FUNDAMENTOS DE COMPUTACIÓN
Repartido: Igualdad y Computación

1. Sobre la Igualdad: Demostraciones ecuacionales y Computación

Hemos probado propiedades de Booleanos, Naturales y otros tipos finitos.
Muchas de estas propiedades han sido igualdades entre expresiones que involucran funciones que hemos definido.
La noción de igualdad se da ordinariamente como obvia en Matemática. Es una **relación de equivalencia** (o sea, **reflexiva**, **simétrica** y **transitiva**), que además es una **congruencia** (podemos reemplazar cualquier cosa por otra igual en cualquier contexto).

Nuestro lenguaje posee el juicio $e_1 = e_2$, y las igualdades concretas que pueden ser demostradas se deberán derivar necesariamente de las reglas que hayamos introducido en el lenguaje.
Algunos ejemplos de estas reglas son:
- las **ecuaciones** que definen la sustitución,
- las reglas α y **β**,
- las **definiciones** de constantes, tales como not, (&&), par, (+), entre otras,
- las reglas de las expresiones **case** asociadas cada tipo.

Un caso particular de demostración de una igualdad es cualquier evaluación de una expresión, por ejemplo en el primer ejemplo de este repartido, cuando demostramos que par (doble 0) = True.
Las evaluaciones son también llamadas **computaciones o reducciones**.
Su característica especial es que consisten en la **eliminación sistemática de toda expresión** que haya sido explícitamente **definida** en el lenguaje, **sustituyéndola por su definición**.
De esta forma, se busca **reducir la expresión original a un <u>valor</u>**.

Como ya se ha explicado, un **valor** es **una expresión que debe entenderse directamente**, y no requiere ningún tipo de evaluación.
Ejemplos de valores son los constructores de los tipos finitos (False, True, Norte, Sur....) así como los naturales 0, S 0, S(S 0),.. y también expresiones como λx.x.
En los ejemplos que hemos visto hasta ahora, las **expresiones que se reducen** son:
- los **β-redexes** : (λx.e) a, que se reducen aplicando la regla β y la sustitución,
- las **constantes definidas**, que se reducen simplemente aplicando la definición, es decir, sustituyendo la constante por la expresión que la define,
- **las expresiones case aplicadas a un valor**, que se evalúan utilizando las ecuaciones que las definen.

Todas estas expresiones, que deben ser eliminadas si se quiere alcanzar un valor, son llamadas colectivamente **redexes** (expresiones reducibles).

Entonces una **computación o reducción** es una demostración de una igualdad, pero con dos características particulares:

1. Lo que se demuestra es que una expresión es igual a un valor (no a una expresión en general)
2. Los redexes siempre se eliminan (nunca se introducen).

La segunda condición equivale a decir que **las ecuaciones son siempre usadas de izquierda a derecha** o, en términos técnicos, como **reglas de reescritura**.
Los resultados intermedios se encadenan utilizando **la propiedad transitiva de la igualdad**, lo cual permite finalmente igualar la expresión original al valor obtenido como resultado.

En resumen, **una computación utiliza solo reescritura y transitividad**.

Claramente, ese no es el caso general. Ya hemos hecho varias demostraciones donde utilizamos también **la simetría de la igualdad** para justificar lo que llamamos **igualdad por reducción a la misma expresión.**

Los lenguajes de programación proveen en general un procedimiento mecánico (automático) de computación. Ese procedimiento es lo que permite ejecutar los programas una vez provistos con datos de entrada adecuados.

Pero volviendo a las demostraciones, ¿qué igualdades podemos demostrar?
La respuesta es, en principio: exactamente las que se deriven de las ecuaciones estipuladas.
Puede ocurrir que la intuición sea un poco más precipitada que lo que este principio marca.
Por ejemplo, consideremos las funciones:

```
nid :: N → N
nid = λn. case n of { 0 → 0 ; S x → S x }

id :: N → N
id =  λn. n
```

Podríamos quizás apresurarnos al afirmar que nid = id.
Pero la ecuación correspondiente, λn.case n of { 0 → 0 ; S x → S x } = λn. n no puede ser derivada de las definiciones dadas en el lenguaje, ya que no hay absolutamente ninguna regla que permita igualar estas dos abstracciones...

¿En dónde radica la discrepancia?
Bueno, la intuición de quien busca igualar esas dos funciones consiste en el hecho de que ellas “funcionan igual”, es decir, retornan los mismos resultados para cualquier entrada.
Esto equivale a identificar la noción de **función** con **comportamiento de entrada/salida**.

Técnicamente, esto se denomina el concepto **extensional** de función, ya que implica identificar una función es el conjunto (extensión) de las parejas (entrada, salida) definida por ella.

Pero no es este concepto es el que está implementado por la igualdad de nuestro lenguaje. En otras palabras, en nuestro lenguaje esas dos funciones son diferentes.

¿Es esto un defecto del lenguaje? No necesariamente. Para convencerse de esto, observemos primero que nuestras funciones no son, de hecho, solamente un cierto comportamiento de entrada/salida, sino también, y fundamentalmente, **expresiones de algoritmos**.

Como tales, las dos funciones precedentes son claramente diferentes:
- la función λn. n computa la identidad simplemente copiando su argumento en la salida, mientras que
- λn.case n of { 0 → 0 ; S x → S x } primero evalúa el argumento y luego, dependiendo del resultado, retorna su valor.

Si siempre identificamos a las funciones por su extensión, no seríamos capaces de distinguirlas por la manera en que están construidas.

Esta última capacidad es fundamental para el **estudio de algoritmos**, por ejemplo para establecer medidas y comparaciones del costo de la computación en términos de eficiencia o uso de recursos.

La denominación para este concepto de función, donde lo relevante es **la manera en que la función es definida** se denomina **definicional**.

2. Expresiones indefinidas

En las demostraciones de igualdades que hemos hecho hasta el momento, la idea usada es simple: consiste en reducir cada expresión a su valor y comparar los resultados.
Este parece ser un buen método para demostrar igualdades.
Pero estamos asumiendo algo que no es necesariamente cierto: que toda expresión tiene valor...

En efecto, cuando introdujimos definiciones en el lenguaje, aceptamos cierto tipo de definiciones que son potencialmente peligrosas: *las definiciones recursivas*, o sea aquellas en las que la misma constante que se define aparece a la derecha del signo =.

Algunas definiciones recursivas pueden no tener sentido, pues pueden producir computaciones que no terminan. Un ejemplo trivialmente erróneo es la siguiente definición:

```
indef :: N
indef = indef
```

Primero que nada, veamos que esta es una definición totalmente aceptable según nuestras reglas, ya que:
- en la primera línea introducimos una nueva constante, de tipo N, y
- en la segunda línea damos su definición, escribiendo a la derecha del símbolo = una expresión del tipo N (ella misma).

Pero cuando queremos calcular el valor de indef, tendremos que:

```
 indef
 = (def. indef)
indef
 = (def. indef)
indef
 = (def. indef)
 .....
```

¡Entramos en una computación infinita!

Por supuesto que no toda definición recursiva es fallida como la precedente.

Hemos visto ya muchas definiciones recursivas que tienen sentido y son además extremadamente útiles. Llamaremos a estas definiciones **bien fundadas**.

También dijimos que es imposible dar una regla formal para la formulación de definiciones recursivas que admita exactamente las que sean bien fundadas, de modo que, o bien nos resignamos a no poder escribir ciertas definiciones bien fundadas (que eventualmente podrán ser útiles), o bien admitimos cualquier forma de recursión, incluyendo las que no tienen sentido (como la del ejemplo de arriba).

En esta presentación, al igual que en todos los lenguajes de programación, optamos por la segunda alternativa.

De todos modos, como ahora sabemos explícitamente que existen expresiones que pueden no tener valor, deberíamos distinguir entre expresión y objeto de los diversos tipos.

Más precisamente:

- los **objetos son las entidades conceptuales** (e.g. funciones, booleanos, números), mientras que

- las **expresiones son entidades sintácticas** (secuencias de símbolos).

Ahora bien:
- **Un objeto siempre es denotado (representado) por alguna expresión**.

- Para **conocer** cuál es el objeto denotado por una expresión se debe usar **definiciones**.

Por ejemplo, la expresión **True && (not False || True)** denota un booleano, y para saber qué booleano es éste, se deben aplicar las definiciones de las operaciones de booleanos a fin de evaluar la expresión.
Por medio de este procedimiento se llegará a un valor (**True**, en este caso).

Como ya dijimos, un **valor** es una expresión que no requiere, ni puede, ser evaluada, es decir, **debe ser entendida tal como se presenta.**
En otras palabras, **cada valor denota directamente a un objeto de cierto tipo**.

Resumiendo, todo **objeto** es denotado por alguna **expresión** y para llegar de una expresión al objeto representado, la expresión se **evalúa** (computa) aplicando una serie de **definiciones** dadas explícitamente en el lenguaje.
Cuando se obtiene el **valor** de la expresión se llega a **conocer** directamente al objeto en cuestión.

Ahora bien, desafortunadamente, nos vemos forzados, por motivos prácticos, a admitir **definiciones recursivas irrestrictas** que traen con ellas el fenómeno de **expresiones que no denotan objeto alguno**.
Llamamos a estas **expresiones indefinidas**, e introduciremos la notacion ⊥ para representar a una expresión indefinida (que no tiene valor)

Ahora, si revisamos la definición de **indef** dada arriba, podremos observar que la misma puede hacerse para un tipo genérico t. O sea, podríamos haber definido:

```
indef :: Bool
indef = indef

indef :: PCard
indef = indef
```

En realidad, **indef** puede tener cualquier tipo, así que la definimos como una expresión de tipo genérico t:

```
indef :: t
indef = indef
```

Como consecuencia de esto: **¡Tenemos expresiones indefinidas en todos los tipos!**

Y entonces, en nuestras demostraciones, deberemos considerar también los casos de expresiones indefinidas, cosa que no hemos hecho hasta ahora.

Por ejemplo, en las expresiones **case**, si el discriminante está indefinido, toda la expresión case lo estará, o sea, para cada tipo nuevo que definamos deberemos agregar una nueva ecuación para sus expresiones case de la forma:

case ⊥ of {... ... ...} = ⊥

Esto tiene consecuencias, ya que algunas igualdades demostradas no se cumplen para expresiones indefinidas.
Por ejemplo, en los booleanos no se cumple la conmutatividad de los conectivos binarios, ya que:

⊥ && False = case ⊥ of { False → False ; True → False } = ⊥
pero
False && ⊥ = case False of { False → False ; True → False } = False

Y en los naturales, tenemos que no se cumple que (∀ n :: N) par (doble n) = True, pues:

par (doble ⊥) = par (case ⊥ of { 0 → 0 ; S x → S(S(doble x)) } = par ⊥ = ⊥ ≠ True.

Pero en cambio otras propiedades que ya demostramos sí se cumplen para expresiones indefinidas, como por ejemplo en Bool (∀ b :: Bool) not (not b) = b,, pues:

not (not ⊥) = not (case ⊥ of { False → True ; True → False }) = not ⊥ = ⊥

Y en los naturales se cumple que (∀ n :: N) n + 0 = 0 + n pues:

⊥ + 0 = ⊥ y **0 + ⊥ = ⊥**
(ejercicio: hacer las cuentas).

Entonces, introduciremos la notación **e⇓** para indicar que la expresión **e** está **definida** (o sea, que **tiene valor**).
Y algunos resultados valdrán solamente para expresiones que están definidas, mientras que otros valdrán para todas las expresiones.

Así, por ejemplo enunciamos la conmutatividad de && como:

$Lema_{Conm\&\&}$: (∀b1⇓ :: Bool)(∀b2⇓ :: Bool) b1 && b2 = b2 && b1

o el lema de par como

$Lema_{pardoble}$: (∀n⇓ :: N) par (doble n) = True