Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación
1er cuatrimestre, 2014

Fecha de entrega: 15 de abril del 2014

Introducción

Este trabajo consiste en modelar lógica de primer orden en Haskell, de modo de poder construir fórmulas y evaluarlas bajo distintas interpretaciones.

Gramática

Cada lenguaje está determinado por un alfabeto de símbolos de funciones $\mathcal{F}$ y otro de símbolos de predicados $\mathcal{P}$. Los términos de un lenguaje se construyen inductivamente así:

- variables (x, y, z, ...)
- aplicación de funciones a otros términos (ej: $f(x)$, $g(y, f(x))$, con $f, g \in \mathcal{F}$)

De forma similar, las fórmulas se construyen del siguiente modo:

- predicados sobre términos (ej: $P(t_1)$, $Q(t_1, t_2)$, con $P, Q \in \mathcal{P}$)
- combinación de fórmulas con conectores lógicos

Estas construcciones han sido modeladas en Haskell con los siguientes tipos de datos:

```
data Termino = Var Nombre
             | Func Nombre [Termino]

data Formula = Pred Nombre [Termino]
             | No Formula
             | Y Formula Formula
             | O Formula Formula
             | Imp Formula Formula
             | A Nombre Formula
             | E Nombre Formula
```

siendo `Nombre` un renombre de `String`. ("A" significa "$\forall$" y "E" significa "$\exists$").

Interpretaciones

En la mayoría de los casos, para determinar la verdad o falsedad de una fórmula será necesario interpretarla en el contexto de un dominio. Por ejemplo, el valor de verdad de

$$(\forall x)\ P(x, f(x))$$

dependerá de cuáles son los posibles valores de x y qué significado se le da a f y P. En general, para poder interpretar una fórmula se necesita definir:

- un dominio, el conjunto de valores sobre los que se predica
- una función del dominio que corresponda a cada símbolo de función
- el conjunto de tuplas que satisfará cada predicado

La definición del tipo `Interpretacion` es la siguiente:

```
data Interpretacion a =
 I {fTerm :: (Nombre->[a]->a), fPred :: (Nombre->[a]->Bool)}
```

Esto es lo mismo que decir:

```
data Interpretacion a = I (Nombre->[a]->a) (Nombre->[a]->Bool),
```

pero se definen automáticamente los proyectores `fTerm` y `fPred`.

Bajo el dominio de los números naturales, una posible interpretación que haría verdadera la fórmula $(\forall x)\ P(x, f(x))$ sería asignar a f la función *sucesor* y a P el predicado *es menor o igual* (aquel que se satisface con todas las listas cuyo primer elemento es menor o igual al segundo). Sin embargo, hay otras interpretaciones bajo las cuales la misma fórmula resulta falsa.

Ejercicios

Ejercicio 1

Decimos que una fórmula es un literal cuando es un predicado o la negación de un predicado. Dar el tipo y definir la función `esLiteral`, que indica si una fórmula es o no un literal. □

Ejercicio 2

Dar el tipo y definir la función `foldTermino` que implemente un esquema de recursión estructural para el tipo `Termino`. Se permite utilizar recursión explícita para definir esta función. □

Ejercicio 3

Dar el tipo y definir la función `foldFormula` que implemente un esquema de recursión estructural para el tipo `Formula`. Se permite utilizar recursión explícita para definir esta función. □

Ejercicio 4

Implementar - usando recursión explícita - la función `recFormula`, que implemente un esquema de recursión primitiva para el tipo `Formula`. Este esquema permite definir los casos recursivos no sólo a base de otros resultados, sino también con los valores (sub-fórmulas) sobre los cuales se calcularon los mismos. Su tipo es:

```
recFormula
  :: (Nombre → [Termino] → b)
     → (Formula → b → b)
     → (Formula → Formula → b → b → b)
     → (Formula → Formula → b → b → b)
     → (Formula → Formula → b → b → b)
     → (Formula → Nombre → b → b)
     → (Formula → Nombre → b → b)
     → Formula
     → b
```

Si bien esta función se puede implementar utilizando el esquema definido por `foldFormula`, se trata de un esquema de recursión y es más claro y natural implementarla usando recursión explícita. □

Ejercicio 5

Implementar la función `show` para el tipo `Termino`, que retorna la representación textual de un término. Asegurarse de que los nombres de las variables se muestren en mayúsculas, y las funciones sin argumentos se vean como constantes.

Por ejemplo:

`Func "f" [Var "x", Var "y", Func "c" []]` debe verse como `f(X,Y,c)`.

Sugerencia: utilizar las funciones `join` y `parentizar` provistas. □

Ejercicio 6

Implementar la función `show` para el tipo `Formula`, que retorna la representación textual de una fórmula. Sólo se deben agregar paréntesis alrededor de una sub-fórmula cuando la misma no sea un literal. Los nombres de las variables deben seguir mostrándose en mayúsculas (asumiremos que no hay variables cuyos nombres sean iguales salvo mayúsculas y minúsculas).

Por ejemplo, la fórmula:

`A "x" (Imp (Pred "p" [Var "x"])(Pred "p" [Var "x"]))`

debe verse como `∀X.(P(X)⊃P(X))`.

El esqueleto del TP provee la lista de caracteres a utilizar para los operadores lógicos. □

Ejercicio 7

Dar el tipo y definir la función `eliminarImplicaciones`, que dada una fórmula retorna una semánticamente equivalente pero sin implicaciones. Recordar que $A \supset B$ es equivalente a $\neg A \vee B$. □

Ejercicio 8

El conjunto de las fórmulas en forma normal negada (FNN) se define inductivamente como:

- Los literales están en FNN.
- Si $\varphi, \psi \in$ FNN, entonces $(\varphi \vee \psi), (\varphi \wedge \psi) \in$ FNN. Por ejemplo, $P(X) \vee \neg P(X)$ está en FNN, pero $\neg(P(X) \wedge \neg P(X))$ no.
- Si $\varphi \in$ FNN, entonces $(\forall x)\varphi, (\exists x)\varphi \in$ FNN. Por ejemplo, $\exists X.\neg P(X)$ está en FNN, pero $\neg(\forall X.P(X))$ no.

En otras palabras, se trata de que las negaciones estén lo más adentro posible. (Notar también que no se admiten implicaciones).

Dar el tipo y definir la función `aFNN`, que dada una fórmula retorna otra semánticamente equivalente pero en forma normal negada. Se sugiere utilizar una función auxiliar para definir el caso de la negación.

Algunos ejemplos de pasaje a forma normal negada:

$$\begin{array}{ll} \neg\neg P(X) & \rightsquigarrow P(X) \\ \neg(Q(X,Y) \wedge R(Z)) & \rightsquigarrow \neg Q(X,Y) \vee \neg R(Z)) \\ \exists Y.(\neg\exists X.(P(X) \supset Q(X,Y))) & \rightsquigarrow \exists Y.(\forall X.(P(X) \wedge \neg Q(X,Y))) \\ \forall X.(\exists Y(P(X) \supset Q(X,Y))) & \rightsquigarrow \forall X.(\exists Y.(\neg P(X) \vee Q(X,Y))) \end{array}$$

El último ejemplo en Haskell:
`aFNN $ A "x" (E "y" (Imp (Pred "p" [Var "x"])(Pred "q" [Var "x", Var "y"])))`
devuelve `∀X.(∃Y.(¬P(X)∨Q(X,Y)))`. □

Ejercicio 9

Dar el tipo y definir la función `fv`, que dada una fórmula retorne el conjunto de variables libres de la misma. Una variable se considera libre en una fórmula si aparece al menos una vez sin estar al alcance de ningún cuantificador.

Puede representar el conjunto resultado como una lista, pero la misma no deberá contener elementos repetidos. □

Ejercicio 10

Definir la función `evaluar`, que retorne el valor que representa un término en un determinado dominio, dadas una asignación de variables y la interpretación de los símbolos de función como funciones del dominio.

Por ejemplo, si tenemos la siguiente función de interpretación para símbolos de función:

```
fTerminos :: (Nombre → [a] → a)
fTerminos nombreF | nombreF == "0" = const 0
                  | nombreF == "suc" = \xs → head xs + 1
                  | nombreF == "suma" = sum
```

y usamos la asignación que da 0 a `X` y 1 a `Y`, entonces al evaluar la fórmula
`Func "suma" [Func "suc" [Var "X"], Var "Y"]` obtendremos el valor `2`.

En el `.hs` encontrarán un ejemplo de interpretación y uno de asignación. Pueden utilizarlos en sus pruebas, pero recomendamos que agreguen también sus propios ejemplos. □

Ejercicio 11

Implementar y dar el tipo de la función `actualizarAsignacion`, que dados un nombre de variable, un valor y una asignación, genera una nueva asignación igual a la original salvo por el valor de la variable indicada, que pasa a ser el que se recibió. □

Ejercicio 12

Definir la función `vale`, que retorne el valor de verdad de una fórmula en el contexto de un domino, una interpretación y una asignación de variables. Notar que el dominio se debe definir explícitamente y no se infiere enumerando los valores del tipo de la interpretación.

Si bien esta función se puede definir utilizando los esquemas de recursión vistos, para este ejercicio se permite el uso de recursión explícita. Sin embargo, se debe explicar qué hace que resulte inconveniente el uso de `fold`.

Por ejemplo, en la interpretación habitual sobre naturales con dominio de 0 a 10, las fórmulas `∀X.(∃Y.¬mayor(X,Y))` y `∀X.(∀Y.(mayor(X,Y)⊃menor(Y,X)))` valen, pero `∀X.(∃Y.mayor(X,Y))` no vale (porque el dominio es acotado).

□

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante Se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en `http://www.haskell.org/onlinereport/haskell2010`.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en `http://learnyouahaskell.com/chapters`.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en `http://book.realworldhaskell.org/read`.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en `http://www.haskell.org/hoogle`.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en `http://holumbus.fh-wedel.de/hayoo/hayoo.html`.