Universidad Nacional
Autónoma
de México

Facultad de Ciencias

Verificación Formal y Programación Certificada en Coq con Program: un ejemplo práctico

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Diego Carrillo Verduzco

TUTORA

Lourdes del Carmen González Huesca

Ciudad Universitaria, Cd. Mx. 2020

1. Datos del alumno
   Carrillo
   Verduzco
   Diego
   55 98 27 40
   Universidad Nacional Autónoma de
   México
   Facultad de Ciencias
   Ciencias de la Computación
   312273510

2. Datos del tutor
   Dra
   Lourdes del Carmen
   González
   Huesca

3. Datos del sinodal 1
   Dr
   Favio Ezequiel
   Miranda
   Perea

4. Datos del sinodal 2
   Dr
   Canek
   Peláez
   Valdés

5. Datos del sinodal 3
   M en C
   Pilar Selene
   Linares
   Arévalo

6. Datos del sinodal 4
   M en C
   Noé Salomón
   Hernández
   Sánchez

7. Datos del trabajo escrito
   Verificación Formal y Programación Certificada en Coq con Program: un ejemplo práctico
   67 p
   2020

Verificación Formal y Programación certificada en Coq con Program: un ejemplo práctico

o, "Seis Lenguajes de Programación y Una Estructura de Datos"

en el que se realiza un breve estudio del estado del arte de las herramientas de verificación formal enfocándose en aquellas que promueven desarrollar la implementación de un programa y la demostración de su especificación simultáneamente y a contiuación se ejemplifica el uso de una de estas herramientas implementando una muy particular estructura de datos

Diego Carrillo Verduzco

Tesis realizada bajo el proyecto PAPIME 102117 "Tópicos en Ciencia de la Computación Teórica"

Agradecimientos

A mis padres, mi abuelita y mi hermana; ambiente que me creó y me crío, sin el cuál yo no estaría aquí hoy.
A Lourdes por tolerar (casi) todas mis atrocidades y guiarme en la escritura de esta tesis.
A mis amixes, incluyendo a Víctor, Ferfy, David, Gilberto y Dan, por ser la más grata compañía. And of course Anshula, despite being so far away.
A Karla, el mejor conejo que conozco.
A Javier, la única persona con la audiacia suficiente para compartir un saludo especial conmigo.
A Itzel, por mantenerme en contacto con mi lado humano (entre muchas otras cosas).
A Rodrigo, por más razones de las que puedo enunciar aquí.
A Sara, por aguantarme durante ya casi una década.
A Claudia, la mejor amiga que uno podría pedir.
A Alma, con quien me tropecé un día en camino a trabajar sobre esta tesis pero decidí mejor pasar un rato (unas horas (unos días (unos meses (y contando rápidamente hacia un año)))) compartiendo el tiempo con ella. Ha valido la pena cada instante.

Índice general

1. Motivación **1**
1.1. Sobre el presente trabajo 3
1.2. Corrección 3
1.3. Software Testing 4
1.3.1. Pruebas unitarias 4
1.3.2. Pruebas de integración 7
1.3.3. Pruebas de sistema 7
1.4. El costo de las pruebas y los métodos formales 7

2. Deslizándose sobre el espectro de la verificación formal **11**
2.1. Verificación Estática Extendida (Extended Static Checking) 16
2.1.1. Java/OpenJML 17
2.1.2. Whiley 20
2.2. Verificación basada en SMT 22
2.2.1. Dafny 22
2.2.2. F✯ 25
2.3. Verificación con tipos dependientes 29
2.3.1. Idris 29
2.3.2. Coq 33

2.4. Sobre el costo de aprender una herramienta de verificación formal 36

3. Construyendo programas certificados en Coq **39**
3.1. Árboles cartesianos 40
3.1.1. Construyendo un árbol cartesiano 42
3.2. Programación funcional en Coq 43
3.3. Programación a partir de especificaciones 44
3.4. Verificación formal, sin `Program` 52
3.5. Y a todo esto, ¿por qué usar `Program`? 57

4. Conclusiones **61**

Referencias **65**

1

Motivación

Adam Chlipala comienza su libro *Certified Programming with Dependent Types* con la siguiente frase[1]: *"We would all like to have programs check that our programs are correct"* ("A todos nos gustaría tener programas que verificaran que nuestros programas son correctos").

La cantidad de sustancia y significado detrás de cada palabra en esa frase es impactante, especialmente para una frase de trece palabras. De particular interés para el autor es la intención de la expresión *would like*: es transparente para quienes se dedican a escribir programas que, en un mundo ideal, cada programa escrito vendría acompañado de una prueba que demuestre que este programa es *correcto*, es decir, que se atiene a la especificación a partir de la cual el programa fue escrito. Además, sería ideal que estas pruebas de corrección fueran escritas no por un humano sino por un programa que pueda generarlas automáticamente[2]. Esperaríamos por lo menos que todo software contara con garantías de que su ejecución no incurrirá en errores catastróficos, que por ejemplo causen que nuestras torres de control pierdan comunicación con los aviones o, peor aún, que los sistemas de inscripción de nuestras escuelas no estén disponibles al regresar a clases.

Los **métodos formales** son diferentes técnicas basadas en principios matemáticos (y particularmente en la estrecha relación entre la computación y la lógica) que se pueden emplear durante el diseño y desarrollo de sistemas tanto de software como de hardware con el fin de hacerlos más robustos y confiables. En particular, la **verificación formal** consiste en producir una demostración de que un sistema es correcto con respecto a cierta especificación. En este trabajo nos enfocaremos únicamente en la verificación formal automática y/o asistida por computadora, ya que el autor considera la idea de demostrar la corrección de un programa con una prueba en lápiz y papel bastante siniestra.

Aunque la finalidad de verificar formalmente nuestro software es evidentemente buena (después de todo, a todos nos gustaría que un programa revisara que nuestros programas con correctos), en la práctica es poco común encontrar proyectos con garantías de confiabilidad y robustez verificadas formalmente. En la mente de muchos programadores, hace falta que el sistema a verificar sea auténticamente crítico[3] para justificar todo el esfuerzo adicional que conlleva crear un programa verificado formalmente.

Es la opinión del autor, y factor motivador central detrás de este trabajo, que **el principal obstáculo al que se enfrentan las herramientas de verificación formal para ser adoptadas más ampliamente en la industria, es la *ergonomía***; el poder ser utilizadas sin alterar de manera tan radical el proceso de desarrollo de un sistema. El proceso de crear un programa verificado debería, idealmente, ser lo más cercano posible al proceso de desarrollar un programa corriente.

Esta aseveración no es particularmente atrevida; por una parte es difícil lograr que la gente abandone los hábitos a los que están acostumbrados (así como lograr que adopten hábitos nuevos) y por otra parte todo esfuerzo adicional dirigido hacia asegurar la corrección de un programa impone un costo en tiempo de desarrollo que podría estar utilizándose para otras actividades, tales como implementar funcionalidad adicional.

En realidad, en el *mundo real*[4], hasta los métodos menos excéntricos de *testing* de software como las pruebas unitarias, pruebas de integración, pruebas de regresión etc. y los procesos que utilizan éstas como elemento central, como *Test-Driven Development (TDD)*, suelen ser vistos como un ideal al cual aspirar; en el caso promedio se integran algunos de estos métodos al proceso de desarrollo sólo en medida de qué tan fácil sea integrarlos a un proyecto ya en progreso, no de manera paralela al desarrollo sino de manera *post-hoc*.

1.1. Sobre el presente trabajo

En este trabajo escrito el autor pretende, además de satisfacer los requisitos necesarios para titularse exitosamente de la licenciatura en Ciencias de la Computación, presentar una breve pero amplia comparativa de varias herramientas para hacer programación certificada enfocándose en su ergonomía a través de un ejemplo común (lo cual haremos en el capítulo 2 de este trabajo), así como detallar a mayor profundidad el uso de una de estas herramientas en particular (Coq) para el desarrollo de un programa más elaborado, con el mismo enfoque mencionado anteriormente (en el capítulo 3). Tras haber realizado este viaje, reflexionaremos acerca de lo aprendido en las conclusiones (en el capítulo 4).

1.2. Pero, ¿qué es la corrección de todos modos?

Anteriormente dimos una definición informal[5] de que un programa sea *correcto*. Procedemos a dar una definición ligeramente menos informal, inspirada en un *blog post* de Hillel Wayne [22]:

Definición: *Decimos que un programa es* ***correcto*** *con respecto a cierta especificación si el programa cumple toda característica que la especificación señale que debe cumplir, ya sea ésta de carácter afirmativo o negativo.*[6]

Introduciendo esta "mejor" definición de corrección[7], surge de manera natural la pregunta: *¿de dónde obtenemos la especificación contra la cual verificar el programa?*. La respuesta a la pregunta, por supuesto, proviene de factores externos al programa en sí y depende del contexto en el que dicho programa se esté desarrollando. Si el programa a desarrollar está siendo requerido por un cliente de una empresa de desarrollo de software, la especificación surgirá de todo aquello que el cliente quiera que el programa haga o no haga. Si el programa fue asignado como tarea a una estudiante de Ciencias de la Computación, la especificación surge de lo que el profesor haya solicitado del programa (o, alternativamente, la especificación será "hacer cualquier cosa que logre convencer al profesor de ponerme una calificación que me parezca suficiente").

Es necesario hacer una distinción entre la "especificación natural" (la descripción en lenguaje natural del propósito del programa) y la "especificación formal" (la traducción a algún mecanismo formal de la descripción

anteriormente dada).

La ***validación*** es asegurarse que el programa y su especificación (formal) empatan con la descripción solicitada del programa[8].

En este trabajo nos enfocaremos únicamente en el problema de la verificación, y no profundizaremos en cuestiones relacionadas a generar especificaciones "buenas" contra las cuales verificar nuestros programas.

Incluso en flujos de trabajo en los que no se emplean los métodos formales para verificar los programas, es indispensable tener *alguna* metodología para asegurarse de que el programa desarrollado sea correcto con respecto a su especificación. En el peor de los casos, esta metodología consistirá en utilizar el programa por veinte minutos, probar quizás un solo caso de uso y decir *"meh, ¡se ve bien para mí!"*. Sin embargo, si el equipo de desarrollo tiene el más ligero interés en entregar un producto de calidad (por ejemplo, si su salario o su calificación dependen de ello), su metodología deberá conformarse a principios más rigurosos, tales como los del *software testing*.

1.3. *Software Testing*: establecer corrección *a la antigüita*

Las **Buenas Prácticas** de la programación[9] establecen que se deben incorporar al proceso de la Ingeniería de Software medidas para asegurar la calidad del producto de software. (Para mayor información sobre las Buenas Prácticas, consultar [23]). Entre estas medidas se encuentran las *pruebas*: métodos que aseguran el funcionamiento de un producto de software a diferentes escalas. En la escala más pequeña encontramos las pruebas unitarias

1.3.1. Pruebas unitarias

Si entendemos a una *unidad* como el componente más pequeño de un programa que cumple con una funcionalidad particular (en lenguajes funcionales una unidad sería una función, en Java sería una clase, en Pascal sería un procedimiento, etc.), una prueba unitaria se encarga de validar que una unidad en particular cumple con su propósito especificado y no incurre en errores de formas inesperadas. En términos prácticos, una prueba unitaria es un bloque de código que se encarga de someter una unidad a usos similares a como se pretende usarla en el programa real, y verificar que ésta produce los resultados que se esperan de ella. Si una unidad *pasa* una prueba unitaria,

se puede considerar correcta (respecto a dicha prueba). Si la ejecución de la prueba unitaria *falla*, la implementación de la unidad es incorrecta y debe ser corregida.

Por ejemplo, supongamos la existencia de una función `suma(int a, int b) ->int` que por alguna inexplicable razón tiene la responsabilidad de sumar dos enteros y regresar el resultado de esta suma. Una (mala) prueba unitaria se aseguraría de que el resultado de ejecutar `suma(2, 2)` es 4. Una (ligeramente) mejor prueba unitaria tendría un mayor número de ejemplos para asegurarse de que el resultado de invocar la función sea el esperado. Una (aún mejor) *suite* de pruebas unitarias para la función se aseguraría de que los resultados de varias invocaciones de la función con argumentos distintos fueran las esperadas, y además se aseguraría de que la función responda adecuadamente a casos excepcionales, tales como si la función se ejecuta con el valor máximo del tipo de dato `int`, y otros casos extremos.

En este ejemplo dilucidamos varios de los problemas con las pruebas unitarias. Primero, los únicos casos en los que podremos estar seguros de que la unidad responde apropiadamente son aquellos que se le puedan ocurrir al equipo de desarrollo encargado de escribirlas. Segundo, siendo que las pruebas unitarias son código en sí, probar exhaustivamente una unidad implica escribir grandes cantidades de código que, además, no aportan directamente hacia el avance del desarrollo del programa, razón por la cual incluso las más simples pruebas unitarias se pueden llegar a omitir del desarrollo de un producto a favor de implementar más de la funcionalidad requerida por éste. Tercero, dado que *las pruebas unitarias son código en sí*, **¡éstas mismas son suceptibles a contener errores!**[10]

A pesar de estas desventajas, en general suele ser buena idea cubrir aspectos elementales de la especificación de una unidad con al menos una prueba unitaria, garantizando que se conozca al menos una ejecución de la unidad que produzca un resultado correcto.

Una brevísima divagación: *Test Driven Development*

Existe una metodología de desarrollo de software comúnmente conocida pero no ampliamente aplicada que establece que antes de siquiera pensar en implementar la funcionalidad de un programa, su especificación debe traducirse a una *suite* de pruebas unitarias, de tal manera que se puede validar el funcionamiento de una unidad antes de que la unidad

haya sido implementada. Tras implementar las pruebas unitarias, se procede a implementar la unidad requerida hasta llegar al punto en el que pase todas las pruebas. Este proceso se sigue para cada requerimiento que se tenga del programa, de tal forma que se sabe que una unidad no se ha implementado por completo y sin errores hasta que logre pasar todas sus pruebas unitarias. A esta metodología se le conoce como Desarrollo Guiado por Pruebas o *Test Driven Development (TDD)*.

A cualquier persona que haya escrito una cantidad no-trivial de código este proceso le sonará, naturalmente, tedioso. Seguir esta metodología no sólo implica que la cantidad de tiempo para implementar cualquier funcionalidad se verá multiplicada, sino que también se le da prioridad a escribir código que sencillamente no es *divertido* de escribir; uno de los placeres simples de la vida es escribir un pedazo de código y observar cómo la computadora utiliza todo su poder para cumplir con nuestros nefastos designios, por triviales que éstos sean. El *Test Driven Development* provoca que se incremente dramáticamente el intervalo de tiempo entre el comienzo de la escritura del código relevante a la solución del problema y la llegada del sublime momento de la verdad a la hora de ver el programa ejecutarse (si bien se puede argumentar que ver todas las pruebas correre exitosamente aumenta esta sensación de satisfacción), haciendo que el proceso sea más tedioso.

Finalmente, al igual que escribir *cualquier pedazo de código*, escribir buenas pruebas unitarias es una habilidad que tiene que desarrollarse para garantizar la efectividad de esta metodología; es irrelevante escribir todas las pruebas unitarias del mundo si estas pruebas no están cuidadosamente implementadas.

Muchas de las herramientas de verificación formal que examinaremos en los siguientes capítulos ponen énfasis en demostrar la corrección de unidades de manera similar a las pruebas unitarias, pero el esfuerzo se traslada de escribir pruebas unitarias que cubran adecuadamente la especificación y no contengan errores en sí mismas, a expresar adecuadamente la especificación de la unidad y asistir en la produccion de una demostración de que la unidad cumple con dicha especificación.

Debido a su naturaleza, las pruebas unitarias por sí mismas no pueden ni pretenden validar que el programa en su conjunto cumple con la especificación requerida (de hecho, por definición, las pruebas unitarias pueden demostrar que cierto código es incorrecto, pero *nunca* se puede decir que demuestren que cierto código es correcto); se requieren pruebas aplicadas a

niveles más amplios para eso, tales como las pruebas de integración.

1.3.2. Pruebas de integración

Una vez que se ha validado el funcionamiento de las unidades independientemente, se debe validar que la interacción entre estas unidades funciona como se espera. Debido a la altamente variable naturaleza de las arquitecturas de los programas que se pueden crear, el proceso de hacer pruebas de integración puede no ser tan automatizable como el de hacer pruebas unitarias; dependiendo de cómo esté construido el programa, puede ser necesario realizar estas pruebas manualmente.

Tras asegurarse que las interacciones entre las unidades funcionan adecuadamente, se puede proceder a probar que el sistema en su totalidad cumple con su especificación, lo cual se hace en las pruebas de sistema.

1.3.3. Pruebas de sistema

Usualmente, las pruebas de sistema se llevan a cabo por un equipo especializado de *testing* o *Quality Assurance (QA)* que se encarga de probar, tan exhaustivamente como sea posible y el tiempo lo permita, cada requerimiento de funcionalidad del sistema, asegurándose de que éste cumpla con su especificación y reportando al equipo de desarrollo cualquier *incidencia* (un caso de uso con resultados incorrectos o inesperados).

Las pruebas de sistema suelen ser pruebas de caja negra (*Black Box Testing*); esto es, el equipo de pruebas no puede leer ni modificar el código del programa, sino que puede únicamente utilizarlo en la misma capacidad que el usuario final del sistema (esto en oposición a las pruebas de caja blanca o *White Box Testing*, en donde el equipo de pruebas tiene algún conocimiento del funcionamiento interno del sistema).

1.4. El costo de las pruebas y los métodos formales

Podría argumentarse que las metodologías de pruebas descritas en la sección anterior (especialmente las pruebas de sistema) son una aproximación *obvia* para asegurar la corrección de un programa; después de todo, lo único que se está haciendo es ir probando partes cada vez más amplias del

programa de manera menos o más automatizada, verificando que el comportamiento de cada parte empate con lo que se espera de ésta y corrigiéndola en caso negativo. A pesar de la naturaleza simple de estas pruebas, implementarlas adecuadamente requiere planeación y esfuerzo adicional; esto se traduce en más tiempo de desarrollo, medida que usualmente se busca minimizar.

Como describiremos más adelante, el uso de métodos formales impone una carga adicional de tiempo y esfuerzo sobre el desarrollo, ya que se modifica el proceso de diseño e implementación de los programas para acomodar a la verificación del programa. Además, tradicionalmente los desarrolladores de software no suelen estar entrenados en el empleo de estos métodos (ni en el trasfondo matemático requerido para comprender su funcionamiento), estableciendo aún una barrera más en el camino a su más amplia adopción en el mundo del desarrollo de software.

Debe quedar claro entonces que una posible estrategia para aumentar la adopción de los métodos formales (y más específicamente la verificación formal) es reducir en la medida de lo posible este costo adicional sobre el proceso de desarrollo de software. En el capítulo siguiente exploraremos el concepto de la *programación certificada* y daremos un breve recorrido a través de varias herramientas que incorporan este concepto directamente en el proceso de diseñar y crear programas, ofreciendo una visión promisoria para la adopción de los métodos formales en el futuro próximo.

Notas

[1]En realidad comienza con la portada y el índice y todo eso, pero vos comprenderéis.

[2]En un mundo más ideal todavía, los programas mismos estarían escritos por otros programas, los primeros serían correctos por construcción, y la mayoría de los programadores se quedaría sin trabajo.

[3]Razón por la cual es común que para convencer a alguien de la utilidad de los métodos formales, un defensor de éstos invoque el ejemplo del software de control de un avión, etc.

[4]Expresión de la jerga del mundo académico que puede traducirse como "fuera del mundo académico".

[5]O, como se le conoce en el mundo de la moda, *casual*.

[6]Como el lector podrá darse cuenta, esta definición está sacada de la manga para beneficio del autor. Lamentablemente, debido a la dificultad de especificar (heh) el concepto de *especificación* de manera formal, raramente se habla de la corrección en terminos cuantificables. En su lugar nos conformamos con lidiar con esta noción intuitiva de especificación.

[7]O, como algunos dicen, "correctitud", haciendo caso omiso a que la palabra suene

horrible.

[8]Problema famosamente complicado en la industria, ya que como es sabido, *el cliente nunca sabe lo que quiere*.

[9]"Buenas Prácticas" es como los programadores se refieren a la sabiduría tradicional sobre desarrollo de software que ellos personalmente han juzgado como digna de practicar cotidianamente, a diferencia de aquella sabiduría que no les resulta cómoda o útil.

[10]Y aquí es donde el lector propondría escribir pruebas unitarias para las pruebas unitarias si el autor no le hubiera ganado el chiste.

2

Deslizándose sobre el espectro de la verificación formal

Como mencionamos en el capítulo anterior, llamamos **verificación formal** al acto de utilizar técnicas fundamentadas en las matemáticas, particularmente en la lógica, para demostrar la corrección de algún programa con respecto a cierta especificación. En el contexto de este trabajo, consideraremos únicamente aquellas técnicas de verificación formal asistidas por computadora, sea ya que la demostración de corrección sea producida de manera interactiva por el desarrollador con asistencia de la máquina o que ésta se encargue (semi-)automáticamente de verificar la corrección del programa.

Una brevísima divagación: orígenes de la verificación formal

La verificación formal comparte sus orígenes con los demostradores de teoremas automáticos. Éstos, a su vez, surgieron a partir del sueño (que como descubrieron eventualmente, era *guajiro*) de obtener un pro-

cedimiento (en el sentido de algoritmo, a pesar de que la búsqueda de tal procedimiento fue concebida antes de que se estudiara la noción de *algoritmo* como tal) capaz de decidir la verdad o falsedad de cualquier proposición matemática.

Aunque tal procedimiento no puede existir, este hecho no impidió el desarrollo de las herramientas de automatización de demostraciones que darían lugar a las modernas técnicas de verificación formal de programas que exploraremos en esta sección.

Una de las primeras herramientas creadas con este propósito fue LCF, desarrollado en sus inicios por Robin Milner a principios de la década de 1970 [14]. LCF, nombrado tras y basado en la *Logic for Computable Functions* desarrollada por Dana Scott en 1969 [18], es un sistema de demostración interactivo que implementa **razonamiento hacia atrás** (*backward reasoning*), descomponiendo una conclusión o **meta** en varias submetas más sencillas utilizando la aplicación de un conjunto fijo de comandos llamados **tácticas**. A partir de la necesidad de extender este conjunto de tácticas se desarrolló ML (*Meta-Language*), antecesor directo de Caml, abuelo de OCaml, y primo-tatarabuelo de Coq.

Algunos otros experimentos tempranos en el mundo de la verificación formal son Automath[a], iniciado por Nicolaas Govert de Bruijn; y Mizar[b], iniciado por Andrzej Trybulec. Ambos proyectos tenían la intención de ser lenguajes para facilitar la escritura de demostraciones matemáticas que pudieran ser verificadas automáticamente por una computadora, con la esperanza de ser utilizados para formalizar los fundamentos de las matemáticas.

[a] Del cual se puede consultar más en `https://www.win.tue.nl/automath/`
[b] `http://mizar.uwb.edu.pl/`

En el contexto de la verificación formal, un *certificado* es un artefacto matemático formal que demuestra que un programa cumple con una especificación dada [2]. De cierto modo, puede decirse que el propósito de verificar formalmente un programa es producir un certificado para dicho programa. La idea de la **programación certificada** surge entonces de combinar el desarrollo y la verificación formal de un programa en el mismo proceso, produciendo tanto el programa como su certificado de manera (casi) paralela.

La investigación en verificación formal ha producido recientemente diversas herramientas que proveen mecanismos para hacer programación certificada, que varían en términos de la expresividad que le conceden al desarrollador para reflejar las especificaciones del programa, y de la cantidad de esfuerzo

requerido para producir el certificado del programa. En este capítulo daremos un recorrido inspeccionando brevemente varias de estas herramientas a través de un programa de ejemplo en común, certificado en cada una de estas herramientas.

Como todo en la vida, las herramientas que estudiaremos caen sobre un espectro; en nuestro caso, el espectro al que nos referimos es una región del espacio de la relación entre la cobertura provista por la herramienta (qué tantas garantías nos permite expresar), y el esfuerzo requerido del desarrollador. Por un lado de este espacio tenemos herramientas que nos permiten una cantidad limitada de propiedades sobre nuestro programa, y gracias a estas limitaciones las herramientas se pueden valer de procesos automáticos para verificar dichas propiedades. Por el otro lado, encontramos herramientas que nos dan una gran expresividad para incluir especificaciones elaboradas sobre nuestros programas, pero que pueden llegar a requerir que el desarrollador elabore demostraciones enteras para garantizar que se cumplan tales especificaciones.

El ejemplo clásico del primer conjunto de herramientas es el sistema de tipos, el cual permite eliminar una clase común de errores (los de tipo, naturalmente) sin requerir del programador nada más que las anotaciones de tipos que sean necesarias en su programa (algunos sistemas de tipos como el de OCaml y el de Haskell, incluso permiten omitir la mayoría de las anotaciones de tipos que se esperarían en otros sistemas, *all hail type inference*). Moviéndose un poco hacia arriba (ver la figura 2.1) encontramos herramientas que con sólo un poco de esfuerzo adicional de parte del programador (usualmente en la forma de anotaciones adicionales dentro del programa) logran capturar clases adicionales de errores en tiempo de compilación. Ejemplos de estas herramientas son ESC/Modula-3 y su sucesor, ESC/Java, el cual exploraremos con más (mas no mucho) detalle en esta sección.

En la Figura 2.1 presentamos una visualización[11] de este espacio[a]. En el estrato superior encontramos herramientas que permiten generar programas con garantías de que cumplen con su especificación funcional (que es el fin último al que podemos aspirar a llegar, solamente debajo de "escribir una especificación y obtener automáticamente un programa que la cumpla", pero eso entra más bien en el dominio de la síntesis de programas), pero que requieren un gran nivel de esfuerzo de parte del programador; usualmente (como en el caso de Coq) hace falta desarrollar una demostración entera

[a] Inspirada en un diagrama encontrado en [6]

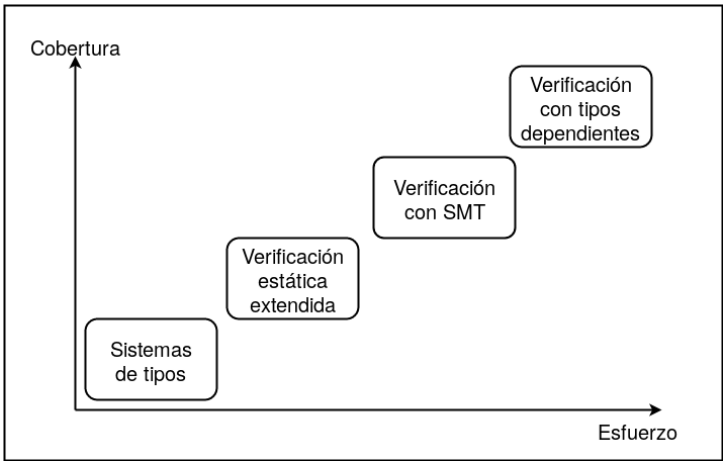


Figura 2.1: El Espectro de la Verificación

de que el programa cumple todas las propiedades. En esta esfera, el programador adquiere un rol secundario de demostrador, ya que la herramienta de verificación es sólamente tan útil como el programador sea capaz de desarrollar las habilidades necesarias para expresar especificaciones y desarrollar pruebas de éstas. Sin embargo, existen muchos desarrollos modernos en herramientas de verificación formal que tienen por objetivo poder generar mejores garantías de corrección a los programas sin tener que invertir tanto tiempo en la generación de pruebas u otros artefactos adicionales al programa mismo. Muchos de estos desarrollos han sido posibles gracias a las mejoras en poder y velocidad que han sufrido los solucionadores SMT más avanzados en tiempos recientes. (De hecho, varios de los lenguajes que exploraremos en esta sección utilizan **el mismo** solucionador de SMT, Z3, para demostrar su especificación).

En esta sección nos encargaremos de dar un breve panorama del funcionamiento y la experiencia de programación de varias herramientas deslizándonos sobre el espectro de la verificación formal, de menos a más automatizado, utilizando un mismo programa en todos los casos. Este programa es `leftPad`.

Sobre `leftPad`

El 22 de marzo de 2016, Azer Koçulu retiró de su repositorio público, en un ataque de frustración motivado por complicaciones legales (*How one*

developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript[b]), varias de las bibliotecas que había escrito y publicado en NPM (*Node Package Manager*), el manejador de paquetes *de facto* utilizado por el ambiente de ejecución Node para el lenguaje JavaScript. Una de las bibliotecas retiradas fue `left-pad`, conformada por una sola función cuyo propósito es, como su nombre sugiere, agregar un relleno de cierto carácter a una cadena por la izquierda. La biblioteca es tan trivial que incluimos el código completo de ésta:

```
1  module.exports = leftpad;
2
3  function leftpad (str, len, ch) {
4    str = String(str);
5
6    var i = -1;
7
8    if (!ch && ch !== 0) ch = ' ';
9
10   len = len - str.length;
11
12   while (++i < len) {
13     str = ch + str;
14   }
15   return str;
16 }
```

Para la mala fortuna de una enorme cantidad de desarrolladores alrededor del mundo, muchos paquetes importantes en NPM, incluyendo React[c], una popular biblioteca para construir interfaces de usuario, dependían ya fuera de manera directa o indirecta de `left-pad`. Como resultado, incontables proyectos en todo el planeta resultaron *rotos*, en muchos casos faltos de una biblioteca de la que los desarrolladores no estaban conscientes que dependían. La crisis fue tal que los administradores de NPM se vieron obligados a publicar de nuevo la biblioteca a sólo unas horas de que fuera removida de la plataforma.

De manera sólo tangencialmente relacionada, el 20 de abril de 2018, el autor, desarrollador y entusiasta de los métodos formales Hillel Wayne[d]

[b] `https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/`
[c] `https://reactjs.org/`
[d] `https://www.hillelwayne.com`

formuló en Twitter un reto abierto a implementar y verificar formalmente (a partir de su especificación) tres funciones que él consideraba de naturaleza fuertemente imperativa. Esto motivado por un sentimiento prevalente en los círculos que prefieren la programación funcional, donde se opina que es más fácil razonar sobre código funcional puro ya que no hace falta preocuparse por el estado o los efectos secundarios.

La primera de estas funciones fue `leftPad`, especificada (informalmente) así: toma un carácter de relleno, una cadena y una longitud total, y regresa la cadena con un relleno por la izquierda con el carácter y con la longitud dada. Si la longitud total es menor a la longitud de la cadena de entrada, devuelve la cadena de entrada sin cambios.

Por ejemplo, el resultado de `leftPad(‘‘hola’’, 10, ’a’)` tendría que ser `‘‘aaaaaahola’’`, y el resultado de `leftPad(‘‘hola’’, 4, ’a’)` sería `‘‘hola’’`.

Ocurrió que varios otros entusiastas de la verificación formal respondieron al reto y Wayne recopiló las soluciones convenientemente en un repositorio en GitHub[e], permitiéndonos examinarlas y utilizarlas como las utilizaremos en esta sección, para comparar su uso y contrastar las formas en las que las herramientas empleadas funcionan para verificar programas.

2.1. Verificación Estática Extendida (Extended Static Checking)

La primera aproximación a la verificación que abordaremos es la verifiación estática extendida. Difiere de las aproximaciones que veremos más adelante (y cabe distinguirla en particular de la verificación usando SMT, ya que en la práctica operan de maneras muy similares hoy en día) principalmente en enfoque: la verificación estática extendida, como su nombre sugiere, pretende principalmente extender las garantías que un sistema de tipos estático provee, permitiendo asegurar que un programa cumple con ciertas propiedades desde tiempo de compilación, pero no necesariamente dando tanta facilidad para probar la especificación entera del programa.

[e] `https://github.com/hwayne/lets-prove-leftpad`

2.1.1. Java/OpenJML

La verificación estática extendida (ESC por sus siglas en inglés, Extended Static Checking) para Java tiene sus orígenes en el Compaq Systems Research Center. El proyecto ESC/Java fue lanzado en 1997, siendo éste sucesor de un proyecto similar, ESC/Modula-3, un verificador estático para el lenguaje Modula-3 (pariente cercano de Pascal). ESC/Java permite anotar un programa con anotaciones para expresar condiciones y garantías, las cuales se verifican con ayuda de un demostrador de teoremas automático. Notoriamente, en el artículo que introduce el sistema [6], se deja declarada la intención de abandonar la corrección y la completitud (esto es, que existe la posibilidad de que el sistema reporte tanto falsos negativos, que el programa contenga errores pero estos no sean detectados; como falsos positivos, que el programa no contenga errores pero se emitan advertencias a pesar de ello), bajo el argumento de que "[...]si el verificador encuentra suficientes errores para reponer el costo de tener que ejecutarlo y estudiar su salida, entonces el verificador será una ganancia[...]". En otras palabras, ESC/Java busca ser *práctico* por encima de ser *ideal*: el fin de la herramienta es explícitamente tratar de eliminar clases comunes de errores de tiempo de ejecución.

En sus inicios, ESC/Java utilizaba un demostrador de teoremas desarrollado igualmente en el SRC llamado Simplify. La arquitectura de ESC/Java en esencia pasa por 5 etapas:

1. ***Front end*** - Transforma programas de Java con anotaciones en árboles de sintaxis abstracta. Asimismo, el *front end* genera lo que llaman un *type-specific background predicate* (BP_T), que es una fórmula en lógica de primer orden que codifica información sobre los tipos y campos que utilizan los métodos en una clase dada.

2. **Traductor** - Traduce el árbol de sintaxis abstracta en un lenguaje basado en el *Guarded Command (GC) Language* formulado por Dijkstra. Aunque profundizar sobre este lenguaje está fuera del alcance de este trabajo, baste decir que se conforma de comandos de la forma `assert E` con `E` una expresión booleanea, y se considera que la ejecución de un programa "salió mal" si en algún momento la ejecución llega a un comando de la forma `assert E` y `E` es falsa.

3. **Generador de VCs** - Toma el programa en el lenguaje de GC y genera condiciones de verificación (VCs) para cada comando con guardia. Estas condiciones son en esencia predicados en lógica de primer orden

que se cumplen solamente para los estados del programa desde los que la ejecución del comando no puede salir mal.

4. **Demostrador de teoremas** - Para cada rutina R del programa, se invoca a Simplify para intentar demostrar la VC de R, suponiendo que se cumple BP_T y una serie de otras suposiciones generales que conciernen a la semántica de Java.

5. **Postprocesador** - La última etapa procesa la salida del demostrador y produce advertencias para todos los casos en los que no se puedan probar las condiciones de verificación.

Esta descripción de la arquitectura de ESC/Java está arrancada casi-textualmente de *Extended Static Checking for Java*. Su inclusión en este trabajo no es gratuita; la arquitectura de varios de los sistemas automatizados que discutiremos en adelante es bastante similar a ésta: tomar código anotado con predicados sobre el estado o los tipos del programa y transformarlo a una forma adecuada para ser procesado por un demostrador de teoremas.

Eventualmente, ESC/Java dio lugar a ESC/Java2[12] que a su vez dio lugar a OpenJML[f]. A diferencia de ESC/Java, que utilizaba el demostrador Simplify únicamente, OpenJML produce especificaciones en un formato soportado por diversos demostradores automáticos (SMT-LIB), lo cual implica que un avance significativo en el progreso de un demostrador en particular representa un beneficio en general para los usuarios de OpenJML.

Nuestra primera implementación de leftPad utiliza OpenJML para anotar el programa con precondiciones y postcondiciones. El programa es, misericordiosamente, bastante corto[13].

```
1 public class LeftPad {
2     //@ requires n >= 0;
3     //@ requires s != null;
4     //@ ensures \result.length == Math.max(n, s.length);
5     //@ ensures \forall int i; i >= 0 && i < Math.max(n -
      s.length, 0);
6     //@         \result[i] == c;
7     //@ ensures \forall int i; i >= 0 && i < s.length;
8     //@         \result[Math.max(n - s.length, 0) + i] == s[i];
9     static char[] leftPad(char c, int n, char[] s) {
```

[f] http://www.openjml.org/

```
10          int pad = Math.max(n - s.length, 0);
11          char[] v = new char[pad + s.length];
12          int i = 0;
13
14          //@ maintaining i >= 0 && i <= pad;
15          //@ maintaining \forall int j; j >= 0 && j < i;
16          //@                       v[j] == c;
17          for(; i<pad; i++) v[i] = c;
18
19          //@ maintaining i >= pad;
20          //@ maintaining \forall int j; j >= 0 && j < pad;
21          //@                         v[j] == c;
22          //@ maintaining \forall int j; j >= pad && j < i;
23          //@                         v[j] == s[j - pad];
24          for(i = pad; i < v.length; i++) v[i] = s[i - pad];
25
26          return v;
27      }
28  }
```

El método estático `leftPad` es precedido por dos tipos de anotaciones:

- `requires` especifica las precondiciones sobre los parámetros del método, que la longitud objetivo sea positiva y que la cadena de entrada no sea `null`.

- `ensures` especifica las postcondiciones del método: el resultado tendrá como longitud el máximo entre el parámetro `n` y la longitud de la cadena de entrada; los primeros `(n - s.length)` caracteres del resultado serán igual al carácter de relleno `c`, y todos los caracteres a continuación serán iguales a los de la cadena de entrada.

El resto de las anotaciones ocurren en los dos ciclos utilizados en el método, proveyendo invariantes de ciclo: propiedades que son ciertas en cada ejecución del ciclo. Esto es un patrón recurrente en las herramientas de verificación para lenguajes imperativos; resulta ser que razonar automáticamente sobre ciclos es en general muy, *muy* difícil y muchas veces se requieren anotar cosas que a simple vista pueden parecer sumamente obvias. Por ejemplo, la primera invariante del primer ciclo establece que `i` será mayor o igual

que 0 y menor o igual que `pad`, lo cual en este caso es obvio pero considerando todo lo que puede ocurrir dentro de un ciclo, no se puede garantizar de manera tan fácil. La segunda invariante del primer ciclo establece que en cada ejecución del ciclo, todas las posiciones del arreglo `v` anteriores a `i` son el carácter de relleno `c`. Invariantes similares se anotan en el segundo ciclo, que copia los caracteres de la cadena de entrada a la de salida.

Podemos observar que, aunque el contenido del programa completo es por superficie tanta especificación e invariantes como código (10 líneas de especificación, 10 líneas de código), las anotaciones sirven incluso como documentación, y no complican la lectura del código de una manera considerable (en la opinión del autor). Aunque ESC/Java y OpenJML no son las herramientas de verificación más populares (dentro de un campo de por sí no muy popular), da la impresión de cumplir su cometido de ser una herramientra pragmática para detectar errores que usando sólamente Java podrían ocurrir en tiempo de ejecución.

2.1.2. Whiley

Whiley[g] es un lenguaje de programación funcional/orientado a objetos con objetivos similares a los de ESC/Java. En realidad, es similar en muchas maneras a ESC/Java, particularmente tiene una arquitectura similar a éste. El compilador de Whiley transforma código de Whiley en un lenguaje intermedio que se utiliza por una parte para generar un archivo de condiciones de verificación escrito en un lenguaje propio llamado *Whiley Assertion Language*, una variante de lógica de primer orden; y por otra parte para generar código de Java para poder ejecutar los programas en la máquina virtual de Java. Whiley utiliza su propio demostrador de teoremas en lugar de utilizar alguno de los demostradores más comúnmente utilizados (tal como Z3), aunque su arquitectura en teoría permite crear un *back end* que traduzca su código de verificación a un formato apropiado para ser analizado por otros demostradores automáticos.

Sin mucho más que agregar, presentamos `leftPad` implementado y verificado en Whiley:

```
1 function leftPad(string s, int size) -> (string result)
2 // Required padding cannot be negative
3 requires size >= |s|
4 // Returned array increased to size
```

[g] http://whiley.org/

```
5  ensures |result| == size
6  // First n elements are padding
7  ensures all { i in 0 .. (size-|s|) | result[i] == '␣'}
8  // Everything else copied from original
9  ensures all { i in 0 .. |s| | result[i+(size-|s|)] == s[i] }:
10
11     int padding = size - |s|
12     string nstr = ['␣'; size]
13     int i = 0
14
15     while i < |s|
16     where i >= 0 && |nstr| == size
17     // All elements up to i copied over
18     where all { j in 0..i | nstr[j+padding] == s[j] }
19     // Untouched s are still padding
20     where all { j in 0..padding | nstr[j] == '␣' }:
21
22         nstr[i+padding] = s[i]
23         i = i + 1
24
25     return nstr
```

Una vez más nos encontramos con un programa escrito en un lenguaje que luce familiar para la mayoría de los programadores (en este caso, la sintaxis está inspirada por Python, por controversial que esto sea), acompañado con anotaciones de pre- y post- condiciones e invariantes de ciclo. Las palabras clave `requires`, `ensures` y `where` corresponden a las vistas anteriormente en OpenJML, donde `requires` declara precondiciones para los argumentos de la función, `ensures` declara garantías sobre los resultados de la función, y `where` especifica invariantes de ciclo. En la línea 3 se especifica que la longitud dada para la nueva cadena debe ser mayor o igual a la de la entrada. En la línea 5, se asegura que la longitud de la cadena resultado será igual al parámetro `size`. En la línea 7, se asegura que los primeros `(size-|s|)` caracteres (donde `|s|` es la longitud de `s`) del resultado serán espacios. La línea 9 asegura que a partir del carácter en la posición `(size-|s|)`, el resto de los caracteres del resultado serán iguales a los de la cadena de entrada.

Las demás anotaciones ocurren en las líneas 16, 18 y 20 y son invariantes de ciclo; se hace notar que la variable de iteración siempre es mayor o igual que 0, que la variable `nstr` siempre tiene tamaño igual al parámetro `size`,

que todos los caracteres en las posiciones anteriores a la variable de iteración son espacios y que todos los caracteres desde `(size - |s|)` y hasta `i` posiciones después son iguales al `i`-ésimo carácter de la cadena de entrada.

Esta implementación es, subjetivamente, más elegante que la implementación en OpenJML pues la especificación se lee más fluidamente como parte del programa en sí, no como un comentario adicional; además, hay menos llaves y las enumeraciones sobre un rango numérico son más concisas.

Como con OpenJML, la dificultad y el potencial de verificación de este lenguaje depende del poder del demostrador de teoremas. En este respecto, en su estado actual, el proyecto OpenJML le lleva ventaja a Whiley ya que actualmente posee la capacidad de utilizar como *back end* varios de los demostradores automáticos más utilizados actualmente. A continaución examinaremos dos lenguajes que utilizan el mismo demostrador de teoremas como herramienta de verificación.

2.2. Verificación basada en SMT

2.2.1. Dafny

El último de los lenguajes con *pinta* imperativa que examinaremos es Dafny[h], creado por Rustan Leino (quien también trabajó en ESC/Modula-3 y ESC/Java[14]) trabajando en Microsoft Research. Dafny en sí es un lenguaje orientado a objetos que verifica programas traduciendo primero el código fuente a un lenguaje intermedio llamado Boogie, el cual se utiliza para generar condiciones de verificación que a su vez alimentan al solucionador de SMT Z3.

Una brevísima divagación: SMT y Z3

El problema de las **teorías de satisfacibilidad módulo** (*Satisfiability Modulo Theories* o SMT [1]) es un problema de decisión para fórmulas lógicas con respecto a diversas teorías expresadas en lógica de primer orden con igualdad. Se puede decir que el problema SMT generaliza al problema SAT, añadiendo razonamiento ecuacional, aritmética, teoría de arreglos, teoría de vectores de bits, cuantificadores y demás

[h] `https://research.microsoft.com/dafny`

otras teorías de primer orden útiles para los fines de la verificación de programas. En otras palabras, estas teorías de fondo (*background theories*) fijan la interpretación de ciertas funciones en fórmulas de lógica de primer orden para facilitar el problema de decisión de la satisficibilidad de estas fórmulas.

A los procedimientos que buscan resolver instancias del problema SMT se les llama **solucionadores SMT** (*SMT solvers*, por analogía a los *SAT solvers*). Uno de estos solucionadores, debatiblemente uno de los más prolíficos, es Z3, desarrollado en Microsoft Research. Z3 integra un solucionador SAT, un solucionador principal que se encarga de igualdades y funciones no-interpretadas, solucionadores satélite para varias teorías tales como aritmética lineal, arreglos, tuplas etc., y un motor para manejar cuantificadores.

Describir el funcionamiento de Z3 con más detalle que el párrafo anterior está más allá del alcance de este trabajo.

Al ser una herramienta de vanguardia en temas de verificación, con un claro énfasis en proveer una solución eficaz y eficiente a la clase de problemas que surgen de construir lenguajes de programación enfocados a la verificación formal, y teniendo además el apoyo de una institución de la magnitud tal como lo es Microsoft[15], Z3 se posiciona actualmente como el demostrador de teoremas automático *de facto* para el desarrollo a futuro de tecnologías de verificación nuevas y el avance de las existentes[16].

Como puede verse, el esbozo de la arquitectura de Dafny es parecido a la de ESC/Java y la de Whiley. El lenguaje intermedio Boogie es interesante por su cuenta; se describe a sí mismo como un lenguaje de verificación intermedio con el propósito de ser usado como una capa sobre la cual construir verificadores para otros lenguajes. En concordancia con esto, Dafny no es la única herramienta que lo utiliza: también lo utiliza Chalice, una herramienta de verificación de código concurrente; Spec#, un lenguaje de programación con verificación formal que funciona como extensión de C#; VCC, una herramienta para demostrar la corrección de programas concurrentes en C con anotaciones; y HAVOC, una herramienta para especificar propiedades de programas en C. Todas las herramientas anteriores son proyectos de Microsoft Research.[17].

Las diferencias no acaban ahí, como veremos a continuación al presentar la implementación de `leftPad` en Dafny:

```
1 // see https://rise4fun.com/Dafny/nbNTl
```

```
2
3  function method max(a: int, b: int): int
4  {
5      if a > b then a else b
6  }
7
8  method LeftPad(c: char, n: int, s: seq<char>) returns (v:
       seq<char>)
9  ensures |v| == max(n, |s|)
10 ensures forall i :: 0 <= i < n - |s| ==> v[i] == c
11 ensures forall i :: 0 <= i < |s| ==> v[max(n - |s|, 0)+i] == s[i]
12 {
13     var pad, i := max(n - |s|, 0), 0;
14     v := s;
15
16     while i < pad decreases pad - i
17     invariant 0 <= i <= pad
18     invariant |v| == |s| + i
19     invariant forall j :: 0 <= j < i ==> v[j] == c
20     invariant forall j :: 0 <= j < |s| ==> v[i+j] == s[j]
21     {
22         v := [c] + v;
23         i := i + 1;
24     }
25 }
```

El método `LeftPad` en sí cuenta con tres postcondiciones: que el resultado será del tamaño máximo entre la cadena de entrada y el parámetro de entrada `n`, que los primeros `(n - |s|)` caracteres de la cadena de salida serán iguales al carácter de entrada `c`; y que los caracteres siguientes serán iguales a los de la cadena de entrada `s`.

Una vez más se utiliza un ciclo, en este caso para "pegar" el carácter de relleno a una copia de la cadena de entrada por la izquierda, y este ciclo está anotado con algunas invariantes; se declara que la variable de iteración `i` se mantiene mayor o igual a 0 y menor o igual al tamaño del relleno, que la longitud del vector resultado se mantiene igual a `i` más la longitud de la cadena de entrada, que los primeros `i` caracteres de la cadena de salida se mantienen igual al carácter de relleno y que los caracteres a partir del `i`-ésimo se mantienen igual a los de la cadena de entrada. Adicionalmente, notemos que el ciclo está anotado con una **medida de terminación**, esto es,

un valor el cual se garantiza que irá decrementando hasta llegar a cero para ayudar al lenguaje a demostrar que el ciclo eventualmente va a terminar. En este caso el valor que decrementa es `(pad - i)`, ya que `i` empieza en cero y va incrementando por 1 en cada iteración, eventualmente llegará al valor de `pad`.

2.2.2. F⋆

Como mencionamos al principio de esta sección, las herramientas de verificación caen en algún punto del espacio de relación entre cobertura y esfuerzo. Hasta este punto los lenguajes de programación que hemos cubierto se encargan de manera mayormente automática de las labores de analizar y demostrar que las propiedades especificadas por el programador dentro del programa en efecto se cumplen y, por lo menos en nuestros ejemplos limitados a implementaciones de `leftPad`, no requieren de mucha asistencia de parte del programador para verificar exitosamente estas especificaciones; a lo mucho necesitan algunas invariantes de ciclo para ayudar al demostrador de teoremas a no perder la cabeza tratando de razonar al respecto.

F⋆[i] se posiciona, de varias formas, en un punto medio entre los lenguajes que hemos visto hasta ahora y los que cubriremos en la siguiente subsección: utiliza un demostrador de teoremas automático (específicamente, Z3) para intentar verificar las especificaciones indicadas por el programador, pero en caso de no ser suficiente el programador puede construir funciones y lemas auxiliares para asistir al demostrador automático, de manera similar a como los lemas y teoremas se construyen en Agda, Idris o Coq(que visitaremos más adelante). De manera similar a éstos en F⋆ las especificaciones se codifican no como pre- y post-condiciones de las funciones, sino como parte del tipo de las mismas; el sistema de tipos de F⋆ cuenta con tipos de refinamiento (del inglés *Refinement Types*) y tipos dependientes (de los cuales hablaremos más adelante), que permiten expresar especificaciones elaboradas directamente sobre los tipos de las funciones y términos que escribimos. F⋆ se encarga de verificar que los programas escritos cumplan con los tipos que dicen cumplir de manera mayormente automática, pero puede requerir asistencia por parte del programador para lograr verificar todas las especificaciones. Además, a diferencia de todos los lenguajes mostrados hasta ahora, F⋆ es un lenguaje funcional, marcadamente influenciado por ML[18]. Al igual que Dafny, el desarrollo de F⋆ es un proyecto en marcha por parte de Mi-

[i] `https://www.fstar-lang.org/`

crosoft Research, en conjunto con Inria[j], el *Institut National de Recherche en Informatique et en Automatique*.

Una brevísima divagación: Tipos de refinamiento

Los tipos de refinamiento fueron introducidos por Tim Freeman y Frank Pfenning en 1991 [7] como una extensión para el sistema de tipos de Standard ML con el propósito de poder detectar más errores en tiempo de compilación. Desde entonces, han surgido sistemas de tipos de refinamiento tanto en forma de extensiones a los sistemas de tipos de lenguajes ya establecidos (tal como LiquidHaskell para Haskell[a]) o integrados desde el principio, como en el caso de F⋆.

El mecanismo de funcionamiento de los tipos de refinamiento varía según su implementación, pero la idea subyacente es la misma: los tipos de refinamiento permiten crear subtipos de un tipo dado mediante un predicado sobre los elementos de dicho tipo. Por ejemplo, suponiendo definido el tipo `int` de los números enteros, podrían definirse los números naturales a partir de ellos con un tipo de refinamiento de la forma

```
type nat = n:int{n >= 0}
```

Al utilizar este tipo a lo largo del programa, el sistema de tipos buscará probar (en el caso de F⋆ y LiquidHaskell, de manera automática utilizando su solucionador SMT) que sus usos son correctos; por ejemplo, si una función toma como entrada un `nat` pero en algún momento del programa se intenta llamar tal función con un entero negativo, el compilador emitirá un error.

A través de este mecanismo se pueden detallar las firmas convencionales que tendrían las funciones para incluir una esepecificación más elaborada y así asegurar con asistencia del solucionador SMT que nuestra implementación es correcta respecto a su especificación.

[a] `https://ucsd-progsys.github.io/liquidhaskell-blog/`

La implementación de la función `leftPad` en F⋆ que se encuentra en el repositorio de Hillel Wayne es como sigue:

```
1 module Leftpad
2
3 open FStar.Char
```

[j] `https://www.inria.fr/`

```
4  open FStar.Seq
5
6  (* Helper *)
7  let max a b = if a > b then a else b
8
9  (* Definition *)
10 let leftpad (c:char) (n:nat) (s:seq char) : seq char =
11   let pad = max (n - length s) 0 in
12   append (create pad c) s
13
14 (* Spec and verification *)
15 let leftpad_correct (c:char) (n:nat) (s:seq char) =
16
17   let r = leftpad c n s in
18
19   (* Two abbreviations to make conditions below more legible *)
20   let ssz = length s in
21   let pad = max (n - ssz) 0 in
22
23   (* These are all statically checked for validity over all inputs
       *)
24   let _ = assert (length r = max n ssz) in
25   let _ = assert (forall (i:nat). i < n - ssz ==> index r i = c) in
26   let _ = assert (forall (i:nat). i < ssz ==>
27                     index r (pad + i) = index s i) in ()
```

Sin embargo, esta implementación no le parecerá satisfactoria al lector (como no le pareció satisfactoria al autor del presente trabajo), ya que hasta ahora todas las implementaciones aquí presentadas han integrado la implementación del programa con su especificación en la misma *unidad* (ya sea método o función), mientras que en esta versión ambas se encuentran separadas, implementación primero y especificación/verificación después. De hecho, ¡ni siquiera aparece un solo tipo de refinamiento en la implementación!

Por esta razón, el autor se dió a la tarea de aprender suficiente F⋆ para reimplementar `leftPad` con su especificación integrada directamente en el tipo de la función, y el resultado fue este:

```
1  module Leftpad
2
3  open FStar.Char
4  open FStar.Seq
```

```
5
6  let max a b = if a > b then a else b
7
8  val leftPad: (c: char) -> (n: nat) -> (s: seq char)
9      -> (res: seq char{
10         length res = max (length s) n /\
11         (forall (i: nat) . i < (n - length s)
12                           ==> index res i = c) /\
13         (forall (i: nat) . i < (length s) ==> (
14             let pad = max (n - length s) 0 in
15                     index res (i + pad) = index s i))})
16 let leftPad c n s =
17     let pad = max (n - length s) 0 in
18     append (create pad c) s
```

¡Mucho mejor! ¿No le parece?[19] En vez de poner tres *afirmaciones* (o `asserts`)[20] tras la función misma para que el demostrador las verifique, integramos las mismas propiedades directamente en el tipo de la función, en la forma de un tipo de refinamiento. En este caso estamos indicando que `leftPad` regresa una secuencia de caracteres tal que su longitud es el máximo entre la longitud de la secuencia de entrada `s` y el parámetro de entrada `n`, en donde todos los elementos del resultado anteriores a `(n - length s)` son igual al carácter de relleno `c`, y donde todos los caracteres siguientes son iguales a los correspondientes en la secuencia de entrada.

En la opinión del autor, esta es la implementación más elegante de las presentadas en esta sección[21], y de todas las implementaciones dadas en el repositorio de Hillel Wayne. En un mundo ideal, todos los programas certificados deberían verse como éste: una especificación clara y fácil de leer integrada en el tipo de un programa, que provee la solución al problema que se quiere resolver de manera sencilla; verificada con interacción mínima, de ser posible nula, por parte del programador gracias a la asistencia de un demostrador automático que nos dé la seguridad de que nuestro programa cumple con su propósito especificado.

Hoy en día, no todo programa puede verse así, y es teóricamente imposible que *todos* los programas puedan ser verificados de esta forma, pero es firme opinión del autor que esta experiencia es a lo que las herramientas de verificación formal deberían aspirar a tener si es que se espera que sean adoptadas ampliamente en el mundo del desarrollo de software.

2.3. Verificación con tipos dependientes

Los sistemas con capacidades más sofisticadas para expresar y demostrar propiedades acerca de programas son también los que requieren mayor esfuerzo de parte del desarrollador para verificar sus programas. Los sistemas como Coq e Idris se basan en sus poderosos sistemas de tipos dependientes para codificar propiedades. El proceso de verificación en estos sistemas consiste entonces en expresar las propiedades a verificar como tipos del sistema, y construir un programa consiste en encontrar, por medio de una variedad de aproximaciones, un término que al final se verificará automáticamente que tenga el tipo correspondiente.

2.3.1. Idris

Idris[k] es un lenguaje de programación funcional puro con tipos dependientes con pretensiones de ser *El Lenguaje* para desarrollo de programas verificados para utilizar en el Mundo Real[22].

A la distancia[23] Idris luce sumamente parecido a Haskell[24], pero con la peculiaridad de que lo que uno tradicionalmente considera valores (hablando en términos de la distinción tradicional entre los tipos de un lenguaje y los valores que habitan esos tipos) pueden ocurrir en las definiciones de tipos. Generar programas verificados consiste sencillamente[25] en definir un tipo de dato que capture la especificación del programa y luego, a través de una combinación de aplicación de funciones auxiliares (o "lemas") y tácticas, construir un término tal que podamos convencer al sistema de tipos que dicho término habita el tipo que construimos.

Sin mucho más preámbulo, presentamos a continuación el código fuente completo de la función leftPad, implementada y verificada simultáneamente en Idris, con una breve disección de la implementación:

```
1 import Data.Vect
2
3 -- ‘minus‘ is saturating subtraction, so this works like we want
     it to
4 eq_max : (n, k :Nat) -> maximum k n = plus (n ‘minus‘ k) k
5 eq_max  n     Z    = rewrite minusZeroRight n in
6                      rewrite plusZeroRightNeutral n in Refl
```

[k] https://www.idris-lang.org/

```
7  eq_max  Z    (S _) = Refl
8  eq_max (S n) (S k) = rewrite sym (plusSuccRightSucc (n `minus` k)
       k)
9                           in rewrite eq_max n k in Refl
10
11 -- The type here says "the result is" padded to (maximum k n),
12 -- and is padding plus the original
13 leftPad : (x : a) -> (n : Nat) -> (xs : Vect k a) ->
14 (ys : Vect (maximum k n) a **  ys = replicate (n `minus` k) x ++
       xs)
15 leftPad {k} x n xs = rewrite eq_max n k in
16             (replicate (n `minus` k) x ++ xs ** Refl)
```

El programa comienza importando el módulo Vect de la biblioteca estándar de Idris, el cual define el tipo de dato Vect (de Vector), que representa las listas de longitud fija. Este es el ejemplo canónico en las introducciones a los sistemas de tipos dependientes y por lo tanto vale la pena examinar la definición completa del tipo aunque sea brevemente:

```
1 data Vect : (len : Nat) -> (elem : Type) -> Type where
2 Nil  : Vect Z elem
3 (::) : (x : elem) -> (xs : Vect len elem) -> Vect (S len) elem
```

La definición toma forma de un GADT, o Tipo de Dato Algebráico Generalizado, e indica que el tipo (o, para ser más precisos, la familia de tipos) tiene dos parámetros: `len`, la longitud del Vect, y `elem`, el tipo de los elementos del Vect. El tipo cuenta con dos constructores: `Nil`, de tipo `Vect Z elem` (es decir, un vector de longitud `Z` (cero) con elementos de tipo `elem`) y `(::)`, que toma un `x` de tipo `elem` y un vector `xs` de tipo `Vect len elem` (osea, un vector de longitud `len` con elementos de tipo `elem`) y devuelve un vector de longitud `S len` (`len + 1`) con elementos del mismo tipo. Esto nos permite construir vectores de longitud conocida *en tiempo de compilación* (o más específicamente, durante la verificación de tipos del compilador/intérprete).

Volviendo al tema relevante, la implementación de leftPad en Idris consiste en dos funciones: un lema y la implementación en sí. El lema `eq_max` dice que el máximo de dos naturales `k, n` es igual a `(n - k) + k` (lo cual es cierto porque la resta de naturales definida en Idris es *de saturación*, es decir que la resta se define de tal forma que `0 - z = 0`. Definir la resta así tiene varias desventajas y ventajas; una de éstas es que la función se puede definir de manera total sobre los naturales). El cuerpo del lema (o de la función, ya que son la misma cosa) se conforma de los tres casos posibles para `k` y `n`:

que `n` sea 0, que `k` sea 0, o que los dos sean sucesores. En el primer caso:

```
eq_max  n     Z    = rewrite minusZeroRight n in
                     rewrite plusZeroRightNeutral n in Refl
```

se utilizan dos lemas definidos en el preludio de Idris:

```
minusZeroRight : (l: Nat) -> minus l 0 = l
plusZeroRightNeutral : (n: Nat) -> plus n 0 = n
```

`minusZeroRight` dice que `l - 0 = l` y `plusZeroRightNeutral` dice que `n + 0 = n`. Hay que probar que `maximum n 0 = (n - 0) + 0`:

- Por `plusZeroRightNeutral`, `(n - 0) + 0 = (n - 0)`.
- Por `minusZeroRight`, `n - 0 = n`.
- Por la definición de `maximum`, `maximum n 0 = n`.
- Quedando por probar que `n = n`, utilizamos el constructor de igualdad de Idris, `Refl`, el cual indica que dos términos son iguales.

En el segundo caso:

```
eq_max Z (S _) = Refl
```

no hace falta hacer más nada; ya que `maximum Z n = n` y el sistema de tipos puede ver fácilmente que `(0 - k) + k = n` por mera simplificación de las definiciones de la suma y la resta.

En el tercer caso:

```
eq_max (S n) (S k) = rewrite sym (plusSuccRightSucc (n `minus` k)
   k)
                       in rewrite eq_max n k in Refl
```

es el caso recursivo. Se utiliza un lema del preludio de Idris:

```
plusSuccRightSucc : (l : Nat) -> (r : Nat) ->
                    S (l + r) = l + (S r)
```

es decir que el sucesor de una suma es igual a la suma del izquierdo más el sucesor del derecho. En este caso se necesita probar que `maximum (S n) (S k) = ((S n) - (S k)) + (S k)`:

- Por la definición de `maximum`, `maximum (S n) (S k) = S (maximum n k)`.

- Aplicando recursivamente `maximum` (i.e. por la hipótesis de inducción), sabemos que `maximum n k = (n - k) + k`.
- Del otro lado de la igualdad, simplificando la resta obtenemos que `((S n) - (S k)) + (S k) = (n - k) + (S k)`.
- Por `plusSuccRightSucc`, `(n - k) + (S k) = S (n - k + k)`
- Por lo tanto, `S (maximum n k) = S (n - k + k)`, que es lo que queríamos demostrar.

Nótese que los pasos de reescritura y simplificación en estas demostraciones explicadas no figuran en el cuerpo del término de prueba en sí; Idris es al menos lo suficientemente poderoso para obviar muchas de ellas, resultando en demostraciones relativamente concisas (al menos para este caso en particular).

Habiendo probado este lema, podemos revisar la implementación verificada de `leftPad`, que es en realidad extraordinariamente simple:

```
1 leftpad : (x : a) -> (n : Nat) -> (xs : Vect k a) ->
2   (ys : Vect (maximum k n) a ** ys = replicate (n `minus` k) x ++
     xs)
3 leftpad {k} x n xs = rewrite eq_max n k in
4                      (replicate (n - k) x ++ xs ** Refl)
```

En esta función, un `Vect` hace las veces de cadena de caracteres. `x` es el carácter con el que se va a rellenar la cadena y `n` es la longitud final que debe tener la cadena con todo y el relleno. La función devuelve un tipo producto, una tupla con dos cosas. El primer objeto de la tupla como lo indica el tipo será un vector cuya longitud será el máximo entre `k`, la longitud del vector original, y `n`, la longitud final (es decir que si `k >= n` se regresará el vector de entrada. El segundo objeto de la tupla debe ser una prueba de que el vector que regrese la función debe ser igual a `(n - k)` veces `x`, seguido del vector de entrada; una expresión adecuada de la especificación de `leftPad`. En los sistemas de tipos dependientes a esta clase de tupla se le suele conocer como *par dependiente*, conformado por un término y una prueba de que el término cumple con alguna propiedad.

Como se puede ver, la implementación de la función es de hecho idéntica a la especificación de cómo debe verse el resultado, gracias a la magia de la programación declarativa. Por esto mismo, para demostrar que el vector resultante de concatenar `replicate (n - k) x` con `xs` en efecto es un `Vect`

`(maximum k n) a`, sólo hace falta aplicar el lema definido anteriormente (ya que el tipo de `replicate (n - k) x` es `Vect (n - k) a`, y como debe quedar claro, la longitud de la concatenación de dos vectores es la suma de sus longitudes, por lo que el tipo del `replicate (n - k) x ++ xs` es `Vect ((n - k) + k) a`).

Podemos observar también que puede llegar a necesitarse demostrar resultados que podrían considerarse sencillos sobre aritmética o similares; hasta este punto no nos habíamos encontrado con la necesidad de hacer esto. Por esta razón, la verificación en herramientas del estilo de Idris y Coq supone una inversión mayor de tiempo y esfuerzo en comparación con las herramientas que hemos explorado hasta ahora.

2.3.2. Coq

De los lenguajes presentados en esta sección, Coq[1] posee las características más sofisticadas para ser utilizado como un asistente de pruebas interactivo. No parecerá sorprendente entonces que la implementación de `leftPad` en Coq que estamos por presentar, separe la implementación y la especificación de la función; como veremos en el siguiente capítulo, esta es la forma tradicional de verificar programas en Coq. Además, examinaremos una forma de integrar la especificación con la implementación de manera similar a como lo hemos hecho hasta ahora en otros lenguajes.

De cierta forma, Coq es un descendiente directo de los primeros asistentes de pruebas como LCF; está fuertemente influenciado por ML, además de estar implementado en OCaml, y la manera más común de producir demostraciones para enunciados es usando su lenguaje de **tácticas**. Las tácticas son esencialmente comandos que transforman una meta o conclusión en submetas. La idea es aplicar una sucesión de tácticas para descomponer una meta en submetas, idealmente más simples, e ir demostrando éstas ya sea con las premisas, lemas y teoremas que ya hayamos demostrado anteriormente. A la sucesión de tácticas que se utilizan para demostrar una meta en particular (es decir, a las tácticas que conforman una sola "demostración") le solemos llamar ***script* de prueba**. Adicionalmente a las tácticas provistas por Coq se pueden definir, mediante el lenguaje `Ltac`, nuevas tácticas para automatizar procesos comunes a la hora de desarrollar *scripts* de prueba, de manera análoga a como desarrollamos funciones para abstraer funcionalidad que se repite en varias partes de un programa.

[1] `https://coq.inria.fr/`

Ya que la implementación de `leftPad` en Coq es más larga que todas las anteriores, la presentamos por partes. En primera instancia se define una función, `cutn` que divide una lista alrededor de un índice `n` y devuelve dos sublistas en un par ordenado; la primera lista es el prefijo, con los primeros `n` elementos de la lista original, y la segunda es el sufijo, con todos los elementos posteriores. Luego se demuestra que `cutn` es la operación inversa a la concatenación de listas cuando su argumento es igual a la longitud de la primera lista a concatenar:

```
Require Import Arith.
Require Import List.
Require Import Omega.

(** [cutn] breaks a list into prefix, suffix at [n]. *)
Definition cutn A n (xs : list A) := (firstn n xs, skipn n xs).

(** [cutn n] is inverse of [(++)] when [n] equals the length
of the first arg to [(++)]. *)
Lemma cutn_app:
  forall A n (xs ys : list A),
    n = length xs → cutn A n (xs ++ ys) = (xs, ys).
Proof.
 induction n; destruct xs; simpl; easy||auto.
 intros.
 unfold cutn in *.
 simpl.
 lapply (IHn xs ys).
 - congruence.
 - omega.
Qed.

Hint Rewrite app_length repeat_length : list.
Hint Resolve repeat_spec : list.
```

Hay que notar que las demostraciones no son fáciles de seguir si no se están ejecutando interactivamente. Varias de las tácticas utilizadas hacen más de una transformación sobre la meta actual, por lo que intentar seguir el desarrollo de la demostración mentalmente (o incluso en papel) se vuelve complicado, si no es que imposible. Por esta razón, no nos enfocaremos demasiado en las demostraciones en sí, sino sólo en los resultados demostrados.

A continuación, de manera similar a la implementación en Idris, se demuestra que `((m - n) + n)` es igual a `(max m n)`,

```
Lemma minus_plus_max:
  forall m n, m − n + n = max m n.
```

```
Proof.
 intros.
 destruct (le_lt_dec m n).
 - rewrite max_r; omega.
 - rewrite max_l; omega.
Qed.

Hint Resolve minus_plus_max : arith.
Hint Rewrite minus_plus_max : arith.
```

Una vez definidos estos lemas auxiliares, se define `leftPad`, y se demuestra su especificación:

```
Parameter char : Set.

Definition leftPad c n (s : list char) :=
  repeat c (n - length s) ++ s.

Definition allEqual A (xs : list A) y :=
  forall x, In x xs → x = y.

Lemma leftpad_correctness:
  forall padChar n s,
    length (leftPad padChar n s) = max n (length s) ∧
    exists m,
      let (prefix, suffix) := cutn _ m (leftPad padChar n s) in
      allEqual _ prefix padChar ∧
      suffix = s.
Proof.
 unfold leftpad, allEqual.
 split.
 - autorewrite with list arith; auto.
 - eexists.
   rewrite cutn_app; eauto with list.
Qed.
```

Vemos que la especificación está escrita en la forma de un existencial, afirmando que "existe un índice `m` tal que cortar el resultado de `leftPad` en la posición `m` resulta en dos listas: la primera contiene solamente el carácter de relleno, y la segunda que es igual a la cadena de entrada". La demostración se da en cinco breves líneas gracias a la aplicación juiciosa de tácticas como `autorewrite`, `auto` y `eauto` que automatizan gran parte del trabajo.

A pesar de que el programa con su prueba de corrección es en su conjunto bastante más largo que en todas nuestras demás implementaciones, el desarrollo interactivo de las demostraciones resulta bastante cómodo (una

vez que se ha ganado cierta familiaridad con Coq y el lenguaje de tácticas, naturalmente).

2.4. Sobre el costo de aprender una herramienta de verificación formal

Aprender a utilizar una herramienta nueva no es sencillo. En computación en particular, los lenguajes de programación tienden a estar diseñados de forma tal que sean fáciles de aprender para *alguna audiencia en particular*, si es que pretenden tener una amplia adopción en la industria. Algunos lenguajes son aptos de aprender para quienes nunca han programado antes en su vida; algunos lenguajes se dan más fácilmente a aprender para gente con una fuerte predisposición a cierto estilo de programación en particular.

Sin embargo, siendo que los lenguajes presentados en esta sección tienen como objetivos tanto el ser lenguajes de programación como ser herramientas de verificación formal, imponen un obstáculo adicional en el proceso de aprender a utilizarlos.

No sólo hace falta aprender los mecanismos de verificación que estos lenguajes nos proveen, sino también acostumbrarse al proceso de verificación en la herramienta de elección. Para los sistemas más automatizados puede hacer falta tener conocimiento del funcionamiento del sistema de verificación automática, de manera que el programador se pueda formar una intuición sobre qué cosas podrán demostrarse automáticamente y qué cosas requerirán crear nuevos lemas o invariantes para completar la verificación.

Además, la mayoría de las herramientas presentadas integran estrechamente el proceso de verificación con el de programación, por lo que puede que el programador tenga que "des-aprender" hábitos o integrar nuevas prácticas distintas a cómo las acostumbraría en un lenguaje de programación "común".

Además, debido a circunstancias como la llana *dificultad* de demostrar, habilidad no muy común entre desarrolladores de software, o la falta de bibliotecas de uso práctico en estos lenguajes para construir software *real*, se tiene un obstáculo para la amplia adopción de estas herramientas en la industria.

Notas

[11]Que hemos de notar, es Altamente Científica™.

[12]ESC/Java2: La venganza.

[13]Porque nadie quiere pasar más tiempo del absolutamente necesario leyendo código fuente en Java.

[14]Qué hombre tan prolífico.

[15]Sí sí, ya sé.

[16]Además, mucha de la investigación sobre verificación formal actualmente está siendo financiada por Microsoft, así que no es como que tengamos otra opción.

[17]Básicamente, Microsoft Research usa Boogie para *casi* todo lo que se le ocurre.

[18]No olvidemos que Microsoft tiene su propio lenguaje OCaml-esco, F#, que corre sobre la plataforma .NET

[19]¿No? Lástima.

[20]Hasta la fecha no estoy muy seguro de cómo traducir *assertion*.

[21]Por favor no se enfoque en el hecho de que de todas las implementaciones presentadas, esta es la única que de hecho *escribió el autor*

[22]Como podemos deducir a partir de su página de inicio, que lo denomina un "general purpose pure functional programming language with dependent types", a diferencia de, digamos, la página de inicio de Coq, cuya cabecera lee "The Coq Proof Assistant"; o el repositorio de GitHub de Agda, cuya descripción pone "Agda is a dependently typed programming language / interactive theorem prover".

[23]Que es como uno debe observar los lenguajes de programación con tipos dependientes salvo que el acercamiento sea estrictamente necesario, como es el caso del autor.

[24]Cosa comprensible dado que la verificación formal es lo que se meten los programadores de Haskell cuando implementar todo usando mónadas deja de hacer efecto.

[25]LaTeX no tiene un comando `sarcasmo` incluido aún, pero si lo tuviera, estaría aplicado cuatro veces a la palabra "sencillamente".

3

Construyendo programas certificados en Coq

En este capítulo nos encargaremos de ilustrar el uso de Coq como herramienta para construir programas certificados, utilizando la construcción `Program`. Ilustraremos su utilización desarrollando una estructura de datos conocida como árbol cartesiano, el cual explicaremos más adelante.

¿Por qué Coq?

La herramienta que hemos elegido para desarrollar nuestro experimento más profundo de programación certificada es Coq, siguiendo la tradición de los asistentes de prueba.

Coq es el resultado de alrededor de 30 años de investigación y desarrollo (partiendo desde 1984, con la implementación del cálculo de construcciones por Gérard Huet, Thierry Coquand [5], y a través de su extensión al cálculo de construcciones inductivas por Christine Pauline-Mohring [17]) y es una

[a] `http://compcert.inria.fr/`

de las herramientas de verificación más maduras disponibles actualmente.

Coq ha sido utilizado para desarrollar tanto aplicaciones verificadas de uso industrial (tal como es el compilador CompCert[a]), como verificaciones formales de resultados importantes como las desarrolladas por Georges Gonthier para el teorema de los cuatro colores [8] y el teorema de Feit-Thompson [9].

3.1. Árboles cartesianos

Nuestro primer paso en dirección a proveer una implementación certificada de un árbol cartesiano es, naturalmente, definir lo que es un árbol cartesiano:

Definición 3.1 *Un árbol cartesiano es un árbol binario construido a partir de una secuencia de valores de un tipo con una relación de orden definida que cumple con las siguientes propiedades:*

- *El recorrido en orden (in order) del árbol devuelve la secuencia original.*
- *El árbol cumple la propiedad de montículo mínimo (máximo) (min-heap (max-heap)): el padre de cada nodo interno tiene un valor menor (mayor) que el nodo mismo.*

Aunque usualmente nos gusta abordar las estructuras de datos de manera inductiva al utilizar lenguajes de programación funcionales, el caso de los árboles cartesianos se complica al tratar de definirlo de esta forma cuando consideramos que el árbol cartesiano depende de los elementos de la lista original. Es por eso que omitimos tratar de concebir una definición inductiva de la estructura en este trabajo.

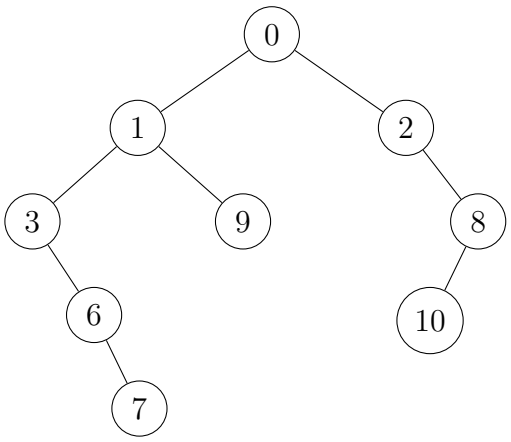
Nos limitaremos a trabajar solamente con números naturales, el valor con relación de orden por excelencia, para facilitar nuestro trabajo. Además trabajaremos solamente con montículos mínimos, ya que los montículos máximo son completamente análogos (sólo reemplace el $\leq$ con $>$ y listo).

Esta definición de árbol cartesiano puede ser algo opaca para quienes no estén familiarizados con árboles binarios, montículos o recorridos; mostraremos un ejemplo para ilustrar mejor la idea.

Supongamos que tenemos una secuencia de números, por ejemplo:

$$[3, 6, 7, 1, 9, 0, 2, 10, 8]$$

El árbol cartesiano construido a partir de esta secuencia debería ser:



Veamos ahora que éste árbol cumple las dos pripiedades mencionadas antes. Como breve recordatorio, podemos extraer el recorrido en orden de un árbol binario (básicamente conseguir una lista a partir de un árbol) siguiendo este procedimiento: empezando por el nodo raíz, tomamos recursivamente el resultado de recorrer *in order* el subárbol izquierdo, concatenamos el elemento en el nodo actual y a esa lista concatenamos el resultado de recorrer en orden el subárbol derecho.

```
1 in_order Empty = [ ]
2 in_order Node(n, left_tree, right_tree) =
3 (in_order left_tree) ++ [n] ++ (in_order right_tree)
```

Si el lector sigue el procedimiento descrito arriba empezando desde la raíz del árbol en papel y lapiz, podrá darse cuenta de que en efecto el recorrido en orden de éste resulta en la secuencia de números original.

Para observar que se cumple la propiedad de montículo, basta ver que empezando desde cualquier nodo del árbol y siguiendo la trayectoria hacia arriba hasta la raíz, siempre se sigue una secuencia decreciente de números[26].

En caso de que el lector se pregunte por qué esta estructura es nombrada "árbol cartesiano", obsérvese que si éste se dibuja adecuadamente y se proyecta sobre el plano cartesiano, resultará que recorrer el dibujo sobre el eje x (de izquierda a derecha) deberá regresar la secuencia de números a partir de la que se creó; y si se recorre sobre el eje y (de abajo hacia arriba) resulta en dibujar trayectorias con valores descendientes desde cualquier nodo hasta la raíz. Presentamos en la figura 3.1 una ilustración de este efecto.

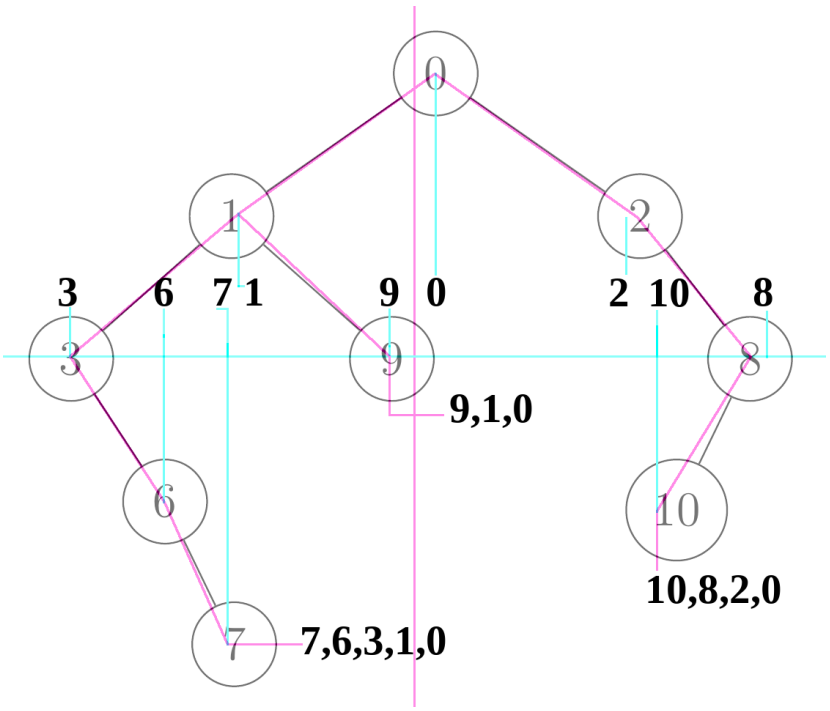


Figura 3.1: Ilustración del árbol cartesiano. Se observa que al subir sobre el eje y (rosa), se obtienen secuencias descendentes (anotadas en las hojas del árbol) y al recorrer sobre el eje x (azul) el árbol de izquierda a derecha se recupera la lista original.

3.1.1. Construyendo un árbol cartesiano

Existen varios algoritmos para construir un árbol cartesiano a partir de una secuencia de números, variando en complejidad en tiempo. Wikipedia[27] sugiere tres algoritmos que toman tiempo lineal sobre la longitud de la secuencia para construir el árbol. Desafortunadamente, todos estos algoritmos son de naturaleza atrozmente imperativa, ya que requieren mantener una semblanza de estado durante la construcción. Por supuesto, todos ellos *pueden* ser implementados en un lenguaje de programación funcional, pero en nuestro caso particular hacer un algoritmo más elaborado de lo necesario resultará también en tener que escribir demostraciones más sinuosas para probar las propiedades que nos interesan acerca de estos algoritmos.

Por lo tanto, utilizaremos un algoritmo ineficiente pero simple para construir árboles cartesianos. La idea es la siguiente: empezamos por obtener el

elemento mínimo de la lista. Este elemento será la raíz del árbol. Encontramos el prefijo y sufijo de la lista alrededor de este elemento y aplicamos recursivamente el procedimiento para encontrar los subárboles izquierdo y derecho respectivamente. Naturalmente, una lista vacía devolverá un árbol vacío.

3.2. Programación funcional en Coq

Conociendo ya la estructura que nos proponemos a implementar, así como el algoritmo con el cuál la construiremos, podemos empezar a programarla en nuestro lenguaje de programación funcional/herramienta de verificación formal predilecta. Naturalmente, en nuestro caso ésta es Coq[28].

Comenzaremos dando una implementación puramente funcional de los árboles cartesianos en Coq, sin preocuparnos (demasiado) por el momento de la verificación.

Primero definimos nuestra estructura fundamental, el árbol binario (de números naturales, aunque podríamos generalizarlos para cualquier tipo de dato sobre el que se defina una relación de orden total)[29]:

```
1  Inductive Tree: Type :=
2  | Empty: Tree
3  | Node: nat → Tree → Tree → Tree.
```

A continuación necesitamos funciones que nos ayuden a partir una lista alrededor de un elemento en particular:

```
1  (* Devuelve el prefijo de l hasta la
2  primera aparicion de n, no-inclusivo *)
3  Fixpoint up_to (l: list nat) (n: nat): list nat :=
4    match l with
5    | [ ] ⇒ [ ]
6    | x:: xs ⇒ if x =? n then [ ] else x::(up_to xs n)
7    end.
8
9  (* Devuelve el sufijo de l empezando desde
10 la primera aparicion de n, no-inclusivo *)
11 Fixpoint starting_from (l:list nat) (n: nat): list nat :=
12   match l with
13   | [ ] ⇒ [ ]
14   | x:: xs ⇒ if x =? n then xs else starting_from xs n
15   end.
```

Después de esto, necesitamos poder encontrar el elemento mínimo de una

lista (dotado de un valor default en el caso de la lista vacía para asegurar que la función sea total):

```
1  Fixpoint min (l: list nat) :=
2    match l with
3    | [ ] ⇒ 0
4    | [x] ⇒ x
5    | x:: xs ⇒ Nat.min x (min xs)
6    end.
```

Con todo esto, estamos preparados para definir la función que construye un árbol cartesiano a partir de una lista de naturales:

```
1  Program Fixpoint construct (l: list nat) {measure (length l)} : Tree :=
2    match l with
3    | [ ] ⇒ Empty
4    | _ ⇒  let m := min l _ in
5               let x := up_to l m in
6               let y := starting_from l m in
7                  Node m (construct x) (construct y)
8    end.
```

(Nota: requerimos del uso de `Program` porque nuestra función no es recursiva sobre la estructura de la lista. Además, omitimos las pruebas de corrección requeridas para que Coq acepte esta definición. Discutiremos todo esto más adelante.)

Todo esto está bien y bonito. Si fueramos programadores comunes, podríamos decir que con estas funciones tendríamos una implementación completa y utilizable de árboles cartesianos que podríamos utilizar para lo que la necesitemos[30]. Podríamos utilizarla libremente y rezar porque nuestra implementación fuera correcta e hiciera lo que se supone que hace.

Pero eso no tiene nada de divertido. Ya que estamos utilizando Coq, es nuestro deber moral utilizar su funcionalidad de verificación formal para asegurarnos de que nuestro programa en efecto construye árboles cartesianos a partir de su entrada, generando así un programa certificado que podamos utilizar con mayor confianza en su funcionamiento.

3.3. Programación a partir de especificaciones

En esta sección tomaremos la implementación funcional presentada en la sección anterior y la extenderemos para integrar las pruebas de correción de

las características que nos interesa probar acerca de cada parte. En particular, nuestro fin último será probar que la función `construct` genera un árbol con las dos propiedades que mencionamos al principio de este capítulo. Es decir, probaremos que nuestra implementación es **correcta** con respecto a esa especificación.

Posiblemente la parte más crítica y delicada del proceso de desarrollar software certificado de esta forma es encontrar una manera de expresar en nuestro lenguaje las propiedades que queremos probar de tal forma que digan lo que queremos decir y además podamos demostrarlos sin muchos rodeos[31].

Afortunadamente, este texto se escribió *después* de que el programa fue implementado y verificado, así que ya conocemos una especificación por lo menos parcialmente satisfactoria. Es importante notar que el proceso de programación-especificación-demostración no es lineal; cambios en una sección del programa certificado pueden tener repercusiones sobre otras secciones, se puede necesitar cambiar la forma en la que se especifican o se implementan las partes del programa, se modifican y agregan definiciones y lemas para facilitar pasos más adelante en otras secciones del programa, etc. Lo que presentamos a continuación es un tour lineal a través del producto terminado que *no* pretende representar este proceso.

Lo primero que definiremos será un predicado que nos permita saber cuándo un árbol es un montículo. Para esto definimos un predicado que nos dice si un número es menor a la raíz de un árbol.

```
1  Definition le_tree (n: nat) (t: Tree): Prop :=
2    match t with
3    | Empty ⇒ True
4    | Node m _ _  ⇒ n <= m
5    end.
```

Este predicado basta para saber cuándo un árbol es montículo: si sus dos subárboles son montículos y la raíz es menor que las raíces de ambos subárboles, entonces es montículo.

```
1  Inductive Heap: Tree → Prop :=
2  | Empty_Heap: Heap Empty
3  | Node_Heap: forall t1 t2 x,
4                Heap t1 →
5                Heap t2 →
6                le_tree x t1 →
7                le_tree x t2 →
8                Heap (Node x t1 t2).
```

Construimos la propiedad de `Heap` como un `Inductive` de manera similar

a como definimos el tipo `Tree` para poder hacer uso de la inducción al tener que demostrar resultados relacionados a esta propiedad.

Para poder definir la propiedad de *ser árbol cartesiano* primero necesitamos poder hablar del recorrido en orden de un árbol, por lo tanto definimos una función que dado un árbol nos devuelve su recorrido en orden en forma de lista. Para definir esta función utilizaremos `Program`.

Una brevísima divagación sobre `Program` y su funcionamiento

`Program` es una construcción de Coq introducida con el fin específico de facilitar la programación con tipos dependientes. Tradicionalmente la programación certificada en Coq se llevaría a cabo en una de dos maneras: en una, se producen por separado la implementación del programa y la traducción de su especificación a lemas y teoremas con sus respectivas demostraciones. La otra manera es desarrollar la especificación en la forma de un tipo que se usa como meta a demostrar, tal que al producir la demostración Coq puede generar un programa correspondiente al tipo (gracias a la correspondencia de Curry-Howard[a]).

`Program` surge como una alternativa nueva a estas dos opciones: permite integrar la especificación directamente en el tipo de una implementación puramente algorítimica, intentando resolver automáticamente todas las pruebas de corrección generadas a partir de la especificación que pueda, y dejando aquellas que no a demostrar por el programador. Antes de explicar el funcionamiento de este mecanismo, permitámonos explicar brevemente los tipos dependientes.

Los sistemas de tipos dependientes nos proveen de dos maneras de definir tipos en función de valores. La primera son los **tipos** Π, que se pueden entender como *familias indexadas de tipos*. El ejemplo canónico de éstos es el tipo de los vectores de longitud fija: `Vec n`, el tipo de vectores de longitud n, es una familia de tipos indexada por el parámetro `n`, un número natural que determina la longitud del vector. En Coq podemos definir este tipo usando `Inductive`:

```
Inductive Vec (A: Type) : nat → Type :=
| nil : Vec A 0
| cons : forall n, A → Vec A n → Vec A (S n).
```

La segunda forma es conocida como **tipos** Σ. Un tipo Σ representa

{x: A & P x} y un elemento de este tipo consiste en un par ordenado donde el primer elemento es un valor de tipo A y el segundo elemento es un valor de tipo P x; esto es, el tipo del segundo elemento depende del valor del primer elemento. Por esta razón, estos tipos también son conocidos a veces como *pares dependientes*

En el caso general, P puede ser de tipo A -> Type, pero si lo restringimos a tener solamente tipo A -> Prop, obtenemos un *tipo subconjunto*, el cual se expresa como {x: A | P x} y se interpreta como "*los elementos de tipo A que cumplen el predicado P*".

La extensión de Coq que implementa Program, llamada Russell, debililta el sistema de tipos de Coq en lo que concierne a los tipos subconjunto. En Coq, para tipificar correctamente un objeto de tipo subconjunto de la forma {x : U | P x}, hace falta proveer una prueba de P x. En Russell, para facilitar la integración de tipos subconjunto en la implementación del programa, se omite la necesidad de proveer estas pruebas en primera instancia, dejando "huecos" del tipo adecuado que serán posteriormente llenados con las pruebas de corrección (o en inglés, *proof obligations*[32]) que el programador desarrollará.

De cierta forma, Program puede verse como el dual de Extraction. Mientras que Extraction toma código en Coq y remueve todo aquello que existe en el universo Prop, quedándose únicamente con el algoritmo en un lenguaje objetivo dado (usualmente, OCaml), Program toma el algoritmo y su especificación expresadas en Russell y lo traduce a un término en Coq rellenando los términos de demostración faltantes con huecos para que el programador los rellene con sus correspondientes pruebas.

[a]Lectura recomendada: https://ncatlab.org/nlab/show/propositions+as+types

Nuestra definición de in_order, entonces, lucirá así:

```
1 Program Fixpoint in_order (t: Tree):
2   {l: list nat| forall x, In x l ↔ in_tree x t} :=
3   match t with
4   | Empty ⇒ [ ]
5   | Node x t1 t2 ⇒ in_order t1 ++ (x::in_order t2)
6   end.
```

Podemos ver que esta definición incluye una especificación en forma del tipo de retorno, '{l: list nat | forall x, In x l <-> in_tree x t}'. Estamos indicando que la función regresa una lista tal que un elemento está

en la lista si y sólo si está también en el árbol. Podemos notar que esta no es una especificación *completa* de lo que debe hacer la función `in_order`; es solamente una de las características que debe cumplir la lista resultado. No desarrollamos una versión más completa de la especificación de `in_order` por dos razones: primero, la función es suficientemente pequeña para poder observar a simple vista que funciona[33]; segundo, esta semi-especificación fue añadida a la función *después* de que al desarrollar el resto del programa se volvió evidente que sería útil tener esta propiedad de `in_order` demostrada. En un mundo ideal especificaríamos y demostraríamos la corrección de todo el código que escribiéramos. Pero en el mundo real nuestros minutos están contados, y el programa tiene que estar listo **ya**; tenemos que elegir qué cosas vale la pena probar y qué cosas no.

Omitimos de este escrito, por misericordia, las pruebas de corrección.

Con estos dos predicados estamos preparados para dar una definición que nos permita identificar cuándo un árbol es el árbol cartesiano de una lista en particular:

```
1 Inductive CartesianTree: (list nat) → Tree → Prop :=
2 | CartesianTreeList: forall l t,
3                     Heap t →
4                     proj1_sig (in_order t) = l →
5                     CartesianTree l t.
```

Salvo por la llamada a la función `proj1_sig`, esta definición es bastante sencilla. Como definimos anteriormente, si el árbol t es montículo y su recorrido en orden regresa la lista original l, entonces t es el árbol cartesiano de l.

> Dado que nuestra definición de `in_order` utilizando `Program` devuelve un par dependiente que contiene la lista resultante junto con la prueba de su especificación, y sólo necesitamos utilizar la lista para hacer esta definición, sacamos únicamente la lista del par dependiente usando la primera proyección sobre éste (definida como es usual, la primera proyección devuelve el primer objeto de una tupla). De ahí el nombre `proj1_sig`: es la primera proyección del par sigma.

En teoría, ya tenemos todo lo que necesitamos para (por lo menos intentar) probar que nuestra función `construct` produce el árbol cartesiano de su entrada. Sin embargo, ya que al escribir esto tenemos el beneficio de la retrospectiva, empezaremos por presentar una serie de lemas que probamos con el fin de que el *script* de prueba sea más sencillo de desarrollar y leer[34].

Comenzamos con un par de lemas acerca de las listas resultantes de las funciones `up_to` y `starting_from`:

```
Lemma up_to_len: forall l x,
     In x l → length (up_to l x) < length l.
Lemma starting_from_len: forall l x,
     In x l → length (starting_from l x) < length l.
```

Probamos que si el elemento x está en la lista l, entonces obtener el prefijo de l hasta la primera aparición de x tiene longitud estrictamente menor a la de l. Probamos igualmente el análogo para el sufijo de l a partir de la primera aparición de x.

Damos ahora una definición algo inusual para la noción de "el mínimo de una lista".

```
Inductive Min : (list nat) → nat → Prop :=
| min_singleton: forall x, Min [x] x
| min_cons: forall x x' xs, Min xs x'→
       x <= x' → Min (x::xs) x
| min_cons_tail: forall x x' xs, Min xs x'→
       x' < x → Min (x::xs) x'.
```

Como el lector podrá darse cuenta, este predicado no se ve como definiríamos el mínimo de una lista en lenguaje "matemático". Una definición más tradicional de esta propiedad podría leerse (en español): "El mínimo de una lista es aquel elemento en la lista que es menor o igual a cualquier otro elemento en la lista". Definimos la propiedad de ser el mínimo de una lista de esta forma para poder facilitarnos la vida en las demostraciones que desarrollaremos más adelante. Esta definición tiene una estructura más bien inductiva: define el mínimo de la lista en términos de la cabeza y el mínimo de la cola de dicha lista. Más adelante probaremos que si se cumple `Min l x` entonces es cierto que x es menor o igual que cualquier otro elemento de l, recuperando la noción de la definición más intuitiva mencionada anteriormente. Cabe notar adicionalmente que esta definición es relacional: recordemos que toda función de n argumentos se puede definir equivalentemente como una relación $(n+1)$-aria.

Ahora definimos una función que devuelva el mínimo (como lo acabamos de definir) de una lista:

```
Program Fixpoint min (l: list nat) (p: l <> [ ]) {measure (length l)}:
  {m | Min l m} :=
  match l with
  | [ ] ⇒ _
  | [x] ⇒ x
```

```
6     | x::( y:: xs) ⇒ Nat.min x (min (y::xs) _)
7     end.
```

Esta definición es básicamente una imagen reflejada de nuestra definición de `Min`. Si la lista tiene un solo elemento, ese elemento es el mínimo. Si tiene más de uno entonces el mínimo es el elemento más pequeño entre la cabeza y el mínimo de la cola. Especificamos en la firma de la función que la lista de entrada no debe ser vacía, pues no tiene sentido conseguir el mínimo de una lista vacía. También especificamos que el elemento de salida de esta función cumple con el predicado `Min l m`, es decir, que cumplirá con nuestra definición de ser el mínimo. (Naturalmente, Coq nos mostrará las pruebas de corrección que deben completarse para poder utilizar esta función, entre las cuales estará probar que para cualquier lista de entrada en efecto se cumple que la salida es el mínimo según `Min`). Es necesario poner el caso en donde la lista es vacía a pesar de que la especificación pide que no lo sea ya que Coq requiere que todas las funciones sean *totales* (es decir, que estén definidas para todas las entradas posibles), y no puede inferir que el caso de la lista vacía es innecesario. Sin embargo, podemos poner un hueco en donde debería estar la expresión correspondiente a ese caso y Coq se dará cuenta de que no hace falta cubrirlo gracias a la especificación.

Una observación importante acerca de esta función es que en la firma incluimos la *medida de terminación* de la función: como el cuerpo de la función no es recursivo *estructuralmente* sobre la lista[35], debemos indicarle a Coq alguna forma de saber que esta función recursiva no resultará en un ciclo infinito. Con este fin, le decimos a Coq que en cada llamada recursiva, la longitud de la lista de entrada l irá disminuyendo. Coq incluirá en las pruebas de corrección este hecho, ya que no es capaz de probarlo solo[36]. Esta característica de Coq solamente se puede emplear en funciones definidas con `Program`.

A continuación, probamos unas propiedades acerca de nuestro predicado `Min`:

```
1  Lemma min_in_list: forall l x, Min l x → In x l.
2
3  Lemma Min_is_min: forall l x y, Min l x → In y l → x<=y.
```

Primero demostramos que el elemento mínimo de la lista es un elemento de la lista[37]. Luego, demostramos que si un elemento es el mínimo de la lista, entonces es, en efecto, el mínimo de la lista[38] (es decir, que es menor o igual a todos los elementos de la lista).

Probamos ahora propiedades que facililtan trabajar con `up_to` y `starting_from`:

```
1  Lemma up_to_is_sublist:
2    forall x l m, In x (up_to l m) → In x l.
3
4  Lemma starting_from_is_sublist:
5    forall x l m, In x (starting_from l m) → In x l.
6
7  Lemma up_to_starting_from:
8    forall l x, In x l → up_to l x ++ x :: starting_from l x = l.
```

Estos lemas dicen, esencialmente, que `up_to` y `starting_from` contienen solamente elementos que se encontraban en la lista de entrada. Adicionalmente, la concatenación del `up_to` y `starting_from` de la misma lista con el mismo elemento es la lista original.

Con todo lo anterior, estamos preparados para soltar la última pieza del rompecabezas: la función `construct` pero esta vez acompañada de su especificación:

```
1  Program Fixpoint construct (l: list nat) {measure (length l)}:
2  {t: Tree | CartesianTree l t} :=
3    match l with
4    | [ ] ⇒ Empty
5    | _ ⇒ let m := min l _ in
6            let x := up_to l m in
7            let y := starting_from l m in
8                 Node m (construct x) (construct y)
9    end.
```

Como se puede observar, es exactamente la misma definición que antes, salvo que ahora especificamos que el árbol resultante cumple la propiedad de ser el árbol cartesiano de la lista de entrada, y señalamos también que en cada llamada recursiva la longitud de la lista de entrada será menor, utilizando `measure (length l)`. Naturalmente, Coq nos solicitará que desarrollemos las pruebas de corrección que muestran que esto sea verdad. Entre estas pruebas se encuentran:

- probar que el árbol vacío es el árbol cartesiano de la lista vacía;
- probar que la longitud de las listas argumento de las llamadas recursivas siempre es menor que la longitud de la entrada,
- y finalmente el paso inductivo: sabiendo que `construct` de una lista

de longitud menor a l es un árbol cartesiano, y sabiendo que l no es vacía, probar que el `construct` de l es un árbol cartesiano[39].

En este punto, podemos probar un ejemplo directamente en Coq (lo cual tarda un poco incluso para una lista pequeña, por una parte porque nuestro algoritmo no es particularmente bueno, y por otra parte porque Coq tampoco es muy rápido a la hora de correr código) y verificar que en efecto el resultado es un árbol cartesiano. También podemos extraer el programa a OCaml (o incluso a Haskell y Scheme), para tener una versión más apta de ejecutar y utilizar en alguna aplicación real, con la seguridad de que nuestra función certificada no errará en sus resultados. ¡Hurra!

3.4. Verificación formal, sin `Program`

Mencionamos anteriormente que la construcción `Program` de Coq es más bien moderna y aún se encuentra en desarrollo. En esta sección, con fines comparativos, analizaremos la implementación de una estructura de datos de manera similar a como hicimos con los árboles cartesianos, pero desarrollada sin utilizar `Program`.

La estructura es un árbol binario de búsqueda, con la cual se espera que el lector esté familiarizado. Esta implementación en particular fue extraída de una colección de estructuras y algoritmos verificados alojada por el usuario de GitHub "foreverbell"[b] [40].

El desarrollo de esta formalización inicia de manera similar a la nuestra, empezando por definir las estructuras y propiedades fundamentales para el resto del trabajo.

```
1  Inductive tree :=
2    | Leaf: tree
3    | Node: nat → tree → tree → tree.
4
5  Fixpoint tree_cmp_n (op : nat → nat → Prop)
6       (t : tree) (n : nat) : Prop :=
7    match t with
8    | Leaf ⇒ True
9    | Node v l r ⇒ op v n ∧ tree_cmp_n op l n ∧ tree_cmp_n op r n
10 end.
11
12 Definition tree_lt_n (t : tree) (n : nat) : Prop := tree_cmp_n lt t n.
```

[b] https://github.com/foreverbell/verified

```
Definition tree_gt_n (t : tree) (n : nat) : Prop := tree_cmp_n gt t n.

Fixpoint BST (t : tree) : Prop :=
  match t with
  | Leaf ⇒ True
  | Node v l r ⇒ BST l ∧ BST r ∧ tree_lt_n l v ∧ tree_gt_n r v
end.
```

Comienza definiendo la estructura de árbol binario de naturales, una función para saber si un cierto natural cumple un predicado (una función de tipo `nat ->nat ->Prop`) con todos los elementos de un árbol, definiciones para saber si un natural es mayor/menor que todos los elementos de un árbol a partir de la definición anterior, y una función que determina si un árbol es un árbol binario de búsqueda (`BST`, por sus siglas en inglés, *Binary Search Tree*). Afortunadamente, las implementaciones de todas las funciones son bastante transparentes.

Continúa definiendo las operaciones usuales sobre un árbol binario de búsqueda: insertar un elemento, verificar si un número está en el árbol, y borrar un elemento del árbol si es que existe.

```
Fixpoint insert (t : tree) (n : nat) : tree :=
  match t with
  | Leaf ⇒ Node n Leaf Leaf
  | Node v l r ⇒ match n ?= v with
                 | Eq ⇒ t
                 | Lt ⇒ Node v (insert l n) r
                 | Gt ⇒ Node v l (insert r n)
                 end
  end.

Fixpoint member (t: tree) (n : nat) : bool :=
  match t with
  | Leaf ⇒ false
  | Node v l r ⇒ match n ?= v with
                 | Eq ⇒ true
                 | Lt ⇒ member l n
                 | Gt ⇒ member r n
                 end
  end.

Fixpoint leftmost (t: tree) : option nat :=
  match t with
  | Leaf ⇒ None
  | Node v Leaf _ ⇒ Some v
  | Node _ l _ ⇒ leftmost l
```

```
 end.

Fixpoint delete_leftmost (t : tree) : tree :=
 match t with
 | Leaf ⇒ Leaf
 | Node _ Leaf r ⇒ r
 | Node v l r ⇒ Node v (delete_leftmost l) r
  end.

Fixpoint delete (t : tree) (n : nat) : tree :=
 match t with
 | Leaf ⇒ Leaf
 | Node v l r ⇒
    match n ?= v with
    | Lt ⇒ Node v (delete l n) r
    | Gt ⇒ Node v l (delete r n)
    | Eq ⇒ match l with
           | Leaf ⇒ r
           | _ ⇒ match leftmost r with
                 | Some v' ⇒ Node v' l (delete_leftmost r)
                 | None ⇒ l
                 end
           end
    end
 end.
```

Después de esto viene la parte divertida: probar las propiedades relevantes acerca de la estructura y sus operaciones.

Comenzamos con las propiedades relacionadas a la operación `insert`:

1. Después de insertar un número al árbol, dicho número es miembro del árbol.

2. Cualquier (no) miembro del árbol antes de una inserción sigue siendo (no) miembro después de la inserción (si es distinto del número insertado).

3. Si el árbol es BST, lo sigue siendo tras insertar un número.

```
Theorem member_after_eq_insert :
  forall (t : tree) (n : nat), BST t → member (insert t n) n = true.

Theorem unchanged_member_after_neq_insert :
  forall (t : tree) (n m : nat), BST t → n <> m →
    member (insert t n) m = member t m.
```

```
7
8  Theorem insert_correct :
9    forall (t : tree) (n : nat), BST t → BST (insert t n).
```

Por el otro lado, tenemos las propiedades relacionadas a la operación `delete`:

1. Un número ya no es miembro de un árbol tras ser eliminado de éste.
2. Cualquier número (no) miembro distinto al que está siendo eliminado sigue siendo (no) miembro después de la eliminación.
3. Si el árbol es BST, lo sigue siendo tras una eliminación.

```
1  Theorem not_member_after_delete :
2    forall (t : tree) (n : nat), BST t → member (delete t n) n = false.
3
4  Theorem unchanged_member_after_delete :
5    forall (t : tree) (n m : nat), BST t → n <> m →
6      member (delete t m) n = member t n.
7
8  Theorem delete_correct :
9    forall (t : tree) (n : nat), BST t → BST (delete t n).
```

Como podemos ver, estas propiedades conforman una especificación bastante buena del comportamiento de estas dos operaciones, y por lo tanto serían buenas candidatas a integrarlas a la firma de las funciones `insert` y `delete` si las quisieramos reescribir con `Program`[41].

Sin embargo, estamos siendo ligeramente deshonestos al presentar sólo los resultados principales, ya que las demostraciones de éstos emplean varios lemas y teoremas auxiliares que igualmente requieren ser demostrados para poder completar las pruebas de corrección. Presentamos a continuación sólo el enunciado (sin demostración) de los teoremas auxiliares utilizados en esta implementación:

```
1  Lemma tree_cmp_n_insert_preserve :
2   forall (op : nat → nat → Prop) (t : tree) (n0 n : nat),
3     tree_cmp_n op t n → op n0 n → tree_cmp_n op (insert t n0) n.
4
5  Corollary tree_lt_n_insert_preserve :
6   forall (t : tree) (n0 n : nat),
7     tree_lt_n t n → n0 < n → tree_lt_n (insert t n0) n.
8
9  Corollary tree_gt_n_insert_preserve :
10  forall (t : tree) (n0 n : nat),
```

```
    tree_gt_n t n → n < n0 → tree_gt_n (insert t n0) n.

Lemma tree_cmp_trans :
  forall (op : nat → nat → Prop) (t : tree) (n n0 : nat),
  (forall a b c, op a b → op b c → op a c) →
  tree_cmp_n op t n → op n n0 → tree_cmp_n op t n0.

Corollary tree_lt_trans :
  forall (t : tree) (n n0 : nat),
  tree_lt_n t n → n < n0 → tree_lt_n t n0.

Corollary tree_gt_trans :
  forall (t : tree) (n n0 : nat),
  tree_gt_n t n → n > n0 → tree_gt_n t n0.

Lemma leftmost_cmp_n :
  forall (op : nat → nat → Prop) (t : tree) (n n0 : nat),
  tree_cmp_n op t n → leftmost t = Some n0 → op n0 n.

Corollary leftmost_lt_n :
  forall (t : tree) (n n0 : nat),
  tree_lt_n t n → leftmost t = Some n0 → n0 < n.

Corollary leftmost_gt_n :
  forall (t : tree) (n n0 : nat),
  tree_gt_n t n → leftmost t = Some n0 → n0 > n.

Lemma delete_leftmost_is_delete :
  forall (t : tree) (n : nat),
  BST t → leftmost t = Some n → delete_leftmost t = delete t n.

Lemma tree_cmp_n_delete_preserve :
  forall (op : nat → nat → Prop) (t : tree) (n0 n : nat),
    BST t → tree_cmp_n op t n → tree_cmp_n op (delete t n0) n.

Corollary tree_lt_n_delete_preserve :
  forall (t : tree) (n0 n : nat),
    BST t → tree_lt_n t n → tree_lt_n (delete t n0) n.

Corollary tree_gt_n_delete_preserve :
  forall (t : tree) (n0 n : nat),
    BST t → tree_gt_n t n → tree_gt_n (delete t n0) n.

Lemma delete_leftmost_gt_n :
  forall (t : tree) (n n0 : nat),
    BST t → leftmost t = Some n0 → tree_gt_n t n →
```

```
57      tree_gt_n (delete_leftmost t) n0.
58
59  Lemma tree_lt_implies_not_member :
60    forall (t : tree) (n : nat),
61      tree_lt_n t n → member t n = true → False.
62
63  Lemma tree_gt_implies_not_member :
64    forall (t : tree) (n : nat),
65      tree_gt_n t n → member t n = true → False.
```

3.5. Y a todo esto, ¿por qué usar `Program`?

Dadas las dos maneras presentadas de desarrollar un programa verificado en Coq, es natural interrogarse qué razones existen para utilizar `Program` en lugar de sólo desarrollar la implementación y la verificación del programa independientemente la una de la otra.

Una de las razones es de naturaleza más bien filosófica que pragmática. Un precepto muy importante que se esconde detrás de la programación certificada es que un programa no está completo hasta que ya se ha verificado que la implementación cumple con la especificación dada. Muchas de las herramientas exploradas en el capítulo 2 simplemente *no permiten ejecutar el programa* si no se pudo producir una prueba de la especificación. Esto es el caso con `Program`: se puede pedir a Coq que admita las pruebas de corrección sin demostrarlas, pero se utilizarán como axiomas, lo cual suele ser mala idea a la hora de demostrar la corrección de un programa (o cualquier otra cosa)[42] y por lo tanto nuestra definición estará incompleta. Siendo este el caso, podemos argumentar que el desarrollo independiente de implementación y verificación no constituye la visión de la programación certificada tal como aspiramos a practicarla.

Si esto no parece como una ventaja, es porque no lo es. La programación certificada nos ofrece el beneficio de poder obtener software más confiable, con el costo de una inversión adicional de tiempo y esfuerzo, así como requerir alguna cantidad de conocimiento lógico-matemático y de la herramienta siendo utilizada.

Si bien `Program` puede no representar una ventaja pragmática sobre la construcción de programas con implementación y verificación independientes, sí representa una ventaja sobre la programación a partir solamente de especificaciones que algunas personas desarrollan, en donde una función se

produce como resultado de demostrar la especificación de dicha función como si fuera un teorema. Al igual que usar `Program`, este proceso nos da un programa certificado; sin embargo, no permite hacer uso del lenguaje de programación funcional embebido en Coq para desarrollar el programa ni siquiera parcialmente (ya que en este caso, la demostración *es* la implementación). Esto puede llegar a complicar tanto el desarrollo como la lectura del programa, además de que el contenido computacional de la función resultante sale del control directo del desarrollador y pasa a ser consecuencia de la forma de la demostración de la especificación.

En el contexto de estas dos opciones, `Program` representa algo parecido a un *intermedio feliz*, incorporando la especificación a la implementación al contrario de desarrollarlas independientemente, pero permitiendo escribir la implementación de una manera más sencilla que generarla puramente a partir de una demostración.

Notas

[26] Adelante, hágalo. No es como que tenga nada mejor que hacer.

[27] Sí.

[28] Advertencia: mucho código a continuación.

[29] Limitamos nuestra estructura de datos a utilizar solamente números naturales y no cualquier tipo que tenga una relación de orden porque trabajar con las relaciones de orden como están definidas en la biblioteca estándar de Coq es, hablando en términos técnicos, una pesadilla; por lo menos lo es para un usuario novicio de Coq como lo es el autor; quizás los expertos en el tema tengan una opinión diferente del asunto.

[30] ¿Para qué se utilizan los árboles cartesianos en la vida real? Buena pregunta.

[31] "Me han contado" que ese es en realidad uno de los problemas más complicados en las matemáticas en general: encontrar la manera de expresar un teorema de forma que exista una manera de demostrarlo. Después de oír eso, mi simpatía por los investigadores de las áreas de las matemáticas incrementó en 200 % y mi corazón triplicó su tamaño.

[32] Traducción engendrada por la asesora de esta tesis y dos de sus tesistas, ya que *obligaciones de prueba* sonaba ridículo.

[33] "Se puede observar a simple vista que funciona" también es mi justificación para nunca escribir ninguna clase de prueba para el código que escribo en privado.

[34] En su mayoría, son resultados que tendríamos que probar de todos modos adentro de una prueba más grande, así que la principal función de ponerlos aparte es la modularización. Además, cualquier teorema suena más importante si dices que tuviste que probar seis lemas antes de poder demostrarlo.

[35] En realidad sí lo es, pero lo es de una forma que a Coq le es imposible ver que las llamadas recursivas eventualmente terminarán.

[36] Medio inútil el Coq, ¿eh?

[37] Ya saben cómo es esto.

[38] *Ya saben cómo es esto.*

[39] Viendo esta demostración tras bastante tiempo de haberla escrito, debo darle un aplauso al yo del pasado por saber cómo descifrar esta porquería.

[40] Que quién sabe quién sea. ¿Acaso *googleé* "formally verified data structures in Coq" y le di click al primer resultado que apareció? Eso fue precisamente lo que hice.

[41] Esta implementación se deja como ejercicio al lector.

[42] Como cuando intentas hacer la tarea de Álgebra y demuestras el resultado suponiendo que el resultado es verdad.

4

Conclusiones

Vivimos en tiempos emocionantes para el campo de la verificación formal. La confluencia entre la investigación en teoría de lenguajes de programación y la evolución de las capcacidades del hardware moderno nos permiten vislumbrar un futuro que permita integrar garantías fuertes sobre las propiedades de nuestros programas, verificadas con asistencia de la computadora de manera cada vez más sencilla para el programador.

Aunque la verificación completa de toda posible especificación para todo programa concebible es teóricamente imposible, verificar formalmente *algunas* propiedades sobre nuestros programas es mejor que meramente hacer pruebas para asegurarnos que las tengan, y es colosalmente mejor que no tener ninguna clase de prueba o garantía de que nuestros programas son correctos.

En cuanto al uso de Coq como herramienta de programación certificada, se pueden sentir sus raíces como asistente de pruebas interactivo: desde el punto de vista de un *programador*, demostrar proposiciones en Coq se siente más parecido a *demostrar* en papel y lápiz, e incluso entonces esa comparación aplica sólamente para las pruebas desarrolladas como en este

trabajo; los usuarios más proficientes de Coq se esmeran mucho en automatizar las demostraciones al punto en que muchas de ellas pueden llegar a ser sumamente concisas, pero al mismo tiempo inescrutables para quienes no estén íntimamente familiarizados con Coq en general y con el programa en cuestión en particular. El autor no esperaría que esta forma de verificar programas fuera la que se hiciera popular en el futuro, ya que facilitar el proceso de demostración requiere una considerable inversión de tiempo aprendiendo técnicas de automatización de demostraciones en Coq.

Perturbar el proceso de desarrollo de software añadiendo procedimientos costosos en tiempo y que requieren conocimiento íntimo de herramientas específicas y de cierta forma de pensar en los problemas a resolver no parece ser una estrategia viable para popularizar la programación certificada. Podemos tomar por ejemplo uno de los proyectos más conocidos dentro de la esfera de la verificación formal, CompCert [a], un compilador verificado para "casi todo" el estándard de ISO C99 con el propósito de descartar la posibilidad de que el compilador incurra en errores que inserten errores en el código compilado (lo que se llama *miscompilation* o compilación errónea), de tal forma que las garantías provistas por métodos formales que analicen el código fuente no se vean invalidadas por el proceso de compilación. Xavier Leroy [10], desarrollador principal del proyecto CompCert, reporta en una plática dada en 2018 [11] un esbozo del esfuerzo invertido en el desarrollo del programa. En este indica que el proyecto está conformado por aproximadamente cien mil líneas de código en Coq, de las cuales quince porciento (al rededor de quince mil líneas) son "código fuente" y 54 porciento son *scripts* de prueba. Aunque es conceptualmente complicado (¿imposible?) comparar el esfuerzo necesario para producir un *script* de prueba contra el necesario para producir un algoritmo, debe resultar claro que de no tener el objetivo de verificar formalmente el compilador no se hubiera tenido que gastar en absoluto todo el esfuerzo invertido en producir esas *cincuenta y cuatro mil* líneas de demostraciones. Popularizar la verificación formal y la programación certificada tal como existen actualmente podría significar multiplicar substancialmente el tiempo y esfuerzo necesario para completar el desarrollo de un proyecto de software cualquiera.

Remitiéndonos de vuelta al capítulo 2, echemos un vistazo de nuevo a la implementación de `leftPad` en F⋆:

```
1 module Leftpad
```

[a] `http://compcert.inria.fr/`

```
2
3  open FStar.Char
4  open FStar.Seq
5
6  let max a b = if a > b then a else b
7
8  val leftPad:
9        (c: char) ->
10       (n: nat) ->
11       (s: seq char) ->
12       (res: seq char{
13         length res = max (length s) n /\
14         (forall (i: nat) . i < (n - length s) ==> index res i = c)
     /\
15         (forall (i: nat) . i < (length s) ==>
16           (let pad = max (n - length s) 0 in
17                       index res (i + pad) = index s i))})
18 let leftPad c n s =
19     let pad = max (n - length s) 0 in
20     append (create pad c) s
```

La razón por la que el autor nombró a esta implementación la más elegante de las presentadas en el capítulo (además de ser la única implementada por él mismo) es que, salvo por la (debatiblemente) rebuscada sintaxis para expresar la especificación del programa, el producto final no contiene rastro alguno del proceso necesario para *certificar* el programa. Para el propósito de la mayoría de la gente que desarrolla programas, el *cómo* se obtienen las garantías sobre las propiedades de los programas no es relevante; importa únicamente que en efecto se obtengan. En este respecto, los extensos *scripts* de prueba presentes en programas certificados en Coq, los lemas sin contenido computacional necesarios en Idris y las invariantes de ciclo encontradas en las implementaciones en Dafny, Whiley y OpenJML son artefactos residuales del proceso de certificación del programa; no están ahí porque los *queremos* (como sí queremos la especificación y la implementación), sino porque los *necesitamos*.

Pretendimos mostrar en el capítulo 2 una visita a través de algunas herramientas de programación certificada con el propósito de que el lector se formara un criterio acerca de su uso y potencialmente pudiera elegir profundizar en el estudio de alguna de ellas.

Aunque buena parte de este trabajo fue dedicada a ilustrar un ejemplo

más elaborado de programación certificada en Coq, es opinión del autor que si alguna forma de programación certificada ha de hacerse popular en el futuro, seguramente será del lado más automatizado del espectro, evitándole en medida de lo posible a los desarrolladores tener que desarrollar demostraciones y resultados secundarios junto con todo el conocimiento necesario para producir tales. En pocas palabras: minimizar la inversión, maximizar las ganancias. Permanece como trabajo futuro explorar de manera más profunda la especificación, implementación y verificación de otras estructuras de datos y programas con estas herramientas. (Como puede sospecharse ya, al autor le interesa de manera particular F⋆).

E incluso logrando la mínima inversión y máxima ganancia posible, la adopción amplia de una tecnología casi nunca depende de las ventajas intrínsecas que ésta ofrezca (después de todo, las meritocracias no existen); incuantificables e impredecibles factores externos juegan un papel importante en el proceso de popularización de una tecnología o un lenguaje de programación. Por ahora, parece probable que la programación certificada continúe siendo utilizada mayormente en ámbitos académicos con contadas aplicaciones en la industria. Quizás los métodos formales nunca sean adoptados en masa, llevando a una revolución del desarrollo de software que mejore dramáticamente la calidad y seguridad de nuestros programas y aplicaciones.

Pero se vale soñar.

Referencias

[1] Clark Barrett y col. «Satisfiability modulo theories». English (US). En: *Handbook of Satisfiability*. 1.ª ed. Vol. 185. Frontiers in Artificial Intelligence and Applications 1. 2009, págs. 825-885. ISBN: 9781586039295. DOI: 10.3233/978-1-58603-929-5-825.

[2] Adam Chlipala. *Certified Programming with Dependent Types : A Pragmatic Introduction to the Coq Proof Assistant*. 2013. URL: http://www.worldcat.org/isbn/9780262026659.

[3] The Idris Community. *The Idris Tutorial*. Extraído: 2019-08-21. URL: http://docs.idris-lang.org/en/latest/tutorial/index.html#tutorial-index.

[4] The Coq Development Team. *The Coq Reference Manual, version 8.9.1*. Extraído: 2019-06-10. 2019. URL: https://coq.inria.fr/distrib/current/refman/.

[5] Thierry Coquand y Gerard Huet. «The Calculus of Constructions». En: *Inf. Comput.* 76.2–3 (feb. de 1988), págs. 95-120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

[6] Cormac Flanagan y col. «Extended Static Checking for Java». En: *ACM SIGPLAN Notices* 37 (oct. de 2002). DOI: 10.1145/512529.512558.

[7] Tim Freeman y Frank Pfenning. «Refinement Types for ML». En: *SIGPLAN Not.* 26.6 (mayo de 1991), págs. 268-277. ISSN: 0362-1340. DOI: 10.1145/113446.113468. URL: http://doi.acm.org/10.1145/113446.113468.

[8] Georges Gonthier. «The Four Colour Theorem: Engineering of a Formal Proof». En: *Computer Mathematics*. Ed. por Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, págs. 333-333. ISBN: 978-3-540-87827-8.

[9] Georges Gonthier y col. «A Machine-Checked Proof of the Odd Order Theorem». En: *Interactive Theorem Proving*. Ed. por Sandrine Blazy, Christine Paulin-Mohring y David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, págs. 163-179. ISBN: 978-3-642-39634-2.

[10] Xavier Leroy. «Formal verification of a realistic compiler». En: *Communications of the ACM* 52.7 (2009), págs. 107-115. URL: http://xavierleroy.org/publi/compcert-CACM.pdf.

[11] Xavier Leroy. *Trust in compilers, code generators, and software verification tools*. 2018. URL: https://xavierleroy.org/talks/ERTS2018.pdf.

[12] Paqui Lucio. «A Tutorial on Using Dafny to Construct Verified Software». En: *Electronic Proceedings in Theoretical Computer Science* 237 (ene. de 2017), págs. 1-19. DOI: 10.4204/EPTCS.237.1.

[13] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2.ª ed. Best Practices for Developers. Redmond, WA: Microsoft Press, 2004. ISBN: 978-0-7356-1967-8.

[14] Robin Milner. *Logic for Computable Functions: Description of a Machine Implementation.* Inf. téc. Stanford, CA, USA, 1972.

[15] Leonardo de Moura y Nikolaj Bjørner. «Z3: an efficient SMT solver». En: *Tools and Algorithms for the Construction and Analysis of Systems* 4963 (abr. de 2008), págs. 337-340. DOI: 10.1007/978-3-540-78800-3_24.

[16] David J. Pearce y Lindsay Groves. «Designing a verifying compiler: Lessons learned from developing Whiley». En: *Sci. Comput. Program.* 113 (2015), págs. 191-220.

[17] Frank Pfenning y Christine Paulin-Mohring. «Inductively Defined Types in the Calculus of Constructions». En: *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics*. Berlin, Heidelberg: Springer-Verlag, 1989, págs. 209-228. ISBN: 3540973753.

[18] Dana S. Scott. «A Type-Theoretical Alternative to ISWIM, CUCH, OWHY». En: *Theor. Comput. Sci.* 121.1–2 (dic. de 1993), págs. 411-440. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90095-B. URL: https://doi.org/10.1016/0304-3975(93)90095-B.

[19] Matthieu Sozeau. «Subset Coercions in Coq». En: *Proceedings of the 2006 International Conference on Types for Proofs and Programs*. TYPES'06. Nottingham, UK: Springer-Verlag, 2007, págs. 237-252. ISBN: 3-540-74463-0, 978-3-540-74463-4. URL: http://dl.acm.org/citation.cfm?id=1789277.1789293.

[20] The F* team. *Verified programming in F*: a tutorial*. Extraído: 2019-08-10. URL: https://www.fstar-lang.org/tutorial.

[21] Hillel Wayne. *The great theorem prover showdown*. Extraído: 2019-08-13. Abr. de 2018. URL: https://www.hillelwayne.com/post/theorem-prover-showdown/.

[22] Hillel Wayne. *Why don't people use formal methods?* Extraído: 2019-07-20. Ene. de 2019. URL: https://www.hillelwayne.com/post/why-dont-people-use-formal-methods/ (visitado 21-01-2019).

[23] Wikipedia contributors. *Best coding practices — Wikipedia, The Free Encyclopedia*. Extraído 2019-08-21. 2019. URL: https://en.wikipedia.org/w/index.php?title=Best_coding_practices&oldid=903877743.