Introducción a los tipos dependientes y a la programación verificada en Agda

Marcelo Lynch - Instituto Tecnológico de Buenos Aires

Índice

1. Introducción

1.1. Objetivos de este trabajo

Este informe se entrega a modo de examen final para la materia *Programación funcional* del Instituto Tecnológico de Buenos Aires. Los objetivos de este trabajo son:

- Dar una breve introducción a la verificación de software y sus técnicas.
- Dar una introducción al lenguaje de programación y asistente de demostraciones *Agda* y sus distintas formas de verificar programas.
- Introducir el paradigma de las *proposiciones como tipos*, con su justificación teórica, y su papel en cómo se realizan demostraciones en Agda.
- Describir el concepto de *tipo dependiente* (ausente en Haskell pero central en Agda), qué rol juega en la teoría de tipos y en Agda.

En lo que sigue se presume que el lector está familiarizado con el lenguaje de programación Haskell.

2. Programación verificada.

2.1. Verificación de software

En un mundo en el que el software comienza a permear toda la actividad humana, y las personas delegan cada vez más responsabilidades a sistemas automáticos, se vuelve cada vez más importante garantizar que el software *hace lo que queremos que haga*. Podemos encontrar sin esfuerzo varios ejemplos en los que un error de software o *bug* podría directamente costar vidas humanas:

- Aviónica
- Controladores en centrales nucleares
- Software en equipamiento médico (para un ejemplo notable consultar [LT93])

Aún cuando el *bug* no resulte en la muerte de nadie, puede tener un impacto financiero altísimo [Har03].

En todo caso, es cada vez más clara la importancia de que los programas y algoritmos sean *correctos* de acuerdo a alguna *especificación*. Al mismo tiempo que se vuelven ubicuos, los sistemas de software se vuelven cada vez más complejos, lo que implica que su correcta implementación y ausencia de errores se vuelve no trivial. Esto lleva a la necesidad de desarrollar herramientas, técnicas y teorías que asistan al programador en esa tarea. Esta tarea se enmarca dentro de la rama de las ciencias de la computación conocida como *ingeniería de software*, y el acto de confirmar esta corrección es la *verificación*.

2.2. Técnicas de verificación

A continuación repasaremos brevemente distintas aproximaciones a la verificación de software. No se pretende hacer un análisis exhaustivo sino presentar el panorama a grandes rasgos para tener un marco de referencia para analizar cómo se realiza esto en el mundo funcional y en particular en Agda.

2.2.1. Testing y revisión de pares

En la práctica los métodos más utilizados para verificar software son la revisión de pares y el testing. La *revisión de pares* es simplemente la aceptación de otro programador de que un código es correcto.

El *testing* es una forma de comprobar el comportamiento de un programa en tiempo de ejecución. Una *suite de tests* comprueba que la ejecución de un programa, porción de programa o componente tiene el comportamiento adecuado ante ciertos escenarios (usualmente dados por entradas o *inputs* al programa) establecidos por el programador.

El *testing* es útil para encontrar bugs y para comprobar que el comportamiento de un sistema se mantiene consistente en el tiempo (ante cambios), pero no *demuestra* la corrección pues solo contempla una sección limitada de entradas. Naturalmente, la revisión por pares tiene la misma limitación.

2.2.2. Análisis estático y dinámico para encontrar bugs

Además del *testing* podemos encontrar otras técnicas que no implican una demostración de corrección pero que ayudan a encontrar errores. Estos métodos pueden basarse en analizar el código estáticamente (sin ejecutarlo) o dinámicamente (ejecutándolo con diversas entradas, como hacen los tests). No nos detendremos a describir estas técnicas, pero podemos nombrar como ejemplos paradigmáticos a la *ejecución simbólica* y al *fuzzing*.

2.2.3. Verificación formal

La necesidad de realmente *demostrar la corrección* nos lleva a los llamados *métodos formales*. Podemos describir a los métodos formales como "la matemática aplicada para modelar y analizar sistemas de software" ([BK08], cap. 1)

Esta verificación se hace proveyendo una demostración formal de un modelo matemático del sistema. Es interesante notar que la correspondencia entre el modelo matemático y el sistema real se presume por construcción: debe establecerse externamente (así como se provee una especificación).

Una aproximación es el *model checking*, donde se provee un modelo matemático que puede explorarse exhaustivamente para verificar las propiedades. En general se

intentan modelar y explorar los distintos estados del sistema y las transiciones entre estos estados.

Otra aproximación, que es la que más nos interesa en este trabajo, es la *verificación deductiva*. Para utilizar estas técnicas es preciso dotar al sistema de una semántica que pueda expresarse en fórmulas lógicas. También se expresan en este formato las especificaciones que debe seguir el sistema. Con esto se acude a *demostradores de teoremas* intentando deducir la verdad de las fórmulas de la especificación a partir de la semántica del sistema.

2.3. Demostradores de teoremas

Como ya mencionamos, la verificación deductiva hace uso de herramientas para la demostración de los teoremas. El nivel de automaticidad de estas herramientas nos deja categorizarlas en *demostradores automáticos*, *asistentes de demostraciones* (o *proof assistants*) y *verificadores de demostraciones* (o *proof verification tools*).

La diferencia está en el nivel de involucramiento del usuario en la demostración: un demostrador automático, como indica el nombre, es completamente automático. Un asistente de demostración, en cambio, requiere el ingreso de *pistas* por parte del usuario: por esto también se llaman *asistentes interactivo*, pues incluyen una interfaz en la que el usuario guía de alguna manera la búsqueda de la demostración. Se habla en estos casos de una "colaboración humano-máquina". Entre los asistentes de demostración notables encontramos a *Coq*, *Isabelle* y el propio Agda, que estudiaremos a continuación.

Finalmente, los verificadores simplemente verifican la corrección de una demostración provista, y no realizan trabajo de deducción.

2.4. La verificación en el mundo Haskell

Existen herramientas para hacer testing de programas Haskell. HUnit [HUn], inspirada en el framework de testing para Java JUnit, permite programar casos de test ad-hoc. QuickCheck es una herramienta de generación automática de tests aleatorios dada una especificación [CH11].

La naturaleza funcional de Haskell lo hace un buen candidato para la verificación deductiva: tiene una semántica denotacional bien definida, es apropiado para razonar ecuacionalmente y su biblioteca estándar promueve el uso de estructuras matemáticas con propiedades algebraicas fuertes y establecidas. Sin embargo, los programadores Haskell suelen razonar informalmente sobre sus programas, haciendo usualmente demostraciones en papel [SBRW17]. Las mayores dificultades de la verificación de programas Haskell es su posibilidad de no-terminación, y también las porciones que incluyen efectos secundarios.

Aproximaciones a la verificación formal incluyen herramientas como `hs-to-coq`, que permite traducir programas Haskell a programas en Coq preservando su semántica y sobre los que se puede razonar en esa herramienta [SBRW17], y Haskabelle, que

hace lo propio, traduciendo Haskell a teorías de Isabelle y también programas de Isabelle a Haskell, con garantías de corrección parcial respecto a la especificación provista en Isabelle [Haf10].

Finalmente, avances en el propio lenguaje Haskell, en particular extensiones del lenguaje sobre el sistema de tipos, permiten expresar invariantes de los programas usando tipos [LM14]. Volveremos a comentar esto sobre el final de este trabajo.

3. Introducción a Agda

En este trabajo estudiaremos el lenguaje de programación *Agda* como herramienta tanto para programar (es un lenguaje funcional) como para verificar esos programas (mediante su poderoso sistema de tipos).

La versión actual de Agda (*Agda 2*) fue desarrollada mayormente por Ulf Norell para su tesis de doctorado ([Nor07]), a partir de un proyecto existente de la Universidad Tecnológica Chalmers en Gotemburgo, Suecia. Chalmers tiene una larga historia de desarrollo de asistentes de demostraciones basadas en la teoría de tipos de Martin-Löf (comenzando en la década del 80): Agda 2 constituye su último desarrollo en esta área. Si bien constituye un asistente de demostración, Agda también pretende ser un lenguaje de programación de propósito general (para programar en sí mismo y no solo para demostrar).

La notación de Agda esta muy inspirada en la de Haskell, y así el estilo de programación es similar: la definición de tipos de datos se hace de forma análoga a Haskell, y la definición de funciones suele ser por *pattern matching*. Sin embargo, la teoría subyacente a Haskell es distinta a la de Agda: mientras que Haskell está basada en el *System F* de Girard (lambda cálculo tipado con polimorfismo paramétrico), Agda es una extensión de la teoría de tipos de Martin-Löf, donde el *tipo dependiente* es el principal objeto teórico, y se utilizan los tipos tanto para especificar como para programar [NPS90]. Otra particularidad que tienen los programas de Agda (a diferencia de Haskell) es que **siempre terminan**: no puedo definir funciones parciales (como en Haskell), o, lo que es lo mismo, toda función es estricta. Esto es una propiedad deseable en un sistema de demostración de teoremas (de lo contrario aparecen inconsistencias), pero nos hace perder expresividad: Agda no es un lenguaje Turing-completo.

Antes de adentrarnos de lleno con las particularidades de Agda, entonces, revisaremos la teoría que tiene detrás, qué justifican que podamos demostrar teoremas con el sistema de tipos.

4. La teoría de tipos de Martin-Löf

Se llama *teoría de tipos* a una serie de sistemas formales que sirven como alternativa a la teoría de conjuntos como fundamento formal de la matemática.

Los matemáticos y científicos de la computación trabajan siempre con *construcciones*: objetos matemáticos y objetos computacionales. Aún si imagina el fundamento en teoría de conjuntos, el científico categoriza a los objetos con los que trabaja, asociándolos a un *tipo*. Las teorías de tipos hacen explícita esta asociación, tratando a los distintos tipos como "ciudadanos de primera clase". Un programador que conozca un lenguaje tipado está naturalmente familiarizado con este concepto: los sistemas de tipos en general se ven fundamentados en alguna teoría de tipos.

La teoría de tipos de Martin-Löf fue presentada por Per Martin-Löf [MLS84] como un fundamento matemático intuicionista (constructivista): por esto es también conocida como *teoría de tipos intuicionista*. Los vínculos de la teoría con la lógica constructivista se harán más claros en la sección siguiente.

La descripción de la teoría y sus elementos se hace a partir de *juicios*, es decir afirmaciones en el lenguaje metalógico, en lugar de definiciones. Así, por ejemplo, en lugar de *definir* el concepto de tipo veremos en qué condiciones se puede decir (es decir, emitir un juicio, o "explicar") que algo *es un tipo*.

4.1. Tipos

El concepto fundamental en la teoría de tipos es, naturalmente, el de *tipo*. Un tipo se explica diciendo cuándo un objeto es de ese tipo y también qué significa que dos objetos del tipo sean iguales. Es decir, podemos emitir el juicio "A es un tipo" cuando conocemos las condiciones de pertenencia al tipo y además una relación de equivalencia que signifique la igualdad de objetos del tipo.

La pertenencia de un objeto a al tipo A se nota $a \in A$ o $a : A$.

Decimos que dos tipos A y B son iguales cuando todos los objetos idénticos de A son objetos idénticos en B y viceversa.

Decimos que B es una *familia de tipos* indexada por A si B es una asignación que para cada $a \in A$ asigna un tipo $B(a)$.

4.2. Términos y objetos

Describiendo a los tipos nombramos a los *objetos* que tienen ese tipo. En rigor la metalógica de la teoría de tipos habla de *términos*. Así, podemos considerar a $2+2$ y 4 como dos términos distintos del tipo $\mathbb{N}$, que además son iguales en $\mathbb{N}$. Existe una noción de cómputo integrada a a la teoría que permite identificar a estos dos términos en $\mathbb{N}$, pues con las reglas de este cómputo podemos reducir $2+2$ a 4 (decimos que $2+2$ reduce a 4): podríamos pensar que los dos términos, refieren al mismo objeto (el "cuatro"), o bien, considerando la noción de cómputo y reducciones, que reducen a una misma forma canónica (4): este concepto de igualdad se llama *igualdad definicional* y se establece a nivel metateórico (al nivel de los juicios sobre la teoría).

Como este trabajo no pretende ser riguroso en la metateoría sino solo presentar a nivel intuitivo la teoría de tipos, en esta sección hablaremos indistintamente de

"objetos", pero valdrá la pena tener esto en cuenta más adelante cuando hablemos de la igualdad en Agda (y volveremos al ejemplo de $2+2$ y 4) así que lo mencionamos aquí.

4.3. El tipo *Set*

Set es un tipo. En la teoría clásica de Martin-Löf sus elementos son conjuntos definidos inductivamente. Dos elementos de ***Set*** son iguales si tienen los mismos elementos (como conjuntos). En Agda, el tipo ***Set*** es un tipo que contiene a los demás tipos, lo que en la teoría de Martin-Löf podría llamarse el tipo ***Type***. En adelante usaremos la noción de Agda.

4.4. Construyendo nuevos tipos

La construcción de nuevos *tipos de datos* a partir de otros ya existentes es ubicua en computación: la teoría de tipos tiene reglas "constructivas" donde dado tipos existentes podemos construir otros nuevos. A continuación, algunos ejemplos de tipos que pueden construirse dentro de esta teoría:

4.4.1. Tipos funcionales

Si A y B son tipos podemos introducir el tipo $A \to B$, el espacio de funciones de A a B. Un elemento $f \in A \to B$ se puede *aplicar* a cualquier $a \in A$, y tenemos $f(b) \in B$.

4.4.2. Pares ordenados

Si A y B son tipos podemos introducir el tipo $A \times B$ de pares ordenados: la primera componente tiene elementos de A y la segunda elementos de B. Los tipos de pares ordenados vienen equipados con proyecciones

$$\pi_1 : A \times B \to A$$

$$\pi_2 : A \times B \to B$$

tales que $\pi_1((a, b)) = a$ y $\pi_2((a, b)) = b$.

4.4.3. El tipo suma

Si A y B son tipos podemos introducir el tipo $A + B$, la suma disjunta de A y B. Un elemento de este tipo será o bien un elemento de A o uno de B, junto con una indicación de si provino de A o de B[1].

[1] Pensando en conjuntos, el tipo A + B podría ser $A \times \{0\} \cup B \times \{1\}$: la segunda componente indica el "origen" del valor. En Haskell existe algo análogo en la forma del tipo `Either a b = Left a | Right b`, donde podemos saber el "origen" por pattern matching en los constructores `Left` y `Right`

4.5. Tipos dependientes

Una herramienta poderosa de esta teoría de tipos (y del sistema de tipos de Agda y otros lenguajes basados en este tipo de teorías) son los *tipos dependientes*, en los que la definición dependen de un *valor*.

4.5.1. Tipos Π: funciones dependientes

Si B es una familia de tipos sobre A, existe el llamado producto dependiente, tipo de funciones dependientes o *tipo pi*:

$$\prod_{a \in A} B(a)$$

Este tipo contiene funciones con dominio (o entrada) en A pero cuyo codominio (o tipo de salida) depende del valor en la que se aplica la función.

Por ejemplo: si llamamos $VecN(n)$ al conjunto de listas de n elementos naturales, podemos considerar una función f que aplicada en un numero natural n resulta en una lista de n ceros. Así:

$$f \in \prod_{n \in \mathbb{N}} VecN(n)$$

Notemos que cuando B es una asignación constante, este tipo corresponde al tipo de funciones $A \to B$ antes mencionado (donde el codominio no depende del valor de entrada).

4.5.2. Tipos Σ: pares dependientes

Los elementos de los tipos Σ son pares ordenados donde el tipo de la segunda componente depende del valor en la primera componente. Esto es, si B es una familia de tipos sobre A, existe el tipo de pares dependientes, o *tipo sigma*

$$\sum_{a \in A} B(a)$$

.

Cuando B es una asignación constante, este tipo corresponde al tipo de pares ordenados $A \times B$.

4.5.3. El tipo igualdad

Dados dos términos x, y puede construirse el tipo igualdad $x \equiv y$.

Existe un único constructor `refl` para cada tipo A que dado un objeto de A devuelve un valor de $a \equiv a$:

$$\texttt{refl} \in \prod_{a \in A} a \equiv a$$

Notemos que `refl` es la única forma de construir un valor del tipo igualdad: esto significa que si bien podemos hablar de un tipo $x \equiv y$ para cualesquiera dos términos x, y, el tipo $x \equiv y$ solo estará habitado si x es igual a y.

¿Para qué querríamos un tipo que represente la igualdad de un valor consigo mismo? Para entender esto, y cómo utilizamos esta teoría de tipos para demostrar teoremas, debemos explorar un resultado central, el isomorfismo de Curry-Howard, que establece un paralelismo entre los tipos de la teoría de tipos y las proposiciones de la lógica.

5. Proposiciones como tipos

5.1. Lógica intuicionista

La lógica tiene distintas versiones: entre las varias distinciones podemos reconocer una diferencia entre la *lógica clásica* y la *lógica intuicionista*. En la lógica clásica vale el principio del tercero excluido: "o bien A es verdadero, o bien A es falso". Esto nos deja por ejemplo demostrar $A \vee B$ con un argumento como el que sigue:

> "A es o bien verdadero o falso (por tercero excluido). Si A es falso, entonces B es verdadero. Entonces $A \vee B$ es verdadero"

La lógica intuicionista rechaza el principio del tercero excluido y con ello este tipo de argumentos: exige una demostración concreta o bien de que A es verdadero o de que B es verdadero para concluir que $A \vee B$ es verdadero: en otras palabras, con la demostración de $A \vee B$ debemos saber *cuál de los dos vale*. La lógica intuicionista es *constructivista*: para demostrar la existencia de un elemento que satisface una propiedad hay que construirlo, exhibirlo.

5.2. La correspondencia de Curry-Howard

El concepto de *proposiciones como tipos*, o *correspondencia de Curry-Howard* es la relación directa que existe entre las fórmulas de la lógica (proposiciones) con los tipos en la teoría de tipos. La correspondencia es más profunda que una mera biyección entre fórmulas y tipos sino un verdadero isomorfismo: también vincula la noción de *demostración* con la de un *programa*: así, dar un programa de cierto tipo (o lo que es lo mismo, presentar un valor de ese tipo) equivale a demostrar una proposición.

La tabla 1 resume la correspondencia. La última fila de la tabla es otra forma de decir que las fórmulas verdaderas van a corresponder a tipos habitados (no vacíos), mientras que una fórmula falsa se asocia a un tipo sin elementos. En la práctica dar

Lógica	***Teoría de tipos***
Proposición, fórmula	Tipo
Demostración	Programa
Evidencia	Habitante de un tipo

Tabla 1: Correspondencia de Curry-Howard

un programa de un tipo es lo mismo que dar el habitante del tipo, luego usaremos "evidencia" y "demostración" de manera intercambiable.

Aquí ya vemos como la correspondencia se da en el marco de la lógica intuicionista (y de hecho la teoría de Martin-Löf, como ya mencionamos, surge como una formalización intuicionista): la *evidencia* la da un habitante concreto: una demostración se da con un *programa*, una verdadera construcción de la evidencia (el programa retorna un habitante de un tipo, es decir, demuestra la proposición que corresponde a ese tipo).

5.3. Fórmulas y tipos

$A \wedge B$ corresponde a $A \times B$:

En efecto, dar una demostración de $A \wedge B$ es dar una demostración de A y una de B: dar un elemento del producto cartesiano $A \times B$ significa dar un elemento de A y uno de B (en el par ordenado), es decir una demostración de A (pensado como proposición) y una de B.

$A \vee B$ corresponde a $A + B$

Como ya mencionamos en la discusión de lógica clásica versus intuicionista, para demostrar $A \vee B$ debemos proveer explícitamente una demostración de A o una de B, indicando cuál se está demostrando. Dar un elemento del tipo suma de A y B es precisamente esto: un elemento de A o uno de B con la "indicación" de cuál de los dos tipos tiene a ese elemento.

$A \Rightarrow B$ corresponde a $A \to B$

Una demostración de la implicación $A \Rightarrow B$ consiste en un procedimiento que dada una demostración de A obtiene una demostración de B. Similarmente, un valor del tipo $A \to B$ es una función, que aplicada a un valor de A (es decir, una demostración de A) devuelve un valor de tipo B (es decir, una demostración de B).

$\forall x : A.\ B(x)$ corresponde a $\prod_{x \in A} B(x)$

Contar con una demostración constructivista de $\forall x : A.\ B(x)$ implica que para cualquier $x : A$ podemos construir una demostración de la proposición $B(x)$. Pre-

cisamente, un elemento del tipo $\prod_{x \in A} B(x)$ es una función dependiente que para cada $x : A$ devuelve un elemento (demostración) de $B(x)$.

$\exists x : A.\ B(x)$ **corresponde a** $\sum_{x \in A} B(x)$

Contar con una demostración constructivista de $\exists x : A.\ B(x)$ implica que podemos mostrar un $x : A$ junto con la demostración de que $B(x)$. Precisamente, mostrar un elemento del tipo $\sum_{x \in A} B(x)$ es mostrar un par ordenado con un elemento $x : A$ en la primera componente y un elemento de $B(x)$ en la segunda, es decir, la demostración de $B(x)$.

5.4. Conclusiones

Hemos repasado la manera en la que corresponden los tipos y las proposiciones[2]. También vimos el papel fundamental que tienen los tipos dependientes en la expresividad de esta correspondencia: sin ellos no tendríamos cuantificadores.

Con esto podemos regresar a la pregunta que nos hicimos al introducir el tipo igualdad: nos preguntamos allí "¿para qué lo queremos?": una respuesta podría ser: "no importa para qué lo queremos, el tipo $x \equiv x$ existe por existir la proposición *x es igual a x*".

Sin embargo, ya estamos en condiciones de observar su utilidad: si podemos encontrar un elemento del tipo $x \equiv y$, habremos demostrado que x es igual a y. Podemos imaginar ahora cómo se demuestran las propiedades de los programas en un lenguaje con tipos dependientes como Agda: **simplemente debemos exhibir un elemento que habite el tipo que corresponde a la proposición que queremos demostrar**.

Es decir, para demostrar una propiedad como

$$\forall n, m : \mathbb{N}.\ n + m = m + n$$

basta con construir una función del tipo

$$\prod_{n \in \mathbb{N}} \prod_{m \in \mathbb{N}} n + m \equiv m + n$$

Habiendo introducido el fundamento teórico que subyace a Agda y a sus demostraciones, podemos finalmente adentrarnos en el lenguaje y en cómo *programamos verificadamente* en la práctica.

[2]Para una exposición más extensa del tema, que profundiza sobre más aspectos de la correspondencia y además incluye notas históricas, referimos al lector al artículo de Wadler [Wad15]

6. Programando en Agda

En las próximas secciones escribiremos programas en Agda, enfocándonos en el aspecto de la verificación utilizando el sistema de tipos. La idea de este informe es dar algunos ejemplos que nos dejen ver las distintas formas de verificar y nos permitan explicitar la correspondencia de Curry-Howard. Muchos de los ejemplos son tomados de los libros [Stu16] y [WK19], que recomendamos como material para profundizar en el tema.

Agda tiene soporte para chequear los tipos y agregarle *syntax highlighting* a código incrustado en documentos LaTeX. Todo el código que aparezca con colores a continuación está chequeado por Agda antes de ser compilado como documento LaTeX, es decir, podemos asegurar que son programas correctos.

6.1. Nuestras primeras definiciones

Comencemos nuestro camino en Agda definiendo dos tipos de datos, los booleanos y los naturales.

```
-- Definimos los booleanos
data 𝔹 : Set where
  tt : 𝔹 -- true
  ff : 𝔹 -- false

-- Un tipo definido por inducción: los naturales
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

-- Un tipo polimórfico: lista de A
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

-- Pragmas
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN LIST List #-}
```

Ya podemos notar varias particularidades del lenguaje a partir de estas definiciones.

- En primer lugar, vemos que los tipos $\mathbb{N}$ y $\mathbb{B}$ se declaran a su vez como objetos del tipo `Set`. En Agda toda expresión tiene un tipo, y los propios tipos son valores. y podemos tener luego funciones que tomen elementos de `Set` que podrían aplicarse con $\mathbb{N}$ o $\mathbb{B}$.
- Los tipos de datos se definen mediante uno o más *constructores*. El estilo es similar al de Haskell, aunque se hace más explícito que los constructores definen

valores concretos del tipo (como `tt`, `ff` o `zero`) o son funciones que construyen un elemento (como `suc`). Vemos con $\mathbb{N}$ una definición de *tipo inductivo*, donde el constructor `suc` recibe como argumento a un elemento de $\mathbb{N}$ para construir un elemento de $\mathbb{N}$.

En Haskell las definiciones análogas serían `data Bool = Tt | Ff` y `data Nat = Zero | Suc Nat`

- Tenemos una idea similar a tipos polimórficos. En este caso decimos que el tipo List está parametrizado por (A : Set). Podrían aparecer muchos parámetros, antes de los dos puntos que definen el tipo del tipo (en este caso el tipo de List A será `Set`). La definición es luego análoga a la que haríamos en Haskell.
- La línea `{-# BUILTIN NATURAL ℕ #-}` es un *pragma*, una directiva al compilador de Agda. En este caso le estamos indicando que identifique a nuestro tipo $\mathbb{N}$ con una noción interna de los números naturales. Con esto podremos utilizar numerales para referirnos a los mismos: es decir escribir `0` será lo mismo que escribir `zero`, `2` lo mismo que `suc suc 0`, etcétera.

6.2. Definiendo funciones

Definamos algunas funciones sobre los tipos de datos que tenemos hasta ahora

```
-- And booleano
_&&_ : 𝔹 → 𝔹 → 𝔹
tt && b = b
ff && _ = ff

-- Suma de naturales
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)

if_then_else_ : {A : Set} → 𝔹 → A → A → A
if tt then a else _ = a
if ff then _ else a = a

-- Declaramos asociatividad
-- y nivel de de precedencia (igual que Haskell)
infixr 9 _+_
infixr 10 _&&_
```

Estas funciones simples nos dejan comentar más particularidades del lenguaje:

- Las funciones se definen por pattern matching, de manera muy similar a Haskell. Una diferencia a comentar es que a Agda le importan los espacios, es decir `tt && ff` no es lo mismo que `tt&&ff`. Esto es porque los identificadores pueden tener (casi) cualquier caracter Unicode (como venimos observando), y `tt&&ff` se interpreta aquí como otro identificador.

- Las funciones pueden ser infijas, como `_&&_`, pero también *mixfijas*, en donde los argumentos pueden intercalarse en posiciones arbitrarias, como muestra `if_then_else_`: las posiciones de los argumentos se marcan con guiones bajos.
- Existe la aplicación parcial manteniendo los guiones bajos donde falta aplicar: por ejemplo `(if_then_else_) tt 3` es lo mismo que `(if tt then 3 else_)` y tiene tipo $\mathbb{N} \to \mathbb{N}$, así como `if tt then_else 2`.
- La definición de `if_then_else_` muestra el uso de un *parámetro implícito*, que se escribe entre llaves (en este caso `{A : Set}`). Esto nos, en este caso, expresar el polimorfismo de `if_then_else_`: el tipo se inferirá automáticamente al ser aplicada, y no hace falta dar el parámetro explícitamente. De todas formas puede explicitarse aplicándola con el argumento entre llaves: así `(if_then_else_) {ℕ}` tiene tipo $\mathbb{B} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

6.3. Nuestros primeros tipos dependientes

Mostremos cómo puede definirse un tipo dependiente en Agda. El ejemplo canónico de tipo dependiente es el de vector: un vector es una lista de tamaño fijo, donde el tamaño es un número natural. Así, el tipo depende de un valor de $\mathbb{N}$.

Consideremos para empezar una definición de vectores que solo admite vectores de tamaño 0, 1, o 2:

```
data Vec012 (A : Set) : (n : ℕ) → Set where
  v012[] : Vec012 A 0
  v012[_] : A → Vec012 A 1
  v012[_,_] : A → A → Vec012 A 2
```

Hay bastante para decir aquí: primero, podemos notar que el tipo `Vec012` está parametrizado por A, como vimos en el caso de las listas. Sin embargo, el tipo de `Vec012 A` es ahora $\mathbb{N} \to$ `Set`, es decir podemos aplicarlo en un número natural `n` para obtener finalmente el tipo dependiente `Vec012 A n`. Se dice aquí que `n` es un *índice* del tipo.

La diferencia entre un parámetro como `A` y un índice como `n` es que todos los constructores comparten el parámetro, mientras que el índice puede variar entre constructores. Notemos que todos los constructores usan el mismo `A`, sin embargo cambian en el índice. ¡La forma del índice afecta los habitantes del tipo! Notemos que los elementos de `Vec012 A 0` son muy distintos a los de `Vec012 A 1`, y ¡los `Vec012 A n` con `n` mayor a dos ni siquiera tienen habitantes, pues no hay forma de construirlos!

¿Cómo extendemos esto a vectores de cualquier longitud? Una definición podría hacerse así:

```
data 𝕍 (A : Set) : ℕ → Set where
  [] : 𝕍 A 0
  _::_ : ∀ {n : ℕ} (x : A) (xs : 𝕍 A n) → 𝕍 A (suc n)
```

Hace su primera aparición en esta definición el símbolo ∀. Lo utilizamos para indicar que el constructor `_::_` puede aplicarse a *cualquier* `x` del tipo `A` y vector `xs` de longitud `n`, donde `n` puede ser *cualquier* natural (notemos que es un parámetro implícito), para obtener un vector de longitud `suc n`.

En rigor esta notación es solamente *syntax sugar*, y todas las siguientes serían equivalentes en su lugar:

```
∀ {n : ℕ} (x : A) (xs : 𝕍 A n) → 𝕍 A (suc n)
{n : ℕ} (x : A) (xs : 𝕍 A n) → 𝕍 A (suc n)
∀ {n : ℕ} → A → 𝕍 A n → 𝕍 A (suc n)
{n : ℕ} → A → 𝕍 A n → 𝕍 A (suc n)
```

Además en este caso el compilador podría hasta inferir el tipo de `n`, y podríamos escribir alguna de:

```
∀ {n : _} (x : A) (xs : 𝕍 A n) → 𝕍 A (suc n)
∀ {n} → A → 𝕍 A n → 𝕍 A (suc n)
```

en lugar de lo anterior.

6.4. La igualdad

En secciones anteriores vimos el tipo dependiente $x \equiv y$, que corresponde a la proposición "x es igual a y", con su constructor `refl` que dado un x nos da un elemento de $x \equiv x$. Una versión en Agda podría ser:

```
data Eq (A : Set) (x : A) : A → Set where
  refl : (Eq A x) x
```

Notemos que, como en el caso que vimos de `Vec012`, no puedo habitar todos los tipos de la forma `(Eq A x) y`: ¡el único constructor que tenemos impone que el índice sea el mismo término que el parámetro! Es decir que sólo estarán habitados los tipos `Eq A x y` donde `x` sea lo mismo que `y`, como antes los únicos `Vec012 A n` que estaban habitados eran `Vec012 A 0, Vec012 A 1, Vec012 A 2`. Esto es lo que esperamos de un tipo que represente la igualdad, pues por la correspondencia de Curry-Howard encontrar un habitante del tipo es lo mismo que demostrar la proposición.

Una definición más general, que usaremos en adelante, hace implícito el parámetro A, que puede inferirse:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x


-- Asociamos ≡ a la noción interna de igualdad de Agda
{-# BUILTIN EQUALITY _≡_ #-}
infix 3 _≡_
```

Notemos que en la definición de `refl` el `x` de la izquierda define el *parámetro* y el de la derecha el *índice* (sería lo mismo escribir `(_≡_ x) x`, como pasaba con `Eq` más arriba).

6.4.1. Igualdad definicional y proposicional

Para los tipos `_≡_` vale la misma observación que hicimos con `Eq`: los únicos tipos igualdad que están habitados son los de la forma `x ≡ x`, es decir solo encontraremos *evidencia* de que `x ≡ y` si efectivamente son lo mismo.

¿Qué quiere decir *lo mismo* en este contexto? Nos estamos refiriendo al concepto de *igualdad definicional* que ya describimos en la sección *Términos y objetos*. Mencionamos allí que la igualdad definicional se establece como juicio a nivel metateórico: en el caso de Agda, esto significa que Agda tiene una especie de "tabla de juicios" que establecen la igualdad definicional entre términos. Cada vez que definimos una función, estamos agregando juicios a esta "tabla".

Por ejemplo, si definimos la función `pred` así:

```
pred : ℕ → ℕ
pred 0 = 0
pred (suc n) = n
```

Le estamos diciendo a Agda que los términos `pred (suc n)` y `n` son *definicionalmente iguales* (lo marcamos con el símbolo `=`). Esto quiere decir que son completamente intercambiables: los dos términos reducen a alguna forma canónica siguiendo alguna cadena de definiciones.

Con este mismo criterio, los términos `1 + 1`, `pred 3`, y `2` también son definicionalmente iguales: siguiendo la definición de `_+_` tenemos:

```
1 + 1 = suc 0 + 1 = suc (0 + 1) = suc 1 = 2
```

Y siguiendo la definición de `pred`

```
pred 3 = pred (suc 2) = 2
```

Por otro lado tenemos la *igualdad proposicional*, que es la igualdad expresada a través de una proposición: esto es lo que captura el tipo que llamamos `_≡_`. A diferencia de la igualdad definicional, la igualdad proposicional (por ser una proposición), requiere una demostración, o lo que es lo mismo, una evidencia en forma de un habitante del tipo.

6.5. Nuestra primera demostración

La igualdad definicional implica la igualdad proposicional. Si sabemos que dos términos `t` y `t'` son definicionalmente iguales, entonces sabemos que `refl` podrá construir un elemento de `t ≡ t'`. Tenemos entonces una demostración "gratis" de la proposición. Basados en el ejemplo anterior, entonces, escribimos nuestra primera demostración en Agda:

```
1+1-es-2 : 1 + 1 ≡ 2
1+1-es-2 = refl
```

Es importante entender que `1+1-es-2` no solo es el título de la demostración: es un habitante del tipo `1 + 1 ≡ 2`, *es la demostración* de esa proposición. Si en alguna demostración posterior necesitáramos evidencia de esta proposición, basta con presentar el *valor* `1+1-es-2`.

Es oportuno remarcar la consecuencia práctica de la correspondencia de Curry-Howard: como la demostración de la proposición es simplemente un elemento del tipo correspondiente, con que nuestro programa pase el *type checker* podemos estar seguros de que es correcta (aunque queda a cargo del programador, por supuesto, asegurarse de que el tipo que estamos utilizando realmente corresponde a la proposición que queremos demostrar).

7. Demostrando propiedades de nuestros programas

7.1. Demostraciones universales

Consideremos la función `~` de negación booleana:

```
-- Negación
~_ : 𝔹 → 𝔹
~ tt = ff
~ ff = tt

infix 7 ~_
```

¿Qué tipo de propiedades podemos demostrar sobre esta función? Algo muy natural que podríamos desear probar es que negar dos veces es como no hacer nada.

Podríamos demostrarlo por separado para cada constructor, sabiendo que por cómo reducirán las expresiones por definición podemos usar `refl`:

```
-- Caso tt
~~-tt : ~ ~ tt ≡ tt
~~-tt = refl

-- Caso ff
~~-ff : ~ ~ ff ≡ ff
~~-ff = refl
```

Sin embargo, sería más conveniente demostrar la propiedad universal: que *para todo elemento* `b` *de* `𝔹` *se cumple* `~ ~ b ≡ b`. Podemos expresar esta proposición con un cuantificador universal, y demostrarla por pattern matching de forma igual a como hicimos cada caso por separado:

```
-- Demostrando un enunciado universalmente cuantificado
-- En este caso basta con enumerar todos los casos por pattern matching
~~-elim : ∀ (b : 𝔹) → ~ ~ b ≡ b
~~-elim tt = refl
~~-elim ff = refl
```

Observemos que el tipo de `~~-elim` **es una función dependiente: dado el argumento** `b : 𝔹` **devuelve un valor del tipo** `~ ~ b ≡ b`. Es exactamente el tipo dependiente que corresponde a una proposición con cuantificador universal que discutimos anteriormente. Su definición puede hacerse entonces como cualquier función. En este caso por pattern matching, en cada caso se reduce a `refl` de la misma manera que cuando demostramos las proposiciones por separado.

7.2. Reutilizando demostraciones

Consideremos la siguiente demostración:

```
~~&&' : ∀ (b1 b2 : 𝔹) → (~ ~ b1) && b2 ≡ b1 && b2
~~&&' tt b2 = refl
~~&&' ff b2 = refl
```

La idea de la demostración es la misma que la anterior: cuando `~ ~ b1` se concretiza a `~ ~ tt` o `~ ~ ff` por pattern matching, sabemos que reduce a `tt` y a `ff` respectivamente y podemos usar `refl`.

¿Podemos reutilizar `~~-elim` para no repetir este argumento en la demostración? La respuesta es sí, utilizando la directiva `rewrite`. Veamos como queda antes de explicarlo:

```
~~&& : ∀ (b1 b2 : 𝔹) → (~ ~ b1) && b2 ≡ b1 && b2
~~&& b1 b2 rewrite (~~-elim b1) = refl
```

¿Qué esta sucediendo aquí? Sabemos que para definir `~~&& b1 b2` debemos dar algo del tipo `(~ ~ b1) && b2 ≡ b1 && b2`. Al escribir `rewrite p`, donde `p` debe ser un elemento de tipo `X ≡ Y`, Agda sabe que puede reemplazar toda aparición de `X` en el tipo que debemos lograr por `Y`, porque **p es evidencia de que el término** `X` **es proposicionalmente igual que** `Y`. Esto funciona solamente con los tipos `_≡_`, porque oportunamente lo asociamos con la noción interna de igualdad proposicional de Agda mediante el pragma `BUILTIN EQUALITY`.

En nuestro caso, al aplicarlo con `~~-elim b1`, que tiene tipo `~ ~ b1 ≡ b1`, ahora en lugar de dar evidencia de `(~ ~ b1) && b2 ≡ b1 && b2` debemos dar evidencia de `b1 && b2 ≡ b1 && b2`, pero ¡estos términos son definicionalmente iguales! Luego la evidencia es `refl`.

Vemos con esto otra ventaja de que `~~-elim` sea una demostración universal: de otra forma no lo podríamos haber aplicado a un `b1` genérico para hacer la reescritura.

7.3. Demostraciones por inducción

Para demostrar cosas sobre funciones de $\mathbb{B}$ podemos usar pattern matching exhaustivo sobre sus constructores, pero para demostrar proposiciones sobre tipos inductivos debemos usar argumentos inductivos. Consideremos la siguiente demostración:

```
+0 : ∀ (a : ℕ) → a + 0 ≡ a
+0 0 = refl
+0 (suc n) rewrite (+0 n) = refl
```

Recordemos que la definición de `_+_` es inductiva en su primer argumento, luego `a + 0 ≡ a` no es trivialmente cierto por igualdad definicional (mientras que `0 + a ≡ a` sí lo es).

La primera línea de la demostración corresponde al caso base de la inducción: demostrar `+0 0` es dar un elemento de `0 + 0 ≡ 0`. Estos dos términos son definicionalmente iguales, luego podemos dar evidencia con `refl`.

En la segunda línea debemos demostrar `+0 (suc n)`: esto es, dar un elemento de `(suc n) + 0 ≡ suc n`. Por definición de `_+_`, `(suc n) + 0 = suc (n + 0)`, luego basta con dar un elemento de `suc (n + 0) ≡ suc n`.

Si pudiéramos ahora reemplazar `n + 0` por `n`, entonces terminaríamos con `refl` (pues los términos serían definicionalmente iguales). Para eso debemos proveerle a rewrite evidencia de que `n + 0 ≡ n`. ¡Pero eso es justamente lo que hace la función que estamos definiendo! Podemos invocarla recursivamente sobre `n` para obtener un elemento de `n + 0 ≡ n`, y es precisamente lo que hacemos. Esto corresponde al paso inductivo en una demostración por inducción, donde invocamos la evidencia de la hipótesis inductiva con el llamado recursivo.

7.4. Demostraciones interactivas

Para terminar, demostremos un par de propiedades más. Comencemos por la simetría de `_≡_`: la proposición de que si `a ≡ b` entonces `b ≡ a`.

```
sym : ∀ {A : Set} {a b : A} → a ≡ b → b ≡ a
sym refl = refl
```

El tipo que corresponde a una implicación es una función: `sym` es entonces una función que recibe evidencia de `a ≡ b` y devuelve evidencia de que `b ≡ a` (y los valores sobre los que depende, es decir las variables cuantificadas, se reciben implícitamente).

La demostración es muy sencilla: al hacer *pattern matching* sobre `refl`, que es el único constructor de un tipo `a ≡ b`, Agda puede darse cuenta de que `a` y `b` son definicionalmente iguales (pues de lo contrario ese tipo no podría ser construido por `refl`), luego la evidencia de `b ≡ a` es simplemente `refl`.

Este hecho parece trivial a simple vista, pero es muy útil cuando queremos "invertir el orden" de dos términos asociados con `_≡_` en un `rewrite`. Veremos un ejemplo de su uso a continuación.

Terminaremos esta sección ilustrando el carácter interactivo de Agda. Agda posee un plugin para el editor *Emacs* (e instalaciones personalizadas de Emacs) que permiten utilizarlo interactivamente para realizar demostraciones de manera incremental.

Intentaremos demostrar la conmutatividad de la suma. La proposición que queremos demostrar es `∀ (a b : ℕ) → a + b ≡ b + a`, y procedemos por inducción en la primera variable. Si en el editor de Agda escribimos los casos que tenemos que contemplar, pero con un `?` en donde deben ir las definiciones:

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = ?
+comm (suc n) b = ?
```

y luego tecleamos `Control-x, Control-l`, veremos aparecer lo siguiente:

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = {}0
+comm (suc n) b = {}1
```

En lugar de los signos de interrogación aparecen lo que se llaman *agujeros*. Además, Agda nos indica qué tenemos que llenar en los agujeros:

```
?0 : zero + b ≡ b + zero
?1 : suc n + b ≡ b + suc n
```

Esto nos ayuda a llenar cada agujero. En primer lugar, vemos que en el caso caso base debemos dar un valor del tipo `0 + b ≡ b + 0`. Esto es definicionalmente igual a `(b ≡ b + 0)`, que es la proposición simétrica a la proposición `(+0 b)`, que demostramos antes. Luego podemos lograr evidencia de eso con `sym (+0 b)`.

Llenando el agujero:

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = {sym (+0 b)}0
+comm (suc n) b = {}1
```

y tecleando `Control-c, Control-r`, si lo que llenamos es correcto el agujero desaparece y nos queda el otro:

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = sym (+0 b)
+comm (suc n) b = {}0
```

En este caso, deberíamos llenarlo con un valor de `suc n + b ≡ b + suc n`. Lo primero que podemos intentar es aplicar recursivamente:

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = sym (+0 a)
+comm (suc n) b rewrite (+comm n b) = {}0
```

Si volvemos a teclear `Control-c, Control-l` Agda nos dice que ahora lo que tenemos que llenar en el agujero es un valor de:

```
suc (b + n) ≡ b + suc n.
```

No es claro como podríamos demostrar esto directamente aquí, pero vemos que esto es una propiedad general de los naturales que podemos demostrar aparte como un lema. Para llegar a esta demostración también podemos hacer uso de los agujeros, pero omitimos esos pasos por brevedad. Obtenemos la demostración:

```
lem-suc : ∀ (a b : ℕ) → suc (a + b) ≡ a + suc b
lem-suc 0 b = refl
lem-suc (suc n) b rewrite lem-suc n b = refl
```

Finalmente, si lo aplicamos como un segundo rewrite (utilizando el símbolo | para separarlo del otro):

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = sym (+0 a)
+comm (suc n) b rewrite (+comm n b) | (lem-suc b n) = {}1
```

Vemos que la pista del agujero es:

```
b + suc n ≡ b + suc n
```

Estos términos son definicionalmente iguales, luego podemos llenar el agujero con `refl`. Obtenemos la demostración final:

```
+comm : ∀ (a b : ℕ) → a + b ≡ b + a
+comm 0 b = sym (+0 b)
+comm (suc n) b rewrite (+comm n b) | (lem-suc b n) = refl
```

8. Verificación interna

Hasta ahora estuvimos tipos de datos (como ℕ) y programas (como `_+_`) y luego demostrando independientemente propiedades sobre los mismos. Esto podría llamarse *verificación externa*: las demostraciones son externas a los programas.

En contraste, podemos considerar un estilo de verificación que podemos llamar *verificación interna*, en donde expresamos las proposiciones *dentro de los mismos tipos de datos y programas*: la idea es escribir tipos y funciones más expresivos: la corrección de las funciones y las invariantes de las estructuras de datos están garantizadas por el propio tipo.

8.1. Vectores

Ya hemos visto un ejemplo cuando definimos $\mathbb{V}$: los vectores tienen su longitud en el mismo tipo. Tener la longitud asociada nos permite expresar relaciones entre las longitudes de entradas y salidas de funciones sobre vectores.

Por ejemplo, si definimos la concatenación de vectores y la función `head`$\mathbb{V}$ así:

```
-- Concatenación de vectores
_++𝕍_ : ∀ {A : Set} {n m : ℕ} → 𝕍 A n → 𝕍 A m → 𝕍 A (n + m)
[] ++𝕍 ys = ys
(x :: xs) ++𝕍 ys = x :: (xs ++𝕍 ys)
```

El simple hecho de que la función compile (es decir, que pase por el *type checker*) nos indica que, realmente, la longitud de la concatenación es la suma de las longitudes de los vectores de entrada.

Para la concatenación de listas, que se define de forma análoga:

```
_++_ : ∀ {A : Set} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

La demostración de esa propiedad habría que hacerla externamente, definiendo una función de longitud y luego usando una estrategia como las que vimos en la sección anterior.

Estos tipos con información (como $\mathbb{V}$ y su longitud) nos dejan definir funciones dependientes que no podríamos de otra manera. Recordemos que en Agda toda función debe terminar (toda función es estricta), luego no es válido definir la función `head` sobre listas de manera similar a Haskell

```
-- Error: Esto no puede hacerse en Agda!
head : ∀ {A : Set} → List A → A
head (x :: xs) = x
```

En cambio podemos utilizar la longitud asociada a los vectores para definir una version segura de `head` para ellos:

```
-- Extraer el primer elemento de vectores no vacíos
head𝕍 : ∀ {A : Set} {n : ℕ} → 𝕍 A (suc n) → A
head𝕍 (x :: xs) = x
```

En efecto, `head`$\mathbb{V}$ solo puede recibir vectores de longitud 1 o más (`suc n` para algún `n` implícito).

8.2. Árboles binarios de búsqueda

El segundo y último ejemplo que consideraremos es el de un árbol binario de búsqueda con elementos naturales. Para definir el árbol binario debemos primero definir

el orden entre los naturales. Lo haremos a través de una función que devolverá el booleano `tt` cuando el primer argumento sea menor o igual al segundo:

```
_≤_ : ℕ → ℕ → 𝔹
zero ≤ zero = tt
zero ≤ (suc a) = tt
(suc b) ≤ zero = ff
(suc a) ≤ (suc b) = a ≤ b

infix 10 _≤_
```

Para expresar la invariante del árbol binario de búsqueda, a cada nodo le asociaremos una cota superior y una inferior, y garantizaremos que todos los valores del subárbol que tiene al nodo como raíz estén entre las cotas. Finalmente, el árbol será de búsqueda si el valor del nodo esta entre la cota superior del subárbol izquierdo y la inferior del árbol derecho:

```
-- Árboles binarios de búsqueda con elementos naturales.
-- Indizados por dos elementos naturales, la cota inferior
-- y la superior de los elementos del árbol
data bstℕ : ℕ → ℕ → Set where
  leaf : ∀ {l u : ℕ} → l ≤ u ≡ tt → bstℕ l u
  node : ∀ {ll lr ul ur : ℕ}
           (elem : ℕ) → bstℕ ll ul → bstℕ lr ur →
           ul ≤ elem ≡ tt → elem ≤ lr ≡ tt →
           bstℕ ll ur
```

Como se ve, los constructores requieren *evidencia* de que las cotas son correctas. Para construir una hoja (vacía) con cotas `l, u` hay que proveer evidencia de que `l` es menor o igual que `u`.

De la misma forma, para construir un árbol a partir de un elemento y dos subárboles se usa el constructor `node`. A ese constructor no solo se le pasa el elemento `elem`, sino evidencia de que `elem` se encuentra entre las cotas de los subárboles, como explicamos antes.

El árbol que se construye está acotado por la cota inferior del subárbol izquierdo y la superior de la del subárbol derecho.

Podemos construir paso a paso un árbol con un nodo con un 5 en la raíz y un nodo con un 3 a la izquierda (y demás hojas):

```
-- Esta hoja estará a la izquierda de algún nodo
-- que tenga un elemento mayor o igual a 2
leaf1 : bstℕ 0 2
leaf1 = leaf refl

-- Esta hoja podrá estar a la derecha de algún nodo
-- que tenga un elemento menor o igual a 4
```

```
-- o a la izquierda de algún nodo que tenga un
-- elemento mayor o igual a 5
leaf2 : bstℕ 4 5
leaf2 = leaf refl

-- Esta hoja podrá estar a la derecha de algún nodo
-- que tenga un elemento menor o igual a 6
-- o a la izquierda de algún nodo que tenga un
-- elemento mayor o igual a 7
leaf3 : bstℕ 6 7
leaf3 = leaf refl


-- Las cotas de los nodos las determinan los subárboles
-- (en este caso las hojas leaf1 y leaf2)
tree[*-3-*] : bstℕ 0 5
tree[*-3-*] = node 3 leaf1 leaf2 refl refl


tree[[*-3-*]-5-*] : bstℕ 0 7
tree[[*-3-*]-5-*] = node 5 tree[*-3-*] leaf3 refl refl


-- El árbol que construimos es:
--------[5]-------
------[3]--*------
-----*--*---------
-- Y las cotas son 0 (inferior) y 7 (superior)
```

8.3. Conclusiones

La verificación interna tiene la ventaja de que obtenemos tipos de datos y operaciones muy ricos semánticamente, con garantías de integridad e invariantes expresadas dentro de las propias operaciones.

La contracara es que es más engorroso trabajar con ellos pues se deben garantizar las invariantes en cada operación. Por ejemplo, para construir función de inserción para `bstℕ` debemos garantizar que el resultado sea correcto, no importa el árbol ni el elemento que se inserte: no es para nada trivial escribir esta función.

9. Conclusiones finales

La tendencia a construir software verificado no puede menos que aumentar en los próximos años. Nos hemos introducido en la teoría de tipos de Martin-Löf y sus tipos dependientes como herramienta teórica y práctica para lograr verificar nuestro

software, sostenidos en la correspondencia de Curry-Howard y en un sistema de tipos potentes como el de Agda.

El carácter interactivo de Agda y su nivel de expresividad lo hacen muy cómodo para la tarea, y nos permiten adoptar estilos de verificación interna o externa según la ocasión lo amerite. Esperamos haber convencido al lector de que vale la pena incursionar un poco en este mundo, aún cuando este trabajo apenas toca la superficie de lo que es posible hacer con estas técnicas. Nuevamente referimos a [Stu16] y [WK19] como lecturas muy completas sobre el lenguaje y sus capacidades que no presumen conocimiento previo.

La gran ventaja de las proposiciones como tipos y la verificación dentro del mismo lenguaje de programación han llevado a un gran interés en tener tipos dependientes dentro del propio Haskell. Si bien existen dificultades al compatibilizar las características de Haskell que entran en conflicto con la teoría de Martin-Löf y los tipos dependientes (en particular con efectos secundarios y no-terminación), ya existen aproximaciones. La funcionalidad es todavía incipiente y se basa en extensiones del lenguaje que hay que explícitamente activar.

Para trabajo teórico al respecto referimos al lector a [McB02] y [LM14]. Para un ejemplo ilustrativo se puede consultar `https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell`.

Teniendo en cuenta que introducir verificación a Haskell es una gran ventaja por su gran adopción en ambientes fuera de la academia como lenguaje de propósito general, otro proyecto interesante es **Idris** [Bra13], un lenguaje de programación inspirado en Agda pero con todavía mayores ambiciones que Agda de ser un lenguaje de propósito general y no tanto un asistente de demostraciones.

Referencias

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[CH11] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[Haf10] Florian Haftmann. From higher-order logic to haskell: there and back again. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 155–158. ACM, 2010.

[Har03] John Harrison. Formal verification at intel. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 45–54. IEEE, 2003.

[HUn] HUnit. `https://github.com/hspec/HUnit`. Accedido: 2019-11-16.

[LM14] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. *ACM SIGPLAN Notices*, 48(12):81–92, 2014.

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[McB02] Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[SBRW17] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. *CoRR*, abs/1711.09286, 2017.

[Stu16] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan Claypool, New York, NY, USA, 2016.

[Wad15] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[WK19] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. 2019. Available at `http://plfa.inf.ed.ac.uk/`.