

Universidad ORT Uruguay

Facultad de Ingeniería

Verificación Formal de Algoritmos Fundamentales para la Industria

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

Guillermo Colmenero – 162332

Tutor: Álvaro Tasistro

2016

Declaración de autoría

Yo, Guillermo Colmenero, declaro que el trabajo que se presenta en esa obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el proyecto final de carrera;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mí;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Guillermo Colmenero

Fecha: 23/08/2016

Abstract

En este trabajo se analiza el paradigma de desarrollo de software actual y se aborda la problemática que éste presenta a la hora de asegurar la corrección de software.

Se describe una posible solución mediante la utilización de métodos formales y se presenta un fundamento teórico compuesto por la lógica de Hoare y el método de la variante.

Una vez establecido el fundamento teórico, se hace hincapié en Dafny, una herramienta que utiliza la metodología vista y se desarrollan una serie de algoritmos fundamentales verificados totalmente.

Palabras Clave

Dafny, Variantes, Programación, Software, Invariantes, Verificación Formal, Lógica de Hoare, Paradigmas de Programación, Métodos Formales, Programación basada en Invariantes, Desarrollo de Software, Corrección de Algoritmos.

Índice

1. INTRODUCCIÓN.....	7
1.1. PREFACIO	7
1.2. PARADIGMA ACTUAL.....	8
1.2.1. Especificación.....	9
1.2.2. Diseño y codificación.....	9
1.2.3. Validación.....	9
1.3. CRÍTICAS AL PARADIGMA ACTUAL	10
1.4. EL DESARROLLO FORMAL COMO SOLUCIÓN.....	12
2. FUNDAMENTO TEÓRICO.....	14
2.1. PRECONDICIONES Y POSTCONDICIONES.....	14
2.2. CORRECCIÓN CONTRA PROPÓSITO	15
2.3. LÓGICA DE HOARE	16
2.3.1. Terna de Hoare.....	16
2.3.2. Reglas	17
2.4. INVARIANTES	19
2.5. EJEMPLO DE VERIFICACIÓN DE UN PROGRAMA ANOTADO.....	21
2.6. VERIFICACIÓN CONTRA DERIVACIÓN.....	23
2.7. VARIANTE	24
2.8. CORRECCIÓN TOTAL.....	24
3. HERRAMIENTAS PARA VERIFICAR CORRECCIÓN.....	25
3.1. FRAMA-C	25
3.2. DAFNY	26
3.2.1. Breve descripción de la herramienta	26
3.2.2. Funcionamiento	28
3.2.3. Relación con el fundamento teórico	29
4. EJEMPLOS DE DERIVACIÓN EN DAFNY	31
4.1. BÚSQUEDA SECUENCIAL.....	31
4.1.1. Descripción.....	31
4.1.2. Especificación.....	31
4.1.3. Diseño de la invariante.....	32
4.1.4. Derivación	32
4.1.5. Implementación.....	33
4.2. SELECTION SORT.....	33

4.2.1.	<i>Descripción</i>	33
4.2.2.	<i>Especificación</i>	34
4.2.3.	<i>Diseño de la invariante exterior</i>	35
4.2.4.	<i>Derivación del método exterior</i>	36
4.2.5.	<i>Implementación</i>	37
4.2.6.	<i>Especificación del método select</i>	37
4.2.7.	<i>Diseño de la invariante</i>	38
4.2.8.	<i>Derivación del método select</i>	38
4.2.9.	<i>Implementación del método select</i>	39
5.	ALGORITMOS IMPLEMENTADOS EN DAFNY	40
5.1.1.	<i>Búsqueda de máximo en array</i>	40
5.1.2.	<i>Búsqueda en array</i>	42
5.1.3.	<i>Radicación discreta</i>	45
5.1.4.	<i>División natural</i>	46
5.1.5.	<i>Potenciación</i>	48
5.1.6.	<i>Máximo común divisor</i>	49
5.1.7.	<i>Logaritmo discreto</i>	50
5.1.8.	<i>Multiplicación utilizando sumas</i>	51
5.1.9.	<i>Operaciones a nivel de bits</i>	52
5.1.10.	<i>Búsqueda ceros</i>	52
5.1.11.	<i>Máximo provecho en bolsa</i>	53
5.1.12.	<i>Ordenamiento</i>	54
6.	LECCIONES APRENDIDAS Y CONCLUSIONES	59
6.1.	EXPERIENCIAS AL DESARROLLAR ESTA TESIS	59
6.2.	CONCLUSIONES	60
7.	REFERENCIAS BIBLIOGRÁFICAS	61

1. Introducción

1.1. Prefacio

En el último medio siglo, el papel del software ha experimentado un cambio sustancial. Avances de hardware, como por ejemplo el abaratamiento de la capacidad de almacenamiento y el desarrollo de nuevas interfaces de entradas y salidas, han hecho posible la existencia de sistemas informáticos complejos. Por cada década que pasa, los sistemas informáticos se tornan más potentes y, como resultado, han ido adquiriendo roles más significativos en nuestra vida cotidiana.

En la actualidad, la industria del software se ha convertido en un factor dominante en las economías del mundo industrializado. La misma se ve cada vez más involucrada en operaciones que son responsables del bienestar de seres humanos [1].

Existen industrias donde la corrección del software es crucial, ya que hasta el más pequeño error puede ser el causante de una gran catástrofe.

Desafortunadamente, el 4 de Junio de 1996, luego de haber transcurrido 39 segundos desde su lanzamiento, el software de navegación de la nave ordenó un giro brusco de 90 grados haciendo que ella se saliera de su trayectoria. Dicha orden activó un mecanismo de autodestrucción que esparció sus escombros prendidos fuego por los manglares de la Guayana francesa. La investigación concluyó que se trató de un error de software que intentó guardar un número de 64-bit en un registro de 16-bit [2].

En 2015 el fabricante de autos Nissan tuvo que retirar unos 4 millones de automóviles del mercado debido a problemas de software en el sistema de bolsas de aire (*airbags*) que permitía que sus pasajeros pudieran experimentar graves lesiones [3].

Estos incidentes son solo dos elementos de una larga lista de accidentes que han costado, en los peores casos, vidas de personas y son una clara prueba de la importancia de escribir software correcto. Los ingenieros de software, al igual que los ingenieros civiles, deberían poder demostrar metódicamente que su trabajo realmente cumple con el propósito para el cual fue concebido y que es, por consiguiente, confiable.

Un extremo de exigencia en este sentido es plantearse la demostración formal del software, es decir, una demostración de su corrección que pueda ser verificada automáticamente. Los artefactos de software capaces de realizar tal *chequeo* son llamados, apropiadamente, compiladores verificadores.

En efecto, de un programa que ha sido chequeado por un compilador verificador puede aseverarse que satisface su especificación, a menos que existan fallas en la implementación del verificador en cuestión [4].

En otras palabras, el diseño y correcta implementación de un compilador verificador equivale a proporcionar un lenguaje cuyos programas cumplen una propiedad por cierto deseable: si compilan, funcionan. El tema de la presente tesis es una experiencia en la utilización de una de estas herramientas, a saber, Dafny.

1.2.Paradigma actual

A pesar de que existen distintos **modelos** / **paradigmas** / **frameworks** de desarrollo de software, al estudiarlos en profundidad podemos encontrar ciertas fases y prácticas que se repiten una y otra vez, independientemente del procedimiento de desarrollo, la aplicación del software y el tamaño del proyecto.

Las fases comunes que se pueden identificar son: **Especificación**, **Diseño y codificación** y **Validación**.



Imagen 1 – 1 Ciclo de vida del software.

1.2.1. Especificación

En esta etapa de la ingeniería del software, se recopila información sobre el problema a resolver. El objetivo principal es generar un documento objetivo (sin ambigüedades) que describe cómo se debe comportar el programa en función a sus entradas [5].

Este documento es crucial ya que, a partir de él, los diseñadores y/o programadores obtendrán una comprensión de las complejidades y sutilezas relacionadas con el problema a resolver.

1.2.2. Diseño y codificación

Luego que la documentación referente a la especificación alcanza un alto grado de madurez, los desarrolladores proceden con la fase de diseño. En esta etapa se consideran varios algoritmos para resolver el problema planteado de una manera eficiente.

Si el problema es conocido, se buscan algoritmos conocidos que lo resuelvan. De lo contrario, se diseña un nuevo algoritmo. Luego de que éstos quedan definidos, se procede a codificarlos en el lenguaje de programación elegido para el proyecto.

1.2.3. Validación

Durante la fase de validación, los desarrolladores se aseguran de que el programa cumpla con los objetivos establecidos en la fase de especificación. La práctica más común en la industria para lograr este objetivo es realizar *testing* [6].

El objetivo del *testing* es detectar fallas en el software para luego corregir defectos en el código. El alcance de esta práctica incluye la revisión del código fuente, así como su ejecución en distintos entornos y con distintas entradas. Cabe destacar que generalmente los entornos y las entradas consisten en valores esperados y casos borde.

Una vez que el software fue validado, el ciclo de desarrollo asume que el programa cumple con su especificación y el ciclo se repite para refinar el programa.

1.3. Críticas al paradigma actual

El principal problema del paradigma de desarrollo descrito en la sección anterior radica en que no puede asegurarse que el software cumple con las especificaciones. En primer lugar, la especificación se tiende a hacer en lenguaje natural usando términos sólo vagamente definidos y por ende podría contener ambigüedades.

Luego, debido a la gran capacidad de las computadoras actuales, el tiempo que transcurre entre escribir un programa y la posibilidad de ejecutarlo es mínimo. Es por eso que algunos programadores poco experimentados diseñan sus algoritmos a base de prueba y error. Esto conlleva a que tengan un vago entendimiento de su funcionamiento y no puedan predecir su comportamiento.

Por último, a la hora de escribir pruebas en la fase de *testing*, los desarrolladores utilizan lógica ad hoc para convencerse de que su código es correcto. Supongamos que queremos probar un programa que recibe un número real como entrada y le suma el número 42. Es imposible probar todos los casos posibles debido a que existen infinitos números reales. Es por eso que las pruebas de validación tienden a probar un número limitado de casos que se supone que son una muestra representativa de las posibles entradas. Esta práctica también podría estar limitada por restricciones de tiempo y dinero y a veces ni siquiera se alcanza a probar dicha muestra [6].

Por otra parte, en casos tales como sistemas distribuidos que presentan una variedad de condiciones de ejecución concurrente de diferentes componentes del sistema de complejidad arbitraria, es de hecho quimérico pretender reproducir por medio del testing siquiera aquellos casos límites característicos. Es, por el contrario, imprescindible razonar a priori sobre las propiedades del sistema para mantener el control sobre el comportamiento producido.

Por lo general, los desarrolladores se quedan satisfechos cuando sus pruebas se ejecutan correctamente y aseguran que el programa es correcto, a pesar de que no hay pruebas suficientes para hacer tal afirmación. En realidad esto es conocido como una falacia de llamada a la ignorancia.

“En lógica, un argumento ad ignorantiam, o argumentum ad ignorantiam, también conocido como llamada a la ignorancia, es una falacia que consiste en sostener la verdad (o falsedad)

de una proposición alegando que no existe prueba de lo contrario, o bien alegando la incapacidad o la negativa de un oponente a presentar pruebas convincentes de lo contrario.

Esta impaciencia con la ambigüedad suele criticarse con la frase: “**la ausencia de prueba no es prueba de ausencia**” es decir, se comete esta falacia cuando se infiere la verdad o falsedad de una proposición basándose en la ignorancia existente sobre ella.” [7].

Esto conlleva a que los programas puedan tener errores y tengan que ser sujetos al proceso de debugging o depuración. En este proceso, los desarrolladores se encargan de realizar cambios en el código para obtener el comportamiento que se deseaba en un principio. Se estima que utilizando el paradigma descrito en la sección anterior, los desarrolladores utilizan un 49.9% del tiempo del proyecto depurando sus propios errores e insumen unos 156 mil millones de dólares anualmente a nivel mundial [8].

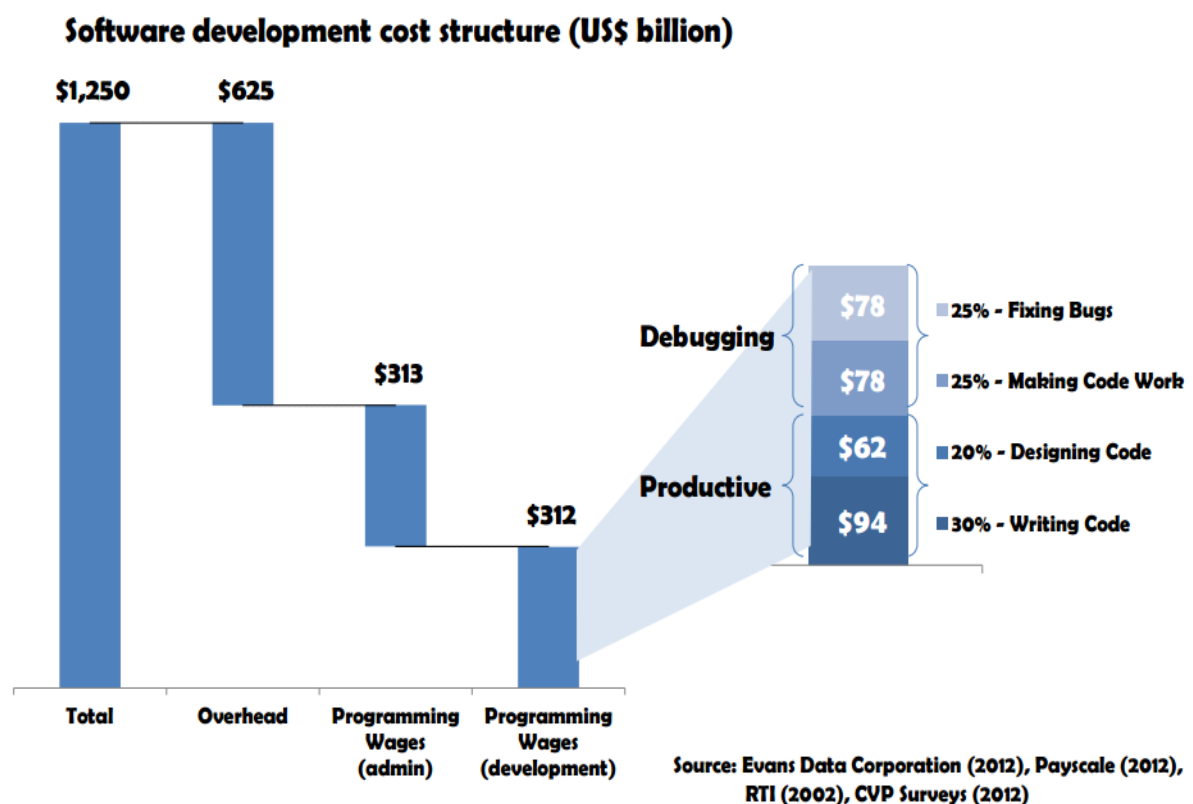


Imagen 1 – 2 Costos del desarrollo de software [8]

1.4.El desarrollo formal como solución

Pese a que se utilizan numerosas herramientas y se dedica un gran número de recursos para escribir software correcto, el paradigma actual carece de rigor lógico/matemático. Esto lo hace poco apto para aplicaciones en donde el software desempeña un rol crítico y/o el bienestar de seres humanos o bienes comerciales dependan de éste (por ejemplo: sistemas de tiempo real, fly by wire, aplicaciones médicas, etc.).

Una posible solución a esta problemática es utilizar un método formal de desarrollo. El modelo de métodos formales agrupa actividades que llevan a la especificación lógica/matemática formal del software y a demostrar su corrección con respecto a la misma [1].

El propósito de los métodos formales es proveer certeza de las propiedades (corrección) del código a través del razonamiento (deducción) es decir, por el método que se emplea en Matemática. En otras palabras, busca fundamentar el desarrollo de software en la Matemática. Como consecuencia, la investigación en métodos formales se orienta a desarrollar notaciones y técnicas de especificación de software y de razonamiento sobre el código ejecutable de forma de poder a priori establecer las propiedades relevantes.

En este sentido, podría aseverarse que se trata de una búsqueda de genuinas prácticas ingenieriles de desarrollo, en oposición a la, desafortunadamente, demasiado común, práctica de tratar al software exclusivamente como conjetural caja negra.

El precio a pagar es un costo más elevado de desarrollo en las etapas tempranas, puesto que la producción de especificaciones y de código requiere más precisión y más tiempo. Sin ir más lejos, uno debe en general producir no sólo código, sino también la demostración de que él es correcto respecto de una especificación que, a su vez, debe haber sido depurada de ambigüedades.

Tanto la especificación como la implementación y demostración de sistemas de software son tareas en general en extremo complejas. Esto afecta la factibilidad del enfoque de los métodos formales y, puede decirse en consecuencia, también el ideal de la confiabilidad del software. Cuando un sistema tiene éxito, la sofisticación y complejidad producen resultados

deslumbrantes, pero al mismo tiempo los sistemas complejos plantean problemas enormes para aquellos que deben construirlos.

Sin embargo, en muchos casos (llamados usualmente críticos) el empleo de métodos que provean certeza a priori es requerido, inclusive en estándares de comercialización, puesto que el costo de errores a posteriori es potencialmente infinito. Y, más allá de esos casos extremos, puede decirse que el empleo de prácticas orientadas a la precisión en la especificación y el razonamiento sobre el código deberían ser siempre bienvenidas, por proporcionar al autor un mayor control sobre su trabajo.

Por ello la investigación en métodos formales es relevante en todo el espectro que va desde los intereses puramente educacionales a los prácticos. En un extremo, los métodos formales se orientan al empleo de lenguaje lógico formal para la expresión de las especificaciones y de las demostraciones.

Este tipo de lenguajes permite el *chequeo automático* de que las demostraciones son correctas y posibilita el desarrollo de los compiladores verificadores ya explicados. También permiten que algunas de estas demostraciones puedan ser desarrolladas automáticamente, aliviándose así la carga del programador.

No obstante, no existe algoritmo capaz de demostrar o refutar cualquier enunciado matemático dado y por lo tanto, en general la demostración formal de los programas es responsabilidad del programador.

Sin embargo, los compiladores verificadores combinan en general la demostración automática y la verificación formal de demostraciones; últimamente el desarrollo de los demostradores automáticos ha recibido un impulso muy considerable, lo cual ha permitido a su vez facilitar notablemente su empleo.

En este trabajo vamos a explorar la derivación y verificación formal de algoritmos mediante herramientas automáticas, analizar sus posibles aplicaciones así como su grado de madurez para la industria actual y compartir nuestras experiencias con el lector.

2. Fundamento teórico

2.1.Precondiciones y Postcondiciones

Intuitivamente, cada vez que se nos plantea un problema, tendemos a pensar en el objetivo a cumplir y en los recursos disponibles hasta el momento (herramientas, reglas a cumplir, materiales, etc.). Análogamente, a la hora de diseñar una solución de software, los programadores determinan las precondiciones y postcondiciones.

Las precondiciones son supuestos que se toman como válidos a la hora de ejecutar un programa. Las postcondiciones son requerimientos que deben cumplir una vez finaliza su ejecución.

A los efectos de la verificación formal, es importante escribir las especificaciones significativas que caracterizan el comportamiento de un programa ya que éstas oficiarán de base para definir la corrección del mismo.

Las postcondiciones debe reflejar propiedades o relaciones de las entradas del programa luego de ser manipuladas en la ejecución, así como también las precondiciones deben establecer los requerimientos para que las postcondiciones puedan ser alcanzadas.

Ejemplo: Se quiere buscar el número más grande dentro de un *array* de enteros no vacío.

Precondición 1: $a \neq \text{null}$;

Esto significa que el *array* no puede ser nulo.

Precondición 2: $a.Length > 0$;

Esto significa que el *array* debe tener al menos un elemento.

Postcondición 1: $\text{forall } j : \text{int} :: j \geq 0 \ \&\& \ j < a.Length \implies mx \geq a[j]$;

Esto significa que el elemento que el programa devuelve (*mx*), tiene que ser mayor o igual a todos los demás elementos del *array*.

Postcondición 2: $\text{exists } j : \text{int} :: j \geq 0 \ \&\& \ j < a.Length \ \&\& \ mx == a[j]$;

2.2. Corrección contra propósito

Existe una gran diferencia entre la corrección de un programa y su propósito. Un programa puede ser correcto y al mismo tiempo no respetar el propósito que se desea que este cumpla. Si se eligen precondiciones y postcondiciones erróneas o insuficientes, podríamos demostrar la corrección de un comportamiento irrelevante o no deseado.

Esto puede ser un indicador de que el problema planteado no está siendo comprendido por el programador.

Contraejemplo 1:

Descripción del problema: Se quiere verificar un programa que suma dos enteros.

Precondición 1: $a, b: \text{int};$

Postcondición 1: $\text{resultado}: \text{int};$

En este caso, cualquier implementación que retorna un número entero será válida según estas precondiciones y postcondiciones. Un programa que retornase el número 42 sería correcto según la especificación pero no respetaría el propósito del enunciado del problema.

Si bien este ejemplo es trivial, existen casos complejos donde errores en la especificación son más difíciles de detectar y podrían determinar el fracaso de un proyecto.

Para evitar este problema, debemos buscar especificar precondiciones y postcondiciones que se desprendan directamente del propósito del programa y que además sean suficientemente abarcativas como para cubrir la totalidad de los requerimientos del problema a resolver.

Una manera de corregir el **contraejemplo 1** sería agregando otra postcondición a la especificación:

Precondición 1: $a, b: \text{int};$

Postcondición 1: $\text{resultado}: \text{int};$

Postcondición 2: $\text{resultado} == a + b;$

Dado la complejidad de esta tarea, existe una disciplina específica para esto llamada “Ingeniería de Requerimientos”.

2.3.Lógica de Hoare

En la década de los 60' la programación era muy distinta a lo que conocemos hoy en día. Si un programador tenía suerte, podía acceder a una computadora una sola vez al día. La compilación podía llevar horas y depurar un programa era una tarea muy tediosa [9].

Los lenguajes de programación no eran tan avanzados como los de hoy en día. Por ejemplo: los programadores eran los encargados de codificar las rutinas que controlaban que se estuviera accediendo a una dirección de memoria válida. Muchas veces, funciones de este tipo se omitían para optimizar la utilización de recursos.

Es por esto, que los programadores de dicha época, tenían que ser muy cautelosos con el diseño de sus algoritmos. Se hacía mucho énfasis en comprender el funcionamiento de los programas y se empleaban más recursos para programar correctamente. Lamentablemente, seguía habiendo casos donde la cautela y la dedicación no eran suficientes para resolver este problema.

Por ese motivo, Sir Charles Antony Richard Hoare desarrolló un modelo teórico para ayudar a razonar la corrección de programas, conocido actualmente como lógica de Hoare. Este modelo describe una serie de pasos para probar que una implementación de un programa es acorde a su especificación y eliminar la necesidad de probar el código, logrando así ahorrar recursos.

2.3.1. Terna de Hoare

La principal característica de la lógica de Hoare es la terna de Hoare, que describe como la ejecución de determinado código, cambia el estado de un programa. Esta terna se suele denotar de la forma $\{P\} S \{Q\}$, donde **P** y **Q**, llamadas precondiciones y postcondiciones respectivamente, son aserciones (expresadas en lógica de predicados) que deben cumplirse para que el programa **S** funcione correctamente [10].

Es decir, que si el programa **S** comienza en un estado válido en **P** y, siempre que finalice su ejecución lo hace en un estado válido para **Q**, entonces es correcto con respecto a su especificación $\{P, Q\}$. Notar que la validez de una terna de Hoare no asegura la terminación

del programa en cuestión. La terna de Hoare expresa, por lo tanto, lo que se llama la corrección parcial del programa, en oposición a su corrección total, que garantiza también la terminación en todos los casos en que la ejecución se inicia en un estado que satisfaga la precondition.

2.3.2. Reglas

La lógica de Hoare provee axiomas y reglas de inferencia para todas las estructuras de control básicas encontradas en lenguajes de programación imperativos.

Axioma para la asignación

$$\frac{}{\{p[x/e]\} \ x = e \ \{p\}}$$

Imagen 2 – 1 Regla asignación [11].

Este axioma refleja el hecho de que después de ejecutar un comando $x := e$, el valor de la variable x se iguala al valor de e , y si se quiere que la postcondición p sea válida después de la asignación, antes debe ser válida para e en lugar de x [10].

Regla para la selección

$$\frac{\begin{array}{c} \{p \wedge e_1\} \ s_1 \ \{q\} \\ \{p \wedge \neg e_1\} \ s_2 \ \{q\} \end{array}}{\{p\} \ \text{if } e_1: \ s_1 \ \text{else: } s_2 \ \{q\}}$$

Imagen 2 – 2 Regla selección [11].

Esta regla refleja la semántica del comando condicional **if – else** e indica que para que una construcción con este comando sea correcta respecto a una especificación $\{p, q\}$, lo debe ser cada uno de los programas alternativos s_1 y s_2 respecto a las especificaciones obtenidas reforzando la precondition con las condiciones necesarias para que se ejecuten dichos programas.

Regla para la secuencia

$$\frac{\{p\} \text{ } s_1 \{r\} \{r\} \text{ } s_2 \{q\}}{\{p\} \text{ } s_1; s_2 \{q\}}$$

Imagen 2 – 3 Regla secuencia [11].

Esta regla refleja semántica del secuenciamiento de instrucciones. Primero se ejecuta s_1 y luego s_2 . Su aplicación para deducir una corrección exige que se determine un predicado intermedio r para la que se cumplan las premisas (llamado generalmente condición media).

Regla para la consecuencia

$$\frac{\{p_1\} \text{ } s \{q_1\} \quad p_2 \implies p_1 \quad q_1 \implies q_2}{\{p_2\} \text{ } s \{q_2\}}$$

Imagen 2 – 1 Regla consecuencia [11].

Esta regla refleja la semántica de las fórmulas de corrección parcial e indica que siempre se puede reforzar la precondition y/o debilitar la postcondición de una fórmula de corrección válida.

Regla para la iteración

$$\frac{\{e_1 \wedge e_2\} \text{ } s \{e_2\}}{\{e_2\} \text{ while } e_1 \text{ where } e_2: s \{ \neg e_1 \wedge e_2 \}}$$

Imagen 2 – 1 Regla iteración [11].

Esta regla nos dice que si e_2 se mantiene cada vez que se ejecuta el cuerpo del ciclo con las condiciones adicionales dadas por e_1 , entonces e_2 se mantendrá invariante para cualquier número de repeticiones del ciclo y se cumplirá al final junto con la condición de terminación del ciclo (negación de e_1) [12].

A estos predicados e_2 se les llama invariantes del ciclo. La aplicación de la regla exige que se determine (derive) una invariante conveniente para la corrección que se quiera deducir.

2.4. Invariantes

En el capítulo anterior, definimos al invariante como una condición que no cambia para cualquier número de repeticiones de un ciclo y que además se sigue cumpliendo junto con la condición de terminación del mismo.

Supongamos que tenemos un *array* **a** de números enteros no vacío de largo **n** y queremos buscar el máximo entero almacenado dentro del mismo. Suponemos que los índices válidos del *array* son aquellos en el intervalo $[0, n)$.

Formalizando el enunciado del problema obtenemos las siguientes precondiciones y postcondiciones:

- **Precondición:** $n > 0$;
- **Postcondición:** $\forall i: i \in \mathbb{Z} : 0 \leq i < n \Rightarrow max \geq a[i]$
- **Postcondición:** $\exists j: j \in \mathbb{Z} : 0 \leq j < n \wedge max == a[j]$

Al resolver este problema utilizando programación imperativa, tendemos a pensar en una estructura de ciclo (*loop*). Dentro del ciclo manejamos 2 variables: una para indicar la posición del *array* a visitar la cual llamaremos **index** y otra para guardar el número más grande encontrado hasta el momento la cual llamaremos **max**.

La programación basada en invariantes se enfoca en hacer explícito y preciso cuáles son las variables a utilizar en cada estructura de ciclo y cuáles son las propiedades relevantes que estas variables cumplen.

En este ejemplo, deseamos que la variable de posición **index** indique posiciones válidas dentro del *array* (excepto al final, cuando el ciclo termina) y que la variable **max**, indique el máximo número encontrado hasta el momento, es decir:

- $0 \leq index \leq n$
- $\forall j: j \in Z :: 0 \leq k < index \Rightarrow max \geq a[k]$

Además, a medida que vamos recorriendo el *array*, efectivamente tiene que existir un máximo en el intervalo recorrido:

- $\exists n: n \in Z :: 0 \leq n < index \Rightarrow max == a[n]$

Estas condiciones son las invariantes del *loop*. Para que una invariante se verifique debemos asegurarnos que se cumplan las siguientes condiciones:

1. La invariante se cumple antes de entrar al *loop*.
2. Cualquier ejecución de las instrucciones del *loop* preserva la invariante.

De estas condiciones se desprende que la invariante tiene que cumplirse una vez finalizado el *loop* por simple inducción matemática. También es importante asegurarse de que los invariantes planteados sean efectivamente útiles para resolver el problema.

Para ello es suficiente verificar tres condiciones:

- a) **Inicialización:** debe poderse (en general por medio de instrucciones simples) garantizar el cumplimiento de la invariante al comienzo del ciclo.
- b) **Terminación:** Se debe diseñar una condición de terminación que, en conjunto con la invariante, implique la postcondición deseada.
- c) **Paso del ciclo:** debe diseñarse la acción a efectuar en cada paso del ciclo de modo de que:
 - i) se preserve la validez de la invariante y
 - ii) se progrese efectivamente hacia la condición de terminación [13].

2.5. Ejemplo de verificación de un programa anotado

El uso práctico de la lógica de Hoare termina siendo la reducción del problema de corrección del programa, a un conjunto de teoremas matemáticos que se llamaremos condiciones de verificación. Ilustraremos esto por medio de un ejemplo, en el cual se tiene un programa que realiza la división entera de m entre n por restas sucesivas. El programa terminará cuando r (resto) sea menor a n . Observemos que n debe de ser mayor a cero para asegurarnos que el programa termine.

Primero comenzamos por la especificación anotando las precondiciones, postcondiciones e invariante de cada *loop*.

```
{m >= 0 && n > 0} // Precondición
q,r := 0,m;
while (r >= n)
  inv {m == qn+r && 0 <= r} // Invariante
  {
    q,r := q + 1; r - n;
  }
{m == qn+r && 0 <= r && r < n} // Postcondición
```

Luego aplicamos la regla para la iteración. Para esto, primeramente se rodea el *loop* con la invariante como precondición y como postcondición agregando la condición de terminación.

```
{m >= 0 && n > 0}
q,r := 0,m;
{m == qn+r && 0 <= r}
while (r >= n)
{
  q,r := q + 1; r - n;
}
{m == qn+r && 0 <= r && !(r >= n)}
{m == qn+r && 0 <= r && r < n}
```

Para terminar de aplicar la regla de la iteración, hace falta mostrar que el paso del *loop* preserva la invariante.

```

{m >= 0 && n > 0}
q,r := 0,m;
{m == qn+r && 0 <= r}
while (r >= n)
{
    {m == qn+r && 0 <= r && r >= n}
    q,r := q + 1; r - n;
    {m == qn+r && 0 <= r}
}
{m == qn+r && 0 <= r && !(r >= n)}
{m == qn+r && 0 <= r && r < n}

```

Por último aplicamos la regla de la asignación en cada una de las asignaciones. Para eso, trasladamos la postcondición a través de la postcondición efectuando la sustitución que esta indica.

```

{m >= 0 && n > 0}
{m == 0n+m && 0 <= m}
q,r := 0,m;
{m == qn+r && 0 <= r}
while (r >= n)
{
    {m == qn+r && 0 <= r && r >= n}
    {m == (q+1)n + (r-n) && 0 <= r-n}
    q,r := q + 1; r - n;
    {m == qn+r && 0 <= r}
}
{m == qn+r && 0 <= r && !(r >= n)}
{m == qn+r && 0 <= r && r < n}

```

Finalmente, las tres parejas de condiciones consecutivas corresponden a aplicaciones de la regla de la consecuencia. Estas tres implicaciones a demostrar son puramente matemáticas y no involucran código. Lo que hemos mostrado a través de este ejemplo es que mecánicamente es posible reducir la corrección de un programa anotado a teoremas matemáticos puros.

$$H1) \{m \geq 0 \ \&\& \ n > 0\}$$

$$T1) \{m == 0n+m \ \&\& \ 0 \leq m\}$$

$$H2) \{m == qn+r \ \&\& \ 0 \leq r \ \&\& \ r \geq n\}$$

$$T2) \{m == (q+1)n + (r-n) \ \&\& \ 0 \leq r-n\}$$

$$H3) \{m == qn+r \ \&\& \ 0 \leq r \ \&\& \ !(r \geq n)\}$$

$$T3) \{m == qn+r \ \&\& \ 0 \leq r \ \&\& \ r < n\}$$

Estos teoremas, son los que el programador debe demostrar para determinar la corrección del código. Se les suele llamar *proof obligations* o condiciones de verificación. En este caso la demostración de los mismos es trivial.

2.6. Verificación contra derivación

Interiorizarse con el proceso de derivación lleva un tiempo considerable. Programar derivando invariantes requiere práctica y un cambio radical en la manera de diseñar algoritmos con respecto a cómo se hace en el paradigma actual.

Por lo general, para lograr un programa que cumpla con las reglas y axiomas de la lógica de Hoare, es más fácil construir el código a partir del proceso de derivación de las invariantes dado que se sigue una línea de razonamiento.

Tratar de deducir las invariantes de un *loop* a partir de un programa ya escrito, es una tarea muy difícil ya que hay que encontrar invariantes que se preserven al principio, durante la ejecución y junto a la condición de terminación, que en este caso ya está determinada desde un inicio.

2.7.Variante

Como vimos anteriormente, la lógica de Hoare no dispone de ningún recurso para demostrar que el código termina su ejecución [10]. Para resolver este problema, se puede utilizar el método de la variante.

Para garantizar la terminación de un código que contiene iteraciones, además de una invariante **i**, hay que asociarle a cada *loop* una variante, la cual llamaremos **v**. La variante es una expresión entera que cumple los siguientes requisitos:

$$- \quad \exists v, i: i \Rightarrow v \geq k$$

Esto quiere decir que la invariante debe estar acotada inferiormente (normalmente $k=0$).

$$- \quad \forall a: \{i \ \&\& \ v == a\} \text{ cuerpo del loop } \{v < a\}$$

Esto quiere decir que el cuerpo del *loop* asegura que la variante disminuye.

En la práctica, se sugiere hacer esta asociación desde los *loops* más internos hasta los más externos, para hacer que la complejidad crezca gradualmente. Cabe señalar que existe software capaz de determinar la variante automáticamente para algunos casos.

2.8.Corrección total

En resumen, un programa es totalmente correcto cuando se cumplen las precondiciones y postcondiciones y además cualquier ejecución de un estado que satisfaga la precondición es finita.

Si bien existen varios métodos para garantizar la corrección total de programas, en este trabajo seguiremos la lógica de Hoare y el método de la variante.

3. Herramientas para verificar corrección

La verificación de un programa utilizando la lógica de Hoare se puede convertir en una tarea tediosa al trabajar con algoritmos no triviales. En este trabajo, buscamos investigar y probar herramientas capaces de automatizar dicha tarea y que además hicieran uso del marco teórico estudiado. Además, dada su popularidad en la industria, tratamos de que dicha herramienta fuese orientada a la programación imperativa [5].

3.1.Frama-C

Frama-C (Framework for Modular Analysis of C programs) es plataforma que agrupa varias herramientas en forma de *plug-ins* (módulos) para el lenguaje C. Este *framework* dispone de una interfaz de usuario amigable y facilita el análisis de comportamiento, chequeo de tipos, verificación formal, generación de árboles de sintaxis y otras funcionalidades que escapan el objetivo de este trabajo.

Cuenta con soporte para el lenguaje de anotaciones ACSL (ANSI/ISO C Specification Language) el cual permite declarar precondiciones, postcondiciones e invariantes. Cabe destacar que se deben especificar propiedades relacionadas al manejo de la memoria, tarea que puede tornarse difícil, dependiendo de la naturaleza del programa en cuestión.

Dispone de dos módulos para verificar la corrección de los programas:

- **Jessie** – Este sistema transforma las anotaciones en ACSL mediante la plataforma *Why* en condiciones de verificación (*proof obligations*) que luego pueden ser procesadas por demostradores de teoremas automáticos como *Z3*, *Simplify*, *Alt-Ergo* o demostradores de teoremas interactivos como *Coq* [14].
- **WP** - Presenta un funcionamiento similar al módulo *Jessie* pero permite la verificación de programas a más bajo nivel (manejo de variables de Hoare con punteros). También puede utilizar la plataforma *Why* para invocar a demostradores de teoremas automáticos e interactivos [15].

Luego de leer documentación y experimentar brevemente con Frama-C, concluimos que Dafny era la herramienta más apropiada para este trabajo principalmente a que presenta un alto grado de automatismo. Dafny encapsula todo lo que respecta a la verificación formal de un programa en una sola herramienta (su compilador verificador) y no requiere que el usuario aprenda a utilizar diversos *plug-ins* ni demostradores de teoremas.

3.2.Dafny

3.2.1. Breve descripción de la herramienta

Dafny es un lenguaje de programación de código abierto diseñado para verificación estática de programas. Fue diseñado y es mantenido activamente por miembros de del grupo “Research in Software Engineering (RiSE)” de Microsoft Research.

Es un lenguaje imperativo, secuencial, fuertemente tipado, genérico, modular y presenta algunas características de la orientación a objetos. Es de tipado estático y realiza una mínima inferencia en expresiones.

El lenguaje tiene integrado un sistema de anotaciones (palabras claves) para poder especificar precondiciones, postcondiciones, invariantes, variantes y propiedades de framing. Para poder realizar especificaciones complejas, se pueden crear funciones recursivas y utilizar tipos predefinidos como sets y secuencias. Mediante el uso de estos recursos, el compilador verificador automático puede validar la corrección funcional de los programas. Cabe señalar que este tipo de anotaciones, solo se utilizan para la verificación del programa y no se encuentran presentes en el código ejecutable [4].

La interacción con el compilador verificador no difiere con la de un compilador de otro lenguaje. Vale la pena señalar que cualquier error de verificación impide que el programa sea compilado. Esto quiere decir que el código compilado por Dafny siempre es correcto según su especificación. Existe la posibilidad de ejecutar el verificador sin iniciar el proceso de compilado.

Existen dos principales construcciones de alto nivel: **métodos** y **funciones**.

Los métodos, denotados con la palabra clave **method** son procedimientos imperativos que reciben parámetros pasados por valor. La firma de estos métodos incluye el nombre y tipo de la variable de retorno. Por lo general los métodos se utilizan para realizar cálculos o ejecutar

instrucciones para lograr resolver determinado problema, pero también se pueden utilizar para proporcionar pruebas al verificador de ser necesario.

Las funciones, denotadas con la palabra clave **function** (o **predicate** si su resultado es booleano), son expresiones que admiten recursión. Utilizar el paradigma funcional nos permite formular de manera relativamente sencilla especificaciones matemáticas de condiciones u operaciones [16].

Los métodos y las funciones se pueden utilizar para generar código ejecutable o como forma de especificación y/o verificación. Estos últimos, son llamados **ghost** (fantasma en inglés) ya que participan solamente del proceso de verificación. Esta palabra clave también sirve para variables y parámetros. Las funciones son ghost por defecto pero si se desea crear un método con el mero hecho de formar parte de una verificación, se debe declarar como un **lemma**.

Las funciones, predicados, métodos y lemas pueden ser anotados con precondiciones y postcondiciones utilizando las palabras clave **requires** y **ensures** respectivamente, seguidas de una expresión o una llamada a una función.

El lenguaje incluye tipos como enteros, naturales, *arrays* genéricos multidimensionales, caracteres, booleanos y números reales (implementados como cocientes de enteros de precisión arbitraria). Cabe señalar que esta implementación para los números reales limita las aplicaciones del lenguaje significativamente. Además, contiene tipos inmutables como secuencias, sets, multisets, maps que son utilizados principalmente para especificación [16].

Si bien Dafny presenta algunas características de la programación orientada a objetos, no podemos afirmar que se trata de un lenguaje de programación orientado a objetos. Por el momento se pueden declarar clases, pero no soportan herencia y el sistema soporte para *generics* está en desarrollo.

Pese a que Dafny compila código ejecutable, la única interacción disponible con el usuario hasta el momento es por medio de la función *print*. El compilador verificador tiene la opción de generar *dlls* para poder utilizar las funciones desde cualquier otro lenguaje de la plataforma .NET. Además puede generar código en C# que si bien se puede utilizar para integrar en otros proyectos, el resultado es muy similar al código generado por herramientas de 4ta generación y no es legible ni mantenible [17].

Si bien el compilador verificador se puede utilizar por línea de comandos, existe una extensión para Visual Studio que provee al programador con reserimiento de sintaxis y verificación en tiempo real mientras que se escribe código. De esta manera, los errores se muestran relacionados con líneas de código además de obtener información adicional al colocar el puntero del mouse sobre determinados elementos.

3.2.2. Funcionamiento

Dafny está basado en el proyecto *Boogie* de Microsoft Research. A diferencia de Dafny, Boogie es solo un lenguaje de verificación y no permite generar código ejecutable. Boogie utiliza el demostrador de teoremas (SMT *solver*) Z3 [18] que se encarga de automatizar las demostraciones de las condiciones de verificación vistas en el fundamento teórico.

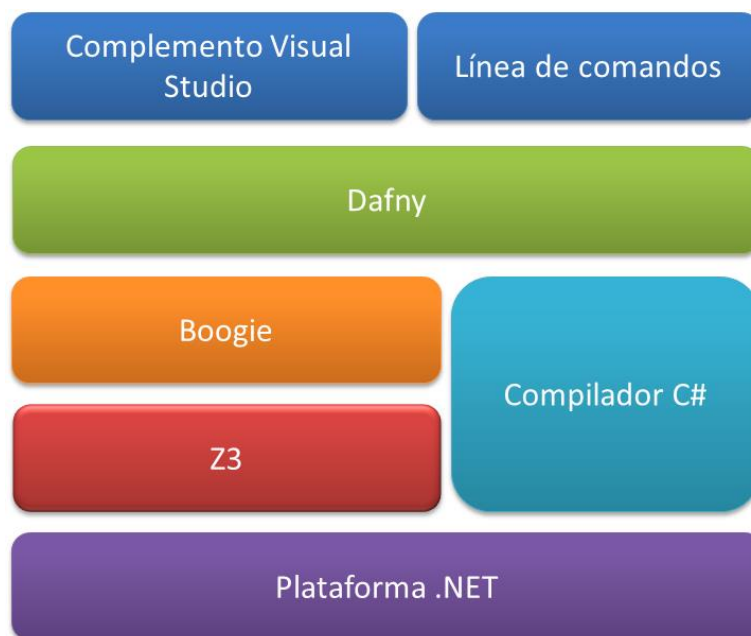


Imagen 3 – 1 Arquitectura Dafny.

Al iniciar el proceso de compilación verificación de un programa, Dafny *parsea* el código, verifica la sintaxis del programa y hace un chequeo de tipos. Si no hay errores, convierte la especificación y los recursos necesarios para la verificación de corrección a Boogie.

Boogie se encarga de generar las pruebas de verificación que luego son ejecutadas por el demostrador de teoremas Z3 [18]. Una vez que el Z3 procesa las condiciones de verificación, el compilador genera código en C# que incluye la implementación del programa y de los tipos incluidos en el lenguaje. Luego este código es procesado por el compilador de C# y es compilado como un ejecutable de la plataforma .NET. Si no hay ningún error de tipos, verificación o compilación, el proceso será transparente para el usuario [17].

Tras bambalinas, Boogie convierte la especificación y el código fuente a ternas de Hoare y aplica los axiomas y las reglas de dicha lógica para lograr generar pruebas de verificación. Además utiliza el método de la variante para determinar si la ejecución del programa en cuestión finaliza.

Esto elimina la necesidad de realizar tediosas demostraciones en papel para realizar la demostración de corrección de un programa. La única tarea que deben realizar los desarrolladores es la de determinar la especificación del programa utilizando las anotaciones proporcionadas por Dafny y escribir código extra en caso de las pruebas para las dichas demostraciones sean insuficientes.

Es por eso que Dafny demuestra la corrección total de los programas que compila.

3.2.3. Relación con el fundamento teórico

La sintaxis de Dafny no difiere de la que podemos encontrar en cualquier lenguaje imperativo. Estas son las mismas estructuras de control a las que hace referencia la lógica de Hoare. Además, cuenta con la palabra clave **decreases** que sirve para indicar cuál es la variante en los *loops*. Por lo general, Dafny puede determinar la variante de un *loop* de forma automática, pero si este procedimiento falla, le indicará al usuario que ingrese una de forma manual.

La siguiente figura muestra una versión simplificada de las reglas para determinar la corrección parcial de un programa utilizando la sintaxis de Dafny. Cabe señalar que no se tomaron en cuenta palabras claves como **break** y **retrun**.

$$\begin{array}{c}
\frac{\{p\} \text{ s}_1 \{r\} \{r\} \text{ s}_2 \{q\}}{\{p\} \text{ s}_1; \text{ s}_2 \{q\}} \qquad \frac{\frac{\{p \wedge e_1\} \text{ s}_1 \{q\}}{\{p \wedge \neg e_1\} \text{ s}_2 \{q\}}}{\{p\} \text{ if } e_1: \text{ s}_1 \text{ else: } \text{ s}_2 \{q\}} \qquad \frac{\{e_1 \wedge e_2\} \text{ s } \{e_2\}}{\{e_2\} \text{ while } e_1 \text{ invariant } e_2: \text{ s } \{\neg e_1 \wedge e_2\}}
\end{array}$$

Imagen 3 – 2 Reglas de Hoare con *keywords* de Dafny.

Cada vez que se desea llamar a un método o a una función, nos debemos asegurar que las precondiciones que esta presenta en su firma se cumplen. Para determinar la corrección de un *loop*, Dafny determina si las siguientes afirmaciones se cumplen:

- La invariante se cumple antes del comienzo del *loop*.
- La invariante se cumple luego de ejecutar el cuerpo del *loop*.
- La invariante y la negación de la condición del *loop* implican la propiedad deseada al final del *loop*.
- La variante no es negativa cuando la invariante del *loop* se mantiene.
- La variante decrementa su valor en cada paso del *loop* siempre y cuando la invariante y la condición del *loop* se mantengan.

Estas reglas, hacen referencia al marco teórico visto en el fundamento teórico, es por eso que podemos afirmar que Dafny está basado en la lógica de Hoare y en el método de la variante.

Cabe señalar que Dafny soporta la asignación en paralelo. Esto significa que el lado derecho de las expresiones son calculadas antes de ser asignadas a las variables. En Dafny, el lado derecho de una expresión de asignación puede ser una llamada a un método siempre y cuando el número de variables de la salida del método coincidan con el número de variables a la izquierda de la expresión.

4. Ejemplos de derivación en Dafny

En este capítulo veremos en detalle algunos ejemplos del razonamiento seguido para derivar algoritmos en Dafny en base a sus precondiciones y postcondiciones.

4.1. Búsqueda secuencial

4.1.1. Descripción

Este algoritmo simple tiene por objetivo comparar uno a uno los elementos de un *array* (no vacío) con un determinado valor y retornar el valor de la posición donde se encuentra el mismo.

4.1.2. Especificación

En este caso implementaremos un algoritmo que opera en un *array* de números reales y recibe como parámetro de búsqueda otro número real. Si el elemento buscado es encontrado, se debe retornar su posición dentro del *array* en una variable llamada **ret**. En caso de no encontrar el elemento buscado, el programa debe devolver el número arbitrario -1.

Precondiciones:

- requires $a \neq \text{null}$;

Postcondiciones:

- ensures $\text{ret} \geq 0 \implies \text{ret} < a.Length \ \&\& \ a[\text{ret}] == \text{key}$;
- ensures $\text{ret} < 0 \implies \text{forall } k :: 0 \leq k < a.Length \implies a[k] \neq \text{key}$;

4.1.3. Diseño de la invariante

En este caso utilizaremos un único *loop*. Dentro del mismo manejamos una variable para indicar la posición del *array* a visitar la cual llamaremos **index** y otra para retornar el dato encontrado llamada **ret**.

En este ejemplo, deseamos que la variable de posición **index** indique posiciones válidas dentro del *array*. Además, sabemos que cuando el programa encuentra el valor buscado, finaliza su ejecución. Por ende, el lado del *array* a la izquierda de **index** contendrá valores distintos al valor buscado.

Invariantes:

- invariant $0 \leq \text{index} \leq \text{a.Length}$;
- invariant forall $k :: 0 \leq k < \text{index} \Rightarrow \text{a}[k] \neq \text{key}$;

4.1.4. Derivación

Inicialización:

- `index := 0;` // Recorreremos el *array* desde la izquierda hacia la derecha.
- `ret := -1;` // Se mantendrá este valor si no encontramos el valor objetivo.

Terminación:

- `index == a.Length;` // El *loop* termina cuando se termina de recorrer el *array* o cuando se haya encontrado el valor buscado.

Paso:

- `index := index + 1;` // Preservamos la invariante y nos acercamos a la condición de terminación en cada paso.

Variante:

- decreases $a.Length - index$; // Decrece en cada paso y está acotado inferiormente.

4.1.5. Implementación

```
method has_sequential(a: array<int>, key: int) returns (ret: int)
  requires a != null;
  requires a.Length > 0;
  ensures ret >= 0 ==> ret < a.Length && a[ret] == key;
  ensures ret < 0 ==> forall k :: 0 <= k < a.Length
    ==> a[k] != key;
{
  var index: int;
  index := 0;
  ret := -1;

  while (index != a.Length)
    invariant 0 <= index <= a.Length;
    invariant forall k :: 0 <= k < index ==> a[k] != key;
    decreases a.Length - index
    {
      if a[index] == key {
        ret := index;
        return;
      }
      index := index + 1;
    }
}
```

4.2. Selection Sort

4.2.1. Descripción

El ordenamiento por selección es un algoritmo de ordenamiento que busca el mínimo elemento entre una posición i y el final del *array* y luego lo intercambia con el elemento de la posición i .

4.2.2. Especificación

En este caso, implementaremos un algoritmo que ordena números enteros, por lo tanto deseamos que este reciba como entrada un *array* de enteros no nulo. Luego de terminada la ejecución, deseamos que los elementos estén ordenados ascendentemente. Además deseamos que el conjunto (con repetición) de elementos de entrada sea el mismo que el de salida. Formalmente:

Precondiciones:

- requires $a \neq \text{null}$;
- requires $a.\text{Length} > 0$;

Postcondiciones:

- ensures $\text{multiset}(a[..\]) == \text{old}(\text{multiset}(a[..\]));$
- ensures $\text{sorted}(a, 0, a.\text{Length});$

La primer postcondición se utiliza para indicarle al compilador verificador que los elementos del *array* solo varían de posición durante la ejecución del programa. Esto puede expresarse en Dafny como: los elementos de **a** (dentro de un *multiset*) antes de comenzar la ejecución del programa, deben coincidir con los elementos de **a** (dentro de otro *multiset*) una vez finalizada la ejecución del mismo. Nótese que la palabra clave *old* hace referencia al estado de la memoria antes de comenzar la ejecución del programa.

Donde *sorted* es la siguiente función booleana, que corresponde a la condición de que el *array* *a* está ordenado en el intervalo [*low*, *high*):

```
predicate sorted(a: array<int>, low: int, high :int)
  requires a != null;
  requires 0 <= low <= high <= a.Length;
  reads a;
  {
    forall j, k :: low <= j < k < high ==> a[j] <= a[k]
  }
```

4.2.3. Diseño de la invariante exterior

Este algoritmo utiliza dos *loops* anidados para lograr su objetivo. En esta implementación dividiremos el algoritmo en dos métodos. El primero se encargará de recorrer el *array* y de llamar al segundo, que seleccionará el mínimo elemento encontrado desde la posición *i* hasta el final. En cada paso del *loop* exterior, los elementos a la izquierda quedarán ordenados y además serán menores que los elementos que se encuentran del lado derecho. Además, los elementos del *array* no cambiarán.

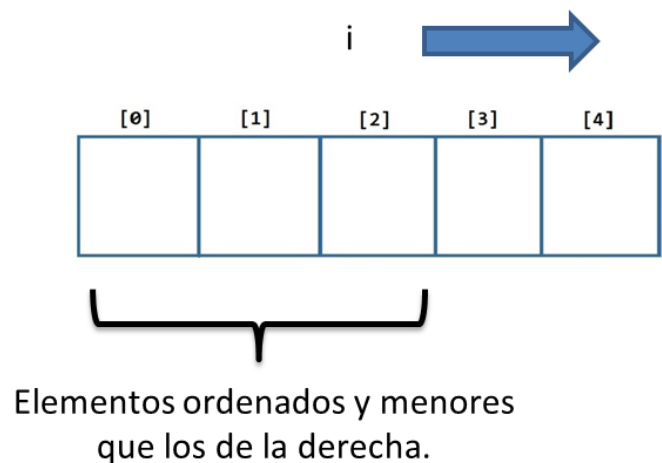


Imagen 4 – 1 Ilustración variable *i*.

Formalmente se puede expresar como:

Invariantes:

- invariant $0 \leq i < a.Length$;
- invariant $multiset(a[..]) == old(multiset(a[..]))$;
- invariant $sorted(a, 0, i)$;
- invariant forall $j, k :: 0 \leq j < i \leq k < a.Length \implies a[j] \leq a[k]$;

4.2.4. Derivación del método exterior

Inicialización:

- `i := 0; // Recorreremos el array desde la izquierda hacia la derecha.`

Terminación:

- `i > a.Length - 1; // El loop termina cuando se termina de recorrer el array.`

Paso:

- `min:=select(a, i); // Seleccionamos el elemento mínimo entre i y el final del array.`
- `a[i], a[min] := a[min], a[i]; // Intercambiamos el valor mínimo encontrado.`
- `i:=i+1; // Nos acercamos a la condición de terminación en cada paso.`

Variante:

Si bien Dafny logra detectar automáticamente la variante en esta implementación, la misma sería:

- `decreases a.Length-i; // Decrece en cada paso y está acotado inferiormente.`

4.2.5. Implementación

```
method selection_sort(a:array<int>)
  modifies a;
  requires a != null;
  requires a.Length > 0;
  ensures multiset(a[..]) == old(multiset(a[..]));
  ensures sorted(a, 0, a.Length);
{
  var min, i : int;
  i := 0;
  while(i < a.Length - 1)
    invariant 0 <= i < a.Length;
    invariant multiset(a[..]) == old(multiset(a[..]));
    invariant sorted(a,0,i);
    invariant forall j,k::0 <= j < i <= k < a.Length
      ==> a[j] <= a[k];
  {
    min := select(a, i);
    a[i], a[min] := a[min], a[i];
    i := i + 1;
  }
}
```

4.2.6. Especificación del método select

En este caso, implementaremos un algoritmo que busca el mínimo número entero dentro de un **array** no nulo, partiendo de una posición válida determinada hasta el final. Además deseamos que el conjunto (con repetición) de elementos de entrada sea el mismo que el de salida. Formalmente:

Precondiciones:

- requires a != null;
- requires 0 <= i < a.Length;

Postcondiciones:

- ensures multiset(a[..]) == old(multiset(a[..]));
- ensures i <= m < a.Length;
- ensures forall x :: i <= x < a.Length ==> a[m] <= a[x];

4.2.7. Diseño de la invariante

Es razonable pensar que necesitamos dos variables para poder solucionar este problema. La primera, debe de ser un índice para ir recorriendo el *array* (la cual llamaremos *j*) y la segunda para guardar la posición del mínimo elemento encontrado hasta el momento (la cual llamaremos *m*).

Sabemos que *j* debe ir desde una posición válida indicada hasta el final del *array* y que su valor inicial es asignado mediante el parámetro *i*. Luego de finalizada la ejecución de este método, deseamos que *m* efectivamente sea el valor mínimo para el rango ya mencionado y que además los elementos no son modificados por este algoritmo.

Invariantes:

- invariant $i \leq j < a.Length$;
- invariant $i \leq m < a.Length$;
- invariant forall $x :: i \leq x \leq j \implies a[m] \leq a[x]$;
- invariant $multiset(a[..]) == old(multiset(a[..]))$;

4.2.8. Derivación del método select

Inicialización:

- $j := i$; // Recorreremos el *array* desde la posición indicada hasta el final.
- $m := j$; // Posición del mínimo valor encontrado hasta el momento.

Terminación:

- $i > a.Length - 1$; // El *loop* termina cuando se termina de recorrer el *array*.

Paso:

- `j:=j+1; // Nos acercamos a la condición de terminación en cada paso.`
- `if (a[j] < a[m]) {m := j;}` // Actualizamos la posición del mínimo valor encontrado hasta el momento.

4.2.9. Implementación del método select

```
method select(a:array<int>, i :int) returns (m: int)
  requires a != null;
  requires 0 <= i < a.Length;
  ensures multiset(a[..]) == old(multiset(a[..]));
  ensures i <= m < a.Length;
  ensures forall x :: i <= x < a.Length ==> a[m] <= a[x];
  {
    var j := i;
    m := j;
    while(j < a.Length - 1)
      invariant i <= j < a.Length;
      invariant i <= m < a.Length;
      invariant forall x :: i <= x <= j ==> a[m] <= a[x];
      invariant multiset(a[..]) == old(multiset(a[..]));
      {
        j := j + 1;
        if (a[j] < a[m]){
          m := j;
        }
      }
  }
}
```

5. Algoritmos implementados en Dafny

A continuación se encuentran algoritmos que se implementaron en Dafny durante la realización de este trabajo. Los mismos fueron escritos en inglés y liberados bajo la licencia **GNU General Public License** como manera de contribución a la comunidad. Se encuentran publicados la siguiente URL: <https://github.com/narrowfail/formal>

5.1.1. Búsqueda de máximo en array

El objetivo de este algoritmo es buscar el entero más grande dentro de un *array* de números enteros.

```
// Array max one way
method max_one_way(a: array<int>) returns (mx: int)
  requires a != null;
  requires a.Length > 0;
  ensures forall j : int :: j >= 0 && j < a.Length ==> mx >= a[j];
  ensures exists j : int :: j >= 0 && j < a.Length && mx == a[j];
  {
    var index: int;
    // Inicialización: Recorremos el array de izquierda a derecha
    index := 0;
    mx := a[index];

    // Terminación: index > a.Length
    while (index < a.Length)
      invariant 0 <= index <= a.Length;
      invariant forall j : int :: j >= 0 && j < index ==> mx >= a[j];
      invariant exists j : int :: j >= 0 && j < a.Length && mx == a[j];
      decreases a.Length - index;
      {
        // Paso: Guardamos el elemento más grande hasta ahora
        if a[index] > mx {
          mx := a[index];
        }

        // Paso: Nos acercamos a la condición de terminación
        index := index + 1;
      }
  }
}
```

El objetivo del siguiente algoritmo es similar al anterior con excepción de que recorre el *array* en el sentido opuesto.

```

method max_one_way2(a: array<int>) returns (mx: int)
  requires a != null;
  requires a.Length > 0;
  ensures forall j : int :: j >= 0 && j < a.Length ==> mx >= a[j];
  ensures exists j : int :: j >= 0 && j < a.Length && mx == a[j];
  {
    var index: int;
    // Inicialización: Recorremos el array de derecha a izquierda.
    index := a.Length - 1;
    mx := a[index];

    // Terminación: index > a.Length
    while (index >= 0)
      invariant -1 <= index <= a.Length;
      invariant forall j :int::j > index && j < a.Length ==> mx >= a[j];
      invariant exists j : int :: j >= 0 && j < a.Length && mx == a[j];
      decreases index;
      {
        // Paso: Guardamos el elemento más grande hasta ahora
        if a[index] > mx {
          mx := a[index];
        }

        // Paso: Nos acercamos a la condición de terminación
        index := index - 1;
      }
  }

```

El objetivo del siguiente algoritmo es buscar el entero más grande dentro de un *array* ordenado de números enteros utilizando dos punteros.

```

// Array max 2 way
method max_two_way(a: array<int>) returns (mx: int)
  requires a != null;
  requires a.Length > 0;
  ensures exists j : int :: j >= 0 && j < a.Length && mx == a[j];
  {
    var low, high: int;
    // Inicialización: Colocamos los punteros de búsqueda en
    // los extremos del array
    low := 0;
    high := a.Length - 1;
    mx := a[low];
  }

```

```

// Terminación: El programa termina cuando los punteros se cruzan
while (low != high)
invariant 0 <= low <= high < a.Length;
invariant exists j : int :: j >= low && j <= high && mx == a[j];
decreases high - low;
{
    // Paso: Nos acercamos a la condición de terminación moviendo
    // el puntero que se encuentra más lejos del máximo
    if a[low] > a[high] {
        high := high - 1;
    }
    else
    {
        low := low + 1;
    }
    // Paso: guardamos el máximo parcial
    mx := a[low];
}
}

```

5.1.2. Búsqueda en *array*

El objetivo de este algoritmo es buscar un entero dado (*key*) dentro de un *array* y retornar la posición del mismo. Si el elemento no se encuentra en dicho array, el resultado será -1.

```

method has_sequential(a: array<int>, key: int) returns (ret: int)
requires a != null;
requires a.Length > 0;
ensures ret >= 0 ==> ret < a.Length && a[ret] == key;
ensures ret < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
{
    // Inicialización: Recorremos el array de izquierda a derecha
    var index: int;
    index := 0;
    ret := -1;

    // Terminación: Cuando recorremos todo el array
    while (index < a.Length)
invariant 0 <= index <= a.Length
invariant forall k :: 0 <= k < index ==> a[k] != key
decreases a.Length - index
    {
        if a[index] == key {
            // Paso: Si encontramos el valor buscado, guardamos su
            // posición
            ret := index;
        }
    }
}

```

```

        // Paso: Nos acercamos a la condición de terminación
        index := index + 1;
    }
}

```

A continuación definimos **orden en un array** como funciones o predicados para luego poder utilizar dicha definición en precondiciones, postcondiciones o invariantes.

```

// Functional version
function sortedf(a: array<int>, index: int): bool
    requires a != null;
    requires index >= 0;
    requires index <= a.Length;
    decreases a.Length - index;
    reads a;
    {
        index == a.Length ||
        index == a.Length - 1 ||
        (a[index] < a[index + 1] && sortedf(a, index + 1))
    }

```

```

// Predicate version
predicate sorted(a: array<int>, l:int, u:int)
    reads a
    requires a != null
    {
        forall j, k :: 0 <= l <= j <= k < u <= a.Length ==> a[j] <= a[k]
    }

```

También definimos el predicado **pertenece a un array** (dentro de ciertos bordes) con el objetivo de reutilizar este en múltiples secciones del código.

```

// Belongs to array predicate
predicate array_member(a : array<int>, l : int, u : int, x : int)
    reads a;
    requires a != null;
    {
        0 <= l <= u < a.Length && exists j :: l <= j <= u && a[j] == x
    }

```

El objetivo de este algoritmo es buscar un entero dado (*key*) dentro de un *array* ordenado y retornan la posición del mismo. Si el elemento no se encuentra en dicho array, el resultado será -1.

```
method has_sequential_sorted(a: array<int>, key: int) returns (ret: int)
  requires a != null;
  requires a.Length > 0;
  requires sorted(a, 0, a.Length);
  ensures ret >= 0 ==> ret < a.Length && a[ret] == key;
  ensures ret < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
{
  // Inicialización: Recorremos el array de izquierda a derecha
  var index: int;
  index := 0;
  ret := -1;

  // Terminación: Cuando recorremos todo el array
  while (index < a.Length)
    invariant 0 <= index <= a.Length ;
    invariant forall k :: 0 <= k < index ==> a[k] != key
    decreases a.Length - index;
    {
      if a[index] == key {ret:=index; return;}
      if a[index] > key {return - 1;}

      // Paso: Nos acercamos a la condición de terminación
      index := index + 1;
    }
}
```

El objetivo del siguiente algoritmo es realizar una búsqueda binaria de un entero dado (*x*) dentro de un *array* ordenado y retornan la posición del mismo. Si el elemento no se encuentra en dicho array, el resultado será -1.

```
method binary_search(a : array<int>, x : int) returns (i : int)
  requires a != null;
  requires sorted(a, 0, a.Length);
  ensures (0 <= i < a.Length && a[i] == x) ||
    (i == a.Length && !array_member(a,0,a.Length-1,x))
{
  // Inicialización: Colocamos dos punteros en los extremos
  // del array y el puntero i en medio de ambos
  var l, r : int;
  l,r := 0, a.Length-1 ;
  i := (l+r)/2;
```

```

// Terminación: Cuando los punteros se cruzen o el puntero i a
// apunte al elemento buscado
while l <= r && x != a[i]
invariant 0 <= l <= a.Length
invariant -1 <= r < a.Length
invariant i == (l+r)/2
invariant array_member(a,0,a.Length-1,x)
          ==> array_member(a,l,r,x)
{
    // Paso: Nos acercamos a la condición de terminación
    // moviendo el puntero que se encuentra más lejos del
    // valor buscado
    if x < a[i] {r := i-1;}
    else if x > a[i] {l := i+1;}
    // Paso: Colocamos el puntero i en medio de los punteros
    // l y r
    i:=(l+r)/2;
}
if r<l {i:=a.Length;}
}

```

5.1.3. Radicación discreta

A continuación se presentan dos algoritmos de radicación discreta.

```

//Discrete square root.
method sqr(x:int) returns (r:int)
requires x > 0;
ensures r*r <= x && (r + 1) * (r+1) > x;
ensures 0 < r <= x;
{
    // Inicialización: r comienza en 0
    r := 0;
    // Terminación: Cuando en el siguiente paso r^2 sea mayor a x
    while (r + 1) * (r + 1) <= x
        invariant r*r <= x
        decreases x-r
    {
        // Paso: Nos acercamos a la condición de terminación
        r := r + 1;
    }
}

```

```

//Discrete square root
method sqr2(x:int) returns (r:int)
  requires x > 0;
  ensures r*r <= x && (r + 1) * (r+1) > x;
  ensures 0 < r <= x;
  {
    // Inicialización: r comienza en x
    r := x;

    // Terminación: Cuando r^2 sea menor o igual a x
    while r*r > x
    invariant (r + 1)*(r+1) > x
    decreases r
    {
      // Paso: Nos acercamos a la condición de terminación
      r := r - 1;
    }
  }

```

5.1.4. División natural

A continuación se presentan dos algoritmos de división natural. Además, se definieron funciones para expresar axiomas matemáticos de la división.

```

function div(m:int,n:int) : int
  requires m >= 0
  requires n > 0
  {
    if m < n then 0 else 1 + div (m-n,n)
  }

```

```

function rem(m:int,n:int) : int
  requires m >= 0
  requires n > 0
  ensures m == div(m,n)*n + rem(m,n)
  ensures 0 <= rem(m,n) < n
  {
    if m < n then m else rem (m-n,n)
  }

```

El objetivo del siguiente algoritmo es realizar la división entera de **m** entre **n** por medio de restas sucesivas y devolver el resultado en la variable **q** y el resto en la variable **r**.

```
method nat_division(m:int,n:int) returns (q:int,r:int)
  requires m >= 0
  requires n > 0
  ensures q == div(m,n)
  ensures r == rem(m,n)
  {
    // Init
    r,q := m,0 ;

    while (r>=n)
      invariant r >= 0;
      invariant q + div(r,n) == div(m,n);
      invariant rem(r,n) == rem(m,n);
      //decreases r - Dafny pudo inferir esto
      {r, q := r-n, q+1;}
  }
```

Otra variante del mismo algoritmo:

```
method nat_division2(m:int,n:int) returns (q:int,r:int)
  requires m >= 0
  requires n > 0
  ensures m == q*n + r
  ensures r < n
  {
    // Init
    q, r := 0, m;

    while (r >= n)
      invariant m == q*n + r
      {q, r := q+1, r-n;}
  }
```

5.1.5. Potenciación

A continuación se presentan dos algoritmos de potenciación. Además, se definieron funciones para expresar axiomas matemáticos de dichas operaciones.

```
function method pow(a: int, b: int): int
  // Dafny no puede demostrar que pow(m,2*n) == pow(m*m,n), se debe
  // crear un lemma (abajo)
  requires b >= 0;
  {
    if b == 0 then 1
    else if b == 1 then a
    else a * pow(a, b - 1)
  }
```

```
lemma L_pr_pow (m:int, n:int)
  requires m >= 0 && n >= 0;
  ensures pow(m*m,n) == pow(m,2*n);
  {}
```

El objetivo del siguiente algoritmo es calcular **a** elevado a la **b** por medio de multiplicaciones sucesivas y retornar el resultado en la variable **r**.

```
method power_simple(a: nat, b: nat) returns (r: int)
  requires b >= 0;
  ensures r == pow(a, b);
  {
    var x: int;
    // Init
    x := 0;
    r := 1;

    while (x != b)
      invariant 0 <= x <= b;
      invariant r == pow(a, x);
      decreases b - x;
      {
        r := r * a;
        x := x + 1;
      }
  }
```

La exponenciación binaria es un algoritmo utilizado para calcular de forma rápida potencias enteras de un número **m** dado. Implícitamente utiliza la expansión binaria del exponente.

```

method power_binary(m:int, n:int) returns (r:int)
  requires m >= 0 && n >= 0;
  ensures r == pow(m,n);
  {
    var x:int, y:int;
    x,y,r := m,n,1 ;
    while y != 0
      invariant x >= 0 && y >= 0;
      invariant pow(m,n) == r * pow(x,y);
      {
        if y % 2 == 0 {
          L_pr_pow (x,y/2); // Lemma
          x, y := x*x, y/2;
        }
        else {
          y, r := y - 1, r * x;
        }
      }
    }
  }

```

5.1.6. Máximo común divisor

A continuación se presentan dos algoritmos de búsqueda del máximo común divisor entre dos enteros. Se definió una versión un funcional y una imperativa.

```

// Functional GCD
function gcd(m: int, n: int): int
  requires m > 0;
  requires n > 0;
  decreases m+n;
  {
    if (n==m) then n
    else if (m>n) then gcd(m-n, n)
    else gcd(m, n-m)
  }

// GCD
method gcd2(a: int, b: int) returns (r: int)
  requires a > 0;
  requires b > 0;
  ensures r == gcd(a, b);
  {
    var x: int;

```

```

//Init
r := a;
x := b;

while (r != x)
  invariant x > 0;
  invariant r > 0;
  invariant gcd(r, x) == gcd(a, b);
  decreases x+r;
  {
    if (r > x)
    {
      r := r - x;
    }
    else
    {
      x := x - r;
    }
  }
}

```

5.1.7. Logaritmo discreto

A continuación se presentan dos algoritmos de búsqueda del logaritmo discreto. Se definió un **lemma** algebraico para lograr que Dafny pudiese verificar el paso del método **skl**.

```

lemma mon_prod (x : int, y : int)
  requires x > 0;
  requires y > 1;
  ensures x < x*y;
  {
    assert 0 < x*(y-1) == x*y - x;
  }

method skl(m:nat, n:nat) returns (l:nat, r:nat)
  requires (m > 0 && n > 1);
  ensures r > 0;
  ensures (m == pow(n,l) * r && r%n != 0);
  {
    l, r := 0, m;
    while (r % n == 0)
      invariant m == pow(n,l) * r
      decreases r
      {
        mon_prod(r/n, n);
        l,r := l+1, r/n;
      }
  }
}

```

```

method log_int(a:int, b:int) returns (r: int)
  requires b > 1;
  requires a > 0;
  ensures r >= 0;
  ensures pow(b, r) <= a;
  ensures pow(b, r + 1) > a;
  {
    var next: int;
    // Init
    r := 0;
    next := power_simple(b, r + 1);

    while (next <= a)
      invariant 0 <= r;
      invariant next == pow(b, r + 1);
      invariant pow(b, r) <= a;
      decreases a - pow(b, r + 1);
      {
        r := r + 1;
        next := power_simple(b, r + 1);
      }
    return r;
  }

```

5.1.8. Multiplicación utilizando sumas

A continuación se presenta un simple algoritmo de multiplicación por sumas sucesivas.

```

method multiply_sum(a: int, b: int) returns (r: int)
  requires a >= 0;
  requires b >= 0;
  ensures r == a*b;
  {
    var x: int;
    //Init
    r := 0;
    x := 0;

    while (x != b)
      invariant 0 <= x <= b;
      invariant r == a * x;
      decreases b - x;
      {
        r := r + a;
        x := x + 1;
      }
  }

```

5.1.9. Operaciones a nivel de bits

A continuación se presenta un algoritmo para realizar un *shift* circular en un *array* de booleanos. Este algoritmo puede oficiar de base para demostrar algoritmos criptográficos.

```
method circular_shift_right(a:array<bool>)
  modifies a;
  requires a != null;
  requires a.Length > 1;
  ensures a[0] == old(a[a.Length - 1]);
  ensures forall j :: 0 < j < a.Length ==> a[j] == old(a[j-1]);
  {
    var i : int;
    var tmp : bool;
    i := a.Length - 1;
    tmp := a[a.Length - 1];

    while(i > 0)
      invariant 0 <= i < a.Length;
      invariant a[0..i] == old(a[0..i]);
      invariant forall j :: i < j < a.Length ==> a[j] == old(a[j-1]);
      {
        a[i] := a[i-1];
        i := i - 1;
      }
      a[0] := tmp;
    }
  }
```

5.1.10. Búsqueda ceros

```
method findzero (a:array<int>) returns (i:int)
  requires a != null;
  requires forall i :: 0 <= i < a.Length ==> a[i] >= 0;
  requires forall i :: 0 < i < a.Length ==> a[i] >= a[i-1] - 1;
  ensures 0 <= i <= a.Length;
  ensures forall j :: 0 <= j < i ==> a[j] != 0;
  ensures i == a.Length || a[i]==0;
  {
    i:=0;
    while (i != a.Length && a[i] != 0)
      invariant 0 <= i <= a.Length;
      invariant forall j :: 0 <= j < i ==> a[j] != 0;
      {
        no_zero(a,i);
        i := min(i+a[i], a.Length);
      }
  }
}
```

```

function method min(a:int, b:int):int
{
    if a < b then a else b
}

// Demostramos que no hay ceros desde I hasta a[i]+i
// Sin modificar el array, le mostramos a Dafny que Pre ==> Post.

lemma no_zero (a:array<int>, i: int)
requires a != null;
requires forall i :: 0 <= i < a.Length ==> a[i] >= 0;
requires forall i :: 0 < i < a.Length ==> a[i] >= a[i-1]-1;
requires 0 <= i < a.Length;
requires a[i] != 0;
ensures forall j :: i <= j < min(a[i]+i, a.Length) <= a.Length ==> a[j] !=
0;
{
    var k: int;
    k := i;
    while (k != min(a[i]+i, a.Length))
    invariant i <= k <= min(a[i]+i, a.Length);
    invariant forall j :: i <= j < k ==> a[j] != 0;
    invariant k != min(i+a[i], a.Length) ==> a[k] >= a[i]-(k-i) > 0;
    {
        k := k + 1;
    }
}

```

5.1.11. Máximo provecho en bolsa

A continuación se muestra la implementación de un algoritmo que busca el máximo provecho a la hora de comprar y vender acciones en bolsa.

```

predicate segment_min (a : array<int>, l : int, u : int, m : int)
    reads a;
    requires a != null;
    requires 0 <= l < u <= a.Length;
    {
        l <= m < u && forall j :: l <= j < u ==> a[m] <= a[j]
    }

predicate segment_max_diff (a : array<int>, l : int, u : int, left : int,
right : int)
    reads a;
    requires a != null;
    requires 0 <= l < u <= a.Length;
    {

```

```

l <= left < right < u &&
forall j, k :: l <= j < k < u ==> a[k] - a[j] <= a[right] - a[left]
}

```

```

method max_profit (a : array<int>) returns (l : int, r : int)
  requires a != null;
  requires a.Length >= 2;
  ensures segment_max_diff (a, 0, a.Length, l, r);
  {
    var i, m : int;
    i, m, l, r := 2, if a[0] <= a[1] then 0 else 1, 0, 1 ;
    while i != a.Length
      invariant 2 <= i <= a.Length;
      invariant segment_min (a, 0, i, m);
      invariant segment_max_diff (a, 0, i, l, r);
      {
        if (a[i] - a[m] > a[r] - a[l]){
          l,r := m,i ;
        }
        if (a[i] < a[m]){
          m := i ;
        }
        i := i + 1;
      }
    }
  }

```

5.1.12. Ordenamiento

A continuación se presentan algoritmos de ordenamiento para números enteros y predicados para definir ordenamiento. Se implementaron los siguientes algoritmos: *Bubble Sort*, *Insertion Sort*, *Selection Sort* y dos versiones del *Dutch National Flag Sort*.

```

predicate sorted(a: array<int>, low: int, high :int)
  requires a != null;
  requires 0 <= low <= high <= a.Length;
  reads a;
  {
    forall j, k :: low <= j < k < high ==> a[j] <= a[k]
  }

```

```

predicate leq_partitioned(a:array<int>, r :int)
  reads a;
  requires a != null;
  {
    forall j, k :: 0 <= j < r <= k < a.Length ==> a[j] <= a[k]
  }

```

```

method bubble_sort(a:array<int>)
  modifies a;
  requires a != null;
  ensures multiset(a[..]) == old(multiset(a[..]));
  ensures sorted(a,0,a.Length)
  {
    var r:int;
    r := a.Length;
    while(r != 0)
      invariant a.Length >= r >= 0;
      invariant multiset(a[..]) == old(multiset(a[..]));
      invariant sorted(a,r,a.Length);
      invariant leq_partitioned(a,r);
      {
        var i : int;
        i := 0;
        while(i+1 != r)
          invariant 0 <= i < r;
          invariant forall j :: 0 <= j < i ==> a[j] <= a[i];
          invariant leq_partitioned(a,r);
          invariant sorted(a,r,a.Length);
          invariant multiset(a[..]) == old(multiset(a[..]));
          {
            if(a[i] > a[i+1]) {
              a[i], a[i+1] := a[i+1], a[i];
            }
            else {}
            i := i+1 ;
          }
          r := r-1 ;
        }
      }
  }

```

// Insertion Sort

```

method insertion_sort(a:array<int>)
  modifies a;
  requires a != null;
  ensures multiset(a[..]) == old(multiset(a[..]));
  ensures sorted(a,0,a.Length)
  {
    var i : int;
    i := 0;
    while(i < a.Length)
      invariant 0 <= i <= a.Length;
      invariant multiset(a[..]) == old(multiset(a[..]));
      invariant sorted(a,0,i);
      {
        insert(a, i);
        i := i + 1;
      }
  }

```

```

method insert(a:array<int>, i :int)
  modifies a;
  requires a != null;
  requires 0 <= i < a.Length;
  requires sorted(a, 0, i);
  ensures sorted(a, 0, i+1);
  ensures multiset(a[..]) == old(multiset(a[..]));
  {
    var j : int;
    j := i;
    while(j > 0 && a[j-1] > a[j])
      invariant 0 <= j <= i;
      invariant sorted(a,0,j);
      invariant sorted(a,j,i+1);
      invariant multiset(a[..]) == old(multiset(a[..]));
      invariant forall m, n :: 0 <= m < j < n <= i ==> a[m] <= a[n];
      invariant a[j] == old(a[i]);
      {
        a[j], a[j-1] := a[j-1], a[j];
        j := j - 1;
      }
  }
}

// Selection Sort
method selection_sort(a:array<int>)
  modifies a;
  requires a != null;
  requires a.Length > 0;
  ensures multiset(a[..]) == old(multiset(a[..]));
  ensures sorted(a, 0, a.Length);
  {
    var min, i : int;
    i := 0;
    while(i < a.Length - 1)
      invariant 0 <= i < a.Length;
      invariant multiset(a[..]) == old(multiset(a[..]));
      invariant sorted(a,0,i);
      invariant forall j, k :: 0<=j<i <= k < a.Length ==> a[j] <= a[k];
      {
        min := select(a, i);
        // Swap
        a[i], a[min] := a[min], a[i];
        // Step
        i := i + 1;
      }
  }
}

```

```

method select(a:array<int>, i :int) returns (m: int)
  requires a != null;
  requires 0 <= i < a.Length;
  ensures multiset(a[..]) == old(multiset(a[..]));
  ensures i <= m < a.Length;
  ensures forall x :: i <= x < a.Length ==> a[m] <= a[x];
  {
    var j := i;
    m := j;
    while(j < a.Length - 1)
      invariant i <= j < a.Length;
      invariant i <= m < a.Length;
      invariant forall x :: i <= x <= j ==> a[m] <= a[x];
      invariant multiset(a[..]) == old(multiset(a[..]));
      {
        j := j + 1;
        if (a[j] < a[m]){
          m := j;
        }
      }
  }
}

// 0 = red, 1 = white, 2 = blue
method dnf(a: array<int>)
  modifies a;
  requires a != null;
  requires forall x :: 0<=x<a.Length==>(a[x]==0||a[x]==1||a[x] == 2);
  ensures sorted(a, 0, a.Length); // Flag order
  ensures multiset(a[..]) == old(multiset(a[..]));
  {
    var low, mid, hig : int;
    low, mid, hig := 0, 0, a.Length;
    while(mid != hig)
      invariant 0 <= low <= mid <= hig <= a.Length;
      invariant forall x :: 0 <= x < a.Length
        ==> (a[x] == 0 || a[x] == 1 || a[x] == 2);
      // Flag order:
      invariant forall x :: 0 <= x < low ==> a[x] == 0;
      invariant forall y :: low <= y < mid ==> a[y] == 1;
      invariant forall z :: hig <= z < a.Length ==> a[z] == 2;
      // Permutation
      invariant multiset(a[..]) == old(multiset(a[..]));
      {
        // Red
        if (a[mid] == 0)
        {
          a[low], a[mid] := a[mid], a[low];
          low := low + 1;
          mid := mid + 1;
        }
        else {
          // White

```

```

        if (a[mid] == 1){
            mid := mid + 1;
        }
        else {
            // Blue
            hig := hig - 1;
            a[hig], a[mid] := a[mid], a[hig];
        }
    }
}
}

```

datatype Color = red|white|blue

predicate color_sorted(a: Color, b: Color)

```

{
    a == red || b == blue || a == b
}

```

method dnf2(a:array<Color>)

```

    modifies a;
    requires a != null;
    ensures forall i,j::0<=i<j<a.Length ==> color_sorted(a[i],a[j]);
    ensures multiset(a[..]) == old(multiset(a[..]));
    {
        var r, w, b := 0, 0, a.Length;
        while ( w != b)
            invariant 0 <= r <= w <= b <= a.Length
            invariant forall i :: 0 <= i < r ==> a[i] == red;
            invariant forall i :: r <= i < w ==> a[i] == white;
            invariant forall i :: b <= i < a.Length ==> a[i] == blue;
            invariant multiset(a[..]) == old(multiset(a[..]));
            {
                match a[w]
                {
                    case red => a[r],a[w] :=a[w], a[r]; r := r + 1; w := w + 1;
                    case white => w := w + 1;
                    case blue => b := b - 1; a[w], a[b] := a[b], a[w];
                }
            }
    }
}

```

6. Lecciones aprendidas y conclusiones

6.1. Experiencias al desarrollar esta tesis

Para poder realizar esta tesis de investigación, se emplearon 6 meses para aprender el marco teórico y lograr comenzar a derivar / verificar algoritmos de complejidad baja y media. Además de la teoría expuesta en este trabajo, se estudiaron técnicas de derivación de invariantes, que fueron fundamentales para determinar las invariantes de los programas desarrollados. Se optó por obviar la explicación de dicha teoría ya que iba a requerir un esfuerzo considerable y pensamos que se desviaba del objetivo primordial de esta tesis: presentar algoritmos verificados formalmente.

Una vez que se decidió utilizar Dafny, comenzamos a experimentar con la herramienta y buscar documentación que sirviera de guía para comenzar a verificar algoritmos. Nos encontramos con que actualmente, la única documentación oficial disponible es un *tutorial* que va recorriendo como utilizar cada palabra clave del lenguaje y muestra algunos ejemplos prácticos, pero no explica la teoría que hay detrás de dichos ejemplos. Además, menciona el demostrador de teoremas Z3, pero no describe cómo funciona.

También se encontraron artículos académicos que describen el lenguaje a grandes rasgos y muestran ejemplos de verificación de algoritmos más complejos, así como contenido generado por la propia comunidad de usuarios.

Investigado como verificar algoritmos criptográficos (*TEA* y *Blowfish*), determinamos que el tiempo necesario requerido superaba ampliamente el tiempo estipulado para este trabajo y que existen investigaciones académicas que se centran sólo en ese propósito. Algoritmos como el QuickSort, se encontraban sin verificar utilizando esta metodología hasta el año 2015.

Una de las principales críticas que se le puede hacer a Dafny radica en que el demostrador de teoremas es transparente para el usuario. Cuando dicho componente no logra demostrar alguna condición de verificación, al usuario se le indica cuál condición no se cumple, pero no se le indica por qué motivo ni se muestran sugerencias de cómo abordar el problema. Esto requiere una alta capacitación por parte del usuario.

6.2. Conclusiones

Actualmente, el software desarrolla un rol crítico en diversos aspectos de nuestra sociedad y por ende es importante escribir software correcto. En este trabajo, estudiamos el paradigma de desarrollo actual y concluimos que si bien las organizaciones de desarrollo realizan rigurosas pruebas, las mismas no pueden asegurar formalmente la corrección total del software.

Luego, estudiamos un marco teórico para especificar correctamente el software y presentamos herramientas de verificación formal como la lógica de Hoare y el método de la variante para luego definir el concepto de corrección total.

Una vez establecida la teoría, investigamos el lenguaje imperativo experimental Dafny y derivamos algoritmos muy utilizados en la industria actual. Concluimos que si bien el lenguaje está en continuo desarrollo, la falta de tipos como los números de punto flotante y la carencia de soporte para concurrencia, lo hacen poco aplicable a nivel comercial.

La curva de aprendizaje de Dafny es pronunciada. Esto quiere decir que el conocimiento teórico requerido para poder comenzar a verificar programas simples es alto. Esto nos lleva a pensar que este tipo de verificación no es apta para técnicos ya que requiere conocimientos de nivel universitario.

Enseñar la teoría para poder lograr derivar algoritmos correctamente, es un verdadero desafío para los docentes ya que los alumnos, están acostumbrados a utilizar el paradigma actual.

Por otro lado, si comparamos esta herramienta con las existentes hace 5 años, el avance que se está realizando hace en esta área es destacable. Si herramientas como esta (o similares) continúan desarrollándose, es probable que en un futuro, se reduzca aún más la interacción con los humanos y las verificaciones, derivaciones y generación de código fuente, se realicen de forma automática.

7. Referencias bibliográficas

- [1] R. S. Pressman, *Software engineering: a practitioner's approach.*, Palgrave Macmillan, 2005.
- [2] J. Gleick, «A Bug and a Crash,» 1996. [En línea]. Available: <https://around.com/ariane.html>. [Último acceso: 3 Nov 2015].
- [3] AP, «Nissan Recalls Nearly 4 Million Cars With Air Bag Problems,» Bloomberg, 30 Abril 2016. [En línea]. Available: <http://www.bloomberg.com/news/articles/2016-04-30/nissan-recalls-nearly-4-million-cars-with-air-bag-problems>. [Último acceso: 28 Jun 2016].
- [4] K. R. M. Leino, «Dafny: An automatic program verifier for functional correctness,» de *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (pp. 348-370), Springer Berlin Heidelberg, 2010.
- [5] C. Abano, G. Chu, G. Eiseman, J. Fu y T. Yu, *Formal Verification with Dafny*, 2014.
- [6] I. ISO, «IEEE. 29148: 2011-Systems and software engineering-Requirements engineering,» 2011.
- [7] T. E. Damer, *Attacking faulty reasoning*, Cengage Learning, 2012.
- [8] T. Britton, L. Jeng, G. Carver, P. Cheak y T. Katzenellenbogen, «Reversible debugging software,» Judge Bus. School, Cambridge, UK, 2013.
- [9] C. A. R. Hoare, «Assertions: A personal perspective,» de *Software pioneers*, Berlin-Heidelberg, Springer, 2002, pp. 356-366.
- [10] C. A. R. Hoare, «An axiomatic basis for computer programming,» *Communications of the ACM*, p. 576–580, 1969.

- [11] D. J. Pearce y L. Groves, «Reflections on verifying software with Whiley,» Springer International Publishing, 2013.
- [12] J. Molina Bravo, «Docente Juan Molina Bravo,» [En línea]. Available: <http://www.lcc.uma.es/~jmmb/ttaadd/ttaadd2-2.pdf>. [Último acceso: 12 Nov 2015].
- [13] R. C. Backhouse, Program construction and verification, Prentice-Hall, Inc., 1986.
- [14] «Krakatoa and Jessie: verification tools for Java and C programs,» Laboratoire de Recherche en Informatique, [En línea]. Available: <http://krakatoa.lri.fr/>. [Último acceso: 8 Oct 2015].
- [15] P. Baudin, F. Bobot, L. Correnson y Z. Dargaye, *WP Plug-in Manual*, Paris: CEA LIST, Software Safety Laboratory, 2016.
- [16] Rise4fun, «Getting Started with Dafny: A Guide.,» RISE, [En línea]. Available: <http://rise4fun.com/Dafny/tutorial/guide>. [Último acceso: Oct 2015].
- [17] R. R. Rubio Cuéllar, «Verifying Algorithms and Data Structures in Dafny,» Universidad Complutense de Madrid, Madrid, 2016.
- [18] L. De Moura y N. Bjørner, «Z3: An efficient SMT solver,» de *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, 2008.