Laboratorio de Programación Funcional 2024
Linter para MiniHaskell

La tarea consiste en la implementación de una herramienta de "linting"[1]. Estas herramientas son programas que reciben como entrada código fuente y reportan errores o mejoras posibles.

Los compiladores/intérpretes se encargan de reportar errores sintácticos y semánticos. Los *linters* (nombre tomado del programa *lint*) trabajan a otro nivel, encontrando y reportando patrones en programas bien formados. Por ejemplo, errores de estilo, expresiones que pueden reducirse por otras más eficientes, usos incorrectos de indentación, etc.

Consideramos para el laboratorio un subconjunto de Haskell, que llamaremos MiniHaskell. El objetivo será construir un *linter* para MiniHaskell. Nuestro linter recibe programas (que asumimos bien formados) escritos en MiniHaskell, e imprime sugerencias para modificar en el código, o las aplica transformando el fuente original, dependiendo de los argumentos que usemos.

1 MiniHaskell

La siguiente EBNF describe la sintaxis de MiniHaskell.

⟨*program*⟩ ::= { ⟨*fundef*⟩ }

⟨*fundef*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*ident*⟩ | ⟨*lit*⟩ | ⟨*expr*⟩ ⟨*expr*⟩ | '\' ⟨*ident*⟩ '`->`' ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ | '`if`' ⟨*expr*⟩ '`then`' ⟨*expr*⟩ '`else`' ⟨*expr*⟩ | '`case`' ⟨*expr*⟩ '`of`' '`[] ->`' ⟨*expr*⟩ '`;`' '`(`' ⟨*ident*⟩ '`:`' ⟨*ident*⟩ '`) ->`' ⟨*expr*⟩ | '`(`' ⟨*expr*⟩ '`)`'

⟨*lit*⟩ ::= ⟨*nat*⟩ | '`True`' | '`False`' | '`[]`'

⟨*op*⟩ ::= '`+`' | '`-`' | '`*`' | '`div`' | '`==`' | '`<`' | '`>`' | '`&&`' | '`||`' | '`:`' | '`.`' | '`++`'

Los no terminales son escritos entre paréntesis angulares, como ser ⟨*program*⟩, ⟨*expr*⟩, etc. Los terminales son escritos entre comillas simples. El no terminal ⟨*ident*⟩ representa un identificador válido y ⟨*nat*⟩ un número natural.

Un programa está formado por una lista (posiblemente vacía) de funciones. Una función se declara mediante la especificación de una ecuación, declarando el nombre de la misma y su definición. En MiniHaskell los argumentos se declaran

[1] `https://en.wikipedia.org/wiki/Lint_(software)`

```
add = \x -> \y -> x + y
```

Figure 1: Programa con una declaración

```
length = \xs -> case xs of
  []       -> 0;
  (x : xs) -> 1 + length xs

null = \xs -> case xs of
  []       -> True;
  (x : xs) -> False

not = \b -> if b then False else True

fac = \n -> if n == 0 then 1 else n * fac (n - 1)

fib = \n -> if n == 0 || n == 1 then 1
            else fib (n - 1) + fib (n - 2)
```

Figure 2: Programa con múltiples declaraciones

mediante λ-abstracciones. En la Figura 1 se puede ver un programa en el que se declara una única función que toma dos enteros y retorna la suma entre ellos.

Los cuerpos de las funciones están dados por expresiones, que pueden contener variables, literales (booleanos, enteros y la lista vacía), la aplicación de una expresión a otra, definiciones de abstracciones, aplicaciones de operadores binarios a dos subexpresiones, expresiones `if` o `case`.

Al igual que en Haskell, la construcción `case` sirve para hacer *pattern matching*. En MiniHaskell el `case` está restringido al pattern matching sobre la lista vacía `[]` y la lista no vacía de la forma (⟨ident⟩`:`⟨ident⟩). Entre las dos ramas del `case` obligatoriamente va el separador '`;`' dado que en MiniHaskell no se aplica la *off-side rule*. En la Figura 2 se puede ver un programa en el que se usan varias de las construcciones sintácticas.

2 *Lintings*

Los *lintings* que deben implementarse como parte del laboratorio trabajan todos de la misma manera: toman como entrada una porción de código, representada por una definición o una expresión, y devuelven el código producto de aplicar las mejoras sugeridas. No hay *lintings* que consideren el programa globalmente.

Los posibles *lintings* a aplicar son: computación de constantes, eliminación de chequeos redundantes de booleanos, eliminación de if redundantes, chequeo de lista vacía con *null*, eliminación de concatenación, introducción de composición, *eta*-reducción y reescritura de recursión en términos de *map*.

A continuación se detallan cada uno de estos *lintings*.

Computación de Constantes Si una expresión que aplica operadores aritméticos (`+`, `-`, `*` o `div`) o booleanos (`&&` o `||`) no contiene variables, entonces puede sustituirse en el código por el resultado de evaluarla. En el caso de una operación aritmética, solo la evalúa si su resultado no es negativo. Por ejemplo, `1 + 1` puede sustituirse por `2` y `x + (1 + 1)` puede sustituirse por `x + 2`. En cambio `1 - 3` no puede sustituirse.

Tampoco se reducen las divisiones por cero. Por ejemplo, `1 div 0` no genera sugerencias.

El *linting* puede afectar a más de una sub-expresión de una misma expresión. Las transformaciones sugeridas empiezan por las expresiones más internas, y se realizan de izquierda a derecha. Así por ejemplo, si aplicamos este *linting* sobre `x + 1 + (1 + 2 * 1)` se sugiere su reemplazo por `x + 1 + 3`. La lista de sugerencias, en orden, es:

- Reemplazar `2 * 1` por `2`

  (lo que corresponde globalmente a cambiar `x + 1 + (1 + 2 * 1)` por `x + 1 + (1 + 2)`).

- Reemplazar `1 + 2` por `3`

  (lo que corresponde globalmente a reemplazar `x + 1 + (1 + 2)` por `x + 1 + 3`).

Notar que no se sugiere el reemplazo de `x + 1 + 3` por `x + 4`, debido a que la suma asocia a la izquierda y entonces esa expresión equivale a `(x + 1) + 3`.

Eliminación de chequeos redundantes de booleanos Expresiones booleanas de la forma `e == True` y `True == e`, pueden sustituirse por `e` (para cualquier expresión `e`). También pueden sustituirse `e == False` y `False == e` por `not e`[2].

La aplicación de estas transformaciones es recursiva, empezando por las expresiones más internas, y de izquierda a derecha. Por ejemplo, ante la expresión `(x == True) || (True == y)`, al aplicar la eliminación de chequeos redundantes se sugieren las siguientes transformaciones en orden:

- Reemplazar `x == True` por `x`

  (lo que corresponde globalmente a cambiar `(x == True) || (True == y)` por `x || (True == y)`).

- Reemplazar `True == y` por `y`

  (lo que corresponde globalmente a reemplazar `x || (True == y)` por `x || y`).

[2]Notar que `not` no es un operador del lenguaje, sino que es una función que asumiremos definida. Por lo tanto no se reduce en el *linting* de computación de constantes.

Eliminación de `if` Redundantes En este caso se presentan tres *lintings* que sugieren reemplazos en expresiones que pueden simplificarse por la eliminación de `if` redundantes.

El primer caso corresponde a expresiones donde la condición es un literal (`True` o `False`) y por lo tanto pueden reemplazarse por la rama correspondiente del `if`, es decir:

- `if True then t else e` puede reemplazarse por `t`
- `if False then t else e` puede reemplazarse por `e`

El segundo caso corresponde a expresiones de la forma `if c then True else e`, las cuales siempre pueden reemplazarse por una disyunción `c || e`.

En forma similar, el tercer caso corresponde a expresiones de la forma `if c then e else False`, las cuales siempre pueden reemplazarse por una conjunción `c && e`.

La aplicación de las transformaciones es recursiva, empezando por las expresiones más internas, y de izquierda a derecha. Por ejemplo, si se aplica la eliminación de `if` redundantes a `if True then x + if False then 1 else 2 else 3`, la sugerencia es reemplazarla por `x + 2`. La lista de sugerencias, en orden, es:

- Reemplazar `if False then 1 else 2` por `2`

  (lo que corresponde globalmente a cambiar `if True then x + if False then 1 else 2 else 3` por `if True then x + 2 else 3`).
- Reemplazar `if True then x + 2 else 3` por `x + 2`

Chequeo de lista vacía con `null` Este *linting* reemplaza ocurrencias de expresiones de la forma `e == []`, o `length e == 0`, o sus simétricas, por `null e`, que es más eficiente y no requiere que los elementos de la lista tengan igualdad.

La aplicación de las transformaciones es recursiva, empezando por las expresiones más internas, y de izquierda a derecha.

Eliminación de concatenación En el caso de expresiones de la forma `e:[] ++ es` se sugiere eliminar la concatenación y hacer directamente `e:es`.

La aplicación de las transformaciones es recursiva, empezando por las expresiones más internas, y de izquierda a derecha. Por ejemplo, si se aplica esta transformación a la expresión `x:[] ++ (y:[] ++ ys)`, la sugerencia es reemplazarla por `x : (y : ys)`. La lista de sugerencias, en orden, es:

- Reemplazar `y : [] ++ ys` por `y : ys`

  (lo que corresponde globalmente a cambiar `x : [] ++ (y : [] ++ ys)` por `x : [] ++ (y : ys)`).
- Reemplazar `x : [] ++ (y : [] ++ ys)` por `x : (y : ys)`

Introducción de Composición Este *linting* reemplaza ocurrencias de expresiones de la forma `e1 (e2 e3)` por expresiones `(e1 . e2) e3` en términos del operador de composición de funciones.

La aplicación de las transformaciones es recursiva, empezando por las expresiones más internas, y de izquierda a derecha. Por ejemplo, si se aplica la introducción de la composición en la expresión `f (g (h x))`, la sugerencia es reemplazarla por `(f . (g . h)) x`. La lista de sugerencias, en orden, es:

- Reemplazar `g (h x)` por `(g . h) x`

  (lo que corresponde globalmente a cambiar `f (g (h x))` por `f ((g . h) x)`).

- Reemplazar `f ((g . h) x)` por `(f . (g . h)) x`

Eta-reducción Este *linting* reemplaza ocurrencias de expresiones de la forma `\x -> e x` por `e`, siempre y cuando `x` no ocurra libre en `e`. Se dice que una variable ocurre libre en una expresión si aparece en la expresión y no está ligada por una lambda abstracción. Por ejemplo, en la expresión `\x -> x + y`, la variable `y` ocurre libre, mientras que la variable `x` ocurre ligada.

La aplicación de las transformaciones es recursiva, empezando por las expresiones más internas, y de izquierda a derecha.

Por ejemplo, si se aplica *eta*-reducción en la expresión `\y -> (\x -> f y x) y`, la sugerencia es reemplazarla por `\y -> f y y`. Notar que no se continúa reduciendo porque `y` ocurre libre en `f y`.

Reescritura de recursión en términos de `map` Este *linting* se aplica a funciones sobre listas definidas por el siguiente esquema:

```
func = \l -> case l of
  []       -> [];
  (x : xs) -> e[|x|] : func xs
```

donde escribimos `e[|x|]` para denotar a una expresión que puede (o no) tener ocurrencias libres de `x`, pero **no tiene** ocurrencias libres ni de `func`, ni de `xs`, ni de `l`.

Funciones escritas usando este esquema pueden reescribirse en términos de `map` de la siguiente manera:

```
func = map (\x -> e[|x|])
```

Por ejemplo, dada la declaración:

```
incr = \ls -> case ls of
  []       -> [];
  (x : xs) -> (x + 1) : incr xs
```

La sugerencia es reemplazarla por:

```
incr = map (\x -> x + 1)
```

```
foo = 1 +  if True then 2 else 4

bar = \xs -> case xs of
    []       -> [];
    (l : ls) -> (\ys -> (\ls -> length ls == 0) ys) l:[] ++ bar ls
```

Figure 3: Programa con posibles sugerencias

3 Aplicación de los Lintings

El *linter* a implementar recibe como argumentos una opción, que puede ser `-s`, `-v` o `-c`, y el nombre de un archivo fuente.

Si la opción es `-s`, se imprimen en la salida estándar las sugerencias del *linter*. Si la opción es `-v`, también se imprimen las sugerencias en la salida estándar pero mostrando el AST de las expresiones en lugar del código. Si la opción es `-c`, se imprime en la salida estándar el programa resultante de aplicar las sugerencias.

Los *lintings* se aplican por función y en el orden en que fueron presentados: computación de constantes, eliminación de chequeos redundantes de booleanos, eliminación de if redundantes, chequeo de lista vacía con *null*, eliminación de concatenación, introducción de la composición, *eta*-reducción y reescritura de recursión en términos de *map*.

Cada *linting* transforma el código al ser aplicado. Dado que cada transformación puede generar la oportunidad de nuevas transformaciones, el proceso se repite hasta que el programa resultante no genere ninguna sugerencia para la función dada.

Por ejemplo, si al programa de la Figura 3 le aplico el *linter* con la opción `-c`, el programa generado es:

```
foo = 3

bar = map null
```

que es producto de las siguientes sugerencias, que se obtienen aplicando el *linter* con la opción `-s`:

```
Función: foo
**Sugerencia para:
if True then 2 else 4
If redundante. Reemplazar por:
2
**Sugerencia para:
1 + 2
Constante. Reemplazar por:
3
--------------------------------------------------------------------------------
Función: bar
```

```
**Sugerencia para:
length ls == 0
Usar null. Reemplazar por:
null ls
**Sugerencia para:
(\ys -> (\ls -> null ls) ys) l : [] ++ bar ls
Eliminar concatenación. Reemplazar por:
(\ys -> (\ls -> null ls) ys) l : bar ls
**Sugerencia para:
\ls -> null ls
Usar eta-reducción. Reemplazar por:
null
**Sugerencia para:
\ys -> null ys
Usar eta-reducción. Reemplazar por:
null
**Sugerencia para:
bar = \xs -> case xs of
    [] -> [];
    (l : ls) -> null l : bar ls
Usar map. Reemplazar por:
bar = map (\l -> null l)
**Sugerencia para:
\l -> null l
Usar eta-reducción. Reemplazar por:
null
--------------------------------------------------------------------------------
```

Notar que por ejemplo en la función `foo` la sugerencia de reemplazar `1 + 2` por `3` surge de una segunda ronda de transformaciones.

También es posible pedirle al *linter* que genere las sugerencias producidas por un sólo *linting*. Esto se hace agregando la opción correspondiente de la siguiente lista: `-lintComputeConstant`, `-lintRedBool`, `-lintRedIfCond`, `-lintRedIfAnd`, `-lintRedIfOr`, `-lintNull`, `-lintAppend`, `-lintComp`, `-lintEta`, `-lintMap`.

Por ejemplo las sugerencias generadas para el programa de la Figura 3 con las opciones `-s -lintEta` son:

```
Función: foo
Sin sugerencias.
--------------------------------------------------------------------------------
Función: bar
**Sugerencia para:
\ys -> (\ls -> length ls == 0) ys
Usar eta-reducción. Reemplazar por:
\ls -> length ls == 0
--------------------------------------------------------------------------------
```

Mientras que las sugerencias generadas con las opciones `-s -lintMap` son:

```
Función: foo
Sin sugerencias.
--------------------------------------------------------------------------------
Función: bar
Sin sugerencias.
--------------------------------------------------------------------------------
```

Notar que no hay sugerencias, dado que al no haberse aplicado las demás transformaciones la función `bar` no tiene la forma requerida por la transformación a `map`.

4 Implementación del Linter

4.1 AST

En el archivo `AST.hs` se definen los tipos que implementan el árbol de sintaxis abstracta (AST) que representa los programas en MiniHaskell. El tipo de datos `Program` representa los programas.

```
data Program = Program [FunDef]
```

Un programa contiene una lista de definiciones de funciones (`FunDef`).

```
data FunDef = FunDef Name Expr
```

Cada función tiene un nombre y un cuerpo, que es una expresión. Las expresiones están modeladas con el tipo de datos `Expr`.

```
data Expr = Var Name
          | Lit Lit
          | Infix Op Expr Expr | App Expr Expr | Lam Name Expr
          | Case Expr Expr (Name, Name, Expr) | If Expr Expr Expr

data Lit = LitInt Integer | LitBool Bool | LitNil

data Op = Add | Sub | Mult | Div
        | Eq | GTh | LTh
        | And | Or
        | Cons | Comp | Append
```

El programa de la Figura 1, por ejemplo, se representa con el árbol:

```
Program [Fundef "add"
                (Lam "x" (Lam "y" (Infix Plus (Var "x")(Var "y"))))
        ]
```

4.2 Lintings

El siguiente tipo, definido en `LintTypes.hs`, representa a los distintos *lintings*:

```
data LintSugg = LintCompCst Expr Expr
              | LintBool    Expr Expr
              | LintRedIf   Expr Expr
              | LintNull    Expr Expr
              | LintAppend  Expr Expr
              | LintComp    Expr Expr
              | LintEta     Expr Expr
              | LintMap     FunDef FunDef
```

Los *lintings* se implementan como funciones que toman un fragmento de AST y retornan un nuevo fragmento resultante de aplicar las sugerencias. Como el resultado de aplicar una regla de *linting* sobre una porción de código puede resultar en múltiples sugerencias, la función devuelve además la lista de las sugerencias que recomienda aplicar, en orden, indicando en cada caso el fragmento sobre la que se aplica y el resultado de aplicarla. Cada sugerencia es de tipo `LintSugg`.

Por lo tanto, el tipo de las funciones de *linting* es:

```
type Linting a = a -> (a, [LintSugg])
```

tal que las funciones de *linting* a nivel de definición de función son de tipo `Linting FunDef` y las funciones a nivel de expresión son de tipo `Linting Expr`.

Todas las funciones que describiremos en el resto de esta sección se encuentran declaradas en el archivo `Lintings.hs`, siendo su implementación la tarea de este laboratorio.

Dado un *linting* a nivel de expresiones, es sencillo extenderlo a funciones, simplemente aplicándolo al cuerpo de la función. Para tratar de manera uniforme todas las transformaciones se provee la siguiente función que implementa dicha conversión:

```
liftToFunc :: Linting Expr  -> Linting FunDef
```

Los *lintings* se pueden componer. Se cuenta con el siguiente operador para encadenar dos *lintings*:

```
(>==>) :: Linting a -> Linting a -> Linting a
```

y la siguiente función que aplica repetidas veces un *linting* (de tipo `Linting a`) sobre un fragmento de AST de tipo `a` hasta que no se generen más cambios en el fragmento:

```
lintRec :: Linting a -> Linting a
```

En la siguiente tabla se resumen todas las funciones que implementan a los distintos *lintings*, indicando qué constructor de `LintSugg` utiliza cada una para retornar las sugerencias. Todas las funciones son de tipo `Linting Expr`, menos `lintMap` que es de tipo `Linting FunDef`.

Linting	Función	Constructor
Computación de constantes	`lintComputeConstant`	`LintCompCst`
Eliminación de chequeos redundantes	`lintRedBool`	`LintBool`

Eliminación de `if` redundantes	`lintRedIfCond`
	`lintRedIfOr`
	`lintRedIfAnd`
		`LintRedIf`
		`LintRedIf`
		`LintRedIf`

Chequeo con *null*	`lintNull`	`LintNull`
Introducción de la composición	`lintComp`	`LintComp`
Eliminación de la concatenación	`lintAppend`	`LintAppend`
eta-reducción	`lintEta`	`LintEta`
Reescritura de recursión con *map*	`lintMap`	`LintMap`

En todos los constructores de `LintSugg` se indica el fragmento del AST sobre el que aplica la sugerencia de transformación y el fragmento de AST resultante de aplicarla.

A manera de ejemplo, el resultado de evaluar la siguiente expresión:

```
lintComputeConstant
   (Infix Add (Infix Add (Var "x") (Lit (LitInt 1)))
              (Infix Add (Lit (LitInt 1))
                         (Infix Mult (Lit (LitInt 2))
                                     (Lit (LitInt 1)))))
```

debe ser:

```
(Infix Add (Infix Add (Var "x") (Lit (LitInt 1))) (Lit (LitInt 3))
, [ LintCompCst (Infix Mult (Lit (LitInt 2)) (Lit (LitInt 1)))
                (Lit (LitInt 2))
  , LintCompCst (Infix Add (Lit (LitInt 1)) (Lit (LitInt 2)))
                (Lit (LitInt 3))
  ]
)
```

5 Archivos

Además de esta letra el obligatorio contiene los siguientes archivos:

`AST.hs` Módulo que contiene el AST (árbol de sintaxis abstracta) que modela al lenguaje.

`Parser.hs` Módulo de parsing.

`PrettyPrinter.hs` Módulo que implementa la generación de código válido del lenguaje a partir de un AST.

`Lintings.hs` Módulo de *linting*.

`LintTypes.hs` Módulo que contiene los tipos que representan a los *lintings* y las sugerencias.

`Linter.hs` Programa Principal.

`ejemplos/ejemplo`*i*`.mhs` Ejemplos de programas MiniHaskell.

`ejemplos/ejemplo`*i*`-linted.mhs` Ejemplos de programas MiniHaskell, luego de aplicado el linter. Son el resultado de llamar al programa principal con opción `-c` sobre los respectivos archivos fuente.

`ejemplos/ejemplo`*i*`-sug` Sugerencias, son las salidas resultado de llamar al programa principal con opción `-s` sobre los respectivos archivos fuente.

6 Se pide

La tarea consiste en modificar el archivo `Lintings.hs` implementando las funciones solicitadas (todas aquellas que aparecen definidas como `undefined`), de manera que el *linter* se comporte como se describe en esta letra y de acuerdo a los casos de prueba. Este es el único archivo que se entregará. Dentro del mismo se pueden definir todas las funciones auxiliares que sean necesarias. No se debe modificar ninguno de los demás archivos, dado que los mismos no serán entregados. Tampoco se debe modificar la signatura de ninguna de las funciones definidas en `Lintings.hs`.