Una formalización del sistema de los números reales

Jorge Ohel Acevedo Acosta
Jose Luis Echeverri Jurado

Tesis
para obtener el título de
Magíster en Matemáticas Aplicadas

Asesor: Andrés Sicard Ramírez

Departamento de Ciencias Matemáticas
Escuela de Ciencias
Universidad EAFIT
Medellín, Colombia
2017

Agradecimientos

A través de estas líneas queremos expresar nuestro más sincero agradecimiento a todas las personas que con su soporte científico y humano han colaborado en la realización de esta tesis de investigación. Muy especialmente a nuestro asesor y director de tesis al profesor Andrés Sicard Ramírez, por la acertada orientación, el soporte y discusión crítica que nos permitió un buen aprovechamiento en el trabajo realizado para que esta tesis llegara a buen término.

Finalmente, agradecemos a nuestras familias por su comprensión, comunicación constante y apoyo. De manera muy especial a nuestras esposas quienes han estado a nuestro lado compartiendo nuestras alegrías y angustias, por el estímulo para que nos superemos día a día, el apoyo incondicional y la ayuda de siempre.

Resumen

Se presenta una formalización de una axiomática del sistema de los números reales en un asistente de pruebas. Además de una axiomática clásica se presenta una axiomática del sistema de los números reales para la matemática intuicionista. Se demuestran formalmente de forma interactiva y automática algunas propiedades del sistema. Las demostraciones automáticas fueron realizadas por demostradores automáticos de teoremas de propósito general.

Índice general

Capítulo 1

Introducción

La formalización de una teoría matemática requiere de elementos y métodos avalados por esta ciencia, aún más cuando ésta es asistida por un computador. Desde mediados del siglo XX la matemática ha entrado en conexión con la utilización del ordenador para facilitar y, por qué no, potenciar el desarrollo de diversas teorías matemáticas. En el campo de las ciencias de la computación y la lógica matemática se encuentran, por ejemplo, los demostradores automáticos de teoremas, éstos son programas que se encargan, entre otras aplicaciones, de demostrar teoremas o propiedades de diferentes teorías. También existen en la actualidad diversos asistentes de prueba como por ejemplo Agda [Norell 2007], Coq [The Coq Development Team 2016], HOL Light [Harrison 2015] o Isabelle [Nipkow, Paulson y Wenzel 2016]; los cuales se vienen utilizando, entre otras aplicaciones, como herramientas para la escritura y verificación de demostraciones matemáticas formales. Estos asistentes son hoy en día una herramienta fundamental para la formalización en diferentes teorías matemáticas, como fue el caso de la demostración de la conjetura de Kepler [Hales 2005] o el teorema de los cuatro colores [Appel y Haken 1989].

En la tesis que se presenta a continuación se desarrolla una formalización del sistema de los números reales utilizando el asistente de pruebas Agda. Algunos teoremas se demuestran de forma interactiva, otros se demuestran de forma automática empleando el programa Apia [Sicard-Ramírez 2015, cap. 6], el cual es una interfaz entre Agda y los diferentes demostradores automáticos de teoremas.

La formalización del sistema de números reales presentada en esta tesis es desarrollada de forma axiomática, tal como lo presentan [Apostol 1974;

Rosenlicht 1968; Royden y Fitzpatrick 2010]. Este desarrollo se focaliza en la definición de contantes, funciones y los axiomas usuales del sistema de los números reales, para luego demostrar algunos teoremas básicos. En la tesis se demuestran formalmente algunas propiedades de la desigualdades, propiedades que requieren demostraciones por inducción, la convergencia de algunas series y algunas propiedades de los límites. Se incluye además una sección desde la matemática intuicionista, donde además de los axiomas usuales se introduce la propiedad Arquimediana como un axioma, debido a que no es posible utilizar la ley del tercero excluido.

En el capítulo 2 se mencionan algunos conceptos básicos del programa Agda, el cual es un asistente de pruebas interactivo y un lenguaje de programación funcional donde se abordan temas tales como tipos inductivos, tipos dependientes, funciones recursivas, adición de axiomas y razonamiento ecuacional.

En el capítulo 3 se formalizan en Agda las definiciones de las operaciones de adición y multiplicación para luego hacerlo con los axiomas de campo, de orden y de completitud. Adicionalmente se presentan en este capítulo las diferencias que se hacen, desde la matemática intuicionista, al desarrollo axiomático del sistema de los números reales.

En el capítulo 4, se formalizan algunas demostraciones empleando Agda. Este tipo de demostraciones es lo que en esta tesis se denomina 'demostraciones interactivas'. En este capítulo además utilizando el programa Apia, un traductor de la representación que hace Agda de fórmulas de primer orden al lenguaje TPTP, y algunos demostradores automáticos de teoremas, se demuestran de forma automática algunas de las propiedades en el sistema de los números reales que se demostraron de forma interactiva con el programa Agda. Finalmente en el capítulo 5 se ilustra la formalización de el límite de una función.

La información sobre cómo obtener el programa Agda y más detalles referentes a éste se pueden encontrar en la *Wiki* de Agda[1], y todo lo necesario para el programa Apia se puede encontrar en un repositorio en GitHub.[2] La formalización y los ejemplos presentados en esta tesis y otros más se encuentran disponibles también en un repositorio en GitHub.[3]

[1] http://wiki.portal.chalmers.se/agda/pmwiki.php.

[2] https://github.com/asr/apia.

[3] https://github.com/jechev28/real-numbers-formalisation.

Capítulo 2

Conceptos básicos de Agda

En los últimos años ha crecido la atención hacia el uso del ordenador por parte de los matemáticos, en donde éstos pueden encontrar datos experimentales, patrones, formular conjeturas, validar teoremas, entre otras actividades. Se mencionan dos casos de demostraciones haciendo uso del computador: una es la demostración del teorema de los cuatro colores [Appel y Haken 1989], verificada en el asistente de pruebas Coq, la otra es la demostración presentada por Hales [2005] de la conjetura de Kepler, verificada combinando los asistentes de prueba Isabelle y HOL Light. En este capítulo se realiza una breve introducción al asistente de pruebas Agda, donde se explican sus características utilizadas en esta tesis. Para entrar en más detalles y mirar lo relacionado con este programa se puede consultar [Bove y Dybjer 2009], [Norell 2009], [Sicard-Ramírez 2015, cap. 2] y la página web de Agda.[1] El lector que este familiarizado con Agda puede omitir este capítulo.

Agda es un lenguaje de programación funcional con soporte para tipos dependientes, es decir tipos de datos que dependen de términos, y también con soporte para definiciones inductivas. Agda también es un asistente de pruebas interactivo, donde se puede realizar la escritura y verificación de la demostración de teoremas. Agda está basado en la teoría de tipo intuicionista, un sistema fundacional para el desarrollo de la matemática intuicionista [Martin-Löf 1984]. Agda es de código abierto y entre sus mayores contribuidores se encuentra la Universidad Tecnológica de Chalmers y la Universidad de Gotemburgo. Como ayuda para escribir programas en Agda se utiliza la interfaz interactiva para el editor Emacs [Stallman et al. 2015], la cual permite y fa-

[1] http://wiki.portal.chalmers.se/agda/pmwiki.php.

cilita crear objetivos, definir tipos inductivos, definir funciones, refinar los términos en cada prueba, *type-check* el código, entre otras actividades.

Este capítulo consta de las siguientes secciones: En la sección 2.1, se definen los tipos inductivos que se introducen en Agda con la declaración `data`. En la sección 2.2, se ofrece una explicación sobre tipos dependientes, tipos que dependen de otros tipos. En la sección 2.4, se definen funciones recursivas por *pattern matching* sobre términos inductivos. En la sección 2.5, se presenta una breve explicación de las funciones que se definen en términos de si mismas. En la sección 2.8, se muestra como Agda permite agregar términos con la palabra reservada `postulate`. En la sección 2.9, se introduce la currificación como una técnica donde se reescriben las funciones de más de un argumento a funciones de un argumento. En la sección 2.10, se define la igualdad empleada en este tesis. En la sección 2.11, se muestra el empleo del razonamiento ecuacional para verificar pruebas en Agda usando un estilo algebraico.

2.1. Tipos inductivos

Los tipos inductivos se introducen en Agda con la declaración `data`. La regla de formación dice como se construye un cierto tipo a partir de otros tipos. Las reglas de introducción dicen como se puede construir términos a partir de otros términos [Martin-Löf 1984]. Las reglas de introducción se implementan nombrando el constructor de datos y su tipo.

Comentario. El símbolo dos puntos '`:`' en Agda denota que un término es de un tipo determinado. Por ejemplo `b : B` denota que el término `b` es de tipo `B`.

Ejemplo 2.1. Para esta tesis se define el conjunto de los números naturales, nombrado `ℕ`, por:

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

En la definición anterior se introduce un nuevo tipo llamado `ℕ` con dos constructores; `zero`, un constructor nulo y `succ`, un constructor unario recursivo.

Otras definiciones que se introducen en Agda con la declaración `data`, que serán utilizadas en el capítulo 3; son la conjunción y el cuantificador existencial.

Ejemplo 2.2 (Definición de la conjunción)**.** Una función básica en la lógica es la conjunción, nombrada `_∧_`, la cual implica que si ésta es verdadera sus términos también lo son.

```
data _∧_ (A B : Set) : Set where
  _,_ : A → B → A ∧ B
```

El constructor `_,_` establece que una prueba de la conjunción consta de dos componentes, el primer componente es un término de tipo `A` que es una prueba de `A` y el segundo componente es un término de tipo `B` que es una prueba de `B`.

Comentario. En Agda, si el nombre de una declaración contiene un guión bajo '_' esta declaración puede ser usada como un operador cuyos argumentos van en los respectivos guiones bajos. De esta manera, el tipo `_∧_` puede ser utilizado como un operador infijo `a ∧ b` en vez de `_∧_ a b`. Cuando una declaración está escrita usando la notación infija se le llama un operador [Bird 1998, pág. 13].

Ejemplo 2.3 (Definición del cuantificador existencial)**.** El cuantificador existencial, nombrado `∃ᵣ`, indica que existe al menos un término que satisface una determinada función proposicional.

```
data ∃ᵣ (P : ℝ → Set) : Set where
  exist : (x : ℝ ) → P x → ∃ᵣ P
```

Una prueba del existencial `∃ᵣ P`, donde `P` es una función proposicional de tipo `ℝ → Set`, está establecida por el constructor `exist` el cual recibe un término `x` de tipo `ℝ`, llamado testigo, y una prueba de que el testigo satisface la función proposicional `P`. En la sección 3.1 se introduce a `ℝ` como el tipo del sistema de los números reales.

2.2. Tipos dependientes

Los tipos dependientes son tipos que dependen de elementos de otros tipos [Bove y Dybjer 2009].

Ejemplo 2.4. Se definen los vectores $\mathbb{R}^n$, donde $\mathbb{R}$ denota los números reales y n denota la longitud del vector.

$$\mathbb{R}^n = \underbrace{\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R}}_{n}.$$

En este caso los vectores $\mathbb{R}^n$ dependen del valor de n.

Ejemplo 2.5. A continuación se define un vector `V`, donde este vector es una lista de ciertos tipos de elementos de tipo `A` y de longitud `n`. El parámetro `A` es constante y no cambia en este caso, mientras que `n` puede variar para cada lista. La familia de vectores de tipo dependiente está dada por:

```
data V (A : Set) : ℕ → Set where
  v₀ : V A zero
  vₛ : (n : ℕ) → A → V A n → V A (succ n)
```

En la definición anterior `V` es de tipo `Set → ℕ → Set` y el tipo de `V A` es `ℕ → Set`.

Ejemplo 2.6. Una familia de vectores Booleanos en Agda está dada por:

```
data V-Bool : ℕ → Set where
  v₀ : V-Bool zero
  vₛ : (n : ℕ) → Bool → V-Bool n → V-Bool (succ n)
```

Se observa que el parámetro `A : Set` de la definición de la familia de vectores `V` ya no aparece y para este caso particular se usó el tipo `Bool` el cual está definido de la forma usual.

2.3. Argumentos implícitos

Agda tiene un mecanismo que permite declarar argumentos implícitos encerrándolos entre llaves `{,}` en lugar de paréntesis `(,)` Norell [2009].

Comentario. En Agda el tipo `Set` es el universo de los *small types*. Entre los universos existe una jerarquía donde por ejemplo, el tipo `Set` (también llamado `Set₀`) es de tipo `Set₁`, el tipo `Set₁` es de tipo `Set₂` y así sucesivamente.

Ejemplo 2.7. Se define la función identidad, nombrada `id`, en los *small types*.

```
id : {A : Set} → A → A
id A x = x
```

Se puede observar que en la función `id` no se tiene que suministrar el argumento al tipo. Este argumento está implícito tanto cuando la función se aplica como cuando se define.

2.4. Tipos de datos y *pattern matching*

En Agda, como en otros lenguajes de programación funcional, se puede definir funciones recursivas por *pattern matching* sobre los términos que pertenecen a tipos inductivos. Por ejemplo, las funciones sobre `Bool` pueden ser definidas por *pattern matching* en su argumento.

Ejemplo 2.8. La función negación, nombrada `not`, está definida para los términos de tipo `Bool` por:

```
not : Bool → Bool
not true  = false
not false = true
```

La definición de funciones en Agda emplea los constructores del tipo de dato, es decir, para la función `not` hay dos casos posibles, i) `not true` y ii) `not false`, los cuales deben ser definidos.

Agda comprueba la definición de las funciones y produce un mensaje de error si falta algún constructor.

Ejemplo 2.9. Se define la función `badNot` por:

```
badNot : Bool → Bool
badNot true = false
```

Agda genera el siguiente mensaje informando que falta un caso:

```
Incomplete pattern matching for badNot. Missing cases:
 badNot false when checking the definition of badNot
```

Ejemplo 2.10. La negación implementada para esta tesis, nombrada `¬_`, está definida por:

```
¬_ : Set → Set
¬ A = A → ⊥
```

La constante ⊥, llamada *bottom*, está definida por:

```
data ⊥ : Set where
```

Se observa que ⊥ no tiene constructores, es un tipo para el cual es imposible producir un término.

Una definición importante que se plantea a partir del tipo ⊥ es la eliminación por *bottom*, nombrada `⊥-elim`, está definida por:

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Dado un término de tipo ⊥ se obtiene un término de tipo A.

2.5. Funciones recursivas

Una función es recursiva cuando se define en términos de si misma.

Ejemplo 2.11 (Adición de números naturales)**.** La adición de números naturales, nombrada `_+_`, se define por *pattern matching* sobre su primer argumento y tiene un llamado recursivo.

```
1 infixl 6 _+_
2 _+_ : ℕ → ℕ → ℕ
3 zero     + n = n
4 (succ m) + n = succ (m + n)
```

La primera línea denota que el operador binario `_+_` asocia por la izquierda con precedencia 6.

Comentario. La asociatividad y precedencia de una función en Agda se denota con las palabras **`infix`**, **`infixl`** o **`infixr`**.

2.6. Clausulas `where`

Una clausula **`where`** se puede usar para introducir una definición local. Las definiciones locales pueden ser una abreviatura o una definición recursiva.

2.7. Abstracción `with`

La palabra reservada `with` permite hacer *pattern matching* en el resultado de un cálculo intermedio, agregando un argumento adicional al lado izquierdo de la igualdad en la definición de una función.

2.8. Adición de axiomas

Agda permite agregar términos de un tipo determinado sin ser necesario definirlos. La palabra reservada para este propósito es `postulate`. A continuación se postula el tipo del sistema de los números reales `ℝ` y dos de sus términos `r₀` y `r₁`, que representan al cero y al uno respectivamente.

Ejemplo 2.12. Se muestra a continuación el tipo del sistema de números reales y dos de sus términos.

```
postulate
  ℝ      : Set
  r₀ r₁ : ℝ
```

2.9. Funciones currificadas

La currificación es una técnica donde se reescriben las funciones de más de un argumento a funciones de un sólo argumento. Bird [1998, pág. 11] plantea: "*A useful device for reducing the number of parentheses in an expression is the idea of replacing a structured argument by a sequence of simpler ones.*" La función $*_1 : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ está currificada y es isomorfa con la función $*_2 : (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$. La función $*_1$ recibe un número real y retorna otra función, esta otra función recibe un número real como argumento y retorna un número real. Las funciones currificadas son de orden superior, debido a que éstas pueden recibir funciones como parámetros y/o retornar funciones como resultados. De manera informal desde la teoría de conjuntos, la currificación es la correspondencia entre el conjunto $A^{B\times C}$ de funciones de $B \times C$ a A y el conjunto $(A^C)^B$ de funciones de B al conjunto de funciones de C a A. En teoría de categorías se demuestra un isomorfismo entre los morfismos $(B \times C) \to A$ y los morfismos $B \to A^C$ (véase, por ejemplo, Awodey [2010, cap. 6]).

Ejemplo 2.13. Reducir el número de paréntesis de una expresión en una secuencia de argumentos más simples.

$$f_1 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$f_1(m, n) = m + n,$$

$$f_2 : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$f_2\, m\, n = m + n.$$

Para cada número natural m la función $f_2\, m$ adiciona m a un número natural. En particular; $f_2\, 1$ es la función sucesora que incrementa su argumento en 1; y $f_2\, 0$ es la función identidad sobre los números naturales. La función f_2 está currificada y es isomorfa con la función f_1.

2.10. Tipo identidad

Se define la igualdad proposicional, es decir, para la igualdad lógica. La regla de introducción es la propiedad reflexiva (refl) y la regla de eliminación (subst) es la propiedad de sustitución de proposiciones [Sicard-Ramírez 2015, pág. 28]. Estas reglas están dadas por:

$$\frac{}{(\mathrm{x} = \mathrm{x})}\ (\mathrm{refl})$$

$$\frac{x = y \qquad A(x)}{A(y)}\ (\mathrm{subst})$$

La igualdad implementada para el tipo $\mathbb{R}$, nombrada `_≡_`, usada en esta tesis se define por:

```
data _≡_: ℝ → ℝ → Set where
  refl  : (x : ℝ) → x ≡ x
```

La igualdad `_≡_` recibe dos términos en $\mathbb{R}$ y retorna la prueba que los dos términos son iguales. Usando la regla de introducción `refl` se puede elaborar una demostración que un término es igual así mismo.

La propiedad de sustitución para el tip $\mathbb{R}$, nombrada `subst`, se define como:

```
subst : (P : ℝ → Set) → {x y : ℝ} → x ≡ y → P x → P y
subst P (refl x) Px = Px
```

La regla de eliminación `subst` indica que si dos términos `x` e `y` de tipo ℝ son iguales y se tiene una demostración que el término `x` satisface la función proposicional `P`, de tipo `ℝ → Set`, también se tiene una demostración que el término `y` satisface la función proposicional `P`.

De otra parte, la negación de la igualdad, nombrada `≢`, está dada por la definición:

```
_≢_ :  ℝ → ℝ → Set
x ≢ y = ¬ x ≡ y
```

2.11. Razonamiento ecuacional

El razonamiento ecuacional [Mu, Ko y Jansson 2009] es empleado en Agda para verificar pruebas usando un estilo algebraico, donde cada paso es justificado al frente enunciando las propiedades empleadas.

Se define el siguiente conjunto de combinadores con el objeto de ser utilizados en el razonamiento ecuacional:

```
infixr 2 _≡⟨_⟩_
infix 3 _∎

_≡⟨_⟩_ : ∀ x {y z} → x ≡ y → y ≡ z → x ≡ z
_ ≡⟨ x≡y ⟩ y≡z = ≡-trans x≡y y≡z

_∎ : ∀ x → x ≡ x
_∎ x = refl x
```

El combinador `_∎` recibe un término a y produce una prueba $a \equiv a$ usando reflexividad. El combinador `_≡⟨_⟩_` toma tres argumentos: a sobre la izquierda, una prueba que a es igual a a' en los paréntesis angulares `⟨,⟩` y una prueba $a' \equiv a''$ en el lado derecho. Estos combinadores producen una prueba que a es igual a a'' usando la transitiva de la igualdad `_≡_`.

Usando estos combinadores se puede establecer un razonamiento en cadena definido por:

```
a₁     ≡⟨ reason₁ ⟩
...
a₍ₙ₋₁₎ ≡⟨ reason₍ₙ₋₁₎ ⟩
aₙ □
```

donde `reasonᵢ : aᵢ ≡ a₍ᵢ₊₁₎`.

Comentario 2.1. Los combinadores arriba presentados pueden ser definidos para cualquier relación binaria que sea reflexiva y transitiva (véase apéndice A).

Capítulo 3

Una formalización de una axiomática para el sistema de los números reales

Se podría hacer una presentación rigurosa del sistema de los números reales a partir de algunos de los principios básicos de la teoría de conjuntos empezando por el sistema de los números naturales, luego el sistema de los números enteros, hasta llegar al sistema de los números racionales e irracionales y finalmente presentar el sistema de los números reales. Una presentación del sistema de los números reales empleando el enfoque anterior se encuentra por ejemplo en Landau [1951, cap. 4] o en Enderton [1977, cap. 5].

Otra forma de presentar el sistema de los números reales es como un modelo de una teoría axiomática [Chang y Keisler 1990]. Este modelo consta de un dominio, constantes, funciones y relaciones. Se puede entonces obtener un cuerpo ordenado, completo y único a partir de una axiomática.[1] La presentación del sistema de los números reales en esta tesis será axiomática.

En los textos de matemáticas que realizan un desarrollo axiomático del sistema de los números reales usualmente se presentan los axiomas clasificados en tres grupos: axiomas de campo, axiomas de orden y un axioma de completitud (o axioma del supremo). Esta tesis empleará la clasificación anterior, en particular, la tesis seguirá la presentación realizada en [Apostol 1974].

En la sección 3.1, se formalizan en Agda las funciones básicas y los axiomas

[1] Los números reales son el único cuerpo en ser ordenado y completo.

de campo del sistema de los números reales. En la sección 3.2, se formaliza los axiomas relacionados con la relación de orden. En la sección 3.3, se formaliza el axioma de completitud, además de los definiciones de cota superior, conjunto acotado y supremo de un conjunto. En la sección 3.4 se presentan algunas observaciones a las formalizaciones anteriores desde el punto de vista de la matemática intuicionista.

3.1. Formalización de los axiomas de campo

En el sistema de los números reales, denotado $\mathbb{R}$, se supone la existencia de dos operaciones binarias, adición $(+)$ y multiplicación $(\cdot)$ que satisfacen los siguientes axiomas:

- Propiedad conmutativa

  Para todo $a, b \in \mathbb{R}$: $a + b = b + a$ y $a \cdot b = b \cdot a$.

- Propiedad asociativa

  Para todo $a, b, c \in \mathbb{R}$: $(a + b) + c = a + (b + c)$ y $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

- Existencia del neutro aditivo

  Existe un elemento 0 de $\mathbb{R}$ tal que para todo $a \in \mathbb{R}$: $a + 0 = a$.

- Existencia del neutro multiplicativo

  Existe un elemento 1 de $\mathbb{R}$ tal que para todo $a \in \mathbb{R}$: $a \cdot 1 = a$.

- No trivialidad

  $$1 \neq 0.$$

- Existencia del negativo

  Para todo $a \in \mathbb{R}$ existe un elemento de $\mathbb{R}$ denotado $-a$ llamado negativo de a.

- Existencia del recíproco

  Para todo $a \in \mathbb{R}$, diferente de 0, existe un elemento de $\mathbb{R}$ denotado a^{-1} llamado recíproco de a.

- Inverso aditivo

  Para todo $a \in \mathbb{R}$ se cumple: $a + (-a) = 0$.

- Inverso multiplicativo

  Para todo $a \in \mathbb{R}$, diferente de 0, se cumple: $a \cdot a^{-1} = 1$.

- Propiedad distributiva

  Para todo $a, b, c \in \mathbb{R}$: $a \cdot (b + c) = a \cdot b + a \cdot c$.

Comentario. En el texto de Royden y Fitzpatrick [2010] se le llama axioma de la no trivialidad, al hecho de que 0 es diferente de 1. En el texto de Apostol [1974] este axioma está implícito al denotar $a + (-a)$ por 0 y $a \cdot a^{-1}$ por 1, para todo $a \in \mathbb{R}$ ($a \neq 0$ en el segundo caso).

La representación de constantes y funciones del sistema de los números reales se hace a partir de postular a `ℝ` como el tipo de los números reales, habitado por las constantes `r₀` y `r₁` (véase ejemplo 2.12), las cuales representan el cero y el uno respectivamente. También se postula las funciones adición, nombrada `_+_`, y multiplicación, nombrada `_*_`. Adicionalmente se postula las funciones `-_` y `_⁻¹` las cuales son la formalización del negativo y el recíproco respectivamente. Lo anterior es ilustrado en la figura 3.1.

```
postulate
  ℝ      : Set
  r₀ r₁  : ℝ

  _+_    : ℝ → ℝ → ℝ
  _*_    : ℝ → ℝ → ℝ
  -_     : ℝ → ℝ
  _⁻¹    : ℝ → ℝ
```

Figura 3.1: Tipos de las constantes y las funciones.

La propiedad conmutativa para la adición, nombrada `+-comm`, y la propiedad de la no trivialidad, nombrada `1≠0`, sirven para ilustrar los axiomas de campo formalizados en Agda.

```
postulate
  +-comm : (x y : ℝ) → x + y ≡ y + x
  1≠0    : r₁ ≢ r₀
```

Comentario. Empleando la palabra reservada `forall`, la cual puede también escribirse con el símbolo `∀`, y la capacidad de inferencia de tipos de Agda, el axioma `+-comm` podría reescribirse como

```
+-comm : ∀ (x y) → x + y ≡ y + x
```

Desde que la primera presentación hace explícito el tipo de las variables, esta será la presentación empleada en esta tesis.

La implementación de los axiomas de campo se encuentra en la figura 3.2.

```
-- Addition axioms.
postulate
  +-comm : (x y : ℝ)   →     x + y ≡ y + x
  +-asso : (x y z : ℝ) → x + y + z ≡ x + (y + z)
  +-neut : (x : ℝ)     →    x + r₀ ≡ x
  +-inve : (x : ℝ)     → x + (- x) ≡ r₀

-- Multiplication axioms.
postulate
  *-comm : (x y : ℝ)   →     x * y ≡ y * x
  *-asso : (x y z : ℝ) → x * y * z ≡ x * (y * z)
  *-neut : (x : ℝ)     →    x * r₁ ≡ x
  *-inve : (x : ℝ)     → x ≢ r₀ → x * (x ⁻¹) ≡ r₁

-- Distributivity axiom.
postulate *-dist : (x y z : ℝ) → x * (y + z) ≡ x * y + x * z

-- The nontriviality assumption.
postulate 1≠0 : r₁ ≢ r₀
```

Figura 3.2: Formalización de los axiomas de campo.

3.2. Formalización de los axiomas de orden

Se introducen los axiomas de orden con el fin de establecer una relación de orden dentro del sistema de los números reales. Para la relación binaria $>$

(mayor que) se introducen las propiedades de orden como un conjunto de axiomas que se describen a continuación:

- Propiedad transitiva

  Para todo $a, b, c \in \mathbb{R}$: si $a > b$ y $b > c$ entonces $a > c$.

- Adición por la izquierda

  Para todo $a, b, c \in \mathbb{R}$, si $a > b$ entonces $c + a > c + b$.

- Producto mayor que cero

  Para todo $a, b \in \mathbb{R}$, si $a > 0$ y $b > 0$ entonces $a \cdot b > 0$.

- Propiedad de tricotomía

  Si $a, b \in \mathbb{R}$ entonces una y sólo una de las siguientes afirmaciones es verdadera:

$$a > b,$$
$$a = b \text{ o}$$
$$a < b.$$

Al formalizar en Agda los axiomas de orden se introduce primero el tipo de la relación `_>_`:

```
postulate _>_ : ℝ → ℝ → Set
```

La relación binaria `_>_` crea un nuevo tipo en **`Set`**. Las relaciones binarias `_<_`, `_≥_` y `_≤_` se definen a partir de la relación `_>_`:

```
-- Less than relation.
_<_ : ℝ → ℝ → Set
y < x = x > y

-- Greater than or equal relation.
_≥_ : ℝ → ℝ → Set
x ≥ y = (x > y) ∨ (x ≡ y)

-- Less than or equal relation.
_≤_ : ℝ → ℝ → Set
x ≤ y = (x < y) ∨ (x ≡ y)
```

Uno de los axiomas de orden formalizados en Agda es la propiedad de tricotomía, nombrada `trichotomy`, la cual se define para elementos de tipo $\mathbb{R}$ de la siguiente forma:

```
postulate
  trichotomy : (x y : ℝ) → ((x > y) ∧ ¬ (x ≡ y) ∧ ¬ (x < y)) ∨
                           (¬ (x > y) ∧ (x ≡ y) ∧ ¬ (x < y)) ∨
                           (¬ (x > y) ∧ ¬ (x ≡ y) ∧ (x < y))
```

La propiedad `trichotomy` representa que para cualquier $x, y \in \mathbb{R}$ se satisface uno y sólo uno de los siguientes tres casos: i) $x > y$, $\neg(x \equiv y)$ y $\neg(x < y)$ son verdaderos; ii) $\neg(x > y)$, $x \equiv y$ y $\neg(x < y)$ son verdaderos; iii) $\neg(x > y)$, $\neg(x \equiv y)$ y $(x < y)$ son verdaderos.

La formalización en Agda del axioma de tricotomía fue un proceso donde se aprendió de los errores, debido a que se sortearon varios intentos de formalización hasta llegar a uno definitivo. Inicialmente se formalizó la propiedad de tricotomía implementando la disyunción inclusiva definida en el apéndice B. Pero esto no representó el "si y sólo si" y como tal la esencia de la propiedad de tricotomía.

Luego se procedió implementando la disyunción exclusiva (también llamada xor), definida mediante tres axiomas: propiedad conmutativa, propiedad asociativa y la propiedad donde cada elemento es inverso de sí mismo [Graham 2005]. La definición por medio de axiomas no fue adecuada para esta tesis, debido a que no tiene constructores y esto imposibilita la demostración de propiedades necesarias que se desprenden del axioma de tricotomía.

Luego se formalizó el axioma utilizando la disyunción exclusiva definida por

```
data _⊕_ (A B : Set) : Set where
   inj : (A ∨ B) → ¬ (A ∧ B) → A ⊕ B
```

Esta definición no fue apropiada para representar el axioma de tricotomía debido a que no es posible definir una disyunción exclusiva ternaria en términos de una disyunción exclusiva binaria [Pelletier y Hartline 2008]; es decir, en el caso binario se produce un valor de verdadero si uno de los argumentos es verdadero pero no ambos, y se puede demostrar que no se puede definir la disyunción ternaria exclusiva por agrupación de disyunciones binarias exclusivas.

Es posible formalizarla el axioma de tricotomía a partir de la disyunción ternaria exclusiva, la cual produce un valor de verdadero si un argumento es verdadero pero no los otros dos, definida por:

```
⊕³: Set → Set → Set → Set
⊕³ A B C = (A ∨ B ∨ C) ∧ ¬ (A ∧ B ∧ C)
```

No se usó la disyunción ternaria exclusiva porque su definición no está en lógica de primer orden, condición necesaria para utilizar Apia (capítulo 4).

Finalmente, la versión del axioma de tricotomía formalizada en esta tesis representa la forma estándar de esta propiedad. Se presentan en el ejemplo C.4 una serie de propiedades que garantizan que el axioma de tricotomía formalizado en esta tesis es apropiado.

Comentario. Mientras se estaba trabajando con la disyunción exclusiva `_⊕_` se encontró un error en el programa Apia. Este error se reportó en el *bug-tracker* de Apia.[2]

La implementación de los axiomas de orden se encuentra en la figura 3.3.

```
postulate
  >-trans  : {x y z : ℝ} → x > y  → y > z → x > z
  >-+-left : {x y z : ℝ} → x > y  → z + x > z + y
  >-∧-*    : {x y : ℝ}   → (x > r₀) ∧ (y > r₀) → x * y > r₀

  trichotomy : (x y : ℝ) → ((x > y) ∧ ¬ (x ≡ y) ∧ ¬ (x < y)) ∨
                           (¬ (x > y) ∧ (x ≡ y) ∧ ¬ (x < y)) ∨
                           (¬ (x > y) ∧ ¬ (x ≡ y) ∧ (x < y))
```

Figura 3.3: Formalización de los axiomas de orden.

3.3. Formalización del axioma de completitud

Hasta ahora se ha axiomatizado que el sistema de los números reales es un campo ordenado. Esta característica no lo distingue de otros campos

[2]Véase https://github.com/asr/apia/issues/98.

ordenados, como los números racionales, lo que lleva a enunciar el axioma de completitud. Antes de presentar esta propiedad, se define el concepto de conjunto acotado.

Se dice que un conjunto E de números reales está acotado superiormente cuando existe un número real b tal que $x \leq b$, para todo $x \in E$. El número b es llamado cota superior de E.

- Axioma de completitud: si E es un conjunto no vacío de números reales que está acotado superiormente, entonces existe para E una menor cota superior llamada supremo de E.

Para la formalización del axioma de completitud, inicialmente se formalizan en Agda las definiciones de: cota superior, conjunto acotado y supremo de un conjunto. Estas formalizaciones fueron adaptadas de la librería estándar de Coq.

La formalización de cota superior, nombrada `UpperBound`, está dada por:

```
UpperBound : (ℝ → Set) → ℝ → Set
UpperBound E ub = (x : ℝ) → E x → x ≤ ub
```

El predicado `E`, de tipo `ℝ → Set`, representa un subconjunto de los números reales. El término `ub`, de tipo `ℝ`, es una cota superior para `E` si se cumple `x ≤ ub`, para cualquier `x` de tipo `ℝ`.

La formalización de conjunto acotado, nombrada `Bound`, está dada por:

```
Bound : (ℝ → Set) → Set
Bound E = ∃ᵣ (λ ub → UpperBound E ub)
```

Si se tiene el predicado `E`, de tipo `ℝ → Set`, que representa un subconjunto de `ℝ`, entonces éste es acotado si existe un término `up`, de tipo `ℝ`, que satisface `UpperBound`. (La definición de `∃ᵣ` se encuentra en el ejemplo 2.3).

La menor cota superior de un conjunto dado, nombrada `sup`, debe cumplir la propiedad `Lub`:

```
Lub : (ℝ → Set) → ℝ → Set
Lub E sup = (UpperBound E sup) ∧ ((ub : ℝ) →
            UpperBound E ub → sup ≤ ub)
```

El predicado `E`, de tipo `ℝ → Set`, que representa un subconjunto de `ℝ`, tiene por supremo al término `sup`, de tipo `ℝ`, si éste satisface `UpperBound` y además,

para cualquier término `ub`, de tipo $\mathbb{R}$, que cumpla `UpperBound` se debe cumplir `sup ≤ ub`. (La definición de `_∧_` se encuentra en el ejemplo 2.2).

Igualmente adaptado de la librería estándar de Coq, se formaliza el axioma de completitud en Agda ilustrado en la figura 3.4.

```
postulate
  completeness : (E : ℝ → Set) →
                 Bound E → ∃ᵣ (λ x → E x) → ∃ᵣ (λ sup → Lub E sup)
```

Figura 3.4: Formalización del axioma de completitud.

Si se tiene un predicado `E`, de tipo `ℝ → Set`, el cual representa un subconjunto de los números reales, además éste es acotado `Bound E` y `E` es diferente del vacío `∃ᵣ (λ x → E x)`, entonces `E` tiene supremo `∃ᵣ (λ sup → Lub E sup)`.

Comentario. El axioma de completitud no está expresado en la lógica clásica de primer orden, debido a que el predicado `E` está precedido por un cuantificador universal, es decir, se está cuantificando sobre un predicado y no sobre una variable.

3.4. Una axiomática del sistema de los números reales en la matemática intuicionista

La axiomática presentada en las secciones anteriores es una axiomática del sistema de los números reales para la matemática clásica. Una de las características de la matemática intuicionista es no aceptar la ley del tercero excluido ni las demostraciones por contradicción. Una presentación axiomática de los números reales en la matemática intuicionista requiere i) adicionar una axioma correspondiente a la propiedad Arquimediana y ii) reemplazar el axioma de tricotomía.

La propiedad Arquimediana establece que para todo $x \in \mathbb{R}$ existe un número natural n tal que $n > x$. Cuando un campo ordenado, como el sistema de los números reales, cumple la propiedad Arquimediana, se está garantizando que los elementos de dicho campo no sean infinitesimales o infinitos.

Una demostración por contradicción de la propiedad Arquimediana es la siguiente [Rosenlicht 1968, pág. 26]:

Demostración. Sea $x \in \mathbb{R}$ el número que no satisface la propiedad Arquimediana, es decir $n \leq x$ para todo $n \in \mathbb{N}$. Por el axioma de completitud, el conjunto $\mathbb{N}$ tiene supremo, el cual será denotado por a. Sin embargo, para cualquier $n \in \mathbb{N}$, $n+1$ también es un número natural, por lo que $n+1 \leq a$ y por lo tanto $n \leq a-1$, lo que implica que $a-1$ es una cota superior para el conjunto $\mathbb{N}$. Por otro lado, puesto que $a-1 < a$, el número a no es el supremo de $\mathbb{N}$ y se obtiene una contradicción. □

En la matemática intuicionista para demostrar la existencia de un objeto que satisface cierta propiedad, debe resultar posible construir dicho objeto [Bridges 1999]. Esto contrasta con la idea que establece que la existencia de un objeto puede ser demostrada razonando por contradicción. Debido a que no es posible una demostración directa de la propiedad Arquimediana esto implica que en el desarrollo axiomático que se hace del sistema de los números reales en la matemática intuicionista, se incluya un axioma adicional correspondiente a la propiedad Arquimediana [Bridges 1999, pág. 103].

Al formalizar el sistema de los números naturales y el sistema de los números reales en Agda, resulta útil tener una función que permita trabajar con estos dos conjuntos, que reciba elementos de tipo `ℕ` y retorne elementos de tipo `ℝ`. La función que hace esto, nombrada `ℕ2ℝ`, está definida empleando *pattern matching*:

```
ℕ2ℝ : ℕ → ℝ
ℕ2ℝ zero     = r₀
ℕ2ℝ (succ n) = r₁ + ℕ2ℝ n
```

La representación de la propiedad Arquimediana está dada por:

```
postulate archimedean : (x : ℝ) → ∃ₙ (λ n → ℕ2ℝ n > x)
```

Comentario. El enunciado de la propiedad `archimedean` corresponde al enunciado presentado por varios autores [Ciaffaglione y Di Gianantonio 2002; Bridges 1999; Rosenlicht 1968], pero este enunciado es diferente al empleado por la librería estándar de Coq (véase sección 6.2).

Adicionalmente, en el desarrollo axiomático del sistema de los números reales, que se plantea en la matemática intuicionista, se sustituye el axioma de tricotomía por el axioma [Bridges 1999, pág. 102]:

$$x > y \to (x > z \vee z > y), \text{ para todo } x, y, z \in \mathbb{R}.$$

Esta sustitución se debe a que el axioma de tricotomía no es válido desde el punto de vista intuicionista [Ciaffaglione y Di Gianantonio 2002].

Para entrar en más detalle sobre el desarrollo axiomático del sistema de los números reales que se hace desde la matemática intuicionista véase [Ciaffaglione y Di Gianantonio 2002; Bridges 1999].

Capítulo 4

Demostraciones interactivas y automáticas

El campo del razonamiento automático se dedica a estudiar como usar un ordenador para ayudar a resolver problemas que requieren un razonamiento. Dentro de este campo se encuentra la demostración automática de teoremas, donde una demostración automática significa que no hay intervención en el proceso deductivo por parte del usuario, éste solo ingresa la propiedad a demostrar y espera que el computador le indique si pudo o no hacer la demostración, suponiendo que los programas y la propiedad se ingresen adecuadamente. Si el computador indica que la prueba fue demostrada no hay problema, pero si indica que no fue capaz por falta de tiempo esto no quiere decir que la propiedad sea falsa, se debe buscar otro programa u otra forma de razonar; como en el caso del ya mencionado razonamiento interactivo, donde el matemático hace parte fundamental del proceso deductivo y la computadora lo asiste.

En la sección 4.1 se presentan algunos ejemplos de demostraciones interactivas con base en la formalización realizada del sistema de los números reales. En la sección 4.2 se presentan los demostradores automáticos de teoremas y un lenguaje de entrada a éstos, este es lenguaje al que se traslada la representación que hace Agda de las fórmulas de primer orden. En la sección 4.3 se introduce una directriz del compilador empleada para hacer las demostraciones automáticas. En la sección 4.4 se introduce el programa Apia, un traductor de la representación que hace Agda de fórmulas de primer orden. Finalmente en la sección 4.5 se presentan algunos ejemplos de demostraciones automáticas y me mencionan algunas dificultades y limitaciones con éstas.

4.1. Demostraciones interactivas

Utilizando las constantes, definiciones, funciones y los axiomas de campo para el sistemas de los números reales se formaliza la demostración de algunas propiedades. La metodología para presentar los ejemplos de esta sección consiste en presentar inicialmente una prueba informal de la propiedad en referencia y después formalizar esta prueba vía una demostración interactiva.

Ejemplo 4.1. Se demuestra que

$$\text{si } x = y \text{ entonces } z + x = z + y, \text{ para todo } x, y, z \in \mathbb{R}. \tag{4.1}$$

Demostración. Se tiene como hipótesis $x = y$, entonces sumando por la izquierda a ambos lados de la hipótesis el número real z se tiene $z+y = z+y$, luego por propiedad reflexiva de la igualdad queda demostrado lo propuesto. □

Ahora se verifica la anterior demostración en Agda. La formalización de la demostración de (4.1), nombrada `≡-+-cong-r`, será realizada usando razonamiento ecuacional (véase sección 2.11) sobre los términos de tipo `ℝ`.

```
1  ≡-+-cong-r : {x y z : ℝ} → x ≡ y → x + z ≡ y + z
2  ≡-+-cong-r {x} {y} {z} h =
3    x + z ≡⟨ subst (λ w → (x + z) ≡ w + z) h (refl (x + z)) ⟩
4    y + z ∎
```

En la línea 3 se utiliza la propiedad de sustitución para términos de tipo `ℝ`, al considerar la sustitución se plantea la hipótesis `x + z ≡ w + z` y la hipótesis `x ≡ y` representada por `h`, además la propiedad `refl` de la igualdad que permite evidenciar que los términos a sustituir son iguales. En la línea 4 se muestra que el objetivo ha sido alcanzado.

Ejemplo 4.2. Se formaliza la demostración de la ley de simplificación para la suma, es decir, para todo $x, y, z \in \mathbb{R}$:

$$\text{si } x + z = y + z \text{ entonces } x = y.$$

Demostración. Utilizando los axiomas de campo y la hipótesis $x + z = y + z$

se tiene:

1	$x = x + 0$	por axioma del elemento neutro de la adición
2	$= x + (z + (-z))$	por axioma del inverso aditivo
3	$= (x + z) + (-z)$	por propiedad asociativa de la adición
4	$= (y + z) + (-z)$	por sustitución con la hipótesis
5	$= y + (z + (-z))$	por propiedad asociativa de la adición
6	$= y + 0$	por axioma del inverso aditivo
7	$= y$	□

La formalización de la demostración anterior en Agda se hace empleando los combinadores para el razonamiento ecuacional ilustrados en el ejemplo 4.1.

```
1  ≡-+-cancel-r : {x y z : ℝ} → x + z ≡ y + z → x ≡ y
2  ≡-+-cancel-r {x} {y} {z} h =
3    x               ≡⟨ ≡-sym (+-neut x) ⟩
4    x + r₀          ≡⟨ subst (λ w → (x + r₀) ≡ (x + w))
5                             (≡-sym (+-inve z))
6                             (refl (x + r₀)) ⟩
7    x + (z + (- z)) ≡⟨ ≡-sym (+-asso x z (- z)) ⟩
8    (x + z) + (- z) ≡⟨ subst (λ w → (x + z) + (- z) ≡ w + (- z))
9                             h
10                            (refl (x + z + - z)) ⟩
11   (y + z) + (- z) ≡⟨ +-asso y z (- z) ⟩
12   y + (z + (- z)) ≡⟨ subst (λ w → y + (z + (- z)) ≡ y + w)
13                            (+-inve z)
14                            (refl (y + (z + - z))) ⟩
15   y + r₀          ≡⟨ +-neut y ⟩
16   y               □
```

En la línea 3 se hace la prueba `x ≡ x + r₀` recurriendo a la propiedad simétrica de la igualdad `≡-sym` (definida en el apéndice A) y el neutro aditivo. En la línea 4 se hace la sustitución de `r₀` por `z + (- z)` utilizando la propiedad simétrica y el inverso aditivo. En la línea 7 se implementa la propiedad asociativa de derecha a izquierda, utilizando primero la propiedad simétrica. En la línea 9 se hace una sustitución utilizando la hipótesis `x + z ≡ y + z`. En la línea 10 se vuelve a utilizar la propiedad asociativa esta vez de izquierda a

derecha. En la línea 11 se sustituye `z + (- z)` por `r₀` utilizando la propiedad del inverso aditivo. Finalmente en la línea 14, implementando la propiedad del neutro aditivo se llega a `y + r₀ ≡ y`.

Ejemplo 4.3. Se demuestra que

$$x < x + 1, \text{ para todo } x \in \mathbb{R}. \tag{4.2}$$

Demostración. Se emplean la propiedad $1 > 0$ y la propiedad cancelativa de la adición, es decir,

$$\text{si } x + z > y + z \text{ entonces } x > y, \text{ para todo } x, y, z \in \mathbb{R}.$$

Sea $(y < x) = (x - y > 0)$ con $x - y = x + (-y)$ para todo $x, y \in \mathbb{R}$, lo que implica que si $x > y$ entonces $y < x$ para todo $x, y \in \mathbb{R}$, así la propiedad a demostrar es $x+1 > x$. Utilizando los axiomas de campo y de orden se tiene:

1	$1 > 0$	por propiedad $1 > 0$
2	$1 + 0 > 0$	por axioma del neutro aditivo
3	$1 + (x + (-x)) > x + (-x)$	por axioma del inverso aditivo
4	$(1 + x) + (-x) > x + (-x)$	por propiedad asociativa de la adición
5	$1 + x > x$	por propiedad cancelativa de la adición
6	$x + 1 > x$	por propiedad conmutativa de la adición □

Antes de la formalización en Agda de la demostración de 4.2, nombrada `x<x+1`, se debe demostrar primero las propiedades: $1 > 0$, la cancelativa de la adición y si $x > y$ entonces $y < x$ para todo $x, y \in \mathbb{R}$. Se puede observar una formalización en Agda de estas propiedades, nombradas `1>0`, `>-+-cancel-r` y `>-to-<` respectivamente, en el apéndice C, ejemplos C.1, C.2 y C.3.

Razonando igual que en la demostración de (4.2), se formaliza para $x+1 > x$, nombrada `x+1>x`, para lo cual se utiliza una serie de pasos auxiliares, llamados `helper`, donde el paso 1 será parte del último paso de la formalización, el paso 2 será parte del último y el penúltimo paso de la formalización y así sucesivamente hasta lograr el objetivo.

```
1  x+1>x : (x : ℝ) → x + r₁ > x
2  x+1>x x =
3    p₁-helper (>-+-cancel-r
4                (p₂-helper (p₃-helper (p₄-helper 1>0))))
5
6    where
7    p₁-helper : r₁ + x > x → x + r₁ > x
8    p₁-helper h = subst₂ (λ t₁ t₂ → t₁ > t₂
9                         (+-comm r₁ x)
10                        (refl x)
11                        h
12
13   p₂-helper : r₁ + (x + (- x)) > r₀ →
14               (r₁ + x) + (- x) > x + (- x)
15   p₂-helper h =
16     subst₂ (λ t₁ t₂ → t₁ > t₂) (≡-sym (+-asso r₁ x (- x)))
17               (≡-sym (+-inve x)) h
18
19   p₃-helper : r₁ + r₀ > r₀ → r₁ + (x + (- x)) > r₀
20   p₃-helper h =
21     subst₂ (λ t₁ t₂ → t₁ > t₂) (subst (λ w → r₁ + r₀ ≡ r₁ + w)
22               (≡-sym (+-inve x)) (refl (r₁ + r₀))) (refl r₀) h
23
24   p₄-helper : r₁ > r₀ → r₁ + r₀ > r₀
25   p₄-helper r₁>r₀ = subst₂ (λ t₁ t₂ → t₁ > t₂)
26                            (≡-sym (+-neut r₁))
27                            (refl r₀)
28                            r₁>r₀
```

En la línea 8 se formaliza `x + r₁ > x` utilizando la sustitución de doble parámetro `subst₂` (definida en el apéndice A), la propiedad conmutativa de la adición y la hipótesis `r₁ + x > x`. La hipótesis se convierte en el nuevo objetivo que se demuestra implementando la propiedad `>-+-cancel-r`, a su vez al aplicar esta propiedad se necesita una prueba de `(r₁ + x) + (- x) > x + (- x)` la cual se hace en la línea 16 utilizando sustitución de doble parámetro, simetría, asociatividad de la adición e inverso aditivo. Similarmente en la línea 16 tiene como hipótesis `r₁ + (x + (- x)) > r₀`, la cual se demuestra en la línea 21 implementando sustitución de doble parámetro, simetría, inverso

aditivo y reflexiva, donde en la línea 21 se tiene como hipótesis `r₁ + r₀ > r₀`, la cual se demuestra en la línea 25 utilizando sustitución de doble parámetro, simetría, neutro aditivo y reflexiva. Finalmente, la hipótesis `r₁ > r₀` en la línea 28 es la propiedad `1>0` formalizada previamente. Por último, se aplica la propiedad `>-to-<`:

```
x<x+1 : (x : ℝ) → x < x + r₁
x<x+1 x = >-to-< (x+1>x x)
```

Se presenta a continuación algunos ejemplos relacionados con el axioma de completitud.

Ejemplo 4.4. El conjunto vacío es acotado debido a que todo número real b es cota superior de este conjunto. Se presenta una formalización de esta afirmación.

El conjunto vacío se representa por:

```
∅ : ℝ → Set
∅ x = x ≠ x
```

donde `∅` es un predicado que representa al conjunto vacío, éste se refiere a la imposibilidad de construir los términos `x` que son diferentes así mismo.

La formulación que el conjunto vacío es acotado está dada por:

```
Bes : Bound ∅
Bes = exist r₁ (λ x ∅x → ⊥-elim (∅x (refl x)))
```

Para la demostración en Agda se utiliza la propiedad de conjunto acotado `Bound`, la cual recibe el predicado `∅` para el cual existe un término `r₁`, de tipo `ℝ`, que satisface la propiedad `UpperBound`, además para cualquier término `x`, de tipo `ℝ`, que cumple `∅ x` se cumple `x ≤ r₁`. En el desarrollo de la demostración el constructor `exist` recibe un término de tipo `ℝ`, en este caso `r₁`, luego para cualquier `x`, de tipo `ℝ`, que cumpla `∅ x` se debe demostrar `x ≤ r₁`, esto se hace utilizando `⊥-elim` donde se logra la contradicción mediante `x ≡ x`, por propiedad `refl`, y `x ≠ x` por definición de conjunto vacío.

No se puede aplicar el axioma de completitud al conjunto vacío para demostrar que éste tiene supremo, debido a que este axioma exige que el conjunto sea diferente de vacío.

Ejemplo 4.5. El axioma de completitud garantiza que todo conjunto de números reales, diferente de vacío y acotado superiormente, tiene supremo.

Sea A un subconjunto de los números reales definido por:

$$A = \{x \in \mathbb{R} \mid x \leq 2\}.$$

El conjunto A es diferente de vacío y acotado superiormente por 2. Por axioma de completitud el conjunto A tiene supremo.

Se formaliza en Agda empleando `UpperBound`, `Bound` y el `completeness` la afirmación anterior. El conjunto A es definido por:

```
A : ℝ → Set
A x = x ≤ ℕ2ℝ 2
```

La formalización que el número 2 es cota superior del conjunto A está dada por:

```
UBA : UpperBound A (ℕ2ℝ 2)
UBA x (inj₁ x<2) = inj₁ x<2
UBA x (inj₂ x≡2) = inj₂ x≡2
```

La formulación que el conjunto A es acotado está dada por:

```
BA : Bound A
BA = exist (ℕ2ℝ 2) UBA
```

Y finalmente la formalización que el conjunto A tiene supremo está dada por:

```
LA : ∃ᵣ (λ sup → Lub A sup)
LA = completeness A BA (exist (ℕ2ℝ 2) (inj₂ (refl (ℕ2ℝ 2))))
```

El axioma de completitud garantiza la existencia del supremo, para conjuntos no vacíos y acotados, pero no indica quien es el supremo del conjunto. Sin embargo, en algunos casos se puede conocer el supremo de un conjunto.

Ejemplo 4.6. Para el conjunto A del ejemplo 4.5 se demuestra en Agda que 2 es el supremo de este conjunto.

```
UBAx : (x : ℝ) → UpperBound A x → ℕ2ℝ 2 ≤ x
UBAx x A2 = A2 (ℕ2ℝ 2) (inj₂ (refl (ℕ2ℝ 2)))

LbA : ∃ᵣ (λ x → Lub A (ℕ2ℝ 2))
LbA = exist (ℕ2ℝ 2) (UBA , UBAx)
```

Se presenta a continuación algunas demostraciones empleando el principio de inducción matemática.

Ejemplo 4.7. En matemáticas se realizan demostraciones de propiedades por el principio de inducción matemática cuando se tienen conjuntos inductivos. Un conjunto A se dice inductivo si y sólo si i) $0 \in A$ y ii) si $a \in A$ entonces $a' \in A$, donde a' es el sucesor de a. Los números naturales son un conjunto inductivo que está contenido en cualquier conjunto inductivo [Enderton 1977, pág. 68].

Se demuestra por el principio de inducción matemática la propiedad $1^n = 1$, para cualquier número natural n. Luego se formaliza la demostración razonando de forma interactiva en Agda.

Se define en Agda la potenciación como una función recursiva, nombrada `_^_` de la siguiente forma:

```
_^_ : ℝ → ℕ → ℝ
x ^ zero     = r₁
x ^ (succ n) = x * (x ^ n)
```

En la definición de la función `_^_ : ℝ → ℕ → ℝ` se sigue la convención que $0^0 = 1$, véase por ejemplo, Knuth [1992] o la definición en algunos asistentes de prueba tales como Coq.

Se asumen las propiedades $1^1 = 1$ y $x^{m+n} = x^m \cdot x^n$, para todo número real x y números naturales m y n. Se puede observar una demostración de estas propiedades en el repositorio.

Demostración. Primero, base de la inducción, se procede a mostrar la validez de la propiedad para cuando $n = 0$. Si se observa la definición de potenciación se confirma tal afirmación.

Segundo, paso inductivo para $n = k+1$, por lo cual la hipótesis inductiva está dada por $1^k = 1$.

$$\begin{aligned} 1^{k+1} &= 1^k \cdot 1 && \text{por propiedad de la potenciación} \\ &= 1 \cdot 1 && \text{por hipótesis inductiva} \\ &= 1 && \text{por axioma del neutro multiplicativo} \end{aligned}$$ □

Se presenta a continuación la formalización de la demostración de la propiedad $1^n = 1$ de manera interactiva en Agda.

```
1   1^n : (n : ℕ) → r₁ ^ n ≡ r₁
2   1^n zero     = refl r₁
3   1^n (succ n) = p-helper
4
5     where
6     p-helper : r₁ * r₁ ^ n ≡ r₁
7     p-helper =
8       r₁ * r₁ ^ n ≡⟨ *-comm r₁ (r₁ ^ n) ⟩
9       r₁ ^ n * r₁ ≡⟨ *-neut (r₁ ^ n) ⟩
10      r₁ ^ n      ≡⟨ 1^n n ⟩
11      r₁          □
```

En la línea 1 se muestra el objetivo que se quiere demostrar, `r₁ ^ n ≡ r₁`, para todo `n : ℕ`. En la línea 2 se demuestra para el caso cuando `n = zero` utilizando la propiedad de la igualdad `refl`. En la línea 3 se demuestra para el caso sucesor `n = succ n` (paso inductivo) la propiedad `r₁ * r₁ ^ n ≡ r₁`, nombrada `p-helper`, utilizando razonamiento ecuacional (véase sección 2.11).

Ejemplo 4.8. Se demuestra la convergencia de una serie por el principio de inducción matemática.

Se demostrará que $S = 1 + 3 + 5 + \cdots + 2n - 1 = n^2$, para todo número natural n mayor que cero.

Demostración. Primero, base de la inducción. Si $n = 1$, se tiene $1 = 1$, por lo que queda demostrado el paso base.

Segundo, paso inductivo para $n = k + 1$, para todo k natural mayor que cero se debe demostrar:

$$1 + 3 + 5 + \cdots + 2k - 1 + (2(k + 1) - 1) = (k + 1)^2.$$

Inicialmente se supone verdadera la propiedad para el caso $n = k$ (hipótesis inductiva):

$$1 + 3 + 5 + \cdots + 2k - 1 = k^2.$$

Luego se procede a sumar $2(k+1)-1$ a ambos lados de la hipótesis inductiva:

$$1 + 3 + 5 + \cdots + 2k - 1 + (2(k + 1) - 1) = k^2 + (2(k + 1) - 1).$$

Utilizando la propiedad distributiva por la izquierda al lado derecho de la igualdad se tiene:

$$1 + 3 + 5 + \cdots + 2k - 1 + (2(k + 1) - 1) = k^2 + 2k + 2 - 1.$$

Finalmente, se aplica la propiedad del trinomio cuadrado perfecto al lado derecho de la igualdad:

$$1 + 3 + 5 + \cdots + 2k - 1 + (2(k+1) - 1) = (k+1)^2. \qquad \square$$

Para formalizar la demostración, inicialmente se define la serie. Desde que es necesario incluir el `zero` en la definición por *pattern matching* sobre los números natural, la serie S se define recursivamente por:

```
S : ℕ → ℝ
S zero     = r₀
S (succ n) = S n + (ℕ2ℝ 2) * (ℕ2ℝ n) + r₁
```

La formalización de la demostración de la convergencia de la serie `S` se presenta a continuación.

```
S-con : (n : ℕ) → S n ≡ sqr (ℕ2ℝ n)
S-con zero = p-helper
  where
  p-helper : S zero ≡ sqr (ℕ2ℝ zero)
  p-helper =
    S zero         ≡⟨ refl r₀ ⟩
    r₀             ≡⟨ ≡-sym r₀-sqr ⟩
    sqr (ℕ2ℝ zero) □
```

```
S-con (succ n) = p-helper
  where
  p-helper : S (succ n) ≡ sqr (ℕ2ℝ (succ n))
  p-helper =
    S (succ n)
      ≡⟨ refl (S n + ℕ2ℝ 2 * ℕ2ℝ n + r₁) ⟩
    S n + ℕ2ℝ 2 * ℕ2ℝ n + r₁
      ≡⟨ subst (λ w → S n + ℕ2ℝ 2 * ℕ2ℝ n + r₁ ≡
                  w + ℕ2ℝ 2 * ℕ2ℝ n + r₁)
               (S-con n)
               (refl (S n + ℕ2ℝ 2 * ℕ2ℝ n + r₁))
       ⟩
    sqr (ℕ2ℝ n) + ℕ2ℝ 2 * ℕ2ℝ n + r₁
      ≡⟨ ≡-sym (PST1 (ℕ2ℝ n)) ⟩
    sqr (ℕ2ℝ n + r₁)
      ≡⟨ subst (λ w → sqr (ℕ2ℝ n + r₁) ≡ sqr w)
               (+-comm (ℕ2ℝ n) r₁)
               (refl (sqr (ℕ2ℝ n + r₁)))
       ⟩
    sqr (r₁ + ℕ2ℝ n) □
```

Las definiciones y propiedades no mencionadas en este ejemplo se encuentran en el repositorio.

4.2. Demostradores automáticos de teoremas y el lenguaje TPTP

Los demostradores automáticos de teoremas son programas que implementan métodos y técnicas para demostrar propiedades y teoremas de diferentes teorías formales. Los demostradores automáticos de teoremas son empleados por ejemplo en la industria para el diseño y la verificación de circuitos integrados. De ahora en adelante estos programas serán llamados 'ATPs'. Este nombre proviene de las siglas de su nombre en inglés (*Automated Theorem Provers*).

La TPTP (Thousands of Problems for Theorem Provers) es una biblioteca de problemas para sistemas que prueban teoremas automáticamente para la lógica clásica [Sutcliffe 2010]. El lenguaje TPTP hace parte de esta biblioteca.

Adicionalmente este lenguaje es soportado por varios de los ATPs, por lo tanto a éste se traslada la representación que hace Agda de las fórmulas de primer orden.

La sintaxis de las formulas anotadas del lenguaje TPTP es

```
fof(name, role, formula)
```

donde `name` identifica la fórmula dentro del conjunto de formulas, `formula` es una fórmula de la lógica clásica de primer orden con identidad y `role` puede ser uno de los siguientes: `axiom`, `definition`, `hypothesis` o `conjecture`. Para el lenguaje TPTP la conjetura es la fórmula a ser demostrada. Los axiomas son las propiedades que se asumen como verdaderas y son la base para demostrar las conjeturas. Las hipótesis son propiedades que se han demostrado previamente y se comportan como un axioma para los ATPs. Las definiciones se utilizan para introducir nuevos símbolos.

4.3. El ATP-pragma

En Agda, al igual que en otros lenguajes de programación, es posible escribir un tipo especial de comentario, en particular unos llamados directivas del compilador o *pragmas*. Los *pragmas* son leídos e interpretados por el compilador de Agda, de esta manera el *pragma* modifica el comportamiento de Agda.

Ejemplo 4.9. En Agda se pueden introducir los números naturales empleando la declaración `data` como fue ilustrado en el ejemplo 2.1.

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

Al adicionar el *pragma*

```
{-# BUILTIN NATURAL ℕ #-}
```

el compilador de Agda sabe que los números literales deben tener su correspondencia con los constructores del tipo de datos ℕ. Por ejemplo, el número literal 0 será interpretado como `zero` y el número literal 1 será interpretado como `succ zero`.

En esta tesis se emplea un *pragma* para Agda llamado el ATP-pragma. Este *pragma* adiciona información relacionada con el lenguaje TPTP a los archivos generados por Agda (archivos con extensión `.agdai`). Estos archivos son interpretados por el programa Apia. El ATP-pragma es la manera de informarle a los ATPs, a través del programa Apia, que una determinada fórmula es un axioma, una definición, una hipótesis o una conjetura.

La sintaxis del ATP-pragma es la siguiente:

```
{-# ATP rol formula #-}
```

donde `rol` corresponde a los diferentes roles del lenguaje TPTP y `formula` es una formula de la lógica de predicados de primer orden definida en Agda.

La forma de indicar que las formulas `A` y `B` deberán ser consideradas como axiomas por los ATPs es:

```
{-# ATP axiom A #-}
{-# ATP axiom B #-}
```

Se puede utilizar la sintaxis alternativa:

```
{-# ATP axioms A B #-}
```

La forma de indicar que se emplearán los ATPs para intentar realizar una demostración automática de una fórmula postulada `A` es la siguiente:

```
{-# ATP prove A #-}
```

También se le puede indicar a los ATPs que empleen como hipótesis adicionales las formulas $\texttt{h}_1$ y $\texttt{h}_2$ cuando intenten demostrar la formula `A` incluyendo estas hipótesis en el ATP-pragma:

```
{-# ATP prove A h₁ h₂ #-}
```

4.4. EL programa Apia

Apia es un programa que traslada la representación que realiza Agda de las fórmulas de primer orden al lenguaje TPTP. Además este programa es una interfase entre Agda y los ATPs. Apia interactúa con los ATPs preguntándoles por pruebas, entregándoles hipótesis e informándoles sobre definiciones y axiomas. Como resultado de esta interacción el programa Apia llama a los

ATPs elegidos por el usuario para que demuestren propiedades automáticamente y le informa al usuario el resultado de esta llamada. Al momento de escribir estas tesis, sólo es posible demostrar automáticamente teoremas escritos en la lógica de primer orden empleando Apia.

El programa Apia soporta los siguientes ATPs, los cuales soportan el lenguaje TPTP y pueden emplearse para realizar demostraciones automáticas: CVC4, Eprover, Equinox, Metis, Spass y Vampire. En esta tesis se trabajó con dos ATPs, Eprover (versión E 1.9.1-001 Sungma) [Schulz 2013] y Vampire (versión 0.6 (revisión 903)) [Kovác y Voronkov 2013], ya que éstos fueron suficientes para demostrar las propiedades propuestas.

Para intentar demostrar automáticamente un formula se realizan los siguientes pasos: i) adicionar los ATP-pragmas necesarios al archivo con la formalización en Agda, ii) *type-check* el archivo empleando Agda, iii) ejecutar Apia en el archivo con la formalización en Agda seleccionado los ATPs de preferencia y iv) verificar si los ATPs seleccionados demostraron o no la formula.

Durante el proceso de demostración automática Apia llama indistintamente a los ATPs seleccionados por el usuario. Si uno de los ATPs demuestra la propiedad, entonces el proceso se detiene y Apia informa cual ATP realizó la demostración. Si al finalizar el tiempo asignado a los ATPs ninguno de ellos pudo demostrar la propiedad, entonces Apia le informa al usuario esta situación. Una situación que se puede presentar es que sí ninguno de los ATPs seleccionados demostró la propiedad, puede ser que otro ATP soportado por Apia la pueda demostrar. En algunos casos resulta necesario agregar propiedades demostradas previamente como hipótesis auxiliares a la propiedad que se desea demostrar.

Por ejemplo, para demostrar automáticamente la propiedad conmutativa de la disyunción, inicialmente se escribe

```
postulate
  A B    : Set
  v-comm : A ∨ B → B ∨ A
{-# ATP prove v-comm #-}
```

en un archivo Agda llamado `CommDisjunction.agda`. Este archivo se compila escribiendo lo siguiente:

```
$ agda CommDisjunction.agda
```

En la pantalla se muestra un mensaje que indica que se ha generado un archivo con extensión `.agdai` que será interpretado por Apia. A continuación se ejecuta Apia indicándole que use los ATPs E y Vampire.

```
$ apia --atp=e --atp=vampire CommDisjunction.agda
```

En la pantalla se muestra un mensaje que indica que se ha generado un archivo en lenguaje TPTP y su locación en un directorio temporal. Además brinda información acerca de la propiedad que fue demostrada y también aparece una descripción sobre el primero de los ATPs en realizar la demostración.

```
Proving the conjecture in /tmp/CommDisjunction/10-8744-comm.tptp
Vampire 0.6 (revision 903) proved the conjecture
```

El archivo TPTP es creado en el directorio `/tmp/CommDisjunction`. La propiedad `v-comm` es renombrada como `10-8744-comm`, el número `10` indica la línea donde la propiedad es postulada, `8744` es el código decimal del símbolo Unicode '`∨`' y Vampire fue el primero de los ATPs en demostrar la conjetura.

4.5. Demostraciones automáticas

Se presenta la forma como se realizan demostraciones automáticas de propiedades del sistema de los números reales. Además se mencionan algunas limitaciones en el uso de este tipo de demostraciones.

Puesto que la presentación del sistema de números reales fue axiomática, el primer paso para emplear los ATPs por medio del programa Apia es emplear el ATP-pragma para indicarle a los ATPs cuales axiomas deben usar.

```
{-# ATP axiom +-comm +-asso +-neut +-inve 1≠0 #-}
{-# ATP axiom *-comm *-asso *-neut *-inve *-dist #-}
{-# ATP axiom >-trans >-+-left >-∧-* trichotomy #-}
```

También se debe emplear el ATP-pragma para señalar las definiciones empleadas en la formalización.

```
{-# ATP definition _≠_ #-}
{-# ATP definition _<_ #-}
{-# ATP definition _≥_ #-}
{-# ATP definition _≤_ #-}
```

Ejemplo 4.10. La propiedad `≡-+-cong-r` fue demostrada de forma interactiva en el ejemplo 4.1. Para demostrar automáticamente esta propiedad, se postula ésta y se emplea el ATP-pragma para indicarle a los ATPs que la intenten demostrar.

```
postulate ≡-+-cong-r : {x y z : ℝ} → x ≡ y → x + z ≡ y + z
{-# ATP prove ≡-+-cong-r #-}
```

De forma similar se demuestra automáticamente la ley de simplificación para la suma presentada en el ejemplo 4.2.

```
postulate ≡-+-cancel-r : {x y z : ℝ} → x + z ≡ y + z → x ≡ y
{-# ATP prove ≡-+-cancel-r #-}
```

Se puede observar en el ejemplo anterior la simplificación considerable en la formalización de las propiedades cuando en lugar de realizar una demostración interactiva, se realiza una demostración automática.

Ejemplo 4.11. Una posible demostración automática de la convergencia de la serie demostrada interactivamente en el ejemplo 4.8 tendría la siguiente forma:

Primero, base de la inducción `S zero`:

```
S-con : (n : ℕ) → S n ≡ sqr (ℕ2ℝ n)
S-con zero = p-helper
  where
  postulate p-helper : S zero ≡ sqr (ℕ2ℝ zero)
   {-# ATP prove p-helper #-}
```

Segundo, paso inductivo `S-con (succ n)`:

```
S-con (succ n) = p-helper
  where
  postulate p-helper : S (succ n) ≡ sqr (ℕ2ℝ (succ n))
  {-# ATP prove p-helper #-}
```

Se observa que la demostración es por inducción, pero se emplean los ATPs en los pasos base e inductivo. Desafortunadamente no es posible emplear este tipo de demostraciones automáticas por las razones que se exponen a continuación.

La demostración automática ilustrada en el ejemplo 4.11 es diferente a las demostraciones automáticas presentadas en los ejemplos anteriores desde el punto de vista lógico. En esta demostración se emplean dos tipos/universos diferentes (el universo de los números reales y el universo de los números naturales). En la actualidad el programa Apia no soporta lógicas con varios universos.

Capítulo 5

Límites

En este capítulo se emplean los elementos presentados hasta el momento en esta tesis para ilustrar la formalización de un concepto de uso cotidiano en las matemáticas: límite de una función.

La noción intuitiva de límite indica que los valores que toma una función dentro de un intervalo se van aproximando arbitrariamente a un punto fijo, independiente que éste pertenezca o no al dominio de la función. Para formalizar el concepto de límite de una función se procede primero a formalizar el valor absoluto de un número real, la distancia entre dos puntos y demostrar algunas de sus propiedades.

5.1. Valor absoluto

Si x es un número real entonces el valor absoluto de x es un número real no negativo, el cual se designa por $|x|$ y se define de la siguiente forma:

$$|x| = \begin{cases} x & \text{si } x \geq 0, \\ -x & \text{si } x < 0. \end{cases}$$

En Agda se formaliza la definición de valor absoluto de la siguiente forma:

```
abs : ℝ → ℝ
abs x with x<0vx≥0 x
... | inj₁ _ = - x
... | inj₂ _ = x
```

donde

```
x<0∨x≥0 : (x : ℝ) → (x < r₀) ∨ (x ≥ r₀)
```

La formalización de $|x|$ se hace implementando una función, nombrada `abs` de tipo `ℝ → ℝ`, la cual si recibe un argumento `x` que cumple `x < 0` entonces devuelve `- x`, de lo contrario devuelve `x`.

Algunas de las propiedades que cumple el valor absoluto, para todo número real x e y, son las siguientes:

$$\begin{aligned} |0| &= 0, \\ \text{si } x > 0 \text{ entonces } |x| &= x, \\ \text{si } x < 0 \text{ entonces } |x| &= -x, \\ |-x| &= |x|, \\ |x| &\geq x, && (5.1) \\ -|x| &\leq x, && (5.2) \\ \text{(desigualdad triangular) } |x+y| &\leq |x| + |y|, && (5.3) \\ |x-y| &\leq |x| + |y|. \end{aligned}$$

Ejemplo 5.1. Se demuestra la desigualdad triangular (propiedad 5.3) para el valor absoluto.

Demostración. A partir de las propiedades (5.1) y (5.2), para $x, y \in \mathbb{R}$, se tiene

$$\begin{aligned} -|x| \leq x \leq |x|, \\ -|y| \leq y \leq |y|. \end{aligned}$$

Luego, sumando miembro a miembro estas desigualdades se obtiene:

$$-|x| - |y| \leq x + y \leq |x| + |y|.$$

lo que es igual a:

$$-(|x| + |y|) \leq x + y \leq |x| + |y|.$$

Finalmente por la definición de valor absoluto se tiene:

$$|x + y| \leq |x| + |y|. \qquad \square$$

La formalización de la demostración en Agda de la desigualdad triangular, nombrada `abs-tri`, está dada dada por:

```
abs-tri : (x y : ℝ) → abs (x + y) ≤ abs x + abs y
abs-tri x y with x<0vx≥0 (x + y)
... | inj₁ p = helper

  where
  helper : - (x + y) ≤ abs x + abs y
  helper = p₁-helper (≤-to-≥ (p₂-helper (-absx+-absy≤x+y x y)))

    where
    p₁-helper : x + y ≥ - (abs x + abs y) →
                          - (x + y) ≤ abs x + abs y
    p₁-helper h₁ = x≥-y→-x≤y (x + y) (abs x + abs y) h₁

    p₂-helper : - (abs x) + ( - abs y) ≤ x + y →
                - (abs x + abs y) ≤ x + y
    p₂-helper h₂ = subst₂ (λ t₁ t₂ → t₁ ≤ t₂)
                          (≡-sym mul-x+y)
                          (refl (x + y))
                          h₂
... | inj₂ _ = x+y≤absx+absy x y
```

Como la formalización de valor absoluto emplea `with` sobre la propiedad `x<0vx≥0`, entonces se da una reducción por la izquierda. Por esta razón se generan dos objetivos:

```
... | inj₁ p = helper
... | inj₂ _ = x+y≤absx+absy x y
```

El primer objetivo es demostrar la propiedad nombrada `helper`, la cual está definida por:

```
helper : - (x + y) ≤ abs x + abs y
```

Esta propiedad a su vez requiere la demostración de las propiedades nombradas `p₁-helper` y `p₂-helper`. El segundo objetivo consiste en demostrar la propiedad

```
x+y≤absx+absy : (x y : ℝ) → x + y ≤ abs x + abs y
```

Una prueba de esta propiedad se encuentra en el repositorio.

La formalización de la demostración en Agda de las demás propiedades de valor absoluto mencionadas al comienzo de esta sección se encuentran en el repositorio.

En la sección 4.5 se señaló una limitación para realizar pruebas automáticas cuando se emplean varios universos en la formalización. Las pruebas relacionadas con valor absoluto no pudieron ser automatizadas debido a una razón diferente. En lo relacionado con el uso de las definiciones en una formalización cualquiera, Apia sólo puede trasladar al lenguaje TPTP las definiciones aceptadas por la lógica de predicados de primer orden. De otra parte, Agda al ser un lenguaje de programación soporta definiciones de mayor complejidad incluyendo definiciones de funciones tal como es ilustrado en la definición de la función valor absoluto.

5.2. Distancia entre dos puntos

Si x es un número real, la distancia de x a 0 es el número $|x|$. De igual forma, si x e y son números reales, la distancia entre x e y es el número $|x - y|$.

Para todo número real x, y y z, la función distancia, nombrada d, satisface las siguientes propiedades:

$$\begin{aligned} d(x, y) &\geq 0, \\ d(x, y) &= d(y, x), \\ d(x, y) = 0 &\leftrightarrow x = y, \\ d(x, y) &\leq d(x, z) + d(z, y). \end{aligned}$$

Se formaliza en Agda la definición de distancia entre dos puntos, nombrada `dist`, a partir de la definición de valor absoluto:

```
dist : ℝ → ℝ → ℝ
dist x y = abs (x - y)
```

Ejemplo 5.2. Demostrar que para cualquier número real x, y, z y w se cumple:

$$d(x + z, y + w) \leq d(x, y) + d(z, w). \tag{5.4}$$

Demostración. Se parte de la desigualdad triangular donde el primer número está dado por $x - y$ y el segundo número por $z - w$, es decir:

$$|(x - y) + (z - w)| \leq |x - y| + |z - w|.$$

Por definición de distancia entre dos puntos, el lado derecho de la desigualdad anterior es equivalente al mismo lado de la propiedad a demostrar, por lo que basta con probar la igualdad:

$$|(x - y) + (z - w)| = d(x + z, y + w).$$

Reagrupando los términos del lado izquierdo de la igualdad se obtiene:

$$|(x - y) + (z - w)| = |(x + z) - (y + w)|.$$

Finalmente por definición de distancia:

$$|(x + z) - (y + w)| = d(x + z, y + w).$$

Queda así demostrada la propiedad. □

La formalización de la demostración de (5.4), nombrada `dis-des`, está dada por:

```
dis-des : (x y z w : ℝ) →
          dist (x + z) (y + w) ≤ dist x y + dist z w
dis-des x y z w = p₁-helper (abs-tri (x - y) (z - w))
```

A partir de la desigualdad triangular evaluada en los números `(x - y)` y `(z - w)` se demuestra la propiedad `p₁-helper`, la cual indica:

```
p₁-helper : abs ((x - y) + (z - w)) ≤ abs (x - y) + abs (z - w) →
            dist (x + z) (y + w) ≤ dist x y + dist z w
p₁-helper h =
  subst₂ (λ t₁ t₂ → t₁ ≤ t₂)
         p₁₁-helper
         (refl (abs (x - y) + abs (z - w)))
         h
```

Se observa que el lado derecho de la desigualdad es equivalente por lo que se implementa la propiedad `refl`. Luego se demuestra la propiedad nombrada `p₁₁-helper` reagrupando términos, lo que se logra implementando razonamiento ecuacional.

```
p₁₁-helper : abs (x - y + (z - w)) ≡ dist (x + z) (y + w)
```

La formalización completa de esta propiedad se encuentra en el repositorio.

Ejemplo 5.3. A diferencia de las propiedades del valor absoluto, las propiedades de la función distancia pudieron ser demostradas automáticamente.

```
postulate d-refl : (x y : ℝ) → (dist x y ≡ r₀) ↔ (x ≡ y)
{-# ATP prove d-refl abs-0 x>0→absx=x x<0→absx=-x #-}

postulate d-pos : (x y : ℝ) → dist x y ≥ r₀
{-# ATP prove d-pos abs-0 x>0→absx=x x<0→absx=-x #-}

postulate d-sym : (x y : ℝ) → dist x y ≡ dist y x
{-# ATP prove d-sym abs-0 x>0→absx=x x<0→absx=-x #-}

postulate d-tri : (x y z : ℝ) → dist x y ≤ dist x z + dist z y
{-# ATP prove d-tri abs-tri #-}

postulate
  dis-des : (x y z w : ℝ) →
            dist (x + z) (y + w) ≤ dist x y + dist z w
{-# ATP prove dis-des abs-tri #-}
```

Las hipótesis auxiliares empleadas en las demostraciones anteriores se encuentran formalizadas en el repositorio.

5.3. Límite de una función

Sea f una función definida sobre algún intervalo abierto que contiene al número a, excepto posiblemente en a misma. Se dice entonces que el límite de $f(x)$ cuando x tiende a a es L, si para todo número $\varepsilon > 0$ existe un número $\delta > 0$ tal que si la distancia de x a a es menor que δ, entonces la distancia de $f(x)$ a L es menor que ε.

La formalización de la definición del límite, nombrado `lim`, de una función `f`, de tipo `ℝ → ℝ`, fue adaptada de la librería estándar de Coq.

```
lim : (ℝ → ℝ) → ℝ → ℝ → Set
lim f a L = (ε : ℝ) → ε > r₀ →
            ∃ᵣ (λ δ → δ > r₀ → (x : ℝ) →
              dist x a < δ → dist (f x) L < ε)
```

Ejemplo 5.4. Demostración de la propiedad que indica que el límite de una función constante es la constante.

Demostración. El límite cuando x tiende a a de la función $f(x) = k$ es igual a k. Se tiene entonces que para todo $\varepsilon > 0$ existe un $\delta > 0$ tal que la distancia de x a a es menor que δ, entonces la distancia de $f(x)$ a k es menor que ε. Como la distancia de k ($f(x) = k$) a k es cero se cumple que $\varepsilon > 0$, entonces se concluye que para todo $\delta > 0$ el límite existe. □

Se formaliza en Agda la demostración de la propiedad, nombrada `cte-lim`, que indica que el límite de una función `f` constante, de tipo `ℝ → ℝ`, es igual a una constante `k` para todo `x` cuando este tiende a `a`, donde `a`, `L` y `k` son de tipo `ℝ`.

```
cte-lim : (a L k : ℝ) → (f : ℝ → ℝ) → lim (λ _ → k) a k
cte-lim a L k f ε ε>0 = exist ε p-helper
```

La demostración emplea el constructor `exist` el cual requiere un testigo `ε` y la prueba de la propiedad `p-helper`:

```
1  p-helper : ε > r₀ → (x : ℝ) → dist x a < ε → dist k k < ε
2  p-helper ε>0 x h = subst₂ (λ t₁ t₂ → t₁ < t₂)
3                           (≡-sym (∧-proj₂ (d-refl k k)
4                           (refl k)
5                           (refl ε)
6                           (>-to-< ε>0)
```

En la línea 1 se muestra el objetivo que se quiere lograr, `dist k k < ε`, donde se tienen las hipótesis: `ε>0`, `(x : ℝ)` y `dist x a < ε`. En la línea 2, se realiza la demostración de la propiedad `p-helper` donde se utiliza: la sustitución de doble parámetro `subst₂`, la propiedad de la igualdad `≡-sym`, la proyección `∧-proj₂`, la propiedad de la igualdad `refl` y la propiedad `>-to-<`.

Como fue mencionado con anterioridad, Apia sólo soporta la lógica de primer orden, es decir, Apia no soporta lógicas de orden superior. La definición del límite de una función no está expresada en la lógica de primer orden, debido a que ésta es una función de orden superior (su primer argumento es una función). Por esta razón, la demostraciones acerca de las propiedades de los límites no se pueden realizar de forma automática empleando la metodología presentada en esta tesis.

Otras demostraciones formalizadas en Agda de las propiedades de los límites se encuentran en el repositorio

Capítulo 6

Conclusiones

En esta tesis se se desarrolló una formalización de las demostraciones de propiedades matemáticas en el sistema de los números reales, demostrando de forma interactiva con el programa Agda y demostrando automáticamente empleando el programa Apia y algunos demostradores automáticos de teoremas. A partir del trabajo realizado se obtuvieron los siguientes resultados:

6.1. Resultados

- Se presentó de forma axiomática el sistema de los números reales.
- Se utilizó el asistente de pruebas Agda para realizar demostraciones interactivas de propiedades matemáticas, por medio de la interfaz para el editor Emacs.
- Se emplearon los demostradores automáticas de teoremas E y Vampire, así como el programa Apia, que es un traductor de la representación que hace Agda de formulas de primer orden al lenguaje TPTP, para demostrar automáticamente diferentes propiedades. Si una propiedad no es demostrada empleando la metodología presentada en esta tesis esto se puede asociar a uno de los siguientes casos: a) el tiempo de espera máximo que se ha programado en el computador para que los ATPs intenten realizar la demostración es corto y en consecuencia no alcanzaron a realizar la demostracion, b) de los ATPs elegidos ninguno logra hacer la demostración, en este caso se puede usar un ATP diferente, c) no sabe si la propiedad se puede demostrar automáticamente.

- Se presentaron ejemplos sobre la manera de realizar demostraciones interactivas y automáticas, ambas maneras de demostrar pueden ser el inicio para otros proyectos de investigación, pero también una motivación para profundizar más al respecto sobre este tipo de demostraciones.
- Como experiencia en la formalización de algunas propiedades del sistema de los números reales en Agda se destaca que el proceso fue arduo y extenso, esto debido a que cada vez que se formalizaba una definición o una demostración se necesitaban una serie de propiedades y definiciones previas. Por ejemplo, cuando se formalizó el concepto del límite de una función se tuvo que definir: la función valor absoluto, la función de distancia entre dos puntos y la función que retorna el mínimo entre dos valores. Además, fue necesario demostrar una serie de propiedades que dichas funciones cumplen y otras tantas más.
- Se reconocieron y experimentaron algunas de las limitaciones para realizar demostraciones automáticas. En el contexto de esta tesis las principales limitaciones están relacionadas con el programa Apia.
- Terminar esta tesis no fue fácil, pero valió la pena trasegar por este camino ya que se obtuvieron grandes aprendizajes.

6.2. Trabajo futuro

Todos las preguntas e ideas que puedan surgir y conceptos que puedan ir mas allá de esta tesis pueden ser tomados como elementos para trabajo futuro. Se propone lo siguiente:

- Coq formaliza una versión más fuerte de la propiedad Arquimediana presentada en esta tesis. Sería interesante conocer si se puede debilitar la versión de esta propiedad empleada por Coq.
- Una experiencia enriquecedora consistiría en ampliar la formalización realizada en esta tesis, por ejemplo los conceptos de continuidad de funciones, derivadas e integrales, tal como se hace por ejemplo en Coq.

Bibliografía

Apostol, T. M. (1974). Mathematical Analysis. 2.ª ed. Addison-Wesley (vid. págs. 1, 13, 15).

Appel, K. I. y Haken, W. (1989). Every Planar Map is Four Colorable. Vol. 98. American Mathematical Society (vid. págs. 1, 3).

Awodey, S. (2010). Category Theory. 2.ª ed. Oxford Logic Guides. Oxford University Press (vid. pág. 9).

Bird, R. (1998). Introduction to Functional Programing using Haskell. 2.ª ed. Prentice Hall (vid. págs. 5, 9).

Bove, A. y Dybjer, P. (2009). Dependent Types at Work. En: LerNet ALFA Summer School 2008. Ed. por Bove, A., Soares Barbosa, L., Pardo, A. y Sousa Pinto, J. Vol. 5520. Lecture Notes in Computer Science. Springer, págs. 57-99 (vid. págs. 3, 5).

Bridges, D. S. (1999). Constructive Mathematics: A Foundation for Computable Analysis. Theoretical Computer Science 219.1, págs. 95-109 (vid. págs. 22, 23).

Chang, C. C. y Keisler, J. H. (1990). Model Theory. 3.ª ed. Vol. 73. Studies in Logic and the Foundations of Mathematics. North Holland (vid. pág. 13).

Ciaffaglione, A. y Di Gianantonio, P. (2002). A Tour with Constructive Real Numbers. En: Types for Proofs and Programs (TYPES 2000). Ed. por Callaghan, P., Luo, Z., McKinna, J. y Pollack, R. Vol. 2277. Lecture Notes in Computer Science. Springer, págs. 41-52 (vid. págs. 22, 23).

Enderton, H. B. (1977). Elements of Set Theory. Academic Press (vid. págs. 13, 32).

Graham, S. (2005). Deduction with XOR Constraints in Security API Modelling. En: Automated Deduction (CADE-20). Ed. por Nieuwenhuis, R. Vol. 3362. Lecture Notes in Computer Science. Springer, págs. 328-333 (vid. pág. 18).

Hales, T. C. (2005). A Proof of the Kepler Conjecture. Annals of Mathematics 162.3, págs. 1065-1185 (vid. págs. 1, 3).

Harrison, J. (2015). HOL Light Tutorial. Intel Corporation (vid. pág. 1).

Knuth, D. E. (1992). Two Notes on Notation. American Mathematical Monthly 99.5, págs. 403-422 (vid. pág. 32).

Kovác, L. y Voronkov, A. (2013). First-Order Theorem Proving and Vampire. En: Computer Aided Verification (CAV 2013). Ed. por Sharygina, N. y Veith, H. Vol. 8044. Lecture Notes in Computer Science. Springer, págs. 1-35 (vid. pág. 38).

Landau, E. (1951). Foundations of Analysis. Chelsea Publishing Co (vid. pág. 13).

Martin-Löf, P. (1984). Intuitionistic Type Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Bibliopolis (vid. págs. 3, 4).

Mu, S.-C., Ko, H.-S. y Jansson, P. (2009). Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. Journal of Functional Programming 19.5, págs. 545-579 (vid. pág. 11).

Nipkow, T., Paulson, L. C. y Wenzel, M. (2016). A Proof Assistant for Higher-Order Logic. Springer-Verlag (vid. pág. 1).

Norell, U. (2007). Towards a Practical Programming Language Based on Dependent Type Theory. Tesis doct. Department of Computer Science and Engineering. Chalmers University of Technology y University of Gothenburg (vid. pág. 1).

— (2009). Dependently Typed Programming in Agda. En: Advanced Functional Programming (AFP 2008). Ed. por Koopman, P., Plasmeijer, R. y Swierstra, D. Vol. 5832. Lecture Notes in Computer Science. Springer, págs. 230-266 (vid. págs. 3, 6).

Pelletier, F. J. y Hartline, A. (2008). Ternary Exclusive *Or*. Logic Journal of the IGPL 16.1, págs. 75-83 (vid. pág. 18).

Rosenlicht, M. (1968). Introduction To Analysis. Dover Publications, Inc. (vid. págs. 1, 21, 22).

Royden, H. L. y Fitzpatrick, P. M. (2010). Real Analysis. Prentice Hall (vid. págs. 2, 15).

Schulz, S. (2013). System Description: E 1.8. En: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-19). Ed. por McMillan, K., Middeldorp, A. y Voronkov, A. Vol. 8312. Lecture Notes in Computer Science. Springer, págs. 735-743 (vid. pág. 38).

Bibliografía

Sicard-Ramírez, A. (2015). Reasoning about Functional Programs by Combining Interactive and Automatic Proofs. Tesis doct. PEDECIBA Informática. Universidad de la República. Uruguay (vid. págs. 1, 3, 10).

Stallman, R. et al. (2015). GNU Emacs Manual. Free Software Foundation (vid. pág. 3).

Sutcliffe, G. (2010). The TPTP World—Infrastructure for Automated Reasoning. En: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). Ed. por Clarke, E. y Voronkov, A. Vol. 6355. Lecture Notes in Computer Science. Springer, págs. 1-12 (vid. pág. 35).

The Coq Development Team (2016). Reference Manual: Version 8.5 (vid. pág. 1).

Apéndice A

Propiedades de la igualdad proposicional

En este apéndice se presentan algunas propiedades de la igualdad proposicional definida en la sección 2.10 empleadas en esta tesis.

Simetría de la igualdad.

```
≡-sym : {x y : ℝ} → x ≡ y → y ≡ x
≡-sym (refl x) = refl x
```

Transitividad de la igualdad.

```
≡-trans : {x y z : ℝ} → x ≡ y → y ≡ z → x ≡ z
≡-trans (refl z) (refl z) = refl z
```

Doble sustitución para la igualdad.

```
subst₂ : (P : ℝ → ℝ → Set) → {x₁ x₂ y₁ y₂ : ℝ} →
         x₁ ≡ x₂ → y₁ ≡ y₂ → P x₁ y₁ → P x₂ y₂
subst₂ P (refl x) (refl y) h = h
```

Apéndice B

Definiciones

Se presentan algunas definiciones empleadas en la formalización del sistema de números reales.

```
-- The inclusive disjunction data type.
data _∨_ (A B : Set) : Set where
  inj₁ : A → A ∨ B
  inj₂ : B → A ∨ B

-- Substraction.
_-_ : ℝ → ℝ → ℝ
x - y = x + (- y)
```

Apéndice C

Demostraciones interactivas auxiliares

En este apéndice se presentan algunas demostraciones usadas en esta tesis pero no presentadas en el cuerpo principal del trabajo.

Ejemplo C.1. Se demuestra la propiedad `1>0`:

```
1>0 : r₁ > r₀
1>0 = p₁-helper (>-sqr (1≠0))
  where
  p₁-helper : sqr r₁ > r₀ → r₁ > r₀
  p₁-helper h = subst₂ (λ t₁ t₂ → t₁ > t₂) r₁-sqr (refl r₀) h
```

Ejemplo C.2. Se demuestra la propiedad `>-+-cancel-r`:

```
>-+-cancel-r : {x y z : ℝ} → x + z > y + z → x > y
>-+-cancel-r h = p₁-helper (p₂-helper (p₃-helper (p₄-helper h)))

  where
  p₁-helper : {x y : ℝ} → x + r₀ > y + r₀ → x > y
  p₁-helper {x} {y} h =
    subst₂ (λ t₁ t₂ → t₁ > t₂) (+-neut x) (+-neut y) h

  p₂-helper : {x y z : ℝ} → x + (z + (- z)) > y + (z + (- z)) →
              x + r₀ > y + r₀
  p₂-helper {x} {y} {z} h = subst₂ (λ t₁ t₂ → x + t₁ > y + t₂)
                                   (+-inve z) (+-inve z) h
```

```
p₃-helper : {x y z : ℝ} → (x + z) + (- z) > (y + z) + (- z) →
                           x + (z + (- z)) > y + (z + (- z))
p₃-helper {x} {y} {z} h = p-helper h

  where
  p-helper : {x y x₁ x₂ : ℝ} → (x + x₁) + x₂ > (y + x₁) + x₂ →
             x + (x₁ + x₂) > y + (x₁ + x₂)
  p-helper {x} {y} {x₁} {x₂} h =
    subst₂ (λ t₁ t₂ → t₁ > t₂)
           (+-asso x x₁ x₂)
           (+-asso y x₁ x₂) h

p₄-helper : {x y z : ℝ} → x + z > y + z →
            x + z + (- z) > y + z + (- z)
p₄-helper {x} {y} {z} h = >-+-right h
```

Comentario. Las propiedades `>-sqr`, `r₁-sqr` y `>-+-right` se encuentran formalizadas en el repositorio archivo `Properties.agda`.

Ejemplo C.3. Propiedad `>-to-<`:

```
>-to-< : {x y : ℝ} → x > y → y < x
>-to-< {x} {y} h = h
```

Ejemplo C.4. A partir del axioma de tricotomía definido en la sección 3.2, se prueba la propiedad asimétrica, nombrada `>-asym`, de la siguiente forma:

```
>-asym : {x y : ℝ} → x > y → x ≮ y
>-asym {x} {y} x>y x<y = case prf₁ prf₂ (trichotomy x y)
  where
  prf₁ : (x > y) ∧ (¬ x ≡ y) ∧ (¬ x < y) → ⊥
  prf₁ (_ , _ , p) = p x<y

  prf₂ : (¬ x > y ∧ x ≡ y ∧ ¬ x < y) ∨ (¬ x > y ∧ ¬ x ≡ y ∧ x < y) → ⊥
  prf₂ (inj₁ (p , _)) = p x>y
  prf₂ (inj₂ (p , _)) = p x>y
```

La propiedad `>-asym` indica que si existe una relación de orden $>$ entre dos elementos distintos de tipo $\mathbb{R}$, es decir $x > y$, y si $y > x$ entonces se produce una contradicción.

Además, a partir del axioma de tricotomía se demuestra que para todo $x, y \in \mathbb{R}$ si $x \not\equiv y$ y $x \not< y$ entonces se tiene $x > y$, es decir:

```
≠∧≮→> : {x y : ℝ} → (x ≠ y) ∧ (x ≮ y) → x > y
≠∧≮→> {x} {y} (x≠y , x≮y) = case prf₁ prf₂ (trichotomy x y)

  where
  prf₁ : x > y ∧ ¬ x ≡ y ∧ ¬ x < y → x > y
  prf₁ h = ∧-proj₁ h

  prf₂ : (¬ x > y ∧ x ≡ y ∧ ¬ x < y) ∨ (¬ x > y ∧ ¬ x ≡ y ∧ x < y) →
         x > y
  prf₂ h = case prf₂₁ prf₂₂ h

    where
    prf₂₁ : ¬ x > y ∧ x ≡ y ∧ ¬ x < y → x > y
    prf₂₁ h₁ = ⊥-elim (x≠y (∧-proj₁ (p-helper h₁)))

      where
      p-helper : ¬ x > y ∧ (x ≡ y ∧ ¬ x < y) →
                 x ≡ y ∧ (¬ x < y ∧ ¬ x > y)
      p-helper h₂ = ∧-assoc₂ (∧-comm h₂)

    prf₂₂ : (¬ x > y) ∧ (¬ x ≡ y) ∧ (x < y) → x > y
    prf₂₂ h₃ = ⊥-elim (x≮y (∧-proj₂ (∧-assoc₁ h₃)))
```

Igualmente se prueba que cada uno de los disyuntos del axioma de tricotomía implica la negación de los otros dos, es decir:

```
>→≮∧≠ : (x y : ℝ) → x > y → x ≮ y ∧ x ≠ y
>→≮∧≠ x y x>y = (λ x<y → >-asym x>y x<y) , (λ x=y → ≡→≯ x=y x>y)

≡→≮∧≯ : (x y : ℝ) → x ≡ y → x ≮ y ∧ x ≯ y
≡→≮∧≯ x y x=y = (λ x<y → <→≠ x<y x=y) , (λ x>y → >→≠ x>y x=y)

<→≯∧≠ : (x y : ℝ) → x < y → x ≯ y ∧ x ≠ y
<→≯∧≠ x y x<y = (λ x>y → <-asym x<y x>y) , (λ x=y → <→≠ x<y x=y)
```

La demostración de las propiedades `<-asym`, `≡→≯`, `>→≠` y `<→≠` se encuentran en el repositorio en el archivo `Properties.agda`.