Simulando Operadores Disfijos

Diego Echeverri Saldarriaga

22 de abril de 2009

Índice general

Introducción

> *"A good Notation has a sublety and suggestiveness which at times makes it almost seem like a live teacher."*
> –Bertrand Russell[1]

La idea de permitir al usuario crear y modificar operadores dentro del lenguaje de programación tiene como objetivo permitir sintaxis más concisa. Algunos lenguajes de programación permiten al programador definir sus propios operadores infijos([11], [13]). Otros permiten el concepto más general de operador disfijo([7], [8], [6] y otros). Operadores disfijos son descritos en [10] y [1]. Escencialmente se trata de operadores de una o mas partes que pueden traer los argumentos embebidos en si mismos. Un ejemplo tomado de OBJ es el siguiente:

```
op if_then_else_fi : Bool Int Int -> Int .
```

La idea de este trabajo se centra en el estudio de los operadores permisivos como mecanismo de extensibilidad en un lenguaje de programación.

El capitulo 1 es un breve recuento de los objetivos generales y específicos. El capítulo 2 hace un recorrido sobre diferentes conceptos necesarios para este trabajo. El capítulo 3 justifíca este trabajo en el área de diseño de lenguajes de programación. El capítulo 4 explica los detalles de diseño y diferentes estrategias utilizadas para el *"parsing"* y evaluación de operadores distfijos permisivos. El capítulo 5 ilustra como puede utilizarse el lenguaje de programación como mecanísmo de extensibilidad para construir características pseudo-objetuales. Finalmente el capitulo 6 describe las conclusiones respecto la implementación de dicho lenguaje y describe posible trabajo futuro alrededor de la inclusión de operadores permisivos.

[1] Prólogo del capítulo *"Minilanguages: Finding a Notation that Sings"* en *"The art of Unix Programming"* por Eric Raymond.

Capítulo 1

Objetivos

1.1. Objetivo General

Exploración del uso de operadores disfijos permisivos en un lenguaje fuertemente tipado.

1.2. Objetivos Específicos

- Entender que son los operadores disfijos.
- Implementar un lenguaje de programación prueba de concepto que simule operadores disfijos mediante el lenguaje de programación Haskell.
- Utilizar los operadores disfijos del lenguaje como mecanismo de extensibilidad para implementar un lenguaje con características objetuales. Dicha construcción se realizará en etapas.

Capítulo 2

Importancia del problema.

Muchos de los avances en el desarrollo de software han tenido su origen en el area de lenguajes de programación. Los desarrollos en lenguajes de programación han permitido:

1. Escribir software en forma portable gracias a la posibilidad de abstraer los detalles arquitectónicos de la plataforma para la cual se desarrolla el software.

2. Escribir software mas seguro, ocultando y restringiendo ciertas construcciones inseguras del lenguaje objeto.[1]

3. Reutilización de código, mediante la incorporación de abstracciones para la generalización de soluciones. [2]

De la misma manera, desarrollos en lenguajes de programación han permitido el desarrollo de lenguajes mas expresivos. Entendiendo por lenguaje expresivo es aquel que brinda construcciones sintácticas y abstracciones que permiten al programador resolver problemas en forma mas "elegante" [3].

La importancia de la elegancia en los lenguajes de programación además de tener ventajas en mantenibilidad, y legibilidad tambien parece ser pieza importante en la reducción de errores. Existen indicios [17] para afirmar que la densidad de errores por número de líneas no se ve alterada por la elección del lenguaje de programación. De esta forma, lenguajes que requieren menos líneas de código para resolver diferentes problemas (lenguajes expresivos) pueden ayudar a disminuir la cantidad de errores.

Brindar expresividad enfrenta al diseñador del lenguaje de programación al dilema de Cardelli [4]: Decidir entre dar una amplia y expresiva notación o

[1] Entendiendo lenguaje objeto como el lenguaje producido por el lenguaje de programación. Por ejemplo, **x86** es uno de los lenguajes objeto de **C**

[2] Un ejemplo de ésto es Polimorfismo. El lenguaje diseñado en esta tésis tiene características polimórficas. Más sobre esto en la sección 4.

[3] Elegante puede tener connotaciones subjetivas, sin embargo el concepto es formalizado en [5]

tener un *"core"* pequeño. Ambas propiedades son deseables en un lenguaje de programación. Los *"cores"* pequeños son mas fáciles de mantener y permiten que el lenguaje sea asimilado más fácil por los programadores. Existe un enfoque híbrido basado en lenguajes extensibles. Estos lenguajes parten de un *"core"* pequeño, pero permiten que el usuario defina su propia sintáxis.

Esfuerzos para construir lenguajes que puedan "crecer" [16] se han realizado anteriormente. Dentro de las técnicas utilizadas en este enfoque hibrido cabe mencionar las siguientes: *"Syntax Macros"* [2], *"Extensible Syntax"* [4], *"Conctypes"* [1].

Este trabajo complementa dichos esfuerzos mostrando como un subconjunto de operadores disfijos puede ser utilizado como mecanismo de extensibilidad en los lenguajes de programación.

Capítulo 3

Preliminares

3.1. Conceptos de diseño de lenguajes de programación

Esta sección presenta conceptos generales sobre construcción de lenguajes de programación. Dichas definiciones han sido basadas en [14]

3.1.1. Sintáxis concreta versus sintáxis abstracta

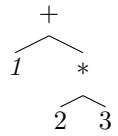
Definición 1. *Sintaxis concreta en lenguajes de programación se refiere a la representación del programa como cadenas de caracteres. Dicha representación es la que sirve de interfaz para el programador. La sintáxis concreta es el primer nivel en la definición de la sintaxis de un lenguaje.*

Ejemplo 1. *El siguiente es un ejemplo de sintaxis concreta para una expresión matemática.*

$$1 + 2 * 3$$

Definición 2. *Sintáxis Abstracta es el segundo nivel en la representación de sintáctica de un programa. Este nivel es alcanzado mediante dos procesos. Analisis léxico y parsing. El analisis léxico descompone la cadena de caracteres en "tokens" o lexemas como identificadores, literales y puntuación. El segundo paso transforma la lista de tokens en un árbol de sintaxis abstracta. Es en este paso es donde se resuelve la presedencia de operadores.*

Ejemplo 2. *El siguiente arbol representa la sintáxis abstracta del ejemplo presentado en 1*



Como se mencionó anteriormente la precedencia es resuelta en el paso anterior a la construcción del arbol sintáctico abstracto. De esta forma, la representación abstracta omite los parentesis.

3.1.2. Gramáticas libres de contexto, Notación y ambigüedad.

Una gramática libre de contexto es un formalismo desarrollado por Noam Chomsky para describir en forma recursiva la estructura de bloques de los lenguajes. Dicho formalismo es ampliamente utilizado para la descripción formal de los lenguajes de programación. Una de las razones es que dada la gramática libre de contexto, o un subconjunto de esta, es posible construir algoritmos que reconozcan el lenguaje expresado por dicha gramática en forma programática. Dos algorítmos ampliamante usados para este propósito son: LL y LR.

Formalmente, una gramática libre de contexto consta de un conjunto de símbolos terminales (V). Un conjunto de símbolos no terminales Σ, un conjunto de reglas generadoras $V \rightarrow (V \cup \Sigma)^*$. Y un $S \in V$ llamado símbolo inicial. Usualmente se utiliza convención Bakus-Naur o una de sus derivadas para definír la gramática. En este trabajo usaremos la siguiente convención:

Ejemplo 3. *Simple gramática para la definición de números.*

$\langle number \rangle \rightarrow \langle digit \rangle | \langle digit \rangle \langle number \rangle$
$\langle digit \rangle \rightarrow \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9}$

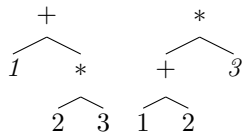
1. Símbolo inicial es el primero en la lista de "producciones'
2. Símbolos no terminales son aquellos en negrilla.
3. Símbolos no terminales están encerrados en $\langle\rangle$

Definición 3. *Una gramática libre de contexto es ambigua si existe una cadena perteneciente al lenguaje generado por dicha gramática y para esta, existe mas de un arbol sintáctico.*

Ejemplo 4. *El siguiente es un ejemplo de expresiones aritméticas ambiguas. (Por brevedad usaremos la misma definición de $\langle digit \rangle$ anteriormente mencionada)*

$\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$
$\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$
$\langle E \rangle \rightarrow \langle digit \rangle$

Dada la anterior gramática y la expresión $1+2*3$ *es posible construir dos árboles sintácticos.*



Esto es suficiente para concluir que la gramática es ambigua.

3.1.3. Operadores, precedencia y asociatividad

El ejemplo 4 muestra dos arboles sintácticos para la misma expresión aritmética. Solo uno de estos arboles cumple con el orden de evaluación esperado. El orden de evaluación esperado no es mas que el concepto de precedencia.

3.2. Lambda Calculo

Lambda cálculo es un formalismo matemático detras de la definición y aplicación de funciones. Dicho formalismo además representa la noción de computabilidad en forma equivalente a la maquina de Turing [3].

Lambda expresiones

El elemento constitutivo del lambda cálculo son "lambda expresiones".

Definición 4. *Una lambda expresión puede ser definida en forma recursiva de la siguiente manera:*

1. *Si* v *es una variable,* v *es una lambda expresión.*

2. *Si* v *es una variable y* M *es una lambda expresión,* $\lambda v.M$ *es una lambda expresión. A esto operación nos referiremos como abstracción de* M *sobre* v*.*

3. *Si* N *y* M *son lambda expresiones entonces* N M *también es una expresión. Esto es llamado aplicación (*M *se aplica a* N*)*

Variables Libres

La operación de abstracción liga[1] la variable asociada. Son variables libres aquellas que no están ligadas por ninguna abstracción.

Ejemplo 5. *En* $\lambda x.x\,y$ *la variable* x *en la expresión está ligada y la variable* y *está libre.*

[1] El termino usual en la literatura es *"bind"*.

α-Conversión

Dos expresiones son alfa-equivalentes (denotado por $=_\alpha$) si es posible transformar una en otra, renombrando las variables.

Ejemplo 6. $\lambda x.x\, y =_\alpha \lambda z.z\, y$

Renombrar las variables ligadas de una lambda expresión es conocido como α-Conversión. Existen dos casos especiales para tener en cuenta al realizar α-Conversión. En primer lugar, no es posible realizar α-Conversión de una expresión por una variable que se encuentre libre en dicha expresión. Esto es llamado captura de nombres[2].

Ejemplo 7. $\lambda x.x\, y \neq_\alpha \lambda y.y\, y$

El segundo caso se refiere al ámbito de las abstracciones. $\lambda x.\lambda x\, y$

Notación

Por practicidad seguiremos ciertas convenciones notacionales descritas en [15].

1. Los paréntesis más externos serán omitidos. De esa forma $(N\, M)$ será escrito: $N\, M$
2. La abstracción se extiende hasta donde sea posible. $\lambda x.N\, M$ es equivalente a $\lambda x.(N\, M)$ y no a $(\lambda x.N)M$
3. Aplicación es asociativa por izquierda. $M\, N\, O$ es equivalente a $(M\, N)\, O$ y no a $M(N\, O)$
4. Podemos abreviar la expresión $\lambda x.\lambda y.M$ de la forma: $\lambda x\, y.M$

Lambda cálculo puro versus lambda cálculo aplicado.

Existen diferentes codificaciones bien conocidas usando lambda expresiones para diferentes tipos de datos usuales en lenguajes de programación [3]. Dichas codificaciones resultan convenientes como mecanismo de exploración del lambda cálculo como formalismo de computabilidad; sin embargo, ineficiente a la hora de implementar lenguajes de programación. Debido a esto suelen añadirse construcciones al lambda cálculo "puro.^{ex}plicado anteriormente. Dichos lambda cálculos extendidos suelen denominarse lambda cálculos aplicados. A continuación se ilustra la relación de ambos conceptos [4]

$$\begin{aligned} \textit{programming language} &= \textit{applied lambda calculus} \\ &= \textit{pure lambda system} + \textit{basic data types} \end{aligned}$$

Ahora agregaremos 2 nuevas reglas a la definición 4 de lambda expresión. Siguiendo la terminología usada lo resultante es un lambda cálculo aplicado.

[2] En la literatura se refieren a *"name capture"*

[3] El capítulo 3 de [15] incluye dichas codificaciones para enteros, tuplas, listas arboles

[4] Tomado de [12] Página 370.

1. Si k es un entero o k es una cadena de caracteres entonces k es una constante.
2. Si k es una constante entonces k es una lambda expresión.

3.2.1. Estrategias de Reducción

3.2.2. Sistema de Tipos

Capítulo 4

Parsing, Análisis y Evaluación.

Esta sección explica los diferentes pasos en la implementación de nuestro lenguaje para soportar operadores permisivos.

4.1. Una visión general

Inicialmente

$$Texto \rightarrow_1 Tokens \rightarrow_2 (Declaraciones, Main)$$

4.2. Lexing y Parsing

El primer paso de parsing detecta las declaraciones de funciones y la función principal main. Esta es la gramática del lenguaje siguiendo la notación del capítulo 3.

⟨Program⟩→⟨Declarations⟩⟨Main⟩

⟨Declarations⟩→⟨Declarations⟩⟨Declaration⟩|ϵ

⟨Declaration⟩→⟨Infix⟩|⟨Prefix⟩|⟨Suffix⟩|⟨Closed⟩

⟨Infix⟩→⟨Init⟩⟨InfixKeyword⟩⟨Precedence⟩⟨Id⟩⟨Id⟩⟨Id⟩=⟨Definition⟩

⟨Precedence⟩→ϵ⟨Number⟩

⟨InfixKeyword⟩→**infixr**|**infixl**

⟨Init⟩→**let**|**let rec**

⟨Prefix⟩→⟨Init⟩⟨Id⟩⟨Ids⟩=⟨Definition⟩

⟨Suffix⟩→⟨Init⟩**suffix**⟨Ids⟩⟨Id⟩=⟨Definition⟩

⟨Closed⟩→⟨Init⟩**closed**IdIdsId=⟨Definition⟩

⟨Ids⟩→⟨Id⟩⟨Ids⟩|ϵ
⟨Definition⟩→⟨ExpresionTokens⟩
⟨ExprTokens⟩→⟨ExprToken⟩⟨ExprTokens⟩|ϵ
⟨ExprToken⟩→⟨Literal⟩|⟨Id⟩
⟨Literal⟩→⟨Number⟩|⟨String⟩
⟨Number⟩→⟨Digit⟩|⟨Digit⟩⟨Number⟩
⟨Digit⟩→**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**
⟨String⟩→**"**⟨Chars⟩**"**
⟨Chars⟩→⟨Char⟩⟨Chars⟩|ϵ
⟨Char⟩→Cualquier caracter
⟨Id⟩→⟨Char⟩|⟨Char⟩⟨Id⟩
⟨Main⟩→**main =** |⟨Definition⟩

Ahora bien. Para definir una expresión añadiremos notación adicional: Op_p^a Indica que el operador tiene precedencia p asociatividad a. Usaremos un entero para denotar p y $\{l, r\}$ son los posibles valores de a. l denota asociatividad por izquierda y r asociatividad por derecha.

$$Expr := Open\, Expr\, Closed$$

$$Expr := Expr_1 Infix_n^r\, Expr_2 | \forall_{Op_m^r \in Expr_1 or Expr_2} n \leq m$$

En español... Una expresión puede estar formada por un operador infijo operando 2 subexpresiones. siempre y cuando el operador tenga la menor precedencia. En la defición falta agregar el manejo de asociatividad (en el caso de $Infix_n^r$ se debe garantizar que no exista ningún operador con la misma precedencia y misma asociatividad en $Expr_2$).

$$Expr := Expr_1 Infix_n^l\, Expr_2 | \forall_{Op_m^j \in Expr_1 or Expr_2} n \leq m$$

$$Expr := PrefixOp\, Expr$$

$$Expr := Expr\, SuffixOp$$

$$Expr := Literal$$

$$Literal := string | natural$$

4.3. Resolución de operadores

Data: Lista de ExprTokens
Result: Arbol de evaluación
initialization;
if *understand* **then**
| go to next section;
end

Algoritmo 1: Resolución de precedencia

4.4. Sistema de tipos

El lenguaje de programación es fuertemente tipado y está construido a partir de la implementación encontrada en [9]. El sistema de tipos esta conformado por los siguientes constructores en Haskell.

```
data Type    =  TVar String
             |  TInt
             |  TBool
             |  TString
             |  TList Type
             |  TProd Type Type
             |  TFun Type Type
```

El sistema de tipos cuenta con variables de tipos, enteros, booleanos, cadenas de caracteres, listas genéricas, funciones y producto de tipos.

4.5. Construcción de notación aritmética

4.6.

Capítulo 5

Conclusiones

Bibliografía

[1] Anika Aasa. *User Defined Syntax*. PhD thesis, Chalmers University of Technology, 1992.

[2] Arthur Baars and Doaitse Swierstra. Syntax macros. Unfinished Draft.

[3] Henk Barendregt and Erik Barendsen. Introduction to lambda calculus. Disponible en línea: `http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/geuvers.pdf`, March 2000.

[4] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. Technical report, Systems Research Center, 1994.

[5] G. J. Chaitin. *People and Ideas in Theoretical Computer Science*, chapter Elegant LISP Programs. Springer-Verlag, 1999. Disponible en línea: `http://www.cs.auckland.ac.nz/CDMTCS//chaitin/lisp.html`.

[6] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewriting logic. Technical report, Computer Science Laboratory SRI International, 1999.

[7] Equipo de Coq. The Coq Proof Assistant Reference Manual, Version 8.1., 2007.

[8] Joseph A. Goguen, Timothy Winkler, Jose Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, Menlo Park, 1996.

[9] Martin Grabmüller. Algorithm w step by step. Draft paper, September 2007.

[10] Simon Peyton Jones. Parsing distfix operators. *Communications ACM*, 29(2):118–122, feb 1986.

[11] Simon Peyton Jones. Haskell 98 Language and Libraries: The Revisited Report. Technical report, Cambridge University Press, 2003.

[12] Jan Leeuwen and Jan van Leeuwen, editors. *Handbook of Theoretical Computer Science: Formal models and semantics*. MIT Press, 1994.

[13] Xavier Leroy. The Objective Caml system. Technical report, "Institut National de Recherche en Informatique et en Automatique", 2008.

[14] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[15] Peter Selinger. Lecture notes on the lambda calculus. Disponible en línea `http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf`. Notas de un curso de lambda cálculo de la universidad de Ottawa de 2001.

[16] Guy L. Steele, Jr. Growing a language. *Higher Order Symbol. Comput.*, 12(3):221–236, 1999.

[17] Ulf Wiger. Four-fold increase in productivity and quality. Technical report, Ericsson, 2001.