Universidad ORT Uruguay

Facultad de Ingeniería

Mecanización de verificación formal de programas en Dafny

Sebastián Colina (201066)
Guillermo de Armas (186263)

Tutor: Álvaro Tasistro

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

2024

Declaración de autoría

Nosotros, Sebastián Colina y Guillermo de Armas, afirmamos que el trabajo presentado en este documento es de nuestra autoría exclusiva y garantizamos los siguientes puntos:

- El presente estudio fue desarrollado en su totalidad durante la ejecución del proyecto correspondiente al programa de Ingeniería en Sistemas.
- En el caso de haber consultado trabajos previamente publicados por otros, hemos atribuido adecuadamente la autoría.
- Cualquier referencia a obras de terceros ha sido debidamente citada, mencionando las fuentes correspondientes. A excepción de estas citas, el contenido es completamente original.
- Hemos reconocido las ayudas recibidas durante la elaboración del trabajo.
- En los casos en que el trabajo se basa en colaboración con otros, hemos explicado de manera clara las contribuciones tanto de otros colaboradores como las nuestras.
- Ninguna parte de este trabajo ha sido publicada con anterioridad a su presentación, salvo aquellas ocasiones en las que se han proporcionado las debidas explicaciones al respecto.

Agradecimientos

Queremos expresar nuestro más sincero agradecimiento al Dr. Álvaro Tasistro, quién amablemente aceptó ser nuestro tutor de tesis en este proyecto. Su invaluable contribución ha sido fundamental para el éxito de nuestra investigación y el desarrollo del marco teórico que sustentó la elaboración del algoritmo final obtenido.

Agradecemos especialmente su dedicación al proporcionar orientación experta, valiosas opiniones y revisiones detalladas tanto de la documentación como de la parte práctica del proyecto final. Su experiencia y conocimiento han sido una guía invaluable a lo largo de todo el proceso, brindándonos una perspectiva enriquecedora que ha elevado la calidad de nuestro trabajo.

Estamos profundamente agradecidos por su apoyo constante, orientación y por compartir su experiencia con nosotros a lo largo de este desafiante pero también gratificante proceso de investigación. Su contribución ha dejado una marca significativa en nuestro trabajo y en nuestro aprendizaje.

Resumen

Se ha desarrollado en Dafny un generador de condiciones de verificación diseñado para un pequeño lenguaje imperativo. Este lenguaje incluye características como asignación múltiple, secuenciación, selección e iteración.
El propósito fundamental de este generador es derivar, a partir de un programa anotado con pre y post condiciones, así como invariantes de cada iteración, un árbol de Hoare y condiciones de verificación asociadas. Es importante destacar que, si las condiciones son demostradas, el programa resulta parcialmente correcto respecto a las pre y post condiciones dadas. Asimismo, se verifica que todo el programa compiló correctamente

Palabras clave

Métodos formales; verificación automática; invariantes; dafny; generador de condiciones de verificación;lógica de Hoare.

Índice general

1. Motivación

1.1. Introducción

En nuestras vidas como programadores, pasamos por el proceso de construir programas, probarlos y ajustarlos hasta que parecen funcionar correctamente según nuestras expectativas. Sin embargo, a menudo nos encontramos con situaciones en las que un usuario realiza ciertas acciones y se enfrenta a resultados inesperados, lo que indica un error en el flujo del programa. Luego, nos vemos obligados a reproducir este caso, investigar las razones detrás del error, corregirlo, realizar más pruebas para finalmente, lanzar la corrección.

Pero, ¿qué pasaría si existiera un método que garantizará la conformidad del programa con las especificaciones sin la necesidad de llevar a cabo pruebas exhaustivas? Esto elevaría considerablemente la certeza de que el programa satisface los requisitos establecidos, en contraste con el enfoque tradicional de pruebas. Aquí, el énfasis se desplaza hacia la definición precisa de las especificaciones en lugar de centrarse exclusivamente en los casos a probar, que, aunque son diseñados por el programador, siguen siendo susceptibles a errores. En esta tesis, al emplear métodos formales de verificación de programas y al programar la lógica subyacente, demostraremos la viabilidad y seguridad de este enfoque. Todo esto se basa en la lógica de Hoare, una herramienta esencial en la verificación formal de programas imperativos.

Pero antes de adentrarnos en los detalles de estos métodos formales, es importante comprender qué significan exactamente.

Estos métodos son técnicas y enfoques sistemáticos que utilizan principios matemáticos y lógicos para garantizar la corrección y fiabilidad de los programas de computadora. Se basan en la formalización de los requisitos y especificaciones del programa, utilizando la aplicación de dichos principios para demostrar que el programa cumple con estos requisitos.

1.2. Proceso de investigación

Este estudio combina elementos de investigación exploratoria, aplicada y documental con el objetivo de avanzar en nuestro entendimiento y desarrollo de *Verification Condition Generators* (VCGs). De esta manera, podemos contribuir al campo de la verificación formal de software, explorando nuevas oportunidades para mejorar la corrección y fiabilidad de los sistemas informáticos.

Construcción de base teórica

Después de recibir la temática de nuestro tutor, nos enfrentamos al desafío de construir una sólida base teórica. Para ello, nos sumergimos en la lectura del libro "Program Proofs" de K.R. Leino, una recomendación directa de nuestro tutor. Esta elección se debió a nuestra falta de experiencia previa con Dafny, un lenguaje de programación centrado en la verificación formal. El libro nos brindó una introducción detallada a Dafny y una comprensión fundamental de los conceptos teóricos esenciales para la verificación de programas.

Una vez adquirido este conocimiento teórico, dirigimos nuestra investigación al estudio del funcionamiento de los *Verification Condition Generators* (VCGs). Nos sumergimos en la lectura de artículos relevantes, como "Formally Validating a Practical Verification Condition Generator (extended version" de Gustav Parthasarathy, Peter Müller y Alexander J. Summers, y "A Mechanically Verified Verification Condition Generator" de Peter V. Homeier y David F. Martin. Estos recursos nos proporcionaron una visión detallada sobre la teoría y la práctica detrás de los VCGs, sentando así las bases para nuestro trabajo de investigación y desarrollo en el campo de la verificación de software.

Construcción del código

Con este conocimiento en mente, nos embarcamos en la construcción de nuestro propio *Verification Condition Generator* utilizando un modelo de lenguaje desarrollado por nosotros mismos. Con la orientación y asistencia de nuestro tutor, logramos concretar la conceptualización del VCG y llevarla a la práctica, marcando así un paso significativo en nuestra investigación y contribuyendo al avance en el campo de la verificación formal de programas imperativos.

1.3. Estructura de la tesis

La estructura de la tesis se compone de la siguiente manera:

1.3.1. Sección 2:

En esta sección, nos sumergiremos en la lógica de Hoare, explorando los fundamentos de los árboles de Hoare y sus reglas fundamentales.

1.3.2. Sección 3:

Aquí se abordará el núcleo de la investigación, donde nos adentraremos en Dafny y sus conceptos esenciales. Además, se presentará el algoritmo en el que se fundamenta esta tesis.

1.3.3. Repositorio del código

https://github.com/colinasebastian/tesis

2. Lógica de Hoare

En nuestra búsqueda por mejorar la fiabilidad y la corrección de los programas de computadora, nos adentramos en el mundo de los métodos formales de verificación de programas. Estas técnicas sistemáticas utilizan principios matemáticos y lógicos para garantizar que los programas se comporten correctamente, sin necesidad de depender únicamente de pruebas empíricas, lo que comúnmente se conocen como pruebas (*tests*). Sin embargo, mientras explorábamos este campo, nos encontramos con una herramienta fundamental que ha sido clave en la verificación formal de programas: la lógica de Hoare.

La lógica de Hoare fue desarrollada por el científico británico Tony Hoare en la década de 1960 en la cual proporciona un marco formal para especificar y verificar la corrección de programas. Su teoría se basa en la noción de tripleta de Hoare, que consiste en una precondición, un programa y una postcondición, representados como {Q} P {R} respectivamente.

Pasemos a explicar cada uno de estos tres elementos fundamentales de la teoría de Hoare:

1. Precondición: es una condición lógica que debe ser verdadera antes de la ejecución del programa. Representa el estado inicial del programa y las condiciones que deben cumplirse para que el programa sea ejecutado correctamente.

2. Programa: representa una acción que puede realizar el programa, como una asignación, una secuencia de comandos, una iteración o una selección condicional.

3. Postcondición: es también una condición lógica que debe ser verdadera después de la ejecución de dicho programa. Indica el estado final del programa y los resultados que se esperan después de ejecutar el programa.

Entonces hasta el momento sabemos que la teoría de la lógica de Hoare proporciona un marco formal para la verificación de programas, permitiendo demostrar su corrección mediante el análisis riguroso de las precondiciones y postcondiciones. En esta teoría, los árboles de Hoare se establecen como una herramienta fundamental para representar cómo se garantiza que un programa cumpla con sus especificaciones bajo ciertas condiciones iniciales.

El árbol de Hoare se construye mediante reglas de inferencia que derivan condiciones necesarias para la corrección del programa a partir de las precondiciones y postcondiciones de sus componentes principales. Estas reglas son:

1. Asignación múltiple: la regla establece que si queremos probar que una asignación $\overline{x} := \overline{e}$ cumple con una postcondición Q, necesitamos sustituir todas las ocurrencias de $\overline{x}$ en Q por $\overline{e}$.

2. Secuenciación: la regla de secuenciación indica que para probar la secuencia P; Q, primero debemos probar P y luego, usando la postcondición resultante de P como precondición para Q, probar Q.

3. Selección: la regla para la instrucción condicional *if* b *then* P *else* Q establece que para probarla, necesitamos probar P bajo la admisión de que b es verdadero y Q bajo la admisión de que b es falso. Luego, combinamos las postcondiciones resultantes usando la disyunción lógica.

4. Iteración: la regla para el bucle *while* b do P establece que para probarlo, necesitamos encontrar una invariante I tal que:

   - Si la condición b es verdadera y la invariante I es verdadera al principio del *while*, entonces P debería preservar I.
   - Si la condición b es verdadera y I se mantiene después de P, entonces la conjunción de I con la negación de b debería ser la postcondición.
   - Si la condición b es falsa al principio del bucle, entonces la postcondición es simplemente I.

En el contexto de la verificación formal de programas utilizando los árboles de Hoare, la regla de asignación múltiple es fundamental para garantizar la corrección de las operaciones de asignación en un programa. Esta regla nos proporciona un marco formal para analizar cómo una asignación múltiple afecta a las condiciones antes y después de su ejecución.

$$\overline{\{c\ [\overline{x} := \overline{e}]\}\ \overline{x} := \overline{e}\ \{c\}}$$

Este axioma de asignación múltiple establece que si queremos probar que una asignación múltiple cumple con una cierta postcondición c, debemos atravesar esa postcondición y sustituir todas las ocurrencias de las variables x: x_1,x_2,..., x_n por los correspondientes valores de las expresiones e: e_1, e_2,...,e_n.

Es importante destacar que la asignación múltiple no tiene efectos laterales, lo que significa que el único cambio que ocurre al ejecutar esta asignación es que las variables x_1,x_2,..., x_n toman los valores de las expresiones e_1, e_2,...,e_n Por lo tanto, si una postcondición c, que hace referencia a las variables x_1,x_2,..., x_n antes de la asignación, es verdadera, entonces debe seguir siendo verdadera después de la asignación, donde se reemplazan esas variables por los valores correspondientes de las expresiones.

Al construir un árbol de Hoare para una asignación múltiple, utilizamos esta regla como punto de partida. Comenzamos con la precondición del árbol, que representa las condiciones que deben cumplirse antes de la ejecución de la asignación múltiple. Luego, aplicamos la regla de asignación múltiple para derivar la postcondición del árbol, que representa las condiciones que deben mantenerse después de la ejecución

de la asignación múltiple.

La regla de asignación múltiple nos permite atravesar la postcondición y sustituir todas las ocurrencias de las variables asignadas por los valores correspondientes de las expresiones asignadas. Esto nos asegura que si la postcondición es verdadera antes de la asignación, seguirá siendo verdadera después de la misma, lo que garantiza la corrección de la operación de asignación múltiple.

Por otro lado, tenemos a la regla de secuenciación que en el ámbito de los árboles de Hoare nos permite formalizar cómo se garantiza la corrección de una secuencia de dos programas. Esta regla se expresa de la siguiente manera:

$$\frac{\{c\}\ p_1\ \{c_1\}\ \{c_1\}\ p_2\ \{c'\}}{\{c\}\ p_1\ ;\ p_2\{c'\}}$$

Esta regla nos indica que si queremos demostrar que la secuencia p_1 ; p_2 cumple con una cierta postcondición c' bajo una cierta precondición c, primero debemos demostrar que el programa p_1 cumple con una postcondición intermedia c_1 bajo la misma precondición c. Luego, utilizamos esta postcondición intermedia c_1 como precondición para demostrar que el programa p_2 cumple con la postcondición c'.

En otras palabras, para que la secuencia de programas p_1 ; p_2 sea válida y garantice la postcondición c', es necesario que se cumplan dos condiciones: que p_1 garantice una cierta condición intermedia c_1 y que p_2 garantice la postcondición final c'. Esta estructura asegura que el resultado de la ejecución de p_1 sea coherente con las condiciones necesarias para la ejecución de p_2, lo que contribuye a la corrección global de la secuencia de programas.

En lo que refiere a la estructura condicional, definimos la siguiente regla que permite formalizar cómo se garantiza dicha estructura *if*:

$$\frac{\{c\ \wedge\ b\}\ p_1\ \{c'\}\ \{c\ \wedge\ \neg b\}\ p_2\ \{c'\}}{\{c\}\ if\ b\ p_1\ p_2\{c'\}}$$

Esta regla indica que si queremos demostrar que la estructura condicional *if* b *then* P1 *else* P2 cumple con una cierta postcondición c' bajo una cierta precondición c, primero debemos demostrar que si la condición b es verdadera, entonces el programa p_1 garantiza la postcondición c', y si la condición b es falsa, entonces el programa p_2 también garantiza la postcondición c'.

Entonces si pasamos en limpio lo anteriormente mencionado, para que la estructura condicional *if* b *then* p_1 *else* p_2 sea válida y garantice la postcondición c', es necesario que se cumplan dos condiciones: que el programa p_1 garantice la post-

condición c' cuando la condición b es verdadera. Que el programa p_2 garantice la postcondición c' cuando la condición b es falsa. Con eso no aseguramos que el resultado de la ejecución de la estructura condicional sea coherente con las condiciones necesarias para la ejecución de los programas involucrados.

Por último tenemos que definir la regla de iteración (*while*) que de esta manera mediante el contexto de los árboles de Hoare podemos formalizar cómo se garantiza la correción de dicha estructura. La regla se expresa de la siguiente manera:

$$\frac{\{c \ \wedge \ b\} \ p \ \{c\}}{\{c\} \ while \ \{c\} \ b \ p \ \{c \ \wedge \ \neg b\}}$$

La regla de iteración (*while*) en los árboles de Hoare es una herramienta fundamental en la verificación formal de programas. Su desarrollo se basa en la noción de invariante, la cual desempeña un papel central en esta teoría, como se describe en el "draft"de Morgan y en el libro de K. Rustan M. Leino.

El "draft"de Morgan proporciona un enfoque formal para la verificación de programas utilizando árboles de Hoare. En este documento, se establecen reglas de inferencia que permiten demostrar la corrección de los programas. Para el bucle *while*, la regla se formula de manera que exige la existencia de un invariante que se cumple antes y después de cada iteración del bucle.

El libro de Leino complementa esta idea, explicando cómo el diseño y la verificación de *whiles* se basan en encontrar un invariante adecuado dada una postcondición. Es decir, el invariante actúa como una propiedad del programa que se mantiene constante a lo largo de las iteraciones del bucle, asegurando que se cumpla la especificación deseada.

La regla de iteración se fundamenta en esta premisa. Establece que si tenemos un invariante C que se cumple junto con la condición de la guarda b, y si el programa p garantiza que el invariante C se mantiene durante la ejecución del *while*, entonces la postcondición del bucle será que el invariante C sigue siendo válido junto con la negación de la guarda del bucle ¬b.

Entonces a la hora de diseñar un invariante y el bucle *while* para esta regla, es crucial tener en cuenta estos tres aspectos:

1. Selección de la guarda (b): la elección de la condición de la guarda (b) debe ser tal que cuando el invariante (c) se cumple y la guarda no lo hace (!b), la postcondición deseada se mantenga. En otras palabras, c ∧ ¬ b debe implicar la postcondición. Esta elección garantiza que el bucle se ejecute mientras se cumplan las condiciones necesarias para alcanzar el objetivo especificado.

2. Inicialización del invariante (c): es importante que exista una manera de inicializar el estado del programa de modo que el invariante comience a valer

antes de entrar al bucle. Esto significa que la precondición del programa debe asegurar que el invariante (c) se cumple inicialmente. Esta inicialización proporciona una base sólida para comenzar la iteración del bucle.

3. Mantenimiento del invariante: durante la ejecución del *while*, es esencial que el cuerpo del bucle mantenga el invariante. Esto implica que cada iteración del *while* no altere la validez del invariante. En otras palabras, el invariante debe mantenerse al final de cada iteración del bucle, garantizando así que se cumpla antes de que se repita la siguiente iteración.

Luego de las cuatro reglas definidas para el árbol de Hoare, nos quedaría definir una última regla conocida como la *regla de la consecuencia* y surge como apoyo para poder generar las premisas deseadas.

$$\frac{c_1 \ \rightarrow \ c_1' \ \{c_1'\} \ p \ \{c_2'\} \ c_2' \ \rightarrow \ c_2}{\{c_1\} \ p \ \{c_2\}}$$

Esta regla presentada establece una forma de razonar sobre la validez de una tripleta de Hoare, que consiste en una precondición c_1, un programa p y una postcondición c_2.
Proporciona un método para demostrar que la ejecución del programa p garantiza que la postcondición c_2 se cumple siempre que la precondición c_1 también se cumpla. Por lo tanto, permite verificar la corrección de los programas mediante el análisis de sus especificaciones.

La regla se puede entender de la siguiente manera:

1. Premisa 1 (implicación de la precondición): la primera premisa $c_1 \rightarrow c_1'$ establece que la precondición inicial c_1 implica una precondición más débil c_1'. Esto significa que cualquier estado que cumpla con c_1 también cumplirá con c_1'. La premisa implica que es válido razonar sobre la validez del programa considerando una precondición más débil.

2. Premisa 2 (validez de la ejecución del programa): la segunda premisa c_1' p c_2' indica que la ejecución del programa p desde un estado que satisface c_1' lleva a un estado final que satisface c_2'. En otras palabras, p cumple con la especificación dada por la precondición c_1' y la postcondición c_2'.

3. Premisa 3 (implicación de la postcondición): la tercera premisa $c_2' \rightarrow c_2$ establece que la postcondición más fuerte c_2 es implicada por la postcondición final c_2'. Esto significa que cualquier estado que cumpla con c_2' también cumplirá con c_2. La premisa implica que el resultado de la ejecución del programa satisface la postcondición deseada c_2.

4. Conclusión (validez de la tripleta de Hoare): la regla concluye que si se cumplen las tres premisas anteriores, entonces la tripleta de Hoare c_1 p c_2 es válida, lo que significa que el programa p garantiza que la postcondición c_2 se cumplirá siempre que la precondición c_1 también se cumpla.

Esta regla es fundamental porque permite la flexibilidad necesaria para adaptar las precondiciones y postcondiciones a medida que evoluciona el proceso de diseño y verificación de programas. La importancia radica en su capacidad para conectar diferentes partes del razonamiento lógico en el contexto de la verificación formal de programas. Por ejemplo, al diseñar el invariante para un *while* en la regla de iteración, la regla de la consecuencia permite establecer una conexión coherente entre la precondición inicial, el invariante del bucle y la postcondición final. Es esencialmente la puerta de entrada para la aparición de las condiciones de verificación, ya que establece cómo se pueden fortalecer tanto la precondición como la postcondición sin afectar la corrección del programa.

Luego de haber explorado en detalle la lógica de Hoare y su papel fundamental en la verificación formal de programas, nos urge y consideramos crucial el hecho de abordar los conceptos de corrección parcial y corrección total. La lógica de Hoare, en su formulación original con la tripleta {Q} P {R}, se centra en garantizar la corrección parcial de los programas. Esta relación garantiza que si el programa P se ejecuta desde un estado que satisface la precondición Q, entonces, bajo ciertas condiciones, el programa terminará en un estado que cumple con la postcondición R. Sin embargo, la lógica de Hoare no aborda directamente la cuestión de si el programa efectivamente termina para todos los *inputs*, lo que da lugar a la distinción entre corrección parcial y corrección total.

La corrección parcial, inherente a la lógica de Hoare en su forma estándar, asegura que si el programa termina su ejecución, lo hará de acuerdo con la especificación dada por la postcondición R. En otras palabras, si el programa P se inicia desde un estado que satisface la precondición Q y finaliza su ejecución, entonces el estado final cumple con la postcondición R.

Por otro lado, la corrección total va más allá al garantizar no solo que el programa produzca un resultado correcto, sino también que termine su ejecución para todos los posibles casos de entrada. En el contexto de la lógica de Hoare, esto implicaría que la tripleta {Q} P {R} no solo garantiza que si el programa P se inicia en un estado que satisface la precondición Q, entonces terminará en un estado que cumple con la postcondición R, sino que además asegura que el programa P efectivamente terminará su ejecución para cualquier entrada válida.

Sin embargo, abordar la corrección total introduce desafíos adicionales, especialmente en lo que respecta a la demostración de la terminación del programa. Para garantizar la terminación, es necesario demostrar que cada *while* o estructura iterativa dentro del programa eventualmente terminará su ejecución. Esto a menudo implica definir funciones de las variables del programa que estén acotadas inferiormente y que decrezcan en cada iteración del bucle. La complejidad radica en la

elección adecuada de estas funciones y en demostrar que efectivamente garantizan la terminación del programa para todas las entradas válidas.

En nuestro enfoque, optamos por centrarnos en la *corrección parcial* debido a su naturaleza más amigable (manejable) y no al desafío adicional que representa la garantía de la terminación total del programa. Al omitir la consideración de la terminación total, nuestro trabajo se enfoca en garantizar que, si el programa se ejecuta, lo hará correctamente según lo especificado por la postcondición, dejando el análisis de la terminación total para futuras investigaciones.

Como ya mencionamos anteriormente, el planteamiento teórico de la lógica de Hoare proporciona un marco sólido para la verificación formal de programas, estableciendo reglas y principios que garantizan la corrección parcial de un programa en relación con una precondición y una postcondición dadas. Sin embargo, al aplicar esta teoría en la práctica, surge la cuestión de cómo los programadores abordan la tarea de garantizar la corrección de sus programas.

Cuando nos sumergimos en la implementación práctica de la lógica de Hoare, se hace evidente que el proceso de derivar un árbol de Hoare completo para cada programa puede resultar excesivamente laborioso y complejo. Si bien la teoría nos indica la necesidad de demostrar formalmente que un programa cumple con sus especificaciones, en la realidad, los programadores a menudo adoptan un enfoque más práctico y eficiente.

En lugar de seguir rigurosamente cada paso de la teoría de Hoare, los programadores, si son disciplinados en su enfoque, tienden a centrarse en el diseño de invariantes para las estructuras iterativas, como los bucles. Estos invariantes actúan como propiedades que se mantienen constantes a lo largo de las iteraciones del *while*, garantizando que el programa cumpla con sus especificaciones incluso en presencia de estructuras repetitivas.

El proceso de diseño de invariantes implica varios aspectos cruciales, muchos de los cuales están relacionados directamente con los principios establecidos por la lógica de Hoare. Primero, el programador se centra en la inicialización del invariante, asegurándose de que este se cumpla antes de que comience la iteración del *while*. Luego, se analiza la condición de terminación del mismo, garantizando que este se detenga en algún punto y no entre en un bucle infinito. Finalmente, se examina el cuerpo del *while* para verificar que preserva el invariante en cada iteración, lo que garantiza la coherencia del comportamiento del programa a lo largo del tiempo.

Para eso, veamos un ejemplo más práctico de cómo es que el programador diseña el invariante; pero antes definamos que es un programa mínimamente anotado ya que en otras palabras es lo que al programador termina realizando en la práctica.

Un *programa mínimamente anotado*, denotado como S, consta de una precondición C1 y una postcondición C2 y es el objeto de estudio en este contexto. Estas condiciones definen los requisitos que deben cumplirse antes y después de la ejecución del programa. Por ejemplo, en un algoritmo de división, la precondición podría establecer que los números involucrados deben ser positivos, mientras que la postcondición especifica que el resultado debe ser correcto y dentro de un rango válido.

Ahora veamos un ejemplo ilustrativo:

```
(Precondición) {m >= 0 && n > 0}
q, r := 0, m
while {m = q * n + r && r >= 0}
    (r >= n)
    q, r := q + 1, r - n
(Postcondición) {m = q * n + r && 0 <= r < n}
```

En este ejemplo, se específica una división utilizando suma y resta para calcular los valores de q y r al dividir m por n. Tomemos en cuenta que tanto los booleanos como las expresiones se consideran condiciones. Las condiciones se definen tanto para la precondición (*require*) como para la postcondición (*ensure*). Dentro del bucle *while*, se establece un invariante que debe mantenerse en cada iteración.

La especificación del programa se realiza mediante la definición de precondiciones y postcondiciones:

- La precondición (*require*) establece las condiciones que deben cumplirse antes de la ejecución del programa. En este caso, se requiere que m sea mayor o igual a 0 y que n sea mayor que 0, lo que asegura que el programa se ejecute correctamente sin generar errores debido a valores no válidos.
- La postcondición (*ensure*) especifica las garantías que deben cumplirse después de la ejecución del programa. Aquí, se establece que al finalizar la ejecución, m debe ser igual a q*n + r y que r debe estar en el rango de 0 a n-1. Esta postcondición asegura que el programa cumple con su objetivo y produce el resultado esperado.

En lo que refiere a la secuencia de ejecución del programa:

- Se inicia asignando a las variables q y r los valores 0 y m, respectivamente. Esta asignación prepara las variables para el proceso de cálculo dentro del bucle *while*.
- Dentro de dicho bucle, se establece un invariante que debe mantenerse en cada iteración. Este invariante garantiza que la relación entre m, q, y r se preserve en cada paso del *while*, lo que es crucial para asegurar la corrección del algoritmo.
- La condición del *while* se define como r >= n, lo que indica que el bucle debe continuar ejecutándose mientras r sea mayor o igual que n. Esta condición controla la ejecución del bucle y garantiza que se alcance el resultado deseado.

Con esta mínima notación, se obtiene una prueba de que el programa es correcto, ya que cumple con las condiciones preestablecidas y produce el resultado esperado de manera consistente. También es importante destacar que en verdad no tiene todo el árbol de Hoare generado con las afirmaciones intermedias.

En conclusión, las condiciones pre y post son esenciales para especificar el comportamiento y las garantías de un programa. *Al escribir las variables matemáticas como expresiones y utilizar booleanos como casos atómicos de condiciones*, se simplifica el proceso de especificación y validación del programa, lo que contribuye a una mayor comprensión y robustez del código.

Es de suma importancia destacar que las iteraciones (*while*) están creadas con un invariante como parte inherente de su diseño. Por ende, es fundamental que el programador tenga la habilidad de diseñar dicho invariante y comprenda las tres partes fundamentales en las que se apoya: inicialización, mantenimiento y terminación. Es imperativo hacer explícito el invariante, ya que esto forma parte de la disciplina de programación y constituye la forma en que se estructura y garantiza la correcta ejecución del bucle. Tanto la lógica de Hoare como Dafny, por ejemplo, manejan este concepto de manera implícita: Hoare lo incorpora en su regla de iteración, mientras que Dafny utiliza la palabra clave *invariant* para su definición.

A partir de este punto, podemos comprender que el *Verification Condition Generator* (VCG) se encarga de tomar el programa mínimamente anotado provisto por el programador y generar el árbol de Hoare. Este proceso es esencial, ya que el árbol de Hoare, al satisfacer los principios de la lógica de Hoare, nos proporciona la certeza de la corrección del programa. En otras palabras, el objetivo es automatizar, mediante métodos formales de verificación de programas, la demostración de la corrección parcial del programa, liberando al programador de esta tarea y que simplemente se centre en crear el programa mínimamente anotado.

El VCG que construiremos a continuación será el resultado de un proceso de investigación que combina elementos de investigación exploratoria, aplicada y documental. A lo largo de este proceso, nos sumergiremos en la teoría y la práctica de los Generadores de Condiciones de Verificación (VCGs) y otros aspectos fundamentales de la verificación formal.

3. Dafny

3.1. Algoritmo VCG

Como se mencionó anteriormente, nuestro enfoque se centra en la construcción del *Verification Condition Generator* (VCG). Este algoritmo tiene como propósito validar la corrección de un programa, garantizando que esté adecuadamente codificado para cumplir con las precondiciones y postcondiciones establecidas. Es importante destacar que, en este contexto, no se considera la sintaxis ni el control de tipos, ya que se presupone que estos aspectos han sido validados previamente.

Existen dos posibles soluciones para poder crear este algoritmo VCG. Ambas parten del mismo objetivo que es a partir de un programa mínimamente anotado llegar a un árbol de Hoare.

La primera solución consiste en que dicho árbol sea explícitamente representado como un datatype de los árboles de Hoare. Sin embargo, debido a la limitación de información en el sistema de tipos de Dafny, necesitamos definir cuándo estos árboles de Hoare son correctos. Surge así la necesidad de definir el predicado o función *correct*, que vincula el árbol de Hoare con ciertas condiciones para garantizar su corrección.

Una alternativa más simple consiste en no considerar la construcción explícita del árbol de Hoare. En este enfoque, definimos un predicado (función booleana) recursivo en Dafny, denominado Hoare, que verifica si un programa anotado con una precondición y una postcondición cumple con la lógica de Hoare. Este predicado Hoare se convierte en nuestra representación de la lógica de Hoare en Dafny.

Ambas soluciones tienen sus ventajas y desventajas. La primera solución proporciona una formalización en profundidad, ya que el árbol de Hoare se expresa explícitamente como un datatype con la condición de corrección incorporada. Por otro lado, la segunda solución, aunque más simple, se clasifica como una formalización superficial. En este caso, no se necesita la construcción explícita del árbol de Hoare; solo se verifica la corrección del programa con respecto a la lógica de Hoare utilizando el predicado Hoare.

Para llevar a cabo la transformación de un programa anotado a la lógica de Hoare, desarrollaremos el algoritmo VCG en su máxima amplitud, como se mencionó anteriormente. Inicialmente, lo bosquejaremos en Haskell y luego lo traduciremos a Dafny. Ya que la segunda solución es más sencilla, remarcaremos los cambios realizados frente al siguiente esbozado.

Antes de definir los tipos de datos, es importante resaltar que la técnica del VCG consiste en proceder o “empujar” desde la postcondición hacia arriba, aplicando las

reglas definidas para el Árbol de Hoare.

Definamos los tipos de datos para representar la gramática de un lenguaje de gramática imperativo simple tanto para p (*program*) y s (*Statement*).

```
p ::=     x̄:= ē (Asignación Múltiple)
        | p1 ; p2 (Secuenciación)
        | if b p1 p2 (Selección)
        | while {c} b p (Iteración)

s ::= {c1} p {c2}
```

Hagamos un paréntesis para explicar que serían cada una de estas cuatro estructuras:

1. **Asignación Múltiple**: en esta operación, múltiples variables reciben valores correspondientes a múltiples expresiones. Esto se logra proporcionando una lista de variables y una lista de expresiones, donde cada variable recibe el valor correspondiente de la expresión asociada. Por ejemplo, en la expresión x, y := 10, 20, tanto x como y reciben los valores 10 y 20 respectivamente.

2. **Secuenciación**: permite la ejecución ordenada de varias instrucciones. En esta estructura, un programa se ejecuta inmediatamente después del otro, proporcionando una forma de controlar el flujo de ejecución. Por ejemplo, $p_1; p_2$ indica que primero se ejecuta p_1 y luego se ejecuta p_2.

3. **Selección**: esta estructura representa una selección condicional, permite la ejecución condicional de programas. Un booleano específica la condición que debe cumplirse para determinar si se ejecuta el programa p_1 o el programa p_2. Si la condición es verdadera, se ejecuta p_1; de lo contrario, se ejecuta p_2. Por ejemplo, *if* b p_1 p_2 indica que si la condición b es verdadera, se ejecuta p_1, de lo contrario se ejecuta p_2.

4. **Iteración**: el bucle *while* permite la repetición de un conjunto de instrucciones mientras se cumpla una condición específica, conocida como invariante. La estructura del *while* consiste en la condición invariante, un booleano que actúa como la condición de salida del bucle, y el cuerpo del bucle, que es un programa. Por ejemplo si consideremos el siguiente código que calcula el factorial de un número utilizando el *while*.

   ```
   while (n > 1) True
       fact = fact * n
       n = n - 1
   ```

   Podemos apreciar los distintos puntos:

   - La condición invariante es n > 1.
   - El booleano de la condición de salida es *True*, lo que significa que el bucle se ejecutará hasta que la condición invariante ya no se cumpla.

- El programa cuerpo del bucle es fact = fact * n y n = n - 1, que se ejecutará repetidamente mientras se cumpla la condición invariante.

Para concluir, nuestro objetivo es traducir un programa mínimamente anotado, denotado como c_1 p c_2, a la lógica de Hoare para obtener un árbol de Hoare correspondiente.

$$S \rightarrow (h\ X\ [c])\ \textit{siendo}\ S\ ::=\ \{c_1\}\ p\ \{c_2\}$$

Durante este proceso, se generarán una serie de implicaciones que conformarán las condiciones de verificación ([c]). La demostración de estas condiciones puede ser llevada a cabo mediante diferentes métodos, como la implementación de un demostrador automático (por ejemplo, Z3, aunque con ciertas limitaciones), u otras técnicas disponibles. Sin embargo, es importante tener en cuenta que la resolución de estas implicaciones excede el alcance de nuestra investigación, *por lo que asumimos que alguien se encargará de abordar este aspecto en profundidad.*

Partiendo de las reglas definidas para el árbol de Hoare anteriormente, apliquémoslas al ejemplo que trabajamos previamente para ver cómo se emplean:

```
(Precondición) {m >= 0 \&\& n > 0}

 3: (m = q * n + r && r >= 0) [q, r := 0, m]
    q, r := 0, m

 m = q * n + r && r >= 0
while   {m = q * n + r && r >= 0}
    (r >= n)

     2: m = q * n + r && r >= 0 && r >= n
    q, r := q + 1, r - n
     m = q * n + r && r >= 0

 1: m = q * n + r && r >= 0 && not(r >= n)

(Postcondición) {m = q * n + r && 0 <= r < n}
```

Comenzando desde el ejemplo proporcionado, aplicamos las reglas del árbol de Hoare para demostrar la validez del programa. Recordemos que las reglas del árbol de Hoare se aplican de manera ascendente, comenzando desde la postcondición y trabajando hacia arriba hasta la precondición. Observemos cómo se aplican estas reglas:

1. Iteración (regla del *while*):

- Comenzamos con la postcondición después del *while*, que establece que m = q * n + r y $0 \leq r < n$. Esta es la condición que queremos probar al finalizar el bucle.
- Aplicamos la regla del *while*, que nos dice que si podemos demostrar que la condición de la guarda ($r \geq n$) junto con la invariante (m = q * n + r y $r \geq 0$) implica la postcondición, entonces el bucle *while* es válido
- Luego, observamos la guarda del bucle y la invariante para derivar una nueva condición de verificación, marcada como condición de verificación 2.
- Después de ejecutar el cuerpo del bucle, necesitamos demostrar que la invariante se mantiene. Esto nos lleva a la condición de verificación 1, que corresponde a la invariante antes de la ejecución del bucle.

2. - Finalmente, comenzamos con la precondición del programa, que establece que $m \geq 0$ y $n > 0$.
   - Aplicamos la regla de asignación múltiple para demostrar que los valores iniciales de q y r cumplen con la invariante.
   - Esto nos lleva a la condición de verificación 1, que representa la invariante antes de entrar al bucle.

Por lo tanto, en cada paso, trabajamos desde la postcondición hasta la precondición, aplicando las reglas correspondientes y derivando las condiciones de verificación necesarias para demostrar la validez del programa. Estas condiciones de verificación se pueden apreciar en azul en el código de ejemplo, indicando las etapas del razonamiento que conducen a la validación del programa.

3.1.1. Bosquejo del algoritmo VCG

Partiendo de la definición de las reglas del árbol de Hoare, procedemos a representar dichos árboles y el bosquejo del VCG en Haskell, siempre teniendo en cuenta el objetivo final de crear un algoritmo en Dafny para verificar programas.

El árbol de Hoare se define de la siguiente manera:

$$S \ ::= \ (\overline{x},\ \overline{e},\ c) \mid ;\ (h_1,\ h_2) \mid if\ (b,\ h_1,\ h_2,\ c) \mid while\ (c,\ b\ h)$$

Donde:

- El constructor de Hoare para la asignación múltiple, representado por :=, toma tres parámetros: la lista de variables que se asignan ($\overline{x}$), la lista de expresiones que se asignan a esas variables ($\overline{e}$), y la postcondición (c). Esta estructura denota la ejecución de una serie de asignaciones en paralelo seguida de la verificación de la postcondición.

- El operador ; representa la secuenciación, donde h_1 y h_2 son dos árboles de Hoare que se ejecutan secuencialmente. Esto implica que primero se ejecuta h_1 y luego h_2, manteniendo la coherencia de las condiciones entre ellos.
- El bloque *if* (selección) representa una selección condicional. Toma una condición booleana (b) y dos árboles h_1 y h_2 que representan las ramas verdadera y falsa, respectivamente. Además, se especifica una postcondición (c) que se verificará al finalizar la ejecución del bloque *if*.
- Como ya mencionamos anteriormente, el bucle *while* representa la iteración condicional. Tiene tres parámetros: el invariante (c), la condición de salida del bucle (b), y el árbol de Hoare (h) que se ejecuta mientras la condición se cumpla. El invariante garantiza que la condición se mantenga en cada iteración del bucle, y la verificación de la postcondición se realiza al salir del *while*.

Es importante destacar que el árbol de Hoare proporciona una estructura formal para la verificación de programas, donde cada nodo representa una acción o estructura de control del programa, y las condiciones asociadas garantizan la validez de la ejecución según las especificaciones dadas. Este enfoque permite analizar y demostrar la corrección de programas de manera sistemática y rigurosa.

Luego de haber definido el esqueleto, procedemos a establecer la condición de corrección basándonos en las reglas definidas para el árbol de Hoare en la parte anterior. Por lo tanto, definimos la función correct de la siguiente manera:

$$correct : \ h \ \rightarrow \ (S, \ [c])$$

Para cada árbol h, obtenemos una pareja que consiste en una conclusión S y las condiciones que deben cumplirse para que esta conclusión sea correcta.

Ahora describamos cada uno de los casos del bosquejo para la función *correct*:

1. $$correct : \ (:= \ (\overline{x}, \ \overline{e}, \ c) \ = \ (c[\overline{x} \ := \ \overline{e}] \ \overline{x} \ := \ \overline{e}\{c\}, \ [\,])$$

   En este caso, al tratarse de una asignación, no tenemos condiciones de verificación adicionales, por lo que la lista de condiciones es vacía [ ].

2. $$\begin{aligned} correct : \ (; \ (h_1, \ h_2) \ = \ let \ (\{c\} \ p_1 \ \{c_1\}, \ cs_1) \ = \ correct \ h_1 \\ (\{c_2\} \ p_2 \ \{c'\}, \ cs_2) \ = correct \ h_2 \\ in \ (\{c\} \ p_1 \ ; \ p_2 \ \{c'\}, \ cs_1 \ ++ cs_2 \ ++ [c_1 \ \rightarrow \ c_2]) \end{aligned}$$

Para la secuenciación, aplicamos recursivamente la función *correct* a ambos subárboles h_1 y h_2. Luego, combinamos las condiciones de verificación obtenidas, asegurándonos de que la postcondición de h_1 implique la precondición de h_2, lo que se representa como $c_1 \rightarrow c_2$.

3.

$$correct: \ (if \ (b, \ h_1, \ h_2, \ c') \ = \ let \ (\{c_1\} \ p_1 \ \{c_1'\}, \ cs_1) \ = \ correct \ h_1$$

$$(\{c_2\} \ p_2 \ \{c_2'\}, \ cs_2) \ = \ correct \ h_2$$

$$in \ (\{b \ \rightarrow \ c_1 \ \&\& \ \neg b \ \rightarrow \ c_2\} \ if \ b \ p_1 \ p_2 \ \{c'\}, \ cs_1 \ ++ \ cs_2 \ ++ \ [c_1 \ \rightarrow \ c', \ c_2 \rightarrow \ c'])$$

Para la estructura de selección, aplicamos recursivamente la función *correct* a los subárboles h_1 y h_2, obteniendo las condiciones de verificación respectivas c_1 y c_2. Luego, combinamos estas condiciones en una única condición compuesta que asegure que c_1 se cumple si la condición b es verdadera y c_2 se cumple si b es falsa. Además, agregamos condiciones de implicación entre las postcondiciones de los subárboles y la postcondición del bloque *if*.

4.

$$correct: \ (while \ (c, \ b, \ h) \ = \ let \ (\{c_1\} \ p \ \{c_1'\}, \ cs) \ = \ correct \ h$$

$$in \ (\{c\} \ while \ c \ b \ p \ \{c \ \wedge \ \neg b\}, \ cs \ + + \ [c \ \wedge \ b \ \rightarrow \ c_1, \ c_1' \rightarrow \ c])$$

Para la iteración (*while*), aplicamos recursivamente la función *correct* al subárbol h para obtener las condiciones de verificación c_1. Luego, combinamos estas condiciones con otras condiciones que garantizan que el invariante se cumple al inicio del bucle y que la postcondición del bucle implica la precondición del mismo, garantizando la terminación del bucle.

Como último paso en nuestro proceso, definimos el generador VCG (*Verification Condition Generator*) que transforma una especificación de programa S en un árbol de Hoare h junto con una lista de condiciones c que deben cumplirse para garantizar la validez del programa.

$$VCG: \ S \ \rightarrow \ h \ X \ [c]$$

$$VCG \ (\{c_1\} \ p \ \{c_2\}) \ = \ VCG' \ (p, \ c_2)$$

El VCG se basa en la técnica de empujar la postcondición hacia arriba, por lo que definimos una versión simplificada, VCG' que se centra únicamente en la postcondición.

$$VCG: \ p \ X \ c \ \rightarrow \ c \ X \ h \ Xs \ [c]$$

1. **Asignación múltiple**:

$$VCG' : (\overline{x},\ :=\ \overline{e},\ c)\ =\ (c[\overline{x}\ :=\ \overline{e}],\ \ :=\ (\overline{x},\ \overline{e},\ c),\ [\,])$$

La función VCG' para la asignación toma una lista de variables $\overline{x}$, una lista de expresiones $\overline{e}$ y una postcondición c. Primero, actualiza la postcondición c sustituyendo las variables $\overline{x}$ por las expresiones $\overline{e}$. Luego, construye el árbol de Hoare correspondiente para la asignación y devuelve una lista vacía de condiciones de verificación, ya que la asignación no introduce ninguna nueva condición.

2. **Secuenciación**:

$$\begin{gathered} VCG' : (p_1\ ;\ p_2,\ c')\ =\ let\ (c_1,\ h_1,\ cs_1)\ =\ VCG'\ (p_1,\ c') \\ (c_2,\ h_2,\ cs_2)\ = VCG'\ (p_2,\ c_2) \\ in(c_1,\ ;\ (h_1,\ h_2),\ cs_1\ +\!+\ cs_2) \end{gathered}$$

Para la secuenciación, aplicamos VCG' recursivamente a p_1 con la postcondición c'para obtener c_1, h_1 y cs_1 y luego a p_2 con la postcondición c_2 para obtener c_2, h_2 y cs_2. Por último, combinamos las condiciones de verificación cs_1, cs_2 construimos el árbol de Hoare correspondiente a la secuenciación.

3. **Selección condicional (if)**:

$$\begin{gathered} VCG' : (if\ b\ p_1\ p_2,\ c')\ =\ let\ (c_1,\ h_1,\ cs_1)\ =\ VCG'\ (p_1,\ c') \\ (c_2,\ h_2,\ cs_2)\ = VCG'\ (p_2,\ c') \\ in((b\ \rightarrow\ c_1)\ \&\&\ (\neg b\ \rightarrow\ c_2),\ if\ b\ h_1\ h_2\ c',\ cs_1\ +\!+\ cs_2) \end{gathered}$$

En este caso, aplicamos VCG' recursivamente a p_1 y a p_2 con la postcondición c' para obtener c_1, h_1 y cs_1 obtenido de la primera recursión (p_1) y c_2, h_2 y cs_2 de la segunda recursión (p_2). Luego, construimos una condición compuesta que especifica que si b es verdadero, la postcondición será c_1 y si no, será c_2. Finalmente, creamos el árbol de Hoare correspondiente a la selección condicional y combinamos las listas de condiciones de verificación.

4. **Iteración (while)**:

$$VCG' : (while\ c\ b\ p,\ c')\ =\ let\ (c_1,\ h,\ cs)\ =\ VCG'\ (p,\ c)$$

$$in(c\ while\ (c,\ b,\ h),\ cs\ +\!+\ [c\ \wedge\ b\ \rightarrow\ c_1,\ c\ \wedge\ \neg b\ \rightarrow\ c'])$$

Como realizamos en los casos anteriores, aplicamos la función VCG' al subárbol h dentro del *while* para obtener las condiciones de verificación c_1. Recordemos que c es la invariante del bucle *while*. Luego, combinamos estas condiciones c_1 con otras condiciones que garantizan dos aspectos fundamentales:

a) Inicio del *while*: garantizamos que la invariante c se cumple al inicio del bucle, lo cual se representa como c.

b) Término del *while*: aseguramos que la postcondición del bucle (c') implica la invariante al finalizar el bucle, lo cual se expresa como c' $\rightarrow$ c.

Estas condiciones de verificación aseguran la corrección del bucle *while*, garantizando que la invariante se mantiene en cada iteración y que el bucle termina eventualmente dado a que contiene la condición de la guarda negada.

Una vez que hemos finalizado la construcción de la función VCG' el siguiente y último paso en nuestro proceso de verificación es establecer un lema que nos permita verificar la corrección de nuestro trabajo hasta el momento. Este lema establece una relación entre la función VCG' y la función *correct*, lo que nos permite verificar que las condiciones de verificación generadas por VCG' sean consistentes con las reglas del árbol de Hoare.

$$(c,\ h,\ cs)\ =\ VCG'\ (p,\ c')$$
$$correct\ h\ =\ (\{c\}\ p\ \{c'\},\ cs)$$

La igualdad (c, h, cs) = VCG'(p, c') indica que al pasar un programa p y una postcondición c' a la función VCG', obtenemos como resultado una precondición c, un árbol de Hoare h y una lista de condiciones cs. Por otro lado, la función *correct* aplicada al árbol de Hoare h nos proporciona las condiciones de verificación necesarias para demostrar la validez del programa p, es decir, c p c'.

Por lo tanto, si la igualdad *correct* h = (c p c') se cumple, significa que las condiciones generadas por la función *correct* coinciden con las condiciones de verificación generadas por VCG' para el programa dado. Esto nos da confianza en la validez de nuestro enfoque.

3.2. ¿Por qué Dafny?

Hasta el momento ya tenemos definido el bosquejo del algoritmo VCG, pero ¿por qué elegimos realizar la conversión de este bosquejo empleado en Haskell a Dafny?

Al momento de elegir el lenguaje de programación más adecuado para desarrollar nuestro código, nuestro tutor nos brindó información sobre dos opciones previamente investigadas por otros estudiantes de la universidad: KeY y Dafny. KeY, basado en Java, y Dafny, que emplea su propio lenguaje, fueron objeto de análisis en la tesis "Metodologīa de Programación con Dafny y KeY" llevada a cabo en 2022 por Gianfranco Drago y Matías Hernández. En su estudio comparativo, llegaron a la conclusión de que, a pesar de la familiaridad con Java que ofrece KeY y su sólido ecosistema, Dafny sobresale por la abundancia de recursos disponibles, que incluyen una amplia colección de ejemplos y una comunidad más activa y diversa.

Una de las principales ventajas de Dafny, según lo identificado en nuestra investigación, se establece en su capacidad para extender el verificador de manera intuitiva y sencilla mediante el uso de lemas y predicados. Esta característica facilita la formalización y verificación de propiedades específicas del código, contribuyendo así a mejorar la calidad y confiabilidad del software desarrollado. Además, en general, los investigadores reportan una experiencia más positiva en términos de soporte técnico y colaboración por parte del equipo detrás de Dafny.

En resumen, aunque KeY ofrece una sólida base en Java y puede ser una opción válida para ciertos contextos, la mayor disponibilidad de recursos y la flexibilidad proporcionada por Dafny lo convierten en la preferencia para aquellos que buscan una herramienta de verificación de programas potente y fácil de utilizar. Dado que nuestro equipo carece de experiencia previa con este tipo de herramientas, optamos por utilizar Dafny.

3.3. Boogie

Al escribir un programa en Dafny, el código se compila a un formato intermedio que puede ser interpretado por Boogie, actuando como una representación intermedia para simplificar el proceso de verificación formal. En este proceso, Boogie realiza un análisis exhaustivo y verifica formalmente que se cumplan las especificaciones establecidas en el código, tales como precondiciones, postcondiciones e invariantes.

Para verificar estas especificaciones, Boogie utiliza uno de sus solucionadores de Satisfacción de Restricciones (SMT) para determinar si se satisfacen. Comúnmente, Dafny utiliza Z3 como su solucionador SMT, el cual trabaja en segundo plano resolviendo las restricciones y determinando si el programa cumple o no con las especificaciones establecidas.

Además, si surgen problemas durante la verificación, Boogie genera un informe de

errores que indica la ubicación del problema y el tipo de error encontrado, esto es de suma importancia a la hora de trabajar con Dafny y poder encontrar donde se cometió la falla a la hora de codificar la solución.

Es importante destacar que Z3 es el solucionador SMT más utilizado por Dafny. Desarrollado por Microsoft Research, Z3 es un solucionador SMT potente y ampliamente reconocido en la comunidad de verificación formal.

En cuanto al concepto de *Satisfiability Modulo Theories* (SMT), se refiere a programas diseñados específicamente para resolver problemas de satisfacibilidad, es decir, determinar si una fórmula lógica dada puede ser satisfecha por una asignación de valores a sus variables. Esta capacidad de resolver problemas de satisfacibilidad es fundamental en el contexto de la verificación formal de programas.

Una posible área de investigación futura podría ser la implementación del mismo algoritmo desarrollado en Dafny, pero esta vez en Boogie. Esto implicaría un nivel adicional de comprensión y análisis, ya que Boogie opera en un nivel más bajo que Dafny y requiere un enfoque más detallado en la representación intermedia y la verificación formal.

3.4. Traducción de Haskell a Dafny

En el mundo de la programación, especialmente en el ámbito de la verificación formal de software, es fundamental comprender y dominar los conceptos clave que pertenecen a las herramientas y lenguajes utilizados. En este contexto, surge la necesidad de disponer de un recurso que sirva como guía rápida y concisa para familiarizarse con los términos más relevantes y utilizados en una herramienta específica como Dafny.

Los atributos de Dafny, como garantizar la corrección del código, reducir los errores y documentar implícitamente el código, se derivan del hecho de que el desarrollador agrega implícitamente el comportamiento esperado mediante precondiciones, postcondiciones e invariantes. Dafny luego ejecuta pruebas para asegurar que el código produzca los resultados esperados.

A continuación, presentamos una selección de conceptos claves para familiarizarse al utilizar Dafny:

- Invariantes (*invariant*): expresiones que deben ser verdaderas en ciertos puntos del programa para garantizar su corrección.
- Precondiciones (*require*) y postcondiciones (*ensure*): condiciones que deben cumplirse antes y después de la ejecución de una función o método, respectivamente.

- Verificación de programas: proceso de utilización de herramientas automáticas para comprobar si un programa cumple con sus especificaciones.
- Expresiones y sentencias: construcciones sintácticas que representan operaciones y acciones en el código, como asignaciones, bucles y condicionales.
- Funciones ghost: funciones que no se ejecutan en tiempo de ejecución y se utilizan para especificar propiedades del programa.
- *Decrease*: una función o expresión que garantiza la disminución de un valor en cada iteración de un bucle, esencial para demostrar la terminación de bucles en la verificación de programas.
- *Multiset*: estructura de datos que permite tener varios elementos repetidos, utilizada en Dafny para modelar conjuntos de elementos con repeticiones.
- *Predicate*: una expresión lógica que se evalúa como verdadera o falsa, fundamental para expresar condiciones en el código de Dafny.
- *Proof obligations*: condiciones que deben ser demostradas para verificar la corrección del programa, surgen de las especificaciones del programa y son verificadas automáticamente por el sistema de Dafny.

Para alcanzar los objetivos establecidos en la tesis, fue imprescindible desarrollar un programa mínimamente anotado. Este proceso implicó la creación de tipos de datos personalizados para modelarlo de manera precisa. Estos tipos de datos se organizaron en diferentes datatypes, cada uno diseñado para representar aspectos específicos del programa anotado.

Comenzando con el datatype *Expression*, este se encarga de representar las expresiones del programa, incluyendo valores numéricos, variables y operaciones aritméticas como la suma, resta y multiplicación.

```
datatype Expression =
            | L(number: int)
            | Var(varName: string)
            | sum(n1: Expression, n2: Expression)
            | substract(n1: Expression, n2: Expression)
            | mul(n1: Expression, n2: Expression)
```

Luego, tenemos el datatype *Condition*, diseñado para representar las condiciones lógicas utilizadas en el programa. Esto incluye condiciones simples como valores booleanos, así como operaciones más complejas como la negación, la implicación y la conjunción. Además, se proporcionan operadores de comparación como *Less, Greater y Equals* para comparar expresiones

```
datatype Condition =
            | K(boolean:bool)
            | Not(condition: Condition)
            | Imply(ConditonA: Condition, ConditonB: Condition)
```

```
    | And(ConditonA: Condition, ConditonB: Condition)
    | Substitution(substitution: map<string, Expression>,
       condition: Condition)
    | Less(e1:Expression, e2: Expression)
    | Greater(e1:Expression, e2: Expression)
    | Equals(e1:Expression, e2: Expression)
```

El datatype Program se utiliza para modelar la estructura y la secuencia de las instrucciones del programa. Incluye constructores para asignaciones, secuencias de instrucciones, instrucciones condicionales y bucles.

```
datatype Program =
          | Assign(assignments: map<string, Expression>)
          | Secuence(p1: Program, p2: Program)
          | If(condition: Condition, pThen: Program,
            pElse: Program)
          | While(pInvariant: Condition,
            condition: Condition, body: Program)
```

Finalmente, el datatype Specification encapsula una única instrucción del programa junto con sus condiciones de precondición y postcondición, proporcionando una representación compacta y completa de una parte del programa anotado. Por ejemplo:

```
datatype Specification =
                | Instruction(precondition: Condition,
                    program: Program, postcondition: Condition)
```

Posteriormente, consideramos importante destacar que existen dos categorías principales de tipos de datos en Dafny: básicos y avanzados.

Los tipos de datos básicos o comúnmente conocidos como tipos primitivos o fundamentales son aquellos que representan valores individuales y son las construcciones más simples en el sistema de tipos del lenguaje:

- *bool*
- *int*
- *string*

Por otro lado, existen los tipos de datos más avanzados o que resuelven otras complejidades y representan estructuras más elaboradas. Dentro de los mismos destacamos:

- *Map* <TipoA, TipoB>: Representa un diccionario donde TipoA es el identificador de la variable y TipoB es el valor asociado.
- *Set*: se utiliza para representar conjuntos de elementos únicos, es decir, no permite la repetición de elementos dentro del conjunto.

- *Multiset* permite la repetición de elementos y pueden estar desordenados. Esto significa que un multiconjunto puede contener múltiples instancias del mismo elemento.

Para finalizar con esta traducción, destaquemos algunos aspectos a considerar para aplicar en Dafny.

La firma de las funciones en nuestro código de Dafny se representa como:

```
VCG(specification: Specification) : (multiset<Condition>)
```

Donde el parámetro recibido corresponde a la especificación como un ejemplo, y el tipo de retorno corresponde a *multiset*, también como ejemplo.

Luego, para representar la siguiente línea en haskell:

```
let ({c} p1 {c1}, cs1) = correct \h1
```

Utilizamos la estructura:

```
match h1
    case x
    case y
```

Donde match h1 correspondería a realizar el *pattern matching* de *correct* h1 en el ejemplo mencionado, y el caso sería del tipo ({c} p1 {c1} , cs1).

3.5. Implementación del algoritmo VCG en Dafny

En este punto, contamos con toda la información necesaria para implementar ambas soluciones en Dafny. El objetivo principal de la creación del VCG en ambos casos es generar un árbol de Hoare a partir de un programa mínimamente anotado. Se siguieron las reglas definidas por la lógica de Hoare como guía para esta implementación.

Para mantener la claridad en el documento y proporcionar un acceso fácil al código de ambas soluciones, hemos creado un repositorio en GitHub. En este repositorio, los lectores encontrarán las implementaciones detalladas del *Verification Condition Generator* (VCG) utilizando tanto la solución superficial como la profunda. Este enfoque nos permite ofrecer una visión completa de cómo se abordan los desafíos de verificación de programas en Dafny.

Animamos a los interesados a visitar el siguiente enlace para explorar el código y comprender mejor las dos metodologías implementadas: algoritmos VCG.

A continuación, destacaremos ciertos puntos críticos que deben considerarse en nuestra implementación, específicamente los definidos para cada solución, ya sea superficial o profunda.

Solución Profunda:

En esta solución, es necesario definir el tipo del Árbol de Hoare. El mismo queda determinado por el siguiente código:

```
datatype THoare =
                Assign(assignments:map<string,Expression>,
                    condition: Condition)
                | Sequence(tree1: THoare, tree2: THoare)
                | If(condition: Condition, tree1: THoare,
                    tree2: THoare,cp:Condition)
                | While(pInvariant: Condition,
                    condition: Condition, tree: THoare)
```

Solución Superficial:

Para este caso es crucial crear el predicado Hoare. La implementación que generamos teniendo en cuenta a lo que queremos llegar es la siguiente:

```
ghost predicate Hoare(prog: Specification, vcs: multiset<Condition>)
decreases var Instruction(precondition, program, postcondition) :=
prog ; program, |vcs|

{
  (match prog
  case Instruction(precondition, program, postcondition) => (
    match program
    case Assign(assignments) => (
      precondition == Condition.Substitution(
      assignments,postcondition
      )
      && vcs == multiset{}
    )
    case Sequence(p1, p2) => (
      exists s,vcs1,vcs2 :: Hoare(Instruction(
      precondition, p1, s
      ),
      vcs1) && Hoare(Instruction(s, p2, postcondition), vcs2)
      && vcs == vcs1 + vcs2
    )
    case If(condition, p1, p2) => (
      exists vcs1,vcs2 :: Hoare(Instruction(
      And(precondition,condition), p1, postcondition), vcs1) &&
      Hoare(Instruction(And(precondition,Not(condition)),
      p2, postcondition), vcs2) && vcs == vcs1 + vcs2
    )
    case While(inv, b, p) => (
```

```
      Hoare(Instruction(And(inv,b),p,inv),vcs) &&
      (precondition == inv) && (postcondition == And(inv,Not(b)))
    )
  )
  || (exists qP,vcs1 :: vcs == vcs1 +
  multiset{Imply(precondition,qP)} && assert |vcs1| < |vcs|;
  Hoare(Instruction(qP,program,postcondition),vcs1))
  || (exists rP,vcs1 :: vcs == vcs1 +
  multiset{Imply(rP,postcondition)} && assert |vcs1| < |vcs|;
  Hoare(Instruction(precondition,program,rP),vcs1))
  )

}
```

Ambas soluciones, tanto la profunda como la superficial, ofrecen enfoques válidos y efectivos para la implementación del VCG en Dafny. Sin embargo, cada enfoque presenta aspectos importantes a considerar:

Solución Profunda: proporciona una representación explícita del árbol de Hoare, lo que puede ser beneficioso para comprender visualmente la estructura del programa y su verificación. Requiere definir un tipo de dato específico para el árbol de Hoare, lo que puede ser más detallado pero también más complejo.

Solución Superficial: utiliza un enfoque más directo mediante la definición de un predicado Hoare, lo que simplifica la implementación y reduce la complejidad del código. No requiere definir un tipo de dato adicional, lo que la hace más accesible y fácil de comprender para aquellos que no están familiarizados con los detalles internos del árbol de Hoare.

3.6. Conclusiones

Como objetivo principal de este proyecto, nos propusimos dar el primer paso hacia la construcción de un algoritmo que emplee internamente Dafny para verificar la corrección de un programa. Limitamos el alcance del lenguaje a un programa mínimamente anotado y asumimos la terminación del mismo, con el propósito de probar su corrección parcial. Si bien reconocemos que el lenguaje podría ampliarse, consideramos que este paso inicial sienta las bases para que otros proyectos continúen en la misma dirección. Alcanzar la completa demostración de la corrección sería un objetivo muy ambicioso, dado el amplio alcance del lenguaje de Dafny.

Durante el desarrollo del proyecto, nos encontramos con desafíos relacionadas con una de las dos aproximaciones consideradas para probar la corrección del programa. Inicialmente, nos enfocamos en una solución que implicaba la construcción de un árbol de Hoare como datatype (solución profunda). Nos vimos obstaculizados por la primera solución, ya que tuvimos dificultades para devolver un árbol de Hoare que se ajustara al VCG. Esto nos llevó a buscar una alternativa, lo que resultó en la solución de utilizar Hoare como predicado en el código final. Esta revisión de enfoque fue el resultado de un proceso de reflexión y análisis, que nos llevó a una solución más efectiva y viable.

En relación a la herramienta utilizada, Dafny, no experimentamos dificultades significativas durante el proceso. Aunque existe una amplia disponibilidad de información en línea, aprender a utilizarla desde cero y adaptarnos a su estilo de programación particular presentó un desafío considerable. Esta adaptación difirió de nuestra experiencia habitual y requirió tiempo para familiarizarnos y practicar con el lenguaje, lo que implicó una curva de aprendizaje más pronunciada.

Como desafío futuro hacia la defensa, nos planteamos en avanzar sobre la implementación de la primera solución y lograr que la misma quede completamente funcional, así como desarrollar un ejemplo práctico para respaldar nuestro trabajo como cierre en defensa del proyecto.

3.7. Trabajo futuro

- Como se mencionó previamente, en este experimento se asumió la terminación del programa, lo que implica que se probó la corrección parcial del mismo, dejando fuera del alcance la corrección total.
- Inicialmente, intentamos comparar el resultado del algoritmo VCG con la función correct. Sin embargo, al enfrentar problemas con la formación del árbol de Hoare y haber tenido que recurrir a la lógica de Hoare como un predicado, la solución más específica quedó sin probar. Esto se debió a que encontramos una alternativa para abordar el problema de manera diferente. Nos proponemos realizar pruebas adicionales de esta solución para la defensa.
- Para ampliar el lenguaje, se contempla la posibilidad de agregar funciones ejecutables y la capacidad de inspeccionar el contenido de cada línea de código. Esto implica agregar asserts a la representación del lenguaje, lo que permitirá una verificación más exhaustiva y detallada de la corrección del programa.
- Una perspectiva de investigación futura podría ser la adaptación del algoritmo desarrollado en Dafny para su implementación en Boogie. Esta tarea requeriría un nivel adicional de comprensión y análisis, dado que Boogie opera en un nivel más bajo que Dafny. Implicaría un enfoque más detallado en la representación intermedia y en la verificación formal del código, lo que podría abrir nuevas oportunidades para la mejora y optimización del proceso de verificación de programas.

Bibliografía

[1] K. R. M. Leino, *Program Proofs*. The MIT Press (March 7, 2023), 2023.

[2] K. Rustan M. Leino. Dafny Reference manual. Accedido el 20/03/2024. [Online]. Available: https://dafny.org/dafny/DafnyRef/DafnyRef

[3] Leonardo de Moura and Nikolaj Bjørner. Z3, An efficient SMT solver. Accedido el 20/03/2024. [Online]. Available: https://www.microsoft.com/en-us/research/project/z3-3/

[4] P. M. Gustav Parthasarathy and A. J. Summers, *Formally Validating a Practical Verification Condition Generator (extended version)*, 2021.

[5] P. V. Homeier and D. F. Martin, *A Mechanically Verified Verification Condition Generator*, 1995.

[6] M. Hernández and G. Drago, *Metodología de Programación con Dafny y KeY*, 2022.