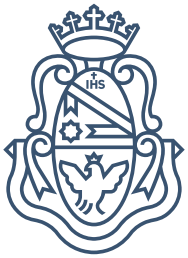
FORMALIZACIÓN DE FUNDACIONES DE LA MATEMÁTICA Y COMPILADORES CORRECTOS POR CONSTRUCCIÓN

EMMANUEL GUNTHER
DIRECTOR: DR. MIGUEL M. PAGANO

En cumplimiento de los requisitos para adquirir el grado
de Doctor en Ciencias de la Computación



Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba

All human activity is prompted by desire. There is a wholly fallacious theory advanced by some earnest moralists to the effect that it is possible to resist desire in the interests of duty and moral principle. (Bertrand Russell)

RESUMEN

El desarrollo de las teorías fundacionales de la Matemática aparecidas en el siglo XX ha dado lugar a numerosos avances científicos en lógica y en ciencias de la computación. La *teoría de conjuntos* propuesta inicialmente por Georg Cantor a finales del siglo XIX y luego axiomatizada por Zermelo y Fraenkel para evitar paradojas constituye una de las teorías fundacionales más estudiadas y aceptadas por la comunidad científica. Paralelamente a ésta se desarrolló la *teoría de tipos* a partir de las primeras propuestas realizadas por Bertrand Russell, luego con el cálculo lambda simplemente tipado de Alonzo Church y más adelante con Per Martin-Löf, entre otros. La teoría de tipos abrió el camino a una prolífica área de estudio en ciencias de la computación: por un lado los lenguajes de programación pueden expresar propiedades de corrección en el tipado de los programas, y por otro permiten escribir formalmente resultados matemáticos que pueden chequearse automáticamente, lo que se conoce como *formalización de matemática*.

En el transcurso de este doctorado hemos estudiado la teoría de tipos y su aplicación para desarrollar programas correctos por construcción, en particular compiladores. Mediante el uso de tipos dependientes se puede especificar en el tipo de un compilador la propiedad de corrección con respecto a las semánticas de los lenguajes; presentamos en la tesis una metodología que muestra los alcances y límites de esta propuesta. La teoría subyacente surge de considerar a los lenguajes como álgebras de términos de una signatura, lo cual nos llevó a estudiar álgebras heterogéneas y traducciones entre álgebras de distintas signaturas. El siguiente aporte de nuestro trabajo es la formalización en el lenguaje Agda de una librería con los principales conceptos de álgebra universal, incluyendo un sistema deductivo para la lógica de cuasi identidades y las nociones de morfismos entre signaturas y álgebras reducto asociadas a dichos morfismos.

Finalmente hemos estudiado uno de los problemas más famosos en teoría de conjuntos: la independencia de la hipótesis del continuo. La misma afirma que con la axiomatización de Zermelo y Fraenkel (ZF) no se puede probar ni refutar que existan conjuntos con cardinalidad mayor estricta a la de los números naturales y menor estricta a la de los números reales. Este resultado fue obtenido gracias a los aportes de Kurt Gödel en 1940 (quien probó que la hipótesis del continuo es consistente con ZF) y luego completado por Paul Cohen veinte años más tarde, quien introdujo la técnica de *forcing* para probar que la negación de la hipótesis del continuo también es consistente con ZF. El desarrollo matemático implicado en estos resultados conlleva

un interés particular para realizarlo formalmente en un asistente de pruebas, y no existe hasta el momento una formalización completa. Lawrence Paulson realizó una importante contribución en el asistente de pruebas Isabelle hasta obtener una prueba formal del resultado de Gödel. En la última parte de esta tesis, presentamos los primeros pasos hacia una formalización de forcing tomando como punto de partida el trabajo de Paulson. En particular, definimos la extensión genérica de modelos transitivos contables y probamos la validez de algunos de los axiomas de ZF en ella.

Clasificación (ACM CCS 2012)

- **Theory of computation∼Logic and verification**
- **Theory of computation∼Software verification**
- **Theory of computation∼Type Theory**
- **Software and its engineering∼Compilers**

Palabras claves
diseño de compiladores, programación con tipos dependientes, corrección por construcción, formalización de matemática, asistentes de prueba, teoría de conjuntos, *forcing*.

AGRADECIMIENTOS

Durante el recorrido necesario para alcanzar esta hazaña llamada doctorado, me he visto acompañado por un gran conjunto de personas a quienes me gustaría agradecer. Si bien la previa descripción alcanzaría para definir dicho conjunto por comprensión, me gustaría intentar realizarlo extensionalmente a riesgo de olvidar algunas, a quienes pido adelantadas disculpas.

A Miguel, que más que director siempre fue un compañero con una increíble paciencia para aguantarme...

A los miembros del jurado, Camper, Beta y Alexandre; por sus dedicadas lecturas, comentarios y discusiones surgidas a partir de la tesis...

Al gran compañero Ale, hermano de andanzas en este camino...

A Pedro, que se sumó en el último tramo y me aguantará un par de años más...

A mi amigo Mati, compañero de viajes, alegrías, depresiones y tanto más; por usurparle la casa cada semana...

A mi 'papá uruguayo' Alberto, por haberme recibido tan cálidamente en mis visitas a Montevideo e integrarme no solo a su trabajo científico, sino también a su familia; y a Marcos con quien compartimos grandes momentos de trabajo y varios 'chivitos'...

A Daniel, Leo, Mallku, Javi, Ara, Renato y demás compañeros de la FaMAF...

A mi familia por estar siempre ahí acompañando aún sin entender bien qué locura hacemos estudiando estas cosas...

A mis amigos de la vida, en especial a José, Pato, Juane y Blas...

A Fla, que bancó los trapos en la etapa inicial cuando todo es incertidumbre y ni siquiera uno sabe lo que está haciendo...

Y a Vero, compañera en este presente alborotado, de pesares de tesis, estreno de casa y búsqueda de rumbos...

ÍNDICE GENERAL

1 INTRODUCCIÓN

1.1 FUNDACIONES DE LA MATEMÁTICA

La Matemática se ha estudiado a lo largo de la historia desde tiempos remotos. Cientos de teorías han sido desarrolladas rigurosamente con sus propias definiciones, postulados y teoremas, y han encontrado campos de aplicación diversos. Entre finales del siglo XIX y comienzos del siglo XX algunos matemáticos, lógicos y filósofos se embarcaron en una ambiciosa tarea: desarrollar una teoría común a toda la matemática, es decir, tomar como objeto de estudio a *La Matemática* misma y probar resultados acerca de ella. Siendo el conjunto de resultados desarrollados a lo largo de la historia tan diverso, ¿pueden convivir las distintas teorías existentes en un mismo marco de estudio? ¿cuáles son los conceptos primitivos a partir de los cuales se pueden desarrollar todas ellas? ¿habrá contradicciones entre los resultados de las distintas teorías? Una abstracción adecuada debería poder responder estos cuestionamientos. Se conoce como *fundaciones de la matemática* a estos estudios filosóficos y lógicos sobre las bases de la matemática. Esta breve introducción, basada en [4, 19, 26, 44, 78], pretende dar un contexto histórico mencionando los principales hitos en el desarrollo de los fundamentos que dan origen al estudio moderno de la matemática y en particular a las Ciencias de la Computación.

Los estudios de Georg Cantor sobre conjuntos infinitos sentaron los primeros pasos hacia un estudio fundacional. El concepto primitivo a partir del cual puede obtenerse cualquier concepto matemático es el de *conjunto* [15, p.85]:

> "Un *conjunto* es una colección de objetos distinguibles y definidos en nuestra intuición o intelecto, que pueden ser concebidos como un todo."[1]

Los objetos que conforman un conjunto son llamados *elementos* (o *miembros*), y el conjunto *contiene* sus elementos, mientras que estos últimos *pertenecen* al conjunto. A partir de esta definición y con algunos principios de base que se asumen válidos, se desarrolla la teoría en la cual se pueden representar los principales conceptos matemáticos como los *pares ordenados* (representando el par (x, y) con el conjunto $\{\{x\}, \{x, y\}\}$), los números naturales (mediante la representación de Von Neumann, $0 \equiv \varnothing$; y $n + 1 \equiv n \cup \{n\}$), el producto cartesiano, las

1 Traducción del autor.

funciones; y a partir de ellos otras construcciones como los números racionales, los reales, etcétera.

Además de representar la "matemática conocida", la teoría de conjuntos desarrolló conceptos nuevos relacionados a la *infinitud*. A partir de la simple definición de Cantor es posible concebir conjuntos con una cantidad infinita de elementos y es interesante estudiar propiedades sobre éstos. La *cardinalidad* es una de ellas: ¿cuál es el *tamaño* de los conjuntos? Para los finitos es sencillo, pues podemos dar un número natural que lo defina; ¿pero qué sucede con los infinitos? Decimos que dos conjuntos son *equipotentes* si hay una función biyectiva entre ellos. De ese modo es posible analizar el tamaño de conjuntos infinitos y se introducen los *números cardinales*. En 1873 Cantor prueba que los números reales, conjunto conocido como *el continuo*, no son biyectivos con los números naturales, sino que su cardinalidad es mayor. Este descubrimiento fue tan importante que algunos lo consideran el hecho que dio nacimiento a la teoría de conjuntos.

Esta nueva teoría, que permitía dar un marco común de estudio a toda la matemática y a su vez estudiar conceptos de infinitud que habían sido inabordables hasta el momento, se desarrollaba despertando cada vez más interés y por consiguiente científicos que la estudiaron en profundidad. Sin embargo hacia el 1900 se descubren varias paradojas lógicas que pusieron en jaque a toda el área. Cantor mismo fue uno de los que encontró una de ellas relacionada a los cardinales, Burali-Forti descubrió otra sobre los números ordinales, y la más famosa debido a su simplicidad es la de Russell, ya que evidencia que la definición misma de conjunto admite construir contradicciones: uno de los principios de la teoría de conjuntos de Cantor, el conocido principio de comprehensión, permite construir a partir de una propiedad φ el conjunto de exactamente todos los elementos que la satisfacen $\{x \mid \varphi(x)\}$,

Paradoja de Russell Sea $\varphi(x)$ la siguiente propiedad:

$$x \text{ es un conjunto y no pertenece a sí mismo, es decir } x \notin x \quad (1)$$

luego podemos construir el conjunto $R = \{x \mid x \notin x\}$. La paradoja aparece al preguntarnos si R pertenece a R. Si así fuera, luego R no satisface φ y entonces $R \notin R$. Por otro lado, si R no pertenece a sí mismo, luego $R \in R$. Es decir llegamos a la contradicción $R \in R$ sii $R \notin R$.

A partir de estas paradojas la comunidad científica se embarcó en una vertiginosa carrera por encontrar soluciones. La teoría de Cantor, si bien estaba desarrollada a partir de algunos principios y los resultados se iban obteniendo mediante deducciones lógicas, no tenía la rigurosidad formal con el que se estudia la matemática actualmente, y por ello se le llama *Teoría Informal de Conjuntos* (Naive Set Theory, en inglés). Eran necesarias teorías más rigurosas para solucionar las

paradojas, y así es que durante los primeros años del siglo XX tres corrientes emergieron: el *logicismo*, el *intuicionismo* y el *formalismo*. Cada una de ellas propuso un marco filosófico distinto para encarar el estudio de los fundamentos matemáticos, y algunos de sus promotores defendieron fuertemente sus ideas llegando a darse importantes disputas, en particular en el año 20 los acalorados debates entre Hilbert y Brouwer escalaron tan alto que se dio en llamar la *Crisis Fundacional* a esta etapa. Un análisis detenido sobre la importancia de esta época para la matemática y la filosofía puede encontarse en [24].

Logicismo La corriente logicista considera a la matemática como una extensión de la lógica. Entre los precursores se encuentran a Boole, Frege y Peano, quienes introdujeron rigurosidad en el estudio de la lógica junto con un lenguaje preciso a partir de un alfabeto de símbolos y reglas de construcción sintáctica. A Boole se le atribuye la introducción del Cálculo Proposicional[2] y Frege la Lógica de Primer Orden. Este último y Peano estudiaron la aritmética a partir de principios lógicos, y la notación introducida por Peano es utilizada hasta el día de hoy. Russell siguió esta línea con el ambicioso objetivo de poder deducir toda la matemática a partir de la lógica. En camino a solucionar las paradojas de la teoría de conjuntos introdujo la *Teoría de Tipos* [80] en la cual los objetos matemáticos son clasificados en *tipos* para los cuales existe una jerarquía comenzando con los *individuos*, luego las clases de individuos, luego clases de clases, etc. Junto con Whitehead escribió tres libros conocidos con el nombre de *Principia Mathematica* [95] en donde desarrollan a partir de la lógica y algunos axiomas adicionales la teoría de conjuntos, los cardinales, ordinales, números reales, evitando las paradojas conocidas. Si bien no fueron recibidos por la comunidad metemática como los autores esperaban y recibieron críticas por la inclusión de axiomas que no son puramente lógicos, Principia Mathematica constituyen un hito en las fundaciones: popularizaron el tratamiento moderno de la lógica matemática mediante el uso de un lenguaje preciso como ya había sido introducido por Frege y Peano; y mostraron el poder deductivo de los nuevos sistemas formales abriendo el camino a la *metalógica*.

Intuicionismo La corriente más revolucionaria fue la *intuicionista*, introducida por Luitzen Brouwer, quien rechazaba algunos principios básicos de la matemática clásica. Filosóficamente, el intuicionismo concibe a los objetos matemáticos como producto de la mente humana, a diferencia del enfoque platónico en el cual la existencia de éstos es independiente y el ser humano los descubre. Para un intuicionista un objeto existe si puede ser *construido* mentalmente, y ésto contrasta con muchos resultados de la matemática *clásica* donde la prueba de la existencia de un objeto e que satisfaga alguna propiedad P se rea-

2 Aunque no como la conocemos actualmente.

liza asumiendo que para todo x no vale $P(x)$ y luego llegando a una contradicción. Dicha prueba no muestra cómo se obtiene e y no es aceptada por los intuicionistas. En particular lo que el intuicionismo rechaza es la *Ley del Tercero Excluido*, en la cual una proposición p es o bien verdadera, o bien falsa, y por lo tanto para probar la validez de p alcanza con llegar a un absurdo a partir de la hipótesis $\neg p$. En contrapartida, una prueba intuicionista de una disjunción $p \vee q$ debe ser una prueba de p o una prueba de q. La consecuencia inmediata de las bases de esta corriente es que gran parte de la matemática debía ser reinventada, puesto que muchísimas pruebas clásicas utilizan el tercero excluído. Brouwer hizo un gran avance definiendo gran cantidad de conceptos intuicionísticamente, pero la mayoría de sus contemporáneos rechazaban la propuesta.

Formalismo El *formalismo* no aceptaba la propuesta radical del intuicionismo y por consiguiente la matemática clásica no debía ser descartada. Tiene su origen en el *Programa de Hilbert*, quien estipuló los siguientes pasos para estudiar con rigurosidad los conceptos matemáticos, presentados de la siguiente manera por Ferreirós en [24]:

> (i) encontrar axiomas y conceptos de una teoría matemática T, como por ejemplo la de los números reales; (ii) encontrar axiomas y reglas de inferencia de la lógica clásica, que a partir de proposiciones permita obtener nuevas siguiendo un procedimiento puramente sintáctico; (iii) formalizar T a partir del cálculo lógico formal, definiendo las proposiciones con un lenguaje preciso, y obteniendo pruebas como una secuencia de palabras de dicho lenguaje siguiendo las reglas de inferencia; y (iv) realizar un estudio *finitario* de las pruebas formalizadas de T que muestre que es imposible que la última línea de una prueba sea una contradicción.

La corriente formalista fue finalmente la que prosperó y dio lugar a los sistemas axiomáticos formales, en particular el de Zermelo y Fraenkel (*ZF*) para la teoría de conjuntos, constituyendo la teoría fundacional más aceptada hasta la actualidad.

Si bien la corriente intuicionista quedaría derrotada frente al programa de Hilbert, con el desarrollo de las ciencias de la computación vuelve a tener relevancia. La lógica intuicionista interpreta a los conectivos lógicos de la siguiente manera[3]:

- $\vee$ (**disjunción**). Para probar $p \vee q$ se debe dar una prueba de p o una de q, indicando cuál de las dos es.
- $\wedge$ (**conjunción**). Para probar $p \wedge q$ se debe dar una prueba de p y una prueba de q.

3 *BHG-interpretation*, en honor a Brouwer, Heyting y Kolmogorov.

- $\to$ (**implicación**). Una prueba de $p \to q$ es un algoritmo que convierte cualquier prueba de p en una prueba de q.
- $\neg$ (**negación**). Una prueba de $\neg p$ es un algoritmo que transforma toda prueba de p en una prueba de $0 = 1$.
- $\exists$ (**existencial**). Para probar $\exists x.P(x)$ se debe construir un objeto x y probar que vale $P(x)$.
- $\forall$ (**universal**). Una prueba de $\forall x.P(x)$ es un algoritmo que, aplicado a cualquier objeto x obtiene una prueba de $P(x)$.

y para esta concepción de la lógica, una prueba no es otra cosa que un algoritmo que puede ejecutarse en una computadora.

Erret Bishop en su trabajo *Foundations of Constructive Analysis* [10] del año 1967 desarrolla constructivamente gran parte del análisis matemático con un tratamiento de fácil acceso para los matemáticos clásicos, a diferencia del desarrollo de Brouwer. De manera casi simultánea[4] Per Martin-Löf escribe *Notes on Constructive Mathematics* [53] y luego desarrolla su teoría de tipos intuicionista [54, 55] en la cual la matemática constructiva de Bishop puede ser fundamentada.

Cerramos este breve repaso por las teorías fundacionales mencionando la reciente aparición de la *Homotopy Type Theory* [91], la cual atrajo a los principales científicos y constituye el *estado del arte*. Un interesante análisis del estado actual de las fundaciones junto con un repaso histórico se encuentra en [23].

1.2 TEORÍA DE TIPOS INTUICIONISTA

Así como la teoría de conjuntos *ZF* constituye el fundamento para la matemática clásica, la teoría de tipos intuicionista de Martin-Löf hace lo propio para la matemática constructivista. Está basada en el principio *propositions-as-types*, también conocido como la correspondencia de Curry-Howard, en el cual toda proposición lógica equivale a un tipo. Un tipo puede representar un conjunto de elementos, como también una proposición lógica: la expresión $x : T$ afirma que x es elemento del tipo T, o que la proposición representada por T vale, y x es una prueba de ello; a su vez todo tipo T representa una proposición, y la misma es válida si el tipo está habitado por algún elemento.

La primera teoría de tipos fue introducida por Russell como una solución a la paradoja que lleva su nombre, creando la noción de *tipos* para diferenciar objetos, proposiciones, propiedades sobre objetos, etcétera. Posteriormente Alonzo Church desarrolla su teoría de tipos simple [18] sobre el cálculo lambda, donde existe el tipo de las funciones de A en B, para todo tipo A y B. La teoría de Martin-Löf extiende estas funciones con la introducción de los *tipos dependientes*,

4 Aunque su publicación se produce un par de años después.

es decir, tipos que dependen de elementos: para cada elemento x de tipo A se obtiene otro tipo $B(x)$, también llamado *familia de tipos*.

A continuación daremos una breve introducción a la teoría explicando las bases del sistema formal y las nociones primitivas que permiten definir tipos. Tomamos como referencia la introducción de [91], y para una precisa descripción formal recomendamos el apéndice del mismo libro.

Sistema deductivo

En las teorías basadas en el formalismo de Hilbert, como la teoría de conjuntos ZF, tenemos dos "niveles" diferenciados: por un lado el sistema deductivo de la lógica de primer orden, y dentro de éste la formulación de axiomas que hablan sobre los conjuntos. La teoría de tipos difiere en este aspecto: *es* su propio sistema deductivo, es decir, no necesita de una estructura superior como la lógica de primer orden, no existen las nociones separadas de proposición y conjunto, sino sólo los *tipos*.

Un sistema deductivo consiste de una serie de reglas sintácticas mediante las cuales se derivan *juicios*. En la lógica de primer orden, los únicos juicios que existen son de la forma "la proposición P tiene una prueba", y las reglas permiten inferir pruebas de proposiciones más complejas como por ejemplo "a partir de una prueba de A y una de B se infiere que $A \wedge B$ tiene una prueba". De esta manera una prueba es una secuencia de instancias de reglas de inferencia donde la última línea de la misma será el juicio que asegura que determinada proposición vale.

En la teoría de tipos tenemos un juicio básico $a : A$ (que se lee como "a tiene tipo A" , o a es elemento de A), llamado *juicio de tipado*, que podrá interpretarse de manera similar a la pertenencia de un elemento en un conjunto, o también que la proposición A vale y su prueba es a. Tenemos además otra forma de juicio $a \equiv b : A$, llamado la *igualdad definicional* o *igualdad de juicio*, que informalmente expresa que dos elementos de un tipo son iguales por definición (y también que dos tipos son iguales, como quedará claro más adelante). A partir de la correspondencia entre proposiciones y tipos, podemos entender esta igualdad como equivalencia lógica, y en consecuencia podremos deducir $a : B$ a partir de $a : A$ y $A \equiv B$; es decir, si A es lógicamente igual a B y a es una prueba de A, luego a también es prueba de B. Esta noción de igualdad no es análoga a la de los conjuntos, donde podemos dar una prueba de la validez o negación de $A = B$. Dicha expresión es una proposición que puede probarse y por lo tanto habrá un tipo que la representa, siendo sus habitantes testigos de la validez. A esta última noción de igualdad se la conoce como *igualdad proposicional*.

Universos En la teoría de tipos un objeto no puede existir si no es como habitante de algún tipo, por lo tanto los tipos mismos deben también habitar uno, al que llamamos *universo*. Para evitar paradojas se introduce una jerarquía:

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : ...$$

donde cada universo $\mathcal{U}_i$ es un elemento de $\mathcal{U}_{i+1}$, y más aún si $A : \mathcal{U}_i$, luego también $A : \mathcal{U}_{i+1}$. Si no es necesario conocer a qué nivel de universo explícitamente pertenece un tipo, se lo suele omitir anotando simplemente $A : \mathcal{U}$.

Definición de tipos

La teoría queda completamente definida mediante la introducción de tipos. Para cada uno se definirán reglas de inferencia que expresen lo siguiente:

- **Regla de formación**. Un juicio de tipado que introduce un nuevo tipo especificando las condiciones necesarias para su formación.
- **Constructores o Reglas de Introducción**. Juicios de tipado que especifican los elementos canónicos que habitan el tipo.
- **Eliminadores**. Juicios de tipado que definen cómo se descompone el tipo definido en otros tipos.
- **Reglas de computación**. Juicios de igualdad definicional describiendo la acción de un eliminador sobre los constructores.

Además puede definirse un principio de unicidad opcional para algunos tipos, estableciendo reglas de igualdad entre formas de construir elementos. En este texto no las mencionaremos ya que no son relevantes en nuestro desarrollo.

En la definición formal del sistema deductivo los juicios de tipado tienen la forma $\Gamma \vdash x : T$, donde Γ es un *contexto* de asignación de tipos a variables (que en teoría de tipos se corresponden con las hipótesis lógicas asumidas), x es algún elemento y T un tipo; y los juicios de igualdad tendrán la forma $\Gamma \vdash a \equiv b : T$. A continuación presentamos los principales tipos de la teoría mostrando sólo algunas reglas de inferencia formalmente, el lector interesado puede encontrar el resto en el apéndice de [91].

Funciones

A diferencia de la teoría de conjuntos, donde las funciones son representadas como conjuntos de pares, en la teoría de tipos las funciones son primitivas. En el cálculo lambda simplemente tipado de

Church se introduce el tipo $A \to B$ a partir de dos tipos A y B; los habitantes de $A \to B$ se corresponden con las funciones de A en B (donde $A : \mathcal{U}$ y $B : \mathcal{U}$). En la teoría de tipos de Martin-Löf podemos definir funciones que dado un elemento $a : A$ devuelvan un $B : \mathcal{U}$, a las que llamamos *familias de tipos* o *tipos dependientes*. Luego podemos definir funciones que, dado un elemento $a : A$ devuelvan un elemento en el tipo $B\,a$, a las que llamamos *funciones dependientes*. A continuación mostraremos las reglas para el tipo de las funciones:

Regla de formación A partir de un tipo A y un tipo B (que puede depender de un elemento de A) se construye el tipo de las funciones $\Pi_{(x:A)} B$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Pi_{(x:A)} B : \mathcal{U}_i} \text{ (Π-Form)}$$

Si B devuelve constantemente el mismo tipo para todo $x : A$ (es decir, en B no ocurre libremente la variable x), la regla anterior define las funciones no dependientes, cuyo tipo se escribe $A \to B$. La expresión $\Pi_{(x:A)} B$ liga las ocurrencias libres de x en B.

Constructor El constructor de las funciones es la abstracción lambda

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \Pi_{(x:A)} B} \text{ (Π-Intro)}$$

La expresión lambda liga las variables libres de x en b. Habitualmente se omite la anotación de tipo.

Eliminador El eliminador para las funciones es la aplicación. Si f es una función con tipo $\Pi_{(x:A)} B$ y $a : A$, entonces la aplicación $f\,a$ tendrá tipo $B[a/x]$, es decir, B donde se reemplazan las ocurrencias libres de x por a.

$$\frac{\Gamma \vdash f : \Pi_{(x:A)} B \quad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B[a/x]} \text{ (Π-Elim)}$$

Regla de computación La regla de computación define la acción de aplicación: se reemplaza en el cuerpo de la función las ocurrencias libres de la variable por el término al cual se está aplicando.

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)\,a \equiv b[a/x] : B[a/x]} \text{ (Π-Comp)}$$

Esta regla es conocida como la *regla Beta*.

Producto

En la teoría de tipos simple, los habitantes del producto son pares donde cada elemento puede ser de un tipo distinto. Con tipos dependientes, el segundo componente del par puede depender del primero.

Regla de formación A partir de un tipo A y una familia de tipos B indexada en A, se construye el tipo de los pares dependientes $\Sigma_{(x:A)} B$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Sigma_{(x:A)} B : \mathcal{U}_i} \text{ (}\Sigma\text{-Form)}$$

Si B devuelve un tipo de manera constante para todo elemento $x : A$ (es decir, si en B no ocurre libremente la variable x), tenemos el producto no dependiente, cuya notación habitual es $A \times B$.

Constructor A partir de un elemento $a : A$ y un $b : B[a/x]$ podemos construir el elemento $(a, b) : \Sigma_{(x:A)} B$.

$$\frac{\Gamma, x : A \vdash B : \mathcal{U}_i \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma_{(x:A)} B} \text{ (}\Sigma\text{-Intro)}$$

Eliminadores y regla de computación Para computar un elemento de un tipo C a partir de un par dependiente $(a, b) : \Sigma_{(x:A)} B$ necesitamos una función de $\Pi_{(x:a)} B$ en C (que también puede depender de a y b). Las proyecciones son un caso particular de las reglas de eliminación y computación, donde $C = A$ para la primera, y $C = \lambda x : A.B\,x$ en el segundo:

$$\begin{aligned} &pr_1 : \Sigma_{(x:a)} B \to A \\ &pr_1\,(a, b) \;=\; a \end{aligned}$$

$$\begin{aligned} &pr_2 : \Pi_{(p:\Sigma_{(x:A)} B)} B \\ &pr_2\,(a, b) \;=\; b \end{aligned}$$

Coproducto

El coproducto se corresponde con la unión disjunta en teoría de conjuntos. Dados dos tipos A y B se define $A + B$ y tiene dos constructores: uno que obtiene un elemento a partir de un $a : A$ y otro que lo obtiene a partir de un $b : B$.

Regla de formación

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i}\ (+\text{-FORM})$$

Constructores

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash a : A}{\Gamma \vdash \mathit{inl}\, a : A + B}\ (+\text{-INTRO1})$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash b : B}{\Gamma \vdash \mathit{inr}\, b : A + B}\ (+\text{-INTRO2})$$

Eliminadores y regla de computación Para descomponer un elemento del coproducto en un elemento de algún tipo C necesitamos dos funciones: una de un $a : A$ en C (que puede depender de a), y otra de un $b : B$ en C (que puede depender de b). Para computar a partir de un $z : A + B$ se aplica una de las dos funciones, dependiendo si $z = \mathit{inl}\, a$ para algún $a : A$, o si $z = \mathit{inr}\, b$ para algún $b : B$.

Tipos Vacío y Unitario

Por último, presentamos dos tipos que tienen un rol importante en la representación de proposiciones como tipos:

El tipo vacío $\mathbf{0}$ no tiene constructores, y la regla de eliminación permite deducir que existe un elemento de cualquier tipo asumiendo que hay un elemento de $\mathbf{0}$. Sin embargo no tiene regla de computación, ya que ésta indica la acción del eliminador aplicado a un término canónico pero $\mathbf{0}$ no tiene tales términos.

El tipo unitario $\mathbf{1}$ tiene un sólo elemento $* : \mathbf{1}$. La eliminación del tipo unitario en otro tipo C dependerá de una función que para cada elemento de $\mathbf{1}$ elija uno de C. Para descomponer $\mathbf{1}$ en otro tipo cualquiera C debe proveerse un elemento de C.

Proposiciones como tipos

Para cada conectivo de la lógica proposicional podemos establecer una correspondencia entre sus reglas de introducción y eliminación, y las reglas de los tipos que presentamos previamente.

- *False* $\leftrightsquigarrow \mathbf{0}$. La proposición atómica *False* establece la noción de que algo no es válido, que no puede probarse; y asumir *False* permite deducir cualquier proposición P. El tipo vacío $\mathbf{0}$ no tiene habitantes; y asumir que existe un $a : \mathbf{0}$ permite deducir que hay un habitante para cualquier tipo.

- *True* $\leftrightsquigarrow \mathbf{1}$. La proposición atómica *True* establece la noción de que algo vale. Asumir *True* no permite deducir nada nuevo. El tipo unitario $\mathbf{1}$ tiene un constructor constante, es decir, puede

construirse un elemento $* : \mathbf{1}$ sin asumir nada. La regla de eliminación exige que para obtener un $a : A$ a partir de $\mathbf{1}$, debe proveerse el elemento a.

- $P \Rightarrow Q \leftrightsquigarrow A \to B$. Para probar $P \Rightarrow Q$ debe darse una prueba de Q asumiendo que vale P. Por su parte, la regla de introducción de $A \to B$ establece que para cada posible $a : A$ se obtiene un término con tipo B.

- $P \wedge Q \leftrightsquigarrow A \times B$. Para dar una prueba de la conjunción de dos proposiciones, debe darse una prueba de cada una de ellas. Si vale la conjunción, luego puede deducirse que vale cada una de las proposiciones. Equivalentemente, para construir un elemento del producto $A \times B$, debe proveerse un $a : A$ y un $b : B$, y a partir del par (a, b) podemos obtener cada elemento mediante las proyecciones.

- $P \vee Q \leftrightsquigarrow A + B$. Para probar una disjunción se debe dar una prueba de alguna de las dos proposiciones. Si se asume que vale $P \vee Q$, para concluir que vale una proposición R se debe proveer una prueba de $P \Rightarrow R$ y una de $Q \Rightarrow R$. El coproducto $A + B$ tiene reglas equivalentes: para construirlo debemos dar o bien un elemento de A o bien uno de B, y para descomponerlo en un tipo C necesitamos funciones de cada tipo en C.

- $\forall(x \in A).P(x) \leftrightsquigarrow \Pi_{(x:A)} B$. Para probar un "para todo" debe darse una prueba del predicado $P(x)$ para cada elemento posible en A. En teoría de tipos un predicado es representado por una familia de tipos $B : A \to \mathcal{U}$. Una función en $\Pi_{(x:A)} B$ determina una regla para obtener un elemento de $B\,a$ para cada posible $a : A$.

- $\exists(x \in A).P(x) \leftrightsquigarrow \Sigma_{(x:A)} B$. Para probar un existencial debe darse un $x \in A$ que cumpla el predicado P. Equivalentemente, para construir un elemento del tipo de pares dependiente $\Sigma_{(x:A)} B$ se debe proveer un $a : A$ y un $b : B\,a$.

A esta correspondencia se la conoce como el isomorfismo de Curry-Howard, y da lugar a lenguajes de programación en los cuales se pueden escribir programas que en su tipado establezcan condiciones lógicas que deben satisfacerse, y las mismas son chequeadas antes de ejecutarse.

1.3 FORMALIZACIÓN DE MATEMÁTICA

El estudio de las teorías fundacionales desde sus inicios en la teoría informal de conjuntos de Cantor hasta finalizada la llamada *crisis fundacional*, obtuvo como resultado un consenso general sobre cómo estudiar rigurosamente la matemática. A partir del desarrollo de la

lógica formal con la introducción de un lenguaje preciso gracias a los trabajos de Boole, Frege y Peano, luego con los avances de Whitehead y Russell en Principia Mathematica y finalmente con el programa de Hilbert, se establecen las reglas de juego mediante las cuales se desarrollan formalmente las teorías. Un sistema deductivo formal permite disponer de un sencillo y poderoso mecanismo para probar teoremas: habiendo definido algunos axiomas, todo el trabajo consiste en la transformación sintáctica de expresiones siguiendo reglas de inferencia. Por su lado, el desarrollo de la matemática constructiva con la teoría de tipos como fundamento, establece también un preciso lenguaje formal en el cual toda demostración puede chequearse mediante mecanismos sintácticos.

El avance de las Ciencias de la Computación en la segunda mitad del siglo XX introduce nuevas posibilidades en el estudio de la matemática. Por un lado las computadoras permiten realizar cálculos a gran escala, aliviando el trabajo tedioso de realizar a mano numerosas cuentas que en algunos casos vuelve imposible su realización. Ejemplos de este uso de la computación para demostrar teoremas son las pruebas del teorema de los 4 colores [2] y de la conjetura de Kepler [37]. Pero además de realizar cálculos, la computadora es una perfecta herramienta para realizar transformaciones mecánicas siguiendo reglas previamente establecidas. Es posible entonces definir un sistema deductivo formal y verificar automáticamente la validez de una prueba. Así se da origen a un área conocida como *mecanización de matemática* o también *formalización de matemática*.

Disponer de un sistema informático que chequee demostraciones tiene un primer beneficio inmediato: la confianza sobre las pruebas es mayor. Verificando *a mano* un núcleo sobre el cual se construye la herramienta (que idealmente debería ser pequeño), luego toda prueba aceptada por ésta será correcta. Un segundo beneficio recae no sólo en la posibilidad de comprobar resultados alcanzados previamente, sino también en el proceso mismo de obtención de éstos: la herramienta puede asistir al matemático en la construcción de la prueba y por ello se le da el nombre de *asistente de prueba*.

Una gran cantidad de asistentes de prueba se han desarrollado, con distintas bases fundacionales. El más antiguo, Automath [13], fue presentado en el año 1968 por Nicolaas de Bruijn y está basado en una variante del isomorfismo de Curry-Howard. Automath reviste de especial importancia siendo precursor del desarrollo de la teoría de tipos de Martin-Löf y del Cálculo de Construcciones [20] de Thierry Coquand. Basados en alguna variante de la teoría de tipos y en el isomorfismo de Curry-Howard se encuentran varios lenguajes o asistentes de prueba como NuPRL [45], Coq [99], Twelf [84] y Agda [11]. En otro grupo de lenguajes encontramos a los influenciados por la *lógica de funciones computables*, LCF [62], desarrollado por Robin Milner basado en el formalismo de Dana Scott [85] que consiste en una

teoría de tipos simple. La lista de lenguajes influenciados por LCF encuentra a HOL Light [39] e Isabelle [93] como los más destacados. Otro desarrollo que merece mención es el proyecto Mizar [56], llevado a cabo por un equipo de matemáticos, lingüistas y científicos de la computación polacos cuyo objetivo era analizar textos de matemática y encontrar una contraparte formal legible y fácil de comprender: comenzado en la década del 70, se han desarrollado lenguajes, asistentes de prueba, se han verificado una gran cantidad de artículos y la cantidad de autores que han trabajado en el proyecto ronda los 250. Actualmente cuenta con una librería de 1200 artículos formales escritos y verificados.

Un repaso minucioso por la historia de los asistentes de prueba desde sus orígenes hasta nuestros días puede encontrarse en [40]. Al momento de elegir en qué sistema formalizar matemática uno puede encontrar un amplio abanico de opciones, pero básicamente todas ellas se enmarcan en alguna de estas fundaciones: la lógica de alto orden o teoría de tipos simples, la teoría de tipos dependientes y la teoría de conjuntos. En su reciente artículo [75], Lawrence Paulson analiza las ventajas de los asistentes basados en la teoría de tipos simples, que si bien es menos expresiva que las otras dos, al tener una base lógica simple es más sencillo verificar la corrección de las implementaciones. Por otro lado la posibilidad de automatizar métodos de prueba es una gran ventaja de estos sistemas comparados con los basados en formalismos más complejos como la teoría de tipos dependientes.

Más allá del fundamento que se elija como base, los sistemas computacionales dedicados a la elaboración y verificación de resultados matemáticos pueden constituir una poderosa forma de hacer matemática. Idealmente estos sistemas deberían tener un formalismo de base que pueda verificarse con un esfuerzo razonablemente pequeño; su sintaxis debería ser amigable y de fácil abordaje, similar a la que se utiliza en pruebas en lápiz y papel; y pequeños pasos de prueba tediosos y repetitivos deberían poder automatizarse. Si toda la matemática conocida se desarrolla en tales sistemas, podría conformarse una enciclopedia unificada donde todos los resultados estén formalmente chequeados, la búsqueda de teoremas sea sencilla como en cualquier base de datos digital y puedan establecerse relaciones entre los trabajos, entre muchas otras ventajas.

Este escenario ideal que permitiría la formalización de matemática en evolucionados asistentes de prueba ha sido propuesto en diferentes proyectos. Ya en el proyecto de de Bruijn estaba la idea presente (una enciclopedia abierta donde los artículos son correctos gracias al chequeo automático). En el año 1994 un texto anónimo en forma de manifiesto propuso el desarrollo de un proyecto en el cual toda la matemática sea formalizada y describió detalladamente cuáles serían los objetivos, problemas posibles y pasos a seguir. El nombre tenta-

tivo que los autores anónimos le dieron a esta idea es QED [90] y en sus motivaciones enumeran las ventajas de tener un sistema en el cual desarrollar matemática certificada, tal como mencionamos previamente, beneficiar al desarrollo de nuevas tecnologías donde para asegurar su corrección es fundamental involucrar pruebas matemáticas, y otros siete ítems entre los que mencionan también beneficios culturales y educativos. Más recientemente, surgieron otros proyectos (Formal Abstracts [38] y Alexandria [74]) con objetivos parecidos, quizás menos ambiciosos.

Si bien podría parecer que los asistentes de prueba constituyen *la* forma de desarrollar matemática por estos días, la realidad muestra que aún falta camino por recorrer. Freej Wiedijk analiza en [96] el avance logrado hacia un proyecto como QED una década después de la aparición del manifiesto, y veinte años después de la misma, en un workshop en honor a dicho aniversario, vuelve a repasar[5] el estado de la formalización de matemática revisando los objetivos trazados en QED y lo que falta por alcanzar [41]. Por su parte Jeremy Avigad dedica unas palabras, en la conclusión de [3], a la dificultad de que las formalizaciones en sistemas computacionales sean algo habitual en la actividad del matemático moderno. Estos análisis coinciden en que los asistentes de pruebas, aún por estos días, siguen siendo complicados de utilizar. Si realizar una demostración en estos sistemas requiere de un esfuerzo mucho mayor al que los matemáticos realizan en sus tareas habituales, será muy difícil que la aceptación sea generalizada. El desafío sigue abierto y las comunidades de algunos asistentes de prueba han crecido considerablemente en los últimos años (como es el caso de Coq e Isabelle, y en menor escala Agda) mientras que conferencias y revistas dedicadas exclusivamente a la formalización de matemática ayudan a que la atención dedicada a estas tecnologías vaya en aumento. Si bien llegar al ideal de un sistema unificado, como propone QED, puede parecer utópico, el hecho de que más científicos, sobre todo en las nuevas generaciones, se formen en el estudio de la matemática formal permite vislumbrar un panorama más optimista.

1.4 CONTRIBUCIONES DE LA TESIS

El rumbo del doctorado que finaliza con esta tesis fue modificándose durante la marcha, comenzando por la exploración de las posibilidades de los lenguajes con tipos dependientes para definir compiladores correctos por construcción y derivando luego en la utilización de asistentes de prueba para formalizar matemática: la teoría de Álgebra Universal, fundamento teórico para el enfoque algebraico de definición de compiladores; y la teoría de conjuntos, en particular

5 junto a John Harrison y Josef Urban

la técnica de forcing para extender modelos de *ZF*. En esta sección damos un vistazo general sobre la estructura de la tesis.

Gran parte de los trabajos realizados fueron implementados en el lenguaje Agda. En el capítulo 2 presentamos una introducción al lenguaje en forma de pequeño tutorial orientado a un programador funcional.

En el capítulo 3 presentamos un enfoque para compiladores correctos por construcción mediante el uso de tipos dependientes utilizando Agda, trabajo publicado en [66]. En el mismo exploramos cómo podemos utilizar tipos indexados para definir las semánticas de los lenguajes, y así poder asegurar la preservación de éstas estáticamente. Si bien las ideas presentadas pueden encontrarse en la teoría de Ornaments [58], la presentación de nuestro trabajo provee un enfoque metodológico de fácil abordaje concentrado específicamente en el problema de la compilación.

El fundamento teórico de la metodología presentada en el capítulo 3 consiste en considerar a los lenguajes como álgebras de términos de signaturas multi-sort, lo que nos llevó al estudio del Álgebra Universal y a su posterior formalización en Agda, siendo la primera en este lenguaje y una de las pocas existentes en teoría de tipos. En el capítulo 4 desarrollamos este trabajo, donde además de los conceptos y resultados básicos, implementamos un cálculo ecuacional y las nociones de morfismos entre signaturas y álgebras reducto. Dicho aporte fue publicado en [34].

El capítulo 5 contiene el último aporte de esta tesis: la formalización en el asistente de pruebas Isabelle de gran parte de la maquinaria de *forcing*, herramienta matemática que permite extender modelos transitivos contables de ZF, con miras a una completa formalización de la independencia de la hipótesis del continuo. Los preliminares de dicho trabajo fueron publicados en [36] y los últimos avances realizados incluyen la prueba de validez del esquema de separación en la extensión genérica [35].

Concluimos esta tesis con algunas palabras finales en el capítulo 6. El código fuente de cada desarrollo presentado aquí se encuentra disponible en la url `http://www.famaf.unc.edu.ar/~gunther/phd-thesis`.

2

AGDA: LENGUAJE DE PROGRAMACIÓN Y ASISTENTE DE PRUEBAS

Agda es un lenguaje de programación con tipos dependientes que implementa la teoría de tipos intuicionista de Martin-Löf. Gracias al isomorfismo de Curry-Howard un programa puede expresar en su tipo proposiciones lógicas, de manera de asegurar estáticamente propiedades que se quieren satisfacer. Toda función definida en el lenguaje puede ser vista como una prueba de que la proposición que se corresponde con su tipo vale, y por ello Agda, junto al entorno interactivo del editor Emacs [83], puede considerarse un *asistente de pruebas*. En los aportes que presentamos en esta tesis tenemos dos desarrollos implementados en este lenguaje. En uno de ellos definimos un compilador certificado y en el otro formalizamos la teoría de álgebras universales, por lo que explotamos las dos facetas de Agda: lenguaje de programación y asistente de pruebas.

En este breve tutorial daremos una mirada general sobre las principales características de Agda, brindando ejemplos sencillos para lo cual asumimos que el lector tiene experiencia en algún lenguaje de programación funcional como Haskell.

2.1 INTRODUCCIÓN

En lenguajes de programación funcionales con un sistema de tipos como Haskell existe una división entre los tipos (`Int`, `Bool`, `String`, etc) y los valores (`0`, `True`, `"Famaf"`, etc). En cambio en un lenguaje con tipos dependientes tal separación es menos clara.

Consideremos como ejemplo un tipo para representar vectores de longitud fija. En Haskell podemos representar las listas de manera sencilla con un tipo inductivo, y podemos definir la función `head` que dada una lista obtiene el primer elemento. Podemos entonces tener en un programa la expresión `head []`, cuyo tipado es correcto, pero que al ejecutarse dará un error: no puede obtenerse el primer elemento de una lista vacía. ¿Podríamos definir un tipo de datos más específico que el de las listas, expresando la cantidad de elementos que debe tener? En Haskell haríamos algo así:

```
data Zero

data Suc n
```

```
data Vec a n where
    Empty :: Vec a Zero
    Cons  :: a -> Vec a n -> Vec a (Suc n)
```

Lo que necesitamos es tener alguna manera de introducir *etiquetas* que nos permitan expresar en el tipo `Vec` cuántos elementos contendrá la colección. Para ello definimos `Zero` y `Suc n` como tipos sin constructores (utilizando polimorfismo para expresar que `Suc` es un constructor de tipo, para cada tipo `n`), y luego el tipo polimórfico `Vec a n` representará colecciones de elementos de tipo `a` con longitud `n` (que también es cualquier tipo). El truco está en que mirando los constructores de `Vec`, la única manera de construir elementos es instanciando `n` con `Zero` o alguna cadena `Suc (Suc (...(Zero)...))`, representando así los números naturales que indicarán cuántos elementos debe tener un vector.

Con el tipo `Vec` definido, podemos dar una función total que toma el primer elemento de un vector:

```
head :: Vec a (Suc n) -> a
head (Cons e _) = e
```

El tipo de `head` ahora es una función que toma un vector cuyo tipo tiene la etiqueta `Suc n`, y sólo `Cons` permite construir un elemento con ese tipo. Hemos eliminado la posibilidad de que se ejecute un programa con la expresión `head Empty`.

Si bien este artilugio nos permite chequear estáticamente el error provocado por pedir el primer elemento de una colección vacía, la manera de conseguirlo no es tan clara y no hay una manera obvia de definir otras funciones como por ejemplo la concatenación de dos vectores con tamaños n y m respectivamente: ¿cuál debería ser la etiqueta que represente el tamaño del vector resultante? Un análisis detallado de las dificultades para realizar este tipo mecanismos en Haskell puede encontrarse en [52]. Un lenguaje con tipos dependientes, en cambio, permite resolver esta clase de problemas de manera sencilla gracias al poder de expresividad del sistema de tipos.

2.2 TIPOS DE DATOS Y PATTERN MATCHING

En Agda, Set corresponde con los universos de tipos, en donde hay una jerarquía $\mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 : \ldots$ (tal cual explicamos en la sección 1.2). Set es equivalente a Set_0. En el texto de esta tesis ingoraremos especificar el nivel de los universos, aún cuando fuera necesario, por cuestiones de simplicidad. En el código fuente de las implementaciones podrá encontrarse el desarrollo completo.

La manera de introducir tipos inductivos en Agda es mediante la declaración de un *datatype*, especificando sus constructores:

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
```

Nat tiene tipo Set y sus constructores son la constante zero y la función suc, que dado un elemento de tipo Nat construye otro.

Las funciones se definen de la misma manera que en Haskell, utilizando pattern matching, con la única salvedad de que en Agda éstas deben ser totales: los patrones deben cubrir todos los posibles elementos del tipo. Esto se debe a que, como en los lenguajes con tipos dependientes los valores pueden ocurrir en los tipos, el type-checker debe normalizarlos y entonces toda función debe estar definida para cualquier caso y más aún, debe terminar.

Como ejemplo de función, definimos la suma de dos naturales. Agda tiene una sintaxis muy flexible, permitiendo utilizar cualquier secuencia de símbolos sin espacios como nombre de función, incluyendo caracteres unicode; y también permite declarar funciones como operadores infijos o sufijos:

```
_+_ : Nat → Nat → Nat
zero + m = m
suc n + m = suc (n + m)
```

En lenguajes como Haskell podemos definir tipos de datos polimórficos especificando mediante una *variable de tipo* el parámetro, que luego podrá instanciarse con algún tipo. En Agda podemos declarar un datatype con parámetros que pueden ser de cualquier tipo. En el caso de las listas el parámetro tendrá tipo Set:

```
data List (A : Set) : Set where
  []  : List A
  _:_ : A → List A → List A
```

Dado que en Agda todas las funciones deben ser totales, no podremos directamente dar una función `head` como en Haskell, donde no definíamos regla para el constructor de la lista vacía.

Tanto el tipo List como los números naturales ℕ son tipos definidos en la librería estándar de Agda. Para el caso de los naturales tenemos definidas constantes numéricas 0, 1, 2, ... obteniendo una sintaxis más fácil de utilizar.

2.3 FAMILIAS DE TIPOS DE DATOS

Para solucionar el problema de la función `head` aplicada a la lista vacía vimos previamente que se necesitaba tener un tipo más específico, en el cual de alguna manera pueda etiquetarse la cantidad de elementos que tiene la colección. En Haskell pudimos hacer esto introduciendo tipos sin constructores que representan a los números

naturales, y luego definiendo un tipo polimórfico con una variable que sería instanciada por estos tipos introducidos. En Agda podemos definir tipos *indexados*, es decir, una familia que para cada elemento de algún tipo, obtiene otro tipo. Los vectores son un ejemplo de éstos:

```
data Vec (A : Set) : ℕ → Set where
  empty : Vec A zero
  cons : (n : ℕ) → A → Vec A n → Vec A (suc n)
```

El tipo Vec está parametrizado en A e indexado por el tipo ℕ, lo que significa que dado un tipo A, Vec A es una función que devuelve un tipo para cada elemento en ℕ. Los constructores empty y cons son las únicas maneras de obtener elementos de estos tipos obtenidos por Vec A, y así empty construye un elemento de tipo Vec A zero, mientras que cons permite obtener vectores indexados en naturales mayores a cero.

Un detalle que tiene la definición anterior es que para el caso del constructor recursivo, debemos proveer un número natural que permita expresar la longitud del vector al que se le agrega el elemento. Esto no es deseable: quisiéramos poder utilizar los vectores de la misma manera que las listas, donde para el constructor no vacío, se debe proveer el elemento que se quiere agregar y la lista correspondiente. Pero, ¿cómo hacemos en el caso de Vec donde necesitamos el valor con el que se indexa el vector? Podemos observar que, dado un vector, podemos calcular su longitud y entonces el número natural con el cual se debe indexar el tipo puede *inferirse*. Para estos casos podemos declarar que el argumento es *implícito*, poniéndolo entre llaves {}:

```
data Vec (A : Set) : ℕ → Set where
  empty : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

y para definir un elemento, no es necesario especificar el argumento implícito:

```
vec_ex : Vec ℕ 2
vec_ex = cons 0 (cons 1 empty)
```

2.4 FUNCIONES Y PARES DEPENDIENTES

Funciones

Los tipos indexados, o familias de tipos, dan lugar a la definición de funciones que dependen de valores. En la sección 1.2 describimos el tipo $\Pi_{(x:A)} B$ de las funciones dependientes, donde A es un tipo y B otro donde puede ocurrir libremente la variable x. La función head de los vectores es una función dependiente:

```
head : {A : Set} {n : ℕ} → Vec A (suc n) → A
head (cons x _) = x
```

esta función toma como argumentos un tipo A y un natural n, obteniendo como resultado una función de vectores de elementos de A con longitud (suc n), en A. Como ambos argumentos pueden inferirse, pueden colocarse como implícitos.

Como en Agda toda función es total, una función dependiente obtiene un valor del tipo resultado *para cada* elemento del dominio.

Producto

Agda tiene una construcción primitiva de registro o *record* para definir tipos, conformados por una colección de campos (*fields*), cada uno posiblemente de un tipo distinto, que puede depender de los definidos previamente. El tipo de los pares dependientes se define utilizando records, de esta manera:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst
```

Un elemento de Σ A B será un par (a , b) donde a tiene tipo A y b tiene tipo (B a). El par no dependiente se define a partir del tipo Σ, donde B es constante:

```
_×_ : (A : Set) → (B : Set) → Set
A × B = Σ A (λ _ → B)
```

2.5 TIPOS PROPOSICIONALES

El isomorfismo de Curry-Howard establece una correspondencia entre los tipos y las proposiciones lógicas. Una proposición representada con un tipo A es válida si el mismo es habitado, es decir, hay un $a : A$.

Las proposiciones atómicas *False* y *True* se representan, respectivamente, con un tipo que no tenga elementos y con uno que tenga exactamente uno.

```
data ⊥ : Set where
```

el tipo ⊥ no tiene ningún constructor, por lo tanto es imposible obtener un elemento. Y si uno asume que tiene un elemento de ⊥, luego puede asumir que tiene un elemento de cualquier tipo, equivalente a

la proposición *False* → P. Esto se implementa en Agda mediante el uso del *pattern absurdo*: cuando se define una función donde un argumento no tiene constructores, entonces no se puede dar una regla de computación.

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

El tipo unitario se define dando un único constructor constante:

```
data ⊤ : Set where
  tt : ⊤
```

como siempre podemos construir el elemento tt, el tipo ⊤ se corresponde con la proposición *True*, que siempre es válida.

El resto de conectivos lógicos se corresponde con tipos de la manera que presentamos en 1.2: la conjunción con el producto no dependiente, la disjunción con el coproducto, la implicación con las funciones no dependientes, el cuantificador universal con las funciones dependientes y el existencial con el producto dependiente.

Un tipo interesante de definir es el de las funciones de algún tipo en ⊥, que se corresponde con la implicación P → *False* y es lógicamente equivalente a ¬P

```
¬_ : Set → Set
¬ P = P → ⊥
```

y luego podemos definir un tipo que expresa que una proposición es decidible: dada P, se puede probar que es verdadera o que es falsa. El tipo Dec está parametrizado en un conjunto P y tiene dos constructores: uno que pide un elemento de tipo P, y otro con un argumento de tipo $\neg\ P$:

```
data Dec (P : Set) : Set where
  yes : P → Dec P
  no : ¬ P → Dec P
```

2.6 PRUEBAS

Veamos ahora un ejemplo de un tipo representando un predicado sobre los números naturales: la paridad. Podemos pensar la definición de este predicado de manera recursiva, teniendo que 0 es par, y luego dado un número par n, el sucesor del sucesor de n, *suc* (*suc* n), también es par. En Agda definimos este predicado mediante un tipo inductivo indexado en los naturales con dos constructores que expresan las reglas que acabamos de describir:

```
data Even : ℕ → Set where
  pz : Even zero
  psuc : {n : ℕ} → Even n → Even (suc (suc n))
```

Dado $n : \mathbb{N}$, si podemos construir un elemento con tipo Even n habremos demostrado que n es par. Por ejemplo 0, 2 y 4 son pares:

```
zeroEven : Even 0
zeroEven = pz

twoEven : Even 2
twoEven = psuc pz

fourEven : Even 4
fourEven = psuc twoEven
```

Para la primera prueba tenemos un regla que lo asegura: cero es par; y para probar que un número n distinto de 0 es par, debemos dar una prueba de que $n - 2$ es par. Podríamos ver que 3 no es par: si lo fuera deberíamos poder probar que 1 es par, pero como 1 es distinto de 0 y no puede construirse mediante la aplicación del constructor *suc* aplicado a otro natural construido con *suc*, resulta que es imposible. Esta descripción de prueba se traduce a Agda de esta manera:

```
threeNotEven : ¬ (Even 3)
threeNotEven (psuc ())
```

El tipo ¬ (Even 3) es igual por definición a (Even 3) → ⊥, es decir una función que tome una prueba de que 3 es par y obtenga un elemento del tipo vacío. Para ello intentaríamos hacer pattern matching en Even 3 mediante el constructor psuc que necesita un argumento con tipo Even 1. Pero un elemento de dicho tipo es imposible de construir y por ello utilizamos el pattern absurdo.

Para realizar pruebas en Agda disponemos del entorno interactivo para el editor Emacs, que asiste al usuario para realizar pruebas. Si en una definición de función uno escribe ? luego del = (se dice que uno deja un *hueco*), dispondrá de distintas opciones como averiguar el tipo del elemento que debe definirse, realizar automáticamente pattern matching sobre algún argumento de la función, intentar obtener automáticamente el término, entre muchos otros[1].

Teniendo definido el tipo Even hemos construido pruebas de que algunos valores naturales son pares y también mostramos una prueba de que un elemento no es par, utilizando el tipo de la negación. Podemos definir una función que *para todo* natural obtenga o bien una prueba de que éste es par, o bien una prueba de que no lo es. Para hacerlo en Agda debemos dar una función que tome un ($n : \mathbb{N}$) y obtenga una prueba en (Even n):

```
isEven : (n : ℕ) → Dec (Even n)
```

1 Una guía de los comandos para utilizar Agda como un asistente de pruebas puede encontrarse en https://agda.readthedocs.io/en/v2.5.2/tools/emacs-mode.html.

Podemos hacer pattern matching diferenciando los casos 0, 1 y (psuc (psuc (*n*))):

```
isEven 0             = ?
isEven (suc 0)       = ?
isEven (suc (suc n)) = ?
```

Para cada hueco debemos dar un elemento en Dec utilizando el constructor yes o el no. Los dos primeros se completan de manera sencilla: La prueba de que 0 es par es directa por definición, como vimos antes, y la prueba de que 1 no es par es una función definida con un pattern absurdo, ya que es imposible construir un elemento de Even 1:

```
isEven 0             = yes pz
isEven (suc 0)       = no (λ ())
isEven (suc (suc n)) = ?
```

Para el caso recursivo debemos analizar si *n* es o no es par, y para cada caso decidir si (suc (suc *n*)) lo es. En Agda podemos hacer pattern matching sobre expresiones, de la misma manera que la *case expression* de Haskell, utilizando una expresión with:

```
isEven 0             = yes pz
isEven (suc 0)       = no (λ ())
isEven (suc (suc n)) with (isEven n)
isEven (suc (suc n)) | yes nEven    = ?
isEven (suc (suc n)) | no nNotEven = ?
```

Si isEven *n* vale entonces tenemos un elemento (*nEven* : Even *n*), en caso contrario tenemos un elemento (*nNotEven* : ¬ Even *n*), es decir, una función de Even *n* en ⊥. Para el primer caso, como *n* es par, podemos probar que (suc(suc(*n*))) lo es utilizando el constructor psuc. Para el segundo caso tendremos que asumir (Even *n*) y luego aplicamos la función *nNotEven* para obtener ⊥, completando la definición:

```
isEven : (n : ℕ) → Dec (Even n)
isEven zero          = yes pz
isEven (suc 0)       = no (λ ())
isEven (suc (suc n)) with (isEven n)
isEven (suc (suc n)) | yes nEven    = yes (psuc nEven)
isEven (suc (suc n)) | no nNotEven = no n+2NotEven
  where n+2NotEven : ¬ Even (suc (suc n))
        n+2NotEven (psuc nEven) = nNotEven nEven
```

Así hemos probado que para todo número natural podemos decidir si es par o si no lo es, dando una prueba para cada caso, y concluimos la introducción al lenguaje. Un tutorial más extenso puede encontrarse en [64].

3

UN ENFOQUE PARA COMPILADORES CORRECTOS POR CONSTRUCCIÓN

Un compilador es una herramienta fundamental relacionada a cualquier lenguaje de programación: Es el programa que traduce el código fuente escrito en un lenguaje de alto nivel (como un lenguaje imperativo simple) a un lenguaje de bajo nivel relacionado a la arquitectura de una máquina (como el código assembly). La corrección de un compilador reviste una importancia crucial: un error en el compilador puede invalidar propiedades probadas sobre el programa. Existe abundante literatura sobre cómo probar un compilador correcto, empezando por McCarthy y Painter [59] en 1967 hasta el desarrollo de un compilador certificado completamente para un subconjunto del lenguaje C [51]. En general, el mecanismo para asegurar la corrección es el mismo: Primero se define el compilador como una función que traduce programas bien formados escritos en el lenguaje de alto nivel (ésto podría implicar un primer chequeo de tipos) al lenguaje de bajo nivel, y luego se demuestra que la semántica de cada término en el alto nivel se corresponde con la semántica del término traducido en el bajo nivel. De esta manera entonces, si se comete un error en la definición de la traducción de los términos, sólo podría detectarse en el proceso de verificación al intentar realizar la prueba.

Un enfoque alternativo consiste en aprovechar las ventajas de los lenguajes con tipos dependientes de manera de especificar a nivel de los tipos la información semántica de cada lenguaje y de esta manera el tipo del compilador capturará la corrección. En la conferencia ICFP de 2012, Conor McBride introdujo un ejemplo en el cual implementa un compilador, definiendo el lenguaje de bajo nivel con un sistema de tipos de manera tal que el tipo de una instrucción es decorado con su semántica operacional [57]. Realizando el mismo trabajo con la semántica del lenguaje de alto nivel y teniendo una relación entre ambas semánticas, el compilador puede expresar a nivel de tipos su corrección, de manera que solo estará bien tipado si es correcto.

En este capítulo estudiaremos en detalle esa propuesta con miras a obtener una metodología para definir compiladores correctos por construcción explotando las ventajas de los tipos dependientes. Para ello presentaremos el enfoque aplicado a un compilador de un lenguaje de expresiones aritméticas a un lenguaje de máquina basada en stack y luego extenderemos los lenguajes definiendo un compilador correcto por construcción de un lenguaje imperativo simple a un lenguaje intermedio con primitiva de loop. Finalmente definiremos un compilador de este lenguaje intermedio a un código de máquina si-

$$\frac{}{\vdash n : nat} \qquad \frac{\vdash e_1 : nat \qquad \vdash e_2 : nat}{\vdash e_1 \oplus e_2 : nat} \qquad \frac{\vdash e_1 : nat \qquad \vdash e_2 : nat}{\vdash e_1 \stackrel{\circ}{=} e_2 : bool}$$

Figura 1: Sistema de tipos del lenguaje fuente

milar al assembly analizando hasta qué punto nuestro enfoque puede aplicarse a ejemplos más realistas.

3.1 EL ENFOQUE APLICADO A UN LENGUAJE DE EXPRESIONES

Veamos con un pequeño ejemplo cómo podemos construir un compilador correcto por construcción. Primero lo definiremos en la manera habitual, junto con una prueba de corrección.

Los lenguajes

Consideremos un lenguaje de expresiones aritméticas con constantes numéricas, un operador de suma y un operador de igualdad[1]:

$$e ::= \quad n \mid e_1 \oplus e_2 \mid e_1 \stackrel{\circ}{=} e_2$$

En Agda lo implementamos de esta manera:

```
data Expr : Set where
  |_|   : ℕ → Expr
  _⊕_   : (e₁ : Expr) → (e₂ : Expr) → Expr
  _≟_   : (e₁ : Expr) → (e₂ : Expr) → Expr
```

Esta gramática nos permite construir expresiones que no están bien tipadas, como por ejemplo $(0 \stackrel{\circ}{=} 1) \oplus 2$. Para evitar esas situaciones anómalas definimos entonces un sistema de tipos en la figura 1, que puede ser implementado directamente en Agda:

```
data Type : Set where
  nat  : Type
  bool : Type

data ⊢_:_ : Expr → Type → Set where
  tnat  : ∀ {n} → ⊢ | n | : nat
  tplus : ∀ {e₁ e₂} → ⊢ e₁ : nat → ⊢ e₂ : nat → ⊢ e₁ ⊕ e₂ : nat
  teq   : ∀ {e₁ e₂} → ⊢ e₁ : nat → ⊢ e₂ : nat → ⊢ e₁ ≟ e₂ : bool
```

1 Si bien el operador de igualdad no tendría mucha utilidad aquí, lo introducimos para mostrar un ejemplo con más de un tipo de datos.

Las expresiones pueden tener tipo nat o bool y un elemento del datatype $\vdash e : t$ es una prueba de que la expresión e tiene tipo t.

Para poder hablar de corrección de un compilador necesitamos tener una noción de semántica para los lenguajes involucrados. Para este lenguaje de expresiones es natural dar una semántica funcional, para lo cual primero damos una interpretación para los tipos:

```
ST : Type → Set
ST nat  = ℕ
ST bool = Bool
```

La ventaja de contar con un sistema de tipos es que alcanza con dar semántica sólo a aquellas expresiones que estén bien tipadas:

```
E⟦_⟧ : ∀ {t} → (e : Expr) → ⊢ e : t → (ST t)
E⟦ ∣ n ∣ ⟧ tnat = n
E⟦ e₁ ⊕ e₂ ⟧ (tplus p₁ p₂) = (E⟦ e₁ ⟧ p₁) + (E⟦ e₂ ⟧ p₂)
E⟦ e₁ ≟ e₂ ⟧ (teq p₁ p₂)   = (E⟦ e₁ ⟧ p₁) == (E⟦ e₂ ⟧ p₂)
```

Pasemos ahora a describir una máquina basada en pila que usaremos para compilar el lenguaje de alto nivel. El lenguaje de la máquina consiste de un conjunto de instrucciones cuya ejecución manipula la pila. Podemos describirlo así:

$$c ::= \mathtt{push}\ v \mid \mathtt{add} \mid \mathtt{eq} \mid c_1, c_2$$

La máquina es capaz de poner un valor v en el tope, sumar o chequear la igualdad de los dos valores que se encuentren en el tope de la pila. Por último dos instrucciones pueden ser sucedidas en forma de secuencia, y contamos con un constructor binario para ello, a diferencia de los lenguajes de ensamblador donde tenemos instrucciones simples agrupadas en una lista. Definimos este lenguaje en Agda de esta manera:

```
data Code : Set where
  push : ℕ → Code
  add  : Code
  eq   : Code
  _,_  : Code → Code → Code
```

Al igual que en el lenguaje de alto nivel, podemos definir aquí un sistema de tipos. Si, por ejemplo al momento de ejecutar la instrucción add la pila no cuenta con dos elementos de tipo nat en el tope, la ejecución no puede realizarse. El sistema de tipos de la figura 2 evitará que se produzcan estas situaciones de *stack-underflow*. Los juicios de tipado son de la forma $st \vdash c \rightsquigarrow st'$, donde st, st' son *stack-types* :

$$\frac{}{st \vdash |push|\, n \rightsquigarrow |nat| :: st} \qquad \frac{}{|nat| :: |nat| :: st \vdash |add| \rightsquigarrow |nat| :: st}$$

$$|nat| :: |nat| :: st \vdash |eq| \rightsquigarrow |bool| :: st \qquad \frac{st \vdash c_1 \rightsquigarrow st' \qquad st' \vdash c_2 \rightsquigarrow st''}{st \vdash c_1, c_2 \rightsquigarrow st''}$$

Figura 2: Sistema de tipos para el lenguaje target

$$\frac{}{\langle |push|\, n, s\rangle \Downarrow_{\mathcal{M}} n \rhd s} \qquad \frac{}{\langle |add|, n \rhd m \rhd s\rangle \Downarrow_{\mathcal{M}} (n + m) \rhd s}$$

$$\frac{}{\langle |eq|, n \rhd m \rhd s\rangle \Downarrow_{\mathcal{M}} (n \equiv m) \rhd s} \qquad \frac{\langle c_1, s\rangle \Downarrow_{\mathcal{M}} s' \qquad \langle c_2, s'\rangle \Downarrow_{\mathcal{M}} s''}{\langle c_1, c_2, s\rangle \Downarrow_{\mathcal{M}} s''}$$

Figura 3: Semántica operacional del lenguaje target

listas con los tipos de los elementos que contiene la pila en el orden que aparecen; st representa los tipos de la pila antes de ejecutar la instrucción c, y st' los tipos de la pila resultante luego de la ejecución. Lo implementamos en Agda mediante un tipo indexado en dos stack-types y un programa:

```
StackType : Set
StackType = List Type

data _⊢_⇝_ : StackType → Code → StackType → Set where
  tpush : ∀ {st} {n : ℕ} → st ⊢ push n ⇝ (nat : st)
  tadd  : ∀ {st} → (nat : nat : st) ⊢ add ⇝ (nat : st)
  teq   : ∀ {st} → (nat : nat : st) ⊢ eq ⇝ (bool : st)
  tseq  : ∀ {st st' st''} {c1 c2} →
            st ⊢ c1 ⇝ st' → st' ⊢ c2 ⇝ st'' → st ⊢ c1 , c2 ⇝ st''
```

En la figura 3 definimos la semántica operacional para el lenguaje de bajo nivel. Escribimos

$$\langle c, s\rangle \Downarrow_{\mathcal{M}} s'$$

para referir que la evaluación de c en la pila inicial s termina en la pila s'. Notar que la pila puede contener valores naturales o booleanos, por lo tanto las reglas para las instrucciones add y eq sólo tendrán sentido si en en el tope de la pila hay dos valores naturales. Podemos definir en Agda esta semántica sólo para configuraciones con los tipos adecuados donde representamos las pilas como listas heterogéneas, mediante un tipo indexado en los stack-type:

```
data Stack : (st : StackType) → Set where
  ϵ : Stack []
  _▷_ : ∀ {t} {st} → (ST t) → Stack st → Stack (t : st)

data ⟨_,_⟩_⇒_ : ∀ {st st'} → (c : Code) → st ⊢ c ⇝ st' →
                          Stack st → Stack st' → Set where
  pushR : ∀ {n st} {s : Stack st} →
          ⟨ (push n) , tpush ⟩ s ⇒ (n ▷ s)
  addR : ∀  {m n st} {s : Stack st} →
          ⟨ add , tadd ⟩ (m ▷ (n ▷ s)) ⇒ ((m + n) ▷ s)
  eqR  : ∀  {m n st} {s : Stack st} →
          ⟨ eq , teq ⟩ (m ▷ (n ▷ s)) ⇒ ((m == n) ▷ s)
  seqR : ∀  {c₁ c₂ st st' st'' p₁ p₂}
          {s : Stack st} {s' : Stack st'} {s'' : Stack st''} →
          ⟨ c₁ , p₁ ⟩ s ⇒ s' → ⟨ c₂ , p₂ ⟩ s' ⇒ s'' →
          ⟨ (c₁ , c₂) , (tseq p₁ p₂) ⟩ s ⇒ s''
```

El compilador y la prueba de corrección

Dados los lenguajes fuente y target, con sus sistemas de tipos y semánticas, procedemos a definir el compilador junto con su prueba de corrección. Definiremos la compilación sólo para expresiones que estén bien tipadas, por lo tanto para obtener un programa en el lenguaje de bajo nivel debemos tener una prueba de tipado de la expresión:

```
compile : ∀ {t} (e : Expr) → ⊢ e : t → Code
compile | n | tnat              = push n
compile (e₁ ⊕ e₂) (tplus p₁ p₂) = compile e₁ p₁ , compile e₂ p₂ , add
compile (e₁ ≟ e₂) (teq p₁ p₂)   = compile e₁ p₁ , compile e₂ p₂ , eq
```

La definición del compilador es simple: una constante natural se compila a la instrucción que consiste en poner esa constante en el tope de la pila; la suma de dos expresiones se compila a la secuencia consistente de la compilación del primer sumando, seguido del segundo, seguido de la instrucción `add` y de igual manera con la igualdad. Una propiedad deseada y que este compilador satisface es la de preservación de tipado, cuya prueba omitimos ya que no aporta nada interesante:

```
typepres : ∀ {t} {st} {e : Expr} → (tp : ⊢ e : t) → st ⊢ compile e tp ⇝ (t : st)
```

De aquí obtenemos dos hechos: (i) la compilación de la expresión e de tipo t resulta en un programa el cual puede tiparse con cualquier

stack-type inicial st; y (ii) la ejecución de éste deja en el tope de la pila un valor de tipo t.

La corrección del compilador expresa que el resultado de ejecutar el programa obtenido de compilar una expresión bien tipada e comenzando en la pila s es la misma pila consistente del valor v en el tope seguido de s, donde v es el valor semántico de e. Esta prueba puede realizarse por inducción en e sin ninguna dificultad y por ello solo mostramos su enunciado:

```
correct : ∀ {t} {st} {e : Expr} {tp : ⊢ e : t} {s : Stack st} →
            ⟨ (compile e tp) , (typepres tp) ⟩ s ⇒ ((E⟦ e ⟧ tp) ▷ s)
```

Así hemos completado la definición del compilador, junto con su prueba de corrección.

Nos concentremos en la metodología que seguimos para desarrollar este compilador correcto: hemos definido la sintaxis de los lenguajes, las relaciones de tipado, la semántica de cada lenguaje solo para expresiones bien tipadas y la función de compilación, de manera separada. Las propiedades de preservación de tipado y corrección fueron formuladas como predicados *externos* que relacionan la función de compilación con los sistemas de tipado y con las relaciones semánticas de los lenguajes, respectivamente. Ambas propiedades no están aseguradas *por construcción*, sino que deben ser chequeadas independientemente de la definición del compilador. A este enfoque se lo suele llamar *externalista*. Como alternativa, a continuación presentamos un enfoque *internalista* en el cual las propiedades son aseguradas por construcción, para ello internalizaremos primero las relaciones de tipado, y luego las semánticas.

Internalización del tipado

Tomando provecho de los tipos dependientes, vamos a *ornamentar* los datatypes que definen la sintaxis de los lenguajes de manera que las relaciones de tipado estén definidas junto con las correspondientes gramáticas.

Definimos primero las expresiones bien tipadas, como un datatype indexado en los tipos `nat` y `bool`:

```
data Expr_t : Type → Set where
  |_|   : ℕ → Expr_t nat
  _⊕_ : (e1 : Expr_t nat) → (e2 : Expr_t nat) → Expr_t nat
  _≟_ : (e1 : Expr_t nat) → (e2 : Expr_t nat) → Expr_t bool
```

Un elemento e del tipo $\mathsf{Expr}_t\ t$ en Agda corresponde al juicio de tipado $\vdash e : t$. Decimos que Expr_t es una versión *ornamentada* del tipo Expr. Debido a que la relación de tipado para las expresiones es

dirigida por sintaxis, la definición de Expr_t es inmediata: cada constructor de las expresiones se corresponde con una regla de tipado. Esta característica es fundamental en el enfoque internalista.

La relación entre una expresión en Expr y su versión internalizada la evidenciamos con una función de *lifting*:

```
_↑t_ : ∀ {t} → (e : Expr) → ⊢ e : t → Exprt t
| x | ↑t tnat                = | x |
(e1 ⊕ e2) ↑t (tplus p1 p2) = (e1 ↑t p1) ⊕ (e2 ↑t p2)
(e1 ≟ e2) ↑t (teq p1 p2)   = (e1 ↑t p1) ≟ (e2 ↑t p2)
```

Gracias a que solo definimos expresiones bien tipadas, no hay inconvenientes para definir la semántica:

```
fnat : ℕ → ℕ
fnat n = n

fplus : ℕ → ℕ → ℕ
fplus n1 n2 = n1 + n2

feq : ℕ → ℕ → Bool
feq n1 n2 = n1 == n2


Et⟦_⟧ : ∀ {t} → Exprt t → (ST t)
Et⟦ | n | ⟧      = fnat n
Et⟦ e1 ⊕ e2 ⟧ = fplus Et⟦ e1 ⟧ Et⟦ e2 ⟧
Et⟦ e1 ≟ e2 ⟧ = feq   Et⟦ e1 ⟧ Et⟦ e2 ⟧
```

Más adelante quedará clara la razón por la cual definimos la semántica modularizando de esta manera.

Es inmediato ver que las definiciones semánticas que dimos para el lenguaje de expresiones tanto para el enfoque externalista como la versión con el tipado internalizado coinciden:

```
semequivExpr : ∀ {t} {e : Expr} {p : ⊢ e : t} → E⟦ e ⟧ p ≡ Et⟦ e ↑t p ⟧
```

Podemos repetir el mismo trabajo para el lenguaje de bajo nivel: definimos una versión ornamentada con el sistema de tipado y damos la relación semántica sobre este nuevo datatype, que será equivalente a la definida en la versión externalista.

El sistema de tipado para el lenguaje target relaciona cada instrucción con dos stack-type, correspondientes a los tipos de los elementos que están en la pila antes y después de la ejecución del programa:

```
data Codet : StackType → StackType → Set where
  push : ∀ {st} → ℕ → Codet st (nat : st)
```

```
add : ∀ {st} → Code_t (nat : nat : st) (nat : st)
eq  : ∀ {st} → Code_t (nat : nat : st) (bool : st)
_,_ : ∀ {st st' st''} → Code_t st st' → Code_t st' st'' → Code_t st st''
```

Al igual que en el caso de las expresiones, definir la versión ornamentada de esta manera es posible gracias a que la relación de tipado es dirigida por sintaxis. La función de lifting entre ambas versiones del tipo Code queda definida así:

```
_↑c_t_ : ∀ {st st'} → (c : Code) → st ⊢ c ⇝ st' → Code_t st st'
(push n) ↑c_t tpush        = push n
add      ↑c_t tadd         = add
eq       ↑c_t teq          = eq
(c1 , c2) ↑c_t (tseq p1 p2) = (c1 ↑c_t p1) , (c2 ↑c_t p2)
```

La relación semántica ahora puede definirse directamente sobre el datatype internalizado:

```
data ⟨_⟩t_⇒_ : ∀ {st st'} → Code_t st st' ->
                 Stack st -> Stack st' -> Set where
  pushRT : ∀ {st n} {s : Stack st} →
             ⟨ push n ⟩t s ⇒ (n ▷ s)
  addRT  : ∀ {m n st} {s : Stack st} →
             ⟨ add ⟩t (m ▷ (n ▷ s)) ⇒ ((m + n) ▷ s)
  eqRT   : ∀ {m n st} {s : Stack st} →
             ⟨ eq ⟩t (m ▷ (n ▷ s)) ⇒ ((m == n) ▷ s)
  seqRT  : ∀ {st st' st'' c1 c2}
             {s : Stack st} {s' : Stack st'} {s'' : Stack st''} →
             ⟨ c1 ⟩t s ⇒ s' → ⟨ c2 ⟩t s' ⇒ s'' → ⟨ (c1 , c2) ⟩t s ⇒ s''
```

la cual está relacionada con la versión externalista mediante esta propiedad:

```
semequivCode : ∀ {c st st' tp} {s : Stack st} {s' : Stack st'} →
                 ⟨ c , tp ⟩ s ⇒ s' → ⟨ c ↑c_t tp ⟩t s ⇒ s'
```

Una vez que definimos las versiones internalizadas con el sistema de tipado de cada lenguaje, podemos dar el compilador en el cual la propiedad de preservación de tipado es asegurada *por construcción*:

```
compile_t : ∀ {t} {st} -> Expr_t t → Code_t st (t : st)
compile_t | n |       = push n
compile_t (e1 ⊕ e2) = (compile_t e1) , (compile_t e2) , add
compile_t (e1 ≟ e2) = (compile_t e1) , (compile_t e2) , eq
```

El tipo de la función $\mathsf{compile}_t$ expresa la propiedad de preservación de tipado: Si se compila una expresión con tipo t se obtiene un

programa que ejecutándose en una pila con stack-type *st*, termina produciendo un satck con tipo $(t : st)$, es decir que sólo agrega un valor de tipo t al tope.

Internalizando las relaciones de tipado en los lenguajes pudimos obtener por construcción la propiedad deseada, sin embargo si queremos obtener un compilador correcto por construcción ésto no es suficiente: el valor de tipo t que queda en el tope podría ser erróneo. Podríamos por ejemplo definir la compilación de la suma de esta manera:

```
compile_t (e1 ⊕ e2) = compile_t e1 , compile_t e2 , add , push 1 , add
```

y la propiedad de preservación de tipado sigue cumpliéndose.

Tenemos nuevamente el problema de que el predicado de corrección es externo a la función de compilación:

```
correct_t : ∀ {t} {st} {e : Expr_t t} {s : Stack st} →
                    ⟨ (compile_t e) ⟩_t s ⇒ ((E_t⟦ e ⟧ ▷ s))
```

Avancemos entonces un paso más en nuestra metodología internalizando la semántica de manera de poder capturar la propiedad de corrección en el tipado de la función de compilación en Agda.

Internalización de la semántica

De la misma manera que hicimos con la internalización de tipado, procedemos de manera sistemática para incorporar la semántica de cada lenguaje en la definición del correspondiente datatype.

Para el caso de las expresiones teníamos definida una función semántica, cuyo dominio es la interpretación del tipo `nat` o `bool`, de acuerdo a si la expresión es una constante natural o una suma, o si es una igualdad de dos expresiones naturales, respectivamente. Como esta función es dirigida por sintaxis, tenemos una correspondencia uno a uno entre los constructores de las expresiones y su semántica; podríamos verlo también como que la semántica forma un álgebra (enfoque que profundizaremos en el capítulo siguiente). Definimos entonces un datatype indexado en la interpretación del tipo de la expresión:

```
data Expr_s : ∀ {t} → (ST t) → Set where
  |_|   : (n : ℕ) → Expr_s (fnat n)
  _⊕_ : ∀ {n1 n2} → (e1 : Expr_s n1) → (e2 : Expr_s n2) →
                                        Expr_s (fplus n1 n2)
  _≟_ : ∀ {n1 n2} → (e1 : Expr_s n1) → (e2 : Expr_s n2) →
                                        Expr_s (feq n1 n2)
```

y nuevamente una función de lifting que asocia la versión ornamentada solo con el tipado, al datatype ornamentado con la semántica:

```
_↑ₛ : ∀ {t} → (e : Exprₜ t) → Exprₛ (Eₜ⟦ e ⟧)
| x |       ↑ₛ = | x |
(e₁ ⊕ e₂) ↑ₛ = (e₁ ↑ₛ) ⊕ (e₂ ↑ₛ)
(e₁ ≟ e₂) ↑ₛ = (e₁ ↑ₛ) ≟ (e₂ ↑ₛ)
```

Continuamos ahora con la internalización de la semántica para el lenguaje de bajo nivel. Indexamos el datatype con un par correspondiente a la pila inicial y final de la ejecución de la instrucción:

```
data Codeₛ : ∀ {st st'} → Stack st → Stack st' → Set where
  push : ∀ {st} {s : Stack st} → (n : ℕ) → Codeₛ s (n ▷ s)
  add  : ∀ {st m n} {s : Stack st} → Codeₛ (m ▷ (n ▷ s)) ((m + n) ▷ s)
  eq   : ∀ {st m n} {s : Stack st} → Codeₛ (m ▷ (n ▷ s)) ((m == n) ▷ s)
  _,_  : ∀ {st st' st''} {s : Stack st} {s' : Stack st'} {s'' : Stack st''} →
         Codeₛ s s' → Codeₛ s' s'' → Codeₛ s s''
```

y definimos la función de lifting:

```
_↑cₛ_ : ∀ {st st'} {s : Stack st} {s' : Stack st'} →
        (c : Codeₜ st st') → ⟨ c ⟩ₜ s ⇒ s' → Codeₛ s s'
(push n) ↑cₛ pushRT = push n
add      ↑cₛ addRT  = add
eq       ↑cₛ eqRT   = eq
(c₁ , c₂) ↑cₛ (seqRT p₁ p₂) = (c₁ ↑cₛ p₁ , c₂ ↑cₛ p₂)
```

Con este segundo paso de internalización hemos obtenido datatypes que reflejan la semántica de cada constructor a nivel de tipos de Agda. Podemos entonces ahora definir un compilador del lenguaje de expresiones al lenguaje de bajo nivel, cuyo tipo asegura la corrección:

```
compileₛ : ∀ {t} {st} {v : (ST t)} {s : Stack st} →
           (e : Exprₛ v) → Codeₛ s (v ▷ s)
compileₛ | n |       = push n
compileₛ (e₁ ⊕ e₂) = compileₛ e₂ , compileₛ e₁ , add
compileₛ (e₁ ≟ e₂) = compileₛ e₂ , compileₛ e₁ , eq
```

Siguiendo esta metodología hemos obtenido un compilador de expresiones donde su corrección está asegurada estáticamente: si la definición pasa el type-checker de Agda entonces es correcta.

Definir la semántica a nivel del tipo del compilador constituye un enfoque metodológico en el cual la corrección es tenida en cuenta en el momento de la construcción del compilador. De esta manera pueden detectarse errores en una etapa temprana y de manera estática.

En la figura 4 podemos ver la relación entre las distintas versiones de los lenguajes: En la línea inferior encontramos el enfoque externalista para el cual necesitamos realizar las pruebas de preservación de tipado (typepres) y de corrección (correct). En la línea superior tenemos el enfoque internalista donde hemos especificado a nivel de tipos las propiedades deseadas pasando por un nivel intermedio donde sólo internalizamos el tipado.

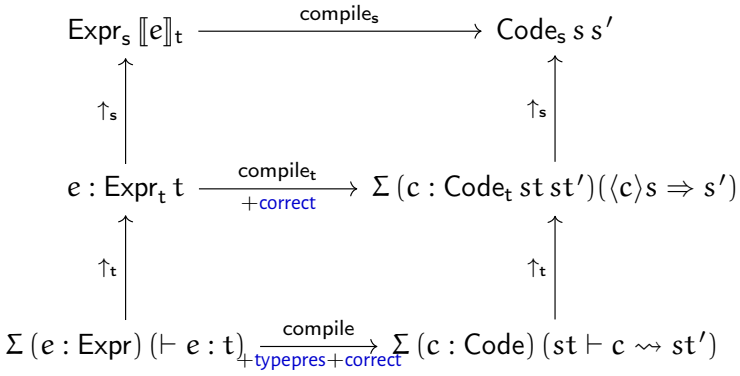


Figura 4: El enfoque internalista

3.2 EL ENFOQUE APLICADO A UN LENGUAJE IMPERATIVO SIMPLE

En esta sección aplicaremos la metodología para construir un compilador correcto de un lenguaje imperativo simple a una máquina basada en stack. Tendremos que lidiar con problemas como la no terminación y con algunas limitaciones que aparecen cuando la semántica de los lenguajes es más compleja.

Lenguaje de alto nivel

El lenguaje de expresiones aritméticas es idéntico al de la sección anterior, al cual le agregamos variables que pueden implementarse con cualquier tipo que tenga una noción de igualdad decidible:

```
data Expr : Set where
  |_|   : ℕ → Expr
  _⊕_   : (e₁ : Expr) → (e₂ : Expr) → Expr
  _≟̊_   : (e₁ : Expr) → (e₂ : Expr) → Expr
  var   : Var → Expr
```

Para las variables podríamos tomar Var = ℕ, o Var = String.

Las sentencias del lenguaje son la asignación, la secuencia y un constructor while para ciclos. Puesto que el condicional puede definirse en términos del while, preferimos evitar su inclusión.

```
data Stmt : Set where
  _:=_  : Var → Expr → Stmt
  while : Expr → Stmt → Stmt
  _;_   : Stmt → Stmt → Stmt
```

Tipado en el lenguaje de alto nivel

Extendemos el sistema de tipos presentado en la sección anterior a las nuevas construcciones del lenguaje. Por cuestiones de simplicidad las variables serán solo de tipo `nat`.

```
data ⊢_:_ : Expr → Type → Set where
  tnat  : ∀ {n} → ⊢ | n | : nat
  tplus : ∀ {e₁ e₂} → ⊢ e₁ : nat → ⊢ e₂ : nat →
                      ⊢ e₁ ⊕ e₂ : nat
  teq   : ∀ {e₁ e₂} → ⊢ e₁ : nat → ⊢ e₂ : nat →
                      ⊢ e₁ ≟ e₂ : bool
  tvar  : ∀ {x} → ⊢ var x : nat
```

Para las sentencias, necesitamos que la expresión asignada a variables sea de tipo `nat` y que la condición del while sea de tipo `bool`:

```
data ⊢_ : Stmt → Set where
  tassgn : ∀ {x} {e} → ⊢ e : nat → ⊢ (x := e)
  twhile : ∀ {e} {stmt} → ⊢ e : bool → ⊢ stmt →
                          ⊢ (while e do stmt)
  tseq   : ∀ {stmt₁ stmt₂} → ⊢ stmt₁ → ⊢ stmt₂ →
                             ⊢ (stmt₁ , stmt₂)
```

Como en la sección anterior, procedemos con el primer paso en la metodología: internalizar la relación de tipado en la definición del lenguaje de expresiones y sentencias.

```
data Exprₜ : Type → Set where
  |_| : ℕ → Exprₜ nat
  _⊕_ : (e₁ : Exprₜ nat) → (e₂ : Exprₜ nat) → Exprₜ nat
  _≟_ : (e₁ : Exprₜ nat) → (e₂ : Exprₜ nat) → Exprₜ bool
  var : (x : Var) → Exprₜ nat


data Stmtₜ : Set where
  _:=_  : Var → Exprₜ nat → Stmtₜ
  while : Exprₜ bool → Stmtₜ → Stmtₜ
  _,_   : Stmtₜ → Stmtₜ → Stmtₜ
```

Dado que no tenemos distintos tipos para las sentencias, no es necesario indizar el tipo `Stmtₜ` (en contraste con `Exprₜ`).

Una vez que definimos el datatype ornamentado, el siguiente paso en la metodología es definir la función de lifting que relaciona la versión externalista con la recién definida:

```
_↑t_ : ∀ {t} → (e : Expr) → ⊢ e : t → Exprt t
| x |       ↑t tnat         = | x |
(e1 ⊕ e2) ↑t tplus p1 p2 = (e1 ↑t p1) ⊕ (e2 ↑t p2)
(e1 ≟ e2) ↑t teq p1 p2   = (e1 ↑t p1) ≟ (e2 ↑t p2)
var x       ↑t tvar         = var x

_↑stmtt_ : (stmt : Stmt) → ⊢ stmt → Stmtt
_↑stmtt_ (x := e) (tassgn enat) = x := (e ↑t enat)
_↑stmtt_ (while e stmt) (twhile ebool p)
                        = while (e ↑t ebool) (stmt ↑stmtt p)
_↑stmtt_ (stmt1 , stmt2) (tseq p1 p2)
                        = (stmt1 ↑stmtt p1) , (stmt2 ↑stmtt p2)
```

Semántica del lenguaje de alto nivel

En el lenguaje tenemos variables, por lo tanto la semántica deberá incluir la noción de estado. Lo implementamos como una función que va del conjunto de variables a los naturales. Y definimos un operador de modificación de estado, que será útil para dar semántica a la asignación:

```
State : Set
State = Var → ℕ

_[_⟶_] : State → Var → ℕ → State
σ [ x ⟶ n ] = λ y → if y ==s x then n else σ y
```

Para dar semántica a las expresiones, necesitamos definir un dominio semántico. En el ejemplo de la sección anterior éste era el conjunto de los naturales o los booleanos de acuerdo al tipo de la expresión, pero ahora tenemos que considerar el estado de las variables. Por lo tanto la semántica de una expresión será una función de estados en la semántica del tipo correspondiente:

```
DomEs : Type → Set
DomEs t = (σ : State) → (ST t)
```

y la semántica de cada expresión estará definida por una función:

```
fcnat : ℕ → DomEs nat
fcnat n = const n
```

```
fplus : DomE_s nat → DomE_s nat → DomE_s nat
fplus f1 f2 = λ σ → f1 σ + f2 σ

f== : DomE_s nat → DomE_s nat → DomE_s bool
f== f1 f2 = λ σ → f1 σ == f2 σ

fvar : Var → DomE_s nat
fvar x = λ σ → σ x

⟦_⟧ : ∀ {t} → Expr_t t → DomE_s t
⟦ | n | ⟧       = fcnat n
⟦ e1 ⊕ e2 ⟧ = fplus ⟦ e1 ⟧ ⟦ e2 ⟧
⟦ e1 ≟ e2 ⟧ = f== ⟦ e1 ⟧ ⟦ e2 ⟧
⟦ var x ⟧      = fvar x
```

Podemos entonces definir el datatype de las expresiones ornamentado con la semántica. Notemos que el índice ahora será una función de estados en naturales o booleanos:

```
data Expr_s : ∀ {t} → DomE_s t → Set where
  |_|   : (n : ℕ) → Expr_s (fcnat n)
  _⊕_ : ∀ {f1 f2} → (e1 : Expr_s f1) → (e2 : Expr_s f2) →
                    Expr_s (fplus f1 f2)
  _≟_ : ∀ {f1 f2} → (e1 : Expr_s f1) → (e2 : Expr_s f2) →
                    Expr_s (f== f1 f2)
  var   : (x : Var) → Expr_s (fvar x)
```

La semántica de las sentencias usualmente se da como una relación entre estados: la ejecución de una sentencia transformará un estado modificando posiblemente sus variables. En la sección anterior dimos una semántica operacional big-step para el lenguaje de bajo nivel y luego definimos el datatype ornamentado que capturaba esta relación.

Intentemos hacer lo mismo para el lenguaje de sentencias. Como dijimos previamente todo nuestro enfoque requiere que la propiedad que se quiere internalizar sea dirigida por sintaxis. Como es habitual, la semántica operacional del constructor while distingue dos casos de acuerdo a la validez de la guarda. Para poder definirla y no perder la dirección por sintaxis, tendremos una relación auxiliar que refleja la semántica del while:

```
data ⟨_,_⟩⇒_ : (s : Stmt_t) → State → State → Set where
  assgn : ∀ {x e σ} → ⟨ x := e , σ ⟩⇒ (σ [ x ⟶ ⟦ e ⟧ σ ])
  seq   : ∀ {s₁ s₂ σ σ′ σ″} → ⟨ s₁ , σ ⟩⇒ σ′ →
                              ⟨ s₂ , σ′ ⟩⇒ σ″ →
                              ⟨ (s₁ , s₂) , σ ⟩⇒ σ″
  while : ∀ {e s σ σ′} → ⟨( ⟦ e ⟧ σ , e )⟩⇒w σ′ →
                         ⟨ while e s , σ ⟩⇒ σ′

data ⟨( _,_ )_,_⟩⇒w_ : Bool → Expr_t bool → Stmt_t →
                       State → State → Set where
  wFSem : ∀ {e s σ} → ⟨( false , e )⟩⇒w σ
  wTSem : ∀ {s σ σ′ σ″ e } → ⟨ s , σ ⟩⇒ σ′ →
                             ⟨( ⟦ e ⟧ σ′ , e )⟩⇒w σ″ →
                             ⟨( true , e )⟩⇒w σ″
```

Ahora intentemos definir el datatype ornamentado con dos índices correspondientes a los estados previo y posterior a la ejecución. ¿Cómo deberíamos indexar el cuerpo del while? Cada vez que se ejecute probablemente partirá y terminará en estados distintos por lo cual no podríamos fijarlos.

```
data Stmt_s : State -> State -> Set where
  _:=_    : ∀ {σ} {f : DomE_s nat} → (x : Var) → (e : Expr_s f) →
            Stmt_s σ (σ [ x ⟶ (f σ) ])
  _,_     : ∀ {σ σ′ σ″} →
            Stmt_s σ σ′ → Stmt_s σ′ σ″ → Stmt_s σ σ″
  while_s : ∀ {σ σ′} {f : DomE_s bool} →
            (e : Expr_s f) → ?? → Stmt_s σ σ′
```

Deberíamos completar el hueco marcado con ?? con el tipo que debería tener el cuerpo del ciclo, de manera tal que refleje la semántica operacional que dimos previamente. Una alternativa podría ser utilizar la semántica operacional directamente en la sintaxis del tipo ornamentado, pero esto nos desviaría de nuestra metodología, en la cual queremos tener los mismos constructores para las diferentes versiones refinadas de cada tipo de dato.

Todo indica entonces que ornamentar el tipo de las sentencias según su semántica operacional no es viable. Sin embargo podríamos definir una semántica funcional, como en [65] y ornamentar las sentencias utilizando como índice esta función. El único inconveniente a sortear es que la semántica es parcial ya que tenemos programas que no terminan.

Podemos definir una aproximación al menor punto fijo mediante un número natural (un *clock*), que lleve la cuenta de cuántos ciclos se

han realizado.[2] Si el índice es cero, entonces la semántica será indefinida. Sólo estará definida si se alcanza el menor punto fijo con el clock aún no nulo.

Para representar la posible indefinición utilizamos el tipo *Maybe*. Podemos definir el dominio semántico como una función que va de un clock y un estado inicial a un (posible) estado final:

```
DomS_s : Set
DomS_s = (clock : ℕ) → (σ : State) → Maybe State
```

Definamos entonces una semántica denotacional para las sentencias bien tipadas, de acuerdo a la siguiente álgebra:

```
fassgn : Var → DomE_s nat → DomS_s
fassgn x fe = λ clock σ → just (σ [ x ⟶ fe σ ])
```

La asignación actualiza el estado modificando el valor de la variable x con la semántica de la expresión e, y siempre está definida, por lo tanto utilizamos el constructor just.

```
fseq : DomS_s → DomS_s → DomS_s
fseq f1 f2 = λ clock σ → f1 clock σ >>= f2 clock
```

La semántica de la secuencia consiste en una secuencia monádica en el tipo Maybe. Si la aplicación de f_1 sobre el clock y el estado está definida entonces el estado resultante es pasado como argumento de f_2 con el mismo clock. Si alguna de las dos aplicaciones es indefinida, entonces la semántica de la secuencia lo será.

```
fwhile : DomE_s bool → DomS_s → DomS_s
fwhile fb fc zero          σ = nothing
fwhile fb fc (suc clock) σ = if fb σ then fc (suc clock) σ >>=
                                          fwhile fb fc clock
                             else just σ

fseq : DomS_s → DomS_s → DomS_s
fseq f1 f2 = λ clock σ → f1 clock σ >>= f2 clock
```

La definición de la semántica del while es recursiva, y consiste en ejecutar el cuerpo mientras la condición sea verdadera y el clock mayor que cero. Si este último se hace nulo, entonces el resultado será indefinido. Si la guarda es falsa, se devuelve el estado sin modificar.

Finalmente podemos definir la semántica denotacional para las sentencias bien tipadas de este modo:

2 Alternativamente podríamos usar la *Delay Monad* para dar semántica del while utilizando el operador de punto fijo como lo hace Benton et al.[7].

```
⟦_⟧S : Stmt_t → DomS_s
⟦ x := e ⟧S      = fassgn x ⟦ e ⟧
⟦ while b c ⟧S = fwhile ⟦ b ⟧ ⟦ c ⟧S
⟦ s₁ , s₂ ⟧S     = fseq ⟦ s₁ ⟧S ⟦ s₂ ⟧S
```

El siguiente paso es internalizar esta semántica definiendo un datatype indexado por el dominio semántico DomS_s.

```
data Stmt_s : DomS_s → Set where
  _:=_  : ∀ {f} → (x : Var) → Expr_s f → Stmt_s (fassgn x f)
  _,_   : ∀ {f₁ f₂} → Stmt_s f₁ → Stmt_s f₂ → Stmt_s (fseq f₁ f₂)
  while : ∀ {fb} {f} → Expr_s fb → Stmt_s f → Stmt_s (fwhile fb f)
```

Las funciones de lifting no aportan ninguna novedad, por ello mostramos únicamente sus signaturas:

```
_↑_s : ∀ {t} → (e : Expr_t t) → Expr_s ⟦ e ⟧
```

```
_↑stmt_s : (stmt : Stmt_t) → Stmt_s ⟦ stmt ⟧S
```

La función de lifting desde el datatype original hasta la versión ornamentada con la semántica es la composición de las funciones ya definidas:

```
_↑t s_ : ∀ {t} → (e : Expr) → (p : ⊢ e : t) → Expr_s (⟦ e ↑_t p ⟧)
e ↑t s p = (e ↑_t p) ↑_s
```

```
_↑stmt_t s_ : (stmt : Stmt) → (p : ⊢ stmt) → Stmt_s ⟦ stmt ↑stmt_t p ⟧S
stmt ↑stmt_t s p = (stmt ↑stmt_t p) ↑stmt_s
```

Lenguaje de bajo nivel

La máquina consiste de una pila, una memoria y la instrucción que está siendo ejecutada. El lenguaje para esta máquina es similar al de la sección anterior, al que le debemos agregar instrucciones que permitan manipular el estado y un operador para poder realizar ciclos.

$$\begin{array}{lll} c ::= & \mathtt{push}\ v \mid \mathtt{add} \mid \mathtt{equ} \mid c_1, c_2 \mid \\ & \mathtt{load}\ x \mid \mathtt{store}\ x \mid \mathtt{loop}[c_1, c_2] \end{array}$$

Este lenguaje también manipula variables mediante las instrucciones `load` y `store`, que obtienen el valor de una variable desde la memoria y lo ubican en el tope de la pila, o actualizan la memoria en el lugar de la variable con el valor que está en el tope de la pila, respectivamente. También consta de una instrucción $\mathtt{loop}[c_1, c_2]$ cuya semántica consiste en ejecutar la instrucción c_1 la cual debe dejar un valor booleano en el tope de la pila, si éste es verdadero se ejecuta

c_2 y vuelve a repetirse la instrucción, hasta que la ejecución de c_1 obtenga un valor falso.

```
data LCode : Set where
  push : (n : ℕ) → LCode
  add  : LCode
  eq   : LCode
  load : (x : Var) → LCode
  store : (x : Var) → LCode
  loop : (c₁ : LCode) → (c₂ : LCode) → LCode
  _,_  : (c₁ : LCode) → (c₂ : LCode) → LCode
```

Es interesante notar que en el lenguaje de bajo nivel no tenemos distinción entre lo que corresponde a expresiones o a sentencias. El compilador traducirá ambas categorías sintácicas de alto nivel al mismo lenguaje de bajo nivel.

Stack-safety del bajo nivel

La primera propiedad que queremos asegurar en el código compilado es que sea *stack-safe*. La máquina que ejecuta el código de bajo nivel tiene instrucciones que exigen requisitos sobre la pila. Por ejemplo, la instrucción `add` requiere que haya dos naturales en el tope para poder ejecutarse. De manera similar, otras instrucciones tienen requisitos y queremos que el código compilado siempre pueda ejecutarse, por lo tanto debe ser stack-safe.

Podemos definir la relación que caracteriza esta propiedad con el siguiente tipo de datos en Agda:

```
data _⊢_⇝_ : StackType → LCode → StackType → Set
  where
    rpush : ∀ {st} {n : ℕ} → st ⊢ push n ⇝ (nat : st)
    radd  : ∀ {st} → (nat : nat : st) ⊢ add ⇝ (nat : st)
    req   : ∀ {st} → (nat : nat : st) ⊢ eq ⇝ (bool : st)
    rload : ∀ {st} {x : Var} → st ⊢ load x ⇝ (nat : st)
    rstore : ∀ {st} {x : Var} → (nat : st) ⊢ store x ⇝ st
    rloop : ∀ {st} {c₁ c₂} → st ⊢ c₁ ⇝ (bool : st) → st ⊢ c₂ ⇝ st →
                                st ⊢ loop c₁ c₂ ⇝ st
    rseq  : ∀ {st st' st''} {c₁ c₂} → st ⊢ c₁ ⇝ st' → st' ⊢ c₂ ⇝ st'' →
                                      st ⊢ c₁ , c₂ ⇝ st''
```

por ejemplo la regla `req` establece que la instrucción `eq` puede ejecutarse si en la pila hay dos naturales en el tope, reemplazándolos por un booleano.

Procedemos entonces a ornamentar la sintaxis del lenguaje de bajo nivel con este sistema de tipos. Definimos un datatype con dos índices de tipo `StackType`.

```
data LCode_t : StackType → StackType → Set where
  push : ∀ {st} → (n : ℕ) → LCode_t st (nat : st)
  add  : ∀ {st} → LCode_t (nat : nat : st) (nat : st)
  eq   : ∀ {st} → LCode_t (nat : nat : st) (bool : st)
  load : ∀ {st} → (x : Var) → LCode_t st (nat : st)
  store : ∀ {st} → (x : Var) → LCode_t (nat : st) st
  loop : ∀ {st} → (c1 : LCode_t st (bool : st)) → (c2 : LCode_t st st) →
                  LCode_t st st
  _;_  : ∀ {st st' st''} → (c1 : LCode_t st st') → (c2 : LCode_t st' st'') →
                           LCode_t st st''
```

y la función de lifting que relaciona el tipo ornamentado con su versión externalista. Omitimos su definición ya que no presenta ninguna novedad.

```
_↑c_ : ∀ {st st'} → (c : LCode) → st ⊢ c ⇝ st' → LCode_t st st'
```

Semántica del código de bajo nivel

La configuración de la máquina abstracta en la sección anterior consistía sólo de la pila. Como ahora tenemos que lidiar con variables, debemos agregar a esta configuración un estado de valores para las mismas.

```
Conf : (st : StackType) → Set
Conf st = Stack st × State
```

Para dar semántica al lenguaje de instrucciones de bajo nivel, al tener la primitiva loop, no podemos dar una relación que fije la configuración antes y después de la ejecución que sea dirigida por sintaxis, característica fundamental para nuestro enfoque. Procederemos de la misma manera que con el lenguaje de alto nivel: Definimos una semántica denotacional y para lidiar con la no terminación, utilizaremos un número natural que contará la cantidad de unfoldings para aproximarse al punto fijo, en caso de que exista. El domininio semántico serán funciones que toman un natural (el reloj), una configuración de la máquina, y obtienen otra configuración en caso de que se alcance la terminación del programa. Representamos la parcialidad de la función con el tipo *Maybe*.

```
DomC_s : (st : StackType) → (st' : StackType) → Set
DomC_s st st' = (clock : ℕ) → (sσ : Conf st) → Maybe (Conf st')
```

La semántica constituye un álgebra para la signatura de las instrucciones de bajo nivel. En las instrucciones push, add, y eq la definición será similar a la de la sección anterior, el reloj será ignorado (ya que estos programas terminan en un solo paso) y el resultado es siempre definido, por eso usamos el constructor just.

```
fpush : ∀ {st} → (n : ℕ) → DomC_s st (nat : st)
fpush n = λ {clock (s , σ) → just ((n ▷ s) , σ)}

fadd : ∀ {st} → DomC_s (nat : nat : st) (nat : st)
fadd = λ {clock ((n ▷ (m ▷ s)) , σ) → just (((n + m) ▷ s) , σ)}

feq : ∀ {st} → DomC_s (nat : nat : st) (bool : st)
feq = λ {clock ((n ▷ (m ▷ s)) , σ) → just (((n == m) ▷ s) , σ)}
```

Las instrucciones load y store ponen el valor de una variable en el tope de la pila y actualizan el valor de una variable de acuerdo al valor que esté en el tope de la pila respectivamente.

```
fload : ∀ {st} → (x : Var) → DomC_s st (nat : st)
fload x = λ {clock (s , σ) → just ((σ x ▷ s) , σ)}

fstore : ∀ {st} → (x : Var) → DomC_s (nat : st) st
fstore x = λ {clock ((n ▷ s) , σ) → just (s , σ [ x ⟶ n ])}
```

La semántica de la secuencia es similar a la del alto nivel, donde en vez de la composición de funciones, utilizamos el operador de *bind* de la mónada Maybe.

```
fseqc : ∀ {st st' st''} → DomC_s st st' → DomC_s st' st'' →
                          DomC_s st st''
fseqc f1 f2 = λ clock sσ → f1 clock sσ >>= f2 clock
```

La semántica de la instrucción loop es definida recursivamente. Cuando el reloj es mayor que cero, el código de la condición es ejecutado. Si éste deja el valor true en el tope de la pila, luego el cuerpo es ejecutado y se aplica recursivamente la función con el reloj decrementado en uno. Si el reloj llega a cero, entonces el resultado es indefinido, y cuando la condición deja false en el tope, la ejecución termina.

```
floop : ∀ {st} → DomC_s st (bool : st) → DomC_s st st → DomC_s st st
floop fb fc zero        sσ = nothing
floop fb fc (suc clock) sσ = fb (suc clock) sσ >>=
                             (λ { ((b ▷ s') , σ') →
                               if b
                               then fc (suc clock) (s' , σ') >>=
                                    floop fb fc clock
                               else just (s' , σ') })
```

Teniendo definida el álgebra de la semántica del código de bajo nivel podemos definir el datatype ornamentado. Además de los constructores correspondientes a cada construcción del lenguaje, vamos a agregar uno extra para lidiar con una complicación que en el pequeño ejemplo de la sección anterior no apareció. En nuestro enfoque la corrección del compilador es chequeada estáticamente mediante el

type-checker de Agda. Éste comprobará que el tipo del término que definimos para la compilación es exactamente el que está declarado, pero hay veces que dos tipos no son sintácticamente iguales (judgmental equality) pero que puede probarse que son *proposicionalmente* iguales (propositional equality). Para esos casos existe la función subst de Agda, que permite cambiar el tipo de un término por otro proposicionalmente igual.

En nuestro ejemplo tenemos que ornamentar el datatype con una función en $\mathsf{DomC_s}$, y como veremos un poco más adelante en la definición del compilador, podemos tener un término con su tipo indizado por una función que es *extensionalmente* igual al esperado. En este caso no podemos usar la función subst de Agda ya que la igualdad extensional no es la proposicional. Para reparar esta situación definiremos un constructor más en el datatype ornamentado que nos permite reemplazar un índice por otro extensionalmente igual.

```
EqSem : ∀ {st st'} → DomC_s st st' → DomC_s st st' → Set _
EqSem f g = ∀ clock sσ → f clock sσ ≡ g clock sσ


data LCode_s : ∀ {st st'} → DomC_s st st' → Set where
  push : ∀ {st} → (n : ℕ) → LCode_s (fpush {st} n)
  add  : ∀ {st} → LCode_s (fadd {st})
  eq   : ∀ {st} → LCode_s (feq {st})
  load : ∀ {st} → (x : Var) → LCode_s (fload {st} x)
  store : ∀ {st} → (x : Var) → LCode_s (fstore {st} x)
  loop : ∀ {st} {f1 f2} → (c1 : LCode_s f1) → (c2 : LCode_s f2) →
                          LCode_s (floop {st} f1 f2)
  _;_  : ∀ {st st' st''}  {f1 f2} → (c1 : LCode_s f1) →
         (c2 : LCode_s f2) → LCode_s (fseqc {st} {st'} {st''} f1 f2)

  substC_s : ∀ {st st'} {f g} → LCode_s {st} {st'} f → EqSem f g → LCode_s g
```

el constructor $\mathsf{substC_s}$ permite convertir una instrucción tipada con la función f en una tipada con la función g, si f y g son extensionalmente iguales.

El compilador correcto

Teniendo los tipos de datos ornamentados con la semántica podemos definir el compilador correcto por construcción. Para el caso de las expresiones, tenemos como índice a una función de estados en valores. El compilador expresará en su tipo que si compilamos una expresión anotada con el índice f, el código resultante tendrá como índice la función que deja en el tope de la pila a f aplicado al estado.

```
compₑ : ∀ {t} {f} {st} → Exprₛ {t} f →
                          LCodeₛ {st} (λ {clock (s , σ) → just ((f σ ▷ s) , σ) })
compₑ | n |       = push n
compₑ (e₁ ⊕ e₂) = compₑ e₂ , (compₑ e₁ , add)
compₑ (e₁ ≙ e₂) = compₑ e₂ , (compₑ e₁ , eq)
compₑ (var x)     = load x
```

Para el caso de las sentencias la relación entre el índice del lenguaje de alto nivel y el de bajo nivel la damos con la siguiente definición:

```
compiledSem : ∀ {st} → DomSₛ → DomCₛ st st
compiledSem f clock (s , σ) = f clock σ >>= (λ σ' → just (s , σ'))
```

Si f es el índice de la sentencia, entonces la compilación deberá tener como índice a la función que consiste de aplicar f al clock y al estado, y si el resultado es definido obtiene la pila sin modificar junto al estado modificado por f.

Podemos entonces definir un compilador de sentencias de alto nivel a instrucciones del bajo nivel que exprese en su tipo la propiedad de corrección:

```
compₛ : ∀ {f} {st} → Stmtₛ f → LCodeₛ (compiledSem {st} f)
```

La compilación de la asignación $x := e$ es la secuencia de compilar la expresión e (que dejará en el tope de la pila su valor semántico) y guardar en la variable x el valor que queda en el tope de la pila

```
compₛ (x := e) = compₑ e , store x
```

y la corrección de esta definición es asegurada por el type-checker de Agda, que la acepta sin dar error.

Para el caso de la secuencia y el while tenemos en el bajo nivel instrucciones que corresponden a estas dos sentencias por lo que la definición aparenta ser obvia. Sin embargo nos encontramos en ambos casos con que el índice que infiere el type-checker de Agda para el término del bajo nivel que damos no es exactamente el mismo que está declarado en el tipo del compilador (no es definicionalmente igual). Aquí es donde usamos el constructor de substitución que explicamos previamente ya que ambos tipos (el inferido y el esperado) tienen índices extensionalmente iguales:

```
compₛ (_,_ {f₁} {f₂} stmt₁ stmt₂) = (compₛ stmt₁ ,c compₛ stmt₂) *
  where _* : LCodeₛ _ → _
        c * = substCₛ c (eqseq f₁ f₂)
compₛ (while {fb} {f} eb stmt) = (loop (compₑ eb) (compₛ stmt)) *
  where _* : LCodeₛ _ → _
        c * = substCₛ c (eqloop fb f)
```

eqseq y eqloop son las pruebas de igualdad extensional mencionadas que no conllevan ninguna dificultad. Mostramos solo su declaración:

```
eqseq : ∀ {st} f₁ f₂ → EqSem {st} {st}
                       (fseqc (compiledSem f₁) (compiledSem f₂))
                       (compiledSem (λ clock σ → f₁ clock σ >>= f₂ clock))

eqloop : ∀ {st} fb f → EqSem {st} {st}
                       (floop (λ { clock (s , σ) → just ((fb σ ▷ s) , σ) })
                              (compiledSem f))
                       (compiledSem (fwhile fb f))
```

Con esto completamos la definición del compilador correcto por construcción. El enfoque internalista permite expresar a nivel del tipado del lenguaje host (en nuestro caso Agda) la corrección del compilador, de manera que la única definición aceptada por el type-checker es la correcta. Si bien tenemos algunas limitaciones del sistema de tipos de Agda (al usar índices donde la noción de igualdad no es la proposicional), podemos sortearlas de manera elegante utilizando un constructor de substitución en el lenguaje target del compilador.

Los lenguajes de alto nivel y bajo nivel del compilador que acabamos de definir presentan diferencias (el último maneja una pila de ejecución), pero todavía estamos un poco lejos de un compilador a un lenguaje más cercano a la máquina (similar a un Assembly). En la siguiente sección presentamos un lenguaje de bajo nivel más realista y estudiamos si la metodología puede extenderse a estos casos.

3.3 HACIA UN COMPILADOR REALISTA

El lenguaje al que compilamos en la sección anterior dista bastante de los lenguajes de bajo nivel como Assembly o LLVM. En particular porque ninguno de estos últimos cuenta con primitivas para realizar ciclos, sino que los mismos son posibles mediante saltos condicionales en el código, ni tampoco primitiva de secuencia. En esta sección presentamos un lenguaje sencillo con estas características con el objetivo de realizar nuestra metodología en un contexto más realista.

El lenguaje de bajo nivel con saltos

El lenguaje consiste de una lista de instrucciones básicas entre las que se encuentra una instrucción para realizar saltos condicionales. Los saltos están indicados con etiquetas relativas a la posición en la que se encuentra la instrucción.

```
data Shift : Set where
  left  : ℕ → Shift
  right : ℕ → Shift
```

```
data Instr : Set where
  nop   : Instr
  add   : Instr
  eq    : Instr
  push  : ℕ → Instr
  jmp   : (l : Shift) → Instr
  jz    : (l : Shift) → Instr
  load  : (x : Var) → Instr
  store : (x : Var) → Instr

Assm : Set
Assm = List Instr
```

De manera similar al lenguaje de la sección anterior, tenemos instrucciones para poner un valor en la pila, para sumar, chequear igualdad, poner el valor de una variable en el tope de la pila y actualizar el valor de una variable. Además tenemos la instrucción para realizar saltos y una instrucción que no realiza nada. Si bien aún este lenguaje presenta una abstracción de la memoria en su manejo de variables, es bastante cercano a las arquitecturas reales. La dificultad principal para poder realizar la metodología internalista con este lenguaje es que no tenemos una manera sencilla de dar semántica dirigida por sintaxis. Lo que podemos hacer es combinar los abordajes internalista y externalista componiendo el compilador que dimos en la sección anterior con uno que tendrá una prueba de corrección para pasar de LCode_s a Assm.

Semántica del lenguaje con saltos

Para dar semántica del lenguaje definimos una máquina abstracta cuya configuración consiste de un contador de programa (el *program counter*, que será un número natural), un estado de valores para las variables y una pila de números naturales (a los booleanos los representamos con naturales).

```
Machine : Set
Machine = ℕ × State × List ℕ

pc : Machine → ℕ
pc (i , _ , _) = i
```

Para los lenguajes de máquina es habitual dar semántica small-step a partir de la transición que produce cada instrucción individualmente.

```
fjmp : Shift → Machine → Machine
fjmp (left i)  (k , σ , s) = k ∸ i , σ , s
fjmp (right i) (k , σ , s) = k + i , σ , s
```

```
data (_)_⇝ᵢ_ : Instr → Machine → Machine → Set where
  nop : ∀ {k σ s} → ( nop ) (k , σ , s) ⇝ᵢ (k +1 , σ , s)
  push : ∀ {k σ s n} → ( push n ) (k , σ , s) ⇝ᵢ (k +1 , σ , n : s)
  add  : ∀ {k σ s n₁ n₂} →
           ( add ) (k , σ , n₁ : n₂ : s) ⇝ᵢ (k +1 , σ , n₁ + n₂ : s)
  eq   : ∀ {k σ s n₁ n₂} →
           ( eq ) (k , σ , n₁ : n₂ : s) ⇝ᵢ (k +1 , σ , n₁ ==ₙ n₂ : s)
  jmp  : ∀ {cf sh} → ( jmp sh ) cf ⇝ᵢ (fjmp sh cf)
  jz0  : ∀ {k σ s sh} →
           ( jz sh ) (k , σ , 0 : s) ⇝ᵢ (fjmp sh (k , σ , s))
  jzs  : ∀ {k σ s sh n} →
           ( jz sh ) (k , σ , n + 1 : s) ⇝ᵢ (k +1 , σ , s)
  load : ∀ {k σ s x} →
           ( load x ) (k , σ , s) ⇝ᵢ (k +1 , σ , σ x : s)
  store : ∀ {k σ s x n} →
           ( store x ) (k , σ , n : s) ⇝ᵢ (k +1 , σ [ x ⟶ n ] , s)
```

Las instrucciones básicas actualizan el estado o la pila (si ésta tiene la forma adecuada) y aumentan en uno el contador. Las instrucciones de salto cambian el contador por el que se indique, y en el caso del condicional sólo si en el tope de la pila se encuentra un 0. Los saltos son relativos a la posición en que se encuentra la instrucción.

Ejecutar un paso de ejecución en una lista de instrucciones podrá realizarse si el program counter es una posición válida en ésta. Expresamos esta condición mediante una función de indexación en listas definida con el tipo Maybe:

```
_‼_ : {A : Set} → List A → ℕ → Maybe A
[] ‼ _ = nothing
(a : ls) ‼ zero = just a
(a : ls) ‼ (suc n) = ls ‼ n
```

Si al indexar la lista de instrucciones obtenemos la instrucción i, el resultado de un paso de ejecución será el que se obtiene al ejecutar i en la configuración de la máquina, en caso contrario la máquina no puede avanzar.

```
data (_)_⇝_ (c : Assm) : Machine → Machine → Set where
  step : ∀ {i cf cf'} → (c ‼ (pc cf) ≡ just i) →
                        ( i ) cf ⇝ᵢ cf' → ( c ) cf ⇝ cf'
```

Y podemos definir la clausura transitiva sobre esta relación:

```
data Tr {A : Set} (R : A → A → Set) : A → A → Set where
  ⇝s : ∀ {a b} → R a b → Tr R a b
  _⇝t_ : ∀ {a b c} → R a b → Tr R b c → Tr R a c


(_)_⇝*_ : (is : Assm) → Machine → Machine → Set
( is ) cf ⇝* cf' = Tr (( is )_⇝_) cf cf'
```

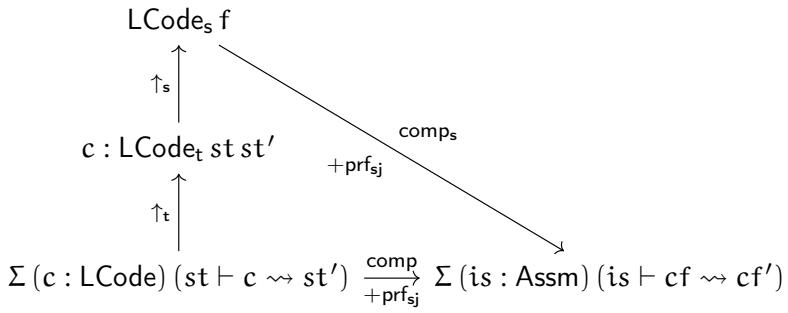


Figura 5: Compilador al lenguaje con saltos

Una propiedad interesante que podríamos querer representar y que el compilador debe cumplir, es que los saltos en el código compilado siempre deben ser válidos. Es un buen ejemplo de propiedad para internalizar en la sintaxis del lenguaje, pero en este trabajo no lo hemos realizado.

El compilador

A continuación definimos un compilador del lenguaje LCode_s cuya semántica se encuentra en el tipo, hacia el lenguaje Assm. La corrección de este compilador estará dada por una prueba de adecuación computacional entre la semántica denotacional del primero y la semántica small-step del segundo.

En la figura 5 vemos la última etapa de compilación en la cual combinamos el desarrollo internalista realizado para ir desde el lenguaje de sentencias de alto nivel al intermedio LCode_s, con la definición de un compilador junto con su prueba de corrección para obtener código de bajo nivel.

El compilador traduce términos del lenguaje intermedio a una lista de instrucciones. Recordemos que al utilizar la metodología externalista, podríamos cometer errores que serán recién detectados al momento de la verificación de corrección, situación que en la sección anterior no podría suceder dado que esta prueba se encuentra codificada en el tipado, y por tanto los errores son detectados estáticamente por el type-checker de Agda.

```
compL : ∀ {st st'} {f : DomC_s st st'} → LCode_s f → Assm
compL (push n)      = [ push n ]
compL add           = [ add ]
compL eq            = [ eq ]
compL (load x)      = [ load x ]
compL (store x)     = [ store x ]
compL (c₀ ,c c₁)    = (compL c₀) ++ (compL c₁)
```

```
compL (loop b c)      = loopAssm (compL b) (compL c)
compL (substC_s c _) = compL c
```

Los primeros cinco casos son triviales, pues cada sentencia del lenguaje intermedio se corresponde con una instrucción en el lenguaje de bajo nivel. La secuencia se compila a la concatenación de la compilación de cada subtérmino y el caso del constructor de substitución simplemente compila el término ignorando la prueba de igualdad extensional. Para el caso del loop definimos la función loopAssm que dadas dos listas de instrucciones, una para el código correspondiente a la guarda del ciclo y otra para el código correspondiente al cuerpo, construye el código de bajo nivel realizando los saltos necesarios.

```
loopAssm : Assm → Assm → Assm
loopAssm isB isC = isB ++ [ jz le ] ++ isC ++ [ jmp li ]
  where le = right (2 + length isC)
        li = left (1 + length isB + length isC)
```

El lector puede convencerse de que la definición anterior efectivamente corresponde a la compilación de un ciclo while, sin embargo la única manera de asegurar la corrección es dando una prueba.

Para dar la prueba de adecuación entre la semántica denotacional de LCode_s y la semántica small-step de Assm definimos cuándo un elmento de este último *realiza* una función semántica:

```
realizes : ∀ {st st'} → DomC_s st st' → Assm → Set
realizes f is = ∀ {cl s s' σ σ'} → f cl (s , σ) ≡ just (s' , σ') →
  ( is ) (0 , σ , unty s) ⇝* (length is , σ' , unty s')
```

El programa de bajo nivel *is realiza* la función f si existe una traza desde la primera hasta la última instrucción de *is* (inclusive), cambiando la configuración $(s,(\sigma))$ por $(s',(\sigma'))$ cada vez que la aplicación $f\ cl\ (s,(\sigma))$ esté definida y cuyo resultado sea $(s',(\sigma'))$.

Es sencillo enunciar la corrección del compilador utilizando la noción de realización recién definida:

```
prf_sj : ∀ {st st'} {f : DomC_s st st'} → (lc : LCode_s f) →
                  realizes f (compL lc)
```

Rápidamente se descubre que este enunciado es muy particular como para poder ser útil: la noción de realización de una función implica que el código compilado se ejecute *siempre* desde la primera instrucción y esto nos impediría dar la prueba para los subtérminos, en otras palabras, no podemos aplicar la hipótesis inductiva en el contexto necesario. Para resolverlo necesitamos generalizar la noción de realización teniendo en cuenta el contexto donde se ejecuta un código:

```
ctx-realizes : ∀ {st st'} → DomC_s st st' → Assm → Set
ctx-realizes f is = ∀ cl s {s'} σ {σ'} {γ γ'} →
                    f cl (s , σ) ≡ just (s' , σ') →
                    ( γ ++ is ++ γ' ) (length γ , σ , unty s) ⇝*
                                      (length (γ ++ is) , σ' , unty s')
```

Esta definición permite que la lista *is* sea parte de una lista de instrucciones más grande dándonos la generalidad necesaria para la prueba de corrección.

Podemos entonces dar la prueba inductiva de adecuación entre la semántica denotacional del lenguaje intermedio y la semántica small-step del lenguaje de bajo nivel. Mostramos solo su declaración ya que la misma es demasiado tediosa debido principalmente a la dificultad del lenguaje Agda para probar equivalencias de operadores asociativos y conmutativos como la concatenación de listas.

```
ctx-prf_sj : ∀ {st st'} {f : DomC_s st st'} → (lc : LCode_s f) →
                ctx-realizes f (compL lc)
```

Obviamente la versión original de la prueba es obtenida tomando como prefijo y sufijo del código compilado a la lista vacía.

Completando esta prueba podemos dar un compilador de punta a punta desde el lenguaje imperativo simple de la sección anterior hasta el lenguaje de instrucciones con saltos componiendo el compilador definido con el enfoque internalista, cuya prueba es asegurada estáticamente, con el compilador compL junto con su prueba de corrección.

3.4 RESUMEN DEL ENFOQUE Y TRABAJOS RELACIONADOS

Hemos presentado una metodología para construir compiladores correctos por construcción para lenguajes cuya semántica es dirigida por sintaxis. Partiendo de un caso sencillo como el lenguaje de expresiones aritméticas lo extendimos con construcciones más complejas, teniendo que lidiar con problemas como la no terminación. Si bien el enfoque no pudo ser aplicado a un compilador que traduzca a instrucciones de bajo nivel, mostramos que podemos componer el enfoque internalista compilando hacia un lenguaje intermedio y luego hacia el lenguaje target definiendo una prueba externalista aprovechando que una parte de la corrección del compilador punta a punta es asegurada estáticamente.

Trabajos relacionados. La teoría de Ornaments fue propuesta por McBride en [58] como una manera de trabajar con tipos de datos decorados con diferentes propiedades. Como mostró McBride en su artículo (siguiendo un ejemplo de McKinna [60]), se puede utilizar ornaments para definir un compilador correcto por construcción. Basados en esta teoría, Ko y Gibbons [48] caracterizaron el enfoque de

McBride como *internalista*, contrastando con el más usual *externalista*, que es más cercano a la verificación de programas. En este último caso, las propiedades son definidas como predicados separados de la definición, mientras que en el primer caso se utilizan familias indexadas embebiendo la propiedad en el tipo como un índice para cada constructor. El isomorfismo entre el tipo de dato externalista (junto con el predicado) y su correspondiente versión internalista es llamado *refinamiento*.

Ko y Gibbons propusieron una metodología para obtener automáticamente ambas versiones del tipo de datos, en lugar de definirlos independientemente. En lugar de definir los tipos de datos inductivamente, uno *describe* los tipos y sus ornamentaciones mediante un universo de descripciones, siguiendo la línea de Martin-Löf para definir universos à la Tarski [55]. A partir de estas descripciones uno obtiene el tipo de datos, el predicado expresando la propiedad (por ejemplo, listas ordenadas), y también el isomorfismo entre las versiones internalista y externalista. Dagand y McBride [21] mostraron que las funciones definidas sobre los tipos de datos sin ornamentos pueden ser lifteadas a las versiones ornamentadas.

Trabajar con descripciones de tipos de datos y ornamentos como es presentado en los trabajos que citamos es sumamente tedioso y complicado, como lo hemos comprobado realizando el ejercicio de implementar el compilador para el lenguaje de expresiones de la sección 3.1 bajo el framework de ornaments. Nuestra metodología aplica estas ideas mostrando los pasos a seguir para decorar los tipos de datos con las propiedades deseadas y definiendo las funciones de lifting para pasar de la versión externalista a la internalista de una manera más realizable en la práctica. Existen trabajos [97, 98] donde se busca que las ideas de ornaments sean soportadas desde el lenguaje host, en particular para el lenguaje ML.

4

FORMALIZACIÓN DE ÁLGEBRA UNIVERSAL

A lo largo de la historia se han estudiado distintas estructuras algebraicas en Matemática tales como Anillos o Grupos, en las cuales se tienen conjuntos con alguna estructura definida y se estudian propiedades. Estas teorías se fueron desarrollando de manera más o menos independiente sin tener un marco común de estudio para conceptos que son comunes a todas. Birkhoff en el año 35 [8] presenta un trabajo sobre "álgebras abstractas" estudiando de una manera general diferentes tipos de estructuras algebraicas y mostrando resultados comunes a todas ellas, como así también construcciones que se encuentran en cada una de las álgebras concretas (como productos, subálgebras, congruencias). A partir de ese momento se abrió un área de estudio importante en matemática y lógica que sigue estando vigente y reviste de importancia en estos días. También en Ciencias de la Computación el Álgebra Universal ha tenido un rol importante desde épocas tempranas, en particular el paper de Birkhoff [9] estudia lenguajes regulares como un buen ejemplo de aplicación; poco tiempo antes Burstall [14] mostró propiedades de programas usando inducción estructural, que está basada en concebir al lenguaje como un álgebra inicial. El grupo ADJ (Goguen, Thatcher, Wagner y Wright) impulsó el estudio de las álgebras multi-sort como una herramienta teórica clave para especificar tipos abstractos de datos [31], semántica [32] y compiladores [89]. Se pueden encontrar conexiones más recientes entre Álgebra Universal y ciencias de la computación en la teoría de Instituciones [29] como fundamento de metodologías y frameworks para especificar y desarrollar software [82].

Si bien existe mucho material sobre la teoría de álgebras heterogéneas (la mayoría heredado del estudio del caso homogéneo o monosort, pero no todo [88]), hay muy pocas publicaciones disponibles sobre formalizaciones en teoría de tipos, como analizaremos al final de esta sección. Esto contrasta con la cantidad de avances en la mecanización de resultados para álgebras particulares, como por ejemplo la prueba del teorema de Feit-Thompson en Coq desarrollada por Gonthier y su equipo [33].

En este capítulo presentamos una librería para el lenguaje Agda de álgebras multi-sort, la primera existente al momento, incluyendo conceptos avanzados como los morfismos entre signaturas y las álgebras reducto. Presentamos una novedosa representación de signaturas heterogéneas, modelando las operaciones como familias indexadas en las aridades y para ello desarrollamos una librería independiente de vectores heterogéneos. Formalizamos la prueba de que el álgebra de

términos es inicial y los tres teoremas de isomorfismo. Definimos también un sistema deductivo para la lógica ecuacional condicional probando su corrección y completitud, y mostrando que la traducción de teorías a partir de un morfismo entre signaturas induce un funtor contra-variante entre los modelos de ellas. Mostraremos en este texto solo algunas definiciones sin entrar en demasiados detalles de la implementación, en el código fuente incluimos algunos ejemplos de uso del sistema ecuacional y de la preservación de modelos via morfismos de signaturas.

Presentaremos en la primera sección los conceptos básicos del Álgebra Universal: signatura, álgebras y homomorfismos, congruencias, cocientes y subálgebras, las pruebas de los tres teoremas de isomorfismo, y la inicialidad del álgebra de términos. En la sección 2 definimos el cálculo ecuacional, introduciendo los conceptos de ecuación, teoría, satisfacción y deducción, finalizando con la prueba de Birkhoff de corrección y completitud. En la sección 3 introducimos una nueva representación de morfismos entre signaturas y álgebra reducto, y exploramos la traducción de teorías.

4.1 ÁLGEBRA UNIVERSAL

Presentaremos la formalización en Agda de los principales conceptos de álgebras heterogéneas, tomando como referencia el handbook de Meinke y Tucker [61].

Signatura, álgebra y homomorfismo

Signatura

Una *signatura* es un par de conjuntos (S, F), llamados *sorts* y *operaciones* (o *símbolos de función*) respectivamente; cada operación es una tripla $(f, [s_1, \ldots, s_n], s)$ consistente de un *nombre*, su *aridad* y el *sort target* (usaremos también la notación $f\colon [s_1, ..., s_n] \Rightarrow s$).

Definimos el tipo Signature como un record-type con dos campos: los sorts y las operaciones. A estas últimas las representamos como una familia indexada por la aridad (que será una lista de sorts) y el sort target:

```
record Signature : Set where
  field
    sorts : Set
    ops   : List sorts × sorts → Set
```

Si queremos definir una signatura concreta, primero debemos declarar un set correspondiente a los sorts y otro para las operaciones. Consideremos como ejemplo la signatura de monoides que tiene un único sort por lo cual podemos usar, para representar a los sorts, el

tipo *unit* ⊤ con su único constructor tt. Las operaciones las definimos con una familia indexada en List ⊤ × ⊤ correspondientes al tipo de las aridades y los target sorts. Los constructores de esta familia indexada corresponden a la constante e y a la operación binaria del monoide:

```
data monoid-op : List ⊤ × ⊤ → Set where
  e : monoid-op ([] , tt)
  ⊙ : monoid-op (tt ∷ [ tt ] , tt)
monoid-sig : Signature
monoid-sig = record { sorts = ⊤ ; ops = monoid-op }
```

Un ejemplo de signatura heterogénea es el de las acciones de monoide la cual tiene dos sorts, uno para el monoide y otro para el conjunto donde el monoide actúa.

```
data actMon_S : Set where
  mon : actMon_S
  set : actMon_S
data actMon_O : List actMon_S × actMon_S → Set where
  e  : actMon_O ([] , mon)
  ⊙ : actMon_O ( mon ∷ [ mon ] , mon)
  *  : actMon_O ( mon ∷ [ set ] , set)
actMon-sig : Signature
actMon-sig = record { sorts = actMon_S ; ops = actMon_O }
```

Definir las operaciones con una familia indexada trae ventajas en el uso de la librería: como se ve en los ejemplos, los nombres de las operaciones son constructores de un tipo indexado por lo cual se puede realizar pattern matching en ellos, también permite expresar a nivel de tipos del lenguaje Agda los símbolos de función de determinada aridad lo cual es muy beneficioso al momento de definir construcciones más complejas que veremos más adelante. También es posible definir signaturas con un número infinito de operaciones.

Álgebra

Un *álgebra* $\mathcal{A}$ para la signatura Σ consiste de una familia de tipos indexada en los sorts de Σ y una familia de funciones indexada en las operaciones de Σ. Utilizaremos $\mathcal{A}_s$ para referirnos a la *interpretación* (o el *carrier*) del sort s. Dada una operación $f : [s_1, ..., s_n] \Rightarrow s$, la interpretación de f es una función total $f_{\mathcal{A}} : \mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n} \to \mathcal{A}_s$. Formalizamos el producto $\mathcal{A}_{s_1} \times ... \times \mathcal{A}_{s_n}$ con *vectores heterogéneos*. El tipo de vectores heterogéneos está parametrizado sobre un tipo I y una familia indexada por I; y es indexado con una lista de I:

```
data HVec {I : Set} (A : I → Set) : List I → Set where
  ⟨⟩  : HVec A []
  _▷_ : ∀ {i is} → A i → HVec A is → HVec A (i ∷ is)
```

Necesitamos un ingrediente más para poder dar la definición de álgebra en nuestra formalización: la noción matemática de carriers (o interpretación de sorts) asume una noción de igualdad en el conjunto. En teoría de tipos es adecuado representar los carriers con *setoides* (un par consistente de un tipo de datos y una relación de equivalencia definida en el mismo), y el lector interesado podrá encontrar detalles en el trabajo de Barthe et al. [6]. Los setoides están definidos en la librería estándar de Agda como un record con tres campos.

```
record Setoid : Set where
  field
    Carrier : Set
    _≈_     : Carrier → Carrier → Set
    isEquivalence : IsEquivalence _≈_
```

La relación de igualdad del setoide está definida como una familia indexada en dos elementos del conjunto (al cual también se lo llama carrier), luego dos elementos a y b de tipo Carrier están relacionados si el tipo $a \approx b$ está habitado. El tipo IsEquivalence _≈_ es también un record cuyos campos corresponden a las pruebas de reflexividad, transitividad y simetría de la relación de igualdad.

Cualquier tipo de datos puede ser convertido en un setoide donde la relación de equivalencia es la igualdad proposicional (donde cada término es igual sólo a sí mismo) mediante la función setoid : Set → Setoid de la librería estándar. Por otro lado definimos una abreviatura ∥_∥ para obtener el tipo subyacente en un setoide.

Como los sorts serán interpretados por setoides, es necesario que la interpretación de operaciones respete esa estructura. Los morfismos entre setoides son funciones que preservan la igualdad y están definidos en la librería estándar con un record parametrizado en dos setoides:

```
record _⟶_ (A B : Setoid) : Set where
  field
    _⟨$⟩_ : ∥ A ∥ → ∥ B ∥
    cong : ∀ {a a'} → _≈_ A a a' → _≈_ B (_⟨$⟩_ a) (_⟨$⟩_ a')
```

El campo cong corresponde a la prueba de preservación de igualdad de la función.

La interpretación de una operación en un álgebra será un morfismo del setoide vector correspondiente a su aridad, al setoide del sort de llegada. El tipo Algebra se define mediante un record con dos campos: la interpretación de los sorts y la interpretación de las operaciones:

```
record Algebra (Σ : Signature) : Set where
  field
    _⟦_⟧s : sorts Σ → Setoid
    _⟦_⟧o : ∀ {ar s} → ops Σ (ar , s) → _⟦_⟧s ∗ ar ⟶ _⟦_⟧s s
```

El operador _∗_ construye el setoide de los vectores heterogéneos, donde cada elemento corresponde con la interpretación de acuerdo a la aridad. Ejemplos de álgebras pueden encontrarse en el código fuente de la librería, y en este texto más adelante mostraremos algunos.

Homomorfismo

Sea Σ una signatura y sean $\mathcal{A}$ y $\mathcal{B}$ Σ-álgebras. Un *homomorfismo* h de $\mathcal{A}$ a $\mathcal{B}$ es una familia de funciones indexada por los sorts $h_s : \mathcal{A}_s \to \mathcal{B}_s$, que respeta la interpretación de las operaciones, es decir, para cada operación $f : [s_1, ..., s_n] \Rightarrow s$ se satisface lo siguiente:

$$h_s(f_{\mathcal{A}}(a_1, ..., a_n)) = f_{\mathcal{B}}(h_{s_1}\, a_1, ..., h_{s_n}\, a_n) \tag{2}$$

Para formalizar homomorfismos definimos una notación más cómoda para referirnos a familias de morfismos entre setoides indexadas por los sorts:

```
_⇝_ : ∀ {Σ} → Algebra Σ → Algebra Σ → Set
_⇝_ {Σ} A B = (s : sorts Σ) → A ⟦ s ⟧s ⟶ B ⟦ s ⟧s
```

La condición (2) la formalizamos como un predicado sobre familias de morfismos entre setoides:

```
homCond : ∀ {Σ} {A B} → A ⇝ B → Set
homCond {A = A} {B} h =
          ∀ {ar s} (f : ops Σ (ar , s) ) → (as : A ⟦ ar ⟧s*) →
          (h s ⟨$⟩ (A ⟦ f ⟧o ⟨$⟩ as)) ≈s (B ⟦ f ⟧o ⟨$⟩ (map h as))
```

donde $\_\approx_s\_$ es la relación de equivalencia del setoide $B \llbracket s \rrbracket_s$ y map h es la extensión del morfismo h sobre vectores. El operador $\_\llbracket\_\rrbracket_s{}^*$ extiende la interpretación de sorts a vectores. Un homomorfismo se define como un record parametrizado por las álgebras conteniendo un campo para el morfismo y otro para la condición:

```
record Homo {Σ} (A B  : Algebra Σ) : Set where
  field
    '_' : _⇝_
    cond : homCond '_'
```

Una vez definido el tipo para los homomorfismos, podemos definir la identidad Id_h A :Homo A A y la composición $G \circ_h F$: Homo A C de los homomorfismos F : Homo A B y G : Homo B C. También es esperable que componer con la identidad resulte en el mismo

homomorfismo. Para ello debemos dar una noción de igualdad entre homomorfismos, que implicará tener igualdad entre morfismos de setoides, y como vimos en el capítulo anterior, en Agda no tenemos igualdad definicional para las funciones por lo cual definimos la igualdad extensional:

```
_≈ext_ : (f g :  A⟶B)  → Set
f ≈ext g = ∀ (a : ∥ A ∥) → (f ⟨$⟩ a) ≈B (g ⟨$⟩ a)
```

donde A y B son setoides, y $\approx_B$ es la relación de equivalencia de B. Comparamos homomorfismos usando esa noción de igualdad:

```
_≈h_ : ∀ {Σ} {A B} → (H G : Homo A B) → Set _
H ≈h G = (s : sorts Σ) → (′ H ′ s) ≈ext (′ G ′ s)
```

Definida la noción de igualdad, es directo probar la asociatividad de la composición entre homomorfismos y que Id_h es la identidad.

Cocientes y subálgebras

Para probar los principales resultados del Álgebra Universal necesitamos formalizar algunas construcciones básicas: subálgebras, congruencias y cocientes.

Subálgebra

Una Σ-álgebra $\mathcal{B}$ es subálgebra de la Σ-álgebra $\mathcal{A}$ si se satisface $\mathcal{B}_s \subseteq \mathcal{A}_s$ para cada sort s, y para cada operación $f : [s_1, \ldots, s_n] \Rightarrow s$, vale la siguiente ecuación:

$$(a_1, \ldots, a_n) \in \mathcal{B}_{s_1} \times \cdots \times \mathcal{B}_{s_n} \text{ implica } f_{\mathcal{A}}(a_1, \ldots, a_n) \in \mathcal{B}_s \quad (3)$$

Para formalizar esta construcción necesitamos definir la noción de subconjunto. Como muestran Salvesen y Smith en [81], extender una teoría de tipos intensional con subconjuntos genera que se pierdan propiedades deseables. Por lo tanto tenemos que formalizarlo de otra manera: si consideramos que un subconjunto B está dado por un predicado P sobre los elementos de A, podemos pensar que $a : A$ pertenece al subconjunto B si podemos dar una prueba $p : P\ a$. Un elemento del sobconjunto, entonces, será un par dependiente `Σ[ a ∈ A ] P` donde $P : A \to \mathsf{Set}$ y $A : \mathsf{Set}$. Observemos que si tenemos $p : P\ a$ y $p' : P\ a$, entonces habrá dos elementos en el subconjunto que en realidad representan al mismo: son dos pruebas de que a satisface el predicado.

En nuestra formalización de álgebras los conjuntos que interpretan a los sorts de la signatura están implementados con setoides, por lo tanto necesitamos dotar a nuestra representación de subconjuntos con la estructura de un setoide, y podremos resolver el problema de

tener más de un representante por cada a que satisface el predicado, mediante la relación de equivalencia, obteniendo lo que se conoce como *proof-irrelevance*. Si A es un setoide y $P : \| A \| \to \mathsf{Set}$ un predicado sobre su carrier, el subsetoide tendrá como carrier a los pares de elementos en $\Sigma[\, a \in \| A \| \,]\, P$ y su relación de equivalencia estará definida por $(a,p) \approx (a',q)$ si y solo si $a \approx a'$. Para que esto tenga sentido, el predicado P debería ser consistente con la relación de equivalencia del setoide. Decimos que P está *bien definido* si cada vez que $a \approx a'$ y se satisface $P\ a$, entonces se satisface $P\ a'$:

```
WellDef : (A : Setoid) → (P : ∥ A ∥ → Set) → Set _
WellDef A P = ∀ {a a' } → a ≈ a'→ P a → P a'
```

El único ingrediente que nos falta para poder formalizar subálgebras es que el conjunto sea cerrado bajo las operaciones, Ec. (3). Sea Σ una signatura y A una Σ-álgebra:

```
OpClosed : (P : (s : sorts Σ) → ∥ A ⟦ s ⟧s ∥ → Set) → Set
OpClosed P = ∀ {ar s} (f : ops Σ (ar , s)) → (P ∗ ⟨→⟩ P s) (A ⟦ f ⟧o ⟨$⟩_)
```

$(Q\ \langle\to\rangle\ R)\ f$ puede interpretarse "la precondición Q implica la postcondición R luego de aplicar la función f", a su vez con la notación $\_*$ nos referimos a la extensión de un predicado sobre vectores, por lo tanto `OpClosed P f` expresa que si un vector $as*$ satisface el predicado P, luego la aplicación de la interpretación $A\ ⟦\ f\ ⟧_o$ al vector $as*$ también satisface P, de acuerdo con la ecuación (3).

Resumiendo, para poder definir una subálgebra de una Σ-álgebra A necesitamos una familia P de predicados indexada en los sorts de Σ, tal que $P\ s$ esté bien definido para cada sort s y P es cerrado por las operaciones de Σ:

```
SubAlgebra : ∀ {Σ} A P → ((s : sorts Σ) → WellDef (P s)) →
                                  OpClosed P → Algebra Σ
```

no mostramos esta definición ya que abunda en detalles de notación. La familia de predicados determinará la interpretación de los sorts mediante el setoide que representa cada subconjunto (por lo tanto su carrier serán pares de elementos y pruebas de satisfacción del predicado), y para las operaciones, la aplicación $\langle\$\rangle$ devolverá la interpretación de A en el primer elemento del par, junto con una prueba de satisfacción de P que se obtiene gracias a `OpClosed`; por otro lado, la preservación de la equivalencia `cong` se hereda directamente de A, ya que tenemos proof-irrelevance.

Congruencias y cocientes

Una *congruencia* en una Σ-álgebra $\mathcal{A}$ es una familia Q de relaciones de equivalencia indexada por los sorts de Σ tal que cada una es cerra-

da por las operaciones: Sean $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in \mathcal{A}_{s_1} \times \ldots \times \mathcal{A}_{s_n}$

$$\begin{aligned} &(a_1, b_1) \in Q_{s_1}, \ldots, (a_n, b_n) \in Q_{s_n} \quad \text{implica} \\ &(f_{\mathcal{A}}(a_1, \ldots, a_n), f_{\mathcal{A}}(b_1, \ldots, b_n)) \in Q_s \end{aligned} \tag{4}$$

para toda operación f.

Para formalizar esta condición definimos la extensión punto a punto _* (*point-wise extension* en inglés) de una familia de relaciones sobre vectores. Como vimos antes para subálgebras, puesto que la interpretación de sorts es formalizada con setoides una relación definida sobre el carrier de un setoide debe estar bien definida, es decir, debe preservar la igualdad subyacente. Una congruencia de un álgebra A será entonces una familia Q de relaciones de equivalencia bien definidas. La condición (4) es capturada por el operador _=[_]⇒_ de la librería estándar, donde si P es una relación binaria en algún tipo A, Q es una relación binaria en algún tipo B y f : A → B, luego P =[f]⇒ Q expresa que si a : A, a' : A, $P\ a\ a'$ entonces $Q\ (f\ a)\ (f\ a')$.

```
record Congruence (A : Algebra Σ) : Set where
  field
    rel : (s : sorts Σ) → ∥ A ⟦ s ⟧s ∥ → ∥ A ⟦ s ⟧s ∥ → Set
    welldef : (s : sorts Σ) → WellDefRel (A ⟦ s ⟧s) (rel s)
    cequiv : (s : sorts Σ) → IsEquivalence (rel s)
    csubst : ∀ {ar s} (f : ops Σ (ar , s)) → rel * =[ A ⟦ f ⟧o ⟨$⟩_ ]⇒ rel s
```

Dada una congruencia Q sobre el álgebra $\mathcal{A}$, se puede definir el *álgebra cociente*, interpretando cada sort s en el conjunto de clases de equivalencia $\mathcal{A}_s/Q$ y cada operación $f : [s_1, \ldots, s_n] \Rightarrow s$ como la función que mapea el vector $([a_1], \ldots, [a_n])$ de clases de equivalencia en la clase $[f_{\mathcal{A}}(a_1, \ldots, a_n)]$, que está bien definida gracias a la condición (4). Para formalizar el álgebra cociente en Agda, tomamos los mismos setoides que interpretan a los sorts en el álgebra original A y utilizamos $Q\ s$ como la relación de equivalencia sobre ∥ A ⟦ s ⟧s ∥; las operaciones son interpretadas de la misma manera que en A, donde la prueba de congruencia está dada por csubst Q.

```
_/_ : (A : Algebra Σ) → (C : Congruence A) → Algebra Σ
```

Los teoremas de isomorfismo

Las construcciones de subálgebras, cocientes y homomorfismos suryectivos están relacionadas por los tres teoremas de isomorfismo. A pesar de que las definiciones de subálgebra y cociente en la formalización en Agda agregan un poco burocracia en el tratamiento de

subconjuntos por el uso de setoides, las pruebas formalizadas no distan de las que se realizan en la matemática tradicional y que se encuentran en la bibliografía sobre Álgebra Universal. Para probar estos resultados hemos definido también la noción de *kernel* y la imagen *homomórfica* de homomorfismos, como también la noción de isomorfismo.

Teorema 1. *Si* $h : \mathcal{A} \to \mathcal{B}$ *es un epimorfismo, luego* $\mathcal{A}/\ker h \simeq \mathcal{B}$.

Teorema 2. *Si* φ, ψ *son congruencias sobre* $\mathcal{A}$*, tal que* $\psi \subseteq \varphi$*, luego* $(\mathcal{A}/\varphi) \simeq (\mathcal{A}/\psi)/(\varphi/\psi)$.

Teorema 3. *Sea* $\mathcal{B}$ *una subálgebra de* $\mathcal{A}$*, y* φ *una congruencia sobre* $\mathcal{A}$*. Sea* $[\mathcal{B}]^\varphi = \{K \in A/\varphi : K \cap B \neq \varnothing\}$ *y* φ_B *la restricción de* φ *a* $\mathcal{B}$*, luego*

(i) φ_B *es una congruencia sobre* $\mathcal{B}$*;*

(ii) $[\mathcal{B}]^\varphi$ *es una subálgebra de* $\mathcal{A}$*;*

(iii) $[\mathcal{B}]^\varphi \simeq \mathcal{B}/\varphi_B$.

Álgebra de términos

Una Σ-álgebra $\mathcal{A}$ es *inicial* si para toda Σ-álgebra $\mathcal{B}$ existe exactamente un homomorfismo de $\mathcal{A}$ a $\mathcal{B}$. Definimos primero el predicado de que un setoide tiene un único elemento:

```
Unique {A = A} _≈_ = A × (∀ a a' → a ≈ a')
```

A tiene un único elemento con respecto a _≈_ si podemos dar un elemento y probar que todo par de elementos está relacionado por _≈_. Gracias a esta definición, podemos formalizar inmediatamente la noción de álgebra inicial:

```
Initial : ∀ {Σ} → (A : Algebra Σ) → Set
Initial {Σ} A = (B : Algebra Σ) → Unique (_≈h_ A B)
```

Dada una signatura Σ definimos el *álgebra de términos* $\mathcal{T}$, cuyos carriers son conjuntos de palabras bien tipadas que pueden construirse a partir de las operaciones. Se suele llamar a este universo con el nombre de *Herbrand Universe* y se define inductivamente:

$$\frac{t_1 \in \mathcal{T}_{s_1} \qquad \cdots \qquad t_n \in \mathcal{T}_{s_n}}{f\,(t_1, \ldots, t_n) \in \mathcal{T}_s}\; f : [s_1, \ldots, s_n] \Rightarrow s$$

y lo definimos directamente en Agda con un tipo inductivo:

```
data HU {Σ : Signature} : (s : sorts Σ) → Set where
  term : ∀ {ar s} → (f : ops Σ (ar , s)) → (HVec HU ar) → HU s
```

Podemos convertir este tipo en un setoide utilizando como relación de equivalencia la igualdad proposicional mediante la función setoid

de la librería estándar. La interpretación de una operación $f : [s_1, \ldots, s_n] \Rightarrow s$ será la función que lleva un vector de términos $\langle t_1,\ldots,t_n \rangle$: HVec HU $[s_1,\ldots,s_n]$ al término term $f\ \langle t_1,\ldots,t_n \rangle$; omitimos la prueba de congruencia, la cual es larga y tediosa para ser mostrada en este texto.

```
|T| : ∀ Σ → Algebra Σ
|T| = record { _⟦_⟧s = setoid ∘ HU
             ; _⟦_⟧o = |_|o
             }
  where |f|o = record { _⟨$⟩_ = term f
                      ; cong = ... }
```

Un término puede interpretarse en cualquier álgebra $\mathcal{A}$ dando lugar a un homomorfismo $h_A : \mathcal{T} \to \mathcal{A}$

$$h_A(f(t_1, \ldots, t_n)) = f_{\mathcal{A}}\ (h_A\ t_1, \ldots, h_A\ t_n)\ .$$

y su definición en Agda es la siguiente:

```
|h| : ∀ {Σ} → (A : Algebra Σ) → {s : sorts Σ} → HU s → ∥ A ⟦ s ⟧s ∥
|h| A (term f ts) = A ⟦ f ⟧o ⟨$⟩ |h|* A ts
```

donde |h|* es la extensión a vectores de la función |h|, definida mediante una definición mutuamente recursiva.

Es sencillo probar que |h| preserva la igualdad y que satisface la condición de homomorfismo. Finalmente probamos que el álgebra de términos |T| Σ es inicial por recursión en la estructura de los términos mostrando que si F y G son homomorfismos de |T| Σ a otra álgebra A entonces son extensionalmente iguales:

```
terminit : ∀ {Σ} → (A : Algebra Σ) → (F G : Homo (|T| Σ) A) → F ≈h G
```

en la prueba se utiliza la condición de homomorfismo de H y G, y la propiedad de congruencia de la interpretación de las operaciones.

4.2 LÓGICA ECUACIONAL

En esta sección introducimos la noción de teorías ecuacionales condicionales y la correspondiente noción de satisfactibilidad de teorías en álgebras. Formalizamos la lógica ecuacional condicional como está presentada por Goguen y Lin en [30] y probamos que el sistema deductivo es correcto y completo.

Álgebra libre con variables

El álgebra de términos que definimos previamente contiene solamente términos sin variables (*ground terms*). Dada una signatura Σ y una familia X : sorts Σ → Set de variables, definimos una nueva signatura extendiendo Σ con X, tomando las variables como nuevas constantes (es decir, operaciones con aridad []).

```
_(|_|) : (Σ : Signature) → (X : Vars Σ) → Signature
Σ (| X |) = record { sorts = sorts Σ ; ops = ops' }
```

```
  where ops' : _
        ops' ([] , s) = ops Σ ([] , s) ⊎ X s
        ops' (ar , s) = ops Σ (ar , s)
```

es interesante notar que gracias a que definimos las operaciones como familias indexadas en la aridad, es muy fácil extender una signatura con operaciones de alguna aridad específica, como en este caso las constantes.

A partir de la signatura extendida Σ⟪ X ⟫ puede obtenerse el álgebra de términos correspondiente |T| (Σ⟪ X ⟫), que puede ser vista como una Σ-álgebra simplemente ignorando las nuevas constantes:

```
T_⟪_⟫ : (Σ : Signature) → (X : Vars Σ) → Algebra Σ
T Σ ⟪ X ⟫ = record { _⟦_⟧s = λ s → |T| (Σ⟪ X ⟫) ⟦ s ⟧s ; _⟦_⟧o = iops }
  where iops : _
        iops {[]} f       = |T| (Σ⟪ X ⟫) ⟦ inj₁ f ⟧o
        iops {s₀ : ar} f = |T| (Σ⟪ X ⟫) ⟦ f ⟧o
```

para interpretar las constantes, tomamos la interpretación en el álgebra de términos donde con inj_1 distinguimos las constantes originales de Σ de las variables.

Si queremos definir una función de |T| (Σ⟪ X ⟫) en un álgebra cualquiera necesitamos dar un valor a las variables. Los *entornos* fijan la interpretación de las variables: Env X A = ∀ {s} → X s → ∥ A ⟦ s ⟧s ∥. El álgebra T Σ ⟪X⟫ tiene la propiedad conocida como *freeness*: Dada una Σ-álgebra A y un entorno θ : Env X A, existe un único homomorfismo ⟦_⟧θ : Homo (T Σ ⟪X⟫) A tal que ⟦ x ⟧θ = θ x, para cada variable x de cada sort s[1].

Cuasi identidades, satisfacción y pruebas

Cuasi identidades

En el contexto de álgebras monosort u homogéneas, una ecuación es un par de términos donde todas las variables se asumen cuantificadas universalmente y una teoría es un conjunto (finito) de ecuaciones. En el caso multisort ambos lados de una ecuación deben ser términos del mismo sort, naturalmente. Definiremos cuasi-identidades que escribiremos como ecuaciones condicionales:

$$t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \ .$$

Sea Σ una signatura y X : sorts Σ → Set una familia de variables para Σ, definimos una identidad e : Eq X s como un par de términos con

1 Aquí hacemos un abuso de notación para referirnos al término correspondiente a la variable x en el álgebra T Σ ⟪X⟫.

variables de sort s, y utilizaremos la notación $\bigwedge\_\approx\_$. Una cuasi identidad o ecuación condicional será modelada con un record conteniendo la identidad y las condiciones mediante un vector heterogéneo de identidades:

```
record Equation (Σ : Signature) (X : Vars Σ) (s : sorts Σ) : Set where
  constructor _if_
  field
    eq   : Eq Σ X s
    cond : Σ[ ar ∈ List (sorts Σ) ] (HVec (Eq Σ X) ar)
```

una cuasi-identidad $t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ puede ser modelada en la formalización mediante $\bigwedge t \approx t'$ if (ar, eqs), donde eqs es un vector de identidades con aridad ar.

Una *teoría* sobre una signatura Σ es un vector de cuasi-identidades:

```
Theory : ∀ Σ → (X : sorts Σ → Set) → (ar : List (sorts Σ)) → Set
Theory Σ X ar = HVec (Equation Σ X) ar
```

notar que en nuestra formalización todas las ecuaciones de una teoría comparten el mismo conjunto de variables, contrastando con el cálculo de Goguen y Lin, donde cada ecuación tiene su propio conjunto de variables cuantificadas.

Satisfacción

Sea Σ una signatura y $\mathcal{A}$ una Σ-álgebra. Decimos que la ecuación condicional $t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ se *satisface* en $\mathcal{A}$ si para todo entorno $\theta : X \to \mathcal{A}$, se cumple $[\![t]\!]\theta = [\![t']\!]\theta$, siempre que $[\![t_i]\!]\theta = [\![t'_i]\!]\theta$, con $1 \leqslant i \leqslant n$. Para formalizar esta noción definimos primero cuándo un entorno *modela* una identidad:

```
_⊨e_ : ∀ {Σ X A} → (θ : Env X A) → ∀ {s} → Eq Σ X s → Set
_⊨e_ θ {s} (⋀ t ≈ t') = ⟦ t ⟧ θ ≈A ⟦ t' ⟧ θ
```

donde _≈A_es la igualdad del setoide correspondiente a la interpretación del sort s en el álgebra A. La satisfacción de una cuasi-identidad la obtenemos directamente de la definición anterior, utilizando la extensión de $\_\models_e\_$ al vector de ecuaciones que conforman la condición:

```
_⊨_ : ∀ {Σ X} → (A : Algebra Σ) → ∀ {s} → Equation Σ X s → Set
_⊨_ A (eq if (_ , eqs)) = ∀θ → ((θ ⊨e_) * eqs) → θ ⊨e eq
```

Decimos que un álgebra $\mathcal{A}$ es un *modelo* de una teoría E si satisface cada ecuación en E. Una ecuación es una consecuencia lógica de una teoría si todo modelo de ésta la satisface.

```
_⊨m_ : ∀ {Σ X ar} → (A : Algebra Σ) → (E : Theory Σ X ar) → Set
A ⊨m E = (A ⊨_) * E
```

```
_⊨Σ_ : ∀ {Σ X ar s} → (E : Theory Σ X ar) → (e : Equation Σ X s) → Set
_⊨Σ_ {Σ} E e = (A : Algebra Σ) → A ⊨m E → A ⊨ e
```

$$\frac{}{E \vdash \forall X,\ t = t} \qquad \frac{E \vdash \forall X,\ t_0 = t_1}{E \vdash \forall X,\ t_1 = t_0}$$

$$\frac{E \vdash \forall X,\ t_0 = t_1 \qquad E \vdash \forall X,\ t_1 = t_2}{E \vdash \forall X,\ t_0 = t_2}$$

$$\frac{\forall Y,\ t = t' \text{ if } t_1 = t_1', \ldots, t_n = t_n' \in E \qquad E \vdash \forall X,\ \sigma(t_i) = \sigma(t_i')}{E \vdash \forall X,\ \sigma(t) = \sigma(t')}\ \sigma \colon Y \to T_\Sigma(X)$$

$$\frac{E \vdash \forall X,\ t_1 = t_1' \qquad \cdots \qquad E \vdash \forall X,\ t_n = t_n'}{E \vdash \forall X,\ f\,(t_1, \ldots, t_n) = f\,(t_1', \ldots, t_n')}\ f : [s_1, \ldots, s_n] \Rightarrow_\Sigma s$$

Figura 6: Sistema deductivo

Sistema deductivo

Como muestran Huet y Oppen [43], la definición de un sistema deductivo para el caso multisort de la lógica ecuacional es un poco más complicado que para el caso homogéneo. Formalizamos el sistema presentado en [30] que mostramos en la figura 6. Las tres primeras reglas corresponden a reflexividad, simetría y transitividad; la cuarta regla, habitualmente llamada sustitución, permite instanciar un axioma con una substitución σ, si se provee una prueba por cada condición del axioma; finalmente la quinta regla internaliza la regla de Leibniz reemplazando iguales por iguales en subtérminos. Es importante notar que el sistema deductivo sólo permite probar identidades y no ecuaciones condicionales.

Definimos un tipo inductivo para representar las pruebas de identidades, el cual está parametrizado en la teoría E e indexado por la conclusión de la prueba.

```
data _⊢_ {Σ X ar} (E : Theory Σ X ar) : ∀ {s} → Eq Σ X s → Set where
  prefl  : ∀ {t} → E ⊢ (⋀ t ≈ t)
  psym   : ∀ {t t'} → E ⊢ (⋀ t ≈ t') → E ⊢ (⋀ t' ≈ t)
  ptrans : ∀ {t₀ t₁ t₂} → E ⊢ (⋀ t₀ ≈ t₁) → E ⊢ (⋀ t₁ ≈ t₂) → E ⊢ (⋀ t₀ ≈ t₂)
  psubst : ∀ {t t' ar'} {eqs : HVec (Eq Σ X) ar'} →
             ((⋀ t ≈ t') if (ar' , eqs)) ∈ E → (σ : Subst) →
             ((λ _ e → E ⊢ e) *) eqs → E ⊢ (⋀ σ t ≈ σ t')
  prepl  : ∀ {ar's ts ts'} → ((λ sᵢ tᵢ tᵢ' → E ⊢ (⋀ tᵢ ≈ tᵢ')) *) ts ts' →
             (f : ops (Σ ⟪ X ⟫) (ar' , s)) → E ⊢ (⋀ term f ts ≈ term f ts')
```

No mostramos en este texto varias definiciones que se utilizan en `_⊢_` como `Subst`, correspondiente a substituciones de variables por términos. Para la regla de sustitución, formalizada con el constructor `psubst`, utilizamos la función de pertenencia a vectores heterogéneos refiriendo de esa manera a un axioma de la teoría, y la extensión de las pruebas a vectores de ecuaciones. En el caso de la regla de reemplazo (o Leibniz), formalizada con el constructor `prepl`, tomamos dos vectores correspondientes a los subtérminos antes y después del reemplazo. Tanto en `psubst` como en `prepl` utilizamos la extensión de predicados y relaciones a vectores.

Sea E una teoría sobre la signatura Σ y el conjunto de variables X, es simple definir un setoide sobre el tipo de los términos, donde la relación de igualdad es $t \approx t'$ si $E \vdash \bigwedge t \approx t'$, que es una congruencia sobre el álgebra de términos. Gracias a las facilidades sintácticas del módulo de razonamiento ecuacional de la librería estándar de Agda, podemos escribir pruebas en nuestro sistema deductivo de una manera muy limpia, como veremos en un ejemplo más adelante.

Sobre este cálculo ecuacional hemos probado corrección y completitud, por inducción en las reglas de pruebas para el primer caso; y en el segundo caso gracias a que el cociente del álgebra de términos por la congruencia de la relación de pruebas es un modelo.

Teorema 4 (Corrección y Completitud). $E \vdash t \approx t'$ *sii* $E \models_\Sigma t \approx t'$.

Dadas dos teorías E y E' sobre una signatura Σ y un conjunto de variables X, decimos que E es *más fuerte* que E' si cada axioma $e \in E'$ puede probarse en el sistema deductivo con E como teoría, y lo escribimos $E \vdash_T E'$. Como consecuencia de esta definición y por corrección, si E es más fuerte que E', luego cada ecuación que puede deducirse de E' también se deduce de E y cualquier modelo de E es modelo de E'.

Ejemplo: Una teoría para álgebras booleanas

Como un ejemplo del cálculo ecuacional mostramos un fragmento[2] de la formalización de una teoría booleana que se menciona en [79]. La signatura tiene un único sort, por lo cual utilizamos el tipo Unit.

```
data bool-ops : List ⊤ × ⊤ → Set where
  t   : bool-ops ([] , tt)
  f   : bool-ops ([] , tt)
  neg : bool-ops ([ tt ] , tt)
  and : bool-ops ((tt : [ tt ]) , tt)
  or  : bool-ops ((tt : [ tt ]) , tt)

bool-sig : Signature
bool-sig = record { sorts = ⊤ ; ops = bool-ops }
```

tenemos dos operaciones con aridad vacía representando las constantes `true` y `false`, una operación unaria para representar la negación, y dos binarias representando los conectivos de conjunción y disjunción.

Como variables utilizaremos el tipo de los números naturales, y definimos *smart-constructors* para facilitar la escritura de las fórmulas:

```
V : Vars bool-sig
V s = ℕ

Form : Set
Form = HU (bool-sig ⟪ V ⟫) tt
```

2 el código completo puede verse en "Examples/EqBool.agda" de la URL citada

```
true : Form
true = term (inj₁ t) ⟨⟩

false : Form
false = term (inj₁ f) ⟨⟩

_∧_ : Form → Form → Form
φ ∧ ψ = term and ⟨ φ , ψ ⟩

_∨_ : Form → Form → Form
φ ∨ ψ = term or ⟨ φ , ψ ⟩

p : Form
p = term (inj₂ 0) ⟨⟩

q : Form
q = term (inj₂ 1) ⟨⟩

r : Form
r = term (inj₂ 2) ⟨⟩

¬ : Form → Form
¬ φ = term neg ⟨ φ ⟩
```

las funciones p, q y r son fórmulas correspondientes a tres variables distinguibles entre sí, necesarias para la escritura de axiomas y teoremas. Gracias a las facilidades de la sintaxis de Agda, mediante estos smart-constructors podemos escribir fórmulas de manera muy similar al papel, como por ejemplo p ∧ (true ∨ r), en vez de utilizar los constructores para los términos del álgebra correspondiente.

De los doce axiomas que tiene la teoría mostramos solo dos:

```
commAnd : Equation bool-sig V tt
commAnd = (⋀ p ∧ q ≈ (q ∧ p)) if ([] , ⟨⟩)

falseDef : Equation bool-sig V tt
falseDef = (⋀ p ∧ (¬ p) ≈ false) if ([] , ⟨⟩)

Tbool : Theory bool-sig V (repeat tt 12)
Tbool = commAnd ▷ falseDef ▷ ... ▷ ⟨⟩
```

una prueba p en la teoría Tbool será un término de Agda con tipo Tbool ⊢ p. A continuación mostramos una prueba en donde se utilizan los dos axiomas de Tbool que mostramos previamente. La regla que nos permite aplicar axiomas donde se sustituyen posiblemente variables por términos es psubst, que requiere mostrar que el axioma utilizado pertenece a la teoría. Para ello utilizamos la facilidad de Agda de patterns-synonyms definiendo los patrones commAnd_ax y falseDef_ax como abreviaturas para las pruebas de commAnd ∈ Tbool y falseDef ∈ Tbool, respectivamente.

```
p₁ : Tbool ⊢ (⋀ ¬ p ∧ p ≈ false)
p₁ = begin
       ¬ p ∧ p
       ≈⟨ psubst commAndax σ ⟨⟩ ⟩
       p ∧ ¬ p
       ≈⟨ psubst falseDefax idSubst ⟨⟩ ⟩
       false
         □
```

la prueba consta de dos pasos. El primero es la aplicación del axioma "conmutatividad de la conjunción", donde se substituye la variable p por el término ¬ p, y la variable q por el término p (la definición de la substitución σ, que no mostramos en este texto, coincide con lo que acabamos de mencionar). El segundo paso es la aplicación del axioma "definición de false", en donde la substitución utilizada es la identidad, ya que el término en la prueba coincide exactamente con el del axioma. El módulo de la librería estándar de Agda para razonamiento ecuacional permite que podamos escribir pruebas como la que mostramos, de una manera muy sencilla, gracias a que la definición que dimos para las pruebas ecuacionales forman un setoide.

4.3 MORFISMOS ENTRE SIGNATURAS

El cálculo proposicional de Dijkstra y Scholten [22] es una teoría booleana alternativa a la que mostramos recién, donde las únicas operaciones no constantes son la equivalencia y la disjunción. Podemos definir la signatura en nuestra librería de este modo:

```
data bool-ops' : List ⊤ × ⊤ → Set where
  f'  : bool-ops' ([] , tt)
  t'  : bool-ops' ([] , tt)
  equiv' : bool-ops' (tt ∷ [ tt ] , tt)
  or' : bool-ops' (tt ∷ [ tt ] , tt)

bool-sig' : Signature
bool-sig' = record { sorts = ⊤ ; ops = bool-ops' }
```

Es claro que podríamos definir recursivamente una traducción de términos de la signatura bool-sig en términos de bool-sig' de manera que se preserve la semántica correspondiente: la fórmula ¬p puede traducirse a $p \equiv \mathit{false}$, y $p \wedge q$ puede traducirse a $(p \equiv q) \equiv (p \vee q)$. Una alternativa más general a definir una función que lleve términos en términos, es especificar cómo se puede representar cada operación de la signatura bool-sig con operaciones de bool-sig'. De esta manera, cualquier bool-sig'-álgebra $\mathcal{A}'$ puede ser vista como una bool-sig-álgebra: la interpretación de una bool-sig-operación f es la que $\mathcal{A}'$ define para la traducción de f. En particular la traducción de términos, que uno podría definir ad-hoc, se obtiene del homomorfismo inicial entre |T| bool-sig y |T| bool-sig', vista como álgebra de bool-sig. En esta sección presentamos nuestra formalización para los conceptos de *morfismos entre signaturas* (en la literatura *derived signature morphisms*) y *álgebras reducto* como están presentados por ejemplo por Sanella et al. [82].

$$\frac{}{[s_1, \ldots, s_n] \Vdash^{\Sigma} \sharp i : s_i}\ (\text{PRJ})$$

$$\frac{f : [s_1, \ldots, s_n] \Rightarrow_{\Sigma} s \quad ar \Vdash^{\Sigma} t_1 : s_1 \quad \cdots \quad ar \Vdash^{\Sigma} t_n : s_n}{ar \Vdash^{\Sigma} f\,(t_1, \ldots, t_n) : s}\ (\text{OP})$$

Figura 7: Sistema de tipos para los términos formales

Morfismos entre signaturas

Un morfismo entre dos signaturas puede pensarse como una especificación que indique cómo representar cada operación de la signatura de origen combinando operaciones de la signatura destino. Para las operaciones constantes t y f, y para la operación binaria or es obvio cómo podemos dar esta especificación, ya que se corresponden uno a uno con las operaciones t', f' y or' de la signatura bool-sig'. Sin embargo para las operaciones neg y and no tenemos estas correspondencias uno a uno.

Introducimos la noción de *términos formales* los cuales serán una composición formal de proyecciones y operaciones. Podemos pensarlos como *meta términos* de una signatura. El sistema de tipos de la figura 7 asegura que estos términos están bien formados: los contextos son aridades (i.e. listas de sorts), y los identificadores son punteros (como los índices de Bruijn). Podemos formalizarlo con un tipo parametrizado por las aridades e indexado en los sorts:

```
data _⊩_ (ar′ : List (sorts Σ)) : (sorts Σ) → Set where
  #     : (n : Fin (length ar′)) → ar′ ⊩ (ar′ !! n)
  _⟨$⟩_ : ∀ {ar s} → ops Σ (ar , s) → HVec (ar′ ⊩_) ar → ar′ ⊩ s
```

Un término formal especifica cómo interpretar una operación de la signatura origen en la signatura destino. La aridad *ar′* corresponde a los sorts de cada argumento de la operación original. Para referirse a uno de los argumentos de la operación se utiliza el constructor # con el índice correspondente al argumento referido. El otro constructor _⟨$⟩_ permite combinar operaciones mediante la aplicación de una operación a un vector de términos formales. Notemos que gracias a que tenemos las operaciones indexadas en las aridades, sólo podemos construir términos formales bien formados. En nuestro ejemplo las dos signaturas comparten el mismo tipo para los sorts, sin embargo en el caso general uno debe considerar mapear los sorts de la signatura origen en los de la de destino.

Un *morfismo entre las signaturas* Σ_s y Σ_t consiste de una función de los sorts de Σ_s en los sorts de Σ_t y una función de las operaciones de Σ_s a términos formales de Σ_t:

```
record _⇝_ (Σₛ Σₜ : Signature) : Set where
field
  ⇝ₛ : sorts Σₛ → sorts Σₜ
  ⇝ₒ : ∀ {ar s} → ops Σₛ (ar , s) → (map ⇝ₛ ar) ⊩ ⇝ₛ s
```

En nuestro ejemplo, ambas signaturas tienen un único sort, por lo cual tenemos la identidad como función entre sorts. Para completar el morfismo tenemos que dar un término formal de bool-sig' por cada operación de bool-sig:

```
ops⇝ : ∀ {ar s} → (f : bool-ops (ar , s)) → (map id ar) ⊩ s
```

Para las operaciones constantes t y f definimos la aplicación formal de las operaciones t' y f' al vector vacío:

```
ops⇝ t = t' |$| ⟨⟩
ops⇝ f = f' |$| ⟨⟩
```

Para la operación de disjunción or, tenemos que dar el término formal correspondiente a aplicar or' a los dos argumentos cuyos índices serán 0 y 1 implementados en el tipo Fin[3]:

```
ops⇝ or = or' |$| ⟨ # 0 , # 1 ⟩
```

Para la operación de negación neg, el término formal será la aplicación de la operación de equivalencia al primer argumento y a la constante false:

```
ops⇝ neg = equiv' |$| ⟨ # 0 , f' |$| ⟨⟩ ⟩
```

Y por último la traducción de la operación de conjunción and:

```
ops⇝ and = equiv' |$| ⟨ equiv' |$| ⟨ # 0 , # 1 ⟩
                        , or' |$| ⟨ # 0 , # 1 ⟩ ⟩
```

Álgebras reducto

Un morfismo entre signaturas $\mathfrak{m} : \Sigma_s \hookrightarrow \Sigma_t$ induce un funtor contravariante de Σ_t-algebras a Σ_s-algebras. Dada una Σ_t-algebra $\mathcal{A}$, denotamos con $\langle \mathcal{A} \rangle$ la correspondiente Σ_s-algebra, conocida como *álgebra reducto con respecto al morfismo* $\mathfrak{m}$. Describamos la construcción de este funtor entre álgebras: la interpretación de un Σ_s-sort s está dado por $\langle \mathcal{A} \rangle_s = \mathcal{A}_{(\mathfrak{m}\, s)}$, y la interpretación de una Σ_s-operación f será la interpretación del término formal $\mathfrak{m} f$ que definimos recursivamente así:

```
⟦_⟧⊩ : ∀ {ar s} → ar ⊩ s → A ⟦ ar ⟧ₛ* → ∥ A ⟦ s ⟧ₛ ∥
⟦ # n ⟧⊩     as = as ‼ n
⟦ f |$| ts ⟧⊩ as = A ⟦ f ⟧ₒ ⟨$⟩ ⟦ ts ⟧⊩* as
```

A los identificadores los interpretamos como proyecciones y la aplicación de una operación f a un vector de términos formales ts se define

3 En este texto los mostramos como números naturales.

recursivamente como como la interpretación de f en el álgebra aplicada a la interpetación de cada uno de los elementos de ts.

Mediante la interpretación de términos formales podemos dar la formalización de álgebra reducto.

```
_⟨~_~⟩ : ∀ {Σₛ Σₜ} → (m : Σₛ ⇝ Σₜ) → Algebra Σₜ → Algebra Σₛ
```

su definición tiene algunos detalles técnicos como el reindizado de vectores de los sorts de una signatura a la otra, por lo cual no la mostramos en este texto.

Y luego es directa la definición de la acción del funtor en homomorfismos:

```
_⟨~_~⟩ₕ : ∀ {Σₛ Σₜ} {A A′ : Algebra Σₜ} →
            (m : Σₛ ⇝ Σₜ) → (h : Homo A A′) →
                                Homo (m ⟨~ A ~⟩) (m ⟨~ A′ ~⟩)
m ⟨~ h ~⟩ₕ = record { ′_′ = ′ h ′ ∘ ⇝ₛ m
                    ; cond = ...
                    }
```

para dar la condición de homomorfismo también debemos lidiar con algunas sutilezas técnicas que no aportan nada a este texto.

Traducción de teorías

A partir de un morfismo $m : \Sigma_s \rightsquigarrow \Sigma_t$ podemos obtener una función que traduce términos de Σ_s en términos de Σ_t mediante el homomorfismo inicial de $(|\mathsf{T}|\ \Sigma_s)$ a $(m \langle\sim |\mathsf{T}|\ \Sigma_t \sim\rangle)$. Con una extensión apropiada para las variables, esta traducción puede aplicarse a una teoría E_s de la signatura Σ_s obteniendo una teoría $E_s\sim$ de Σ_t. Es esperable que si un álgebra A_t es modelo de $E_s\sim$, entonces el reducto $(m \langle\sim A_t \sim\rangle)$ sea modelo de E_s. Más aún, si E_t es más fuerte que la teoría traducida $E_s\sim$ y A_t es modelo de E_t, entonces también el álgebra reducto debería ser modelo de E_s. En Agda podríamos enunciar este resultado de la siguiente manera:

```
⊨T⇝ : ∀ Aₜ Eₜ Eₛ → Aₜ ⊨ₘ Eₜ → Eₜ ⊢T (⇝* Eₛ) → m ⟨~ Aₜ ~⟩ ⊨ₘ Eₛ
```

La dificultad para poder formalizar esta preservación de modelos via morfismos entre signaturas es que hay que lidiar con las variables de cada teoría. Con un morfismo $m : \Sigma_s \rightsquigarrow \Sigma_t$ se puede definir la traducción de términos con variables de $|\mathsf{T}|\ (\Sigma_s \langle\!|\ X_s\ |\!\rangle)$ a $|\mathsf{T}|\ (\Sigma_t \langle\!|\ X_t\ |\!\rangle)$ si también tenemos un renombre de variables `⇝v : {s : sorts Σₛ} → Xₛ s → Xₜ (m ⇝s s)`. Sin embargo no podremos probar la *propiedad de satisfacción*: si una Σ_t-álgebra satisface la traducción de una Σ_s-ecuación, luego su reducto satisface la Σ_s-ecuación original. La dificultad técnica es la imposibilidad de definir un Σ_t-entorno a partir de un Σ_s-entorno. La solución que podemos encontrar a este inconveniente es

restringir el conjunto de variables de la signatura destino tomando $X_t = \bigcup_{s \in \Sigma_s, t = m \hookrightarrow s} X_s$ (es decir, tomamos como conjunto de variables de destinto a las mismas que en el origen, para aquellos sorts que sean imagen del morfismo). Bajo esta restricción, hemos probado la propiedad de satisfacción y la definición para $\models_{T \rightsquigarrow}$, y puede encontrarse el desarrollo completo en el repositorio.

4.4 CORRECCIÓN DE UN COMPILADOR VIA ÁLGEBRA UNIVERSAL

El interés por formalizar álgebra universal en teoría de tipos (y en particular en Agda) surgió a partir del estudio sobre cómo definir compiladores correctos relacionado al trabajo presentado en el capítulo 3. Un primer trabajo sobre compiladores utilizando álgebras heterogéneas fue propuesto por Morris [63], que luego fue expandido a partir de los resultados sobre *initial algebra semantics* del grupo ADJ [32] con el trabajo [89]. Más recientemente, en el año 1998 Janssen [46] propuso un marco general para la traducción de lenguajes, donde la compilación es un caso particular.

La idea de este enfoque algebraico es considerar a los lenguajes fuente y destino de un compilador como las álgebras de términos de dos signaturas. Las semánticas de los lenguajes quedarán determinadas por los homomorfismos iniciales de las respectivas signaturas a las álgebras que se obtienen de interpretar las operaciones correspondientes. El compilador quedará definido por un morfismo entre las signaturas, y junto con un homomorfismo entre las semánticas de ambos lenguajes, la corrección está asegurada por la unicidad del homomorfismo inicial. En el diagrama de la figura 8 vemos de manera gráfica los componentes que intervienen en el enfoque algebraico: T_s y T_t son las álgebras de términos de los lenguajes; Sem_s y Sem_t sus respectivas semánticas vistas como álgebras; los homomorfismos iniciales $hsem_s$ y $hsem_t$ definen las funciones semánticas; enc y dec son dos posibles homomorfismos que relacionan las semánticas entre los lenguajes, necesarios para especificar la noción de corrección del compilador. Denotamos con $\langle \_ \rangle$ las álgebras y homomorfismos reductos, y $comp$ es el homomorfismo inicial desde el álgebra de teŕminos fuente al reducto del álgebra de términos destino. La corrección del compilador queda determinada por la conmutación del diagrama, que es a su vez asegurada por la inicialidad del álgebra de términos.

A continuación ilustramos el enfoque algebraico aplicándolo al ejemplo del compilador para expresiones aritméticas (cf. Sec. 3.1); para esto usaremos la librería de Álgebra Universal presentada en las secciones precedentes.

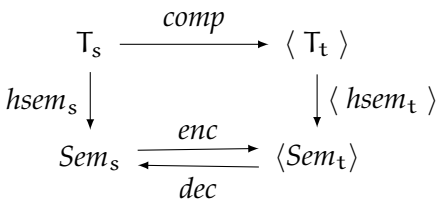


Figura 8: Esquema algebraico para la traducción de lenguajes

Los lenguajes

El lenguaje de alto nivel consta de tres construcciones: Constantes numéricas, suma e igualdad.

$$e ::= \quad n \mid e_1 \oplus e_2 \mid e_1 \stackrel{\circ}{=} e_2$$

En la signatura Σ_e tenemos dos sorts: uno para las expresiones naturales y otro para las booleanas. Una ventaja de permitir signaturas infinitarias es que podemos representar convenientemente las constantes naturales.

```
data Sorts_e : Set where
  N : Sorts_e
  B : Sorts_e

data Ops_e : List Sorts_e × Sorts_e → Set where
  val  : (n : ℕ) → Ops_e ([] , N)
  plus : Ops_e ( N ∷ [ N ] , N )
  eq   : Ops_e ( N ∷ [ N ] , B )

Σ_e : Signature
Σ_e = record { sorts = Sorts_e ; ops = Ops_e }
```

La sintaxis del lenguaje de expresiones está definida por el álgebra de términos de Σ_e:

```
Expr : (sorts Σ_e) → Set
Expr S = ‖ (|T| Σ_e) ⟦ S ⟧_s ‖
```

Por ejemplo el término correspondiente a la expresión de sumar las constantes 3 y 4 tiene tipo `Expr N`, y es `term plus ⟨ term (val 3) ⟨⟩ , term (val 4) ⟨⟩ ⟩`. Para facilitar la escritura de expresiones definimos smart-constructors, obteniendo una sintaxis sencilla:

```
|_| : ℕ → Expr N
| n | = term (val n) ⟨⟩

_⊕_ : Expr N → Expr N → Expr N
```

```
e₁ ⊕ e₂ = term plus ⟨⟨ e₁ , e₂ ⟩⟩

_:=_ : Expr N → Expr N → Expr B
e₁ := e₂ = term eq ⟨⟨ e₁ , e₂ ⟩⟩
```

de este modo la expresión anterior se escribe `| 3 | ⊕ | 4 |`.

El destino del compilador es un lenguaje para una máquina basada en stack, como vimos en 3.1.

$$c ::= \mathtt{push}\, v \mid \mathtt{add} \mid \mathtt{eq} \mid c_1, c_2$$

La signatura Σ_m describe la sintaxis de este lenguaje con un único sort. También tendremos infinita cantidad de operaciones para representar la operación push, una por cada número natural.

```
data Sorts_m : Set where C : Sorts_m

data Ops_m : List Sorts_m × Sorts_m → Set where
  push_m : (n : ℕ) → Ops_m ([] , C)
  add_m  : Ops_m ([] , C)
  equ_m  : Ops_m ([] , C)
  seq_m  : Ops_m (C : [ C ] , C)

Σ_m : Signature
Σ_m = record { sorts = Sorts_m ; ops = Ops_m }
```

De la misma manera que en el lenguaje de alto nivel, la sintaxis está definida por el álgebra de términos y podemos dar smart-constructors para facilitar la notación.

```
Code : Set
Code = ‖ (|T| Σ_m) ⟦ C ⟧_s ‖

push : ℕ → Code
push n = term (push_m n) ⟨⟩

add : Code
add = term add_m ⟨⟩

equ : Code
equ = term equ_m ⟨⟩

_⊙_ : Code → Code → Code
c₁ ⊙ c₂ = term seq_m ⟨⟨ c₁ , c₂ ⟩⟩
```

el término correspondiente a poner en la pila los valores 3 y 4, y sumarlos, es `push 3 ⊙ push 4 ⊙ add`.

Semántica

Las semánticas de los lenguajes quedarán determinadas por álgebras de las respectivas signaturas junto con el homomorfismo inicial. Para definir un álgebra debemos dar la interpretación de los sorts y de las operaciones.

En el lenguaje de alto nivel tenemos un sort para las expresiones naturales y otro para las booleanas. Las interpretaciones de cada uno serán el setoide trivial (donde la relación de igualdad es la proposicional) de los números naturales y el de los valores booleanos.

```
⟦_⟧e : sorts Σe → Setoid
⟦ N ⟧e = setoid ℕ
⟦ B ⟧e = setoid Bool
```

Para interpretar las operaciones debemos dar las funciones entre los setoides correspondientes según la aridad. Esto implica también la prueba de congruencia, que no mostraremos aquí:

```
iopse : ∀ {ar s} → (f : ops Σe (ar , s)) → ∥ ⟦_⟧e ∗ ar ∥ → ∥ ⟦ s ⟧e ∥
iopse (val n) ⟨⟩ = n
iopse plus ⟨⟨ n1 , n2 ⟩⟩ = n1 + n2
iopse eq ⟨⟨ n1 , n2 ⟩⟩ = n1 == n2

iconge : ∀ {ar s vs vs'} → (f : ops Σe (ar , s) ) → vs ≋ vs' →
                                        (iopse f vs) ≈ (iopse f vs')

ie : ∀ {ar s} → ops Σe (ar , s) → ⟦_⟧e ∗ ar ⟶ ⟦ s ⟧e
ie f = record { _⟨$⟩_ = iopse f ; cong = iconge f }
```

Con la interpretación de los sorts y las operaciones, tenemos la definición del álgebra semántica para el lenguaje de expresiones, y junto con ésta el homomorfismo inicial correspondiente:

```
Seme : Algebra Σe
Seme = record {_⟦_⟧s = ⟦_⟧e ; _⟦_⟧o = ie}


hsem : Homo (∣T∣ Σe) Seme
hsem = ∣h∣ Seme
```

a partir del homomorfismo podemos extraer la función semántica:

```
⟦_⟧ : ∀ {S} → Expr S → ∥ ⟦ S ⟧e ∥
⟦_⟧ {S} e = ′ hsem ′ S ⟨$⟩ e
```

y entonces podemos calcular por ejemplo ⟦ ∣ 3 ∣ ⊕ ∣ 4 ∣ ⟧ que será igual al natural 7.

Para definir la semántica del lenguaje de máquina, representamos la pila con una lista donde cada elemento puede ser un número natural o un booleano, y la interpretación del único sort de la signatura Σ_m corresponde al setoide compuesto por funciones que toman una pila y devuelven otra o un error en caso que la configuración no sea la esperada por la operación a ejecutar. La relación de igualdad de este setoide es la extensional.

```
Stack : Set
Stack = List (ℕ ⊎ Bool)

⟦_⟧m : sorts Σm → Setoid
⟦ C ⟧m = Stack ⟶-setoid Maybe Stack
```

La interpretación de las operaciones definen el álgebra para la semántica del lenguaje de máquina: por cada operación se define una función que toma una pila y obtiene otra en caso que no haya error. Como antes, no mostramos la prueba de congruencia:

```
iopsm : ∀ {ar s} → ops Σm (ar , s) → ‖ ⟦_⟧m ∗ ar ‖ → ‖ ⟦ s ⟧m ‖
iopsm (pushm n) ⟨⟩ = λ s → just (inj₁ n : s)
iopsm addm ⟨⟩ = λ { (inj₁ n₁ : inj₁ n₂ : s) → just (inj₁ (n₁ + n₂) : s)
                  ; _ → nothing }
iopsm equm ⟨⟩ = λ { (inj₁ n₁ : inj₁ n₂ : s) → just (inj₂ (n₁ == n₂) : s)
                  ; _ → nothing }
iopsm seqm ⟨⟨ f₁ , f₂ ⟩⟩ = λ s → f₁ s >>= f₂
```

El álgebra Exec describe la semántica del lenguaje de máquina, y tenemos el homomorfismo inicial a partir del cual podemos extraer la función semántica:

```
im : ∀ {ar s} → ops Σm (ar , s) → ⟦_⟧m ∗ ar ⟶ ⟦ s ⟧m
im f = record { _⟨$⟩_ = iopsm f ; cong = icongm f }

Exec : Algebra Σm
Exec = record {_⟦_⟧s = ⟦_⟧m ; _⟦_⟧o = im}

hexec : Homo (|T| Σm) Exec
hexec = |h| Exec

⟪|_|⟫ : Code → Stack → Maybe Stack
⟪| c |⟫ = ′ hexec ′ C ⟨$⟩ c
```

la aplicación (⟪| push 3 ⊙ push 4 ⊙ add |⟫ []) obtendrá como resultado just (inj₁ 7 : []).

Traducción

Mediante la definición de signaturas, álgebras y con la obtención de los homomorfismos iniciales tenemos la sintaxis de los lenguajes con sus respectivas semánticas. Para obtener un compilador debemos dar un morfismo entre las signaturas, mediante el cual podremos ver a las álgebras del lenguaje de máquina como álgebras de la signatura Σ_e.

Recordemos que un morfismo entre signaturas consta de una traducción de los sorts y una función que asigna a cada operación de la signatura de origen un término formal de la signatura destino. En nuestro caso ambos sorts de Σ_e serán traducidos al único sort de Σ_m, y la traducción de operaciones describe el compilador: las constantes naturales se traducirán a la instrucción que pone ese valor en el tope de la pila; la suma y la igualdad a la secuencia de poner en la pila los valores de los operandos junto con la instrucción respectiva.

```
s⇝ : sorts Σe → sorts Σm
s⇝ _ = C


op⇝ : ∀ {ar s} → ops Σe (ar , s) → lmap s⇝ ar ⊩ s⇝ s
op⇝ (val n) = (pushm n) |$| ⟨⟩
op⇝ plus = seqm |$| ⟨⟨ # 1) , seqm |$| ⟨⟨ # 0 , addm |$| ⟨⟩ ⟩⟩ ⟩⟩
op⇝ eq = seqm |$| ⟨⟨ # 1) , seqm |$| ⟨⟨ # 0 , equm |$| ⟨⟩ ⟩⟩ ⟩⟩

e⇝m : Σe ⇝ Σm
e⇝m = record { ⇝s = s⇝ ; ⇝o = op⇝ }
```

A partir del morfismo e⇝m podemos transformar cualquier Σ_m-álgebra en una Σ_e-álgebra, como el álgebra de términos del lenguaje de máquina y el álgebra semántica del mismo. A su vez también transformamos los Σ_m-homomorfismos en Σ_e-homomorfismos:

```
Tme : Algebra Σe
Tme = e⇝m ⟨~ (|T| Σm)~⟩

Exece : Algebra Σe
Exece = e⇝m ⟨~ Exec ~⟩

hexece : Homo Tme Exece
hexece = e⇝m ⟨~ hexec ~⟩h
```

y el homomorfismo que determina el compilador se obtiene por inicialidad del álgebra de términos de Σ_e hacia Tm$_e$, a partir del cual podemos extraer la función de compilación de expresiones a instrucciones:

```
hcomp : Homo (|T| Σₘ) Tmₑ
hcomp = |h| Tmₑ

compₑ : ∀ {S} → Expr S → Code
compₑ {S} e = ' hcomp ' S ⟨$⟩ e
```

la aplicación comp_e (| 3 | ⊕ | 4 |) obtendrá como resultado el término push 3 ⊙ push 4 ⊙ add.

Corrección

El único ingrediente que falta para completar el diagrama de la figura 8 es un homomorfismo entre las semánticas de ambos lenguajes. Éste codificará la noción de corrección del compilador: si tenemos un valor natural o booleano v correspondiente a la semántica de una expresión, la función correspondiente en la semántica del lenguaje de máquina será la que dada una pila s le agrega en el tope el valor v. Lo definimos en Agda, para lo cual debemos dar las correspondientes pruebas de congruencia y condición de homomorfismo, que no mostramos aquí:

```
enc : Semₑ ⇝ Execₑ
enc N = record { _⟨$⟩_ = λ n s → just (inj₁ n : s) ; cong = ...}
enc B = record { _⟨$⟩_ = λ b s → just (inj₂ b : s) ; cong = ...}

encH : Homo Semₑ Execₑ
encH = record { '_' = enc ; cond = ... }
```

La conmutación del diagrama está asegurada por la igualdad de los homomorfismos (hexecₑ ∘ₕ hcomp) y (encH ∘ₕ hsem) que es válida porque el homomorfismo entre el álgebra de términos y cualquier otra álgebra es único.

```
eqH : (hexecₑ ∘ₕ hcomp) ≈ₕ (encH ∘ₕ hsem)
eqH = terminit Execₑ (hexecₑ ∘ₕ hcomp) (encH ∘ₕ hsem)
```

La igualdad de dos homomorfismos asegura que si aplicamos cada uno al mismo elemento se obtendrá el mismo resultado y a partir de eqH extraemos exactamente este predicado, que es la noción de corrección que deseamos al definir el compilador:

```
correctN : (e : Expr N) → (s : Stack) →
           ⟨| compₑ e |⟩ s ≡ just (inj₁ ⟦ e ⟧ : s)
correctN = eqH N

correctB : (e : Expr B) → (s : Stack) →
           ⟨| compₑ e |⟩ s ≡ just (inj₂ ⟦ e ⟧ : s)
correctB = eqH B
```

De esta manera hemos definido un compilador correcto siguiendo el esquema propuesto por Morris, luego extendido por Thatcher y generalizado por Jannsen, utilizando nuestra librería de álgebra universal, como un ejemplo de aplicación de los conceptos formalizados.

4.5 RESUMEN Y TRABAJOS RELACIONADOS

En este capítulo hemos desarrollado una librería en Agda con los principales conceptos del Álgebra Universal heterogénea, incluyendo la prueba de los tres teoremas del isomorfismo y la inicialidad del álgebra de términos. Para definir las operaciones de una signatura hemos utilizado familias indexadas en los sorts, y junto con la implementación de una librería para vectores heterogéneos, dan muchas ventajas al definir una cantidad de conceptos como ecuaciones, el conjunto de axiomas de una teoría ecuacional o los términos formales en la traducción de signaturas. Formalizamos también un sistema deductivo para la lógica de cuasi-identidades (ecuaciones con condiciones), probando su corrección y completitud. Sobre el final del trabajo, presentamos una novedosa representación para morfismos entre signaturas y sus asociadas álgebras reducto, mostrando que bajo algunas restricciones el funtor que convierte álgebras de la signatura destino en la signatura fuente, preserva modelos de una teoría. Hemos ilustrado los conceptos con algunos ejemplos de uso, mostrando de qué manera uno puede utilizar esta librería, en particular desarrollamos el compilador de expresiones aritméticas presentado en la primera sección del capítulo anterior, dentro de un marco algebraico, como fue presentado por distintos investigadores.

Trabajos relacionados. Existen muy pocos trabajos disponibles sobre formalización de Álgebra Universal Heterogénea en teoría de tipos. Según el sondeo que realizamos en nuestra investigación, a partir del trabajo de Capretta [16] quien formalizó Álgebra Universal Heterogénea en Coq y la extensión a lógica ecuacional en su tesis, los trabajos cercanos más actuales son el de Kahl [47] formalizando alegorías y el desarrollo de jerarquías algebraicas por Spitters [87]. En el caso de Capretta, solo se consideran signaturas finitas y su trabajo no incluye la formalización de morfismos entre signaturas, mientras que Spitters y sus colegas solo desarrollaron definiciones preliminares del Álgebra Universal, necesarios para los objetivos que persiguen en su trabajo.

5

HACIA UNA FORMALIZACIÓN DE LA INDEPENDENCIA DE LA HIPÓTESIS DEL CONTINUO

A fines del siglo XIX Georg Cantor introdujo la teoría de conjuntos como fundamento teórico para razonar sobre el concepto de *infinitud* en matemática. El infinito había sido hasta el momento motivo de controversia y en los desarrollos matemáticos se podía concebir que algún elemento pueda crecer indefinidamente, o que una medida se acerque a otra tanto como uno desee, pero no que un objeto sea en sí mismo infinito. La teoría de Cantor propone el concepto primitivo de *conjunto*, mediante el cual cualquier objeto matemático puede ser representado.

Para razonar sobre el tamaño de los conjuntos infinitos se introducen los *números cardinales*. Dos conjuntos tienen el mismo cardinal si puede darse una biyección entre ellos (llamándolos *equipotentes*), y a los conjuntos biyectivos con los naturales se les llama *contables*. Bajo estas definiciones puede verse que el conjunto de los enteros es contable, como así también el de los racionales. Sin embargo no sucede lo mismo con el conjunto de los números reales, también conocido como el continuo: no existe biyección entre $\mathbb{N}$ y $\mathbb{R}$. Dado este hallazgo surge la pregunta de si algún otro conjunto tendrá cardinal mayor al de los naturales y menor al de los reales. Cantor responde que no a dicha pregunta enunciando la siguiente conjetura:

> **Hipótesis del Continuo (CH)**. Todo subconjunto incontable de $\mathbb{R}$ es equipotente a $\mathbb{R}$.

Decidir la validez de esta hipótesis se convirtió en uno de los problemas más importantes a comienzos del siglo XX, siendo incluso el primero de la lista de 23 problemas de Hilbert [42].

Dentro de una teoría formal como la axiomatización de *ZF* una sentencia lógica puede ser válida (se puede demostrar a partir de los axiomas), inválida (su negación puede ser probada a partir de los axiomas) o independiente (no existe una prueba de su validez ni de su negación). El primer gran avance sobre la hipótesis del continuo lo dio Kurt Gödel en el año 1940 [27]: no puede refutarse CH a partir de los axiomas de ZF. Ésto dio lugar a pensar que estaría pronto a resolverse el problema: la hipótesis del continuo podría ser válida. Sin embargo, 23 años después, Paul Cohen demuestra que su validez tampoco puede ser probada, concluyendo que la hipótesis del continuo es independiente de la axiomatización de Zermelo y Fraenkel.

Si bien la independencia de CH es uno de los avances matemáticos más importantes en la teoría de conjuntos, no existe hasta el momento una formalización completa en un asistente de pruebas. En este capítulo presentamos el último trabajo que emprendimos dentro de este doctorado: los primeros pasos hacia una formalización de CH, llegando a definir la extensión genérica de un modelo y probando la satisfacción de todos los axiomas de ZF en ella, menos el esquema de reemplazo. La primera tarea consistió en realizar un sondeo en las mecanizaciones existentes sobre teoría de conjuntos, llegando a la conclusión de que el trabajo realizado por Lawrence Paulson es el mejor punto de partida. Paulson implementó en Isabelle [68] desde los conceptos básicos en teoría de conjuntos [67] hasta el desarrollo necesario para probar la consistencia relativa de CH respecto de ZF [71].

En la primera sección daremos una breve introducción a Isabelle y la teoría de conjuntos ZF. En la segunda sección presentamos la noción de *consistencia relativa* de modelos en ZF y los conceptos necesarios para desarrollar la técnica de *forcing*, a la vez que señalamos cómo aprovechar definiciones y resultados del trabajo de Paulson para formalizarlos. En la tercera sección mostramos el diseño general de nuestra formalización de forcing junto con las definiciones y resultados preliminares. A continuación de esto presentamos la definición de extensión genérica y la prueba de validez de los axiomas de ZF menos el esquema de reemplazo. Finalmente comentamos el trabajo restante necesario para alcanzar nuestro objetivo final y los trabajos relacionados que se desprenden de ello.

5.1 ISABELLE/ZF

Isabelle[93] es un asistente de pruebas genérico en el cual se pueden implementar distintos formalismos como la lógica de alto orden (HOL por sus siglas en inglés), la lógica de primer orden (FOL) y la teoría de conjuntos (ZF). Consiste de un meta lenguaje basado en una teoría de tipos simple sobre el cual se definen teorías axiomáticas. Su flexibilidad y generalidad lo convierten en una poderosa herramienta capaz de soportar en simultáneo una gran cantidad de lógicas, permitiendo extender una teoría sobre otra, como es el caso de la teoría ZF construida sobre FOL.

El núcleo de Isabelle define una meta-lógica (un fragmento de la lógica de alto orden intuicionista), donde las reglas de inferencia son de la forma:

$$[\![\varphi_1; \ldots; \varphi_n]\!] \Longrightarrow \varphi. \tag{5}$$

cada φ_i es una fórmula o proposición. En cada lógica objeto se definen las fórmulas atómicas, los axiomas y reglas de inferencia que

permiten construir derivaciones para justificar nuevas proposiciones. Además del conectivo de implicación, la meta-lógica define un cuantificador universal $\bigwedge$ y un operador de igualdad definicional $\equiv$. Los términos del meta-lenguaje son los del cálculo lambda tipado, donde tenemos identificadores, abstracción y aplicación.

En el desarrollo de una prueba en Isabelle hay un estado de prueba, o *proof state*, que consiste de una regla de la forma de (5) donde φ es el objetivo final, y cada uno de los φ_i los sub-objetivos. Las *tácticas* son funciones que transforman proof states. La táctica más simple consiste en la aplicación de una regla de inferencia, donde la conclusión unifica con el sub-objetivo actual de la prueba, para lo cual Isabelle realiza unificación de alto orden. También se pueden desarrollar estrategias de prueba en las cuales se combinan tácticas para obtener resultados de manera automática. El *simplificador* es una de ellas, el cual aplica reglas de reescritura para operadores reflexivo/transitivos; otras estrategias más complejas son los métodos *auto* o *blast*. El modo procedural para realizar pruebas, descripto previamente, puede resultar anti-intuitivo ya que uno debe ir aplicando reglas desde el objetivo final descomponiéndolo en otros, y así sucesivamente, observando el proof state en cada paso. En general las demostraciones en papel y lápiz se realizan probando primero resultados parciales que luego justifican el objetivo mayor y todo el proceso queda explicitado en el mismo texto. Isabelle cuenta con una herramienta para escribir pruebas de esta manera declarativa, llamada *Isar* [94] y constituye el estándar actual, en parte gracias a la legibilidad del proof-script que se genera. Cada paso es justificado mediante axiomas o teoremas probados previamente, y las partes más burocráticas son realizadas mediante herramientas automáticas.

La teoría ZF

Isabelle/ZF implementa la teoría de conjuntos de Zermelo-Fraenkel como una extensión de la teoría FOL (lógica de primer orden). Daremos un breve repaso sobre las principales características de la implementación, una descripción detallada puede encontrarse en [73].

La teoría FOL está definida sobre la lógica de primer orden intuicionista (IFOL) agregando el axioma de doble negación:

$$(\neg P \Longrightarrow P) \Longrightarrow P$$

El resto de los axiomas están incluidos en IFOL, como por ejemplo la introducción de la conjunción y del para todo:

$$[\![P; Q]\!] \Longrightarrow P \wedge Q$$
$$(\bigwedge x.P(x)) \Longrightarrow (\forall x.P(x))$$

El tipo de las fórmulas de primer orden es "o" y los términos sobre los cuales predican estas fórmulas pueden ser de cualquier ti-

po, gracias al polimorfismo. Así, por ejemplo la igualdad tiene tipo $eq :: "['a,'a] \Longrightarrow o"$. En Isabelle/ZF los tipos de datos están definidos axiomáticamente, describiendo las constantes y operaciones que construyen los elementos que lo componen, y luego axiomas que los relacionan. Esto contrasta con las habituales definiciones de tipos inductivos que tenemos en otros lenguajes, y por ello por ejemplo no podemos dar funciones recursivas sobre el tipo o.

Los dos principales sistemas axiomáticos para la teoría de conjuntos son Bernays-Gödel (BG) y Zermelo-Fraenkel, que difieren en el tratamiento de las *clases*: colecciones demasiado grandes para ser conjuntos. El sistema BG tiene una cantidad finita de axiomas y por ello es preferido para implementar en asistentes de prueba [12, 77] a pesar de que ZF es el más estudiado por la comunidad de teoría de conjuntos. En Isabelle no hay inconvenientes para definir esquemas de axiomas y entonces es posible desarrollar el sistema axiomático de Zermelo-Fraenkel.

Isabelle/ZF formaliza una gran cantidad de conceptos y resultados, desde las nociones básicas hasta desarrollos más complejos como las pruebas de cardinales [76] o la consistencia relativa del axioma de elección [72].

El lenguaje de ZF no contiene ninguna constante ni operación, sólo dos relaciones: la igualdad y la pertenencia, por lo que los únicos términos que pueden construirse son las variables. Así por ejemplo si se expresa una propiedad $\varphi(x)$ aplicada al conjunto vacío $\varnothing$, no tenemos un término que nos permita escribir simplemente $\varphi(\varnothing)$, sino que debe expresarse:

$$\exists v(\neg\exists u(u \in v) \wedge \varphi(v)).$$

Sólo con este lenguaje primitivo alcanza para desarrollar toda la teoría de conjuntos. En la práctica, sin embargo, se vuelve insostenible tener que escribir en este nivel sintáctico, y como es usual en matemática, se utiliza azúcar sintáctico que permite referir de manera más sencilla a las expresiones más usadas. Por ello Isabelle/ZF declara constantes para los conjuntos primitivos como el vacío, facilitando la escritura e implementando precisamente la teoría de Zermelo y Fraenkel.

Los términos en Isabelle/ZF tienen tipo i, mientras que las fórmulas tienen tipo o. Así por ejemplo 0 ó *Inf* son términos constantes de tipo i que refieren al conjunto vacío e infinito respectivamente, y $Pair$ es una función con tipo $[i, i] \Rightarrow i$ que construye pares ordenados. Las facilidades sintácticas de Isabelle permiten escribir de una manera muy similar a la matemática informal, por ejemplo $A \cup B$ refiere a la unión de conjuntos, $\langle a, b\rangle$ al par ordenado y puede definirse un conjunto por extensión como $\{0, 1, 2, 3\}$.

Axiomas de ZF

Los axiomas de Zermelo y Fraenkel forman una lista infinita de sentencias de primer orden. Una introducción detallada explicando los fundamentos que hay detrás de cada uno puede encontrarse en [86]. A continuación presentamos la lista de axiomas de ZF como están presentados en https://www.dropbox.com/s/sqn88rgwbax44vm/apunte_st.pdf?dl=1. La parte finita de la axiomatización está compuesta por los siguientes:

- Extensión (*dos conjuntos son iguales si y solo si tienen los mismos elementos*).

  $\forall x, y(x = y \leftrightarrow \forall z(z \in x \leftrightarrow z \in y))$.

- Fundación (*la relación $\in$ es bien fundada en el universo de conjuntos*).

  $\forall x(x \neq \varnothing \rightarrow \exists y \in x : \forall z \in x(z \notin y))$.

- Pares (*dados x e y, existe un conjunto que contiene a ambos como elementos*).

  $\forall x, y \exists c(x \in c \wedge y \in c)$.

- Unión (*dado un conjunto x, hay un conjunto que incluye a la unión de todos los conjuntos que pertenecen a x*).

  $\forall x \exists u : \forall y \in x \forall z \in y(z \in u)$.

- Partes (*También llamado potencia: dado un conjunto x, existe un conjunto que contiene a todos los subconjuntos de x*).

  $\forall x \exists p \forall y(\forall z \in y(z \in x) \rightarrow y \in p)$.

- Infinito (*existe un conjunto inductivo*).

  $\exists x(\varnothing \in x \wedge \forall y(y \in x \rightarrow y \cup \{y\} \in x))$.

y su parte infinita está definida por dos esquemas de axiomas:

- Separación (*también llamado Comprensión: para cada conjunto x y cada propiedad, existe el conjunto que sólo contiene a los elementos de x que la satisfacen*). Para cada $\varphi(x)$ una fórmula de primer orden con una variable libre,

  $\forall x \exists y \forall a(a \in y \leftrightarrow a \in x \wedge \varphi(a))$.

- Reemplazo (*para cada función, la imagen de un conjunto por dicha función está incluida en un conjunto*). Para cada $\psi(x, y)$ fórmula de primer orden con dos variables libres,

  $\forall x(\forall y \in x \exists! z : \psi(y, z)) \implies \forall y \in x \exists z(\psi(y, z) \wedge z \in r)$.

En algunas presentaciones de ZF no se incluye el esquema de separación ya que puede obtenerse a partir del esquema de reemplazo, es decir no es independiente de la lista de axiomas.

Como mencionamos previamente, Isabelle/ZF define algunas construcciones primitivas para facilitar el desarrollo de la teoría. La unión, partes y el conjunto infinito son definidos de esta manera, como así también un constructor primitivo de términos para definir conjuntos de acuerdo al esquema de reemplazo.

```
Pow :: "i ⇒ i"
Inf :: "i"
Union :: "i ⇒ i"
PrimReplace :: "[i, [i, i] ⇒ o] ⇒ i"
```

A partir de un conjunto y una relación binaria (implementada como una función de dos elementos de tipo `i` en uno de tipo `o`), la primitiva de reemplazo construye un conjunto. Otras definiciones incluyen a `Replace(A,P)`, similar a PrimReplace pero `P` debe ser una relación funcional, y puede escribirse con la sintaxis `{y. x∈A, Q} == Replace(A, λx y. Q)`. Con `Collect(A,P)` se construye un conjunto utilizando el principio de separación, que se obtiene a partir de reemplazo mediante `Collect(A,P) == {y . x∈A, x=y ∧P(x)}`, y puede escribirse también con la sintaxis `{x∈A.P}`.

Utilizando éstas y otras definiciones, la axiomatización de Isabelle/ZF está dada de la siguiente manera:

```
axiomatization
where
  extension:   "A = B ⟷ A ⊆ B ∧ B ⊆ A" and
  Union_iff:   "A ∈ ⋃(C) ⟷ (∃B∈C. A∈B)" and
  Pow_iff:     "A ∈ Pow(B) ⟷ A ⊆ B" and
  infinity:    "0 ∈ Inf ∧ (∀y∈Inf. succ(y) ∈ Inf)" and
  foundation:  "A = 0 ∨ (∃x∈A. ∀y∈x. y∉A)" and
  replacement: "(∀x∈A. ∀y z. P(x,y) ∧ P(x,z) ⟶ y = z)
             ⟹ b ∈ PrimReplace(A,P) ⟷ (∃x∈A. P(x,b))"
```

5.2 MODELOS DE ZF Y LA TÉCNICA DE FORCING

Daremos un breve repaso de algunos de los conceptos más importantes involucrados en las pruebas de consistencia relativa de modelos de ZF y de la maquinaria necesaria para extender modelos mediante la técnica de *forcing*. Una excelente introducción sin detenerse demasiado en los detalles tecnicos puede encontrarse en [92] o en [17]. Para mayor rigurosidad en el desarrollo de las pruebas recomendamos el libro de Teoría de Conjuntos de Kunen [49].

Consistencia relativa

En lógica, una teoría es *consistente* si a partir de los axiomas no puede probarse una contradicción ($0 = 1$). Un *modelo* de una teoría es un objeto matemático en el cual los términos del lenguaje son interpretados y los axiomas resultan válidos, escribimos $M \models \varphi$ para indicar que M satisface la sentencia φ, es decir que M es modelo de φ[1]. Una teoría es consistente si y solo si existe un modelo.

Considerando la teoría ZF como fundamento de toda la matemática surge la pregunta de cómo construir un modelo, es decir un objeto matemático en el cual se interprete la relación de pertenencia y los axiomas valgan dentro del mismo. Gödel probó con su Segundo Teorema de Incompletitud que no puede probarse la existencia de un modelo de ZF. Más formalmente, si asumimos que las pruebas matemáticas pueden ser codificadas como teoremas en ZF y que ésta no tiene contradicciones, luego no podemos probar que exista un conjunto M y una relación binaria E tal que $\langle M, E\rangle$ satisface los axiomas de ZF. Por lo tanto es imposible determinar si la teoría de conjuntos es consistente.

Lo que sí podemos obtener es, dado un axioma A, una prueba de *consistencia relativa*, que consiste en asumir que existe un modelo de ZF, digamos $\langle M, E\rangle$, y construir otro modelo $\langle M', E'\rangle$ para $\mathsf{ZF} + A$. Nos concentraremos en una clase especial de modelos:

1. Un conjunto M (de conjuntos) es *transitivo* si para todo $x \in M$ e $y \in x$, tenemos que $y \in M$ (i.e., todo elemento de M es subconjunto de M).

2. $\langle M, E\rangle$ es un *modelo transitivo* si M es transitivo y E es la relación de pertenencia $\in$ restringida a M. Es *contable* si M es equipotente a un subconjunto de los naturales; diremos que el modelo M es un *ctm* (por las siglas de Modelo Transitivo Contable en inglés).

Usualmente se refiere al modelo transitivo solo nombrando el conjunto M, ya que la relación es fija.

Las pruebas de satisfacción relativas a un modelo constituyen una herramienta fundamental en el desarrollo de la independencia de la hipótesis del continuo. Si consideramos un modelo M y un submodelo $N \subseteq M$, es posible que un axioma valga en N y no en M. Lo ilustremos con un ejemplo: sean $M := \{a, b, c, \{a, b\}, \{a, b, c\}\}$, $N := \{a, b, \{a, b, c\}\}$ y

$$\varphi(x, y, z) := \forall w.\, (w \in z \leftrightarrow w = x \vee w = y).$$

1 para el caso de fórmulas con variables libres $\{x_0, ..., x_n\}$ la noción de satisfacción requiere de un entorno para asignar valores $\{a_0, ..., a_n\}$ a las mismas, y lo escribiremos $M, [a_0, ..., a_n] \models \varphi(x_0, ..., x_n)$.

Tenemos

$$M \not\models \varphi(a, b, \{a, b, c\}) \quad \text{pero} \quad N \models \varphi(a, b, \{a, b, c\}).$$

Para decidir la validez relativa de una fórmula en un conjunto sólo importan los elementos que pertenecen al mismo. En el ejemplo, como c no pertenece a N, luego los únicos elementos de N que pertencen a $\{a, b, c\}$ son a y b, por lo que el axioma de pares se satisface; no sucede lo mismo en M. Diremos que φ vale para $a, b, \{a, b, c\}$ *relativo* a N. Podemos definir formalmente qué significa que la fórmula φ valga relativamente a un conjunto de esta manera simplemente restringiendo la cuantificación a elementos en N:

$$\varphi^N(x, y, z) := \forall w.\ w \in N \to (w \in z \leftrightarrow w = x \vee w = y)$$

φ^N es la *relativización* de φ a N. Podemos luego generalizar esta definición a la clase de todos los conjuntos que satisfacen un predicado C:

$$\varphi^C(x, y, z) := \forall w.\ C(w) \to (w \in z \leftrightarrow w = x \vee w = y)$$

Es fácil ver que si M y N son transitivos, φ^M si y solo si φ^N, para $x, y, z \in N$. Decimos entonces que φ es absoluta con respecto a N y M. Los conceptos de relativización y absolutez son fundamentales en la tarea de decidir la validez de los axiomas en un submodelo de ZF a partir de su validez en el modelo correspondiente. Un resultado importante es que una gran cantidad de fórmulas de primer orden son absolutas en modelos transitivos contables, es decir que si son válidas dentro del modelo también lo son para todo el universo de conjuntos. En general, si una propiedad ψ habla solamente de los elementos de algún x que pertenece a un ctm M, ψ es absoluta ya que por transitividad todos los elementos de x deben pertenecer también a M.

En la prueba de Gödel para mostrar la consistencia relativa del axioma de elección y de la hipótesis del continuo con ZF se construye un submodelo siguiendo este lineamiento: Asumiendo que M es un ctm de ZF, se muestra que M contiene un submodelo minimal L^M que satisface ZF + CH. Los conjuntos en L^M son llamados *construibles* y existe una fórmula de primer orden L tal que $L^M = \{x \in M : M \models L(x)\}$. Luego prueba $L^M \models ZF + CH$ a partir de la hipótesis $M \models ZF$.

Para formalizar los resultados de Gödel en Isabelle, Paulson realizó un gran trabajo dando las definiciones necesarias para poder expresar la validez relativa de fórmulas en un modelo, siendo de mucha utilidad para la formalización de la consistencia relativa de la negación de CH, objetivo de nuestro trabajo. Por ejemplo el axioma de pares relativo a una clase es implementado de esta manera:

```
definition
  upair :: "[i=>o,i,i,i] => o" where
  "upair(M,a,b,z) == a ∈ z & b ∈ z & (∀x[M]. x∈z ⟶
                                      x = a | x = b)"
definition
  upair_ax :: "(i=>o) => o" where
  "upair_ax(M) == ∀x[M]. ∀y[M]. ∃z[M]. upair(M,x,y,z)"
```

donde $\forall$`x[C].`φ es la cuantificación universal relativa al predicado C. Muchos resultados de absolutez para modelos transitivos también fueron implementados en el trabajo de Paulson, así para el caso del axioma de pares tenemos:

```
lemma (in M_trivial) upair_abs [simp]:
     "M(z) ==> upair(M,a,b,z) ⟷ z={a,b}"
apply (simp add: upair_def)
apply (blast intro: transM)
done
```

y otra gran cantidad de fórmulas están probadas absolutas en su desarrollo.

Forcing

Para probar la consistencia relativa de la negación de la hipótesis del continuo con respecto a ZF, Paul Cohen desarrolló la técnica conocida como *forcing* que permite extender modelos transitivos contables preservando la satisfacción de los axiomas. Dado un ctm M de ZF y un conjunto G, se construye un nuevo ctm M[G] que incluye a M y contiene a G, probando que bajo algunas hipótesis M[G] satisface ZF. Este ctm es llamado la *extensión genérica* de M, y para construirlo necesitamos algunas definiciones preliminares:

Definición 5. Una *noción de forcing* $\langle \mathbb{P}, \leqslant, \mathbb{1} \rangle$ es un preorden con elemento máximo.

Definición 6. Un subconjunto $G \subseteq \mathbb{P}$ es *creciente* si para todo $x \in G$, si $x \leqslant p$ entonces $p \in G$.

Definición 7. Dos elementos $p, q \in G$ son *compatibles* en G si tienen cota inferior en G.

Definición 8. Un subconjunto $D \subseteq \mathbb{P}$ es *denso* si todo elemento en $\mathbb{P}$ tiene cota inferior en D.

Definición 9. Un subconjunto $F \subseteq \mathbb{P}$ es un *filtro* si es creciente y todo par de elementos es compatible en F.

Definición 10. Dada una secuencia $\mathcal{D}$ de subconjuntos densos en $\mathbb{P}$, un filtro G es *$\mathcal{D}$-genérico* si intersecta todos los conjuntos densos de la secuencia.

El lema de Rasiowa-Sikorski (RSL) asegura que para todo preorden P y una familia contable $\{\mathcal{D}_n : n \in \mathbb{N}\}$ de subconjuntos densos en P, existe un filtro $\mathcal{D}$-genérico. La extensión genérica $M[G]$ se construye a partir de un ctm M que contendrá una noción de forcing $\langle \mathbb{P}, \leqslant, \mathbb{1} \rangle$ y como es M es contable podemos obtener filtros M-genéricos por RSL.

Estamos entonces en condiciones de dar la definición de la extensión genérica de un modelo de ZF. Dados un ctm M de ZF, una noción de forcing en M y un filtro M-genérico G, la extensión genérica de M está dada por:

$$M[G] := \{val(G, \tau) : \tau \in M\}.$$

donde la función *val* está definida recursivamente con la siguiente ecuación:

$$val(G, \tau) := \{val(G, \sigma) : \exists p \in \mathbb{P}.\, (\langle \sigma, p \rangle \in \tau \wedge p \in G)\}. \tag{6}$$

Esta definición recursiva tiene sentido gracias al principio de recursión en relaciones bien fundadas [49, p. 48]. Una función recursiva $F\colon A \to A$ se define eligiendo una relación $R \subseteq A \times A$ bien fundada y un funcional $H\colon A \times (A \to A) \to A$, luego existe F tal que para toda $a \in A$, $F(a) = H(a, F \restriction (R^{-1}(a)))$. Para la función *val* puede utilizarse la siguiente relación:

$$x \; ed \; y \iff \exists p. \langle x, p \rangle \in y.$$

De la definición de $M[G]$ vemos que cada elemento $a \in M[G]$ se define a partir de un elemento en M, al que llamaremos su *nombre*. Para que $M[G]$ sea efectivamente una extensión de M, debe darse que $M \subseteq M[G]$ y esto sucede gracias a que para cada $x \in M$, $check(x) \in M$ y $val(G, check(x)) = x$, donde

$$check(x) := \{\langle check(y), \mathbb{1} \rangle : y \in x\} \tag{7}$$

El primer hito importante en el desarrollo de Cohen es mostrar que $M[G]$ es modelo de ZF. En general el trabajo necesario para probar la validez de un axioma en $M[G]$ consiste en encontrar un nombre adecuado en M. Los axiomas básicos de la teoría pueden probarse sin mayores dificultades a partir de las definiciones, pero los esquemas de separación y reemplazo, junto con el axioma de potencia requieren de los *Teoremas de Forcing*, que relacionan la satisfacción de una

fórmula φ en $M[G]$ con la satisfacción de otra fórmula `forces`(φ) en M. La definición de `forces` es recursiva en fórmulas de primer orden, por lo cual para poder implementarla en Isabelle/ZF es necesario contar con una internalización de las fórmulas dentro de la teoría ZF. Paulson realizó este trabajo definiendo el conjunto `formula` de tipo `i`, utilizando índices de Bruijn para las variables, y dando solo un par de conectivos básicos pudiendo luego definir el resto como combinación de éstos:

```
consts   formula :: i
datatype
  "formula" = Member ("x ∈ nat", "y ∈ nat")
            | Equal  ("x ∈ nat", "y ∈ nat")
            | Nand ("p ∈ formula", "q ∈ formula")
            | Forall ("p ∈ formula")
```

Esta simple y bonita definición esconde el trabajo necesario para poder definir conjuntos inductivos, que gracias a un paquete para Isabelle/ZF realizado por Paulson [69, 70] permite obtener el principio de inducción y la posibilidad de definir funciones recursivas a partir de una declaración como la de `formula`. La función de satisfacción expresa cuándo una fórmula de primer orden vale en un conjunto, para lo cual es necesario contar con un entorno que asigne valores a las variables libres:

```
consts   satisfies :: "[i,i]=>i"
primrec
  .....

abbreviation
  sats :: "[i,i,i] => o" where
  "sats(A,p,env) == satisfies(A,p)‘env = 1"
```

La función `satisfies` (cuya definición obviamos mostrar) obtiene a partir de un conjunto y una fórmula, una función de entornos en fórmulas; la abreviación `sats` devuelve True en el tipo `o` si la fórmula se satisface, y constituye el predicado que necesitamos para expresar la satisfacción de fórmulas en conjuntos. Es importante notar la dificultad de trabajar en un contexto no tipado como es la teoría ZF. La declaración de tipos de `satisfies` nos indica que la función toma dos conjuntos y devuelve otro, y cuando el segundo de esos conjuntos es un elemento de `formula`, devuelve una función de entornos (que son implementados como listas, otro conjunto definido mediante el paquete de datatypes) en el conjunto 2 o `bool`. Cuando uno define funciones recursivas en Isabelle/ZF en general es esperable definir un lema de tipado, que gracias a los métodos automáticos se prueba de manera inmediata:

```
lemma "p ∈ formula ==> satisfies(A,p) ∈ list(A) -> bool"
by (induct set: formula) simp_all
```

Como es esperable, la satisfacción de una fórmula internalizada se corresponde con la validez de la correspondiente en el tipo `o`, para ello en Isabelle/ZF se prueban los lemas que atestiguan esta correspondencia:

```
lemma sats_Forall_iff [simp]:
  "env ∈ list(A)
   ==> sats(A, Forall(p), env) ⟷ (∀x∈A. sats(A, p, Cons(x,env)))"
by simp
```

y de la misma manera con el resto de los conectivos. Cargar este tipo de lemas en el simplificador hace que en las pruebas la equivalencia entre la satisfacción de una fórmula y su semántica en el tipo `o` sea transparente.

Con fórmulas internalizadas podemos entonces definir la función `forces` que relaciona satisfacción en la extensión genérica con satisfacción en el ctm correspondiente. La notación que utiliza Kunen [49, 50] para $\texttt{forces}(\varphi)$ es

$$p \Vdash^*_{\mathbb{P},\leqslant,\mathbb{1}} \varphi(x_0,\ldots,x_n).$$

donde $\mathbb{P},\leqslant,\mathbb{1}$ es la noción de forcing y p es un elemento de $\mathbb{P}$. En nuestra notación será

$$M, [\mathbb{P},\leqslant,\mathbb{1},p,a_0,\ldots,a_n] \models \texttt{forces}(\varphi(x_0,\ldots,x_n)).$$

Los siguientes lemas caracterizan la relación de forcing:

Lema 11 (Definición de Forcing)**.** *Sea M un ctm de ZF, $\langle\mathbb{P},\leqslant,\mathbb{1}\rangle$ una noción de forcing en M, $p \in \mathbb{P}$, y $\varphi(x_0,\ldots,x_n)$ una fórmula en el lenguaje de la teoría de conjuntos con variables libres $\{x_0,\ldots,x_n\}$. Son equivalentes, para todo $\tau_0,\ldots,\tau_n \in M$:*

1. $M, [\mathbb{P},\leqslant,\mathbb{1},p,\tau_0,\ldots,\tau_n] \models \texttt{forces}(\varphi(x_4,\ldots,x_{n+4}))$.
2. *Para todo filtro M-genérico G tal que $p \in G$,*
   $M[G], [val(G,\tau_0),\ldots,val(G,\tau_n)] \models \varphi(x_0,\ldots,x_n)$.

Lema 12 (Truth Lemma)**.** *Asumiendo las mismas hipótesis del Lema 11. Son equivalentes, para todo $\tau_0,\ldots,\tau_n \in M$, y G filtro M-genérico:*

1. $M[G], [val(G,\tau_0),\ldots,val(G,\tau_n)] \models \varphi(x_0,\ldots,x_n)$.
2. *Existe $p \in G$ tal que $M, [\mathbb{P},\leqslant,\mathbb{1},p,\tau_0,\ldots,\tau_n] \models \texttt{forces}(\varphi(x_4,\ldots,x_{n+4}))$.*

5.3 DISEÑO DE LA FORMALIZACIÓN Y DEFINICIONES PRELIMINARES

Habiendo hecho un repaso por las principales características del asistente de pruebas en el que realizaremos nuestra formalización y de los principales conceptos involucrados en forcing, mostraremos en esta sección una vista general del diseño que elegimos para la implementación y de algunas definiciones y resultados básicos.

En los textos matemáticos es usual fijar un contexto donde se definen parámetros y se asumen propiedades necesarias para desarrollar los teoremas y resultados. Isabelle permite definir estos contextos mediante *locales* [5]. Un locale puede extenderse agregando nuevas definiciones e hipótesis para formar uno nuevo, como también puede incluirse dentro de uno ya definido probando que sus hipótesis

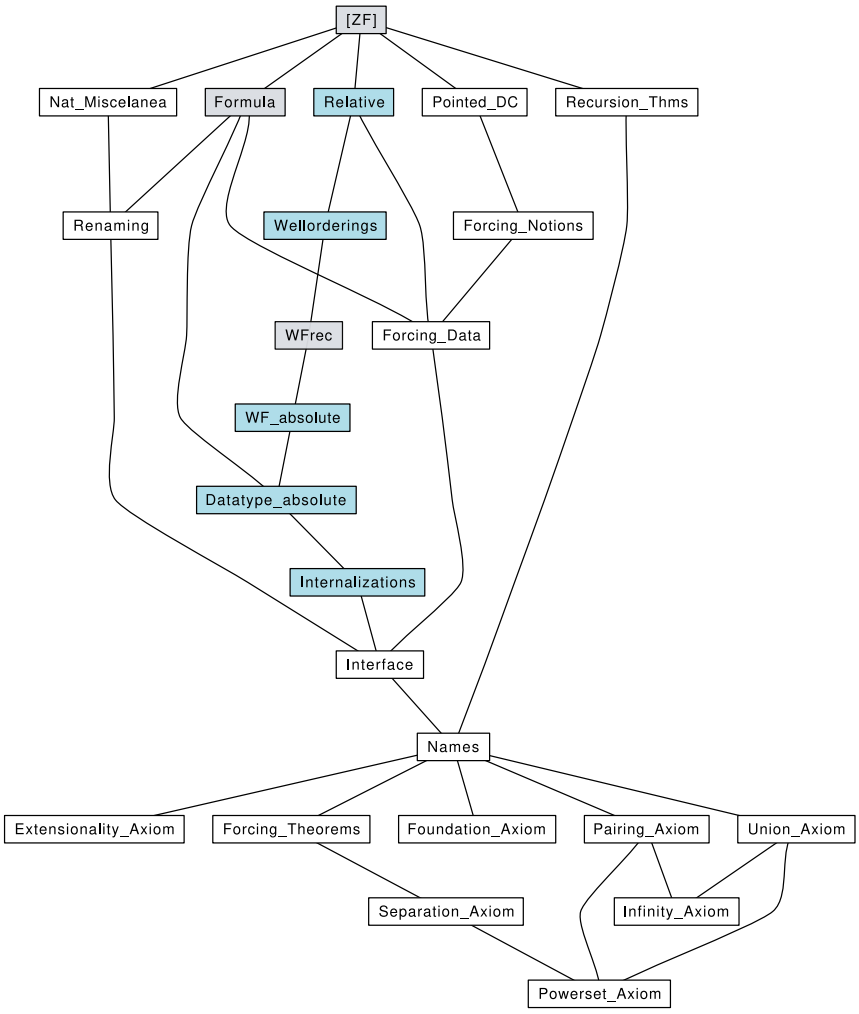


Figura 9: Gráfico de dependencias

valen en el nuevo contexto (y se dice que es un *sublocale* del definido previamente). En nuestro desarrollo utilizamos locales para organizar la formalización asumiendo los parámetros e hipótesis necesarias, y también utilizaremos la definición de sublocales para poder utilizar una gran cantidad de resultados realizados por Paulson que también están organizados de esta manera.

En el gráfico 9 mostramos las dependencias de nuestra formalización. Los recuadros en blanco son los módulos (que en Isabelle toman el nombre de teorías[2]) que hemos desarrollado en nuestro trabajo, mientras que los demás forman parte de las librerías de Isabelle desarrolladas por Paulson, mostrando con fondo gris aquellos en los que realizamos modificaciones.

El primer objetivo en la formalización de forcing es la definición de extensión genérica. Como vimos en la sección anterior, este concepto involucra las nociones de forcing, la definición de un modelo M de ZF, la definición de filtros genéricos y de nombres en M. Las teorías (o módulos) `Forcing_Notions`, `Forcing_Data` y `Names` contienen las definiciones de estos conceptos, y las describiremos en detalle en las siguientes subsecciones. Las teorías `Renaming` y `Recursion_Thms` contienen mejoras sobre el desarrollo de algunos conceptos implementados en Isabelle/ZF: el primero permite realizar renombre de variables en las fórmulas internalizadas y el segundo contiene una batería de teoremas sobre recursión bien fundada. `Nat_Miscellanea` contiene pequeños resultados sobre números naturales que son de utilidad para el desarrollo de renombres, y `Pointed_DC` la prueba de una versión del principio de Elección Dependiente, necesario para la prueba del lema de Rasiowa-Sikorski. Una característica importante de nuestro desarrollo es el aprovechamiento del gran trabajo realizado por Paulson para la prueba de independencia del axioma de elección, donde implementó resultados de absolutez. Para ello definimos en `Interface` las pruebas necesarias para relacionar el contexto de nuestro trabajo con los locales de su desarrollo y tener acceso a los resultados. Por último en las teorías `Extension_Axiom`, ..., `Powerset_Axiom` formalizamos las pruebas de validez de cada axioma en la extensión genérica, revistiendo particular importancia el esquema de separación incluido en `Separation_Axiom` donde es necesario utilizar la maquinaria de forcing, y que en palabras de Kunen constituye el resultado más difícil para probar que $M[G]$ es modelo de ZF: «[...] in verifying that $M[G]$ is a model for set theory, the hardest axiom to verify is [Separation].» [49] .

2 No hay que confundir una teoría como Isabelle/ZF que corresponde al concepto matemático en el sentido de un conjunto de axiomas y reglas para desarrollar resultados, con los archivos de Isabelle que se nombran como teorías, concepto similar al de módulos en otros lenguajes.

Forcing notions

En la teoría `Forcing_Notions` definimos las nociones de forcing, filtros genéricos y resultados sobre los conceptos involucrados finalizando con el lema de Rasiowa-Sikorski que asegura la existencia de filtros genéricos.

El siguiente locale caracteriza una noción de forcing:

```
locale forcing_notion =
  fixes P leq one
  assumes one_in_P:          "one ∈ P"
      and leq_preord:        "preorder_on(P,leq)"
      and one_max:           "∀p∈P. ⟨p,one⟩∈leq"
```

expresando que ⟨`P`,`leq`⟩ es un preorden con elemento máximo `one`.

Las definiciones 6,7,8 y 9 son implementadas de manera sencilla:

```
definition
  increasing :: "i⇒o" where
  "increasing(F) == ∀x∈F. ∀p∈P. ⟨x,p⟩∈leq ⟶ p∈F"

definition compat_in :: "i⇒i⇒i⇒i⇒o" where
  "compat_in(A,r,p,q) == ∃d∈A . ⟨d,p⟩∈r ∧ ⟨d,q⟩∈r"

definition
  dense :: "i⇒o" where
  "dense(D) == ∀p∈P. ∃d∈D . ⟨d,p⟩∈leq"

definition
  filter :: "i⇒o" where
  "filter(G) == G⊆P ∧ increasing(G)
                  ∧ (∀p∈G. ∀q∈G. compat_in(G,leq,p,q))"
```

Y para definir filtro genérico necesitamos contar con una familia de subconjuntos densos en la noción de forcing. En el locale `countable_generic` especificamos el contexto necesario:

```
locale countable_generic = forcing_notion +
  fixes 𝒟
  assumes countable_subs_of_P:  "𝒟 ∈ nat→Pow(P)"
  and     seq_of_denses:        "∀n ∈ nat. dense(𝒟‘n)"
```

y luego podemos expresar que un filtro que interseca a cada subconjunto es genérico:

```
definition
  D_generic :: "i⇒o" where
  "D_generic(G) == filter(G) ∧ (∀n∈nat.(𝒟`n)∩G≠0)"
```

El lema más importante incluido en `Forcing_Notions` es Rasiowa-Sikorski enunciado de esta manera:

```
theorem rasiowa_sikorski:
  "p∈P ⟹ ∃G. p∈G ∧ D_generic(G)"
```

El argumento para su prueba es sencillo: Fijando un elemento $p_0 = p \in \mathbb{P}$, se puede elegir p_{n+1} tal que $p_n \geqslant p_{n+1} \in \mathcal{D}_n$, ya que $\mathcal{D}_n$ es denso en $\mathbb{P}$. Luego el filtro generado por $\{p_n : n \in \omega\}$ interseca cada conjunto en la secuencia $\mathcal{D}_n$.

Forcing data

Definir la extensión genérica de un modelo requiere de la caracterización de ctms de ZF. Para ello utilizamos las definiciones de los axiomas relativos a una clase, que se encuentran en la teoría `Relative` del desarrollo de Paulson. Para el caso de los esquemas debemos expresar que para toda fórmula de primer orden, se satisface la instancia correspondiente de separación y reemplazo. Como explicamos anteriormente, en nuestro desarrollo utilizamos fórmulas internalizadas, por lo que la safisfacción estará definida mediante la función `sats`. Permitiremos a lo sumo una o dos variables libres para la fórmula que se instancia en cada esquema respectivamente, pudiendo expresar luego cualquier cantidad de parámetros mediante una n-upla construida gracias al axioma de pares.

En la librería de Paulson los axiomas relativos están definidos en el contexto de clases, por lo cual utilizamos el operador `##` que a partir de un conjunto A obtiene el predicado $\lambda x.x \in A$. El siguiente locale define el contexto de un conjunto M que satisface todos los axiomas de ZF:

```
locale M_ZF =
  fixes M
  assumes
        upair_ax:       "upair_ax(##M)"
    and Union_ax:       "Union_ax(##M)"
    and power_ax:       "power_ax(##M)"
    and extensionality: "extensionality(##M)"
    and foundation_ax:  "foundation_ax(##M)"
    and infinity_ax:    "infinity_ax(##M)"
    and separation_ax:
```

```
      "⟦ φ ∈ formula ; arity(φ)=1 ∨ arity(φ)=2 ⟧ ⟹
       (∀a∈M. separation(##M,λx. sats(M,φ,[x,a])))"
  and replacement_ax:
    "⟦ φ ∈ formula ; arity(φ)=2 arity(φ)=succ(2) ⟧ ⟹
(∀a∈M. strong_replacement(##M,λx y. sats(M,φ,[x,y,a])))"
```

Las condiciones sobre la aridad de la fórmula φ en los esquemas de separación y reemplazo expresan que φ podrá tener uno o ningún parámetro. Si necesitamos instanciar una fórmula con mayor cantidad de parámetros se pueden utilizar lemas de *tupling* definidos en el módulo `Interface` cuyas pruebas utilizan el axioma de pares.

En el locale `forcing_data` definimos que M es un ctm de ZF y contiene una noción de forcing:

```
locale forcing_data = forcing_notion + M_ZF +
  fixes enum
  assumes M_countable:      "enum∈bij(nat,M)"
      and P_in_M:           "P ∈ M"
      and leq_in_M:         "leq ∈ M"
      and trans_M:          "Transset(M)"
```

Dentro de este locale podemos dar la definición de filtro M-genérico y la prueba de que existe: al ser M contable, tenemos una biyección con los naturales a partir de la cual podemos construir una secuencia $\mathcal{D}_\setminus$ de subconjuntos densos en P: dado un natural n, si al aplicar la biyección obtenemos un subconjunto denso de P, luego D aplicado a n será ese subconjunto, en caso contrario será todo P, que es denso por definición. Utilizando el lema de Rasiowa-Sikorski se completa la prueba de la existencia de un filtro M-genérico.

```
lemma generic_filter_existence:
  "p∈P ⟹ ∃G. p∈G ∧ M_generic(G)"
```

5.4 SATISFACCIÓN DE ZF EN LA EXTENSIÓN GENÉRICA

Antes de entrar en los detalles de las definiciones de nombres y extensión genérica, mostramos brevemente la interfaz entre el contexto de nuestra formalización y el del desarrollo de Paulson que se encuentra en la sesión de Isabelle `ZF-constructible`. En la teoría `Relative` se encuentra una gran cantidad de resultados de absolutez de fórmulas de primer orden para conjuntos transitivos. La organización de dicha implementación está realizada mediante dos *locales*: `M_trivial` y `M_basic` (que extiende al primero). En estos locales se

asume una clase M transitiva y una cantidad de hipótesis sobre la validez de algunos axiomas en M, en particular instancias de separación y todo el esquema de reemplazo para cualquier predicado en `P::"i⇒o"`. Todas estas asunciones pueden probarse en el contexto del locale `forcing_data` (utilizando la función `##_` para liftear un conjunto en una clase) menos esta versión de reemplazo para clases, ya que hay más elementos de tipo `"i⇒o"` que los predicados definibles como fórmulas. Para resolverlo, modificamos levemente los archivos originales en `ZF-constructible` eliminando algunos resultados que dependen de dicha hipótesis ya que no son necesarios en nuestra formalización.

En la teoría `Interface` probamos que `forcing_data` es sublocale de `M_trivial` y `M_basic`.

sublocale *forcing_data* ⊆ *M_trivial* "*##M*"

sublocale *forcing_data* ⊆ *M_basic* "*##M*"

La mayor parte del trabajo consiste en probar instancias de separación para predicados en `"i⇒o"` a partir de la hipótesis

```
separation_ax:   "⟦ φ ∈ formula ; arity(φ)=1 ∨ arity(φ)=2 ⟧
                ⟹ (∀a∈M. separation(##M,λx. sats(M,φ,[x,a])))"
```

en la que asumimos el esquema de separación utilizando la internalización de fórmulas. Por ejemplo, una de las instancias que se encuentran en `M_basic` es la que define el productor cartesiano entre dos conjuntos.

```
lemma (in forcing_data) cartprod_sep_intf :
  assumes
          "A∈M"
          and
          "B∈M"
   shows
          "separation(##M,λz. ∃x∈M. x∈A ∧ (∃y∈M. y∈B
          ∧ pair(##M,x,y,z)))"
```

Estas pruebas son bastante burocráticas y requieren de la internalización en `formula` de cada predicado, que en el caso del producto cartesiano es:

```
definition
  cartprod_sep_fm :: "i" where
  "cartprod_sep_fm ==
```

```
    Exists(And(Member(0,2),
           Exists(And(Member(0,2),pair_fm(1,0,4)))))"
```

Otro detalle técnico que tenemos que sortear para poder expresar cualquier instancia de los esquemas es el uso de n-uplas para definir fórmulas con cualquier cantidad de parámetros mediante una que tiene uno solo. El truco consiste en utilizar el axioma de pares y expresar que ese único parámetro contiene a todos los demás. En el caso del producto cartesiano tenemos dos y debemos probar:

```
lemma (in forcing_data) tupling_sep_2p :
    "(∀v∈M. separation(##M,λx. (∀A∈M. ∀B∈M. pair(##M,A,B,v)
                                ⟶ Q(x,A,B))))
    ⟷
     (∀A∈M. ∀B∈M. separation(##M,λx. Q(x,A,B)))"
```

Nombres y la extensión genérica

En la teoría `Names` hemos implementado las nociones y resultados involucrados en la extensión genérica, siempre dentro del contexto `forcing_data`. La definición de M[G] requiere de la implementación de la función recursiva *val* (ver 6), y en Isabelle/ZF tenemos disponible el principio de recursión sobre relaciones bien fundadas a través del operador `wfrec` [69]. Definimos el funcional Hν que implementa la función *val* de la siguiente manera:

```
definition
  Hv :: "i⇒i⇒i⇒i" where
  "Hv(G,x,f) == { f`y .. y∈ domain(x), ∃p∈P. <y,p> ∈ x ∧ p ∈
G }"
```

La relación bien fundada a partir de la cual realizamos el principio de recursión es *ed* y la definimos relativa a un conjunto. Luego podemos definir *val* restringiendo la relación a la clausura transitiva del parámetro sobre el cual se está calculando la función:

```
definition
  edrel :: "i ⇒ i" where
  "edrel(A) == {<x,y> ∈ A*A . x ∈ domain(y)}"

definition
  val :: "i⇒i⇒i" where
  "val(G,τ) == wfrec(edrel(eclose({τ})), τ ,Hv(G))"
```

Esta definición satisface la ecuación 6 y para probarlo necesitamos agregar a las herramientas sobre recursión disponibles en Isabelle/ZF una propiedad que permite calcular una función recursiva sobre un elemento a restringiendo la relación a cualquier conjunto que incluya a a y a todos sus predecesores; la misma está implementada en `Recursion_Thms`:

```
lemma wfrec_restr :
  assumes "relation(r)" "wf(r)"
  shows  "a∈A ⟹ (r^+)-‘‘{a} ⊆ A ⟹
          wfrec(r,a,H) = wfrec(r∩A×A,a,H)"
```

Luego podemos probar el lema de definición de *val*, y como consecuencia inmediata que la misma es monótona:

```
lemma def_val:
 "val(G,x) = {val(G,t) .. t∈domain(x) ,
                  ∃p∈P. <t,p>∈x ∧ p∈G }"

lemma val_mono: "x⊆y ⟹ val(G,x) ⊆ val(G,y)"
```

Teniendo definido *val*, podemos dar la definición de extensión genérica

```
definition
  GenExt :: "i⇒i"     ("M[_]")
  where "GenExt(G)== {val(G,τ). τ ∈ M}"
```

Para probar que efectivamente M[G] es extensión de M necesitamos dar nombres para los elementos que allí se encuentran. Esto lo hacemos mediante la función *check* (en la literatura usualmente denotado por $\check{x}$) y luego probamos que $val(G, check(x)) = x$. Para definirla utilizamos recursión bien fundada sobre la relación $\in$ y probamos que satisface la ecuación 7.

```
definition
  Hcheck :: "[i,i] ⇒ i" where
  "Hcheck(z,f)  == { <f‘y,one> . y ∈ z}"

definition
  check :: "i ⇒ i" where
  "check(x) == transrec(x , Hcheck)"
```

```
lemma def_check : "check(y) = { <check(w),one> . w ∈ y}"
```

Una prueba inductiva asegura que *check* nombra a los elementos de M, asumiendo que el máximo de $\mathbb{P}$ está en G:

```
lemma valcheck : "one ∈ G ⟹  val(G,check(y))  = y"
```

Para poder concluir que $M \subseteq M[G]$ es necesario probar $check(x) \in M$ para todo $x \in M$. Dicha prueba requiere de la internalización de la función recursiva como elemento de `formula` para lo cual Paulson realizó un gran trabajo en el locale `M_eclose`. Parte de nuestro próximo trabajo es realizar la interfaz entre el contexto de nuestra formalización y dicho locale; y hasta tenerlo realizado asumimos esta hipótesis, junto con una instancia de reemplazo necesaria para probar $G \in M[G]$ en el locale `M_extra_assms`:

```
locale M_extra_assms = forcing_data +
  assumes
       check_in_M : "⋀x. x ∈ M ⟹ check(x) ∈ M"
    and repl_check_pair :
            "strong_replacement(##M,λp y. y =<check(p),p>)"
```

M[G] *es modelo de* ZF

Probar que la extensión genérica es modelo de Zermelo-Fraenkel constituye una parte importante en el desarrollo de forcing. En el presente trabajo hemos concluído la prueba de todos los axiomas básicos, incluido el de potencia, más el esquema de separación; quedando pendiente la satisfacción del esquema de reemplazo.

Extensionalidad, Fundación, Pares, Unión e Infinito pueden probarse de manera directa y no requieren de los teoremas de forcing:

```
lemma extensionality_in_MG : "extensionality(##(M[G]))"

lemma foundation_in_MG : "foundation_ax(##(M[G]))"

lemma pairing_in_MG :
  assumes "M_generic(G)"
  shows "upair_ax(##M[G])"

lemma union_in_MG : assumes "filter(G)"
  shows "Union_ax(##M[G])"

lemma infinty_in_MG : "infinity_ax(##M[G])"
```

Para probar el axioma de infinito es necesario asumir las hipótesis extra del locale `M_extra_assms` y que `G` es genérico.

El resto de los axiomas requiere de la maquinaria de forcing. Gracias al uso de locales, podemos enunciar y asumir los teoremas de forcing sin dar por el momento las pruebas ni la definición de `forces`. En el locale `forcing_thms` tendremos el contexto necesario para poder usar forcing:

```
locale forcing_thms = forcing_data +
  fixes forces :: "i ⇒ i"
  assumes definition_of_forces: "p∈P ⟹ φ∈formula ⟹ env∈list(M)
           ⟹ sats(M,forces(φ), [P,leq,one,p] @ env) ⟷
           (∀G.(M_generic(G)∧ p∈G)⟶sats(M[G],φ,map(val(G),env)))"
      and  truth_lemma: "φ∈formula ⟹ env∈list(M) ⟹ M_generic(G)
       ⟹ (∃p∈G.(sats(M,forces(φ), [P,leq,one,p] @ env))) ⟷
                 sats(M[G],φ,map(val(G),env))"
...
```

implementando con precisión los enunciados de los lemas 11 y 12.

Dentro de este locale probamos la validez del axioma de partes y del esquema de separación en la extensión genérica. Los detalles técnicos están explicados con precisión en [35].

```
theorem power_in_MG :
  "power_ax(##(M[G]))"

theorem separation_in_MG:
  assumes
    "φ∈formula" and "arity(φ) = 1 ∨ arity(φ)=2"
  shows
    "(∀a∈(M[G]). separation(##M[G],λx. sats(M[G],φ,[x,a])))"
```

5.5 RESUMEN Y TRABAJO RESTANTE

El objetivo final del trabajo presentado en este capítulo es una completa formalización de forcing y la prueba de consistencia relativa de ¬CH en *ZF*. Pretendemos también que la formalización de los conceptos involucrados puedan servir para mecanizar otros resultados en los cuales la técnica de forcing es utilizada.

El mayor logro de lo realizado hasta el momento es haber podido definir las nociones de extensión genérica y nombres, junto con las pruebas de validez de los axiomas de *ZF*. Fue fundamental para esta implementación la utilización de locales para asumir hipótesis y prototipar la función `forces`; consideramos un acierto organizar así

nuestro diseño, pues incluso nos da la libertad de variar la implementación actual siempre y cuando respetemos la interfaz especificada. Esta modularización nos permitió basar nuestro desarrollo en el trabajo de Paulson. En ese sentido, todavía queda mucho trabajo por realizar, instanciando otros locales dentro de los cuales Paulson implementó los resultados necesarios para internalizar en el tipo `formula` funciones definidas recursivamente. Realizado eso, podremos probar que la función `check` es cerrada para M.

En el texto presentado aquí nos hemos concentrado en el contexto general de la formalización, y no en los detalles técnicos. El lector interesado puede encontar más información en los artículos que hemos publicado, como así también en el código fuente de la implementación.

6

PALABRAS FINALES

SOBRE EL TRABAJO REALIZADO

Hemos comenzado esta tesis haciendo un pequeño recorrido histórico por los desarrollos que han dado origen a la matemática moderna y por consiguiente a las ciencias de la computación. El interés por encontrar báses sólidas, que justifiquen acabadamente la matemática hasta ese momento, ha tenido consecuencias de magnitudes impensadas. El desarrollo de la teoría de conjuntos, la aparición de las paradojas y la crisis fundacional, las corrientes filosóficas que emergieron y la consolidación de un sistema formal preciso permitieron avances científicos que van mucho más allá de las ideas originarias. Con el avance de las ciencias de la computación, el estudio formal de la matemática encuentra nuevos horizontes: los programas pueden asistir en el desarrollo de las teorías y chequear la corrección de los resultados, dando origen a la mecanización o formalización de matemática. A su vez, la posibilidad de expresar y comprobar resultados matemáticos mediante programas, permite describir propiedades deseadas a través de especificaciones y asegurar su preservación en desarrollos tecnológicos.

Los lenguajes de programación basados en la teoría de tipos dependientes, mediante el principio de proposiciones como tipos, posibilitan escribir programas en los cuales se pueden chequear estáticamente propiedades especificadas en el tipado. De esta manera es posible tener programas *correctos por construcción*, es decir, en vez de verificar a posteriori la satisfacción de las propiedades, éstas se aseguran a través del sistema de tipos: si el programa escrito pasa el chequeo de tipos, entonces es correcto. El primer aporte de esta tesis fue el estudio de una metodología para desarrollar compiladores, utilizando como metalenguaje uno con tipos dependientes, que consiste en definir la sintaxis de los lenguajes fuente y destino mediante una familia de tipos indexada en la semántica. De esta manera el compilador podrá expresar en su tipado que el término que se obtiene en el lenguaje destinto tiene una semántica que se adecúa a la del término de origen. Referimos como *enfoque internalista* a este método, siguiendo la terminología que proponen Ko y Gibbons en [48], en contraposición con el *externalista*, en el cual debe realizarse una prueba de corrección luego de definir el compilador. Esta idea aplicada a un ejemplo sencillo como un lenguaje de expresiones aritméticas ya había sido presentada por Connor McBride a partir de un ejemplo de James McKinna [60]. Sin embargo su motivación era presentar la teoría de Ornaments [58]

que da un marco formal para programar, definiendo propiedades en los índices del tipado. En nuestro trabajo concentramos el esfuerzo en investigar hasta qué punto puede ser factible definir compiladores así, y para ello aplicamos el método a un compilador de un lenguaje imperativo simple a un lenguaje intermedio que manipula una pila, logrando la corrección por construcción. El siguiente paso fue estudiar la posibilidad de compilar hasta un lenguaje de máquina con saltos condicionales al estilo Assembly, y concluimos que es necesario que las semánticas de los lenguajes sean dirigidas por sintaxis para poder aplicar la metodología; sin embargo, se puede aplicar un enfoque mixto, logrando tener un compilador correcto por construcción desde el lenguaje fuente a uno intermedio, y luego definir de manera externalista la etapa de compilación al lenguaje de máquina.

El grupo ADJ [28] estudió la aplicación de la teoría de álgebras universales heterogéneas como herramienta formal en la definición de lenguajes de programación. La sintaxis de un lenguaje queda determinada por una signatura, y la semántica por un álgebra de ésta, denotada por el homomorfismo inicial. Dadas dos signaturas, puede definirse un *morfismo entre ellas*, que consiste en asignar para cada operación de la signatura fuente una regla para construir términos en la segunda, para lo cual también se debe definir una función entre los sorts. A partir de este morfismo entre signaturas, toda álgebra de la signatura destino puede verse como un álgebra de la signatura fuente, que se llamará *álgebra reducto*. Si los lenguajes de programación quedan definidos a partir de signaturas y sus semánticas mediante las álgebras de éstas, un morfismo entre signaturas determina un compilador. Jansen [46] presenta un esquema general para traducir lenguajes mediante esta mirada algebraica. Estudiar estas herramientas teóricas para definir compiladores nos llevó a la implementación en Agda de una librería de álgebras universales heterogéneas. En un análisis sobre las publicaciones disponibles, vimos que hay muy poco sobre formalizaciones de álgebra universal en teoría de tipos. Nuestra formalización incluye la definición de las principales nociones como signatura, álgebra, homomorfismos, subálgebras, congruencias, etc; también las pruebas de los teoremas de isomorfismo y la incialidad del álgebra de términos. Además implementamos un sistema deductivo para la lógica de cuasi-identidades (ecuaciones condicionales) probando su corrección y completitud; y presentamos la única formalización hasta el momento de los conceptos de morfismos entre signaturas (en inglés *derived signature morphism*) y álgebras reducto. Utilizar familias indexadas en los sorts como implementación de las operaciones de una signatura es un punto clave en nuestro trabajo, permitiendo grandes ventajas en el aprovechamiento del type-checker de Agda en muchas definiciones.

El último aporte de nuestra tesis está relacionada al estudio de las teorías fundacionales. La independencia de la hipótesis del continuo

es uno de los resultados matemáticos más importantes obtenidos en el siglo XX: en el formalismo de la teoría de conjuntos de Zermelo-Fraenkel no puede probarse ni refutarse la CH; es decir, puede asumirse o no asumirse sin afectar la consistencia de la teoría. Gödel probó la consistencia de CH con ZF; luego Paul Cohen probó que ¬CH también es consistente con ZF introduciendo la técnica de *forcing*, única manera conocida de extender modelos transitivos contables de ZF. La técnica de forcing no solo se aplica a la independencia de CH, sino que encuentra utilidad en otras áreas de la matemática, revistiendo de especial importancia. El abordaje de estos conceptos requiere un nivel de detalle formal muy preciso, por lo que la implementación en un asistente de pruebas es particularmente interesante. Lawrence Paulson implementó en Isabelle una gran cantidad de conceptos y resultados de la teoría de conjuntos, en particular la consistencia relativa del axioma de elección y de CH. A partir de su trabajo hemos formalizado parte de la maquinaria de forcing, incluyendo la noción de extensión genérica de un modelo y la prueba de validez relativa en ella de todos los axiomas de ZF menos el esquema de reemplazo. Habiendo alcanzado estos objetivos hemos dado los primeros pasos hacia la primera mecanización completa de la independencia de CH.

SOBRE EL TRABAJO POR HACER

De los objetivos alcanzados en esta tesis se desprenden varios trabajos posibles.

- La utilización de ornamentos para decorar tipos de datos que aseguren propiedades por construcción puede aplicarse a otros contextos más allá de los compiladores. Parte del trabajo futuro es explorar en qué otros problemas puede ser beneficioso internalizar propiedades en el tipado, como por ejemplo aplicar el enfoque en algoritmos de matching para expresiones regulares.
- La librería que hemos implementado en Agda sobre álgebras universales puede ser de utilidad para formalizar más resultados del área como el teorema de variedades de Birkhoff [1] . Otro paso natural del trabajo es extender la librería para contemplar álgebras de segundo orden como propone [25]. Los morfismos entre signaturas pueden generalizarse en la teoría de instituciones, y otro posible trabajo es continuar con su formalización. También consideramos la formalización de sistemas de reescritura.
- En cuanto a la formalización de forcing, aún queda mucho trabajo por realizar. Luego de completar la prueba de que la extensión genérica es modelo de *ZF*, debemos formalizar las pruebas de los teoremas fundamentales de forcing, y para ello dar

la definición de la función `forces`. Debemos definir el poset $Add(w, \kappa)$ y probar que cumple la condición de cadenas contables (se dice que es ccc). Luego probar que si un preorden con top es ccc entonces preserva cardinales. Y por último, si G es M-genérico para $Add(w, \aleph_2)$, entonces $M[G] \models ZFC + \neg CH$.

CONCLUSIÓN DE LA CONCLUSIÓN

El camino realizado en el doctorado que finaliza con esta tesis podría considerarse algo sinuoso ya que ha encontrado el rumbo durante su desarrollo. Comenzando con el estudio de la teoría de tipos y en particular de lenguajes con tipos dependientes para escribir programas con buenas propiedades aseguradas por construcción, ha devenido en la utilización de asistentes de prueba para mecanizar matemática. El desarrollo del formalismo en la matemática ha permitido grandes avances científicos permitiendo disponer de un lenguaje común para comunicar y chequear los resultados por la comunidad. La utilización de sistemas computacionales asistiendo el desarrollo matemático y comprobando su validez promete grandes posibilidades. Un escenario como el planteado en el proyecto QED, en el cual toda la matemática esté formalizada en un mismo sistema con un pequeño núcleo probado correcto, que sea fácil de abordar mediante una sintaxis simple, que disponga de una base de datos global en la cual buscar teoremas y realizar conexiones entre distintas teorías sea sencillo, entre otras, suena a utópico y está lejos de lograrse. Aún hoy los asistentes de prueba no son fáciles de utilizar y por consiguiente el grueso de la comunidad matemática es ajeno a ellos. La formación de científicos de la computación en los fundamentos de la matemática y en las bases teóricas de los asistentes de prueba puede contribuir a mejorar la perspectiva, para lo cual es fundamental la articulación con científicos de la matemática. En ese sentido, la última parte de este doctorado fue muy fructífera, conformando equipos de trabajo no solo en áreas específicas de computación sino también con matemáticos experimentados.

BIBLIOGRAFÍA

[1] Jiři Adámek, Evelyn Nelson y Jan Reiterman. «The Birkhoff Variety Theorem for continuous algebras». En: *algebra universalis* 20.3 (1985), págs. 328-350. ISSN: 1420-8911. DOI: 10.1007/BF01195142. URL: https://doi.org/10.1007/BF01195142.

[2] Kenneth Appel y Wolfgang Haken. *Every planar map is four colorable*. Vol. 98. Contemporary Mathematics. With the collaboration of J. Koch. American Mathematical Society, Providence, RI, 1989, págs. xvi+741. ISBN: 0-8218-5103-9. DOI: 10.1090/conm/098. URL: http://dx.doi.org/10.1090/conm/098.

[3] Jeremy Avigad y Daniel Velleman. «The Mechanization of Mathematics». En: 2018.

[4] Joan Bagaria. «Set Theory». En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta. Winter 2017. Metaphysics Research Lab, Stanford University, 2017.

[5] Clemens Ballarin. «Tutorial to locales and locale interpretation». En: *Contribuciones científicas en honor de Mirian Andrés Gómez*. Universidad de La Rioja. 2010, págs. 123-140.

[6] Gilles Barthe, Venanzio Capretta y Olivier Pons. «Setoids in type theory». En: *J. Funct. Program.* 13.2 (2003), págs. 261-293.

[7] Nick Benton, Andrew Kennedy y Carsten Varming. «Some Domain Theory and Denotational Semantics in Coq». English. En: *Theorem Proving in Higher Order Logics*. Ed. por Stefan Berghofer, Tobias Nipkow, Christian Urban y Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, págs. 115-130. ISBN: 978-3-642-03358-2.

[8] Garrett Birkhoff. «On the Structure of Abstract Algebras». En: *Mathematical Proceedings of the Cambridge Philosophical Society* 31.4 (1935), 433–454. DOI: 10.1017/S0305004100013463.

[9] Garrett Birkhoff y John D Lipson. «Heterogeneous algebras». En: *J. of Combinatorial Theory* 8.1 (1970), págs. 115-133.

[10] Errett Bishop. «Foundations of constructive analysis». En: (1967).

[11] Ana Bove, Peter Dybjer y Ulf Norell. «A Brief Overview of Agda - A Functional Language with Dependent Types.» En: *TPHOLs*. Ed. por Stefan Berghofer, Tobias Nipkow, Christian Urban y Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, págs. 73-78. ISBN: 978-3-642-03358-2. URL: http://dblp.uni-trier.de/db/conf/tphol/tphol2009.html#BoveDN09.

[12] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel y Lawrence Wos. «Set theory in first-order logic: Clauses for Gödel's axioms». En: *Journal of Automated Reasoning* 2.3 (1986), págs. 287-327. ISSN: 1573-0670. DOI: 10.1007/BF02328452. URL: https://doi.org/10.1007/BF02328452.

[13] N.G. de Bruijn. *AUTOMATH, a language for mathematics*. T.H. Department of Mathematics, Eindhoven University of Technology, 1968.

[14] R. M. Burstall. «Proving Properties of Programs by Structural Induction». En: *The Computer Journal* 12.1 (feb. de 1969), págs. 41-48. ISSN: 1460-2067. DOI: 10.1093/comjnl/12.1.41. URL: http://dx.doi.org/10.1093/comjnl/12.1.41.

[15] G. Cantor y P.E.B. Jourdain. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications, 1915. URL: https://books.google.es/books?id=Jw9SAQAAMAAJ.

[16] Venanzio Capretta. «Universal algebra in type theory». En: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 1999, págs. 131-148.

[17] Timothy Y. Chow. «A beginner's guide to forcing». En: *Communicating Mathematics: A Conference in Honor of Joseph A. Gallian's 65th Birthday, July 16-19, 2007, University of Minnesota, Duluth, Minnesota*. Contemporary mathematics. American Mathematical Society, 2009, págs. 25-40. ISBN: 9780821843451. URL: http://adsabs.harvard.edu/abs/2007arXiv0712.1320C.

[18] Alonzo Church. «A formulation of the simple theory of types». En: *The journal of symbolic logic* 5.2 (1940), págs. 56-68.

[19] Thierry Coquand. «Type Theory». En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University, 2018.

[20] Thierry Coquand y Gérard P. Huet. «The Calculus of Constructions». En: *Inf. Comput.* 76.2/3 (1988), págs. 95-120.

[21] Pierre-Évariste Dagand y Conor McBride. «Transporting functions across ornaments». En: *J. Funct. Program.* 24.2-3 (2014), págs. 316-383. DOI: 10.1017/S0956796814000069. URL: https://doi.org/10.1017/S0956796814000069.

[22] Edsger W. Dijkstra y Carel S. Scholten. *Predicate Calculus and Program Semantics*. New York, NY: Springer New York, 1990. ISBN: 978-1-4612-7924-2. DOI: 10.1007/978-1-4612-3228-5. URL: http://dx.doi.org/10.1007/978-1-4612-3228-5.

[23] Mirna Džamonja. «A new foundational crisis in mathematics, is it really happening?» En: *arXiv e-prints*, arXiv:1802.06221 (feb. de 2018), arXiv:1802.06221. arXiv: 1802.06221 [math.LO].

[24] José Ferreirós. «The crisis in the foundations of mathematics». En: *The Princeton companion to mathematics* (2008), págs. 142-156.

[25] Marcelo Fiore y Ola Mahmoud. «Second-order algebraic theories». En: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2010, págs. 368-380.

[26] Abraham A. Fraenkel. *Abstract Set Theory*. Second. Studies in Logic and Foundations of Mathematics. Amsterdam: North-Holland, 1961.

[27] Kurt Gödel. *The Consistency of the Continuum Hypothesis*. Annals of Mathematics Studies, no. 3. Princeton University Press, Princeton, N. J., 1940, pág. 66.

[28] Joseph A. Goguen. «Memories of ADJ». En: *Bulletin of the EATCS* 39 (1989), págs. 96-102.

[29] Joseph A. Goguen y Rod M. Burstall. «Institutions: Abstract Model Theory for Specification and Programming». En: *J. ACM* 39.1 (ene. de 1992), págs. 95-146. ISSN: 0004-5411. DOI: 10.1145/147508.147524. URL: http://doi.acm.org/10.1145/147508.147524.

[30] Joseph A Goguen y Kai Lin. «Specifying, Programming and Verifying with Equational Logic.» En: *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*. College Publications, 2005, págs. 1-38.

[31] Joseph A Goguen, James W Thatcher, Eric G Wagner y Jesse B Wright. «Abstract data types as initial algebras and the correctness of data representations». En: *Conference on Computer Graphics, Pattern Recognition, & Data Structure, UCLA*. IEEE Computer Society, 1975, págs. 89-93.

[32] Joseph A Goguen, James W Thatcher, Eric G Wagner y Jesse B Wright. «Initial algebra semantics and continuous algebras». En: *J. ACM* 24.1 (1977), págs. 68-95.

[33] Georges Gonthier y col. «A Machine-Checked Proof of the Odd Order Theorem». En: *ITP*. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, págs. 163-179.

[34] Emmanuel Gunther, Alejandro Gadea y Miguel Pagano. «Formalization of Universal Algebra in Agda.» En: *Electr. Notes Theor. Comput. Sci.* 338 (2018), págs. 147-166. URL: http://dblp.uni-trier.de/db/journals/entcs/entcs338.html#GuntherGP18.

[35] Emmanuel Gunther, Miguel Pagano y Pedro Sánchez Terraf. «Mechanization of Separation in Generic Extensions». En: *arXiv e-prints*, arXiv:1901.03313 (ene. de 2019), arXiv:1901.03313. arXiv: 1901.03313 [cs.LO].

[36] Emmanuel Gunther, Miguel Pagano y Pedro Sánchez Terraf. «First steps towards a formalization of Forcing.» En: *CoRR* abs/1807.05174 (2018). URL: http://dblp.uni-trier.de/db/journals/corr/corr1807.html#abs-1807-05174.

[37] Thomas C. Hales. «A proof of the Kepler conjecture». En: *Ann. of Math. (2)* 162.3 (2005), págs. 1065-1185. ISSN: 0003-486X. DOI: 10.4007/annals.2005.162.1065. URL: http://dx.doi.org/10.4007/annals.2005.162.1065.

[38] Thomas Hales. *Big Conjectures*. Talk given at "Big Proof" symposium. see also the "Formal Abstracts" project (https://formalabstracts.github.io/). Jul. de 2017.

[39] John Harrison. «HOL Light: A Tutorial Introduction». En: *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings*. 1996, págs. 265-269. DOI: 10.1007/BFb0031814. URL: https://doi.org/10.1007/BFb0031814.

[40] John Harrison, Josef Urban y Freek Wiedijk. «History of Interactive Theorem Proving.» En: *Computational Logic*. Ed. por Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. Elsevier, 2014, págs. 135-214. ISBN: 978-0-444-51624-4. URL: http://dblp.uni-trier.de/db/series/hhl/hhl9.html#HarrisonUW14.

[41] John Harrison, Josef Urban y Freek Wiedijk. «Preface: Twenty Years of the QED Manifesto.» En: *J. Formalized Reasoning* 9.1 (2016), págs. 1-2. URL: http://dblp.uni-trier.de/db/journals/jfrea/jfrea9.html#HarrisonUW16.

[42] Michiel Hazewinkel. «Hilbert's 1990 ICM Lecture : the 23 problems». En: *Journal of Computational and Applied Mathematics - J COMPUT APPL MATH* (ene. de 1990).

[43] Gerard Huet y Derek C. Oppen. *Equations and rewrite rules: a survey*. Inf. téc. STAN//CS-TR-80-785. Stanford University, Department of Computer Science, 1980.

[44] Andrew David Irvine. «Principia Mathematica». En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta. Spring 2017. Metaphysics Research Lab, Stanford University, 2017.

[45] Paul Jackson. «Nuprl.» En: *The Seventeen Provers of the World*. Ed. por Freek Wiedijk. Vol. 3600. Lecture Notes in Computer Science. Springer, 6 de mar. de 2006, págs. 116-126. ISBN: 3-540-30704-4. URL: http://dblp.uni-trier.de/db/conf/tphol/lncs3600.html#Jackson06.

[46] Theo MV Janssen. «Algebraic translations, correctness and algebraic compiler construction». En: *Theoretical Computer Science* 199.1 (1998), págs. 25-56.

[47] Wolfram Kahl. «Dependently-Typed Formalisation of Relation-Algebraic Abstractions». En: *RAMICS*. Vol. 6663. Lecture Notes in Computer Science. Springer, 2011, págs. 230-247.

[48] Hsiang-Shang Ko y Jeremy Gibbons. «Modularising Inductive Families». En: *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*. WGP '11. Tokyo, Japan: ACM, 2011, págs. 13-24. ISBN: 978-1-4503-0861-8. DOI: 10.1145/2036918.2036921. URL: http://doi.acm.org/10.1145/2036918.2036921.

[49] K. Kunen. *Set Theory*. Second. Studies in Logic. Revised edition, 2013. College Publications, 2011. ISBN: 9781848900509.

[50] Kenneth Kunen. *Set theory: An Introduction to Independence Proofs*. Studies in logic and the foundations of mathematics. Amsterdam, Lausanne, New York: Elsevier Science, 1980. ISBN: 0-444-86839-9. URL: http://opac.inria.fr/record=b1117475.

[51] Xavier Leroy. «Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant». En: *SIGPLAN Not.* 41.1 (ene. de 2006), págs. 42-54. ISSN: 0362-1340. DOI: 10.1145/1111320.1111042. URL: http://doi.acm.org/10.1145/1111320.1111042.

[52] Sam Lindley y Conor McBride. «Hasochism: the pleasure and pain of dependently typed haskell programming.» En: *Haskell*. Ed. por Chung chieh Shan. ACM, 2013, págs. 81-92. ISBN: 978-1-4503-2383-3. URL: http://dblp.uni-trier.de/db/conf/haskell/haskell2013.html#LindleyM13.

[53] Per Martin-Lo*f. *Notes on constructive mathematics*. English. Stockholm : Almqvist & Wiksell, 1970, 109 p. ; 24 cm.

[54] Per Martin-Löf. *An Intuitionistic Theory of Types*. Inf. téc. 1972. URL: /brokenurl#{http://www.cs.chalmers.se/~peterd/kurser/tt03/martinlof72.ps}.

[55] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[56] Roman Matuszewski y Piotr Rudnicki. «Mizar: the first 30 years». En: *Mechanized mathematics and its applications* 4.1 (2005), págs. 3-24.

[57] Conor Thomas McBride. «Agda-curious?: An Exploration of Programming with Dependent Types». En: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: ACM, 2012, págs. 1-2. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364529. URL: http://doi.acm.org/10.1145/2364527.2364529.

[58] Conor McBride. «Ornamental algebras, algebraic ornaments». En: *To appear in Journal of Functional Programming* ().

[59] John McCarthy y James Painter. «Correctness of a compiler for arithmetic expressions». En: vol. 1. Proceedings of Symposia in Applied Mathematics. American Mathematical Society, 1967. Cap. 3, págs. 33-41. ISBN: 9780821867280.

[60] James McKinna y Joel Wright. «A type-correct, stack-safe, provably correct, expression compiler». En: *Submitted to the Journal of Functional Programming* (2006).

[61] K. Meinke y J. V. Tucker. «Universal Algebra». En: *Handbook of Logic in Computer Science (Vol. 1)*. Ed. por S. Abramsky y T. S. E. Maibaum. New York, NY, USA: Oxford University Press, Inc., 1992, págs. 189-368. ISBN: 0-19-853735-2.

[62] Robin Milner. «Implementation and applications of Scott's logic for computable functions». En: *ACM sigplan notices*. Vol. 7. 1. ACM. 1972, págs. 1-6.

[63] F Lockwood Morris. «Advice on structuring compilers and proving them correct». En: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1973, págs. 144-152.

[64] Ulf Norell. «Dependently Typed Programming in Agda». En: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. 2008, págs. 230-266. DOI: 10.1007/978-3-642-04652-0_5. URL: https://doi.org/10.1007/978-3-642-04652-0_5.

[65] Scott Owens, Magnus O. Myreen, Ramana Kumar y Yong Kiam Tan. «Functional Big-Step Semantics». En: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. por Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, págs. 589-615. DOI: 10.1007/978-3-662-49498-1_23. URL: https://doi.org/10.1007/978-3-662-49498-1_23.

[66] Alberto Pardo, Emmanuel Gunther, Miguel Pagano y Marcos Viera. «An Internalist Approach to Correct-by-Construction Compilers.» En: *PPDP*. Ed. por David Sabel y Peter Thiemann. ACM, 2018, 17:1-17:12. URL: http://dblp.uni-trier.de/db/conf/ppdp/ppdp2018.html#PardoGPV18.

[67] Lawrence C Paulson. «Set theory for verification: I. From foundations to functions». En: *Journal of Automated Reasoning* 11.3 (1993), págs. 353-389.

[68] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[69] Lawrence C Paulson. «Set Theory for Verification. II: Induction and Recursion». En: *Journal of Automated Reasoning* 15.2 (1995), págs. 167-215.

[70] Lawrence C Paulson. «A fixedpoint approach to (co) inductive and (co) datatype definitions.» En: *Proof, Language, and Interaction*. 2000, págs. 187-212.

[71] Lawrence C. Paulson. «The Relative Consistency of the Axiom of Choice Mechanized Using Isabelle/ZF». En: *LMS Journal of Computation and Mathematics* 6 (2003), págs. 198-248. DOI: 10.1112/S1461157000000449.

[72] Lawrence C. Paulson. «The relative consistency of the axiom of choice mechanized using Isabelle/ZF». En: *LMS J. Comput. Math.* 6 (2003). Appendix A available electronically at http://www.lms.ac.uk/jcm/6/lms2003-001/appendix-a/, págs. 198-248. ISSN: 1461-1570. DOI: 10.1007/978-3-540-69407-6_52. URL: http://dx.doi.org/10.1007/978-3-540-69407-6_52.

[73] Lawrence C. Paulson. *Isabelle's logics: FOL and ZF*. Inf. téc. 2008.

[74] Lawrence C. Paulson. *ALEXANDRIA: Large-Scale Formal Proof for the Working Mathematician*. Webpage. EC Project: https://bit.ly/2Nb26ys. 2017 — accessed September 2018.

[75] Lawrence C Paulson. «Formalising Mathematics In Simple Type Theory». En: *arXiv preprint arXiv:1804.07860* (2018).

[76] Lawrence C. Paulson y Krzysztof Grabczewski. «Mechanizing Set Theory». En: *J. Autom. Reasoning* 17.3 (1996), págs. 291-323. DOI: 10.1007/BF00283132. URL: https://doi.org/10.1007/BF00283132.

[77] Art Quaife. «Automated deduction in von Neumann-Bernays-Gödel set theory». En: *Journal of Automated Reasoning* 8.1 (1992), págs. 91-147. ISSN: 1573-0670. DOI: 10.1007/BF00263451. URL: https://doi.org/10.1007/BF00263451.

[78] Borut Robič. «The Foundational Crisis of Mathematics». En: *The Foundations of Computability Theory*. Springer, 2015, págs. 9-30.

[79] Camilo Rocha y José Meseguer. «Theorem Proving Modulo Based on Boolean Equational Procedures». En: *RelMiCS*. Vol. 4988. Lecture Notes in Computer Science. Springer, 2008, págs. 337-351.

[80] Bertrand Russell. «Mathematical logic as based on the theory of types». En: *American journal of mathematics* 30.3 (1908), págs. 222-262.

[81] Anne Salvesen y Jan M. Smith. «The Strength of the Subset Type in Martin-Löf's Type Theory». En: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, págs. 384-391. ISBN: 0-8186-0853-6. URL: https://doi.org/10.1109/LICS.1988.5135.

[82] Donald Sannella y Andrzej Tarlecki. *Foundations of algebraic specification and formal software development*. Springer Science & Business Media, 2012.

[83] Michael A. Schoonover, John S. Bowie y William R. Arnold. *GNU Emacs - UNIX text editing and programming*. Hewlett-Packard Press Series. Addison-Wesley, 1992. ISBN: 978-0-201-56345-0.

[84] Carsten Schürmann. «The Twelf Proof Assistant.» En: *TPHOLs*. Ed. por Stefan Berghofer, Tobias Nipkow, Christian Urban y Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, págs. 79-83. ISBN: 978-3-642-03358-2. URL: http://dblp.uni-trier.de/db/conf/tphol/tphol2009.html#Schurmann09.

[85] Dana S. Scott. «A Type-Theoretical Alternative to ISWIM, CUCH, OWHY.» En: *Theor. Comput. Sci.* 121 (1993), págs. 411-440. URL: http://dblp.uni-trier.de/db/journals/tcs/tcs121.html#Scott93.

[86] J. R. Shoenfield. «Axioms of set theory». En: *Handbook of mathematical logic*. Vol. 90. Stud. Logic Found. Math. North-Holland, Amsterdam, 1977, págs. 321-344.

[87] Bas Spitters y Eelis van der Weegen. «Type classes for mathematics in type theory». En: *Mathematical Structures in Computer Science* 21.4 (2011), págs. 795-825.

[88] Andrzej Tarlecki. «Some Nuances of Many-sorted Universal Algebra: A Review». En: *Bulletin of the EATCS* 104 (2011), págs. 89-111. URL: http://albcom.lsi.upc.edu/ojs/index.php/beatcs/article/view/79.

[89] James W Thatcher, Eric G Wagner y Jesse B Wright. «More on advice on structuring compilers and proving them correct». En: *Theoretical Computer Science* 15.3 (1981), págs. 223-249.

[90] «The QED Manifesto». En: *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*. 1994, págs. 238-251. URL: http://link.springer.com/chapter/10.1007/3-540-58156-1_17.

[91] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[92] Nik Weaver. *Forcing for Mathematicians*. World Scientific, 2014. ISBN: 9789814566957. URL: https://doi.org/10.1142/8962.

[93] Makarius Wenzel, Lawrence C. Paulson y Tobias Nipkow. «The Isabelle framework». En: *Theorem proving in higher order logics*. Vol. 5170. Lecture Notes in Comput. Sci. Springer, Berlin, 2008, págs. 33-38. DOI: 10.1007/978-3-540-71067-7_7. URL: http://dx.doi.org/10.1007/978-3-540-71067-7_7.

[94] Markus Wenzel. «Isar - A Generic Interpretative Approach to Readable Formal Proof Documents». En: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*. Ed. por Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring y Laurent Théry. Vol. 1690. Lecture Notes in Computer Science. Springer, 1999, págs. 167-184. DOI: 10.1007/3-540-48256-3_12. URL: https://doi.org/10.1007/3-540-48256-3_12.

[95] A.N. Whitehead y B. Russell. *Principia Mathematica*. Principia Mathematica v. 2. University Press, 1912. URL: https://books.google.com.ar/books?id=sbTVAAAAMAAJ.

[96] Freek Wiedijk. «The QED manifesto revisited». En: *Studies in Logic, Grammar and Rhetoric* 10.23 (2007), págs. 121-133.

[97] Thomas Williams, Pierre-Évariste Dagand y Didier Rémy. «Ornaments in practice». En: *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*. Ed. por José Pedro Magalhães y Tiark Rompf. ACM, 2014, págs. 15-24. DOI: 10.1145/2633628.2633631. URL: http://doi.acm.org/10.1145/2633628.2633631.

[98] Thomas Williams y Didier Rémy. «A principled approach to ornamentation in ML». En: *PACMPL* 2.POPL (2018), 21:1-21:30. DOI: 10.1145/3158109. URL: http://doi.acm.org/10.1145/3158109.

[99] The Coq development team. *The Coq proof assistant reference manual*. Version 8.0: http://coq.inria.fr. LogiCal Project. 2004. URL: http://coq.inria.fr.