Parsers funcionales genéricos

Pablo E. Martínez López Gustavo A. Nestares

LIFIA, Departamento de Informática, Universidad Nacional de La Plata.
C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.
E-mail: {fidel,gusnes}@info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar/

Resumen

En este trabajo se propone el uso de técnicas avanzadas de programación funcional para ofrecer una solución genérica al problema de *parsing*. Este problema consiste en que, dada una lista de *tokens* que representan una estructura, debe obtenerse una representación elaborada de la misma.

La solución propuesta es genérica en dos sentidos. Primero, la técnica de diseño utilizada consiste en dar una biblioteca de *combinadores*, los cuales pueden combinarse para escribir soluciones a problemas complejos (instanciando la técnica de programación modular). Segundo, el uso de mónadas y *overloading* permite independizar la biblioteca de una implementación específica. La solución propuesta incluye como instancias a varias propuestas distintas de combinadores de parsing, siendo éstas implementaciones en el modelo aquí presentado.

Palabras clave: Programación funcional, Parsing, Mónadas, *Overloading*

1 Introducción

La programación funcional ha crecido, en los últimos diez años, más que en todo el resto de su historia. El desarrollo de temas avanzados tales como teoría de mónadas [Wad92, JW93, Wad95, NM96], sistemas de clases para proveer *overloading* [Jon95a, Jon95b, PH+96], entrada/salida puramente funcional [HS88, HJW+92, PH+96, GM96], etc., hacen que la Programación Funcional comience a ser de utilidad, no sólo en la prototipación, sino también en la implementación de sistemas de envergadura en el comercio y la industria [RW95].

Además, las características clásicas de los lenguajes funcionales (funciones de alto orden y currificación, sistemas fuerte de tipos con inferencia automática, transparencia referencial, evaluación *lazy*, etc.) hacen de ellos una herramienta poderosa a la hora de aplicar nociones tales como reusabilidad de software, programación modular, prototipación rápida, etc. [BW88]. En particular, en este trabajo, utilizaremos una técnica de diseño que consiste en ofrecer una biblioteca de funciones específicas (llamadas *combinadores*), que permiten dar solución a todos los problemas de un área dada; la utilización de funciones de alto orden para la implementación de los combinadores da gran flexibilidad y facilidad de uso al software resultante (pues expresa correctamente la noción de programación modular).

El problema a solucionar es conocido con el nombre de *parsing*, y su solución es componente fundamental de un compilador. Este problema consiste en obtener una representación elaborada de una estructura a partir de una descripción elemental de la misma, generalmente dada en formato de texto. Una generalización posible es que, en lugar de considerar al texto como una lista de caracteres, se la considerará como una lista de objetos más complejos (llamados *tokens*), de los cuales los caracteres serán sólo una instancia. Otra generalización consiste en no precisar el tipo de datos con el que se expresará la representación elaborada de la estructura, utilizando polimorfismo paramétrico. Existen varias soluciones con estas características [Hut93, Fok95, Wad95, Röj95, GN95, HM96], pero cada una de ellas provee sólo una implementación específica del conjunto de combinadores, limitando de esa manera la generalidad de la solución.

El sistema de clases constructoras de Mark Jones [Jon95b] utiliza predicados (llamados *clases*) para representar conjuntos de constructores de tipo; la propuesta de este trabajo es utilizar dicho sistema de clases para describir al conjunto de constructores de tipo que sirven para representar combinadores de parsing, permitiendo de esa manera que distintas implementaciones de los combinadores de parsing sean consideradas bajo una visión unificadora. Este trabajo se encuentra aún en desarrollo, por lo que se presenta sólo una versión preliminar del mismo; de todas maneras, el trabajo ya realizado es promisorio y permite suponer que lo que resta puede

ser llevado a cabo con éxito.

El artículo se organiza de la siguiente manera. En la segunda sección se presenta con más detalle el problema de parsing y la solución por combinadores. En la sección tres se presenta la idea de mónadas, y se muestra que los parsers son mónadas. En la cuarta sección se presenta rápidamente la noción de clase constructora y se muestra cómo utilizar esta noción para dar una versión genérica de las mónadas. En la última sección se muestra la definición genérica propuesta para los combinadores de parsing. El trabajo finaliza con conclusiones y perspectivas.

Para la presentación de los ejemplos se utilizará la sintaxis del lenguaje Haskell [PH+96].

2 Combinadores de Parsing

El problema de parsing consiste en obtener una representación elaborada de una estructura a partir de una descripción elemental de la misma generalmente dada en formato de texto.

Un ejemplo clásico de esto es el *front end* de un compilador de un lenguaje de programación: dado el código fuente, se obtiene el código intermedio correspondiente. Habitualmente, el compilador procede primero con una fase de análisis lexicográfico, transformando el código fuente (lista de caracteres) en una lista de palabras reservadas, literales numéricos, etc. (llamados *tokens*), sobre los que luego se realiza el análisis sintáctico; por ello, es habitual considerar una generalización de este problema, que consiste en cambiar la descripción elemental, que en lugar de ser un texto (lista de caracteres), es una lista de *tokens* (cuyo tipo se abstrae, utilizando polimorfismo paramétrico). De la misma manera, la representación elaborada puede tener variadas formas, dependiendo del problema específico: en el análisis lexicográfico la forma elaborada es una lista de tokens, en el análisis sintáctico es un árbol de parsing (que varía de lenguaje en lenguaje); por ello también se considera la abstracción del tipo de dicha representación elaborada de la misma forma en que se abstrajo el tipo de los tokens.

Por otro lado, se pretende que el parsing sea realizado de una manera incremental; o sea que para realizar el parsing de dos objetos, se realizará primero el de uno y luego, sobre los tokens restantes, el otro. Para ello se requiere que un parser dado retorne no sólo la representación elaborada pedida, sino también la lista de tokens que no fueron utilizados.

Se intenta, entonces, la definición de un tipo que represente a los parsers: `type Parser token struct = [token] -> (struct, [token])`, donde `token` y `struct` son variables de tipo. Este tipo captura lo dicho en los párrafos anteriores.

Ahora bien, si se quiere representar algún otro efecto (que el parsing falle, que sea no-determinístico, que reporte la posición del primer error, etc.), se requiere modificar este tipo

para agregar la información correspondiente al efecto en cuestión. Por ejemplo, en [Fok95] el tipo de los parsers está definido como `[token] -> [(struct, [token])]`, permitiendo que los parsers fallen y sean no-determinísticos, en [Röj95] está definido como `[token] -> Pos -> Return (struct, [token])`, permitiendo fallar (expresado por el constructor de tipos `Return`) y reportar la posición donde se produjo el primer error (expresado por los tipos `Pos` y `Return`), etc. Cualquier tipo que permita representar lo expuesto hasta aquí sirve para representar un parser; es esta observación la que guía este trabajo.

El siguiente paso es ofrecer alguna manera de definir parsers. Para ello se utilizará la técnica de programación modular, que consiste en brindar un conjunto de soluciones elementales y un conjunto de formas de combinar soluciones, pudiendo expresarse cualquier solución como combinación de soluciones elementales. En este caso, los parsers elementales son aquellos que permiten terminar con éxito y fracaso sin consumir ningún token, y los que permiten consumir un solo token de manera incondicional y condicional:

```
succeed :: a -> Parser token a
fail    :: Parser token a
item    :: Parser token token
satisfy :: (token -> Bool) -> Parser token token
```

Por ejemplo, `succeed [ ]` es el parser que retorna la lista vacía sin consumir input, `fail` es el parser que siempre falla y sin retornar nada, `satisfy (=='a')` consume el primer token si existe y es una letra `a`, y si no falla, `item` consume el primer token, si existe, y lo retorna.

Los combinadores primitivos permiten expresar secuenciación, alternación y repetición de parsers y transformación de la estructura obtenida por un parser:

```
seq   :: Parser token a -> Parser token b -> Parser token (a, b)
alt   :: Parser token a -> Parser token a -> Parser token a
many  :: Parser token a -> Parser token [a]
using :: Parser token a -> (a -> b) -> Parser token b
```

Por ejemplo, el parser `satisfy (=='i') 'seq' satisfy (=='f')` consume los dos primeros caracteres, si forman la palabra "if", el parser `satisfy (=='i') 'alt' satisfy (=='I')` consume una letra `i`, en mayúscula o minúscula, el parser `many (satisfy isDigit)` consume cero o más dígitos, y el parser `satisfy isDigit 'using' dig2num` consume el primer token si es un dígito y retorna el número correspondiente (la función `dig2num` transforma un caracter numérico en un número).

Utilizando los parser elementales y los combinadores primitivos pueden definirse otros parsers y otros combinadores; como ejemplo, se definen un combinador `parens :: Parser token a -> Parser token a` que reconoce lo mismo que su argumento, pero encerrado entre paréntesis, y un combinador `ap:: Parser token (a -> b) -> Parser token a -> Parser token b`, que aplica la función obtenida por el primer parser al elemento obtenido por el segundo:

```
parens p = (satisfy (=='(') `seq` p) `seq` satisfy (==')')
> ap pf pa = (pf `seq` pa) `using` (\(f,a) -> f a)
```

Adicionalmente, debe existir una manera de "ejecutar" un parser, o sea que dado el parser y dada la lista de tokens, produzca el resultado esperado (quizás vaciando completamente la lista de tokens). Para ello se provee un combinador primitivo `parse :: Parser token a -> [token] -> [a]`. Si el parsing falla, `parse` retorna una lista vacía; si es determinístico, retorna una lista unitaria; y si es no-determinístico, retorna una lista con todos los valores obtenidos.

Las defincciones de los parsers elementales y de los combinadores primitivos dependen del tipo de datos elegido para representar al tipo `Parser`.

3 Mónadas

Las mónadas fueron popularizadas por Wadler [Wad92, Wad95] como un método de estructurar programas funcionales para expresar efectos imperativos, pero manteniendo la legibilidad y modificabilidad tradicionales del estilo funcional.

Un mónada es un tipo (posiblemente abstracto) `M a`, que representa al tipo `a` enriquecido con información que permite expresar la ocurrencia de efectos adicionales. Se definen dos operaciones sobre el constructor de tipos `M`: `return :: a -> M a` y `bind :: M a -> (a -> M b) -> M b`. Las propiedades que estas operaciones deben satisfacer para que `M` sea una mónada establecen que `return` representa al efecto nulo, y que `bind` representa a la secuenciación de efectos [Wad95].

Por ejemplo, para representar el efecto de posibilidad de error, se define el siguiente tipo:

```
data Error a = Fail String | Return a
returnE a = Return a
bindE m f = case m of {Fail s -> Fail s; Return a -> f a}
```

El constructor `Fail` expresa un error (identificado mediante el string), y el constructor `Return` expresa que no se produjo ningún error, y se pudo construir un elemento de tipo `a`. La operación `returnE` captura el efecto nulo (no producir error), y la operació `bindE` captura la secuencia de efectos (propagación de error).

Un ejemplo clásico de mónadas son las listas (el efecto es retornar más de un valor):

```
returnL a = [a]
bindL m f = case m of {[ ] -> [ ]; (x:xs) -> f x ++ bindL xs f}
```

Al considerar de manera abstracta a las mónadas, las diferencias entre ellas quedan establecidas por las operaciones adicionales sobre ellas. Por ejemplo, para establecer que una mónada captura el efecto de producir un posible error, se pide que la misma tenga una operación `zero :: M a`, que cumpla con propiedades de absorción respecto de `bind`; una mónada con esta operación es conocida con el nombre de *mónada con cero*. De la misma manera, para establecer que una mónada con cero puede recuperarse del error producido por la operación `zero`, se pide que posea una operación `plus :: M a -> M a -> M a`, que sea asociativa y que tenga a `zero` como neutro; estas mónadas se conocen como *mónadas con adición*. Tanto la mónada `Error` como la mónada de listas son mónadas con cero y adición.

Es interesante observar que el tipo `Parser` con las operaciones definidas hasta ahora no conforma una mónada, pues ninguna de ellas puede hacer el papel de `bind`; pero si pedimos que `Parser` tenga una operación `bind`, tal que junto con la operación `succeed` formen una mónada (siguiendo la notación de [Röj95] esta operación será llamada `into`), entonces las operaciones `seq` y `using` pueden definirse de la siguiente manera:

```
> p `seq` q   = p `into` \a -> q `into` \b -> succeed (a,b)
> p `using` f = p `into` \a -> succeed (f a)
```

Además, la operación `fail` funciona como cero, y la operación `alt` funciona como adición, y con ellas puede definirse la operación `using`:

```
> many p = (p `into` \a -> many p `into` \as -> succeed (a:as)) `alt`
>          succeed [ ]
```

El tipo `Parser` puede ser visto como una mónada con cero y adición, con operaciones adicionales `parse`, `item` y `satisfy`, pudiendo definirse la última en términos de la segunda y viceversa:

```
item = satisfy (\a -> True)
satisfy p = item `into` \t -> if (p t) then (succeed t) else fail
```

4 Clases Constructoras

Observando los ejemplos de mónadas de la sección anterior, se vé que es necesario un nombre distinto para las operaciones de cada una de ellas (p.ej. `bindE` y `bindL` para `Error` y para listas),

a pesar de que su tipo es prácticamente el mismo (salvo por el constructor de la mónada). Esto es así porque el sistema de tipos Hindley-Milner clásico (que tienen los lenguajes funcionales tradicionales) no provee un mecanismo por el cual dos funciones distintas puedan tener el mismo nombre (característica conocida habitualmente con el nombre de *overloading*, o polimorfismo *ad-hoc*). Para ello se puede extender el sistema de tipos con un mecanismo de clasificación de tipos que comparten funciones distintas con el mismo nombre; esta propuesta se conoce con el nombre de sistema de clases, y fue realizada por Philip Wadler [WB89]. Mark Jones ([Jon95a, Jon95b]) propuso una extensión de este sistema para operar con constructores de tipos, con la cual es posible expresar el *overloading* que ocurre en ejemplos como las mónadas.

Una clase es un conjunto de tipos (o constructores de tipos, en el sistema extendido) que comparten el nombre de una o más funciones; cada uno de los tipos que pertenece a la clase debe proveer una implementación específica de las funciones compartidas. La declaración de una clase establece su nombre y los nombres y tipos de las funciones que se compartirán (llamadas *funciones miembro*), así como implementaciones por defecto para algunas de estas operaciones. Por ejemplo, para sobrecargar el nombre `(==)` con la operación de igualdad entre dos elementos de un mismo tipo se define la siguiente clase:

```
class Eq a where (==), (/=) :: a -> a -> Bool
                    a /= b = not (a == b)
```

que establece que un tipo, para pertencer a la clase `Eq` debe proveer implementación para las funciones de igual y de no-igual, donde esta última tiene como implementación por defecto a la negación de la igualdad.

Los tipos que pertenecen a una clase se conocen con el nombre de instancias. Para ser instancia de una clase, un tipo debe proveer implementaciones para las funciones de la clase. La única manera de que un tipo sea instancia de una clase es a través de una declaración de instancia. Por ejemplo, para declarar al tipo de los números enteros como instancia de la clase `Eq`, se define la siguiente instancia:

```
instance Eq Int where (==) = primEqInt
```

donde la operación `primEqInt` es una primitiva del procesador que compara dos enteros por igualdad.

Las clases sirven para agrupar aquellos tipos que comparten el nombre de operaciones; las funciones que utilicen una de tales operaciones deben garantizar que sólo la aplicarán a valores del tipo correcto. Para expresar esto en el sistema de tipos, se define la noción de contexto: un contexto es un conjunto de predicados que establecen que ciertos tipos son instancia de

ciertas clases; cada tipo es extendido con un contexto que restringe sus variables para que sean instancias de ciertas clases. Por ejemplo, una función que toma una lista y un elemento y retorna un valor de verdad que indica si dicho elemento está o no en la lista puede implementarse como:

```
elem :: Eq a => [a] -> a -> Bool
elem [ ] y = False
elem (x:xs) y = if (x==y) then True else (elem xs y)
```

donde `Eq a` es el contexto del tipo de `elem`, que establece que el tipo `a` debe ser instancia de la clase `Eq` para poder aplicar esta función.

Los contextos pueden usarse en las declaraciones de instancias. Por ejemplo, para declarar a los pares como instancia de `Eq`, ambas componentes deben ser también instancias de `Eq`,

```
instance (Eq a, Eq b) => Eq (a,b) where
   (x1,y1) == (x2,y2) = (x1 == x2) `and` (y1 == y2)
```

declarando la igualdad de pares como la igualdad componente a componente.

Además de usarse en tipos y en instancias, los contextos pueden utilizarse en la declaración de nuevas clases, expresando en este caso que para ser instancia de la nueva clase, un tipo debe ser primero instancia de lo especificado en el contexto. Por ejemplo, si queremos tener una clase que establezca que un tipo tiene operadores relacionales de orden, podemos pedir que los elementos de ese tipo puedan compararse por igualdad agregando `Eq a` como contexto en la declaración de la clase `Ord`.

```
class Eq a => Ord a where (<=), (<), (>=), (>) :: a -> a -> Bool
```

En este caso se dice que `Ord` es *subclase* de `Eq`.

El sistema de clases descrito hasta aquí (llamado de primer orden), no permite que estructuras con parámetros compartan nombres de función. Por ejemplo, la función `map` transforma cada elemento de una lista, mientras que la función `mapE` transforma el valor contenido en un elemento del tipo `Error` (ver Secc. 3):

```
map :: (a -> b) -> [a] -> [b]
map f ys = case ys of {[ ] -> [ ]; (x:xs) -> f x : map f xs}

mapE :: (a -> b) -> Error a -> Error b
mapE f err = case err of {Fail s -> Fail s; Return x -> Return (f x)}
```

Como el constructor de listas y `Error` tienen un parámetro, no pueden, en el sistema de primer orden, compartir el nombre de la función (que en escencia hace lo mismo, teniendo en cuenta la estructura). Para ello se propone un sistema de clases de alto orden, que clasifique no sólo a los tipos, sino también a los constructores de tipos (estructuras). Entonces, para que los constructores lista y `Error` compartan el nombre `map`, se declara una clase y dos instancias:

```
class Functor m where map :: (a -> b) -> m a -> m b
instance Functor [ ] where map f xs = ...
instance Functor Error where map f err = ...
```

Con este sistema de alto orden, es sencillo expresar que todas las mónadas comparten el nombre de sus funciones, y que algunas son mónadas con cero o adición:

```
class Monad m where return :: a -> m a
                    (>>=)  :: m a -> (a -> m b) -> m b
class Monad m => MonadZero m where zero :: m a
class MonadZero m => MonadPlus m where (++) :: m a -> m a -> m a
```

Estas declaraciones sólo sirven para indicar que distintos tipos, o constructores de tipos, comparten el nombre de una o más funciones. A pesar de ello, una declaración de clase habitualmente lleva una intención implícita, que se puede expresar mediante un conjunto de ecuaciones; estas ecuaciones no pueden ser verificadas automáticamente, por lo que se deja como responsabilidad del programador demostrarlas. Por ejemplo, la declaración de la clase `Eq` lleva implícito que `(==)` representa a una relación reflexiva, simétrica y transitiva, la de `Ord` que `(<=)` representa a una relación reflexiva, antisimétrica y transitiva, etc.; ninguna de estas propiedades puede ser verificada por el sistema, por lo que es perfectamente posible declarar una función `(==)` que no sea una igualdad.

Existen otras propiedades que sí pueden ser expresadas en el sistema. Por ejemplo, la propiedad de que toda mónada determina a un functor puede expresarse como:

```
instance Monad m => Functor m where map f m = m >>= \x -> return (f x)
```

De todas maneras, las leyes que un functor debe satisfacer deben probarse a mano, a partir de las leyes de mónadas y la definición de `map`.

Adicionalmente, el lenguaje Gofer [Jon95a] permite utilizar más de un argumento en la declaración de una clase, aumentando el poder expresivo, pero introduciendo ciertos problemas técnicos que no se discuten aquí por problemas de espacio.

El sistema de clases permite aplicar con mayor facilidad la técnica de programación modular. Para ello se define una clase que exprese los nombres de las operaciones de un tipo dado (por ejemplo, `Monad`), y luego se programa la aplicación utilizando dichas operaciones (en [Jon95b] se muestra un ejemplo de esto, implementando un algoritmo simple de inferencia de tipos); finalmente, para que la aplicación esté completa debe proveerse al menos una instancia de la clase, completando así la implementación de la aplicación.

5 Parsers Genéricos

Al final de la Secc. 3 se establece que los parsers son mónadas con cero y adición que además tienen operaciones `parse`, `item` y `satisfy`. Utilizando el sistema de clases presentado en la Secc. 4 se puede expresar esto, capturando la noción de parser sin por ello proveer ninguna implementación específica del tipo base ni de los combinadores primitivos, como se hace en todos los trabajos estudiados.

```
> class MonadPlus (p token) => Parser p token where
>     item    :: p token token
>     satisfy :: (token -> Bool) -> p token token
>     parse   :: p token a -> [token] -> [a]
>     item         = satisfy (\a -> True)
>     satisfy prop = item >>= \a -> if (prop a) then (return a) else zero
> succeed :: Parser p => a -> p token a
> succeed = return
> into :: Parser p => p token a -> (a -> p token b) -> p token c
> into = (>>=)
> fail :: Parser p => p token a
> fail = zero
> alt :: Parser p => p token a -> p token a -> p token a
> alt = (++)
```

Esto nos permite trabajar con combinadores de parsing sin establecer *a priori* cuales son los efectos que se desean (errores posicionales, no-determinismo, combinadores más eficientes, etc.).

A modo de ejemplo se ofrece un programa que, dado un string que representa a un número entero con signo, retorna el número correspondiente. La gramática BNF que usaremos para representar a los números enteros es:

```
int ::= uint | sign uint          uint ::= dig | dig uint
dig ::= 0 |    | 9                sign ::= + | -
```

A partir de la gramática, y utilizando los combinadores introducidos, se provee una función de parsing para cada una de las categorías sintácticas; además de reconocer, estas funciones transforman el resultado para colaborar en la construcción del número en cuestión.

```
> intP, uintP :: Parser p Char => p Char Int
> intP = uintP `alt` (signP `ap` uintP)
> uintP = (digP `seq` many digP) `using` \(d, ds) -> digs2num (d:ds)
> digP :: Parser p Char => p Char Char
> digP = satisfy isDigit
> signP :: Parser p Char => p Char (Int -> Int)
> signP = (symbol '+' `using` (const id))     `alt`
>         (symbol '-' `using` (const negate))
```

La función `digs2num :: [Char] -> Int` transforma una lista de dígitos en un número (ej: `digs2num "231" = 231`).

El ejemplo dado es extremadamente sencillo, pero muestra el poder que tienen los combinadores de parsing. El código presentado no es ejecutable a menos que se provea una instancia de la clase `Parser` (ver el final de la Secc. 4). Existen varias implementaciones posibles para los combinadores de parsing; [Fok95] y [Röj95] pueden ser definidas como instancias de `Parser`, y [Hut93] y [HM96] están aún bajo estudio. Se presenta a modo de ejemplo la definición de los combinadores de [Fok95] como instancia de `Parser`; se ha cambiado la declaración del tipo como sinónimo por una de tipo algebraico, pues el sistema de clases no soporta el uso de sinónimos de tipo, y se agregó un combinador para la operación `bind`, pero la esencia se preserva.

```
> data FokParser token a = MkP [token] -> [([token],a)]
> instance Monad (FokParser token) where
>    return a = MkP (\ts -> [(ts,a)])
>    (MkP p) >>= k =
>      MkP (\ts -> [ (ts'', b) | (ts', a) <- p ts,
>                                (ts'', b) <- let MkP q = k a in q ts'])
> instance MonadZero (FokParser token) where zero = MkP (\ts -> [])
> instance MonadPlus (FokParser token) where
>    (MkP p) ++ (MkP q) =  MkP (\ts -> p ts ++ q ts)
> instance Parser FokParser token where
```

```
>    item = MkP (\ts -> case ts of {[ ] -> [ ]; (t:ts') -> [(ts',t)]})
>    parse (MkP p) ts = map snd (p ts)
```

Finalmente, para indicar que debe usarse la instancia `FokParser` al usar el parser `intP`, se utiliza una signatura explícita de tipos: `parse (intP::FokParser Char Int)`

6 Conclusiones

Los lenguajes funcionales puros y perezosos, como Haskell, son especialmente aptos para la aplicación de la técnica de programación modular. Ello se debe, primero, a que al ser puros (no poseer efectos laterales), una pieza de código puede ser reemplazada por otra del mismo valor sin alterar el significado; en segundo lugar, la evaluación perezosa permite que se invoque una función sólo lo necesario para computar el resultado pedido, y no más, permitiendo el diseño de funciones costosas (o infinitas) que sólo serán llamadas lo mínimo necesario; en tercer lugar, el sistema de clases permite que sólo los nombres de las funciones sean declarados *a priori*, postergando la implementación de las mismas hasta el final, y favoreciendo el cambio de la instancia utilizada sin modificar el programa de aplicación.

El problema de parsing es central a la gran mayoría de los programas actuales, ya que todos ellos deben leer los datos en algún formato y traducirlos a sus propias estructuras de datos. En la mayoría de los casos este problema se sobresimplifica o directamente se lo deja de lado, y sólo en algunos casos se utilizan herramientas como Lex o Yacc para intentar solucionarlo. La solución de combinadores, permitida por los lenguajes funcionales a través de las funciones de alto orden, es altamente intuitiva, y de gran flexibilidad, permitiendo que todos los programas realicen correctamente la lectura de sus datos de entrada.

Este trabajo puede continuar de varias maneras. Por un lado, para capturar la noción de que un parser puede tener efectos (como no-determinismo o recuperación de errores, etc.) se puede introducir la noción de transformador de mónadas y utilizarla en la definición genérica de los parsers. Por otro lado, aún no se ha establecido cuales son las propiedades que `parse`, `item` y `satisfy` deben satisfacer – o sea cuál es la estructura algebraica de los parsers. Finalmente, más implementaciones específicas de parsers deben ser expresadas con este modelo, proveyendo al mismo tiempo una demostración cabal de que esta abstracción es correcta y una biblioteca de implementaciones que permita elegir los efectos que se deseen.

Referencias

[BW88] R.Bird and P.Wadler. *Introduction to Functional Programming.* CAR HOARE. Prentice Hall, 1988.

[Fok95] J.Fokker. Functional parsers. In J.Jeuring and E.Meijer, editors, *Advanced Functional Programming, LNCS 925.* Springer, May 1995.

[GM96] R.García and P.E.Martínez López. Un concepto globalizador en la modelización de E/S en lenguajes funcionales. In *25Jornadas Argentinas de Informática e Investigación Operativa (JAIIO)*, 1996.

[GN95] R.García and G.Nestares. Parsers funcionales, combinadores, tranformadores y aplicaciones. In *Anales de las 2das Jornadas Universitarias de Informática.* Univ. de San Juan, Noviembre 1995.

[HJW+92] P.Hudak, S.Peyton Jones, P.Wadler, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.2. Technical report, Yale Univ., March 1992.

[HM96] G.Hutton and E.Meijer. Monadic parser combinators. Technical report, Univ. of Nottingham, 1996.

[HS88] P.Hudak and R.Sundaresh. On the expressiveness of purely functional I/O systems. Technical report YALEU/DCS/RR665, Yale Univ., Dept. of Computer Science, December 1988.

[Hut93] G.Hutton. Higher-order functions for parsing. In *Journal of Functional Programming, Vol. 1.* Univ. of Utretch, Cambridge University Press, January 1993.

[Jon95a] M.Jones. Gofer 2.30 release notes. Technical report, Yale Univ., 1995. http://www.cs.nott.ac.uk:80/Department/Staff/mpj/.

[Jon95b] M.Jones. Functional programming with overloading and higher-order polymorphism. In J.Jeuring and E.Meijer, editors, *Advanced Functional Programming, LNCS 925.* Springer, May 1995.

[JW93] S.Peyton Jones and P.Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (PoPL).* January 1993.

[NM96] G.Nestares and P.E.Martínez López. Mónadas como generalización de funciones. In *Taller de Programación Funcional, 25 JAIIO*, 1996. Enviado para su evaluación.

[PH+96] J.Peterson, K.Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale Univ., May 1996.

[Röj95] N.Röjemo. Efficient parsing combinators. In *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages.* Chalmers Univ. of Technology. Sweden, May 1995.

[RW95] C.Runciman and D.Wakeling, editors. *Applications of Functional Programming.* UCL Press, 1995.

[Wad92] P.Wadler. Monads for functional programming. In M.Broy, editor, *Program design calculi, proc. of the Marktoberdorf Summer School.* Springer, 1992.

[Wad95] P.Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925.* Springer, May 1995.

[WB89] P.Wadler and S.Blott. How to make ad-hoc polimorphism less ad-hoc. In *16'th Symposium on Principles of Programming Languages*, Texas. ACM Press, January 1989.