un valor metido en una caja y devuelve un valor encerrado en una caja aplicando la función. Veamos cómo se implementa la pertenencia a esta clase para algunos tipos:

```
instance Applicative Maybe where
  pure x = Just x
  (<*>) Nothing _ = Nothing
  (<*>) _ Nothing = Nothing
  (<*>) (Just f) (Just x) = Just (f x)
  --(Just f) <*> x = fmap f x -- Otra implementación posible

instance Applicative [] where
  pure x = [x]
  fs <*> xs = concatMap (\f -> map f xs) fs

instance Applicative ((->) c) where
```

```
  pure x = (\c -> x)
  f <*> f' = (\c -> f c (f' c))
```

El más sencillo es el primero, la implementación de `Maybe`. Si la función o el valor son Nothing, el resultado es Nothing, ya que nos faltan argumentos con los que trabajar. Si tenemos los valores, los sacamos de sus respectivas cajas, aplicamos la función y volvemos a meter en una caja el resultado.

La lista es más complicada. Por la izquierda, `fs`, recibimos una lista de funciones, por la derecha, `xs`, una lista de valores. Tenemos que aplicar cada función a todos los valores. Para ello hacemos un `map` con cada función y luego concatenamos todas las listas generadas.

El ejemplo más complicado, si duda, es el tercero. `(->)` es una constructora primitiva de Haskell que crea tipos funcionales. Necesita dos tipos, el tipo origen y el tipo destino. Por ejemplo, `(+) :: Int -> (Int -> Int)`. Así, `((->) c)` es una aplicación parcial de la constructora de tipos. De este modo, su familia de tipos es `* -> *`, por lo que es apto para ser un funtor aplicativo. De la declaración de tipo de `(<*>)` se deduce que `f :: c -> a -> b`, `f' :: c -> a`, por lo tanto el tipo del resultado debe ser `c -> b`, una función. Así, primero aplicamos `f' c :: a`, después aplicamos `((f c :: a -> b) (f' c :: a))`, que unido a la lambda expresión que engloba a todo, se obtiene el tipo `c -> b`, como se quería. Visto así puede ser complicado, veamos un ejemplo no muy práctico con números, donde `a, b, c :: Int`:

```
   ((+) <*> (\a -> a + 10)) 3
= (\c -> (c+) ((\a -> a + 10) c) 3
= (\c -> (c+) (c+10)) 3
= (3+) (13) = 16
```

Este ejemplo viene bien como entrenamiento para lo que viene, ya que en las mónadas con estado se usan funciones para 'esconder' valores, aprovechando las características de memoización de Haskell, de la misma manera que en un lenguaje funcional sin evaluación perezosa se puede conseguir el mismo efecto pasando funciones. En el ejemplo, creamos una lambda para recuperar el valor `c`.

6.5. Mónadas

Podemos seguir bajando en la jerarquía de clases de tipos para encontrarnos con las mónadas. Para seguir con la casuística de meter cosas en cajas, ahora la función no está metida en una caja... ¡pero devuelve un valor metido en una caja! Veamos la declaración de la clase:

```
class Applicative m => Monad (m :: * -> *) where
  return :: a -> m a
```

```
  (>>=) :: m a -> (a -> m b) -> m b
```

Como antes, para ser una mónada es requisito indispensable ser un funtor aplicativo. Si antes había que implementar la función prima hermana de `fmap`, ahora hay que implementar la primo-segunda, `(>>=)`. Ahora la función recibe un valor metido en una caja y una función que recibe un valor y devuelve otro metido en una caja. Devuelve esa caja intacta. La función `return` es exactamente igual que `pure`. Veamos algunas implementaciones:

```
instance Monad Maybe where
  return x = pure
  Nothing >>= _  = Nothing
  (Just x) >>= f = f x

instance Monad [] where
  return x = pure
  [] >>= _ = []
  (x:xs) >>= f = f x ++ xs >>= f

instance Monad (Either a) where
  return = pure
  (Left l) >>= _  = Left l
  (Right r) >>= f = f r
```

Las mónadas permiten, entre otras cosas, escribir bibliotecas modulares y trabajar de una forma elegante con errores en Haskell. Por ejemplo, la implementación de una pila usando Maybe para devolver Nothing en caso de errores:

```
data Stack a = Stack [a]

emptyStack :: Maybe (Stack a)
emptyStack = Just (Stack [])

push :: a -> Stack a -> Maybe (Stack a)
push x (Stack xs) = Just (Stack (x:xs))

pop :: Stack a -> Maybe (Stack a)
pop (Stack []) = fail "Empty Stack" -- Generación de error
pop (Stack (x:xs)) = Just (Stack xs)

top :: Stack a -> Maybe a
top (Stack []) = fail "Empty Stack" -- Generación de error
```

```haskell
top (Stack (x:xs)) = Just x

ejemplo :: Maybe (Stack Int)
ejemplo = emptyStack >>= push 1 >>= push 2 >>= pop >>= pop
```

Ha aparecido una nueva función, `fail`. Esta función recibe un String y devuelve un valor monádico. Con la mónada `Maybe`, esta función simplemente devuelve `Nothing`, pero otras mónadas más complejas podrían llevar la traza del error. Por ejemplo, en la biblioteca `parsec` se sigue esta filosofía. Los errores y el estado en en análisis se llevan a cabo con una unión de mónadas (que forman una mónada), cada una ocupándose de su tarea. Lo ideal es adaptar el código para que use cualquier tipo de mónada, dando al usuario de tu biblioteca una gran flexibilidad y facilidad para usar tu código en cualquier parte. La biblioteca `parsec` también permite esto, bajo ciertas condiciones, lo que permite que se desarrollen ciertas funcionalidades encima de ésta que pueden facilitar mucho el análisis. En concreto, se usan las `MonadTransformer`, que permiten la composición de mónadas.

Otro aspecto secundario (pues es mera sintaxis) pero importante en la programación monádica en Haskell es la introducción de la notación `do`, que permite de forma elegante traer la programación imperativa a Haskell. Así, otra posible función ejemplo para nuestro código podría ser:

```haskell
ejemplo :: Stack Int -> Maybe (Stack Int)
ejemplo stack = do
  valor1 <- top stack
  stack  <- pop stack
  valor2 <- top stack
  stack  <- pop stack
  push (valor1 + valor2) stack
```

Saca dos valores de la pila, los suma y deposita el resultado. Para terminar, la notación `do` es solo azúcar sintáctico:
La expresión
`do { v1 <- e1 ; v2 <- e2 ; en }`
se transforma en
`e1 >>= (\v1 -> e2 >>= (\v2 -> en))`.

6.6. Mónadas 'Plus'

Como dice el nombre, las mónadas 'plus' no son más que mónadas ampliadas con dos simples operaciones. Estas dos operaciones serán muy útiles para la programación no determinista que tenemos entre manos. Veamos cómo está definida:

```haskell
class (Alternative m, Monad m) => MonadPlus (m :: * -> *) where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

No hay que prestar especial atención a la clase Alternative, es, en casi todos los casos, exactamente igual que esta clase. Para estar en la clase `MonadPlus`, hay que estar en la clase Monad e implementar dos operaciones. La primera operación, `mzero`, debe ser una función que retorne un valor vacío, por defecto, para esa mónada. La función `mplus` debe ser una función que tome dos valores monádicos y devuelva otro que represente la unión de los dos primeros. La implementación para algunos tipos es como sigue:

```haskell
instance MonadPlus [] where
  mzero = []
  mplus a b = a ++ b

instance MonadPlus Maybe where
  mzero = Nothing
  mplus Nothing b = b
  mplus a Nothing = a
  mplus a b = a
```

Usaremos estas operaciones para implementar algunas características de la programación no determinista, por ejemplo, la elección de dos valores. `a?b` en Sharade se traducirá como
`mplus (return a) (return b)`, el cómputo vacío, cuando no hay ningún resultado, se representará con `mzero`. Esta situación se puede dar en una expresión `case` cuando no se encaja en ningún patrón. Haskell en esos casos hace saltar una excepción. En la biblioteca explicit-sharing se extiende además la clase MonadPlus para permitir otra operación, `share`, para permitir que en una expresión el valor de una variable se comparta entre todas las apariciones de esa variable. Más adelante veremos cual es la mónada que se usa para evaluar estos cómputos no deterministas.

6.7. Mónadas con estado

Podemos pensar las mónadas con estado como mónadas normales pero que además de un valor tienen un estado 'oculto'. Veamos como se define la clase y su implementación para el tipo `State`:

```haskell
class Monad m => MonadState s (m :: * -> *) | m -> s where
  get :: m s
  put :: s -> m ()
```

La función `get` obtiene el estado oculto y lo pone como valor, así lo podremos obtener fácilmente en la notación `do`. La función `put` recibe un valor del tipo `s` (el tipo del estado), lo oculta y encierra el valor `()`. Vamos a ver cómo se implementa la pertenencia a esta clase para un tipo concreto:

```haskell
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses #-}
import Control.Monad.State hiding (State)

data State s a = State { runState :: s -> (a, s) }
```

Como anotación, `FlexibleInstances` y `MultiParamTypeClasses` son dos extensiones del lenguaje de las muchas que hay en Haskell. `FlexibleInstances` permite que las instancias de las clases no tengan que ser obligatoriamente un constructor de tipos y opcionalmente una lista de variables de tipos. Por ejemplo, la instancia `Num (Maybe a)` está permitida pero no `Num (Maybe Int)`, ya que `Int` no es una variable de tipo. La segunda extensión permite que las clases de tipos reciban varios argumentos, como `MonadState`. Así, gracias a estas dos extensiones se puede implementar la instancia `MonadState s (State s)`. La primera permite que aparezca `State s`, ya que sin ella la variable de tipos `s` no puede volver a aparecer y la segunda permite que haya más de un argumento para la clase de tipos.

Esta es la definición de nuestro tipo. Nuestro tipo `State` encierra un estado y un valor en una función. Esta función tiene como objetivo hacer las modificaciones necesarias al estado. Puede ser tan sencillo como dejar el estado intacto tal y como lo recibe. El valor tiene que ser dado desde fuera, igual que es dado desde fuera en Maybe, las listas...

```haskell
instance Functor (State s) where
  -- f :: a -> b
  -- sf :: s -> (a, s)
  -- s, s2 :: s
  -- v :: a
  fmap f (State sf) = State (\s ->
    let (v, s2) = sf s
    in (f v, s2))
```

`fmap` está implementada de la siguiente forma. Se obtiene la función de dentro del `State` que se recibe, `sf` y se crea un nuevo `State` con una nueva función.

Esta función, como todas, recibe un estado, se aplica `sf` a ese estado, se obtiene un valor y un nuevo estado. Tenemos que transformar el valor obtenido con la función `f` y devolverlo conjunto al nuevo estado generado.

```
instance Applicative (State s) where
  pure a = State (\s -> (a, s))

  -- f  :: s ->  (a -> b, s)
  -- sf :: s -> (a, s)
  -- f' :: a -> b
  -- v2 :: a
  -- s, s2, s3 :: s
  (State f) <*> (State sf) = State (\s ->
    let (f', s2) = f s
        (v2, s3) = sf s2
        in (f' v2, s3))
```

En este caso, recibimos una función dentro de un `State`. Tenemos que hacer lo mismo de antes, aplicar la función al valor de la derecha, pero con más cuidado. Para obtener la función, aplicamos `f` a `s`. Una vez conseguida la función, hacemos lo mismo que `fmap`, ¡incluso podríamos copiar y pegar! Si no fuera por los nombres de las variables...

```
instance Monad (State s) where
  return = pure

  -- sf :: s -> (a, s)
  -- f :: a -> State s b
  -- s, s1 :: s
  -- v :: a
  -- f' :: s -> (b, s)
  (State sf) >>= f = State (\s ->
    let (v, s1) = sf s
        State f' = f v
        in f' s1)
```

Puede que este sea más sencillo. Como siempre, tenemos que aplicar la función al estado que viene. Después, aplicamos la función al valor obtenido, obteniendo de nuevo un `State`. Finalmente aplicamos la nueva función al estado anterior.

```
instance MonadState s (State s) where
  get = State (\s -> (s,s))
```

```
  put s = State (\_ -> ((),s))
```

Implementar la nueva clase es más sencillo que todas las demás. Recordemos que `get` tenía que poner el estado como valor sin mutarlo. Recordemos también que `put` tenía que poner el estado que se le pasaba como argumento y vaciar el valor.

Veamos un ejemplo. En este ejemplo, el estado es una lista, representando una pila. El valor puede ir cambiando de tipo a lo largo del programa, como con cualquier otro tipo de mónada:

```
push :: a -> State [a] ()
push x = do
  xs <- get
  put (x:xs)

pop :: State [a] a
pop = do
  (x:xs) <- get
  put xs
  return x

ejemplo :: State [Int] Int
ejemplo = do
  -- ¡`pop` está actuando sobre 0 argumentos! Está usando el
  -- estado oculto, gracias a las funciones `get` y `put`.
  v1 <- pop
  v2 <- pop
  push (v1 + v2)
  pop

startState = [1, 2]
main = print $ runState ejemplo startState
```

Si evaluamos main, se muestra por pantalla `(3,[])`'. El primer valor de la pareja es el resultado de nuestro programa y el segundo, `[]` es el resultado del estado interno de nuestro programa. El estado inicial empezó con dos valores, se extrajeron, se sumaron, se depositó el resultado, y a modo de ejemplo, se sacó el resultado. Como se puede ver, `pop` tiene cero argumentos, la función `ejemplo` tampoco tiene, ¡`pop` está cogiendo los valores del estado interno del programa! Por este motivo, y entre otros, las mónadas con estado son tan potentes. Permiten escribir código Haskell que dependen de un estado que a priori está oculto.

Las `MonadState` tienen una fuerte presencia en este proyecto. Se encargan de implementar de una manera eficiente la compartición de valores no deterministas en mi lenguaje, como explicaré en el siguiente punto. También se encargan de una parte importante del análisis sintáctico. En ese caso, el estado actúa como la cadena de caracteres que faltan por reconocer y los valores van cambiando dependiendo de la estructura sintáctica. Unas veces es una expresión, otras un número y otras el programa entero (y esperemos que ahí el estado sea la cadena vacía, si no hay un error sintáctico, pero de eso se encarga otra mónada).

A modo de nota final, me gustaría aclarar que el tipo `State` como tal no existe en Haskell, realmente es un alias del tipo `StateT s Identity a`. `StateT` es una mónada transformadora. Como ya he explicado de manera superficial en otros puntos, permiten componer varias mónadas distintas, para conseguir un buen desacoplamiento de las tareas que hay que hacer en el programa.

7. Call-time choice y evaluación perezosa

Uno de los problemas que tiene la programación indeterminista en Haskell es que no cuenta con construcciones que permitan la compartición de valores indeterministas en una expresión. Recordemos que la semántica por defecto es la ya comentada 'run-time choice'. Se podría simular el comportamiento de la semántica 'calltime choice' si se extrajesen los valores de las mónadas para luego operar con ellos, pero esto plantea un inconveniente bastante importante, se pierde la evaluación perezosa, ya que al extraer el valor de la mónada estamos forzando su evaluación. Fischer en su trabajo propone una solución a este problema, que comentaremos en el siguiente apartado. En el apartado 7.2 veremos algunos ejemplos del uso de su biblioteca, donde finalmente quedará clara la necesidad de crear un lenguaje para mostrar la combinación de semánticas.

7.1. Biblioteca explicit-sharing

Aunque el paradigma del no determinismo ya está completo en Haskell con las herramientas que he explicado más arriba, en concreto con cualquier tipo que pertenezca a la clase `MonadPlus`, perdemos ciertas características, como la evaluación perezosa. Sin esta característica, no podríamos trabajar con estructuras de datos infinitas o desechar cómputos que que no nos interesen. Vamos a ver una serie de ejemplos para ilustrar esta pérdida de características [8]:

```haskell
import Control.Monad

-- Definición auxiliar
duplicate :: Monad m => m a -> m (a, a)
duplicate a = do
  u <- a
  v <- a
  return (u,v)
-- Definición auxiliar
const' :: a -> a -> a
const' = const

strict_bind :: (MonadPlus m) => m Int -> m (Int, Int)
strict_bind x = do
  v <- x
  duplicate (const' (return 2) (return v))

lazy_share x = do
```

```
v <- share x
duplicate (const' (return 2) v)
```

Si pasamos un valor infinito o que se tarde mucho en calcular o directamente, provoque un error de cómputo a `strict_bind`, no se comportará correctamente, incluso cuando no se está usando para nada el argumento. Esto es debido a la implementación del operador `>>=`, para cualquier tipo de mónada en la que se evalúe la función. En otras palabras, perdemos la evaluación perezosa. Sin embargo, como veremos más adelante, esto no es así en la otra función, `lazy_share`. Con esa conseguimos la evaluación perezosa y sin perder la importante característica de que todas las apariciones de `v` correspondan con el mismo valor.

7.1.1. Leyes de share

La implementación del `share` tiene que cumplir una serie de leyes para garantizar unas características mínimas:

- Fail → `share mzero = return mzero`.
  Compartir un cómputo vacío debe tener el mismo efecto. Es decir, se debe poder compartir un cómputo sin resultados sin que eso afecte al comportamiento del programa.

- Ignore → `share a >>= \x -> b = b`.
  Si la variable `x` no aparece en la expresión de la lambda, el share se debe comportar como si no existiera. Esto garantiza el comportamiento explicado en el ejemplo anterior. Por ejemplo, si `a` fuera un cómputo vacío o no terminante, el cómputo global no debería ser vacío o no terminante si no se usa ninguno de sus resultados.

- Choice → `share (a ? b) = (share a) ? (share b)`.
  Compartir un cómputo no determinista debe ser lo mismo que una elección de las comparticiones. Es decir, si una pieza del cómputo genera múltiples resultados, el resultado debe ser el mismo que si se coge cada uno de esos resultados, se evalúan en la expresión que contiene esa pieza y se juntasen todos los resultados nuevamente generados [11][13]. Es una característica básica de la semántica 'Call-time choice', por lo que es bastante deseable que esta primitiva, `share`, la cumpla.

La tercera ley es fundamental para el correcto funcionamiento de la semántica 'call-time choice'. La diferencia entre las dos semánticas se aprecia bastante bien en términos abstractos a través de una propiedad esencial que verifica 'call-time choice' y no 'run-time choice'. Para formularla, introducimos una notación que explicamos vía ejemplos.

- $[\![e]\!]$ → Es el conjunto de valores en el que se puede reducir la expresión `e`. Por ejemplo, $[\![coin]\!] = \{0, 1\}$.

- $C[e]$ → Indica que la expresión `e` aparece dentro de un contexto sintáctico $C[]$.

La propiedad fundamental de 'call-time choice' es entonces:
$[\![C[e\ ?\ e']]\!] = [\![C[e]]\!] \cup [\![C[e']]\!] = [\![C[e]\ ?\ C[e']]\!]$.
Con palabras, si una pieza en un contexto es indeterminista, semánticamente debería ser equivalente a extraer esa pieza del contexto y hacer la unión de los resultados incorporando cada uno de los valores de la pieza. Explicado con un ejemplo puede quedar más claro:

$[\![duplicate\ (0\ ?\ 1)]\!] = \{0, 2\}$
$[\![duplicate\ 0]\!] \bigcup [\![duplicate\ 1]\!] = \{0\} \bigcup \{2\}$
$[\![duplicate\ 0\ ?\ duplicate\ 1]\!] = \{0\} \bigcup \{2\}$
Esta propiedad la describe F. J. López Fraguas en [11].

Las leyes Fail, Choice e Ignore hacen que se cumplan ciertos comportamientos en las implementaciones que aseguran la evaluación perezosa y la compartición de nodos en una expresión:

- Elección temprana → La elección de un valor se tiene que hacer como si se ejecutara de manera impaciente. Pero no se puede ejecutar impacientemente, por el ejemplo anterior, perderíamos la evaluación perezosa.

- Demanda tardía → Se puede hacer una elección sin necesidad de efectuar el cómputo. Es decir, se elegirá el cómputo que hay que hacer pero no se efectuará hasta que sea realmente necesario, consiguiendo de nuevo la evaluación perezosa.

7.1.2. Simples pero malas implementaciones

Para entender mejor estas leyes e intuiciones, vamos a ver dos formas incorrectas de implementar `share` y ver por qué no serían aptas.

```
import Control.Monad

duplicate :: Monad m => m a -> m (a, a)
duplicate a = do
  u <- a
  v <- a
  return (u,v)

coin :: MonadPlus m => m Int
coin = mplus (return 0) (return 1)

dup_coin_let :: MonadPlus m => m (Int, Int)
dup_coin_let = let x = coin in duplicate x

dup_coin_bind :: MonadPlus m => m (Int, Int)
dup_coin_bind = do
  x <- coin
  duplicate (return x)

dup_coin_share :: MonadPlus m => m (Int, Int)
dup_coin_share = do
  x <- share coin
  duplicate x

share :: MonadPlus m => m a -> m (m a)
share a = return a
```

Esta implementación cumple las leyes 'Fail' e 'Ignore', pero no cumple la ley 'Choice'. El resultado de `dup_coin_share` debería ser {(0,0), (1,1)}, pero tiene las cuatro combinaciones. Esto es debido a que esta implementación comparte sólo el cómputo no determinista, no los valores que genera. Así, en este caso `dup_coin_share` es equivalente a `dup_coin_let`, le pasa el mismo cómputo no determinista.

La siguiente implementación de share sí cumple la ley de 'Choice', pero no cumple 'Ignore' y 'Fail', ya que hace una elección temprana del valor no determinista. Así, en el primer ejemplo si se pasa como argumento un cómputo infinito a `lazy_share` se convertiría en otro cómputo infinito, a pesar de descartar el pri-

mer el argumento. Igualmente, si se pasa un cómputo sin resultados, ¡la función no tendrá resultados! Este comportamiento sí que es alarmante.

```
share :: Monad m => m a -> m (m a)
share a = a >>= \x -> return (return x)
```

7.1.3. Implementación sencilla

Esta es una implementación sencilla que solo es capaz de compartir cómputos no deterministas de enteros respetando las leyes. Este ejemplo sirve como toma de contacto para entender una implementación más cercana a la real, que explicaré en el siguiente punto.

Para combinar la demanda tardía con la elección temprana del cómputo Fischer utiliza la memoización. La idea es atrasar todo lo posible la elección del cómputo y recordar la misma elección cuando se pide el valor real. Para ello, crea un tipo de datos que usa para recordar si un valor ha sido evaluado o no:

```
data Thunk a = Uneval (Memo a) | Eval a
```

También se crea un nuevo tipo, `Memo`, que es el que se encargará de tener toda la lista de los cómputos no deterministas, `Thunk`s.

```
newtype Memo a = Memo {
  unMemo :: [Thunk Int] -> [(a, [Thunk Int])] }
```

`Memo` forma una mónada sobre `a` y además se puede meter en la clase de tipos `MonadState` con `[Thunk Int]` como estado. Es decir, nuestros programas no deterministas tendrán un 'estado oculto', que unido a la memoización, se encargará de recordar las decisiones sobre la compartición. Veamos como se implementa todo esto:

```
instance Functor Memo where
  fmap f m = Memo (\ts ->
    map (\(a, ts') -> (f a, ts')) (unMemo m ts))

instance Applicative Memo where
  pure = return
  f <*> m = Memo (\ts ->
  concatMap (\(a, ts') ->
    map (\(f, ts'') -> (f a, ts'')) (unMemo f ts'))
    (unMemo m ts))
```

Las instancias de Functor y Applicative no son importantes, pero es necesario implementarlas debido a que tenemos que meter `Memo` en la clase de tipos `Monad`.

Recordemos que `fmap` recibía una función y el valor monádico. Tenemos que transformar el valor metido en la 'caja' aplicando la función. Pero esta vez, ese valor está en el resultado de la función que tiene `Memo`. Por lo tanto, aplicamos esa función, `unMemo m ts` y simplemente hacemos un map sobre la lista de parejas aplicando la función al primer valor.

Applicative es la más complicada. Recibimos una función encerrada en un valor monádico y un valor monádico. Tenemos que obtener el valor encerrado dentro del segundo valor, lo obtenemos como antes. Después, usamos la lista de `Thunk` que hemos obtenido para obtener la función, que aplicamos a cada valor. Mientras tanto hemos obtenido otra lista de `Thunk`, que formará parte del resultado final.

```
instance Monad Memo where
  return x = Memo (\ts -> [(x,ts)])
  m >>= f = Memo (\ts -> concatMap (\(x,ts) ->
    unMemo (f x) ts) (unMemo m ts))

instance Alternative Memo where
  empty = mzero
  (<|>) = mplus

instance MonadPlus Memo where
  mzero = Memo (\ts -> [])
  a `mplus` b = Memo (\ts -> unMemo a ts ++ unMemo b ts)

instance MonadState [Thunk Int] Memo where
  get = Memo (\ts -> [(ts,ts)])
  put ts = Memo (\_ -> [((),ts)])
```

Para el operador `>>=`, tenemos que sacar el valor encerrado aplicando la función a la lista que recibimos. Después, como `f` devuelve otro `Memo`, usamos la lista generada para obtener la lista final.

La implementación de `Alternative` es idéntica a la de `MonadPlus`, que simplemente concatena las listas generadas.

`MonadState` se parece mucho a los otros ejemplos. Con `get` simplemente tenemos que poner el valor como estado y con `put` tenemos que poner el argumento que nos dan como estado, además de limpiar el valor, poniendo `()`.

```
share :: Memo Int -> Memo (Memo Int)
share a = memo a
```

```haskell
memo :: Memo Int -> Memo (Memo Int)
memo a = do
-- Obtenemos la lista
  thunks <- get
  let index = length thunks
  -- Metemos el nuevo valor al final
  put (thunks ++ [Uneval a])
  return $ do
    -- Obtenemos de nuevo la lista
    thunks <- get
    -- Probablemente haya cambiado, por eso es importante
    -- recordar (memoizar) la decisión tomada, `index`
    case thunks !! index of
      Eval x -> return x -- Devolver el valor
      Uneval a -> do
        -- Aquí se generaran realmente todos los valores
        -- posibles
        x <- a
        thunks <- get
        -- Sustituir el antiguo valor como valor "evaluado"
        let (xs,_:ys) = splitAt index thunks
        put (xs ++ [Eval x] ++ ys)
        -- Devolver el valor
        return x
```

Creo que los comentarios entre las líneas del código ya son explicativos, pero aún así lo explicaré con otras palabras. El estado 'oculto' es una lista de valores que pueden estar sin evaluar o evaluados, `[Thunk Int]`. Con `get` somos capaces de obtener esta lista. Incorporamos el nuevo valor y guardamos su posición, `index`. Devolvemos una 'función' (de 0 argumentos), que, cuando sea evaluada la primera vez (elección temprana), nos encontraremos con que el valor en esa posición no está evaluado. Entonces se efectúa el cómputo (evaluación tardía) y actualizamos la lista, depositándola de nuevo en el estado 'oculto'. Justo en la línea `x <- a` surgirán todos los valores posibles. Esa línea actúa como 'fuente' de todos los resultados. Para entender el ejemplo de la fuente, un ejemplo:

```haskell
f :: [Int]
f = do
  v <- [1,2,3,4]
  return (v + 1)
```

La línea `a <- [1,2,3,4]` actúa como fuente, `v` irá tomando los valores y

se devolverán incrementados en uno. Los cómputos solo se efectúan cuando es necesario, proporcionando una evaluación perezosa bastante sofisticada.

Como ya he dicho al principio de este punto, esta implementación respeta las tres leyes básicas de las que habla Fischer, pero solo es capaz de compartir enteros. En el siguiente punto explicaré una implementación muy parecida a la que usa Fischer en su biblioteca.

7.1.4. Implementación real

En la implementación real, Fischer declara el tipo `Lazy`, que tendrá la misma función que `Memo` del ejemplo anterior. También usa un nuevo tipo, `Store` que usará como estado 'oculto', que tendrá la misma función que la lista de `Thunk Int`. Para implementar `Lazy` como instancia de `MonadState` y las demás clases, delega en otra clase de tipos algunas operaciones, `Nondet`. Es decir, está implementado de tal manera que cualquier tipo `a`que pertenezca a esta clase, `Lazy a` pertenecerá a las clases de tipos `MonadPlus`, `MonadState`... En concreto, el tipo `Set a` está metido en esa clase, por lo que podremos observar casi cualquier cómputo en esta estructura de datos. Vamos a ver las definiciones, esta vez no pondré la implementación porque es bastante parecida a los ejemplos:

```
class Nondet n where
  failure :: n
  (?)     :: n -> n -> n

newtype Lazy n a = Lazy {
    fromLazy :: (a -> Store -> n) -> Store -> n
  }

data Store = Store { nextLabel :: Int, heap :: M.IntMap Untyped }
```

En esta implementación, `Store` es como la lista de `Thunk`, guarda los resultados en un mapa y proporciona la claves para recordar la decisión. Fischer implementa funciones auxiliares para actualizar y observar esta estructura, como inserciones, eliminaciones, consultas... `Lazy` es la mónada sobre la que se realizarán todas las operaciones. Veamos la implementación del share:

```
class Shareable m a where
  shareArgs :: Monad n =>
    (forall b . Shareable m b => m b -> n (m b)) -> a -> n a

class MonadPlus s => Sharing s where
  share :: Shareable s a => s a -> s (s a)
```

```haskell
instance Nondet n => Sharing (Lazy n) where
  share a = memo (a >>= shareArgs share)

memo :: MonadState Store m => m a -> m (m a)
memo a = do
  key <- freshLabel
  return $ do
    thunk <- lookupValue key
    case thunk of
      Just x  -> return x
      Nothing -> do
        x <- a
        storeValue key x
        return x
```

Lo primero es la existencia de una clase de tipos que se pueden compartir. Fischer mete en esta clase casi todos los tipos primitivos de Haskell. Está pensada para compartir recursivamente estructuras de datos más complejas, donde hay que compartir cada elemento y además las sucesivas colas de la lista. Después está la clase de tipos `Sharing`, donde gracias a la implementación de Fischer, están todos los tipos `Lazy a` si `a` pertenece a la clase de tipos `Nondet`. La implementación del `share` se basa en compartir recursivamente el cómputo y después memoizarlo. La memoización es bastante parecida a la de antes. Se obtiene una clave (antes era la longitud de la lista) y se devuelve una función (de cero argumentos) que cuando sea evaluada, 'recordará' la clave y el cómputo a compartir. Si es la primera vez que se evalúa, se encontrará al consultar el estado 'oculto' que no está, generando los resultados e insertándolos. Si ya ha sido generado, tan solo hay que devolverlo.

7.2. Codificación directa

Veamos unos ejemplos de cómo usar la biblioteca de Fischer para mostrar cómo se pueden mezclar los dos tipos de semántica en Haskell sin perder evaluación perezosa:

```haskell
import Control.Monad
import Control.Monad.Sharing
import Data.Monadic.List

coin :: Sharing s => s Int
coin = (return 0) `mplus` (return 1)
```

```haskell
duplicate :: Sharing s => s Int -> s Int
duplicate x = do
    u <- x
    v <- x
    return (u + v)

f :: Sharing s => s Int -> s Int -> s Int
f x y = do
  x' <- share x
  duplicate y

e1, e2, e3 :: Sharing s => s Int
e1 = duplicate coin
e2 = do
    c <- share coin
    duplicate c
e3 = f undefined coin
```

La primera definición se corresponde con `coin = 0 ? 1` de Sharade. Como `mplus` opera con dos valores monádicos, es necesario meter el 0 y el 1 en sus respectivas mónadas para luego juntarlas. La definición de `duplicate` se corresponde con la definición en Sharade de `duplicate x = x + x`. Para conseguir las propiedades de 'run-time choice' en Haskell, es necesario 'extraer' dos veces un valor del argumento. La función `f` la pongo de ejemplo para mostrar la evaluación perezosa y demostrar que la ley 'Fail' e 'Ignore' funcionan en el tercer ejemplo.

El primer ejemplo llama a la función `duplicate` directamente con `coin`. Por lo tanto, como el valor no está compartido, se generarán las cuatro posibilidades (dos repetidas). El siguiente ejemplo muestra cómo conseguir la semántica 'call-time choice'. Como el valor de `c` está ahora compartido, `duplicate` solo generará dos resultados. Por último, el tercer ejemplo muestra la evaluación perezosa. Como la función `f` ignora el primer argumento, el cómputo no da lugar a error, aunque se intente hacer una compartición con él.

```haskell
import Control.Monad
import Control.Monad.Sharing
import Data.Monadic.List

mconcatenate :: Sharing s => s (List s a) -> s (List s a)
                          -> s (List s a)
mconcatenate mxs mys = do
    xs <- mxs
```

```
    case xs of
        Nil -> mys
        Cons x mxs -> cons x (mconcatenate mxs mys)

mreverse :: Sharing s => s (List s a) -> s (List s a)
mreverse mxs = mreverse' nil mxs where
    mreverse' rvs mxs = do
        xs <- mxs
        case xs of
            Nil -> rvs
            Cons mx mxs -> mreverse' (cons mx rvs) mxs

letter :: Sharing s => s Char
letter = (return 'a') `mplus` (return 'b') `mplus` (return 'c')

word :: Sharing s => s (List s Char)
word = nil `mplus` (cons letter word)

palindrome :: Sharing s => s (List s Char)
palindrome = do
    w <- share word
    let l = cons letter nil
    w `mconcatenate` (l `mplus` nil) `mconcatenate` (mreverse w)
```

En este ejemplo las cosas se empiezan a complicar. Como todos los argumentos son monádicos, no se puede trabajar con los valores de una manera directa y clara como se haría tradicionalmente. Siempre hay que sacar los valores del interior de la mónada para manipularlos, ensuciando mucho la sintaxis. Esta es una de las razones por las que propongo Sharade, para poder trabajar con el indeterminismo de una manera fácil y limpia, con el añadido de la combinación de las dos semánticas.

Las funciones `mconcatenate` y `mreverse` no tienen ningún tipo de indeterminismo, pero tienen que tratar con la programación monádica. Podrían recibir un argumento indeterminista, pero no están pensadas para esa situación. Lo primero que hay que hacer es extraer el valor de la mónada y evaluarlo, distinguiendo entre el caso base y el caso recursivo. Hay que mencionar que `nil` y `cons` son constructoras auxiliares del tipo lista. Tienen la función añadida de meter el resultado en un valor monádico. La definición de `letter` es parecida a la de `coin`, hay que meter cada valor en una mónada y juntarlas. La definición de `word` genera infinitos resultados indeterministas de listas finitas e infinitas. Una palabra puede ser la lista vacía o una letra seguida de otra palabra. En la definición de `palindrome`

se puede apreciar la compartición. Como `w` aparece dos veces en la expresión, es necesario que compartan el valor, si fueran independientes `palindrome` generaría todas las palabras posibles, es decir, no tendría ninguna diferencia con `word`. Para permitir los palíndromos impares la pieza intermedia puede contener una letra o ser la lista vacía. El palíndromo se definiría en Sharade como

```
palindrome = choose w = word in
             let l    = Cons letter Nil in
             mconcatenate w (mconcatenate (l ? Nil) w) ;
```

habiendo definido antes las funciones auxiliares, claro.

Estos ejemplos muestran cómo es posible expresar directamente en Haskell la programación con indeterminismo combinando 'call-time choice' y 'run-time choice'. También deja claro que esta tarea en Haskell es complicada, propenso a errores y difícil de entender para nuevos ingresados en la materia.

Sin embargo, como ya he dicho antes, Sharade es capaz de reducir esta complejidad, descargando bastante la sintaxis y ofreciendo una abstracción de la programación monádica, indeterminismo y combinando los dos tipos de semánticas muy potentes.

8. Biblioteca parsec

La biblioteca Parsec nos permite analizar sintácticamente un lenguaje mediante la creación de funciones de orden superior de una manera muy expresiva. Se puede crear un analizador sintáctico descendente sin la necesidad de crear un analizador léxico directamente. También ofrece muchas primitivas muy útiles, como reconocer números, identificadores, especificar nombres reservados, el formato de los comentarios, especificar la prioridad de los operadores, tener en cuenta la indentación...

Parsec implementa todas estas operaciones haciendo uso de la mónada con estado '`ParsecT s u m a`'. Además, es una `MonadTrans`. Es un tipo que permite un flujo de caracteres `s`, un tipo de estado de usuario, `u`, una mónada `m` y un valor de retorno `a`. Este valor de retorno serán las sucesivas partes del lenguaje, un número, una expresión, un patrón... `ParsecT` es una mónada con estado, donde el estado es la porción de la entrada que queda por consumir. Las primitivas de la biblioteca son las que se encargan de actualizar este estado 'oculto'.

Vamos a dar algunos ejemplos de cómo sacar el máximo partido a las primitivas de la biblioteca:

```
import Text.Parsec.Indent
import qualified Text.Parsec.Token as Tok

langDef = Tok.LanguageDef
  { Tok.commentStart    = "{-"
  , Tok.commentEnd      = "-}"
  , Tok.commentLine     = "--"
  , Tok.nestedComments  = True
  , Tok.identStart      = letter
  , Tok.identLetter     = alphaNum <|> oneOf "_'"
  , Tok.opStart         = oneOf ":!#$%&*+./<=>?@\\^|-~"
  , Tok.opLetter        = oneOf ":!#$%&*+./<=>?@\\^|-~"
  , Tok.reservedNames   = ["choose", "let", "in", "case", "of"]
  , Tok.reservedOpNames = [",", "->", "\\", "=", "?", "&&", "||",
                           "<", "<=", ">", ">=", "==", "!=", "+",
                           "-", "*", "/"
                          ]
  , Tok.caseSensitive   = True
  }
lexer = Tok.makeTokenParser langDef

parens = Tok.parens lexer
```

```haskell
reserved = Tok.reserved lexer
reservedOp = Tok.reservedOp lexer
identifier  = Tok.identifier lexer
integer = Tok.integer lexer

expression :: ParsecT String () Identity Expression
expression = do
  left <- identifier
  reservedOp "+"
  right <- identifier
  return (Add left right)

parseExpr :: String -> Either ParseError Expr
parseExpr s = runParser expression "" s
```

Suponiendo que existiera el tipo `Expr`, este ejemplo es capaz de reconocer y de crear el árbol de expresión de secuencias como 'a + b'. Las primitivas como `identifier`, `reservedOp` y `reserved` permiten reconocer identificadores, operadores reservados y palabras reservadas tal como están descritas en el 'analizador léxico'. También ofrece primitivas algo mas complejas como `parens`, que dada una función que es capaz de reconocer algo del tipo `a`, es capaz de reconocer lo mismo pero rodeado de paréntesis.

Para permitir el backtracking en el análisis, por ejemplo de un lenguaje LL(0), existe la primitiva `try`, que intenta utilizar el analizador pasado por parámetro. Si ese analizador tiene éxito, no hace nada, pero en caso de que falle, actualiza el estado 'oculto' y falla, para poder utilizar otro analizador.

El lenguaje también ofrece primitivas para operadores binarios e infijos de distinta asociatividad. No he aprovechado estas primitivas en mi implementación debido a la poca flexibilidad que ofrecían para generar expresiones.

Gracias a la gran flexibilidad del tipo `ParsecT` se pueden crear gran cantidad de extensiones para la biblioteca. Dos extensiones que he utilizado son `parsec3-numbers` y `indents`. La extensión `parsec3-numbers` permite analizar números enteros y reales en gran cantidad de formatos, con más potencia que el analizador por defecto de `Parsec`. La extensión `indents` es capaz de tener en cuenta la indentación, muy útil para analizar ciertas expresiones, como 'case of', un grupo de definiciones de funciones o un conjunto de expresiones 'choose' o 'let'.

9. Implementación de Sharade

Toda la implementación se encuentra presente en el siguiente repositorio de GitHub: https://github.com/ManuelVs/Sharade.

La implementación cuenta con 14 módulos y alrededor de mil líneas de código Haskell. La mayor parte de las líneas de código corresponden a la traducción y el análisis de tipos. Este relativamente pequeño tamaño resulta, en mi opción, engañoso para apreciar una gran dificultad. La programación en Haskell, y especialmente la programación monádica, puede ser extraordinariamente concisa y expresiva, requiriendo eso sí, una pericia técnica del programador no trivial de adquirir. Por otra parte, también hay que destacar que la implementación hace uso de una versión modificada de la biblioteca `explicit-sharing` de Fischer para hacerla funcionar con las versiones actuales de Haskell. Realizar esa modificación nos ha exigido comprender en profundidad la biblioteca de Fischer.

La implementación de `Sharade` se encuentra amparada por la licencia de software libre `MIT`. La versión modificada de la biblioteca de Fischer es completamente dominio público, igual que la versión original.

9.1. Sintaxis concreta de Sharade

En la sección 5.4 presentamos las diferentes construcciones sintácticas consideradas en Sharade en un formato que podríamos considerar de sintaxis abstracta. Damos aquí la sintaxis concreta en forma de reglas gramaticales que pueden servir ya de base directa a la construcción de un analizador sintáctico, que describimos en la sección 9.2.

Program → `{ FunDecl }`

FunDecl → `Identifier { Identifier } "=" Expression ";"`

Expression → `E0`

E0 → `E1 "?" E0`

E0 → `E1`

E1 → `E2 "&&" E1 | E2 "||" E1`

E1 → `E2`

E2 → `E3 RelOperator E2`

E2 → `E3`

```
      E3 → E4 "+" E3 | E4 "-" E3

      E3 → E4

      E4 → E5 "*" E4 | E5 "/" E4

      E4 → E5

      E5 → Lambda | ChooseIn | LetIn | CaseExpr
         | FExp | Pair | "(" E0 ")"

    FExp → Identifier { AExp }

    AExp → LitExpr | Variable | "(" E0 ")"

  Lambda → "\" Identifier "->" Expression

ChooseIn → "choose" Identifier "=" Expression "in" Expression

   LetIn → "let" Identifier "=" Expression "in" Expression

    Pair → "(" Expression "," Expression ")"

CaseExpr → "case" Expression "of" { CaseMatch }

CaseMatch → Pattern "->" Expression ";"

 Pattern → Constructor { Identifier }

Constructor → UpperCase { AlphaNum }
```

Un identificador válido es cualquier cadena de caracteres alfanumérica que empiece por un carácter. El lenguaje también permite comentarios de una línea y multilínea, igual que Haskell. Los comentarios multilínea empiezan con '`{-`' y terminan con '`-}`'. Los comentarios de una sola línea empiezan por '`--`'. La sintaxis no permite la creación de nuevos operadores infijos por el usuario, al contrario que Haskell [14]. La sintaxis está claramente influenciada por Haskell y Curry. En la sintaxis se incluyen también las primitivas necesarias para expresar el no determinismo con compartición y sin compartición. El operador `?` expresa la elección de dos valores no deterministas y la estructura `choose ... in ...` expresa una nueva ligadura de una expresión a un identificador que compartirá el valor en todas sus apariciones, cumpliendo la propiedad que Francisco J. Fraguas muestra en 'A simple rewrite notion for call-time choice semantics' [11].

9.2. Analizador sintáctico con parsec

Como ya hemos visto, la biblioteca `parsec` permite especificar una gramática L-atribuida aprovechando todas las características de Haskell. Un inconveniente que tiene es que en análisis es descendente, por lo que a la hora de la implementación hay que transformar la gramática para evitar la recursión a izquierdas. En mi implementación no he hecho un cálculo de los directores, por lo que el análisis se basa en un backtracking LL0.

La implementación es bastante sencilla, pero hace un uso interesante de las características de orden superior de Haskell. Tan solo quiero mostrar una pequeña parte de la implementación para enseñar cómo he llevado a cabo la transformación de la gramática de atributos para poder analizar descendentemente usando dichas características de orden superior de Haskell.

```
createExpr :: String -> Expr -> Expr -> Expr
createExpr op lexp rexp = (App (App (Var prefixNot) lexp) rexp)
  where prefixNot = "(" ++ op ++ ")"

expr' :: Int -> IParser Expr
expr' 0 = do
  le <- expr' 1
  cont <- expr'' 0
  return (cont le)

expr'' :: Int -> IParser (Expr -> Expr)
expr'' 0 = do
    reservedOp "?"
    rexp <- expr' 1
    cont <- expr'' 0
    return (\lexp -> createExpr "?" lexp (cont rexp))
  <%> return (\e -> e)
```

Esta parte del código corresponde al análisis de las expresiones con un operador infijo de prioridad cero, que es la máxima prioridad. En este caso es solo corresponde al operador `(?)`, origen del indeterminismo de mi lenguaje.

Como se puede ver, las definiciones auxiliares en vez de obtener un valor heredado con la expresión, como se haría en un analizador descendente tradicional, devuelven una función que espera dicha expresión. Es decir, las definiciones auxiliares analizan una función de Haskell. Esto puede sonar raro, pero no lo es teniendo en cuenta que empiezan reconociendo un operador infijo, por lo que la expresión que falta es la parte izquierda del operador. La otra opción, la que viene después del operador `<%>` es el caso base. En este caso se devuelve la función

identidad, análogo a devolver el valor heredado. La implementación de esta parte se puede encontrar en el módulo `Sharade.Parser`.

9.3. Análisis de tipos

Como se ha indicado varias veces, la implementación de Sharade procede por traducción a un programa Haskell cuya ejecución recoge la funcionalidad esperada del programa original. La traducción utiliza intensivamente la programación monádica, como se ha explicado en el apartado 7.2. El programa traducido, al ser Haskell, debe estar bien tipado. En un primer momento se podría pensar en delegar en Haskell el análisis de tipos, reportando su propio interpretador un error de tipos si fuese necesario. Pero debido a la naturaleza de la traducción (al uso intensivo de la programación monádica), se incurre fácilmente en ambigüedades de tipos, por lo que Haskell rechaza el programa con un caso particular de error de tipos. Para evitar esta situación, es necesario realizar un análisis de tipos sobre el programa en Sharade, con el clásico sistema de Hindley-Milner [15]. En el apartado en el que se explica la traducción a Haskell indicaré cómo se transforman estos tipos inferidos a Haskell.

Veamos los siguientes ejemplos para entender las situaciones de ambigüedad de tipos y cómo solucionarlas:

```
import Control.Monad

zero = return 0

coin = (return 0) `mplus` (return 1)
```

Si se intenta interpretar esto en Haskell dará un error, debido a la ambigüedad de tipos. Haskell podría elegir una mónada cualquiera para analizar el tipo, pero eso quebraría el supuesto de 'mundo abierto'. Si se añadiese una nueva instancia que se ajustase mejor a la situación, el código que has escrito podría dejar de funcionar o ni siquiera compilar. Veamos el segundo ejemplo donde se ve más claro [12]:

```
main :: IO ()
main = putStrLn (show (yesno 12))

class YesNo a where
  yesno :: a -> Bool

instance YesNo [a] where
  yesno [] = False
```

```
  yesno _  = True

instance YesNo Int where
  yesno 0 = False
  yesno _ = True
```

Este ejemplo no compilará. No sabe qué instancia coger para evaluar `yesno 12`. Es evidente, con solo estas líneas de código, que la única instancia que encaja es `YesNo Int`. Pero si en el futuro se añade la instancia `YesNo Integer`, habría ambigüedad e incluso podría cambiar el comportamiento esperado del programa. Esto es lo que evita el supuesto del mundo abierto. Si no se sabe que puede haber otra instancia que encaje, el sistema de tipos dará un error de ambigüedad.

Esto se puede solucionar permitiendo que las expresiones traducidas sean polimórficas, es decir, permitiendo que se puedan evaluar en la mónada que se desee. Para conseguir esto es necesario especificar directamente los tipos, por lo que es necesario un análisis de tipos de mi lenguaje y traducirlos correctamente a Haskell. Veámoslo en el ejemplo:

```
import Control.Monad

zero :: MonadPlus m => m Int
zero = return 0

coin :: MonadPlus m => m Int
coin = (return 0) `mplus` (return 1)
```

De alguna manera se está indicando a Haskell que el programa es correcto evaluando dichas expresiones en cualquier instancia de la clase `MonadPlus`, como puede ser la constructora de las listas o la mónada que usa Fischer.

9.3.1. Traducción de tipos a Haskell

Tras inferir los tipos de las funciones en Sharade, es necesario traducirlos a Haskell. Como ya se ha dicho en anteriores ocasiones, cada tipo en Haskell debe ser monádico. En la traducción existen dos excepciones, que son la traducción del tipo de las listas y del tipo de las parejas.

El tipo de una función de Sharade `f :: t` se traduce a Haskell como `f :: Sharing s => s (translate t)`. El tipo `t` puede caer en uno de los siguientes casos:

- Un tipo primitivo `p` se traduce como `s p`.

- Una variable de tipos `a` se traduce como `s a`.

- El tipo funcional, `a -> b`, se traduce como
  `s (translate a -> translate b)`.

- El tipo de las listas, `[a]`, se traduce como
  `s (List s (translate' a))`.

- El tipo de las parejas, `(a, b)`, se traduce como
  `s (Pair s (translate' a) (translate' b))`

La función de traducción `translate'` se comporta como `translate` pero sin poner delante el tipo monádico en el primer nivel de recursión, llamando a `translate` inmediatamente.

Los tipos de las listas y de las parejas se traducen de esta manera debido a que estos tipos en Haskell gestionan el no determinismo (y las mónadas) dentro de su implementación. Por ejemplo, la cola de las listas es un valor indeterminista, lo que no se conseguiría si simplemente fuese una lista de valores indeterministas (`[s a]`). Esto tiene fuertes y buenas repercusiones en el funcionamiento de la evaluación perezosa de Sharade.

9.3.2. Inferencia de tipos Hindley-Milner

La familia de tipos de Hindley-Milner tiene la fantástica propiedad de tener un simple algoritmo para determinar los tipos de una sintaxis sin tipos. El algoritmo se basa en unas simples reglas para obtener el tipo de una expresión. El elemento principal de este algoritmo es la 'unificación' de variables de tipos. Las reglas son las siguientes [16][15][17]:

- T-Var: Si '`x : a`' pertenece al ámbito de tipos, entonces del ámbito de tipos se puede deducir '`x : a`'.

- T-App: Si '`e1 : t1 -> t2`' y '`e2 : t1`' pertenecen al ámbito de tipos, entonces se puede deducir que `e1 e2 : t2`.

- T-Lam: Si con el ámbito de tipos y '`x : t1`' se puede deducir '`e : t2`', entonces '`\x . e : t1 -> t2`'.

- T-Gen: Si del ámbito de tipos se deduce '`e : a`' y la variable de tipo '`b`' no está presente en el ámbito, entonces '`e : forall b . a`'.

- T-Inst: Si del ámbito de tipos se deduce '`e : a`' y `b` es una instancia de '`a`', entonces del ámbito de tipos también se deduce '`e : b`'.

El ámbito de tipos es una colección de expresiones y sus tipos. La regla `T-Var` es básica. La regla `T-App` caracteriza la aplicación de una expresión de tipo funcional

a otra expresión. Si la parte derecha es del tipo que la función espera, entonces el tipo de la aplicación es el resultado del tipo funcional. `T-Lam` especifica el tipo de una función (expresión lambda). Una expresión `(\x -> e)` tiene un tipo tipo funcional, donde el primer argumento debe tener el mismo tipo de '`x`' y el resultado tiene el tipo de '`e`'. `T-Gen` expresa la generalización de un tipo. Por ejemplo, decir `a : Int` es equivalente a decir que `a : forall b . Int`, ya que '`b`' no aparece en la parte derecha. `T-Inst` caracteriza la instanciación. Por ejemplo una posible instanciación de la función `id : a -> a` es `id : Int -> Int`.

La unificación de dos tipos es el proceso de inferir si estos dos tipos pueden confluir en uno solo, creando una sustitución en el proceso. Por ejemplo, los tipos '`a`' y '`Char`' se pueden unificar y dan como resultado la sustitución `[a/Int]`. Otra sustitución válida sería `a -> a ~ Int -> Int : [a/Int]`, para la función identidad. Se puede dar el caso que existan substituciones infinitas. Por ejemplo, si intentamos `a ~ a -> b`, tenemos la sustitución '`[a/a->b]`'. Si aplicamos esa sustitución, siempre obtendremos un tipo más grande, debido a que '`a`' aparece en la parte derecha de la sustitución. La única sustitución posible sería `a/((..((a->b)->b)..)->b)`, pero ni Haskell ni Curry ni Sharade tienen tipos infinitos. Debido a esto, la unificación tiene una precondición conocida como 'occurs-check'. Si se va a unificar '`a`' con '`b`', '`a`' no debe aparecer en '`b`' (ni viceversa), ya que terminaría en una sustitución infinita. Esta parte de la implementación se puede encontrar en el módulo `Sharade.Translator.Sharade`.

9.4. Traducción a Haskell

La última etapa de la compilación es la generación de código Haskell. Para explicar la traducción me basaré en la sintaxis abstracta presentada en 5.4. En la traducción de cualquier expresión se cumple un invariante, las subexpresiones generadas son expresiones Haskell monádicas, por lo tanto nuestra función de traducción siempre debe generar una expresión monádica. Esto es necesario para tener presente en todo momento el indeterminismo. De esta manera, las expresiones funcionales pasan a estar dentro de mónadas, por lo que para traducir la aplicación de una función es necesario definir un operador auxiliar que se encargue de aplicar la función monádica a un valor monádico:

```
infixl 4 <#>
(<#>) :: (Sharing s) => s (s a -> s b) -> s a -> s b
f <#> a = f >>= (\f' -> f' a)
```

Este operador recibe una función atrapada en una mónada (que recibe valores monádicos) y un valor monádico. Aplica la función indeterminista (puede que sean varias) al valor indeterminista (puede que sean varios) y devuelve el resultado. El

resultado puede ser de un tipo primitivo o ser un tipo funcional (encerrado en una mónada). Así, este operador se puede aplicar varias veces.

Veamos caso a caso como funciona la traducción de una expresión. Me he ahorrado los paréntesis que serían necesarios en Haskell para facilitar la lectura.

- l ⇒ Un literal se traduce como `return l`

- x ⇒ Dependiendo de la variable se traduce de una forma u otra. Por ejemplo, el operador `(+)` se traduce como `mAdd`, el operador `(?)` como `mPlus`. Un identificador simple se traduce sin modificación.

- C ⇒ Una constructora es un caso especial de variable, existe una traducción para cada constructora disponible. Por ejemplo, `Cons` se traduce por la función auxiliar `cons`.

- $e1\ e2$ ⇒ Ambas subexpresiones son monádicas, por lo tanto hay que usar el operador explicado antes, `translate e1 <#> translate e2`.

- $\backslash x \rightarrow e$ ⇒ Una función lambda es una función normal, tiene que estar metida en un valor monádico. Por lo tanto, se usa la función `return` sobre la traducción de la expresión: `return (\x -> translate e)`.

- $e1\ ?\ e2$ ⇒ Esta expresión cae en los casos anteriores, traducción de una variable y dos aplicaciones funcionales. Su función auxiliar es `mPlus`.

- $(e1,\ e2)$⇒ Parejas. Igual que el anterior, se traduce una constructora y dos aplicaciones funcionales. Su constructora en Haskell es `mPair`.

- $choose\ x = e1\ in\ e2$ ⇒ Compartición de una variable o expresión. Hay que usar la primitiva `share` sobre la expresión y añadir al ámbito la nueva variable. He decidido usar una expresión lambda para conseguir este objetivo y el operador `(>>=)`. Así, la traducción se hace de la siguiente manera: `share (translate e1) >>= (\x -> translate e2)`

- $let\ x = e1\ in\ e2$ ⇒ Una expresión `let` se traduce sin ninguna complicación, se puede usar la misma estructura de Haskell:
  `let x = translate e1 in translate e2`.

- $e1\ (infixOp)\ e2$ ⇒ Una expresión con un operador infijo. Como antes, debido a la representación interna de las operaciones infijas, se traduce como una variable y dos aplicaciones funcionales.

- $case\ e\ of\ t1 \rightarrow e1; \ldots tn \rightarrow en;$ ⇒ Una expresión `case` es complicada de traducir, ya que hay un caso especial en el que Haskell no puede deducir

los 'tipos intermedios' de las expresiones, por culpa de ambigüedades en las clases de tipos, dando un error en tiempo de compilación. Para traducir esta expresión, uso una función auxiliar llamada `translateMatch`, que traduce cada uno de los casos del `case`. Así, la expresión se traduce como

```
translate e >>= (\pec -> case pec of {
translateMatch (t1 -> e1); ...; translateMatch (tn -> en);
_ -> mzero;})
```

Veamos cual es el funcionamiento de `translateMatch`:

- $C\ x1\ \ldots\ xn\ \rightarrow\ e$ $\rightarrow$ Un patrón estándar con la constructora y $n >= 0$ variables se traduce sin complicaciones. Se puede usar la misma estructura que en Haskell. `C x1 ... xn -> translate e`.
- $v\ \rightarrow\ e$ $\rightarrow$ Una simple variable en el patrón. Aquí surge la complicación y además un caso especial. Hay que tener en cuenta si la variable `v` aparece en la expresión `e`.
  - Si aparece, hay que volver a meter este valor en una mónada y crear un nuevo ámbito de variables con una lambda para sustituir el anterior nombre. Así, este caso se tiene que traducir como: `v -> (\v -> translate e) (return v)`.
  - Si no aparece, para evitar que la inferencia de tipos de Haskell se encuentre con ambigüedades, hay que traducir este caso de la siguiente forma: `_ -> translate e`.

Por último, para traducir una función hay que traducir su tipo y su expresión asociada. Una función con argumentos se puede entender como una secuencia de expresiones lambda. Por lo tanto, traducir `f = e` consiste en traducir su tipo, `f :: translateType f` y su expresión, `f = translate e`. Las funciones con más de un argumento se traducen como una sucesión de expresiones lambda.

La traducción es mejorable, ya que recae muchísimo en el uso de la programación monádica para reflejar el indeterminismo, pero se pueden llevar a cabo optimizaciones para mejorar el rendimiento. El impacto en el rendimiento de la primera mejora puede ser cercana al 50 %, como comentaremos más adelante en el apartado de conclusiones.

La primera tarea se puede realizar para evitar el uso excesivo de la programación monádica es tener en cuenta una consecuencia lógica de las leyes del 'call-time choice' para eliminar por completo la compleja estructura de las funciones monádicas. Recordemos que actualmente una función de mi lenguaje traducida a Haskell tiene el tipo `s (s a1 -> s (s a2 -> ... (s an)...))`. Este tipo es equivalente a `s a1 -> ... -> s an`. Es decir, eliminamos recursivamente del tipo de la función la mónada que las envuelve. Esto es así ya que, según las leyes del 'call-time choice', siendo `f` y `g` funciones del mismo tipo, $[\![C[(f?g)x]]\!] = [\![C[fx]]\!] \cup [\![C[gx]]\!]$. Es

decir, claramente nos da igual elegir entre las dos funciones y juntar los resultados que aplicar las dos funciones por separado y juntar los resultados. En la semántica 'run-time choice' también se cumple si consideramos que todas las apariciones de un identificador son independientes (que es básicamente el comportamiento de esta semántica).

La segunda optimización a realizar para mejorar el rendimiento es tener en cuenta en tiempo de compilación si una subexpresión es determinista o indeterminista. La propiedad de ser determinista es una propiedad semántica y por tanto típicamente indecidible, pero podrían usarse aproximaciones sintácticas simples basadas en el hecho de que la fuente sintáctica del indeterminismo es el operador `(?)`. En el árbol de sintaxis abstracto se puede anotar esta propiedad, de manera que mientras un cómputo sea determinista no se llevará a cabo ninguna sobrecarga, usando funciones Haskell sin decorar, es decir, sin recurrir a la programación monádica. Si alguna pieza es indeterminista, hay que traducir según las reglas explicadas anteriormente. Esto permitiría que las piezas deterministas se ejecuten a la máxima velocidad y las piezas no deterministas casi inevitablemente más lentas.

Estas tareas no se han llevado a cabo por la necesidad de acotar el desarrollo del lenguaje, ya que no se persigue la eficiencia sino mostrar una posible solución simple a la combinación de dos tipos de semántica y mostrar las ventajas que esto puede llegar a ofrecer.

10. Conclusiones y trabajo futuro

Hemos presentado e implementado un pequeño lenguaje funcional con características indeterministas y con combinación de las semánticas 'call-time choice' y 'run-time choice', que era el núcleo del proyecto. La implementación de todas las fases se ha hecho en Haskell, un lenguaje puramente funcional de referencia en este paradigma. La parte esencial de la implementación viene dada por un proceso de traducción a Haskell de un programa Sharade; es decir, lo que podemos considerar como un proceso de compilación a Haskell de Sharade. La traducción hace un uso intensivo de programación monádica para representar el indeterminismo y la compartición de expresiones, usando la primitiva `share` de la biblioteca de Fischer. También creemos que la traducción es sencilla, ya que cada expresión en Sharade se corresponde con una expresión en Haskell, siguiendo un modelo recursivo muy básico y fácil de entender.

Una vez más queda demostrado el enorme potencial y abstracción de Haskell para describir de una manera clara y concisa cómputos muy complicados y a priori nada triviales. Gracias a su sistema de tipos fuerte asegura gran seguridad acerca del resultado. Incluso se puede razonar acerca del comportamiento del programa tan solo mirando los tipos de las funciones que están involucradas. Esto supone una gran ventaja, pero es necesario desarrollar una cierta soltura para poder realizar ese tipo de deducciones. Al principio el programador se puede encontrar con errores crípticos y por lo tanto tener la sensación de que Haskell es un lenguaje complicado y que no ayuda al programador, pero todo lo contrario. Incluso los errores están aportando una gran información si se saben interpretar.

También es necesario citar que cualquier páramo inexplorado de Haskell para un programador puede resultar algo completamente nuevo, extraño y sin ningún concepto sólido sobre el que basarse para entender el nuevo aspecto. Esto me hace considerar que, siendo uno de mis lenguajes favoritos, es el más complicado con diferencia, incluso más que las nuevas revisiones de C++. ¡Haskell es un lenguaje tan bien definido que hasta los tipos tienen tipos!

Por último, creemos que la combinación de semánticas en Sharade ha quedado con una sintaxis clara y consistente, pudiendo expresar algoritmos indeterministas con una riqueza semántica bastante importante, por lo que creemos que esta combinación o modelos similares puedan ser la base de futuros trabajos sobre programación con funciones no deterministas.

Aunque el trabajo aquí desarrollado satisface los objetivos propuestos, dista mucho de agotar todas las posibilidades del tema abordado. De hecho, hay mucho espacio para nuevos desarrollos. Tenemos claro que algunos aspectos admiten mejoras, de las cuales daremos pistas sobre cómo se pueden llevar a cabo e incorporar al trabajo existente, como examinamos a continuación.

- En cuanto a la sintaxis, se puede añadir mucho azúcar sintáctico para las

listas, listas intensionales, múltiples definiciones ecuacionales para sustituir las expresiones `case`, etc.

- Podría ser interesante dotar al lenguaje un sistema de módulos como en Haskell, añadir funciones predefinidas y aumentar las características de las listas.

- Para mejorar el rendimiento se pueden poner en práctica las dos propuestas realizadas, que como se ha mostrado en un ejemplo, puede mejorar el tiempo de ejecución en un 50 %. La primera propuesta está parcialmente desarrollada en la rama `optv1` del repositorio.

- También se puede realizar la derivación automática de clases, con alguna de las herramientas propuestas o actualizando la implementación de Fischer. Esto luego se tendría que incorporar al proceso de traducción. Una vez hecho esto, podrían incorporarse clases de tipos a Sharade, pudiendo unificar los operadores de suma, resta... de los tipos `Integer` y `Double`, modificando así el sistema de inferencia de tipos[18].

- Se puede hacer más evidente la incorporación de nuevas formas de recolectar los resultados. Actualmente está escondido y requiere un conocimiento básico del funcionamiento de la biblioteca de Fischer para implementarlos.

- Es necesario añadir más funcionalidades al compilador. Actualmente no permite nombrar el fichero de salida, no tiene opciones para crearlo como un módulo con nombre propio y podría ser interesante incorporar a la traducción funciones auxiliares para no tener que tratar directamente con el código traducido.

10.1. Rendimiento

El rendimiento de Sharade es mejorable. El ejemplo de 'permutation sort' de Fischer[8] tarda 16 segundos en ordenar una lista de 20 elementos, un rendimiento muy parecido al que se obtiene en Curry con MCC[19]. Sin embargo Sharade tarda en ordenar la misma lista 43 segundos.

Esto es debido al intensivo uso de la programación monádica. En la implementación de Fischer, las funciones que implementa son deterministas, en Sharade todas deben ser no deterministas. Esto produce una sobrecarga bastante importante de programación monádica, lo que lleva a una pérdida del rendimiento de cerca del 50 %. Fischer en su biblioteca también hace un uso intensivo de la programación monádica, pero es capaz de llevar a cabo muchas optimizaciones en el código para reducir este uso, tarea que no he realizado en mi implementación de la traducción.

Aplicando a mano la primera optimización propuesta en el apartado de 'Traducción a Haskell', el ejemplo de 'permutationsort' observa una mejora de rendimiento muy notable, más del 50 %, tardando 18 segundos. La segunda optimización no se puede aplicar en este ejemplo ya que es intrínsecamente indeterminista.

```
import Sharade.Prelude

insert :: Sharing s => s a -> s (List s a) -> s (List s a)
insert e ls = mPlus (cons e ls) (ls >>= (\pec -> case pec of
  Cons x xs -> cons x (insert e xs)
  _ -> mzero))

perm :: Sharing s => s (List s a) -> s (List s a)
perm ls = ls >>= (\pec -> case pec of
  Nil -> nil
  Cons x xs -> insert x (perm xs))

isSorted :: Sharing s => s (List s Integer) -> s Bool
isSorted ls = ls >>= (\pec -> case pec of
  Nil -> true
  Cons x xs -> xs >>= (\pec -> case pec of
    Nil -> true
    Cons y ys -> mAnd (mLeq x y) (isSorted cons y ys)))

sort :: Sharing s => s (List s Integer) -> s (List s Integer)
sort ls = share (perm ls) >>= (\sls -> (isSorted sls) >>= (\pec ->
  case pec of
    True -> sls
    _ -> mzero))
```

Como se explica en la traducción, cada aplicación funcional se debe hacer con el operador infijo `<#>`, ya que la función esta escondida en una mónada y también el valor. Quitando toda esa sobrecarga y devolviendo a las funciones su aspecto habitual, queda un código bastante parecido al de Fischer. Como ya he dicho antes, tarda en ordenar una lista de 20 elementos 18 segundos, una marca muy cercana a la que consigue Fischer en su implementación. Este resultado me resulta bastante satisfactorio, ya que es una optimización relativamente fácil de llevar a cabo en un futuro. De hecho, hay una implementación parcial de esta idea en la rama `optv1` del repositorio de GitHub. Es parcial porque no realiza compartición de valores funcionales. Salvando eso, el resto de características del lenguaje funcionan correctamente, incluido el orden superior y, por supuesto, la compartición de tipos primitivos.

10.2. Recolección de resultados

La forma de recolectar los resultados parece ser bastante importante en el mundo de la programación indeterminista. Por ejemplo, el compilador KiCS2 de Curry ofrece varias formas de hacer la búsqueda en el espacio de resultados[20].

Mi lenguaje proporciona dos formas de recolectar los resultados, explicadas anteriormente, que son las implementaciones de Fischer. Pero también se da al usuario la posibilidad de implementar sus propias fórmulas de búsqueda, dando la posibilidad de descartar resultados para aumentar el rendimiento, como una búsqueda de ramificación y poda. También se pueden implementar búsquedas más simples. como una búsqueda en anchura, en profundidad iterativa o una búsqueda paralela multihilo. Por último quería destacar que no hay ningún motivo técnico en la implementación que pueda hacer que una búsqueda que no cumpla con las leyes de Fischer provoque un mal funcionamiento de la biblioteca y, por extensión, en mi implementación.

Fischer tiene un trabajo[21] sobre cómo modificar el sistema de backtracking en programación monádica en Haskell. En concreto, muestra cómo implementar la búsqueda en anchura y la búsqueda en profundidad por niveles, además de reinventar la búsqueda en profundidad que hace Haskell. Estas ideas se pueden usar para implementar tu propia mónada que realice la búsqueda deseada e incorporarla a Sharade.

10.3. Tipos de usuario

Sharade no soporta la definición de tipos arbitrarios. Esto es debido a que esos tipos serían indeterministas, por lo que tienen que estar sujetos a las reglas del Sharing de Fischer y pertenecer a una serie de clases de tipos. Fischer tiene implementado una derivación automática para tipos arbitrarios a sus clases de tipos, pero no funciona con las versiones actuales de Haskell. Actualmente se puede conseguir con múltiples herramientas como `Template-Haskell`[22] o con `DrIFT`[23] o con `derive`[24]. Esta última es la que usa Fischer para implementar la derivación automática de las clases de tipos. Abordar esta cuestión estaba fuera de los objetivos, debido a la necesidad de acotar el trabajo tanto en tiempo como en características. Por lo tanto, Sharade no soporta tipos de usuario, dejándolo como un trabajo futuro.

11. Conclusions and future work

We have introduced and implemented a small functional language with indeterministic features and a combination of 'call-time choice' and 'run-time choice' semantics, which was the core of the project. The implementation of all phases has been done in Haskell, a purely functional language of reference in this paradigm. The essential part of the implementation is given by a process of translation into Haskell of a Sharade program; that is, what we can consider as a process of compilation into Haskell of Sharade. Translation makes intensive use of monadic programming to represent indeterminism and the sharing of expressions, using the primitive verb'share' of the Fischer library. We also believe that translation is simple, as each expression in Sharade corresponds to an expression in Haskell, following a very basic recursive model and easy to understand.

Once again Haskell demonstrates his enormous potential and abstraction to describe in a clear and concise way very complicated and not trivial computations. Thanks to his strong type system it ensures great security about the result. You can even reason about the behavior of the program just by looking at the types of functions that are involved. This is a great advantage, but it is necessary to develop a certain fluency in order to be able to make such deductions. At first the programmer may encounter cryptic errors and therefore have the feeling that Haskell is a complicated language and does not help the programmer, but quite the opposite. Even errors are providing great information if you know how to interpret.

It also needs to be mentioned that any unexplored Haskell feature for a programmer can be something completely new, strange and without any solid concept on which to build to understand the new aspect. This makes me consider that, being one of my favorite languages, it is by far the most complicated language, even more so than the new C++ revisions. Haskell is such a well-defined language that even types have types!

Finally, we believe that the combination of semantics in Sharade has remained with a clear and consistent syntax, being able to express indeterministic algorithms with a quite important semantic richness, reason why we believe that this combination or similar models can be the base of future works about programming with nondeterministic functions.

Although the work developed here satisfies the proposed objectives, it is far from exhausting all the possibilities of the topic addressed. In fact, there is plenty of room for new developments. It is clear to us that there is room for improvement in some areas, of which we will provide clues as to how this can be done and incorporated into existing work, as discussed below.

- As for syntax, a lot of syntax sugar can be added for lists, list comprehension, multiple equational definitions to replace the expressions `case`, etc.

- It might be interesting to provide the language with a system of modules such as Haskell, to add predefined functions and increase the characteristics of the lists.

- In order to improve performance, the two proposals made can be put into practice, which, as shown in an example, can improve execution time by 50 %. The first proposal is partially developed in the branch `optv1` of the repository.

- It is also possible to automatically derive classes with one of the proposed tools or by updating the Fischer implementation. This would then have to be incorporated into the translation process. Once this is done, type classes could be incorporated to Sharade, being able to unify the operators of addition, subtraction... of the types `Integer` and `Double`.

- It can be made more evident the incorporation of new ways of collecting the results. It is currently hidden and requires a basic knowledge of how the Fischer library works to implement them.

- More functionality needs to be added to the compiler. Currently it does not allow naming the output file, it does not have options to create it as a module with its own name and it might be interesting to incorporate auxiliary functions into the translation so as not to have to deal directly with the translated code.

11.1. Performance

The performance of Sharade can be improved. The 'permutation sort' Fischer's example takes 16 seconds to sort a list of 20 elements, very closely to Curry's MCC compiler [19]. Sharade takes 43 seconds to sort the same list.

This is due to the intensive use of monadic programming. In the Fischer's implementation, the functions are deterministic, in Sharade all must be nondeterministic. This produces an important overhead of monadic programming that leads to a loss of performance of about 50 %. Fischer does in his library a lot of monadic programming, but he is able to do a lot of inline optimizations to reduce this use. This kind of task could be done in my implementation, but as previously explained, the objetive of Sharade is not to persue performance.

Applying by hand the first optimization proposed in the section of 'Translation to Haskell', this example observes a performance gain of about 50 %. It takes 18 seconds to sort a list of 20 elements. The second optimization proposed is not applicable in this example because is indtrinsically nondeterministic.

```
import Sharade.Prelude

insert :: Sharing s => s a -> s (List s a) -> s (List s a)
insert e ls = mPlus (cons e ls) (ls >>= (\pec -> case pec of
  Cons x xs -> cons x (insert e xs)
  _ -> mzero))

perm :: Sharing s => s (List s a) -> s (List s a)
perm ls = ls >>= (\pec -> case pec of
  Nil -> nil
  Cons x xs -> insert x (perm xs))

isSorted :: Sharing s => s (List s Integer) -> s Bool
isSorted ls = ls >>= (\pec -> case pec of
  Nil -> true
  Cons x xs -> xs >>= (\pec -> case pec of
    Nil -> true
    Cons y ys -> mAnd (mLeq x y) (isSorted cons y ys)))

sort :: Sharing s => s (List s Integer) -> s (List s Integer)
sort ls = share (perm ls) >>= (\sls -> (isSorted sls) >>= (\pec ->
  case pec of
    True -> sls
    _ -> mzero))
```

Previously explained, each funcional application must be done with the infix operator `<#>`, because the function and the value are inside a monad. If we delete this overhead and returning to the normal aspect of functions in Haskell, we obtain code similar to Fischer's implementation. As I said before, it takes to sort a list of 20 elements nearly 18 seconds, very close to Fischer's implementation. This result is very satisfying since it is a relatively easy optimization to carry out in the future. In fact, there is a partial implementation in `optv1` branch on the GitHub repository. It's partial because it lacks the functionality to share functions, since it requires a deep modification of the translation function. The remaining features such as high order, lazyness, sharing of primitive types and indeterminism are working as expected.

11.2. Observing results

Sharade provides two forms of observing results, explained before. Sharade gives the user the posibility of implementing its own ways os observing the results.

These implementations can be very complex like a 'Branch and bound' algorithm or simpler like a 'breadth first search' or a 'Iterative deepening depth-first search' or even a multithread search.

Fischer has a very interesting research [21] on how to modify the backtacking system in monadic programming in Haskell. Specifically, he shows how to implement a 'breadth first search' or a 'Iterative deepening depth-first search' in Haskell. He also reinvents the 'depth-first search' of Haskell. These ideas can be used to implement your own monad that perform the desired search and incorporate it into Sharade.

The way of collecting the results seems to be quite important in the world of indeterministic programming. For example, Curry's KiCS2 compiler offers several ways to explore the search space [20].

11.3. User defined types

Sharade does not support the definition of arbitrary types. This is because those types would be indeterministic, so they have to be subject to Fischer's Sharing rules and belong to a series of type classes. Fischer has implemented an automatic derivation for arbitrary types to its type classes, but it does not work with the current versions of Haskell. Currently, it can be done with multiple tools such as `Template-Haskell` [22] or with `DrIFT` [23] or with `derive` [24]. The latter is the one used by Fischer to implement automatic derivation of type classes. Addressing this issue was outside the objectives, due to the need to limit the work in both time and features. Therefore, Sharade does not support user types, leaving it as a future job.

Referencias

[1] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[2] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.

[3] FJ López Fraguas and J Sánchez Hernández. Toy: A multiparadigm declarative system. In *International Conference on Rewriting Techniques and Applications*, pages 244–247. Springer, 1999.

[4] Heinrich Hussmann. *Nondeterminism in algebraic specifications and algebraic programs*. Birkhäuser, 1993.

[5] Juan Carlos González Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.

[6] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: the standard: reference manual*. Springer Science & Business Media, 2012.

[7] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Kics2: A new compiler from curry to haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[8] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy nondeterministic programming. *Journal of Functional programming*, 21(4-5):413–465, 2011.

[9] Adrián Riesco and Juan Rodríguez-Hortalá. Singular and plural functions for functional logic programming. *Theory and Practice of Logic Programming*, 14(1):65–116, 2014.

[10] Francisco J Lopez-Fraguas, Juan Rodriguez-Hortala, and Jaime Sanchez-Hernandez. A flexible framework for programming with non-deterministic functions. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 91–100. ACM, 2009.

[11] Francisco J López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 197–208. ACM, 2007.

[12] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[13] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[14] Simon Marlow et al. Haskell 2010 language report. *Available online at* *https://www.haskell.org/onlinereport/haskell2010/*, 2010.

[15] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[16] Stephen Diehl. Write you a haskell: Building a modern functional compiler from first principles. `http://dev.stephendiehl.com/fun/WYAH.pdf`.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[18] Geoffrey S Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[19] Wolfgang Lux. The münster curry compiler, 2003.

[20] Michael Hanus, Björn Peemöller, and Fabian Reck. Search strategies for functional logic programming. *Software Engineering 2012. Workshopband*, 2012.

[21] Sebastian Fischer et al. Reinventing haskell backtracking. *GI Jahrestagung*, 2009:2875–2888, 2009.

[22] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[23] John Meacham Noel Winstanley. Drift: Program to derive type class instances. Available at `http://repetae.net/computer/haskell/DrIFT/`.

[24] Neil Mitchell. derive: A program and library to derive instances for data types. Available at `https://github.com/ndmitchell/derive`.