Una Extensión de Caml-Light para permitir Definiciones de Listas por Comprensión

Gabriel Tamura, Rodrigo López
Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes
Bogotá - Colombia
e-mail: rlopez@uniandes.edu.co, gtamura@uniandes.edu.co

16 de febrero de 1996

Resumen

Los lenguajes de programación en general siempre han tendido a adoptar en sus construcciones sintácticas la simbología usual de las matemáticas, en parte por la familiaridad que se tiene con éstas, en parte por la expresividad y brevedad de sus notaciones. Un claro ejemplo que presenta tales condiciones es la notación que se usa para la definición de conjuntos, y que particularmente aparece en los lenguajes de programación, adaptada para definir listas. En este artículo se plantea la realización de una extensión sobre caml-light, una implementación portable y "liviana"del lenguaje ML, para soportar definiciones de listas por comprensión; la extensión se especifica de manera precisa y se muestran los pasos seguidos en su implementación, comentando las decisiones tomadas en cada uno de ellos. Como resultado, se obtiene una versión extendida de caml-light, no más poderosa pero sí con un mecanismo que permite escribir, al menos algunos programas, de una manera más breve y más cercana a lo que se conoce como *especificación ejecutable*, un atributo actualmente reconocido en muchos lenguajes de programación funcional modernos.

Palabras clave: lenguajes de programación, paradigmas de programación, programación simbólica, programación funcional, compiladores.

1 Introducción

Los lenguajes de programación funcional, a diferencia de los imperativos, históricamente han sido diseñados sin las "ataduras" impuestas por una máquina, cuidando más bien un criterio de simplicidad conceptual, hasta cierto punto heredado de las matemáticas, teniendo como esencia el cálculo lambda [Wik92].

Esta simplicidad conceptual rige la definición y notación tanto de los datos como de los algoritmos, los dos constituyentes básicos de un programa, aunque cada lenguaje presenta variaciones al determinar sus reglas léxicas y sintácticas. En cuanto a los datos, la mayoría de estos lenguajes tienen implementados los tipos básicos usuales, y de manera sobresaliente, el tipo *lista* casi como único estructurador de datos, razón por la que las listas resultan de gran importancia en ellos; de este modo, se permite su definición de diversas maneras, aunque todas representan variantes de dos formas básicas: la definición por extensión y la definición por comprensión, ambas

conceptualmente derivadas de la noción de definición de conjuntos en las matemáticas. Concretamente, algunas implementaciones de lenguajes de programación funcional, como Miranda [Tur85], admiten variantes de las dos formas básicas de definición de listas, mientras que otras, como Caml-Light [Ler90], sólo permiten variantes de la primera, sin restringir en absoluto (dadas las características del lenguaje) el conjunto de listas así definibles.

En este artículo se describe una extensión a la implementación Caml-Light para que pueda soportar definiciones de listas por comprensión. Aunque éstas no son más poderosas en el sentido de la definición misma (es decir, en los datos), ofrecen un mecanismo que permite escribir expresiones de una manera más breve, y de tal modo que la correspondencia con la especificación de lo que ellas denotan resulta evidente. Incluso, podría decirse que brindan una simplicidad conceptual adicional al permitir un nivel de abstracción mayor en la que se hace más patente el concepto de *especificación ejecutable*, usualmente reconocido en los lenguajes de programación funcional modernos [Fro90] [Wen93].

La extensión se presenta en detalle y se muestra cómo enriquece el lenguaje , dando una descripción precisa de las nuevas construcciones permitidas y algunos ejemplos. La organización del artículo es como sigue: La sección 2 presenta algunas características importantes de la implementación caml-light; la sección 3 describe la definición de listas por comprensión en sí y el mecanismo a través del cual éstas pueden transformarse en construcciones caml-light puras, de manera que no es necesario modificar ni el sistema de tipos, ni la generación de código ni demás componentes de dicha implementación; la sección 4 presenta los detalles más importantes de cómo se implementó la extensión, recopilando de manera general los pasos seguidos; en la sección 5 se aclaran algunas restricciones que resultan del esquema escogido para realizar la implementación de la extensión; y finalmente, en el apéndice A se incluyen algunos ejemplos que ilustran las distintas formas de definición de listas por comprensión soportadas en la extensión y sus restricciones.

2 Caml-Light y su implementación

El lenguaje de programación ML fue propuesto por Robin Milner en 1978, originalmente con el propósito de servir como el lenguaje de control del sistema de deducción automática LCF. Es un lenguaje fuertemente tipado y usa el peculiar pero poderoso sistema de tipos de Damas-Milner, con un régimen de evaluación aplicativo y una semántica estricta[DM82].

Caml-Light es una implementación de este lenguaje, desarrollada por Xavier Leroy desde 1990 [Ler90], [Mau95][1]. Diseñada con el objetivo principal de ser portable y "liviana", puede correr confortablemente incluso en un microcomputador, el principal vehículo para difundir software en general; en su diseño siempre ha prevalecido la modularidad y el crecimiento armónico a costa de una mayor eficiencia y del soporte de características más avanzadas y poderosas, lo que le ha permitido mantener una gran consistencia en el código fuente, facilitando su entendimiento y por lo tanto también la realización de extensiones sobre el mismo.

El código fuente del sistema caml-light [2] está escrito fundamentalmente en caml-light mismo (la gran mayoría) y en C. Consta de un compilador para el lenguaje y su librería, un sistema de soporte de ejecución (runtime system) que incluye un intérprete del código intermedio (bytecode)

[1] La versión utilizada en este desarrollo fue la 0.7beta4 [Ler95].

[2] Los fuentes de todo el sistema están disponibles vía ftp anónimo en `ftp.inria.fr:/lang/caml-light`.

generado, un encadenador, un manejador de librerías[3], un ambiente interactivo estilo intérprete, un generador de analizadores léxicos, un generador de analizadores sintácticos, y los compiladores para *bootstrapping*. Aunque está disponible en variantes (ligeramente) distintas para UNIX, MS-DOS y MacIntosh, en este trabajo sólo se examinaron las dos primeras. Los archivos fuente son prácticamente idénticos (exceptuando los que están escritos en C), lo que permite desarrollar la extensión en cualquiera de las plataformas, y luego pasar, con algo de cuidado, los archivos modificados a las otras.

El compilador opera en múltiples pasadas y está implementado como un compilador "de frase", que genera un código intermedio denominado *bytecode*. Este es un código hilado que se presta para ser interpretado muy eficientemente y que resulta portable entre máquinas con arquitecturas diferentes, propiedad que resulta importante sobre todo para hacer más sencillo el proceso de *bootstrapping*, pues de este modo, inicialmente sólo es necesario compilar el sistema de soporte de ejecución (escrito en C) [Ler90].

El sistema caml-light ofrece dos modos de uso: el primero, a través de un ambiente interactivo del estilo de los intérpretes de lenguajes de programación funcional clásicos, ideal para aprender a usar el lenguaje; y el segundo, en modo *batch*, como los compiladores de línea utilizables con herramientas como make.

3 Definición de listas por comprensión

Usualmente, para especificar los elementos de un conjunto en matemáticas, se usa la notación conocida como "notación de lista" (o notación de "definición por extensión") [Wal72], en la que los nombres de los miembros del conjunto se escriben en una línea y separados por comas, encerrados entre llaves. Por ejemplo, el conjunto de los enteros positivos pares menores que 10, podría escribirse como:

$$\{2, 4, 6, 8\}$$

Sin embargo, un conjunto también puede especificarse por medio de las restricciones o propiedades que todos (y solamente) sus miembros poseen. El mismo ejemplo anterior, en esta "notación de predicado" (o notación de "definición por comprensión") [Wal72] podría escribirse así:

$$\{x \mid x \text{ es entero par positivo y menor que } 10\}$$

o, separando y distinguiendo las restricciones:

$$\{x \mid x > 0, x \text{ es par }, x < 10\}$$

donde la barra vertical que aparece después de la primera ocurrencia del identificador x se lee "tal que", y las comas se entienden como separadores.

A continuación se verá cómo, en los lenguajes de programación que ofrecen el tipo lista como tipo predefinido, se adaptan estas notaciones para definir, no conjuntos sino listas, presentando primero una caracterización sintáctica de la adaptación, y seguidamente algunos ejemplos ilustrativos.

[3] Deberíamos decir *bibliotecas de programas*

3.1 Descripción de las definiciones de listas por comprensión

Como ya se mencionó, el tipo *lista* es importante en los lenguajes de programación funcional por el papel que desempeñan en ellos, incluyendo a caml-light. Si se acepta la definición de *lista* como colección de valores linealmente ordenados [BW88], puede verse su relación con el concepto matemático de conjunto, en el sentido en que enumerar un conjunto es producir una lista con sus miembros en algún orden. Sin embargo, en un conjunto no tiene sentido decir que un elemento está más de una vez, o que sus elementos aparecen en él en un orden dado, mientras que en una lista sí. A partir de esta similitud, en estos lenguajes de programación se adaptan las dos maneras de definir conjuntos en matemáticas (por extensión, y por comprensión) para definir listas, teniendo en cuenta las diferencias mencionadas anteriormente. La adaptación se realiza sintácticamente de manera distinta en cada lenguaje, según la simbología que cada uno establece. Por ejemplo, en caml-light, los delimitadores de lista son los corchetes (es decir, [y]), y el separador de los elementos nombrados explícitamente es el punto y coma (;).

De este modo, en la definición de listas por extensión es necesario enumerar todos los elementos de la lista, en el orden que se requiera, y separarlos por algún símbolo, como en los siguientes ejemplos[4]:

i. `[ 1; 2; 3; 4; 5 ]`
ii. `[ (1,'a'); (2,'b'); (3,'c'); (4,'d') ]`
iii. `'a'::'c'::'z'::[]`

Por otro lado, la enumeración de los elementos de una lista definida por comprensión se obtiene mediante la caracterización de los mismos, a través de cualificadores (restricciones o propiedades) usando la siguiente sintaxis, correspondientemente adaptada de la convencional en matemáticas [BW88]:

```
[ <expresion> | <cualificador1> , ... , <cualificadorN> ]
```

donde

- la barra vertical se sigue leyendo "tal que", y como separador estandarizado para las restricciones se usa la coma.
- `<expresion>` denota cualquier expresión válida del lenguaje en el que se esté haciendo la definición (con algún sentido en ese contexto)
- `<cualificador>` puede ser una de las dos opciones siguientes:
  - un predicado (expresión booleana).
  - un `<generador>`, que a su vez puede tener una de las formas siguientes:
    * `<identificador> : <lista>`
    * `<N-identif-separados-por-coma> : <lista-de-N-tuplas>`

Y, naturalmente,

- `<N-identif-separados-por-coma>` denota una N-tupla de identificadores.

[4] En éste y en los siguientes ejemplos se usará la notación de caml-light, a menos que se especifique otra cosa.

- <lista> y <lista-de-N-tuplas> son a su vez listas, que pueden estar definidas de alguna de las siguientes maneras:
  - por un rango de enteros, es decir, una expresión de la forma [<expr_entera> .. <expr_entera>].
  - por extensión.
  - por comprensión.
  - por un identificador (ya sea que se defina local o globalmente).
  - por una expresión arbitraria de tipo lista, entre paréntesis.
  - por concatenación repetida de cualquiera de las anteriores.

Un <generador> establece una correspondencia entre:

- Un <identificador> y una <lista>, o, entre
- <N-identif-separados-por-coma> y <lista-de-N-tuplas>.

desde el punto de vista semántico, define una variable (en el caso de <identificador>) o varias (en el caso de <N-identif-separados-por-coma>) local(es) a la definición de lista por comprensión más externa en la que aparecen, y su alcance se limita a ésta. Por lo tanto, el nombre asignado a cada una de ellas no es importante, en el sentido en que puede ser reemplazado consistentemente por cualquier otro, siempre y cuando ésto no genere colisiones. En el proceso de generación de los elementos de la lista en definición, cada variable se instanciará sucesivamente en los distintos valores dados en la <lista> o <lista-de-N-tuplas> que le corresponde, en el sentido arriba mencionado. Además, en una definición pueden aparecer varios generadores, en cuyo caso los posteriores varían más rápidamente que sus precedentes e incluso pueden depender de las variables definidas en ellos.

La otra clase de <cualificador>, una expresión booleana, impone una restricción sobre los valores en los que se instancia una o algunas de las variables definidas por un <generador>, actuando como un "filtro"; es decir, dejando pasar sólo los valores que finalmente deben usarse en la <expresión> para la generación de elementos. Por lo tanto, siempre deberá aparecer después de estos generadores.

Conceptualmente, los cualificadores "generan" valores mediante combinaciones de restricciones, y estos valores se usan en la evaluación de la <expresión>, para así ir obteniendo sucesivamente los elementos de la lista. Debe notarse que la <expresión> puede no depender de las variables definidas por los generadores, en cuyo caso estos valores no tendrán sentido en la evaluación de la misma.

Para dar algunos ejemplos ilustrativos de lo anterior, considérese par e intsqrt respectivamente como:

```
let par x = (x mod 2 = 0);;
par : int -> bool

let intsqrt x = int_of_float(sqrt (float_of_int x));;
intsqrt : int -> int
```

Entonces, según las definiciones sintácticas previas, los siguientes son ejemplos de definiciones de listas bien formadas:

i. `[ 3*x+2 | x : [ 3..12 ], par x ]`
ii. `[ (i,10*j) | i : [2..3], j : [1..7], par j ]`
iii. `[ d | d : [1..n], n mod d = 0 ]`
iv. `[ '#' | i : [1..4] ]`

que se podrían leer como (para el caso *i.*) "la lista de los valores $3 * x + 2$, tales que x esté en el rango 3..12 y x sea par". El sistema calcula, respectivamente, los siguientes valores[5]:

```
- : int list = [14; 20; 26; 32; 38]
- : (int * int) list = [2, 20; 2, 40; 2, 60; 3, 20; 3, 40; 3, 60]
- : int list = [1; 2; 3; 4; 6; 12]
- : char list = ['#'; '#'; '#'; '#']
```

Naturalmente, estas definiciones pueden usarse en expresiones más complejas, como en el siguiente ejemplo, en el que se da una definición para decidir la primalidad de un entero, y luego ésta se usa para construir la lista de los números primos entre dos enteros, inclusive:

```
let primo' n = (divisores_prop n = [])
    where divisores_prop m = [ d | d:[2..intsqrt m], m mod d = 0 ]
;;

let primos n m = [ x | x:[n..m], primo' x ]
;;
```

Dadas estas definiciones, la expresión

```
primos 500 570;;  (* los primos entre 500 y 570 *)
```

da como resultado el siguiente valor

```
- : int list = [503; 509; 521; 523; 541; 547; 557; 563; 569]
```

3.2 Mecanismo de transformación sintáctica

Como se dijo inicialmente, las definiciones de listas por comprensión descritas en la sección anterior pueden reescribirse sintácticamente como construcciones básicas de cualquier lenguaje de programación funcional puro y ejecutarse simplemente suministrando unas cuantas funciones de soporte. Esto puede demostrarse *(i)* estableciendo algunas equivalencias encontradas entre funciones de órden superior y definiciones de listas por comprensión simples, y *(ii)* diseñando un mecanismo de transformación recurrente que, respetando la descripción sintáctico-semántica dada, permita reescribir éstas en expresiones funcionales puras, haciendo uso de las equivalencias dadas en *(i)*.

[5] $n = 12$ para el caso *iii.*

Las equivalencias necesarias son las siguientes (después de cada equivalencia se da el tipo de la función):

```
map f xs = [ f x | x:xs ]
map : (`a -> `b) -> `a list -> `b list

filter p xs = [ x | x:xs, p x ]
filter : (`a -> bool) -> `a list -> `a list

flat xss = [ x | xs:xss, x:xs ]
flat : `a list list -> `a list

interval_int eint1 eint2 = [ eint1 .. eint2 ]
interval_int : int -> int -> int list
```

y el mecanismo de transformación consta, básicamente, de cinco reglas[6]:

1. `[ eint1 .. eint2 ]` $\Longrightarrow$ `interval_int eint1 eint2`
2. `[ x | x:xs ]` $\Longrightarrow$ `xs`
3. `[ expr | x:xs ]` $\Longrightarrow$ `map (function x -> expr) xs`
4. `[ expr | x:xs, p x, ... ]` $\Longrightarrow$ `[ expr | x:(filter p xs), ... ]`
5. `[ expr | x:xs, y:ys, ... ]` $\Longrightarrow$ `flat [ [ expr | y:ys, ... ] | x:xs ]`

Donde `expr` denota cualquier expresión y `eint` cualquier expresión entera, según lo establecido en la sección 3.1. Como puede verse, la regla 2 es un caso particular de la regla 3, y las reglas 4 y 5, aunque simplifican "en un paso" la definición, requieren que se recurra a aplicaciones adicionales de transformación. Los tres primeros casos son, pues, los casos "base" de la recurrencia, mientras que los dos últimos, los casos "recurrentes". Por construcción, y como los casos son todos excluyentes, el mecanismo de transformación eventualmente siempre terminará.

3.3 Un ejemplo de transformación

A continuación se presenta un ejemplo de aplicación, paso por paso, del mecanismo de transformación descrito, mostrando la regla usada en cada uno. Para esto, se usará la notación '$\Longrightarrow_{ReglaUsada}$'. La definición que se transforma es la siguiente:

```
[ d | d:[2..intsqrt n], n mod d = 0 ]
```

y los pasos seguidos son:

```
      [ d | d:[2..intsqrt n], n mod d = 0 ]
⟹4   [ d | d:(filter (function d -> n mod d = 0) [2..intsqrt n]) ]
⟹1   [ d | d:(filter (function d -> n mod d = 0) (interval_int 2 (intsqrt n) )) ]
⟹2   filter (function d -> n mod d = 0) (interval_int 2 (intsqrt n))
```

[6] Adaptadas de las sugeridas en [BW88]

4 Implementación de la extensión

4.1 Revisión de algunas consideraciones para realizar la implementación

La extensión se implementó de una manera sencilla, en el nivel sintáctico, esto es, *(i)* agregando las funciones de soporte necesarias para ejecutar las definiciones transformadas a la librería; *(ii)* definiendo un tipo con los constructores de sintaxis abstracta requeridos para representar las nuevas construcciones (i.e. las definiciones de listas por comprensión); *(iii)* extendiendo las reglas gramaticales del lenguaje para aceptar la sintaxis de las nuevas definiciones; y *(iv)* implementando el mecanismo de transformación de una manera consistente con *(i), (ii) y (iii)*, de manera que no fuera necesario modificar el sistema de tipos (uno de los componentes más delicados de caml-light) ni demás partes del sistema.

Hay que anotar que igualmente se hubiera podido hacer la extensión a un nivel más bajo, de modo que su integración con los otros componentes del compilador fuera más fuerte, es decir, añadiendo los constructores nuevos al tipo del árbol de sintaxis abstracta ya existente, y no hacer la transformación de las nuevas construcciones en el momento en que se hace el análisis sintáctico, sino dejar que éstas siguieran los mismos pasos que las construcciones propias de caml-light, pasando por el sistema de tipos, el generador de código y demás componentes. Esto implicaría, sin embargo, extender (o al menos modificar) el tipo predefinido 'lista'de caml-light, modificar el sistema de tipos, y extender el sistema de generación de código, lo que representa una cantidad de trabajo mucho más grande y compleja.

La decisión de realizar la extensión de la manera ya planteada tuvo en cuenta esto último, aunque considerando además que, si bien esta última alternativa es más factible de compilar eficientemente y facilita dar mensajes de error más precisos y apropiados (del sistema de tipos, especialmente), el esquema de transformación usado mantiene el compilador más "liviano" y la extensión misma tiene más posibilidades de refinarse y permanecer compatible con futuros desarrollos del compilador, sin que el trabajo de "aplicarla" sobre nuevas versiones sea muy exigennte[7] .

4.2 Pasos seguidos en el desarrollo de la implementación

La implementación de la extensión se realizó básicamente en dos etapas, cuyos pasos se enumeran a continuación, junto con algunos comentarios pertinentes.

La primera etapa consistió en desarrollar la extensión de manera independiente del sistema caml-light, como un programa caml-light cualquiera, siguiendo los siguientes pasos:

1. Se dio una especificación formal para la definición de listas por comprensión según lo planteado en la sección 3.1, usando para ello las mismas herramientas usadas en la implementación caml- light, es decir, camllex para la parte léxica y camlyacc para la parte sintáctica.

2. Se definió la representación abstracta para las construcciones del lenguaje (el tipo con sus respectivos constructores) y se añadieron acciones semánticas a la especificación gramatical para construir dicha representación.

[7] La versión de caml-light sobre la que se implementó esta extensión, la 0.7, todavía está en período de prueba (i.e. en su fase "beta4"), y de acuerdo con el autor de la misma, la gramática, escrita en CamlYacc, sigue en evolución

3. Se implementó el mecanismo de transformación sintáctica planteado en la sección 3.2. Fue necesario hacerle algunos ajustes para que la función de transformación quedara consistente desde el punto de vista del sistema de tipos; para ello bastó adicionar algunos constructores en la representación abstracta, y las reglas (triviales) de transformación correspondientes.

4. Para poder "ver" en caml-light los resultados obtenidos hasta ese momento, se implementó una función de impresión estructurada (por niveles de indentación) de las construcciones transformadas (básicamente expresiones simples: aritméticas, booleanas, funciones, aplicaciones).

Por ejemplo, dado un archivo con la siguiente expresión

```
[ d | d:[1..a], a mod d = 0 ]
```

el sistema imprime lo siguiente:

```
(map
   (function d -> d
   )
   (filter
      (function d -> ((a mod d) = 0) )
      (interval_int 1 a)
   )
)
```

La implementación independiente obtenida se integró al compilador mismo de caml-light, procediendo de la siguiente manera:

1. Se extendió la librería con las funciones de soporte requeridas para la ejecución de las expresiones transformadas.

2. Se adicionaron los tipos para la representación abstracta de las nuevas construcciones, así como las reglas gramaticales, las acciones semánticas y el mecanismo de transformación.

En esta segunda etapa se presentaron algunos problemas debidos principalmente a la dificultad que siempre existe cuando se añaden reglas de producción a una gramática dada y sujeta a los requerimientos del esquema de parsing LALR. Los problemas que se resolvieron parcialmente o no se pudieron resolver, generaron las restricciones que se explican en la siguiente sección.

4.3 Restricciones de la extensión

En esta sección se enumeran las restricciones de la extensión que aparecieron por causa de conflictos entre la extensión sintáctica propuesta y la implementación de caml-light, así como por las decisiones de implementación que se plantearon en la sección 4.1. Debe notarse que la mayoría de las restricciones resultan de la dificultad de añadir reglas de sintaxis a la gramática de caml-light sin generar más conflictos de los que ya existen y de modo tal que el conjunto de reglas resultante, aún con conflictos, sirva en realidad para el propósito propuesto. Las restricciones, presentadas con respecto a la sección 3.1, son:

1. `<N-identif-separados-por-coma>` (que denota una N-tupla de identificadores) en ningún caso puede ir encerrada entre paréntesis.

2. En la especificación de un rango como lista, si la primera `<expresión>` del rango termina en un número entero, éste debe separarse del símbolo del rango (..) por un espacio, o encerrar la `<expresión>` entre paréntesis.

3. Los rangos, aceptados como casos particulares de lista, sólo se admiten de tipo entero.

La segunda restricción se debe a que caml-light permite escribir números flotantes con punto pero sin decimales, como por ejemplo '`23.`', cosa que genera un conflicto con la especificación de los rangos en las definiciones de lista por comprensión, tales como `[23..35]`. Para no cambiar los símbolos '..'tan usuales en ese contexto, se prefirió modificar el analizador léxico, adicionándole una regla para considerar la construcción '`INT..`'como un todo, y modificar de manera acorde el sintáctico. Esta solución, aunque alivia algo el problema, no es suficiente, pues como la única restricción que se le impone a los límites del rango es que sean expresiones de tipo entero, la construcción `[2+1..5]` debería ser válida, pero debido a la nueva regla léxica no lo es (en cambio debe escribirse, por ejemplo, `[2+1 ..5]` o `[(2+1)..5]`).

Con respecto a la tercera restricción, dado que las listas en caml-light son polimórficas, los rangos (permitidos por la extensión como casos particulares de listas) también deberían serlo, al menos para los tipos entero y caracter (i.e. el rango `[`c`..`g`]` debería ser tan válido como `[2..5]`). Sin embargo, por la decisión de implementar la extensión en el nivel sintáctico, los rangos sólo se admiten de tipo entero, por lo que, para obtener lo que se esperaría con `[`c`..`g`]` es necesario escribir, por ejemplo:

```
[ char_of_int x | x:[int_of_char `c`..int_of_char `g`] ]
```

o, algo más abreviadamente:

```
map char_of_int [int_of_char `c`..int_of_char `g`]
```

La razón para que ésto sea así, radica en que, para procesar correctamente el rango de caracteres, habría que hacer la transformación de las listas por comprensión después de que el sistema de tipos actuara y no en el nivel sintáctico, ya que los límites de los rangos podrían ser expresiones arbitrarias de tipo caracter, y no únicamente caracteres constantes, en cuyo caso el problema sí resulta sencillo de resolver.

5 Conclusiones

Se planteó una extensión a caml-light para soportar definiciones de listas por comprensión, dando una especificación precisa para la misma y mostrando los pasos seguidos en su realización, así como los problemas más sobresalientes encontrados y la manera como se resolvieron.

La implementación de la extensión es muy sencilla y adecuada para respetar, sobre cualquier otra consideración, el esquema y los formalismos con los cuales se construye caml-light: se trató de

mantener como objetivo la independencia y aplicabilidad de la extensión a futuras versiones del lenguaje.

La extensión en sí, aunque no representa más poder para la implementación caml-light, permite escribir prácticamente "especificaciones ejecutables" de una manera cómoda y sencilla para problemas cuya solución se plantea usando listas.

Por último, como consecuencia de la dificultad de añadir reglas a la gramática de caml-light de una manera adecuada, surgieron algunas restricciones sintácticas en la extensión, que se resolvieron en la medida en que lo permitió la implementación.

A Algunos ejemplos

A continuación se presentan dos ejemplos de definiciones de listas por comprensión. Se trata de los archiconocidos cálculos del máximo común divisor (gcd), y quicksort (qsort) [BW88].

A.1 Máximo común divisor

```
# let rec fold f val_ini = function
      []    -> val_ini
    | a::b  -> fold f (f a val_ini) b;;
fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>

# let maxlist = function
      []    -> raise(Failure "la lista esta vacia")
    | a::b  -> fold mayor a b
               where mayor x y = if x > y then x else y;;
maxlist : 'a list -> 'a = <fun>


# let gcd a b = maxlist [ d | d : diva, b mod d = 0 ]
    where diva = [ d | d:[1..a], a mod d = 0 ]
;;
gcd : int -> int -> int = <fun>

# gcd 21 350;;
- : int = 7
```

A.2 Quicksort

```
# let rec qsort = function
      []    -> []
    | h::t  -> ( (qsort lmen) @ [h] @ (qsort lmay)
                    where lmen = [ x | x:t, x<=h ]
                    and   lmay = [ x | x:t, x>h  ]
               )
```

```
;;
qsort : 'a list -> 'a list = <fun>

# qsort [ 4; 8; 5; 3; 20; 18; 1; 3; -33; 15; 11; 7; 2 ];;
- : int list = [-33; 1; 2; 3; 3; 4; 5; 7; 8; 11; 15; 18; 20];;
```

Referencias

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

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Nineteenth Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Fro90] R. A. Frost. Constructing programs in a calculus of lazy interpreters. In *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pages 30–42, 1990.

[Ler90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[Ler95] X. Leroy. The caml light system, release 0.7. Technical report, INRIA, 1995.

[Mau95] M. Mauny. Functional programming using caml light. Technical report, INRIA, 1995.

[Tur85] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional programming languages and computer architecture. LNCS 201*, pages 1–16. Springer Verlag, 1985.

[Wal72] R. Wall. *Introduction to Mathematical Linguistics.* Prentice-Hall, 1972.

[Wen93] E. P. Wentworth. Generalized Regular Expressions - A programming exercise in Haskell. *ACM Sigplan Notices*, 28(5):49–54, May 1993.

[Wik92] A. Wikstrom. *Functional Programming Using Standard ML.* Prentice-Hall, 1992.