UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

AUTOMATIZACIÓN DEL ANÁLISIS DEL TIEMPO DE EJECUCIÓN DE PROGRAMAS PROBABILÍSTICOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

LUIS ENRIQUE PINOCHET GONZÁLEZ

PROFESOR GUÍA:
FEDERICO OLMEDO BERÓN

MIEMBROS DE LA COMISIÓN:
ÉRIC TANTER
JAVIER BUSTOS JIMÉNEZ

SANTIAGO DE CHILE
2022

Resumen

Los programas probabilísticos son un tópico relevante en la computación moderna, por lo tanto, razonar sobre su tiempo de ejecución asociado es un objetivo de alto interés, tanto en lo teórico como en lo práctico. Uno de los enfoques existentes para abortar esto último es la técnica de la transformada ert[·]. Esta transformada es una función recursiva sobre el conjunto de programas probabilísticos que permite calcular el tiempo de ejecución *esperado* de un programa dado.

Si bien, la transformada ert[·] posee suficiente expresividad para calcular de forma precisa el tiempo de ejecución de un programa, debido a su definición, posee algunos inconvenientes para ser utilizada en la práctica. El principal inconveniente es la elevada cantidad de cálculos necesarios para obtener un tiempo de ejecución válido. Otro inconveniente asociado es el cálculo del tiempo de ejecución de un ciclo `while`; esto requiere comprobar una desigualdad entre funciones. Este proceso no posee una metodología clara para llevarse a cabo y, en general, se resuelve mediante la manipulación algebraica de expresiones.

El objetivo general de este trabajo es desarrollar una herramienta que permita calcular, de forma automática, el tiempo de ejecución de un programa probabilístico, usando la transformada ert[·]. Para lograr esto, se procedió a implementar la función ert[·] y algunas estructuras asociadas en el lenguaje Haskell. Luego, a través de una herramienta denominada como *smt-solver*, se procedió a comprobar la condición asociada a los ciclos `while`. Para comprobar el desempeño de la herramienta desarrollada se diseñaron *tests* que permitieron comparar los resultados manuales con los cálculos automatizados. La herramienta logra reproducir los cálculos manuales para cada *test* desarrollado, por lo tanto, se concluye que el resultado final es positivo. Dado esto, es posible afirmar que el trabajo desarrollado es un primer acercamiento para que en un futuro la técnica ert[·] pueda ser utilizada en la práctica.

Le dedico esta memoria a mis padres que me han apoyado durante toda mi vida. Gracias por todo.

Agradecimientos

Agradezco en primer lugar a mis padres por darme la oportunidad de estudiar en la casa de estudios que yo quería. Gracias a Sergio, ojalá que nos sigamos ayudando por el resto de nuestras vidas. Gracias a Amanda, por apoyarme y acompañarme durante todo este proceso, dándome ánimos y ayudándome a sobrellevar los momentos más difíciles de mi carrera.

Gracias a Felipe y Benja, por ser mis amigos mas cercanos en el DCC y carrearme en tantas tareas. Gracias a Lucas, Candy, JP, Porro, Gaete, Barri, Marticorena, Basti, Max, Vale, Bego, Mariana y Graci, les deseo lo mejor en su vida personal y profesional. Gracias a Nicolas, por ser tan brígido y orientarme en tantas cosas. Gracias a Poblete, Armijo, Monse, MrSmoke, Villela y Max, por hacer de la Chacra una familia.

Gracias Yari, Juanita, Karla, Palena, Melissa, Lucho, Pérez, Pancho, Cristobal y Seba, por los lindos años vividos y compartidos en la peni.

Gracias al profesor Olmedo, por ayudarme y guiarme durante este largo proceso. Gracias además por darme la oportunidad de trabajar en un proyecto tan lindo como este. Gracias a Paula, por ayudarme a mejorar me deficiente escritura

Gracias a Hernán y Pablo Calvo por guiarme en mi carrera deportiva durante mi estadía en la universidad. Gracias a Pedro Aravena por introducirme al ajedrez, el que es sin duda el juego/deporte/ciencia más bonito del mundo.

Gracias a Mario, Dino, Melón, Santos, Miquel, Horacio y Lastra. Espero seguir contando con ustedes, aunque no siga en Linares. Gracias a mis tíos y familiares, ojalá sigamos en contacto por mucho tiempo más.

Tabla de Contenido

Capítulo 1

Introducción

1.1. Programas probabilísticos

Los algoritmos probabilísticos son, sin duda, un tópico importante en la computación moderna. Estos ofrecen aproximaciones eficientes a problemas importantes en casi todos los ámbitos de la ingeniería y la ciencia. Por ejemplo, el método Montecarlo para la integración, mediante la generación aleatoria de puntos, permite aproximar integrales que de otra forma serían muy difíciles de calcular. Otro ejemplo se puede encontrar en un área concreta como *machine learning*. En esta área encontramos la técnica de *dropout*. Esta consiste en apagar aleatoriamente neuronas durante el entrenamiento de una red neuronal, permitiendo de esta manera evitar el sobre ajuste de la misma y, en consecuencia, que la red logre representar un modelo generalizable frente a nuevos datos.

Para poder ejecutar un algoritmo probabilístico en un computador, primero se le debe representar con un programa probabilístico. Un programa probabilístico es simplemente un programa descrito con las construcciones estándares de los lenguajes de programación, a las cuales se le agrega una construcción adicional que permiten tomar muestras de distribuciones de probabilidad [6]. A modo de ejemplo, en el lenguaje Python la expresión `random.uniform(1, 10)` entrega una muestra de una variable aleatoria uniforme con soporte sobre el intervalo [1, 10].

Un ejemplo de programa probabilístico es el expuesto en la Figura 1.1. Se le denomina C_{trunc} debido a que representa el truncamiento de una distribución geométrica hasta el lanzamiento de la segunda moneda. El programa comienza con la evaluación de la guarda del `if` más externo, la que representa a una muestra de una distribución Bernoulli con parámetro $\texttt{p} = \frac{1}{2}$. Le asigna la probabilidad de $\frac{1}{2}$ a `true` y $1 - \frac{1}{2}$ a `false` (lo que se representa como $< \texttt{1/2} >$). Si el resultado observado es `true` el programa termina inmediatamente, asignándole el valor 1 a la variable `succ`. De no ser así se evalúa la guarda del `if` más interno, que también representa el lanzamiento de una moneda balanceada. Si el resultado observado es `true`, acaba el programa y se asigna 1 a la variable `succ`; por el contrario, si el resultado observado es `false` se le asigna 0 a `succ` terminando el programa.

```
if (< 1/2 >)
      {succ ≔ 1}
else {
      if (< 1/2 >)
            {succ ≔ 1}
      else{
            {succ ≔ 0}}
```

Figura 1.1: Ejemplo de programa C_{trunc}

1.2. Tiempos de ejecución

Dada la relevancia de los programas probabilísticos, es fundamental poder estimar de manera eficiente su tiempo de ejecución. Formalmente, el tiempo de ejecución de un programa probabilístico `C` es una variable aleatoria $\mathbb{X}$ que le asigna una probabilidad p_i a un tiempo de ejecución concreto t_i. Por ejemplo, considerando que el lanzamiento de una moneda y la operación de la asignación consume una unidad de tiempo, la variable aleatoria que modela el tiempo de ejecución del programa C_{trunc}, le asigna la probabilidad $\frac{1}{2}$ al tiempo de ejecución 2 y $\frac{1}{2}$ al tiempo de ejecución 3.

Dada la imposibilidad de obtener un solo valor como tiempo de ejecución, es natural estimar este a través del valor esperado de la variable aleatoria $\mathbb{X}$ o simplemente tiempo de ejecución promedio (o esperado) de `C`. Retomando el caso del programa C_{trunc}, su tiempo de ejecución promedio asociado es $\frac{5}{2}$ unidades de tiempo.

Para lograr este objetivo de calcular el tiempo de ejecución se han propuesto diversos enfoques. Por ejemplo, Wang *et al.* [12] aborda el problema basándose en teoría de martingalas. Los autores a través de martingalas logran estudiar los programas probabilísticos, permitiendo razonar sobre el tiempo de ejecución de los mismos.

En este trabajo, sin embargo, se adopta la técnica de la transformada $\mathsf{ert}[\cdot]$ (*expected runtime*) [8]. Esta transformada es definida de forma inductiva sobre la estructura de los programas probabilísticos (a cuyo conjunto llamaremos pProg) y permite calcular el tiempo de ejecución promedio de los mismos. Se elige este enfoque debido a que permite razonar con mayor precisión sobre los programas que la técnica basada en martingalas.

En la técnica de la transformada $\mathsf{ert}[\cdot]$ el tiempo de ejecución se representa a través de una función $f\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, donde Σ es el conjunto de los estados (asignación de valores a las variables del programa) a partir del cual se empieza a ejecutar el programa y $\mathbb{R}^{\infty}_{\geq 0}$ es el conjunto de los reales no-negativos junto con infinito. A su vez usamos $\mathbb{T} \triangleq \{f \mid f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$ para presentar el conjunto de los tiempos de ejecución. Dado lo anterior, la transformada $\mathsf{ert}[\cdot]$ tiene la siguiente firma:

$$\mathsf{ert}[\cdot] : \mathsf{pProg} \to (\mathbb{T} \to \mathbb{T})$$

Concretamente, $\mathsf{ert}[\mathtt{C}](f)(\sigma)$ entrega el tiempo de ejecución promedio (o esperado) del programa `C` partiendo desde un estado inicial σ y asumiendo que f captura el tiempo de ejecución del programa que le sigue a `C`. En particular, para estimar el tiempo de ejecución de un "único" programa `C` se toma $f \equiv \mathbf{0}$, es decir, como la función constante que a cada estado en Σ le asigna 0.

Antes de ilustrar informalmente el uso de la transformada $\mathsf{ert}[\cdot]$ a través de un ejemplo (la definición formal se dará en la Sección 3), es necesario aclarar que se usa el modelo de costo presentado anteriormente. Cada asignación y la evaluación de cada guarda (tanto de un condicional `if-then-else` como de un ciclo `while`) consume 1 unidad de tiempo. Con esto en mente, la aplicación de la transformada sobre el programa C_{trunc} procede como sigue:

Ejemplo $\mathsf{ert}\,[C_{trunc}](\mathbf{0})$

$$
\begin{aligned}
&\mathsf{ert}\,[C_{trunc}](\mathbf{0}) = \\
&\qquad \textit{Desarrollo del } \texttt{if} \textit{ más externo} \\
&\qquad = \mathbf{1} + \frac{1}{2} \cdot \mathsf{ert}\,[\mathtt{succ} \coloneqq 1](\mathbf{0}) \\
&\qquad + \frac{1}{2} \cdot \mathsf{ert}\,[\mathtt{if}\ (...)\{\mathtt{succ} \coloneqq 1\}\ \mathtt{else}\ \{\mathtt{succ} \coloneqq 0\}](\mathbf{0}) \\
&\qquad \textit{Desarrollo del } \texttt{if} \textit{ más interno} \\
&\qquad = \mathbf{1} + \frac{1}{2} \cdot \mathsf{ert}\,[\mathtt{succ} \coloneqq 1](\mathbf{0}) \\
&\qquad + \frac{1}{2} \cdot \left(\mathbf{1} + \frac{1}{2} \cdot \mathsf{ert}\,[\mathtt{succ} \coloneqq 1](\mathbf{0}) + \frac{1}{2} \cdot \mathsf{ert}\,[\mathtt{succ} \coloneqq 0](\mathbf{0}) \right) \\
&\qquad \textit{Desarrollo de las asignaciones} \\
&\qquad = \mathbf{1} + \frac{1}{2} \cdot \mathbf{1} + \frac{1}{2} \cdot \left(\mathbf{1} + \frac{1}{2} \cdot \mathbf{1} \right) \\
&\qquad = \mathbf{\frac{5}{2}}
\end{aligned}
$$

El resultado $\frac{\mathbf{5}}{\mathbf{2}}$ se puede interpretar de dos formas. Desde un punto de vista informal se puede concluir que C_{trunc} tiene un tiempo de ejecución promedio de 2.5 unidades de tiempo, en cambio, desde la formalidad del modelo adoptado, el resultado $\frac{\mathbf{5}}{\mathbf{2}}$ representa a una función constante que ante todo estado inicial σ, retorna el tiempo de ejecución $\frac{5}{2}$. Sin importar el modo en que se interprete el resultado, vale la pena notar la manera en que la técnica sigue de forma natural la estructura del código, propagándose desde las estructuras más complejas hasta las más simples.

1.3. Inconvenientes de la técnica

Como se mencionó anteriormente la técnica $\mathsf{ert}[\cdot]$ es sumamente expresiva. Al estar definida de manera inductiva sobre la estructura del programa, permite razonar de forma natural

desde un caso base (lo que corresponde a instrucciones atómicas, por ejemplo, una asignación) a un caso inductivo (que corresponde a instrucciones compuestas, por ejemplo, un condicional o la composición secuencial de múltiples instrucciones atómicas). A esto se le agrega que es una técnica robusta, ya que posee un modelo descrito en el trabajo original, el cual avala su correctitud. Pese a lo anterior, la técnica $\mathsf{ert}[\cdot]$ posee algunos inconvenientes que se describirán a continuación.

1.3.1. Cálculos extensos

El primer inconveniente es que, debido a la definición inductiva de la transformada, el desarrollo de los cálculos de esta se vuelven engorrosos y extensos. Basta con observar nuevamente el ejemplo de cálculo de $\mathsf{ert}\,[C_{trunc}\,](\mathbf{0})$, en este se necesitan varios cálculos previos para llegar a los casos base (en este caso las asignaciones). En el caso de un código más extenso se volvería difícil calcular la transformada de forma manual sin cometer errores en el camino. Dado esto, es fundamental poder automatizar este proceso para que esta técnica sea usada en la práctica.

1.3.2. Prueba de invariantes

Por otro lado los ciclos representan en sí un desafío especial. La definición de la transformada de un `while` implica el cálculo del menor punto fijo de un transformador de tiempos de ejecución. El obtener este punto fijo, por lo general no es directo, sino que requiere un trabajo mayor. Para abordar este problema, en el modelo adoptado se utiliza un razonamiento basado en *invariantes* de ciclos. Estos invariantes son tiempos de ejecución en el conjunto $\mathbb{T}$ que permiten establecer cotas superiores al tiempo de ejecución del ciclo.

La aplicación de *invariantes* a su vez requiere probar desigualdades de la forma $f \leq f'$, donde $f, f' \in \mathbb{T}$ son tiempos de ejecución. Estas desigualdades están definidas punto a punto, es decir, si $\forall \sigma \in \Sigma$, se cumple que $f(\sigma) \leq f'(\sigma)$ entonces se tiene que $f \leq f'$. La verificación de estas desigualdades es también un proceso complicado para hacer de forma manual, ya que en principio, los tiempos de ejecución podrían tener una forma arbitraria. Dado lo expuesto, es claro que se debe automatizar la verificación de las desigualdades para comprobar que los *invariantes* sean correctos o no.

1.3.3. Síntesis de invariantes

Por último agregar que estos *invariantes* deben ser propuestos por el usuario de la técnica (son un *input* del análisis) y aunque el trabajo original describe brevemente una técnica basada en plantillas para la síntesis de invariantes, esta sigue requiriendo de significativo esfuerzo y además, no se analiza cómo usar este enfoque para todo ciclo `while` posible. Dada esta situación, es claro que se debe profundizar en cómo sintetizar de manera automática estos invariantes.

1.4. Objetivos

1.4.1. Objetivo general

El objetivo general de esta memoria es desarrollar una herramienta que calcule de manera automática el tiempo de ejecución de programas probabilísticos, usando la técnica de la transformada $\mathsf{ert}[\cdot]$ presentada por Kaminski *et al.*[8].

En particular, la herramienta va a tomar un programa donde cada ciclo está anotado con su respectivo *invariante*, y va a devolver una cota superior del tiempo de ejecución del programa. En caso de que el programa no contenga ciclos se espera calcular con exactitud el tiempo de ejecución del mismo.

1.4.2. Objetivos específicos

Los objetivos específicos son:

Aplicar la transformada $\mathsf{ert}[\cdot]$ Dado un programa con las anotaciones de *invariantes* respectivas, se busca aplicar la transformada $\mathsf{ert}[\cdot]$ sobre el programa. En el caso de que no contenga ciclos, se retorna el tiempo de ejecución esperado exacto del programa. En el caso que sí contenga ciclos, se retorna una cota superior de su tiempo de ejecución.

Generar el conjunto de obligaciones de prueba. Dado un programa con las anotaciones de *invariantes* respectivas, se busca generar el conjunto de obligaciones de prueba (las desigualdades antes mencionadas) que deben satisfacer los *invariantes* para que sean válidos.

Verificar el conjunto de obligaciones de prueba. En este punto se busca verificar de forma automática que las obligaciones de prueba antes mencionadas. En el caso de que alguna no sea válida, se debe reportar un contraejemplo en el cual la obligación no es cierta.

Desarrollar parser. El objetivo es desarrollar un *parser* para escribir de forma cómoda los programas a analizar. El *parser* tendrá la labor de recibir un *string* y retornar un elemento perteneciente al conjunto de programas.

Validar la herramienta desarrollada. El último objetivo es la validación de la herramienta desarrollada. Para esto se plantea comparar los resultados de la herramienta con los cálculos teóricos de un conjunto de programas. La herramienta debe coincidir tanto en el tiempo de ejecución calculado como en la validación de las obligaciones de prueba.

1.5. Solución propuesta

La herramienta fue desarrollada en el lenguaje Haskell. Este lenguaje al ser funcional logra representar de manera natural los conjuntos y funciones inductivas necesarios en el trabajo. Algunos ejemplos de conjuntos inductivos definidos son las expresiones aritméticas lineales `AExp`, los tiempos de ejecución `RunTime` y los programas probabilísticos `Program`, este último conjunto es una versión refinada del conjunto pProg antes mencionado. En cuanto a los funciones recursivas, la principal es la función vcg[·], la que calcula el tiempo de ejecución de un programa y además, obtiene el conjunto de obligaciones de prueba. Pese a lo anterior, para la verificación de las obligaciones de prueba se necesitó una herramienta adicional. En el presente trabajo se usó un *smt-solver* para comprobar la validez de las obligaciones de prueba.

Los *smt-solver* son herramientas que, dada una teoría subyacente, logran verificar si un conjunto de fórmulas lógicas son satisfacibles y en el caso que lo sean, entregan un modelo válido. En el caso particular de este trabajo un conjunto de restricciones se deriva de cada obligación de prueba o_i. La existencia del modelo certifica que la obligación de prueba o_i no es válida y en el que caso que no exista dicho modelo, se certifica su validez. El *smt-solver* específico usado es Z3[2] y para acceder a él a través de Haskell se usó la librería SBV[4].

Un ejemplo de proceso del *input*/*output* de la herramienta desarrollada se puede observar en la Figura 1.2. El *input* necesario es un *string* con el programa escrito en sintaxis concreta, en este caso se usa el programa C_{trunc} antes introducido. El *output* está conformado principalmente por el tiempo de ejecución calculado de manera automática y la información referente a las obligaciones de prueba del programa.

```
Programa Analizado:
pif(<1/2>){succ:= 1} pelse {pif(<1/2>) {succ:= 1} pelse{succ:= 0}}

Tiempo de ejecución calculado:
5/2

El tiempo de ejecución calculado es válido porque no hay obligaciones de prueba
asociadas, ya que el programa no contiene ciclos.

Análisis Finalizado.
```

Figura 1.2: Cálculo automático del tiempo de ejecución de C_{trunc}

Capítulo 2

Marco Teórico

En la presente sección se entregan los conocimientos previos para entender el desarrollo posterior del escrito. En primer lugar, se introducen los *smt-solvers*, herramientas necesarias para comprobar la correctitud de los invariantes. En segundo lugar, se presenta el concepto de VCgen, concepto fundamental para entender el mecanismo de las obligaciones de prueba. Por último, presenta algunos trabajos relacionados con el hecho en esta memoria.

2.1. *Smt-solvers*

El problema de la satisfacibilidad booleana (SAT por sus siglas en inglés) es uno de los planteamientos más importantes en la computación moderna. Este consiste en decidir si una fórmula de la lógica proposicional admite una valuación sobre sus variables, de tal manera que la fórmula sea verdadera. Por ejemplo, la fórmula

$$(x \vee \neg t) \wedge (z \vee t \vee x)$$

es verdadera cuando x lo es.

En el caso que la fórmula lógica esté en su forma normal (como la introducida), es decir, una conjunción de disyunciones, a cada elemento de la conjunción se le denomina *cláusula*. La expresión anterior, por ejemplo, tiene dos cláusulas $(x \vee \neg y)$ y $(z \vee t \vee x)$. Cada cláusula es una disyunción de *literales*. Para la cláusula $(x \vee \neg y)$, los literales son x y $\neg t$. Por último, cada literal es un *átomo* o su negación. Para la cláusula anterior los átomos son x y t.

El planteamiento original de SAT puede ser enriquecido mediante el estudio más minucioso de los átomos. En vez de trabajar con variables booleanas, se puede trabajar con expresiones más complejas. Por ejemplo, en el caso de expresiones aritméticas sobre los números enteros, la expresión $x > 2 * y$ es un átomo y

$$(x > 2 * y \vee y == 0) \wedge z < -1$$

es una fórmula.

Las herramientas que pueden procesar fórmulas como la anterior son denominadas *satisfiability modulo theories solvers* o simplemente *smt-solvers*. Estas herramientas mediante el apoyo de una o más *teorías* logran verificar si la fórmula es satisfacible y en el caso de que lo sea, logran entregar un modelo. Para el caso de la fórmula anterior, la teoría usada es la teoría de las expresiones aritméticas sobre los números enteros y un modelo válido es:

$$x = 3 \;\; y = 1 \;\; z = -2$$

En cuanto a las aplicaciones que tienen los *smt-solvers*, De Moura y Bjørner [3] junto con hacer una introducción a los *smt-solvers*, describen algunas aplicaciones relacionadas a la ingeniería de software. En general, las teorías usadas por los *smt-solvers* poseen suficiente expresividad para modelar los diferentes estados de un programa. Una de las aplicaciones mencionadas en el trabajo es la ejecución simbólica y dinámica (DSE por sus siglas en inglés).

DFS es una técnica de *testing* de carácter híbrido, es decir, combina el análisis estático (sin ejecutar el programa) con el análisis dinámico (ejecutando el programa). El enfoque estático se evidencia en el cómo la técnica trabaja con las variables del programa de forma simbólica, generando *inputs* para que luego sean evaluados dinámicamente. En esencia, lo que busca esta técnica es asociar cada posible ejecución de un programa a una fórmula lógica. Si esa fórmula lógica es satisfacible, entonces existe un *input* que genere esa ejecución. El caso contrario, implica que hay código que nunca se ejecutará y por lo tanto puede ser retirado.

Otro ejemplo de uso de *smt-solvers* se encuentra en la verificación formal de programas. En específico, en el uso de las funciones VCGen, las que serán presentadas al final del siguiente apartado.

2.2. VCGen

Antes de presentar los generadores de condiciones de verificación (VCgen), es necesario introducir algunas ideas previas. La primera de ellas es la Lógica de Hoare [11].

La Lógica de Hoare es un formalismo que permite razonar sobre las propiedades de un programa en base a la definición de tripletas

$$\{\texttt{P}\}\ \texttt{C}\ \{\texttt{Q}\}$$

donde `C` es un programa, `P` y `Q` son condiciones lógicas, a las que se les denomina precondición y postcondición respectivamente. Intuitivamente, la interpretación de una tripleta es:

Si es cierto `P` *antes de la ejecución de* `C`, *entonces luego de la ejecución de* `C`, `Q` *es cierto o* `C` *diverge* .

A modo de ejemplo, la tripleta $\{\texttt{x} = 4\}\ \texttt{x} := 3\ \{\texttt{x} > 0\}$ es cierta, ya que, una vez que se ejecute $\texttt{x} := 3$, la variable `x` es positiva, considerando que al comienzo era igual a 4. El hecho de que se permita la divergencia de los programas significa que se está trabajando dentro de

Asignación	$\{\mathtt{P}[\mathtt{x}/\mathtt{a}]\}\ \mathtt{x} \coloneqq \mathtt{a}\ \{\mathtt{P}\}$
Skip	$\{\mathtt{P}\}\ \mathtt{skip}\ \{\mathtt{P}\}$
Composición Secuencial	$\dfrac{\{\mathtt{P}\}\ \mathtt{C}_1\ \{\mathtt{Q}\} \quad \{\mathtt{Q}\}\ \mathtt{C}_2\ \{\mathtt{R}\}}{\{\mathtt{P}\}\ \mathtt{C}_1; \mathtt{C}_2\ \{\mathtt{R}\}}$
Condicional	$\dfrac{\{\mathtt{P} \wedge \mathtt{b}\}\ \mathtt{C}_1\ \{\mathtt{Q}\} \quad \{\mathtt{P} \wedge \neg\mathtt{b}\}\ \mathtt{C}_2\ \{\mathtt{Q}\}}{\{\mathtt{P}\}\ \mathtt{if}\ (\mathtt{b})\ \mathtt{then}\ (\mathtt{C}_1)\ \mathtt{else}\ (\mathtt{C}_2)\ \{\mathtt{Q}\}}$
Ciclo	$\dfrac{\{\mathtt{I} \wedge \mathtt{b}\}\ \mathtt{C}\ \{\mathtt{I}\}}{\{\mathtt{I}\}\ \mathtt{while}\ (\mathtt{b})\ \mathtt{C}\ \{\mathtt{I} \wedge \neg\mathtt{b}\}}$
Consecuencia	$\dfrac{\{\mathtt{P}'\}\ \mathtt{S}\ \{\mathtt{Q}'\}}{\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}}\ \mathtt{Si}\ \mathtt{P} \Longrightarrow \mathtt{P}' \wedge \mathtt{Q}' \Longrightarrow \mathtt{Q}$

Figura 2.1: Reglas de derivación de la Lógica de Hoare

lo que se denomina como *corrección parcial*. La contraparte a este enfoque es la *corrección total*, la cual exige la terminación de un programa para validar una tripleta. Este último enfoque no es abordado y sólo se menciona por completitud.

Para comprobar la validez de una tripleta de Hoare, se requiere del uso de las reglas de inferencias propias del método. Las reglas pueden se encontradas en la Figura 2.1 (la notación $\mathtt{P}[\mathtt{x}/\mathtt{a}]$ representa el reemplazo de todas las incidencias de la variable $\mathtt{x}$ por la expresión $\mathtt{a}$ en la condición $\mathtt{P}$). Las tripletas de la asignación y el programa `skip` son tripletas que no exigen ninguna hipótesis para ser válidas. En cambio, las reglas de la forma $\frac{\mathtt{A}}{\mathtt{B}}$ se interpretan de la siguiente manera:

Si la tripleta $\mathtt{A}$ *se cumple, entonces se cumple la tripleta* $\mathtt{B}$.

De este conjunto de reglas, la más relevante para este trabajo es la regla del ciclo `while`. Esta regla hace uso de la condición $\mathtt{I}$ denominada como *invariante*. Un invariante, en este contexto, es una condición que es preservada por el cuerpo del ciclo. Esto se representa en la regla misma. En la tripleta $\{\mathtt{I} \wedge \mathtt{b}\}\ \mathtt{C}\ \{\mathtt{I}\}$, se exige en la postcondición (después de ejecutar el cuerpo) se cumpla I, y en la precondición (antes de ejecutar el cuerpo) se exige que se cumpla I y la condición necesaria para ejecutar el cuerpo del ciclo ($\mathtt{b}$).

Si se cumple $\{\mathtt{I} \wedge \mathtt{b}\}\ \mathtt{C}\ \{\mathtt{I}\}$, entonces se cumple $\{\mathtt{I}\}\ \mathtt{while}\ (\mathtt{b})\ \mathtt{C}\ \{\mathtt{I} \wedge \neg\mathtt{b}\}$. Esta expresión exige que en la precondición (antes del ciclo) se cumpla I, y en la postcondición (después del ciclo) se cumpla I (ya que es invariante) y la condición $\neg\mathtt{b}$. Un punto importante a considerar es que estos invariantes deben ser propuestos, por lo tanto, puede que no sean correctos.

La demostración de la validez a través de estas reglas, es un proceso extenso si es que se hace de forma manual, por lo mismo, es necesario recurrir a la automatización mediante el uso de una función generadora de condiciones de verificación. A esta función se le denomina VCGen y su labor es generar una serie de condiciones (denominadas obligaciones de prueba)

Tabla 2.1: Definición de la transformada $\mathsf{wlp}[\cdot]$

C	$\mathsf{wlp'}[\mathtt{C}](\mathtt{Q})$
$\mathtt{x} := \mathtt{a}$	$\mathtt{Q}\lceil \mathtt{x}/\mathtt{a} \rceil$
skip	Q
$\mathtt{C}_1 \ ; \ \mathtt{C}_2$	$\mathsf{wlp'}[\mathtt{C}_1](\mathsf{wlp'}[\mathtt{C}_2](\mathtt{Q}))$
if (b) then $\mathtt{C}_1$ else $\mathtt{C}_2$	$\big(\mathtt{b} \implies \mathsf{wlp'}[\mathtt{C}_1](\mathtt{Q})\big) \wedge \big(\neg \mathtt{b} \implies \mathsf{wlp'}[\mathtt{C}_2](\mathtt{Q})\big)$
while (b) C {I}	I

que permitan validar a una tripleta $\{\mathtt{P}\}\ \mathtt{C}\ \{\mathtt{Q}\}$, junto a la correctitud de los invariantes que contenga el programa C. De esta forma, si el conjunto de obligaciones de prueba es válido, entonces la tripleta original es válida y sus invariantes son correctos. En el caso contrario, significa que hay un error, ya sea en el invariante propuesto o en la tripleta original. En este punto se requiere del análisis humano para entender cuál es el error y cómo proceder. En este punto se retoman los *smt-solvers* introducidos en el apartado anterior, ya que pueden ser utilizados para validar estas obligaciones de prueba de forma automática.

Antes de presentar la definición formal de VCGen es necesario introducir una función transformadora. La función necesaria es la transformada $\mathsf{wlp}[\cdot]$ (*weakest liberal precondition*). Esta función aborda el problema de encontrar la precondición P más *débil* que haga válida la tripleta $\{\mathtt{P}\}\ \mathtt{C}\ \{\mathtt{Q}\}$. A esta precondición se le denota como $\mathsf{wlp}[\mathtt{C}](\mathtt{Q})$. Con respecto al concepto de débil, este se refiere a la condición más general posible, por ejemplo, si $\mathtt{P} \implies \mathtt{P}'$ entonces $\mathtt{P}'$ es más débil que P. En general, si la tripleta $\{\mathtt{P}\}\mathtt{C}\{\mathtt{Q}\}$ es válida, entonces se tiene que $\mathtt{P} \implies \mathsf{wlp}[\mathtt{C}](\mathtt{Q})$. Como última aclaración, la condición más débil posible es la constante true y la más fuerte la constante false.

A modo de ejemplo, $\mathsf{wlp}[\mathtt{a} := \mathtt{a} - 1; \mathtt{b} := \mathtt{b} + 1](\mathtt{a} \cdot \mathtt{b} = 0)$ es igual a la condición $\mathtt{a} = 1 \vee \mathtt{b} = -1$. Es importante notar que la precondición $\mathtt{a} = 1$ también es válida para la lógica de Hoare, pero $\mathsf{wlp}[\cdot]$ busca la condición más general posible, esa es la razón del resultado. Otro ejemplo es $\mathsf{wlp}[\mathtt{while}(\mathtt{true})\ \mathtt{C}\ ](\mathtt{Q})$, el resultado es el valor más débil posible true, ya que el programa diverge sin importar el estado o la postcondición.

Una versión modificada de la transformada es representada en la Tabla 2.1 (denominada wlp'), la diferencia radica en la transformada del ciclo while. En la definición original de $\mathsf{wlp}[\cdot]$, el valor señalado es igual al mayor punto fijo de una transformada. En esta versión el valor retornado es un invariante.

En este punto ya están las condiciones para introducir la función VCGen. Esta función se divide en dos partes, la primera es la función $\mathsf{VC}[\cdot]$ presentada en la Tabla 2.2. Esta se encarga de calcular las obligaciones asociadas a los invariantes, por lo mismo, sólo se generan obligaciones si el programa es un ciclo while. Intuitivamente, la obligación $(\mathtt{I} \wedge \mathtt{b}) \implies \mathsf{wlp'}[\mathtt{C}](\mathtt{I})$ busca comprobar que I es conservado por el cuerpo del ciclo (es decir, es un invariante), y la obligación $(\mathtt{I} \wedge \neg \mathtt{b}) \implies \mathtt{Q}$, busca comprobar que I sea lo suficientemente fuerte como para implicar la postcondición Q a la salida del ciclo.

La segunda parte es la encargada de comprobar que la tripleta $\{\mathtt{P}\}\mathtt{C}\{\mathtt{Q}\}$ sea válida. Es

Tabla 2.2: Definición de la transformada $\mathsf{VC}[\cdot]$

$\mathtt{C}$	$\mathsf{VC}[\mathtt{C}](\mathtt{Q})$
$\mathtt{x} \coloneqq \mathtt{a}$	$\emptyset$
$\mathtt{skip}$	$\emptyset$
$\mathtt{C}_1 \ ; \ \mathtt{C}_2$	$\mathsf{VC}[\mathtt{C}_1](\mathsf{wlp'}[\mathtt{C}_2](\mathtt{Q})) \cup \mathsf{VC}[\mathtt{C}_2](\mathtt{Q})$
$\mathtt{if}\ (\mathtt{b})\ \mathtt{then}\ \mathtt{C}_1\ \mathtt{else}\ \mathtt{C}_2$	$\mathsf{VC}[\mathtt{C}_1](\mathtt{Q}) \cup \mathsf{VC}[\mathtt{C}_2](\mathtt{Q})$
$\mathtt{while}\ (\mathtt{b})\ \mathtt{C}\ \{\mathtt{I}\}$	$\{(\mathtt{I} \wedge \mathtt{b}) \Longrightarrow \mathsf{wlp'}[\mathtt{C}](\mathtt{I}), (\mathtt{I} \wedge \neg \mathtt{b}) \Longrightarrow \mathtt{Q}\} \cup \mathsf{VC}[\mathtt{I}](\mathtt{Q})$

posible que las obligaciones asociadas a los invariantes sean válidas, pero aún así la tripleta original no lo sea. Para comprobar este punto es necesario agregar la obligación $\mathtt{P} \Longrightarrow \mathsf{wlp'}[\mathtt{C}](\mathtt{Q})$. De esta forma, la función VCGen queda definida de la siguiente manera:

$$\mathsf{VCGen}(\{\mathtt{P}\}\mathtt{C}\{\mathtt{Q}\}) = \{\mathtt{P} \Longrightarrow \mathsf{wlp'}[\mathtt{C}](\mathtt{Q})\} \cup \mathsf{VC}[\mathtt{C}](\mathtt{Q})$$

2.3. Trabajo relacionado

En cuanto al razonamiento sobre programas probabilísticos, un enfoque basado en martingalas es presentado por Wang *et al.*[12]. Una martingala es una secuencia de variables aleatorias, tales que el valor esperado de variable número n es igual al valor de la variable n - 1. El trabajo inicia describiendo los conceptos teóricos previos necesarios para entender los aportes hechos y algunos casos de estudio. Luego detalla como a través de las martingalas pueden obtener cotas del tiempo de ejecución un programa. Finaliza demostrando como sintetizar estas cotas de forma automática. Se prefiere la transformada $\mathsf{ert}[\cdot]$ debido a que requiere una menor cantidad de *inputs*. La técnica basada en martingalas requiere un invariante por cada instrucción de código, en cambio la transformada $\mathsf{ert}[\cdot]$ requiere un invariante por cada ciclo. Además de lo anterior, se tiene que el enfoque de Wang *et al.* sólo genera cotas polinomiales, y por lo tanto no puede razonar sobre programas cuyos tiempos de ejecución no esté acotado polinomialmente, además de que no podría encontrar la cota exacta de programas cuyo tiempo de ejecución no sea polinomial. Debido a estos inconvenientes se prefiere la transformada $\mathsf{ert}[\cdot]$.

Otro planteamiento es presentado por Giels *et al.* [5]. Los autores presentan como razonar sobre un tipo particular de programa probabilístico. El trabajo inicia definiendo el tipo de programa, denominados *CP programs*, el cual es una clase particular de ciclo `while`. En las secciones posteriores trabajan con los conceptos de terminación casi segura y tiempo de ejecución esperado. Para finalizar, presentan un algoritmo para calcular, de manera exacta, el tiempo de ejecución esperado del tipo particular de programa presentado. A pesar de que los autores logran estudiar de forma exhaustiva los *CP programs*, el hecho que el trabajo se acote a este tipo particular de programa hace que se prefiera la transformada $\mathsf{ert}[\cdot]$.

Un último enfoque, aunque acotado a los programas deterministas, es presentado por Nielson [10]. El autor analiza el tiempo de ejecución de un programa extendiendo las reglas de la lógica de Hoare antes presentadas. Inicia definiendo el lenguaje que será objeto de estudio y su semántica. Luego extiende las reglas de la lógica de Hoare para razonar sobre condiciones lógicas que hagan referencia al tiempo de ejecución de un programa. Por ejemplo, se puede estudiar la condición $\texttt{P} : \texttt{time} \leq \texttt{n}^2$, la cual será cierta si el tiempo de ejecución del programa `P` es menor a $\texttt{n}^2$, siendo $\texttt{n}^2$ una variable del programa. El trabajo finaliza con un ejemplo de uso. La técnica presentada es sumamente expresiva, pero requiere como *input* alguna condición sobre el tiempo de ejecución de un programa, no lo calcula a partir del mismo. Debido a esta razón, se prefiere la transformada ert[·].

En cuanto a la implementación, en el marco de una charla sobre el análisis de programas con Z3, Jelvis[7] desarrolló un proyecto que aborda el problema de comprobar si dos programas son semánticamente equivalentes, es decir, si ante un mismo *input*, los programas retornan un *output* diferente. Para lograr este objetivo, Jelvis implementó el conjunto de programas de manera inductiva, para luego, a partir de estos elementos, generar un fórmula lógica, la cual evaluaba con la API de Z3 para Haskell. Vale la pena mencionar que este trabajo es de suma importancia para para el desarrollo hecho en esta memoria, ya que la implementación hecha por Jelvis fue usada como punto de partida para la propia.

2.4. Síntesis del Capítulo

En este capítulo se presentaron los conceptos necesarios para entender el trabajo desarrollado en esta investigación. El primer concepto presentado son los *smt-solvers*. Un *smt-solver* es una herramienta que permite analizar la satisfacibilidad de una fórmula lógica. Estas herramientas pueden potenciadas a través de teorías, con las cuales pueden razonar sobre expresiones complejas como las expresiones lineales.

El siguiente concepto presentado son las funciones generadoras VCGen e invariantes. Los invariantes son condiciones que son preservadas luego de la ejecución del cuerpo de un ciclo `while`, y su utilidad es que permiten razonar sobre estos. Las funciones VCGen son funciones generadoras de obligaciones de prueba. Una obligación de prueba, en el contexto presentado, es una condición lógica que permite validar la correctitud de de un invariante y de una tripleta de Hoare. El concepto de obligación de prueba puede ser utilizado en para analizar otras propiedades de un programa, en particular, en esta investigación se estudia el tiempo de ejecución.

Por último, se presentan algunos trabajos relacionados a al estudio del tiempo de ejecución de un programa, además de un último trabajo relacionado a la implementación.

Capítulo 3

Problema

En el presente capítulo se describe y discute en detalle el problema abordado en esta memoria. Así, en la primera parte se estudian los inconvenientes prácticos que posee la técnica ert[·]; además, se discute la relevancia de resolver estos inconvenientes antes mencionados. En la segunda parte se abordan las características que debiese tener una solución y, además, se detalla la metodología elegida para desarrollarla.

3.1. Teoría e inconvenientes de la transformada Ert

Para describir de manera apropiada los inconvenientes de la transformada es necesario hacerlo mientras se detalla su teoría. La primera parte de este apartado analiza la naturaleza inductiva del lenguaje probabilístico pProg y de la transformada ert[·]. En la segunda parte del apartado se discuten los tópicos relacionados a los invariantes de ciclos `while`. Este concepto se origina en la definición de la técnica estudiada, pero su relevancia merece ser analizada de forma separada.

3.1.1. Lenguaje y transformada

El primer concepto descrito en el trabajo original es el lenguaje pProg. Este es un lenguaje definido de forma inductiva y posee los principales constructores usados en la mayoría de lenguajes imperativos. A continuación, se introduce la definición de la sintaxis de este lenguaje.

Definición 3.1 *Sintaxis del lenguaje* pProg

pProg →	`empty`	*Programa vacío sin gasto de tiempo*
	`skip`	*Programa vacío con gasto de tiempo*
	$\mathtt{x} :\approx \mu$	*Asignación probabilística*
	pProg ; pProg	*Composición secuencial*
	`if` (ξ){pProg} `else` {pProg}	*Condicional*
	`while` (ξ) {pProg}	*Ciclo*

En la asignación probabilística $\mathtt{x} :\approx \mu$, x representa una variable y μ una distribución probabilística sobre esa variable. Por ejemplo, se podría tener la expresión $\mathtt{x} :\approx \mathtt{uniform}[0, 1]$ donde se le asigna a x un valor equiprobable entre 0 y 1 (distribución de probabilidad uniforme). En los condicionales y ciclos, la expresión ξ se refiere a una distribución de probabilidad sobre los booleanos. Estas distribuciones pueden tener dos formas, ya sea una distribución Bernoulli (como aparece en el programa C_{trunc}) o condiciones deterministas (como x + y == 0 o 2x < 0), las cuales son interpretadas como distribuciones de Dirac que le otorgan todo el peso a un solo punto. Misma situación tenemos para las asignaciones probabilistas, donde por ejemplo $\mathtt{x} := 8$ le asigna 8 a la variable x con probabilidad 1.

En general el lenguaje pProg, pese a que es minimalista, tiene la expresividad suficiente como para generar programas interesantes, los que serán presentados y estudiados a lo largo de esta investigación. Pese a lo anterior, es relevante mencionar que en la versión extendida del trabajo original[9], los autores añaden una segunda sintaxis para permitir la generación de funciones recursivas. Esta sintaxis da la posibilidad de estudiar algoritmos como una versión probabilista de la búsqueda binaria, la que es analizada en los casos de estudio mediante su representación en un programa probabilístico recursivo. Este tipo de programas no son analizados en este trabajo, pero se mencionan para dar una muestra de los alcances que podría tener la técnica adoptada.

Un punto interesante para discutir es la definición de las guardas ξ. El comportamiento de una distribución Bernoulli es claramente dispar con respecto al comportamiento clásico de una condición determinista. Para evidenciar este punto, primero se debe introducir la definición de la transformada ert[·].

Tabla 3.1: Definición de la transformada ert[·]

C	ert[C](f)
`empty`	f
`skip`	$1 + f$
$\mathtt{x} :\approx \mu$	$1 + \lambda\sigma.\mathbf{E}_{\mu}(\lambda \mathtt{v}.\ f[\mathtt{x}/\mathtt{v}](\sigma))$
$\mathtt{C}_1$; $\mathtt{C}_2$	ert[$\mathtt{C}_1$](ert[$\mathtt{C}_2$](f))
`if` $(\xi)\{\mathtt{C}_1\}$ `else` $\{\mathtt{C}_2\}$	$1 + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}_1](f) + [\![\neg\xi]\!] \cdot \mathsf{ert}[\mathtt{C}_2](f)$
`while` (ξ) $\{\mathtt{C}'\}$	$\mathtt{lfp} X.\ 1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}'](X)$

Recordemos que ert[C](f) entrega el tiempo de ejecución del programa C más el programa que le sigue, asumiendo que f representa precisamente el tiempo de ejecución del programa que le sigue a C.

El valor de `empty` es f debido a que no tiene costo asociado ejecutar este programa. Lo contrario ocurre con el programa `skip` que tiene el costo de una unidad para ejecutarlo más el tiempo f. En el caso de la asignación la expresión $\mathbf{E}_\mu(h) \triangleq \Sigma_v\ \mathbf{Pr}_\mu(v) \cdot h(v)$ representa la esperanza de la variable aleatoria h sobre la distribución μ, además sea $\sigma \in \Sigma$, $f[\mathrm{x/v}](\sigma) \triangleq f(\sigma[\mathrm{x/v}])$, donde $\sigma[\mathrm{x/v}]$ es el estado obtenido al actualizar en σ el valor de x por v. Se suma una unidad de costo debido a que en este modelo la asignación tiene ese coste.

El primer caso inductivo es la ejecución secuencial. El valor es calculado a través de la composición de las transformadas (notar que la transformada más interna es la transformada del bloque $\mathtt{C}_2$).

En las transformadas de las condiciones se suma 1 por la evaluación de la guarda. La notación $[\![\xi]\!]$ indica la probabilidad de que la distribución ξ tome el valor `true`. En el caso del `if` se suman los tiempos de ejecución de las dos ramas, ponderados por la probabilidad de tomar cada una de ellas. Por último, en la definición de $\mathsf{ert}[\mathtt{while}](\cdot)$, $\mathbf{lfp}X.F(X)$ representa el menor punto fijo del transformador

$$F\colon X \to 1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}'](X)$$

A este transformador se le denomina función característica del ciclo y es uno de los objetos centrales en el análisis de los ciclos `while`, por esta razón se introduce su definición formal.

Definición 3.2 (Función característica) *Sea un ciclo* `while` (ξ) $\{\mathtt{C}\}$ *y f un tiempo de ejecución $\in \mathbb{T}$, se define*

$$F_f^{\langle \xi, \mathtt{C} \rangle} : \mathbb{T} \to \mathbb{T},\ X \to 1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}](X)$$

como la función característica del ciclo `while` (ξ) $\{\mathtt{C}\}$ *con respecto a f.*

En caso de que no haya ambigüedades con los valores de f, ξ y $\mathtt{C}$, se omitirán los super/subíndices con el fin de hacer la notación más amigable al lector. Luego de conocer la definición de la transformada $\mathsf{ert}[\cdot]$ están las condiciones necesarias para detallar sus principales inconvenientes. Tal como se describió en la Introducción (Sección 1.3) los cálculos inducidos por esta técnica tienden a ser largos y tediosos. Para entender porqué sucede esta situación basta con enfocarse en la transformada de la composición secuencial. La mayoría de programas usados en la práctica son extensas composiciones secuenciales de programas de menor tamaño (un ciclo `while`, seguido de una asignación etc.). Además, se tiene que la transformada de la composición implica calcular la transformada de dos programas menores. Estas dos situaciones provocan que para calcular la transformada de un programa se necesite una cantidad de operaciones al menos proporcional al tamaño del código. Considerando la escala del tamaño de los programas usados en la práctica, es clara la necesidad de automatizar el proceso de cálculo.

El siguiente inconveniente se origina en las expresiones ξ. Debido a que estas expresiones tienen dos comportamientos posibles, las expresiones $[\![\xi]\!]$ también tienen dos comportamientos posibles. Si ξ es una guarda probabilista (lanzamiento de una moneda), la interpretación de $[\![\xi]\!]$ es directa y es el valor `p` de la distribución Bernoulli, pero si ξ es determinista, por

ejemplo $\mathtt{x} + \mathtt{y} == 0$, se interpreta como una indicatriz que toma los valores de 0 o 1 (dependiendo si la expresión es verdadera o falsa). Esta situación se extrapola al conjunto pProg, donde el programa $\mathtt{if}\backslash\mathtt{while}$ con guarda determinista se estudia con la misma estructura que su contraparte probabilista. Podemos agregar a las asignaciones, donde se les trata como probabilistas, y las deterministas son un subconjunto de estas. La diferencia marcada entre el comportamiento determinista y el probabilista sugiere tratarlos con estructuras diferentes, de tal manera que el análisis sea más expresivo que con las definiciones actuales.

3.1.2. Invariantes de ciclos

Como se mencionó antes, el valor "exacto" de la transformada de un ciclo `while` es el menor punto fijo de su función característica. El obtener este valor requiere un esfuerzo considerable, ya que, en general, se necesita un análisis detallado sobre el transformador. Uno de los métodos útiles para este propósito es el que propone el teorema de punto fijo de Kleene [13]. Este método inicia en el elemento *bottom* (en este caso $\mathbf{0}$), luego aplica de manera iterativa el transformador respectivo (en este caso la función característica) hasta converger en el punto fijo del mismo. Si bien este enfoque es una opción interesante y merece un estudio profundo en futuros trabajos, en el presente solo se usa para ganar cierta intuición sobre candidatos a invariantes de un ciclo `while`.

Recordemos que un invariante `I`, en este contexto, es un tiempo de ejecución $\in \mathbb{T}$. Intuitivamente, un invariante es una cota superior del tiempo de ejecución calculado de un ciclo `while`, pero no todo elemento de $\mathbb{T}$ es un invariante válido. A continuación, se presenta la condición necesaria para comprobar la correctitud de un invariante.

Definición 3.3 (Invariantes) *Sea* `I` *y* f *dos tiempos de ejecución* $\in \mathbb{T}$. *Se dice que* `I` *es un* ***invariante*** *del ciclo* `while` (ξ) $\{\mathtt{C}\}$ *con respecto a* f *ssi*

$$1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}](\mathtt{I}) \leq \mathtt{I}$$

Teorema 3.4 (Invariante como cota superior) *Sea* `I` *un invariante del ciclo* `while` (ξ) $\{\mathtt{C}\}$ *con respecto a* f, *entonces se tiene que*

$$\mathsf{ert}[\mathtt{while}\ (\xi)\ \{\mathtt{C}\}](f) \leq \mathtt{I}$$

La intuición y justificación del Teorema 3.4 puede encontrarse en el trabajo original y no se profundizará en mayor medida en este escrito.

El Teorema 3.4 entrega la herramienta para acotar superiormente el tiempo de ejecución de un `while`, lo que se logra a través de la condición impuesta en la Definición 3.3. Esta condición tiene un planteamiento equivalente. El término de la izquierda de la desigualdad $1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}](\mathtt{I})$, es igual a la función característica del ciclo `while` (ξ) $\{\mathtt{C}\}$, evaluada en el invariante I. Con esa consideración la condición se puede plantear como:

$$F_f^{\langle \xi, \mathtt{C} \rangle}(\mathtt{I}) \leq \mathtt{I} \tag{3.1}$$

A partir de la condición anterior se puede extraer el siguiente corolario :

Corolario 3.5 (Puntos fijos como invariantes) *Sea* $\mathtt{I}$ *y* f *dos tiempos de ejecución* $\in \mathbb{T}$ *y* $F_f^{\langle \xi, \mathtt{C} \rangle}$ *la función característica del ciclo* $\mathtt{while}\ (\xi)\ \{\mathtt{C}\}$ *con respecto a* f*. Si* $\mathtt{I}$ *es un punto fijo de* $F_f^{\langle \xi, \mathtt{C} \rangle}$ *entonces* $\mathtt{I}$ *es un invariante de* $\mathtt{while}\ (\xi)\ \{\mathtt{C}\}$ *con respecto a* f*.*

Demostración. En efecto, si $\mathtt{I}$ es un punto fijo de $F_f^{\langle \xi, \mathtt{C} \rangle}$, entonces se tiene que $F_f^{\langle \xi, \mathtt{C} \rangle}(\mathtt{I}) = \mathtt{I}$. Al aplicar esto a la condición de la Definición 3.3 se obtiene $\mathtt{I} \leq \mathtt{I}$, concluyendo que $\mathtt{I}$ es un invariante. □

Como se expuso antes en los inconvenientes de la transformada $\mathsf{ert}[\cdot]$, el uso de invariantes no es inmediato. En primer lugar, el encontrar el invariante es un proceso poco claro y en general se requiere algunas técnicas basadas en la intuición y, en segundo lugar, se encuentra el demostrar la desigualdad de la Definición 3.3. En general la demostración se logra desde el razonamiento algebraico, operando cada término hasta llegar a una expresión conveniente. En las secciones posteriores se introducirá el modo en que se resuelve este punto, mediante un método claro y automático.

Para finalizar esta sección, se introduce un ejemplo de invariante de un ciclo `while`. El programa de ejemplo está en la Figura 3.1 y representa a una distribución geométrica real (no truncada como C_{trunc}). Inicia evaluando si la variable $\mathtt{c}$ es igual a 1. De ser así, entra al cuerpo y reasigna el valor de $\mathtt{c}$ de manera equiprobable entre los valores 0 y 1. Dado que el programa es un ciclo, evalúa la guarda hasta que sea falsa e inmediatamente finaliza el programa.

$$\begin{array}{c} \mathtt{while}\ (\mathtt{c} == 1) \\ \{\ \mathtt{c} :\approx \frac{1}{2} \cdot \langle 0 \rangle + \frac{1}{2} \cdot \langle 1 \rangle\ \ \} \end{array}$$

Figura 3.1: Distribución geométrica $C_{\mathtt{geo}}$

Una vez introducido el programa, se debe proponer un invariante. En este caso se propone el invariante

$$\mathtt{I}_{\mathtt{geo}} = 1 + 4 \cdot [\![\mathtt{c} == 1]\!]$$

con respecto al tiempo de ejecución $\mathbf{0}$. La intuición detrás de este invariante se puede obtener analizando el programa $C_{\mathtt{geo}}$. En primer lugar, si la guarda es positiva, el programa consume dos unidades de tiempo, una por la evaluación de la guarda y una por la asignación del cuerpo. Además de lo anterior, el programa, en promedio, debiese iterar dos veces. Con esas consideraciones se obtiene el término $2 \cdot 2 \cdot [\![\mathtt{c} == 1]\!] = 4 \cdot [\![\mathtt{c} == 1]\!]$. Por último, si la guarda es falsa se consume una unidad de tiempo, ese es el origen de la constante 1 sumada.

Sea $F_{geo}(X) = 1\ +\ [\![\neg \mathtt{c} == 1]\!] \cdot \mathbf{0}\ +\ [\![\mathtt{c} == 1]\!] \cdot \mathsf{ert}[\mathtt{c} :\approx \frac{1}{2} \cdot \langle 0 \rangle\ +\ \frac{1}{2} \cdot \langle 1 \rangle](X)$, la función característica del ciclo. Se debe demostrar la desigualdad de la Definición 3.3. En este caso la condición es

$$F_{geo}(\mathtt{I}_{\mathtt{geo}}) \leq \mathtt{I}_{\mathtt{geo}}$$

A continuación, se ejemplifica como verificar la validez del invariante $\mathtt{I}_{\mathtt{geo}}$.

Ejemplo Comprobar validez del invariante $\mathtt{I}_{\mathtt{geo}}$

$$\begin{aligned} F_{geo}(\mathtt{I}_{\mathtt{geo}}) &= 1 \ + \ [\![\neg \mathtt{c} == 1]\!] \cdot \mathbf{0} \ + \ [\![\mathtt{c} == 1]\!] \cdot \mathsf{ert}[\mathtt{c} :\approx \frac{1}{2} \cdot \langle 0 \rangle \ + \ \frac{1}{2} \cdot \langle 1 \rangle](\mathtt{I}_{\mathtt{geo}}) \\ &= 1 + \ [\![\mathtt{c} == 1]\!] \cdot (1 + \frac{1}{2} \cdot \mathtt{I}_{\mathtt{geo}}[\mathtt{c}/0] + \frac{1}{2} \cdot \mathtt{I}_{\mathtt{geo}}[\mathtt{c}/1]) \\ &= 1 + \ [\![\mathtt{c} == 1]\!] \cdot (1 + \frac{1}{2} \cdot (1 + 4 \cdot [\![0 == 1]\!]) + \frac{1}{2} \cdot (1 + 4 \cdot [\![1 == 1]\!])) \\ &= 1 + 4 \cdot [\![\mathtt{c} == 1]\!] = \mathtt{I}_{\mathtt{geo}} \leq \mathtt{I}_{\mathtt{geo}} \end{aligned}$$

En el ejemplo presentado se evidencia que $\mathtt{I}_{\mathtt{geo}}$ es un punto fijo del transformador F_{geo}. Esta situación no es la habitual, en general para una función característica F y un invariante $\mathtt{I}$, la expresión $F(\mathtt{I})$ tendrá una forma arbitraria y no será directa la comparación con $\mathtt{I}$.

3.2. Características de la solución

La solución desarrollada debe lograr cumplir con los objetivos propuestos (Sección 1.4.1) en esta investigación. Estos propósitos, como es lógico, buscan resolver los inconvenientes descritos en la sección anterior. El inconveniente que no es abordado por algún objetivo es la síntesis de invariantes, esto se debe a que este tópico es un tema extenso y no hubiese sido posible considerarlo por limitaciones de tiempo. Pese a lo anterior, para obtener cierta intuición a la hora de proponer un invariante, se desarrolló una herramienta basada en el teorema del punto fijo de Kleene.

Además de cumplir los objetivos descritos en la introducción, la herramienta debe seguir un flujo para que sea agradable de usar. Este flujo aplica tanto al *input* como al *output* de la herramienta. En el caso del *input* se necesita que la herramienta reciba el programa y los invariantes de ciclo de manera rápida y cómoda. En cuanto al *output*, la herramienta debe imprimir en la salida estándar, tanto el tiempo de ejecución estimado como la información referente a las obligaciones de prueba generadas. Con respecto a este último punto, en caso de que las obligaciones de prueba no sean correctas, se debe entregar el contraejemplo respectivo. En el caso en que sean correctas, debe entregar el mensaje respectivo que señala que el proceso ha sido exitoso.

En cuanto a la metodología, esta tiene una relación directa con los objetivos planteados. Recordando que el lenguaje elegido para la implementación es Haskell, la primera parte del desarrollo se enfocó en reproducir las estructuras presentadas en el trabajo original. La estructura base son las expresiones aritméticas, luego sigue tanto las expresiones booleanas como los tiempos de ejecución y por último, se define el lenguaje de los programas probabilísticos.

El siguiente paso es definir las funciones asociadas a las estructuras anteriores. La principal función es una modificación de la transformada $\mathsf{ert}[\cdot]$ que, además de calcular el tiempo de ejecución, calcula las obligaciones de prueba. Además de la función anterior, se definen funciones para simplificar los resultados obtenidos. Considerando la extensión que puede llegar a tener los cálculos de $\mathsf{ert}[\cdot]$, una función de este tipo es más que necesaria. También es necesario mencionar el *parser* asociado a la herramienta desarrollada. Debido a la naturaleza

funcional de Haskell, la adaptación de la teoría a la implementación misma de estas funciones y las estructuras antes introducidas fue un proceso claro y expedito.

El tercer paso es integrar al desarrollo el *smt-solver*. El *smt-solver* se encarga de decidir si las obligaciones de prueba antes calculadas (y también los respectivos invariantes) son correctas o no. Para esto se hace uso de la librería SBV, la que permite utilizar un *smt-solver*, a través de instrucciones escritas en Haskell. Para finalizar, es necesario validar la herramienta. Para esto se compara los resultados teóricos de la transformada ert[·] con los resultados entregados por la herramienta para un conjunto de programas que actúan como casos de testeo. Para que el test sea exitoso, se necesita que los cálculos coincidan tanto en el tiempo de ejecución, como el análisis de las obligaciones de prueba.

3.3. Síntesis del capítulo

En el capítulo se describe el problema abordado en la memoria. Este es conformado por los inconvenientes asociados al uso práctico de la transformada ert[·]. Estos inconvenientes son:

Cálculos extensos y propenso a errores. La definición inductiva de la transformada ert[·] provoca que para obtener el tiempo de ejecución estimado de un programa `C` se necesite una cantidad de pasos al menos proporcional a su tamaño. Lo anterior, combinado a que en la práctica los programas son extensos, tiene como consecuencia que no sea factible aplicar la transformada de forma manual.

Definiciones que no diferencian entre estructuras deterministas y probabilistas. Las estructuras booleanas tienen dos comportamientos diametralmente diferentes, pueden ser deterministas o probabilistas. Como consecuencia de lo anterior, los programas probabilísticos y deterministas se tratan con la misma estructura. El no tener una división entre estos dos comportamientos provoca que no se pueda estudiar, de manera óptima, cada programa de forma particular.

Ausencia de un método para sintetizar un invariante de un ciclo. La transformada de un ciclo `while` implica el cálculo del menor punto fijo de una transformada. Para sobrellevar esta situación, los autores proponen el uso de invariantes. La síntesis de estos invariantes es un proceso poco claro y, en ocasiones, sustentado en la intuición. Para poder llevar a la práctica la transformada ert[·] es necesario tener proceso claro y robusto que permita sintetizar el invariante de un ciclo o decidir cuando esto no sea posible. Este punto debido a límites de tiempo fue abordado de manera superficial en este trabajo y se propone seguir su investigación en el futuro.

Ausencia de un método para demostrar la condición que avala la correctitud de un invariante de un ciclo. Para demostrar la correctitud de un invariante `I`, sobre un

programa `while` (ξ) $\{\mathtt{C}\}$ se debe demostrar que:

$$1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}](\mathtt{I}) \leq \mathtt{I}$$

Esta condición es una desigualdad entre funciones, es decir, una desigualdad punto a punto. Debido a la forma arbitraria que puede tener `I`, no es directa la comparación y, en general, requiere un trabajo minucioso por parte del analista. Como todo proceso que requiere la revisión humana existe la posibilidad de equivocarse o tardar más de lo necesario. Debido a esto último, la automatización de esta demostración es necesaria para usar la transformada $\mathsf{ert}[\cdot]$.

Capítulo 4

Solución

En este capítulo se describen las principales características de la solución desarrollada. En primer lugar, se detallan las diferentes estructuras necesarias para la automatización de los cálculos de los tiempos de ejecución y obligaciones de prueba. En segundo lugar, se describen las estructuras de tiempo de ejecución y la función $\mathsf{vcg}[\cdot]$, que calcula el tiempo de ejecución de un programa y sus obligaciones de prueba. Por último, se describe el proceso de la verificación de invariantes, en específico, la forma en que se adaptaron las obligaciones de prueba a un *input* compatible con un *smt-solver*.

La Figura 4.1 muestra el flujo de la herramienta desarrollada para analizar el tiempo de ejecución de un programa probabilístico. Inicia recibiendo un programa escrito en sintáxis concreta, luego se convierte ese programa a su representación abstracta, para luego poder calcular tanto el tiempo de ejecución como las obligaciones de prueba.

Si el programa tiene ciclos, se procede a analizar sus invariantes. Este paso es uno de los procesos que requirió más esfuerzos para poder concretarse, necesitando un algoritmo sistemático que se describirá en este capítulo. Si los invariantes no son correctos, se imprime uno contraejemplo que demuestra que la obligación de prueba asociada al invariante falla. La información anterior es necesaria para que el analista pueda corregir los invariantes e iniciar nuevamente el proceso de análisis.

En el caso que los invariantes sean correctos o el programa no contenga ciclos, se imprime el tiempo de ejecución antes calculado y se da por finalizado el análisis del programa. Para ejemplificar el procedimiento diseñado para la verificación de las obligaciones de prueba se usará el programa de la Figura 4.2, el cual es una modificación de un ejemplo presentado en el trabajo original. Este programa está representado en su sintáxis concreta y es ciclo `while` con su respectivo invariante, el cual se presenta en la línea número 3. Este está escrito en sintaxis concreta y es la representación del tiempo de ejecución $2 \cdot \mathtt{x} \cdot [\![\mathtt{x} > 0]\!] + 1$. Este invariante es incorrecto y a continuación se demuestra superficialmente la manera de encontrar un contraejemplo.

- El primer paso es calcular la obligación de prueba asociada al invariante, en este caso la obligación es $1 + [\![\mathtt{x} > 0]\!] \cdot (2 + 2 \cdot (\mathtt{x} - 1) \cdot [\![\mathtt{x} - 1 > 0]\!]) \leq 1 + 2 \cdot \mathtt{x} \cdot [\![\mathtt{x} > 0]\!]$.

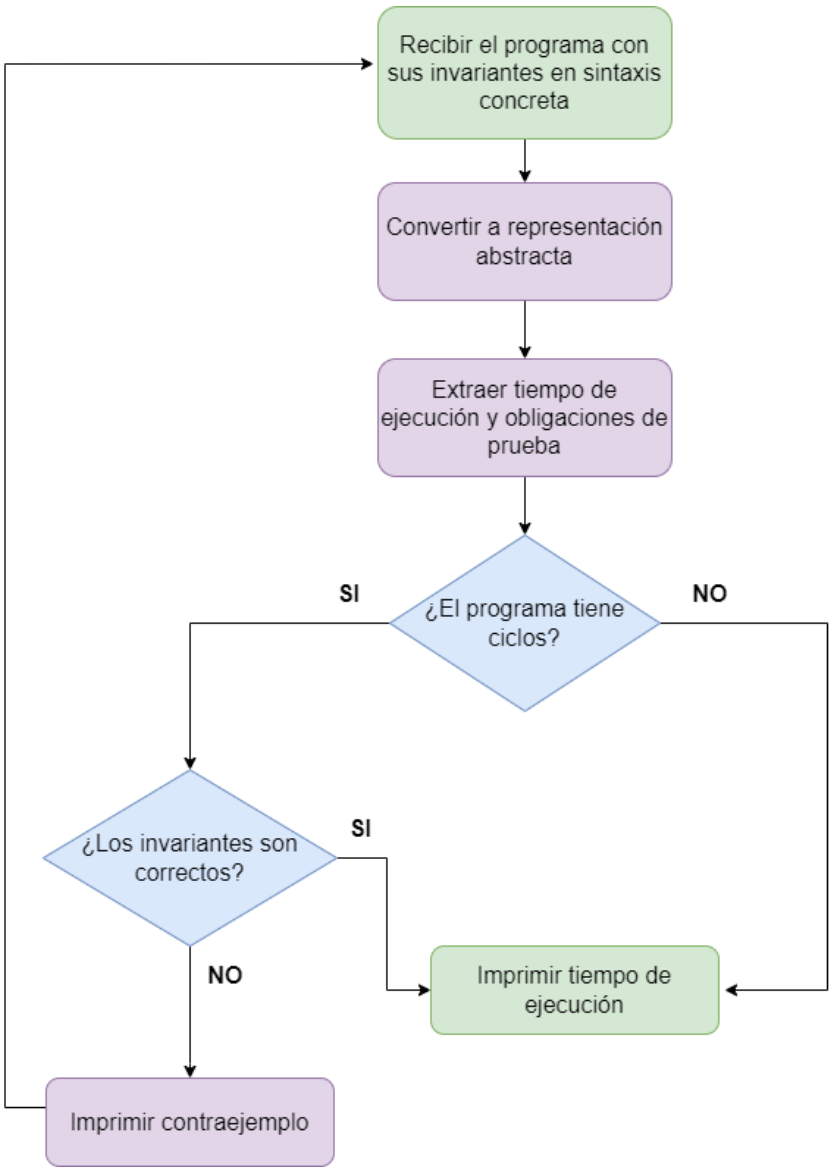

Figura 4.1: Flujo para analizar el tiempo de ejecución de un programa probabilístico.

- El segundo paso es identificar todas las expresiones booleanas que sean sub-expresiones de la obligación de prueba. En este caso las expresiones booleanas son $(\texttt{x} > \texttt{0})$ y $(\texttt{x} - \texttt{1} > \texttt{0})$.
- El tercer paso es tomar como hipótesis un valor de verdad de las expresiones booleanas anteriores y evaluarlas dentro de las indicatrices de la obligación de prueba. Se debe repetir este proceso para todas posibles combinaciones de valores de verdad. Por ejemplo, una combinación es asumir que $\texttt{x} > \texttt{0}$ es verdadero y $\texttt{x} - \texttt{1} > \texttt{0}$ es falso, la obligación se transforma en $1 \leq \texttt{x}$. Este proceso se puede representar de manera compacta a través del operador implica $(\Rightarrow)$, y se interpreta como una fórmula lógica. Todos las posibles

```
1  x := 3;
2  while (x > 0)
3        { inv = 1 ++ 2  ** [x > 0] <> x}
4        { x := x − 1 }
```

Figura 4.2: Ciclo `while` con invariante incorrecto.

fórmulas lógicas se detallan a continuación:

$$\begin{aligned} (\mathtt{x} > 0) \land (\mathtt{x} - 1 > 0) &\Rightarrow 0 \leq 0 \\ \neg(\mathtt{x} > 0) \land (\mathtt{x} - 1 > 0) &\Rightarrow 0 \leq 0 \\ (\mathtt{x} > 0) \land \neg(\mathtt{x} - 1 > 0) &\Rightarrow 1 \leq \mathtt{x} \\ \neg(\mathtt{x} > 0) \land \neg(\mathtt{x} - 1 > 0) &\Rightarrow 0 \leq 0 \end{aligned} \tag{4.1}$$

- El cuarto paso es validar cada una de las fórmulas generadas en el paso anterior. Si todas las fórmulas son válidas, entonces la obligación de prueba original lo es. Para lograr este paso se procede mediante la contradicción. Con este enfoque se necesita que la negación de las fórmulas anteriores no sean válidas para que la obligación original lo sea. En el caso que alguna fórmula sea válida, entonces existe un modelo que la satisface; este modelo es un contraejemplo que demuestra que la obligación original no es válida y el invariante asociado no es correcto.

  Siguiendo el ejemplo, si se toma como verdadero ambas expresiones booleanas, la fórmula resultante no tiene un modelo válido, pero si se asume $\mathtt{x} > 1$ verdadero y $\mathtt{x} - 1 > 0$ falso, entonces $\mathtt{x} = \frac{1}{2}$ es un modelo válido y, en consecuencia, un contraejemplo para la obligación de prueba. Esto último es fácilmente comprobable al reemplazar $\mathtt{x} = \frac{1}{2}$ en la obligación original. Al hacer este ejercicio se obtiene $1 \leq \frac{1}{2}$, lo que es claramente falso.

A grandes rasgos, la descripción anterior es la forma de comprobar la correctitud de una obligación de prueba asociada a un invariante. Claramente faltaron muchos detalles importantes, pero estos serán descritos a lo largo del presente capítulo.

4.1. Estructuras recursivas

En esta sección se detallan las estructuras recursivas implementadas y sus principales características, por ejemplo, la sintaxis concreta y sintaxis abstracta[1]. Un punto importante a tener en cuenta es que este apartado guarda una relación directa con la teoría descrita en el trabajo original. Muchas de las estructuras son descritas en mayor o menor medida en ese escrito y en el presente, solo se presenta una adaptación para facilitar la implementación en Haskell.

[1]Con el fin de mantener la lectura lo más clara posible, en esta sección se profundiza en la sintáxis abstracta y las principales ideas de las sintáxis concreta. Para obtener más detalles de la sintáxis concreta ver Apéndice A

La principal diferencia entre las estructuras descritas en la teoría de la transformada $\mathsf{ert}[\cdot]$ y las expuestas en este apartado, es que en esta investigación se aborda uno de los inconvenientes expuestos en la sección anterior: estructuras que no diferencian el comportamiento determinista y su contraparte probabilística. Para solucionar esta situación, se optó por usar dos estructuras diferentes. Las distribuciones aritméticas μ (que al ser evaluados en un estado σ se convierten en una distribución sobre los racionales), las distribuciones de probabilidad sobre los booleanos ξ y los programas probabilísticos pProg fueron modificados con este enfoque.

4.1.1. Expresiones aritméticas deterministas `AExp`

Las expresiones aritméticas (`AExp`) son la estructura medular para la construcción de la mayoría de las estructuras que se detallarán en el apartado. Dada esta situación, es natural que la sintaxis de `AExp` influye de forma directa o indirecta en los alcances y limitaciones que tiene la herramienta. A modo de ejemplo, el tipo de variable numérica escogida define la teoría base del *smt-solver* usada en la verificación de invariantes.

Definición 4.1 (Sintaxis abstracta de las expresiones aritméticas deterministas)

`AExp` $\rightarrow$	`q`	*Número racional*
	`x`	*Variable*
	`q` $*$ `AExp`	*Ponderación de una expresión* `AExp` *por constante racional*
	`AExp` $+$ `AExp`	*Suma de expresiones* `AExp`

La estructura inicia definiendo las variables numéricas de tipo racional. El principal motivo para elegir a los racionales es la posibilidad de representar distribuciones de probabilidad sin recurrir a aproximaciones. Por ejemplo, la probabilidad de éxito de un dado de seis caras se representa satisfactoriamente como $\frac{1}{6}$.

El siguiente caso es la definición de variables. Las variables, como es natural, se representan a través de *strings* y son necesarias para representar expresiones no constantes como $\mathtt{x} + \mathtt{1}$.

Los casos inductivos inician con la ponderación de expresiones `AExp`. Con el propósito de mantener la consistencia, es necesario que la ponderación se mantenga dentro de los racionales de tal manera que, a la hora de resolver los problemas en el *smt-solver*, baste una sola teoría subyacente. El último caso inductivo es la suma de `AExp`, necesaria para ir formando expresiones más complejas y extensas en el caso de que se requiera.

Al hacer un análisis sobre la sintaxis expuesta, es claro que las expresiones representadas son las expresiones aritméticas lineales. El principal motivo de acotar `AExp` a este tipo de expresiones es la capacidad resolutiva de los *smt-solvers*, además de facilitar el proceso de validación. El tener expresiones no lineales como $\mathtt{x}^2 + \mathtt{y}^2 + \mathtt{sin}(\mathtt{x})$ dificulta mucho los cálculos teóricos y, en consecuencia, la comparación entre la teoría y la automatización.

La sintaxis concreta de `AExp` es bastante similar a la abstracta. En términos generales, el objetivo fue disminuir la cantidad de paréntesis en la notación, haciendo especial énfasis en anotar monomios de manera cómoda y clara. Relacionado con lo anterior, la notación para los racionales también buscó la comodidad. En este caso se buscó escribir las fracciones de

manera natural (por ejemplo, 1/7), además de tener la posibilidad de escribir los enteros sin la necesidad usar el denominador (3 y no 3/1). Otro punto importante es que se añade la resta como azúcar sintáctica, ya que es una operación altamente utilizada. Un ejemplo de expresión aritmética en sintáxis concreta es:

$$\mathtt{x} + 2 * \mathtt{y} - 3/2 + 3 * (\mathtt{x} + \mathtt{y})$$

4.1.2. Expresiones aritméticas probabilísticas `PAExp`

Las expresiones aritméticas probabilísticas (`PAExp`) son la contraparte probabilística de `AExp` y, a la vez, son el complemento de las mismas para formar las expresiones originales μ. Aunque el uso de estas expresiones se encuentra acotado a las asignaciones probabilísticas, no dejan tener gran importancia, ya que de no tenerlas, no se podría tomar muestras de distribuciones de probabilidad con expresiones como $\mathtt{x} :\approx \mathtt{uniform}[0, 1]$.

Definición 4.2 (Representación abstracta de las expresiones aritméticas probabilísticas)

$$\mathtt{PAExp} \rightarrow \quad \Sigma_1^{\mathtt{n}} \mathtt{p_i} \cdot \mathtt{a_i} \quad \textit{Distribución sobre } \mathtt{AExp} \textit{ con } \mathtt{p_i} \textit{ racional} \in [0, 1], \mathtt{a_i} \in \mathtt{AExp} \textit{ y } \Sigma_1^{\mathtt{n}} \mathtt{p_i} = 1$$

Se denomina representación y no sintaxis abstracta debido a que se usó la implementación de listas Haskell, en vez de crear una sintaxis nueva. Conceptualmente, una expresión perteneciente al conjunto `PAExp` es la unión de los puntos de una distribución donde $\mathtt{p_i}$ es la probabilidad que le asigna a una expresión $\mathtt{a_i}$.

La sintaxis concreta de `PAExp` es similar a la definición antes presentada, ya que, se usa la operación de la suma para ir agregando puntos a una distribución. A modo de ejemplo, la expresión

$$1/2 * < 0 > + \, 1/2 * < 1 >$$

representa el lanzamiento de una moneda equilibrada sobre el conjunto $\{0, 1\}$. Vale la pena mencionar que, además de esta notación, se implementaron algunos *shortcuts* para representar algunas distribuciones conocidas, por ejemplo, la distribución uniforme se representa con la expresión `uniform`.

4.1.3. Expresiones Booleanas deterministas `BExp`

Las expresiones booleanas deterministas (`BExp`) representan las condiciones usadas en las guardas de los `if`, `while` e indicatrices. Es clara la importancia de esta estructura, ya que permite elaborar programas imperativos cercanos a los usados comúnmente y, además, forman parte de la estructura de tiempos de ejecución a través de las indicatrices antes mencionadas.

Definición 4.3 (Sintaxis abstracta de las expresiones booleanas deterministas)

`BExp` $\rightarrow$	`true`, `false`	*Constante booleana*
	`AExp` == `AExp`	*Igualdad de expresiones* `AExp`
	`AExp` <= `AExp`	*Desigualdad de expresiones* `AExp`
	`BExp` && `BExp`	*Conjunción de* `BExp`
	`BExp` || `BExp`	*Disyunción de* `BExp`
	!`BExp`	*Negación de* `BExp`

La sintaxis abstracta, como es natural, comienza con las constantes booleanas `true` y `false`. Los siguientes casos son la fuente de variabilidad de la estructura, la desigualdad e igualdad de `AExp`. Por último, se encuentran las operaciones habituales entre booleanos, disponibles en cualquier lenguaje de programación.

La sintaxis concreta suma varias operaciones entre `AExp`, todas son azúcar sintáctica de las definidas en la abstracta. Algunos ejemplos son el mayor o igual (>=), la no igualdad (! =) o el menor estricto (<). Un ejemplo de expresión `BExp` en sintáxis concreta es:

$$(\texttt{x} <= \texttt{y}) \;\&\&\; (!\texttt{true}) \;\&\&\; (\texttt{x} > \texttt{10})$$

4.1.4. Expresiones booleanas probabilística `PBExp`

Las expresiones booleanas probabilísticas (`PBExp`) son el complemento de `BExp` para formar las expresiones ξ descritas en el trabajo guía y representan a las distribuciones Bernoulli sobre los booleanos. Aunque su uso se limita a las guardas de las versiones probabilísticas de un `while` o un `if`, es clara su utilidad, ya que, junto a `PAExp`, son las fuentes de aleatoriedad de los programas probabilísticos estudiados en este trabajo.

Definición 4.4 (Sintaxis abstracta expresiones booleanas probabilistas)

`PBExp` $\rightarrow$	`Ber`(`p`)	*Distribución booleana Bernoulli con p racional* $\in$ *[0, 1]*

La sintaxis abstracta tiene solo un caso, aun así, es sustancial hacer énfasis que el valor `p` es un número racional, al igual que todas las variables numéricas del trabajo. En cuanto a la sintaxis concreta, un ejemplo de distribución es:

$$< 1/3 >$$

que representa una distribución Bernoulli con probabilidad de éxito $\frac{1}{3}$, es decir, asigna la probabilidad de $\frac{1}{3}$ a `true` y $1 - \frac{1}{2} = \frac{2}{3}$ de probabilidad al valor `false`.

4.1.5. Tiempos de ejecución `RunTime`

Los tiempos de ejecución (`RunTime`) son unos de los objetos centrales a lo largo del presente escrito y son la representación del conjunto $\mathbb{T}$ del trabajo guía. La principal aplicación de esta estructura es ser el conjunto al que pertenecen las predicciones de tiempos de ejecución e invariantes de ciclos.

Definición 4.5 (Sintaxis abstracta de los tiempos de ejecución)

`RunTime` $\rightarrow$	`AExp`	*Expresión aritmética* `AExp`
	$[$`BExp`$] \cdot$ `RunTime`	*Multiplicación por una indicatriz*
	`q` $*$ `RunTime`	*Ponderación por constante racional*
	`RunTime` $+$ `RunTime`	*Suma de expresiones* `RunTime`

La principal justificación tras el diseño de esta sintaxis es su capacidad de representar de manera óptima los cálculos propuestos en la Tabla 3.1 (demostración Sección B.2), con la excepción de la transformada del ciclo `while` y la función de sustitución. Para el primer caso no hay inconvenientes, ya que en este trabajo se usa invariantes para razonar sobre los ciclos, pero el segundo caso representa un problema. La sintáxis abstracta no permite representar la operación de sustitución. Para ejecutar esta operación se requirió una función extra. A esta función se le denomina `sustRunTime` (definición Apéndice B.3) y es la encargada de reemplazar todas las incidencias de una variable x en un tiempo de ejecución $f \in$ `RunTime` por una expresión aritmética `a`, retornando el tiempo de ejecución $f\lceil$`x`$/$`a`$\rceil \in$ `RunTime`.

En cuanto a la sintaxis concreta, esta sigue los mismos principios que la hecha para el conjunto `AExp`. La gran diferencia es que las operaciones denotan con símbolos dobles, por ejemplo, la suma de `x` más `y` se anota como `x` $++$ `y`. Misma situación ocurre la ponderación y la resta (que es azúcar sintáctica). Por último, la operación para el caso de la multiplicación por una indicatriz en sintaxis concreta se representa con el símbolo $<>$. Es importante detallar que expresiones de la forma k $**$ $[$`BExp`$]$ $<>$ (`AExp`) pueden ser escritas sin necesidad de un paréntesis entre la multiplicación y la indicatriz, y expresiones de la forma k $**$ $[$`BExp`$]$ son azúcar sintáctica de k $**$ $[$`BExp`$]$ $<>$ `1`. Era necesario poder escribir las expresiones anteriores de manera cómoda debido a que son altamente utilizadas al momento de escribir un invariante.

Un ejemplo de un elemento perteneciente a `RunTime` es :

$$2 ** [\mathtt{x} > 10] -- 3/2 ** [\mathtt{y} == 5] <> (\mathtt{x} + 4)$$

4.1.6. Programas probabilísticos `Program`

El conjunto de programas probabilísticos (`Program`), en esencia, es equivalente al conjunto pProg antes mencionado. La diferencia radica en que la sintaxis de `Program` es más refinada, ya que separa los programas deterministas de su contraparte probabilista. Lo anterior permite razonar de manera más cómoda sobre los programas, por ejemplo, con `Program` se analiza de manera separa el `if` probabilístico y el `if` clásico, cosa que no ocurre con la sintaxis original.

Definición 4.6 (Sintaxis abstracta de los programas probabilísticos)

`Program` →	`empty`	*Programa vacío sin gasto de tiempo*
	`skip`	*Programa vacío con gasto de tiempo*
	`x` $\coloneqq$ `AExp`	*Asignación determinista*
	`x` $\coloneqq\approx$ `PAExp`	*Asignación probabilística*
	`Program ; Program`	*composición secuencial*
	`if (BExp){Program} else {Program}`	*Condicional*
	`pif (PBExp) {Program} pelse {Program}`	*Condicional probabilístico*
	`while (BExp) {RunTime} {Program}`	*Ciclo determinista*
	`pwhile (PBExp) {RunTime} {Program}`	*Ciclo probabilístico*

La sintaxis abstracta, con el fin de seguir el enfoque antes mencionado, agrega tres nuevos constructores con respecto a lo planteado en pProg. El primer constructor es la asignación determinista, que en la sintaxis original era reemplazada con la asignación `x` $\coloneqq\approx \mu$, donde μ es una distribución de dirac.

El segundo constructor es el condicional probabilístico `pif`, el que se encarga de representar programas como el C_{trunc}. Por último, se agrega el ciclo probabilístico `pwhile`, cuya semántica es evaluar el cuerpo hasta que la variable aleatoria de la guarda sea falsa.

Otra diferencia con respecto al lenguaje pProg es la adición del respectivo invariante en el constructor de un ciclo. Como se mencionó anteriormente, la síntesis de invariante no es parte fundamental del trabajo y solo fue abordada de manera superficial.

Definición 4.7 (Sintaxis concreta de los programas probabilísticos)

`program` →	`empty`	*Programa vacío sin gasto de tiempo*
	`skip`	*Programa vacío con gasto de tiempo*
	`x` $\coloneqq$ `aexp`	*Asignación determinista*
	`x` $:\sim$ `paexp`	*Asignación probabilística*
	`program ; program`	*Composición secuencial*
	`if (bexp){program} else {program}`	*Condicional de dos ramas*
	`pif (pbexp) {program} pelse {program}`	*Condicional de dos ramas probabilístico*
	`it (bexp) {program}`	*Condicional de una rama*
	`pit (pbexp) {program}`	*Condicional de una rama probabilístico*
	`while (bexp) {runtime} {program}`	*Ciclo determinista*
	`pwhile (pbexp) {runtime} {program}`	*Ciclo probabilístico*
	`for (n) {program}`	*For estático con n entero*

La sintaxis concreta agrega algunos constructores como azúcar sintáctica. Los programas `it`/`pit` (abreviación de `if then`) son condicionales cuya rama `false` es el programa vacío sin efecto `empty`. El otro programa agregado es el for estático, que representa la composición secuencial de un programa `C` un número `n` de veces. Ambos casos permiten facilitar la escritura de programas usados, en mayor o menor medida, en los lenguajes imperativos habituales.

4.1.7. Simplificaciones

Un punto no detallado hasta el momento es el modo de simplificar las diferentes expresiones antes presentadas. Recordemos que los cálculos de las predicciones pueden crecer rápidamente, esto en principio, no es una dificultad para el *solver*, pero sí lo es al momento de la validación. Al ojo humano, no es trivial reconocer que dos expresiones extensas sean iguales.

El alcance de simplificaciones varía entre las estructuras. Para `AExp` la simplificación llega hasta una forma normal, en cambio para `BExp` y `RunTime`, las simplificaciones llegan hasta un nivel medio, pero lo suficientemente profundo como para que el proceso de validación sea factible. Dado lo anterior, a lo largo de esta memoria se discutirá sobre expresiones simplificadas y no normalizadas. A continuación, se presenta un ejemplo de simplificación sobre un elemento de `RunTime`, que explícitamente ocupa la simplificación de `BExp` y `AExp`.

Ejemplo

Expresión original

$1 \mathbin{++} 1 \mathbin{++} 2 \mathbin{**} [\texttt{true} || (\texttt{x} > 0)] \diamond \texttt{x} - - \texttt{y}$

Suma de constantes, simplificación `AExp`

$2 \mathbin{++} 2 \mathbin{**} [\texttt{true} || (\texttt{x} > 0)] \diamond \texttt{x} - - \texttt{y}$

Disyunción de booleanos, simplificación `BExp`

$2 \mathbin{++} 2 \mathbin{**} [\texttt{true}] \diamond \texttt{x} - - \texttt{y}$

Evaluación de la indicatriz, simplificación `RunTime`

$2 \mathbin{++} 2 \mathbin{**} \texttt{x} - - \texttt{y}$

4.1.8. *Parser*

Desde lo conceptual, las reglas del *parser* desarrollado tiene mínimas diferencias con respecto a las sintáxis concretas de las diferentes estructuras antes presentadas. Debido a lo anterior, lo más interesante se encuentra en el cómo se desarrolló la implementación. Al igual que el resto del proyecto, el desarrollo del *parser* fue hecho en Haskell, es especifico, se usó la librería Text.Parsec, y algunas sub-librerías provenientes de esta.

Como es esperable, al ser Text.Parsec una librería de Haskell, la mayoría de la implementación se hizo a través de una mónada. La mónada Parser permite definir un *parser* para las estructuras más simples y usar esa definición, de manera inductiva, en una estructura más compleja. A modo de ejemplo, el *parser* de los elementos `RunTime` necesita su homólogo en `AExp`.

Cada *parser* tiene asociado una serie de reglas que definen la sintáxis concreta. Estas reglas las podemos dividir en dos tipos. El primer tipo son las reglas que hacen referencia a constantes o estructuras de menor complejidad. Por ejemplo, para las expresiones `BExp` se necesitó una regla para las constantes `true` y `false` y una para cada operación relacional entre expresiones `AExp`.

Para finalizar, se explica el segundo tipo de reglas. Este tipo son las reglas que hacen referencia a la misma estructura solamente. Nótese que una regla que haga referencia a `BExp` y `RunTime` no caería en esta categoría, sino que en la anterior. Estas son más interesantes, ya que es posible decidir, de manera cómoda, sobre la precedencia, asociación y tipo de notación. Siguiendo el caso de las expresiones `BExp`, la negación es la única operación unaria y tiene mayor precedencia que las operaciones binarias de la conjunción y disyunción. Estas operaciones binarias asocian hacia la izquierda y se escriben con notación infija (al igual que la mayoría de lenguajes imperativos).

4.2. Cálculo de obligaciones de prueba

Tal como se introdujo en el Marco Teórico (Ver sección 2.2), las obligaciones de prueba avalan la correctitud de un invariante. En el presente contexto una obligación de prueba se origina a partir de la condición de la Definición 3.3. Para poder trabajar con estas obligaciones de prueba es necesario crear una estructura que permita la manipulación de los tiempos de ejecución que las conforman.

Definición 4.8 ($\mathbb{O}$ Sintaxis abstracta)

$\mathbb{O} \rightarrow \quad \texttt{RunTime} \leq \texttt{RunTime}$ *Menor igual entre* `RunTime`

Una vez introducida la sintaxis de $\mathbb{O}$, tenemos los elementos necesarios para introducir la función más importante de este trabajo, la función $\mathsf{vcg}[\cdot]$ (siglas que hacen referencia *verification condition generator* al igual que la función VCGen, presentada en la Sección 2.2). La firma de esta función es

$$\mathsf{vcg}[\cdot] : \texttt{Program} \rightarrow \big(\texttt{RunTime} \rightarrow (\texttt{RunTime}, 2^{\mathbb{O}})\big)$$

donde $2^{\mathbb{O}}$ es el conjunto potencia de $\mathbb{O}$.

La expresión $\mathsf{vcg}[\texttt{C}](f)$ entrega un par donde el primer componente es el tiempo de ejecución estimado de `C` (coincide en la mayoría de los casos con $\mathsf{ert}[\texttt{C}](f)$) y el segundo es el conjunto de obligaciones de prueba asociadas a los invariantes del programa `C`, es decir, es un VCgen. Vale la pena hacer énfasis en que el segundo elemento es un conjunto, por lo tanto, tiene asociado todas las operaciones de conjuntos (usaremos la unión de conjuntos $\cup$) y además, existe el conjunto vacío $\emptyset$. En el caso que el programa no tenga invariantes se retorna una tupla cuya segunda componente es precisamente ese elemento.

Definición 4.9 *Definición de la función* $\mathsf{vcg}[\cdot]$

$\mathsf{vcg}[\texttt{empty}](f)$ $= (f, \emptyset)$

$\mathsf{vcg}[\texttt{skip}](f)$ $= (1 + f, \emptyset)$

$\mathsf{vcg}[\texttt{x} \coloneqq \texttt{a}](f)$ $= (1 + f[\texttt{x}/\texttt{a}], \emptyset)$

$\mathsf{vcg}[\texttt{x} :\approx \Sigma_1^{\texttt{n}} \texttt{p}_\texttt{i} \cdot \texttt{a}_\texttt{i}](f)$ $= (1 + \Sigma_1^{\texttt{n}} \texttt{p}_\texttt{i} \cdot f[\texttt{x}/\texttt{a}_\texttt{i}], \emptyset)$

$\mathsf{vcg}[\texttt{C}_1 \ ; \ \texttt{C}_2](f)$ $= (f_1, S_1 \cup S_2)$
donde $(f_2, S_2) = \mathsf{vcg}[\texttt{C}_2](f)$ *y* $(f_1, S_1) = \mathsf{vcg}[\texttt{C}_1](f_2)$

$\mathsf{vcg}[\texttt{if (b)}\{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\}](f)$ $= (1 + [\![\texttt{b}]\!] \cdot f_1 + [\![\neg \texttt{b}]\!] \cdot f_2, \ S_1 \cup S_2)$
donde $(f_2, S_2) = \mathsf{vcg}[\texttt{C}_2](f)$ *y* $(f_1, S_1) = \mathsf{vcg}[\texttt{C}_1](f)$

$\mathsf{vcg}[\texttt{pif } (< \texttt{p} >) \ \{\texttt{C}_1\} \texttt{ pelse } \{\texttt{C}_2\}](f)$ $= (1 + \texttt{p} \cdot f_1 + (1 - \texttt{p}) \cdot f_2, \ S_1 \cup S_2)$
donde $(f_2, S_2) = \mathsf{vcg}[\texttt{C}_2](f)$ *y* $(f_1, S_1) = \mathsf{vcg}[\texttt{C}_1](f)$

$\mathsf{vcg}[\texttt{while (b) } \{\texttt{C}'\} \ \{\texttt{I}\}](f)$ $= (\texttt{I}, \ \{ \ 1 + [\![\texttt{b}]\!] \cdot f_1 + [\![\neg \texttt{b}]\!] \cdot f \leq \texttt{I}\} \cup S)$
donde $(f_1, S) = \mathsf{vcg}[\texttt{C}'](\texttt{I})$

$\mathsf{vcg}[\texttt{pwhile } (< \texttt{p} >) \ \{\texttt{C}'\} \ \{\texttt{I}\}](f)$ $= (\texttt{I}, \ \{ \ 1 + \texttt{p} \cdot f_1 + (1 - \texttt{p}) \cdot f \leq \texttt{I}\} \cup S)$
donde $(f_1, S) = \mathsf{vcg}[\texttt{C}'](\texttt{I})$

La definición de $\mathsf{vcg}[\cdot]$ está basada en la de $\mathsf{ert}[\cdot]$ y las funciones generadoras presentadas en el Marco Teórico (Sección 2.2). Por lo mismo, existe una gran cercanía entre $\mathsf{vcg}[\cdot]$ y sus funciones predecesoras. La principal diferencia es algorítmica, en un solo recorrido sobre un programa $\mathsf{vcg}[\cdot]$ logra calcular el tiempo de ejecución y las obligaciones de prueba.

Para los casos base (`empty`, `skip`, asignación probabilística y determinista) la primera componente de $\mathsf{vcg}[\texttt{C}](f)$ coincide exactamente con $\mathsf{ert}[\texttt{C}](f)$ y la segunda componente es $\emptyset$ debido a que no generan obligaciones de prueba.

Para los casos inductivos `if`, `pif` y la composición secuencial la primera componente de $\mathsf{vcg}[\texttt{C}](f)$ también coincide con $\mathsf{ert}[\texttt{C}](f)$. En la segunda componente se retorna la unión de las obligaciones de pruebas generadas en los dos programas que componen a `C`.

Los casos que tienen el mayor interés son los ciclos `while` y `pwhile`. Para la primera componente se retorna el invariante `I` (recordemos que el invariante representa una cota superior). En la segunda componente se retorna la obligación de prueba generada por `I` (generada según la Definición 3.3), unión las obligaciones generadas en el cuerpo del ciclo `while`.

4.3. Verificación de obligaciones de prueba

En este apartado se describe el proceso diseñado para verificar las obligaciones de prueba generadas con la función $\mathsf{vcg}[\cdot]$. Se reitera que las obligaciones tienen la forma de $f_1 \leq f_2$ y

estas son válidas cuando:

$$\forall \sigma \ f_1(\sigma) \leq f_2(\sigma)$$

Para comprobar la condición anterior, a diferencia de las estructuras y funciones anteriores, no basta solo con las librerías habituales de Haskell, sino que se necesita de la librería SBV para usar los *smt-solver*.

SBV es una librería que provee una interfaz uniforme para trabajar con diferentes *smt-solvers*. En el que caso de esta investigación se trabajó exclusivamente con Z3, pero las funcionalidades son extensibles a otros *smt-solvers* sin mayores inconvenientes.

Con SBV se pueden plantear y resolver fórmulas lógicas (encontrar un modelo que satisface las fórmulas), como $x > y \wedge y > 0$, usando la teoría de las expresiones aritméticas. En los siguientes apartados se detallará cómo, a partir de una obligación de prueba se obtienen una o más de estas fórmulas. A este tipo de fórmula se le denominará *problema lineal*. En el contexto actual, un problema lineal es una conjunción de desigualdades e igualdades de expresiones `AExp`, a las que se les intenta encontrar un modelo que las satisfaga.

Para obtener un problema lineal, a partir de una obligación de prueba $f_1 \leq f_2$, se necesitan dos pasos. El primer paso es la linealización, es necesario dejar de trabajar con expresiones `RunTime` y empezar a manipular expresiones `AExp`. El segundo paso es cambiar el cuantificador del problema, en vez de trabajar con un para todo ($\forall$), es necesario trabajar con el existe ($\exists$). El motivo de este último punto es el funcionamiento de los *smt-solver*. Estas herramientas buscan encontrar un modelo que satisface un conjunto de restricciones o, expresado de otra forma, buscan comprobar la existencia de un modelo.

4.3.1. Linealización de las obligaciones de prueba

Antes de la linealización de una obligación de prueba, hay que atacar la linealización de los tiempos de ejecución que la conforman. Una vez hecho este paso previo, se puede extender el procedimiento mediante modificaciones menores. Para este paso previo, el que es transformar una expresión `RunTime` a una expresión `AExp`, es necesario manipular las expresiones no lineales de la sintaxis abstracta de `RunTime`. Mediante la exploración, es claro que el único constructor que incluye la no linealidad es la multiplicación por una indicatriz, por ejemplo, $\mathtt{I_{geo}} = \mathtt{1} + \mathtt{4} \cdot [\![\mathtt{c} == \mathtt{1}]\!]$ ocupa este constructor.

Para trabajar con las indicatrices es necesario recordar que estas se forman a partir de una expresión booleana interna, la evaluación de esa expresión (`false` o `true`) determina el valor de la respectiva indicatriz (0 o 1). Asimismo, el valor de una indicatriz determina el valor de respectivo tiempo de ejecución. A modo de ejemplo, para $\mathtt{I_{geo}}$, en el caso que $\mathtt{c} == \mathtt{1}$ sea verdadero (sentencia que se abrevia con $\mathtt{c} == \mathtt{1}_+$) se tiene que

$$\mathtt{I_{geo}} = \mathtt{5}$$

En el caso contrario, si $\mathtt{c} == \mathtt{1}$ es falso (sentencia que se abrevia con $\mathtt{c} == \mathtt{1}_-$) se tiene que

$$\mathtt{I_{geo}} = \mathtt{1}$$

En términos generales, para linealizar una expresión `RunTime` se debe tomar como hipótesis el valor de 0 o más expresiones `BExp`, llegando de esta manera a una expresión `AExp`. Este procedimiento tiene la forma del operador lógico implica y es, en esencia, la forma de linealizar los tiempos de ejecución. Por ejemplo, la *linealización* de $\mathtt{I_{geo}}$ se representa de forma compacta como

$$\begin{aligned} \mathtt{c} == \mathtt{1}_+ &\Rightarrow \mathtt{I_{geo}} = \mathtt{5} \\ \mathtt{c} == \mathtt{1}_- &\Rightarrow \mathtt{I_{geo}} = \mathtt{1} \end{aligned} \tag{4.2}$$

Con respecto a esta representación, es necesario introducir algunas observaciones. En primer lugar, debido al comportamiento dual de la negación, la expresión $\mathtt{c} == \mathtt{1}_-$ es reemplazable por $\mathtt{!c} == \mathtt{1}_+$. En segundo lugar, y complementando la observación anterior, debido a que es posible trabajar solo con la evaluación verdadera, se pueden omitir los subíndices $+$.

Estas dos observaciones no solo afectan la notación, también ayudan de gran manera a la implementación. El Ejemplo 4.2 describe la linealización con el uso de expresiones `BExp` "puras", de no existir esta posibilidad hubiese sido necesario implementar una estructura que permita guardar la relación entre una expresión booleana y su respectivo valor de verdad.

Para extender el procedimiento anterior a las obligaciones de prueba, basta con aplicar el mismo a los dos tiempos de ejecución que la conforman. A modo de ejemplo, para la obligación $\mathtt{W} = \mathtt{I_{geo}} \leq [\![\mathtt{!c} < \mathtt{0}]\!] \cdot \mathtt{c}$, la linealización se puede plantear como:

$$\begin{aligned} (\mathtt{c} == \mathtt{1}) \wedge (\mathtt{!c} < \mathtt{0}) &\Rightarrow \mathtt{5} \leq \mathtt{c} \\ (\mathtt{!c} == \mathtt{1}) \wedge (\mathtt{!c} < \mathtt{0}) &\Rightarrow \mathtt{1} \leq \mathtt{c} \\ (\mathtt{c} == \mathtt{1}) \wedge (\mathtt{c} < \mathtt{0}) &\Rightarrow \mathtt{5} \leq \mathtt{0} \\ (\mathtt{!c} == \mathtt{1}) \wedge (\mathtt{c} < \mathtt{0}) &\Rightarrow \mathtt{1} \leq \mathtt{0} \end{aligned} \tag{4.3}$$

El proceso anterior necesita algunos conceptos previos para ser formalizado. El primer concepto es el *conjunto atómico* de un $f \in$ `RunTime` (conjunto que se denota como $\mathcal{A}_f$). Intuitivamente, el conjunto atómico de f contiene a las expresiones `BExp` que sean sub-expresiones propias, es decir, las expresiones internas de las indicatrices. Se le denomina *atómico* debido a cercanía que tiene con las expresiones atómicas introducidas en la teoría de *smt-solvers*. Un átomo, en este contexto, es una expresión simplificada que no sea una negación y, en el caso de la negación, la expresión interna de esa operación. Por ejemplo, definamos la expresión $\mathtt{V} = [\![\mathtt{!c} > \mathtt{0}]\!] \cdot \mathtt{x} + \mathtt{2} \cdot [\![\mathtt{f} <= \mathtt{2/3} || \mathtt{true}]\!]$, su conjunto atómico es :

$$\mathcal{A}_{\mathtt{V}} = \{\mathtt{c} > \mathtt{0}, \mathtt{f} <= \mathtt{2/3}\}$$

Una vez clara la intuición del conjunto atómico, es posible introducir su definición formal.

Definición 4.10 (Conjunto Atómico de un tiempo de ejecución) *Sea $f \in$* `RunTime`*, se dice $\mathcal{A}_f$ es el conjunto atómico de f ssi,*

$\mathcal{A}_f$ es el conjunto de sub-expresiones `BExp` *de f, obtenidas mediante la función* `getBExp`*.*

La definición de `getBExp` se puede encontrar en Apéndice B.4 , aunque su funcionamiento no es más que extraer los átomos antes descritos. La extensión de la definición de conjunto atómico a una obligación de prueba es natural. El conjunto atómico $\mathcal{A}_{f_1 \leq f_2}$ de la obligación $f_1 \leq f_2$ se define como:

$$\mathcal{A}_{f_1 \leq f_2} = \mathcal{A}_{f_1} \cup \mathcal{A}_{f_2}$$

La necesidad de definir el conjunto atómico es clara; al asumir el valor de verdad de los átomos, se generan las hipótesis de los implicas antes ejemplificados.

Una vez introducido el concepto de conjunto atómico, es factible estudiar la estructura de los implicas, en específico, los conceptos de *contexto* y *restricción derivada*. El contexto es la conjunción de las hipótesis generadas a partir de los átomos, es decir, el antecedente del implica. Por el otro lado, la restricción derivada es la desigualdad que podemos extraer a partir de los valores del contexto, es decir, el consecuente del implica. Por ejemplo, en la Ecuación 4.3 un contexto es $(\texttt{c} == \texttt{1}) \wedge (!\texttt{c} < \texttt{0})$ y su restricción derivada es $5 \leq \texttt{c}$.

Si $|\mathcal{A}_{f_1 \leq f_2}| = n$, entonces un contexto es la conjunción de n hipótesis diferentes. Además, como cada hipótesis puede ser positiva(verdadera) o su negación (falsa), existen 2^n contextos posibles. Lo anterior provoca que existan 2^n implicas posibles. Debido a estas observaciones, la validación de una obligación de prueba tiene la siguiente forma equivalente:

$$\forall \sigma \ f_1(\sigma) \leq f_2(\sigma) \Longleftrightarrow \bigwedge_{i=1}^{2^n} \left(\forall \sigma \bigwedge_{j=1}^{n} h_{ij}(\sigma) \Rightarrow a_i(\sigma) \leq b_i(\sigma) \right) \tag{4.4}$$

donde h_{ij} es la hipótesis j del implica i, $\bigwedge_1^n h_{ij}$ representa el contexto del implica i y la desigualdad $a_i \leq b_i$ es la restricción derivada del implica i. Nótese que los implica se relacionan mediante la conjunción, es decir, se debe cumplir cada uno de los predicados, desde el 1 al 2^n para validar la obligación de prueba. Con la figura anterior y los conceptos introducidos, es posible abordar el siguiente punto, el cambiar el cuantificador del problema.

4.3.2. Cambio del cuantificador del problema

El segundo paso es cambiar el cuantificador del problema, transformar un para todo a un existe, para esto es necesario recordar que la expresión $\neg p \vee q$ es equivalente a $p \Rightarrow q$. Con esa observación se tiene la siguiente equivalencia.

$$\forall \sigma \ f_1(\sigma) \leq f_2(\sigma) \Longleftrightarrow \bigwedge_{i=1}^{2^n} \left(\forall \sigma \neg \Big(\bigwedge_{j=1}^{n} h_{ij}(\sigma) \Big) \vee a_i(\sigma) \leq b_i(\sigma) \right) \tag{4.5}$$

Dada la equivalencia anterior, están las condiciones de trabajar directamente con el cuantificador. La idea clave para lograr el objetivo es proceder mediante la contradicción. Al proceder por contradicción se cambia el cuantificador, y además, por teorema de Morgan, las conjunciones externas y la disyunción interna de cada implica cambian igualmente.

$$\forall \sigma \; f_1(\sigma) \leq f_2(\sigma) \Longleftrightarrow \neg \bigvee_{i=1}^{2^n} \left(\exists \, \sigma \Big(\bigwedge_{j=1}^{n} h_{ij}(\sigma) \Big) \wedge a_i(\sigma) > b_i(\sigma) \right) \tag{4.6}$$

El planteamiento anterior permite transformar una obligación de prueba a 1 o más problemas lineales. De esta manera, ya es posible comprobar la correctitud de una obligación con un *smt-solver*. El procedimiento se limita a comprobar que cada uno de los problemas lineales no sean satisfacibles, es decir, no existe el σ que satisfaga las restricciones. En el caso que exista alguno, entonces existe el contraejemplo para el cuantificador original y, por ende, la obligación no es correcta.

La siguiente figura entrega un seudo-algoritmo que permite calcular el tiempo de ejecución y comprobar las obligaciones de prueba, desde un programa `C` $\in$ `Program`.

Algorithm 1 Estimar tiempo de ejecución de `C` $\in$ `Program`

Require: $C \in String$
Ensure: $f' \in$ `RunTime`
  `C` $\leftarrow$ `parser` (C)
  $(f, \{o_i\}_1^n) \leftarrow$ `vcg[C](0)`
  $f' \leftarrow$ `simplificar` (f)
  $contraejemplos \leftarrow \emptyset$
  $satisfacibilidad\_problemas \leftarrow$ **false**
  for $i = 1...n$ **do**
    $p_i \leftarrow$ `transformar_problema_lineal` (o_i)
    $b_i \leftarrow$ `es_satisfacible?` (p_i)
    $satisfacibilidad\_problemas \leftarrow satisfacibilidad\_problemas \vee b_i$
    if b_i **then**
      $m_i \leftarrow$ `extraer_modelo` (p_i)
      $contraejemplos \leftarrow contraejemplos \cup (m_i, p_i)$
    end if
  end for
  if $satisfacibilidad\_problemas$ **then**
    print Algunos invariantes no son correctos
    print $contraejemplos$
  else
    print El análisis ha finalizado
    print f'
  end if

4.4. Síntesis del capítulo

En el presente capítulo se detallaron las características más relevantes de la solución elaborada. Se inicia describiendo las estructuras inductivas, las más importantes son las expresiones lineales sobre los números racionales (`AExp`), las expresiones booleanas deterministas (`BExp`),

los tiempos de ejecución (`RunTime`) y los programas probabilísticos (`Program`).

A continuación, se introduce la definición de la función más importante $\mathsf{vcg}[\cdot]$. Esta función calcula el tiempo de ejecución estimado de un programa y, además, sus obligaciones de prueba. Por último, se describe el proceso de verificación de las obligaciones de prueba. Este proceso se divide en dos etapas. La primera etapa se encarga de la linealización de las obligaciones, transformando cada una en un conjunto de implicas. La segunda etapa, mediante la contradicción, se encarga de cambiar el cuantificador del problema de un $\forall$ a un $\exists$. De esta forma, se logra transformar cada implica en un problema lineal. Si este problema es satisfacible, entonces existe un contraejemplo para la obligación de prueba que lo origina, entonces el invariante que produce esa obligación no es correcto y, en consecuencia, el tiempo de ejecución calcula no es válido. En caso contrario, que no exista el modelo para ningún problema lineal, significa que todos los invariantes son correctos y, por lo tanto, el tiempo de ejecución calculado es válido.

Capítulo 5

Validación

En esta sección se presentan las pruebas usadas para comprobar la correctitud de la herramienta desarrollada. Para lograr este objetivo se sigue un enfoque basado en *tests*. Como se menciona en los objetivos (Sección 1.4.1), para aceptar la validez del desarrollo, los resultados calculados manualmente deben coincidir con los resultados automatizados retornados por la herramienta desarrollada. Además de validar la herramienta, algunos *tests* introducen ideas para la síntesis de invariantes, una de estas ideas es la iteración de punto fijo del Teorema de Kleene.

5.1. Diseño de la evaluación

El objetivo general de este trabajo es automatizar los cálculos generados por la transformada $\mathsf{ert}[\cdot]$. Para verificar que la herramienta cumpla con este objetivo, es necesario analizar su correctitud. Se entiende por correctitud como la capacidad de reproducir los cálculos asociados a la función $\mathsf{vcg}[\cdot]$, tanto en el tiempo de ejecución como en las obligaciones de prueba, como en la validación de las mismas.

Para evaluar esta propiedad se procede con un enfoque basado en *tests*. Cada *test* es una comparación entre los cálculos manuales y los cálculos automatizados entregados por la herramienta. Es relevante mencionar que esta comparación es hecha en un sentido *matemático*, es decir, no es relevante la representación mientras se refiera al mismo objeto. A modo de ejemplo, para los efectos de esta investigación, la expresión $-1 + 2 \cdot x \leq 0$ es equivalente a $\neg(2 \cdot x - 1 > 0)$.

Con respecto al flujo que sigue cada *test*, el *input* que se entrega es un *string* que representa al programa probabilístico. La herramienta convierte este *string* a un elemento perteneciente a `Program`. Una vez obtenido este elemento, se procede a aplicar la función $\mathsf{vcg}[\cdot]$ e imprimir la información relevante para el análisis.

Para mejorar el cubrimiento de los *tests* hay que considerar 3 categorías de programas: sin ciclos, con ciclos e invariantes correctos y con ciclos e invariantes incorrectos. Además de lo anterior, hay que examinar la información relevante en cada uno de los casos. Para

un programa sin ciclos, el análisis se concentra solo en obtener la predicción del tiempo de ejecución. Si el programa contiene ciclos, la situación se complejiza. Si los invariantes son correctos, entonces el tiempo de ejecución calculado es válido, pero si no, el tiempo de ejecución obtenido es inválido. Si un analista se encuentra en esta última situación, le será relevante toda información relacionada a los invariantes incorrectos, por ejemplo, el problema lineal y su contraejemplo asociado.

Con estas consideraciones en mente, se platean los siguientes criterios de aprobación.

Criterios de aprobación para un programa sin ciclos

1. La herramienta retorna un tiempo de ejecución equivalente al calculado manualmente.
2. La herramienta no retorna obligaciones de prueba.

Criterios de aprobación para programa con ciclos e invariantes correctos

1. La herramienta retorna un tiempo de ejecución equivalente al calculado manualmente.
2. La herramienta retorna un conjunto de obligaciones de prueba equivalente al calculado manualmente.
3. La herramienta valida todas las obligaciones de prueba.

Criterios para programas con ciclos e invariantes incorrectos

1. La herramienta retorna un tiempo de ejecución equivalente al calculado manualmente.
2. La herramienta retorna un conjunto de obligaciones de prueba equivalente al calculado manualmente.
3. La herramienta valida invalida las obligaciones de prueba, coincidiendo con los cálculos manuales.
4. En el caso que una obligación sea inválida, la herramienta retorna un contraejemplo junto a su contexto y restricción derivada correspondiente.

Con el establecimiento de estos criterios que bien definida la validación de las 3 etapas del desarrollo: el *parsing*, cálculo de los tiempos de ejecución y verificación de las obligaciones de prueba.

5.2. Diseño de los *tests*

A pesar de que la categorización presentada anteriormente es suficiente para establecer el criterio de la evaluación, esta es poco fina. Para establecer la correctitud de la herramienta,

es necesario someterla a una amplia variedad de programas, cosa que no está garantizada con las 3 categorías anteriores. Con el fin de abordar todos los posibles tipos de programas se dividieron estos en diferentes categorías y subcategorías, para luego diseñar un *test* que represente a cada subcategoría. Las principales categorías, como es natural, son los programas deterministas y los probabilistas. Como se detalló en capítulos anteriores, el comportamiento de un programa varía considerablemente dependiendo si es probabilístico o no, debido a esto, esta es la principal división.

Las siguientes categorías son los programas con ciclo y sin ciclo. El hecho de tener uno o más ciclos induce al uso de invariantes y, por lo tanto, provoca un análisis más trabajoso que un programa sin ciclos. A su vez, a los programas con ciclos se le puede dividir entre los programas con invariantes correctos e invariantes incorrectos. La razón para introducir esta división es la presentada anteriormente, el *output* retornado es más detallado en los programas con invariantes incorrectos (se retorna un contraejemplo e información asociada), que en los programas con invariantes correctos.

Por último, se encuentra la división entre programas con tiempo de ejecución variable (dependiente del estado inicial del programa) y tiempo de ejecución constante (no dependiente del estado). Para los programas sin ciclos, el análisis no cambia sustancialmente. Para los programas deterministas con ciclos `while`, entonces el tiempo de ejecución constante se logra a través de ciclos con un número estático de iteraciones. En cambio, para un ciclo `pwhile`, la condición anterior no es posible. Dada su naturaleza probabilista, existen múltiples resultados ante su ejecución; sin embargo, si la probabilidad de éxito (`p`) para ejecutar el cuerpo del ciclo es menor a 1, el número *esperado* de iteraciones si es constante y finito. Lo anterior se tiene debido a que el número de iteraciones que ejecuta en un ciclo `pwhile` puede ser modelado por un variable aleatoria geométrica. De esta manera, el número esperado de iteraciones de un ciclo `pwhile` es $\frac{1}{1-\texttt{p}}$. Esta situación permite obtener tiempos de ejecución constantes a través de invariantes constantes, cosa no habitual en los programas deterministas con ciclos `while`. Esta diferencia de comportamientos es la principal razón para introducir estas categorías.

A continuación, se presentan las 12 subcategorías usadas en la validación:

- Programas deterministas
  - Con tiempo de ejecución constante y sin ciclo `while`
  - Con tiempo de ejecución variable y sin ciclo `while`
  - Con tiempo de ejecución constante, con ciclo `while` e invariante correcto.
  - Con tiempo de ejecución constante, con ciclo `while` e invariante incorrecto.
  - Con tiempo de ejecución variable, con ciclo `while` e invariante correcto.
  - Con tiempo de ejecución variable, con ciclo `while` e invariante incorrecto.
- Programas probabilísticos
  - Con tiempo de ejecución constante y sin ciclo `pwhile`.
  - Con tiempo de ejecución variable y sin ciclo `pwhile`.
  - Con tiempo de ejecución constante, con `pwhile` e invariante correcto.

- Con tiempo de ejecución constante, con `pwhile` e invariante incorrecto.
- Con tiempo de ejecución variable, con `pwhile` e invariante correcto.
- Con tiempo de ejecución variable, con `pwhile` e invariante incorrecto.

Como se ha mencionado anteriormente, para cada una de las 12 subcategorías anteriores se diseñó un *test* que la represente. La razón para acotar la validación a este número es el trabajo necesario para completar un *test*. En primer lugar, se necesita calcular manualmente la función $\mathsf{vcg}[\cdot]$. Como se ha discutido anteriormente, este tipo de cálculo es tedioso y sobre todo tendiente al error. Además de los anterior, para lo programas con ciclos, es necesario validar sus obligaciones de prueba. Recordemos que si el conjunto atómico de una obligación de prueba tiene tamaño n, entonces existen 2^n problemas lineales asociados, los que deben ser analizados para validar una obligación de prueba. Por último, es necesario mencionar que en algunos programas se exploraron técnicas para la síntesis de invariantes, técnicas que también requieren una cuota de trabajo manual.

Complementando las subcategorías anteriores, se buscó generar un conjunto de programas que aborden la mayor cantidad posible de constricciones del lenguaje y formas de combinación. Por ejemplo, en estas pruebas se ocuparon todos los constructores de las expresiones `AExp`, `BExp`, `RunTime` y `Program`. Además, se incluyeron programas que contienen una composición secuencial de ciclos y otros que contienen una anidación de ciclos. Se agregaron estos casos, ya que es provechoso estudiar cómo se comportan las obligaciones de prueba cuando existe más de un ciclo en un programa.

5.3. Consideraciones previas

Notación para representar a los programas. Para presentar las figuras de manera cómoda, se introduce una notación para referirse a cada uno de los programas estudiados. Para representar a un programa se usará la notación $\mathtt{C}_{\mathtt{wxyz}}$. El primer subíndice `w` señala si el programa es determinista o probabilista. Se usa $\mathtt{w} = \mathtt{d}$ si el programa es determinista y $\mathtt{w} = \mathtt{p}$ si es probabilista. El segundo subíndice `x` hace referencia al tiempo de ejecución del programa. Se usa $\mathtt{x} = \mathtt{k}$ si el tiempo de ejecución es constante y $\mathtt{x} = \mathtt{v}$ si el tiempo de ejecución es variable. El tercer índice `y` hace referencia a los ciclos del programa. Si el programa no contiene ciclos se usa $\mathtt{y} = \mathtt{s}$ y si contiene ciclos se usa $\mathtt{y} = \mathtt{c}$. Por último, el cuarto subíndice `z` hace referencia a la correctitud de los invariantes del programa. Si los invariantes asociados son correctos se ocupa $\mathtt{z} = +$ y en caso contrario se usa $\mathtt{z} = -$. En el caso de un programa sin ciclos se omite el subíndice `z`. Nótese que lo se definió es una notación para denotar (sub)categorías, pero al sólo haber un programa por subcategoría, no hay espacio para la ambigüedad.

Usando la notación anterior, un programa determinista, con tiempo de ejecución constante y sin ciclos se denota como $\mathtt{C}_{\mathtt{dks}}$. Un programa probabilista, tiempo de ejecución variable, con ciclos e invariantes correctos se denota como $\mathtt{C}_{\mathtt{pvs}+}$. Además de referirse a programas usados en los *tests*, se puede usar la notación presentada para denotar a categorías que contengan más de un elemento. Para hacer esto, se omiten los índices que no son de interés, por ejemplo,

$\lvert \mathtt{C_d} \rvert$	$\lvert \mathtt{C_p} \rvert$	$\lvert \mathtt{C_k} \rvert$	$\lvert \mathtt{C_v} \rvert$	$\lvert \mathtt{C_s} \rvert$	$\lvert \mathtt{C_c} \rvert$	$\lvert \mathtt{C}_+ \rvert$	$\lvert \mathtt{C}_- \rvert$
6	6	6	6	4	8	4	4

Figura 5.1: Resumen de la cantidad de programas utilizados en los *tests*

para representar a los programas probabilísticos se usa $\mathtt{C_p}$, y los programas con tiempo de ejecución constante y ciclos se representan como $\mathtt{C_{kc}}$.

Notación para representar los cálculos de la función vcg[·] sobre un programa. Para detallar de manera óptima los cálculos manuales de la función vcg[·], se introduce una notación que representa el modo en que esta se va propagando a través de un programa. Sea t y $f \in \mathtt{RunTime}$, o y $s \in \mathbb{O}$ y $\mathtt{C} \in \mathtt{Program}$. Si $\mathtt{C}$ es un programa que inicia en la línea N_i con la instrucción $\mathtt{C_i}$ y termina en la línea N_j con la instrucción $\mathtt{C_j}$, y $(f,\ o)$ es una simplificación de $\mathsf{vcg}[\mathtt{C}](t)$. Lo anterior se puede representar como:

$$
\begin{array}{ll}
\diamond & (f,\ o) \\
\diamond & \mathsf{vcg}[\mathtt{C}](t) \\
N_i & \mathtt{C_i} \\
\vdots & \vdots \\
N_j & \mathtt{C_j} \\
\diamond & (t,\ s)
\end{array}
\quad .
$$

En el caso de programas sin ciclos la notación se simplifica, ya que no habrá obligaciones de prueba asociadas, es decir, los cálculos vuelven a representar a la transformada ert[·]. A continuación, un ejemplo que utiliza las variables presentadas en el ejemplo anterior.

Cálculo de la transformada del programa $\mathtt{C}$ con respecto al tiempo de ejecución t

$$
\begin{array}{ll}
\diamond & f \\
\diamond & \mathsf{ert}[\mathtt{C}](t) \\
N_i & \mathtt{C_i} \\
\vdots & \vdots \\
N_j & \mathtt{C_j} \\
\diamond & t
\end{array}
\quad .
$$

Con respecto a los cálculos hechos por la herramienta desarrollada, estos, a diferencia del ejemplo de *output* en la introducción (Figura 1.2), no son presentados mediante una imagen, sino que con un *listing* de LaTeX. Para adaptar de mejor manera el texto del *output* de la herramienta a su figura correspondiente, se hicieron algunas modificaciones en el espaciado, sin embargo, el contenido del texto no fue alterado.

Para finalizar la sección se presenta la Figura 5.1, donde se detalla un resumen de la cantidad de programas utilizados en cada categoría.

5.4. *Tests* utilizados

A continuación, se presentan 3 de los 12 *tests* utilizados en la validación. Se prefirió presentar una selección de *tests* para evitar extender innecesariamente este capítulo, ya que muchos de los cálculos y procedimientos se repiten constantemente a lo largo de este proceso. Para obtener visualizar el resto de *tests* ver Apéndice C.1.

Los programas presentados en esta selección responden a la categorías introducidas en los criterios de evaluación. De esta manera, es posible presentar los diferentes tipos de *output* que puede retornar el trabajo desarrollado.

5.4.1. Programa determinista, con tiempo de ejecución constante y sin ciclo ($\texttt{C}_{\texttt{dks}}$)

```
1  x := 10;
2  y := 3;
3  if (x >= y)
4       { skip;
5        skip }
6  else {
7       if (true || (x == 0)){
8            z := 3/5;
9            w := 3 }
10      else {
11           skip;
12           empty }}
```

Figura 5.2: Programa $\texttt{C}_{\texttt{dks}}$

El primer programa a analizar se encuentra en la Figura 5.2. Inicia fijando los valores de las variables `x` e `y`. Luego evalúa un condicional `if`, donde la condición de la guarda es `x >= y`. Si la condición es verdadera, ejecuta la composición secuencial de dos programas `skip`, finalizando la ejecución. En el caso que la condición sea falsa, se ejecuta un nuevo condicional `if`. En este caso la condición es equivalente a la constante `true`, por ende, siempre se ejecutará la rama `true`. En esta rama se fijan los valoras de las variables `x` y `w` y luego termina el programa.

◇ 5
◇ $1+1+1+[\![10 >= 3]\!]\cdot 2+[\![\neg 10 >= 3]\!]\cdot 3$
1 x := 10;
◇ $1+1+[\![x >= 3]\!]\cdot 2+[\![\neg x >= 3]\!]\cdot 3$
2 y := 3;
◇ $1+[\![x >= y]\!]\cdot 2+[\![\neg x >= y]\!]\cdot 3$
3 if (x >= y)
◇ 2
◇ $1+1+0$
{ skip;
◇ $1+0$
5 skip }
◇ 0
6 else {
◇ 3
◇ $1+[\![\texttt{true} \mid\mid (\texttt{x} == 0)]\!]\cdot 2+[\![\neg(\texttt{true} \mid\mid (\texttt{x} == 0))]\!]\cdot 1$
7 if (true || (x == 0)){
◇ 2
◇ $1+1+0$
8 z := 3/5;
◇ $1+0$
9 w := 3 }
◇ 0
10 else {
◇ 1
◇ $1+0$
11 skip;
◇ 0
12 empty }}
◇ 0
◇ 0

Figura 5.3: Cálculo ert[·] manual - Programa $\texttt{C}_{\texttt{dks}}$

```
Programa Analizado:
x:=10; y:=3; if(x>=y){skip; skip}
             else{if(true || (x == 0)){z:=3/5; w:=3}else{skip; empty}}

Tiempo de ejecución calculado:
5

El tiempo de ejecución calculado es válido
porque no hay obligaciones de prueba asociadas,
ya que el programa no contiene ciclos.

Análisis Finalizado.
```

Listing 5.1: Cálculo ert[·] automático - Programa $\texttt{C}_{\texttt{dks}}$

Comparación entre los cálculos manuales y la herramienta desarrollada La herramienta desarrollada logra representar de manera óptima los resultados manuales. Por lo tanto, el *test* se considera aprobado.

5.4.2. Programa determinista, tiempo de ejecución constante, con ciclo e invariante incorrecto ($\texttt{C}_{\texttt{dkc}-}$)

En este *test* se retoma el ejemplo de la Figura 4.2. Para este ciclo se proponen dos invariantes, el primer invariante es $1+2\cdot[\![x>0]\!]\cdot x$. La intuición tras el invariante es simple, el ciclo en cada iteración consume dos unidades de tiempo (una por la asignación y otra por la evaluación de la guarda) y hará x iteraciones si $x>0$. Por último, se añade una unidad por la evaluación de la guarda cuando esta sea falsa.

A pesar de que la intuición presentada es coherente, el invariante es incorrecto. Este sería correcto si x fuese entero, pero en esta investigación se trabaja con números racionales. Para números racionales el invariante correcto es $1+2\cdot[\![x>0]\!]\cdot\lceil x\rceil$, pero, al sólo poder representar expresiones lineales, se usa $x+1$ como una aproximación de $\lceil x\rceil$. Con esa consideración, el invariante correcto es $1+2\cdot[\![x>0]\!]\cdot(x+1)$ y es analizado en el Apéndice C.1.2.

```
1  x ≔ 3;
2  while (x > 0)
3        { inv = 1 ++ 2 ** [x > 0] <> x}
4        { x ≔ x − 1 }
```

Figura 5.4: Programa determinista $\texttt{C}_{\texttt{dkc}-}$

```
◇  (8, {[[x > 0]] · (1 + [[x − 1 > 0]] · (x − 1)) ≤ [[x > 0]] · x})
◇  (1 + 1 + 2 · [[3 > 0]] · 3, {[[x > 0]] · (1 + [[x − 1 > 0]] · (x − 1)) ≤ [[x > 0]] · x})
1  x ≔ 3;
◇  (1 + 2 · [[x > 0]] · x, {[[x > 0]] · (1 + [[x − 1 > 0]] · (x − 1)) ≤ [[x > 0]] · x})
◇  (1 + 2 · [[x > 0]] · x, {1 + [[x > 0]] · (2 + 2 · [[x − 1 > 0]] · (x − 1)) ≤ 1 + 2 · [[x > 0]] · x})
2  while (x > 0)
3        { inv = 1 ++ 2 ** [x > 0] <> x}
◇         (2 + 2 · [[x − 1 > 0]] · (x − 1), ∅)
◇         (1 + 1 + 2 · [[x − 1 > 0]] · (x − 1), ∅)
4        { x ≔ x − 1 }
◇         (1 + 2 · [[x > 0]] · x, ∅)
◇  (0, ∅)
```

Figura 5.5: Cálculo vcg[·] manual - Programa $\texttt{C}_{\texttt{dkc}-}$

El programa genera una obligación de prueba. Para comprobar su validez se ocupará el procedimiento descrito en el capítulo anterior. El conjunto atómico es

$$\mathcal{A}_{[\![x>0]\!]\cdot(1+[\![x-1>0]\!]\cdot(x-1))\leq[\![x>0]\!]\cdot x}=\{(x>0),(x-1>0)\}$$

En total se tienen 4 posibles contextos y a continuación se analiza cada uno.

1.

Contexto	$= (x > 0) \land (x - 1 > 0)$
Restricción derivada	$= 1 \cdot (1 + 1 \cdot (x - 1)) \leq 1 \cdot x$
	$= 1 + x - 1 \leq x$
	$= 0 \leq 0$
Problema Lineal	$= (x > 0) \land (x - 1 > 0) \land \neg(0 \leq 0)$
	$= (x > 0) \land (x - 1 > 0) \land (0 > 0)$
	$= (x > 0) \land (x - 1 > 0) \land \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe un modelo válido.

2.

Contexto	$= \neg(x > 0) \land (x - 1 > 0)$
Restricción derivada	$= 0 \cdot (1 + 1 \cdot (x - 1)) \leq 0 \cdot x$
	$= 0 \leq 0$
Problema Lineal	$= \neg(x > 0) \land (x - 1 > 0) \land \neg(0 \leq 0)$
	$= \neg(x > 0) \land (x - 1 > 0) \land (0 > 0)$
	$= \neg(x > 0) \land (x - 1 > 0) \land \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe un modelo válido.

3.

Contexto	$= (x > 0) \land \neg(x - 1 > 0)$
Restricción derivada	$= 1 \cdot (1 + 0 \cdot (x - 1)) \leq 1 \cdot x$
	$= 1 \leq x$
Problema Lineal	$= (x > 0) \land \neg(x - 1 > 0) \land \neg(1 \leq x)$
	$= (x > 0) \land \neg(x - 1 > 0) \land (x < 1)$
	$= (x > 0) \land (x \leq 1) \land (x < 1)$
	$= (x > 0) \land (x < 1)$
Modelo	= El problema es satisfacible, $x = \frac{1}{2}$ es un modelo válido

---

4.

Contexto	$= \neg(x > 0) \land \neg(x - 1 > 0)$
Restricción derivada	$= 0 \cdot (1 + 0 \cdot (x - 1)) \leq 0 \cdot x$
	$= 0 \leq 0$
Problema Lineal	$= \neg(x > 0) \land \neg(x - 1 > 0) \land \neg(0 \leq 0)$
	$= (x \leq 0) \land (x - 1 \leq 0) \land (0 > 0)$
	$= (x \leq 0) \land (x \leq 1) \land \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe un modelo válido.

Conclusión El invariante $1 + 2 \cdot [\![x > 0]\!] \cdot x$ no es correcto, ya que la obligación de prueba asociada no es válida. Un contraejemplo es $x = \frac{1}{2}$.

$$
\begin{array}{lll}
 & [\![\frac{1}{2} > 0]\!] \cdot (1 + [\![\frac{1}{2} - 1 > 0]\!] \cdot (\frac{1}{2} - 1)) & \leq [\![\frac{1}{2} > 0]\!] \cdot \frac{1}{2} \\
= & 1 \cdot (1 + 0 \cdot (\frac{1}{2} - 1)) & \leq 1 \cdot \frac{1}{2} \\
= & 1 & \leq \frac{1}{2} \\
= & \texttt{false} &
\end{array}
$$

```
  Programa Analizado:
  x:=3 ;while(x > 0){inv = 1 ++ 2**[x>0]<>x }{x:= x-1}

  Tiempo de ejecución calculado:
  8

  Obligaciones de prueba asociadas:
  [1]  1 ++ [!(x <= 0)]<>(2 ++ 2**([!(-1 + x <= 0)]<>(-1 + x)))
       :!<=:
       1 ++ 2**([!(x <= 0)]<>x),
       No es válida

**********************************************************************
  Obligación de prueba [1]
  1 ++ [!(x <= 0)]<>(2 ++ 2**([!(-1 + x <= 0)]<>(-1 + x)))
  :!<=:
  1 ++ 2**([!(x <= 0)]<>x)

  La obligación de prueba tiene asociada 4 restricciones derivadas
   diferentes.

----------------------------------------------------------------------

  Restricción derivada  [1, 2]
  [!(x <= 0),-1 + x <= 0]   |-  3 :!<=: 1 + 2*x

  La restricción no es válida.

  Un contraejemplo encontrado es:
  x = 1/2 Racional
----------------------------------------------------------------------

  El tiempo de ejecución calculado no es válido
  porque alguna obligación de prueba no es válida.
  Ajuste los invariantes de ciclo y vuelva a realizar el análisis.

  Análisis Finalizado.
```

Listing 5.2: Cálculo vcg[·] automático - Programa $\mathtt{C}_{\mathtt{dkc}-}$

Comparación entre los cálculos manuales y la herramienta desarrollada. Los cálculos automatizados logran representar los cálculos manuales. Existen diferencias en la representación de algunas estructuras, por ejemplo, la expresión $!(-1 + \mathtt{x} <= 0)$ y su equivalente $x - 1 > 0$. Además se agrega que la simplificación de la obligación de prueba no es óptima. Pese a estas consideraciones, el *output* informa claramente cuál es el contraejemplo que demuestra que el invariante es incorrecto, junto a su información asociada y cual es el tiempo de ejecución, por esta razón el *test* se considera aprobado.

5.4.3. Programa probabilístico, tiempo de ejecución constante, con ciclo e invariante correcto ($\mathtt{C}_{\mathtt{pkc+}}$)

```
1 pwhile (< 1/2 >)
2       { pinv = I1}
3       { skip };
4 pwhile (< 1/2 >)
5       { pinv = I2}
6       { skip };
7 pwhile (< 1/2 >)
8       { pinv = I3}
9       { skip }
```

Figura 5.6: Programa $\mathtt{C}_{\mathtt{pkc+}}$

El programa analizado es una composición secuencial de ciclos `pwhile`, cuyo número esperado de iteraciones es constante. A diferencia de lo hecho hasta ahora, se proponen métodos para sintetizar los invariantes. Como punto de partida es necesario iniciar estudiando sólo uno de los ciclos y una vez obtenido el invariante para ese ciclo, encontrar los otros invariantes.

Para encontrar el invariante de un ciclo primero, y como es de costumbre, se inicia recurriendo a la intuición. El número esperado de iteraciones es 2, por cada iteración se consumen dos unidades de tiempo, una por la evaluación de la guarda y otra por el `skip`. En total, el tiempo de ejecución esperado es

$$1 + 2 \cdot 2 = \mathbf{5}$$

unidades de tiempo. Se le suma una unidad por la evaluación de la guarda cuando esta es falsa. Este invariante es correcto, pero pude ser disminuido.

Para encontrar el invariante más ajustado se necesita del teorema del punto fijo de Kleene. Este teorema plantea que partiendo del elemento *bottom* (en este caso $\mathbf{0}$) y mediante la aplicación consecutiva de una función característica se llega al menor punto fijo de la misma. En este caso la función característica del ciclo `pwhile` con respecto a $\mathbf{0}$ es:

$$F_3(\mathtt{I}_3) = 1 + \frac{1}{2} \cdot \mathbf{0} + \mathsf{ert}[\mathtt{skip}](\mathtt{I}_3) = 1 + \frac{1}{2} \cdot (1 + \mathtt{I}_3)$$

Las primeras 10 iteraciones de esté método son :

1. $\mathtt{I}_3^1 = F_3(0) = \frac{3}{2}$
2. $\mathtt{I}_3^2 = F_3(\frac{3}{2}) = \frac{9}{4}$
3. $\mathtt{I}_3^3 = F_3(\frac{9}{4}) = \frac{21}{8}$
4. $\mathtt{I}_3^4 = F_3(\frac{21}{8}) = \frac{45}{16}$
5. $\mathtt{I}_3^5 = F_3(\frac{45}{16}) = \frac{93}{32}$
6. $\mathtt{I}_3^6 = F_3(\frac{93}{32}) = \frac{189}{64}$
7. $\mathtt{I}_3^7 = F_3(\frac{189}{64}) = \frac{381}{128}$
8. $\mathtt{I}_3^8 = F_3(\frac{381}{128}) = \frac{765}{256}$
9. $\mathtt{I}_3^9 = F_3(\frac{765}{256}) = \frac{1533}{512}$
10. $\mathtt{I}_3^{10} = F_3(\frac{1533}{512}) = \frac{3069}{1024}$

Figura 5.7: Primeras 10 iteraciones de punto fijo sobre la función F_3

Mediante exploración se puede concluir que la fórmula que modela la sucesión anterior es $\mathtt{I}_3^{\mathtt{k}} = \frac{3\cdot(2^{\mathtt{k}}-1)}{2^{\mathtt{k}}}$.

Demostración. Para el caso base $\mathtt{k} = 0$: $\mathtt{I}_3^0 = \frac{3\cdot(2^0-1)}{2^0} = 0$.
Para el caso inductivo $\mathtt{k} > 0$:

$$\begin{aligned} F(\mathtt{I}_3^{\mathtt{k}}) &= 1 + \frac{1}{2}\cdot\left(1 + \frac{3\cdot(2^k-1)}{2^k}\right) \\ &= 1 + \frac{2^k + 3\cdot 2^k - 3}{2^{k+1}} \\ &= 1 + \frac{2\cdot 2^{k+1} - 3}{2^{k+1}} \\ &= \frac{2^{k+1} + 2\cdot 2^{k+1} - 3}{2^{k+1}} \\ &= \frac{3\cdot(2^{k+1}-1)}{2^{k+1}} \\ &= \mathtt{I}_3^{\mathtt{k+1}} \end{aligned} \tag{5.1}$$

□

Para encontrar el invariante se necesita obtener el valor en el límite, $\lim_{\mathtt{k}\to\infty} \mathtt{I}_3^{\mathtt{k}}$. Al hacer este cálculo se concluye que

$$\mathtt{I}_3 = \mathbf{3}$$

Una vez resuelto el subproblema del invariante de un sólo ciclo `pwhile`, es posible abordar el problema con la composición secuencial de 3 ciclos. Primero, es natural suponer que cada

uno de los ciclos consume **3** unidades de tiempo y, por ende, el programa completo consume **9** unidades de tiempo. Este pensamiento es correcto, pero falta una observación para proponer los invariantes $\mathtt{I}_1$, $\mathtt{I}_2$ y $\mathtt{I}_3$.

Hasta el momento, las obligaciones de prueba en los ejemplos y *tests* han sido planteadas con respecto al tiempo de ejecución constante **0**. Con esto se está asumiendo que al finalizar el ciclo se termina la ejecución del programa y no se consumen más unidades de tiempo. Esta asunción es correcta, ya que esos ciclos estaban al final del código de su respectivo programa.

En el presente caso solo el último ciclo está en esa situación. Para los invariantes de los demás ciclos se debe considerar el tiempo de ejecución que sigue a su ejecución. De esta forma el invariante del último ciclo es $\mathtt{I}_3 = \mathbf{3}$, el invariante del segundo ciclo con respecto a la continuación **3** es $\mathtt{I}_2 = \mathtt{3} + \mathtt{3} = \mathbf{6}$ y el invariante del primer ciclo con respecto a la continuación **6** es $\mathtt{I}_1 = \mathtt{3} + \mathtt{6} = \mathbf{9}$.

A la misma conclusión se llega si se procede mediante la iteración de punto fijo sobre las funciones características. Para $\mathtt{I}_3$ el análisis es el hecho previamente. Para $\mathtt{I}_2$ su función característica es $F_2(\mathtt{I}_2) = \frac{5}{2} + \frac{1}{2} \cdot (\mathtt{1} + \mathtt{I}_2)$, y el término que modela la iteración de punto fijo es $\frac{3 \cdot (2^n - 1)}{2^{n-1}}$ (demostración en Apéndice C.1.9) cuyo límite es **6**. Para $\mathtt{I}_3$ su función característica es $F_3(\mathtt{I}_1) = \frac{5}{2} + \frac{1}{2} \cdot (\mathtt{1} + \mathtt{I}_1)$, y el término que modela la iteración de punto fijo es $\frac{9 \cdot 2^{n-1}}{2^n}$ (demostración en Apéndice C.1.9) cuyo límite es **9**.

A modo de conclusión, los invariantes $\mathtt{I}_3 = \mathbf{3}$, $\mathtt{I}_2 = \mathbf{6}$ y $\mathtt{I}_1 = \mathbf{9}$ son correctos y además, los más ajustados posibles. Una versión con los invariantes incorrectos de este programa es estudiada en el Apéndice C.1.7.

◇ $(9,\ \{3 \leq 3, 6 \leq 6, 9 \leq 9\})$
◇ $(9,\ \{3 \leq 3, 6 \leq 6, 1 + \frac{1}{2} \cdot 6 + \frac{1}{2} \cdot 10 \leq 9\})$
1 `pwhile` $(< 1/2 >)$
2 $\{\ \texttt{pinv} = 9\}$
◇ $(10,\ \{3 \leq 3, 6 \leq 6\})$
◇ $(9 + 1,\ \{3 \leq 3, 6 \leq 6\})$
3 $\{\ \texttt{skip}\ \};$
◇ $(9,\ \{3 \leq 3, 6 \leq 6\})$
◇ $(6,\ \{3 \leq 3, 6 \leq 6\})$
◇ $(6,\ \{3 \leq 3, 1 + \frac{1}{2} \cdot 3 + \frac{1}{2} \cdot 7 \leq 6\})$
4 `pwhile` $(< 1/2 >)$
5 $\{\ \texttt{pinv} = 6\}$
◇ $(7,\ \{3 \leq 3\})$
◇ $(6 + 1,\ \{3 \leq 3\})$
6 $\{\ \texttt{skip}\ \};$
◇ $(6,\ 3 \leq 3)$
◇ $(3,\ \{3 \leq 3\})$
◇ $(3,\ \{1 + \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 4 \leq 3\})$
7 `pwhile` $(< 1/2 >)$
8 $\{\ \texttt{pinv} = 3\}$
◇ $(4,\ \emptyset)$
◇ $(3 + 1,\ \emptyset)$
9 $\{\ \texttt{skip}\ \}$
◇ $(3,\ \emptyset)$
◇ $(0,\ \emptyset)$

Figura 5.8: Cálculo vcg[·] manual - Programa $\mathtt{C}_{\mathtt{pkc+}}$

Es claro que las obligaciones son válidas, por lo que se procede a presentar el cálculo automatizado.

```
Programa Analizado:
pwhile(<1/2>){pinv = 9}{skip};
pwhile(<1/2>){pinv = 6}{skip};
pwhile(<1/2>){pinv = 3}{skip}

Tiempo de ejecución calculado:
9

Obligaciones de prueba asociadas:
[1]  9 :!<=: 9, Es válida
[2]  6 :!<=: 6, Es válida
[3]  3 :!<=: 3, Es válida

El tiempo de ejecución calculado es válido
porque las obligaciones de prueba son válidas.

Análisis Finalizado.
```

Listing 5.3: Cálculo vcg[·] automático - Programa $\mathtt{C}_{\mathtt{pkc+}}$

$\mathtt{C}$	`RunTime` automático
$\mathtt{C}_{\mathtt{dks}}$	5
$\mathtt{C}_{\mathtt{dvs}}$	$2 \mathbin{+\!\!+} [\mathtt{y} <= -1 + \mathtt{x}] \diamond 2 \mathbin{+\!\!+} [!(\mathtt{y} <= -1 + \mathtt{x})] \diamond (1 \mathbin{+\!\!+} [8 <= \mathtt{w}] \diamond 2 \mathbin{+\!\!+} [!(8 <= \mathtt{w})])$
$\mathtt{C}_{\mathtt{dkc}+}$	10
$\mathtt{C}_{\mathtt{dkc}-}$	8
$\mathtt{C}_{\mathtt{dvc}+}$	$1 \mathbin{+\!\!+} 2 * *([\mathtt{y} <= \mathtt{x} \;\&\&\; \mathtt{x} <= \mathtt{z}] \diamond (2 + -2 * \mathtt{x} + 2 * \mathtt{z}))$
$\mathtt{C}_{\mathtt{dvc}-}$	$[0 <= \mathtt{x}] \diamond \mathtt{x}$
$\mathtt{C}_{\mathtt{pks}}$	$5/2$
$\mathtt{C}_{\mathtt{pvs}}$	$1 \mathbin{+\!\!+} 9/10 * * (1 \mathbin{+\!\!+} [!(x <= 10)] \mathbin{+\!\!+} [x <= 10] \diamond 2)$
$\mathtt{C}_{\mathtt{pkc}+}$	9
$\mathtt{C}_{\mathtt{pkc}-}$	3
$\mathtt{C}_{\mathtt{pkc}+}$	$10 \mathbin{+\!\!+} 9 * * ([!(\mathtt{c} == 1)]) \mathbin{+\!\!+} 207 * * ([\mathtt{c} == 1])$
$\mathtt{C}_{\mathtt{pvc}-}$	$19/10 \mathbin{+\!\!+} 9/10 * *([!(\mathtt{c} == 1)]) \mathbin{+\!\!+} 207/10 * * ([\mathtt{c} == 1])$

Tabla 5.1: Tiempos de ejecución automáticos obtenidos

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta calcula las obligaciones de prueba de manera óptima. Es capaz de simplificar las expresiones hasta tener una desigualdad entre constantes y además retorna el tiempo de ejecución correcto. El *test* se considera aprobado.

5.5. Resultados

Cada *test* usado en la validación cumplió con los criterios de aceptación y fue aprobado. Por esta razón, es posible afirmar que la herramienta desarrollada logra cumplir el objetivo general planteado en la Sección 1.4.1. A continuación, se discute las principales funcionalidades de la herramienta desarrollada y los resultados obtenidos con esa funcionalidad. Es conveniente mencionar que estas funcionalidades responden de manera directa a los objetivos específicos planteados al inicio de este trabajo.

***Parser* desarrollado.** El *parser* desarrollado se mostró suficiente para cumplir con las necesidades del proyecto. Se destaca la posibilidad de escribir los invariantes sin la necesidad de paréntesis excesivos y el uso de algunos *shorcuts* para definir programas. Por ejemplo, en el Apéndice C.1.7 se hace uso de un ciclo `for`. Dada a esta situación, es posible afirmar que se cumplió el objetivo específico asociado a la desarrollo de un *parser*.

Tiempos de ejecución obtenidos. En la Tabla 5.1 se presentan los tiempos de ejecución obtenidos de manera automática, los que son equivalentes a los realizados de forma manual. Es interesante notar la cantidad de indicatrices obtenidas. Esto se debe a que cada vez que se trabaje con un guarda, esta generará una indicatriz en un tiempo de ejecución. Al obtener tiempos de ejecución coincidentes con los cálculos de la transformada $\mathsf{ert}[\cdot]$, es posible afirmar que se cumplió el objetivo específico asociado a esa tarea.

C	Contexto	Restricción derivada	Contraejemplo
$\texttt{C}_{\texttt{dkc}-}$	$[!(\texttt{x} <= 0), -1 + \texttt{x} <= 0]$	$3 :! <=: 1 + 2 * \texttt{x}$	$1/2$
$\texttt{C}_{\texttt{dvc}-}$	$[!(0 <= \texttt{x})]$	$1 :! <=: 0$	-1
$\texttt{C}_{\texttt{dvc}-}$	$[0 <= \texttt{x}]$	$1 :! <=: \texttt{x}$	0
$\texttt{C}_{\texttt{pkc}-}$	$[\ ]$	$9/2 :! <=: 3$	$\emptyset$
$\texttt{C}_{\texttt{pvc}-}$	$[!(\texttt{c} == 1)]$	$19/5 :! <=: 2$	$19/2$

Tabla 5.2: Contextos, restricciones derivadas y contraejemplos encontrados.

Obligaciones de prueba obtenidas. Las obligaciones de prueba obtenidas, al igual que los tiempos de ejecución, son coincidentes con sus homólogas calculadas a mano. Lo más destacable es la variedad de obligaciones obtenidas, las cuales varían desde desigualdades entre constantes (ver Figura 5.8), hasta desigualdades entre funciones complejas (ver Figura 5.5). Un punto a mejorar es la simplificación. La herramienta no logra llevar la desigualdades de las forma $f \leq f$ a $0 \leq 0$ (ver *Listing* C.9). Pese a lo anterior, la herramienta calcula las obligaciones correctamente, por lo que se puede afirmar que el objetivo especifico asociado a este punto fue logrado.

Validación de las obligaciones. La herramienta logra validar correctamente cada obligación de prueba. En el caso que alguna obligación no sea válida, se retorna un contraejemplo que lo demuestra. En la Tabla 5.2 se presentan los contraejemplos encontrados, junto a su respectivo contexto y restricción derivada. Los casos más interesantes son el programa $\texttt{C}_{\texttt{dvc}-}$, el cual posee más de un contraejemplo y el programa $\texttt{C}_{\texttt{pvc}-}$, el que no tiene un contraejemplo encontrado, ya que su obligación de prueba es una desigualdad entre constantes. Dado que la herramienta logra encontrar los contraejemplos de forma correcta, es posible afirmar que el objetivo específico asociado fue logrado.

5.6. Síntesis del capítulo

En este capítulo se presentó el procedimiento diseñado para validar el desarrollo hecho. La propiedad evaluada es la correctitud de los cálculos automatizados. Para hacer esto, se procedió con un enfoque basado en *tests*, donde cada *tests* es una comparación entre los cálculos automatizados y los cálculos manuales de la transformada $\mathsf{ert}[\cdot]$, frente a un mismo programa. En el capítulo se definen los criterios de aceptación de cada *test*, estos criterios buscan evaluar los diferentes tipos de *output* que la herramienta puede retornar. Por lo mismo, en primer lugar se define un criterio para los programas sin ciclos, donde la herramienta sólo retorna el tiempo de ejecución. En segundo lugar, se define un criterio para los programas con ciclos e invariantes correctos, donde la herramienta retorna el tiempo de ejecución, sus obligaciones de prueba y la validación de esas obligaciones. Por último, se define el criterio para los programas con ciclos e invariantes incorrectos, donde se retorna un tiempo de ejecución, las obligaciones de prueba y un contraejemplo para las obligaciones inválidas.

A continuación, se define una categorización sobre los programas usados en los *tests*. El

objetivo de esta categorización es abordar los diferentes tipos de comportamientos que poseen los programas `Program`.

En el capítulo se presentan, en detalle, 3 casos representativos de programas de los 12 que se analizaron en total. El primer caso abordar un programa determinista sin ciclos, el segundo un programa determinista con ciclo e invariante incorrecto y el tercero un programas probabilista con ciclo e invariante correcto. Este último es el más interesante, ya que introduce un método para sintetizar sus invariantes.

Con respecto a los resultados, los 12 *tests* propuestos fueron aprobados, por lo que se puede concluir que la herramienta consigue los objetivos propuestos.

Capítulo 6

Conclusión

6.1. Resumen del trabajo realizado

El trabajo realizado responde de manera directa al objetivo general de esta investigación: Desarrollar una herramienta que permita automatizar los cálculos de la transformada ert[·]. Dado los resultados obtenidos en la Validación (ver Sección 5), es posible afirmar que se logró abordar este propósito con éxito.

Para lograr este objetivo se implementaron una serie de estructuras inductivas en el lenguaje Haskell. Estas estructuras son necesarias para luego implementar la transformada vcg[·] (una modificación de ert[·]), y un *parser* que permite escribir los programas de manera cómoda.

En el caso que el programa analizado contenga ciclos, la función vcg[·] retorna un conjunto de obligaciones de prueba, las que se verifican con la ayuda del *smt-solver* Z3. Este *solver* fue manipulado a través de SBV, una librería que provee una interfaz para usar *smt-solvers*.

En cuanto a la validación, se siguió un enfoque basado en *tests*, comparando los resultados automatizados con los resultados calculados a mano. En los *tests* asociados a los programas sin ciclos, los cálculos coinciden con la transformada ert[·], y en los programas con ciclos, la herramienta desarrollada logra razonar de forma correcta sobre las obligaciones de prueba, validándolas cuando son satisfacibles y entregando un contraejemplo cuando no lo son. A lo anterior se le agrega que fue posible escribir los *inputs* de la herramienta como un *string*, por lo que se concluye que el *parser* desarrollado es correcto. Dado los resultados expuestos, cada objetivo específico fue cumplido, y en consecuencia, se alcanzó el objetivo general.

6.2. Discusión de los resultados

Pese a que se alcanzó el objetivo general, existen ciertos puntos que merecen ser discutidos. A continuación los más importantes.

En cuanto a las estructuras implementadas, existen algunas diferencias entre la representación matemática y la forma en que se imprimen los resultados. Estas diferencias son mínimas y no representan un inconveniente mayor. La mayor de las deficiencias se encuentra en la dimensión de las estructuras calculadas.

Para las expresiones `AExp` se implementó la normalización, por lo que cada expresión aritmética se reducía a una expresión minimal. Para las expresiones `BExp` se implementaron algunas simplificaciones que se mostraron suficientes para reducir las expresiones booleanas utilizadas. Para las expresiones `RunTime` y, en consecuencia, las obligaciones de prueba $\mathbb{O}$, el proceso de simplificación se mostró insuficiente. Algunas de las obligaciones generadas tenían la forma de $f \leq f$ y la herramienta no logra simplificarlas a $0 \leq 0$.

Las razones tras de este comportamiento son dos. La primera es que la simplificación sobre las obligaciones se hace por componente, es decir, la simplificación de $f_1 \leq f_2$ es $f_1' \leq f_2'$, si f_1' es la simplificación de f_1, y f_2' es la simplificación de f_2. Esta simplificación es deficiente, ya que en ninguna etapa de la simplificación se comparan las expresiones f_1' y f_2'. Además de lo anterior, se tiene que la operación de igualdad entre tiempos de ejecución ($==_{\texttt{RunTime}}$), no está bien definida. Esto provoca que no sea posible comparar dos tiempos de equivalentes de manera efectiva. Por ejemplo, para efectos de la implementación hecha, el tiempo de ejecución $x - 1$ es diferente a $-1 + x$.

Para lograr definir de manera adecuada la operación $==_{\texttt{RunTime}}$, es necesario definir la normalización de los tiempos de ejecución `RunTime`. La simplificación actual de los tiempos de ejecución se compone de una serie de reglas, las que no llegan a la profundidad necesaria. La definición de la normalización permitiría, en primer lugar, reducir el tamaño de los tiempos de ejecución asociados a las obligaciones de prueba y, en segundo lugar, compararlos entre si. Dada una obligación de prueba $f_1 \leq f_2$, si la versión normalizada de f_1 es igual a la versión normalizada de f_2, entonces se tiene que $f_1 ==_{\texttt{RunTime}} f_2$ y, por lo tanto, la obligación original puede ser reemplazada por $0 \leq 0$.

Es provechoso agregar que la simplificación de los elementos `RunTime`, no afecta solo a la visualización de las obligaciones de prueba, sino que también afecta al rendimiento. La herramienta no hace diferencias entre las obligaciones de prueba, por lo que ejecuta el procedimiento de validación, aún cuando se podría verificar la obligación $f \leq f$ de forma inmediata. Otra aplicación de la simplificación, también relacionada al rendimiento, es la formación de los problemas lineales. Si se tuviese la obligación $[\![a]\!] \cdot [\![b]\!] \cdot [\![c]\!] \leq 0$, se generarían 8 problemas lineales. En cambio, con la versión normalizada $[\![a \;\&\&\; b \;\&\&\; c]\!] \leq 0$, sólo se generaría un problema lineal. A pesar de que, para los programas estudiados, estos problemas de rendimiento no son relevantes, para programas más extensos si podrían representar un problema.

Otro punto que es necesario mencionar es la síntesis de invariantes. Para los programas deterministas se ocupó la intuición para justificar los invariantes correctos. Por el contrario, para los programas probabilistas se siguió un procedimiento para justificar su creación. Para el ciclo `pwhile` presentado en la Validación (ver Sección 5.4.3) se ocupó una iteración de punto fijo para encontrar los invariantes asociados. Otro enfoque se aprecia en el Apéndice C.1.8, donde se sintetiza los invariantes mediante la manipulación algebraica de las obligaciones de prueba. En mayor o menor medida, está la intuición que para el conjunto estudiado de

programas existe la posibilidad de abordar la síntesis automática de invariantes.

6.3. Reflexión del trabajo realizado

Como se dijo en la introducción, los programas probabilísticos son activamente usados en la cada día. El resultado entregado es un buen inicio para que en un futuro se pueda predecir el tiempo de ejecución de un programa real (o el *core* de un programa real). Para llegar a ese punto, aún se requiere mucha investigación, pero al menos para los programas sin ciclos, la herramienta se muestra óptima.

Con respecto a las decisiones de diseño, en vista a los resultados, se mostraron como correctas. La elección de Haskell como el lenguaje de programación guía fue, sin dudas, un acierto. Su comportamiento funcional permitió implementar, de forma directa, las estructuras y funciones presentadas en el trabajo original.

Relacionado con el punto anterior, la elección de SBV también fue correcta. En los comienzos de esta investigación se usó SMT-LIB 2.0 [1] para trabajar con Z3. Sin embargo, esta herramienta mostró ser poco práctica. Para este tipo de desarrollo se necesita una herramienta que manipule *smt-solvers* en un alto nivel, como lo es SBV.

Con respecto a las decisiones que pudieron ser mejores, una de las más relevantes es el estudio sobre las mónadas. Se debió invertir más tiempo en el estudio y ejercitación de este tipo de estructuras. Esta decisión hubiese permitido optimizar tiempo de desarrollo, por ejemplo, una de las funciones que más tiempo tardó en implementarse usa el `let`. En un principio, no era claro que se podría ocupar este *binding* dentro de una mónada y, por lo tanto, la función pasó por muchas iteraciones hasta llegar a la versión correcta. El tener un estudio previo, más profundo, de la estructura mónada hubiese permitido llegar a la versión final de la función más rápidamente.

Siguiendo con la línea de optimizar el tiempo de desarrollo, se menciona a la comunidad de Haskell en StackoverFlow. Esta comunidad muestra estar bien preparada para compartir su conocimiento al que lo requiera. Un intercambio más activo con esta comunidad hubiese permitido entender de forma más rápida la documentación de las librerías ocupadas, o algunos ejemplos encontrados durante la investigación previa al desarrollo.

A modo de anécdota, mientras se trabajaba con SBV, surgió un error con la resolución de problemas lineales. El sistema fallaba al tratar de resolver si $5 <= 3$. Pese a que no es necesario ocupar un *smt-solver* para resolver este problema, claramente, esta falla era un error y debía ser resuelta. Luego de investigar por varios medios, se decidió consultar por el foro StackoverFlow. En el foro se averiguó que el error era propio de la librería y no del desarrollo. Un usuario generó un *issue*[1] en el repositorio de github de SBV y a las pocas horas se solucionó.

[1]Para más detalles visitar https://github.com/LeventErkok/sbv/issues/591

6.4. Posibles trabajos futuros

Pese a que la herramienta cumple con los objetivos propuestos, esta admite muchas mejoras aplicables, ya sea para enriquecer la implementación o para extender las funcionalidades actuales. A continuación, se presentan algunas de estas mejoras.

Expansión del conjunto `AExp`. Como un primer paso para la realización de este trabajo, se decidió acotar el problema dentro de las expresiones lineales. Como un siguiente paso se plantea la expansión hacia los polinomios. Para usar este tipo de estructuras, se debería cambiar el tipo de variable a los números *algebraicos*, ya que SBV permite representar polinomios con ese tipo de número. Este cambio representaría un gran desafío, no sólo por el cambio en el tipo de variable, sino que también implica implementar nuevamente las expresiones `AExp`, reemplazando la ponderación por la multiplicación en la sintáxis abstracta.

$$\texttt{AExp} \rightarrow \texttt{AExp} * \texttt{AExp}$$

Situación similar ocurre en los tiempos de ejecución `RunTime`, donde existen los constructores de ponderación y multiplicación por una indicatriz. Estos serían reemplazados por la multiplicación entre tiempos de ejecución.

$$\texttt{RunTime} \rightarrow \texttt{RunTime} * \texttt{RunTime}$$

Es claro que este cambio afecta a las funciones relacionadas a las estructuras (simplificadores, función de sustitución, *parser*, etc), por lo mismo se considera como un cambio importante.

Definir forma normal de los tiempos de ejecución `RunTime`. Definir una forma normal para los tiempos de ejecución permitiría poder simplificar, hasta su mínima expresión, un elemento de esta estructura. A su vez, esto permitiría abordar la simplificación de obligaciones de prueba. Como se mencionó en el capítulo anterior, la herramienta desarrollada no simplifica las obligaciones a su versión minimal, por lo que obligaciones de prueba de la forma $f \leq f$ son tratadas como cualquier otra. De existir la normalización de los elementos `RunTime`, estas obligaciones se simplificarían hasta llegar a $0 \leq 0$, pudiendo ser evaluadas sin la necesidad del proceso introducido en la Sección 4.3.1.

Síntesis de invariantes. De los puntos no abordados en este trabajo, el de mayor relevancia es la síntesis de invariantes. En el trabajo original los autores plantean una técnica basada en plantillas, la cual no ha sido explorada en esta investigación. Otro enfoque, el cual produjo un resultado positivo en la Sección 5.4.3, es usar el Teorema del punto fijo de Kleene. A continuación, algunas ideas para seguir con esta perspectiva.

Se propone que en un futuro la herramienta pueda extraer, de forma automática, la función característica de un ciclo y ejercer las iteraciones de punto fijo hasta un límite establecido o hasta cumplir con algún criterio de convergencia.

Con respecto al último punto, se propone como criterio la siguiente condición:

$$\forall \sigma \ |\mathtt{I}^{\mathtt{n+1}}(\sigma) - \mathtt{I}^{\mathtt{n}}(\sigma)| \leq \varepsilon$$

La condición anterior es equivalente a:

$$\forall \sigma \ \big(-\varepsilon \leq \mathtt{I}^{\mathtt{n+1}}(\sigma) - \mathtt{I}^{\mathtt{n}}(\sigma)\big) \wedge \big(\mathtt{I}^{\mathtt{n+1}}(\sigma) - \mathtt{I}^{\mathtt{n}}(\sigma) \leq \varepsilon\big)$$

Es decir, es necesario resolver dos obligaciones de prueba, proceso que ya se ha discutido como abordarlo.

Bibliografía

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, sep 2011.

[4] Levent Erkök. Smt based verification (sbv). `https://github.com/LeventErkok/sbv`.

[5] Jürgen Giesl, Peter Giesl, and Marcel Hark. Computing expected runtimes for constant probability programs, 2019.

[6] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, page 167–181, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Tikhon Jelvis. Analyzing programs with z3. https://jelv.is/talks/compose-2016/, 2016.

[8] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 364–389, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[9] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. *CoRR*, abs/1601.01001, 2016.

[10] Hanne Riis Nielson. A hoare-like proof system for analysing the computation time of programs. *Science of Computer Programming*, 9(2):107–136, 1987.

[11] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*, chapter 9. Springer-Verlag, Berlin, Heidelberg, 2007.

[12] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 204–220, New York, NY, USA, 2019. Association for Computing Machinery.

[13] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*, chapter 10.2. Foundations of computing. MIT Press, 1993.

Anexos

Anexo A

Sintaxis concreta de las diferentes estructuras recursivas

En esta sección se describe la sintaxis concreta perteneciente a cada estructura presentada en la Sección 4.1. La mayoría de las definiciones son auto explicativas, con excepción de las estructuras `Base`. Tanto `aexpBase` como `runTimeBase` son el equivalente al "átomo" de su estructura respectiva, es decir, un elemento minimal. Es objetivo de estos elementos es disminuir la cantidad de paréntesis. En el caso que se expanda el conjunto `AExp` a los polinomios, será posible eliminar estas definiciones.

Definición A.1 (Sintaxis concreta de los números racionales)

`racional`	$\rightarrow$	`n`	*Número entero*
		`p/q`	*División entre enteros*

Definición A.2 (Sintaxis concreta de las expresiones aritméticas base)

`aexpBase`	$\rightarrow$	`racional`	*Número racional*
		String	*Variable*

Definición A.3 (Sintaxis concreta de las expresiones aritméticas deterministas)

`aexp`	$\rightarrow$	`aexpBase`	*Expresiones aritméticas base*
		`racional` $*$ `aexpBase`	*Ponderación de una expresión base por constante*
		`racional` $*$ (`aexp`)	*Ponderación por constante racional*
		`aexp` $+$ `aexp`	*Suma de expresiones* `aexp`
		`aexp` $-$ `aexp`	*Resta de expresiones* `aexp`*, Azúcar sintáctica*

Definición A.4 (Sintaxis concreta de las expresiones booleanas deterministas)

bexp →	true, false	*Constante booleana*
	aexp == aexp	*Igualdad de expresiones* aexp
	aexp != aexp	*Desigualdad de expresiones* aexp*, Azúcar sintáctica*
	aexp <= aexp	*Menor igual de expresiones* aexp
	aexp >= aexp	*Mayor igual de expresiones* aexp*, Azúcar sintáctica*
	aexp < aexp	*Menor estricto de expresiones* aexp*, Azúcar sintáctica*
	aexp > aexp	*Mayor estricto de expresiones* aexp*, Azúcar sintáctica*
	bexp || bexp	*Conjunción de* bexp
	bexp && bexp	*Disyunción de* bexp
	! bexp	*Negación de* bexp

Definición A.5 (Sintaxis concreta expresiones aritméticas probabilistas)

paexp →	racional ∗ < aexp >	*Punto de una distribución*
	paexp + paexp	*Suma de puntos*

Definición A.6 (Sintaxis concreta de las expresiones booleanas probabilistas)

pbexp → < racional >

Definición A.7 (Sintaxis concreta de los tiempos de ejecución base)

runTimeBase →	aexp	*Expresión aritmética*
	[bexp]	*Indicatriz*
	[bexp] <> (aexp)	*Multiplicación Indicatriz por expresión aritmética.*

Definición A.8 (Sintaxis concreta de los tiempos de ejecución)

runtime →	runTimeBase	*Tiempos de ejecución base*
	racional ⋇ runTimeBase	*Ponderación de una expresión base por constante*
	racional ⋇ (runtime)	*Ponderación de tiempo de ejecución por constante*
	[bexp] <> runTimeBase	*Multiplicación de una expresión base por una indicatriz*
	[bexp] <> (runtime)	*Multiplicación por una indicatriz*
	runtime ++ runtime	*Suma de expresiones* runtime
	runtime −− runtime	*Resta de expresiones* runtime

Anexo B

Funciones y demostraciones asociadas a RunTime

B.1. Funciones de sustitución

Las funciones de sustitución son las encargadas de sustituir una variable $\mathtt{x}$ por una expresión aritmética $\mathtt{a_i}$ en una cierta estructura. De las tres funciones de sustitución implementadas, en estricto rigor, solo la sustitución sobre los tiempos de ejecución (`sustRuntime`) es ocupada en la función $\mathtt{vcg}[\cdot]$, pero debido a la naturaleza inductiva del conjunto `RunTime`, era necesario definir la sustitución para `AExp` y para `BExp`.

Antes de definir formalmente las funciones, para simplificar la notación se introduce la función ϕ.

$$\phi(c, x, y) = \begin{cases} x & \text{si } c \\ y & \text{si } \neg c. \end{cases} \tag{B.1}$$

Tabla B.1: Definición función sustAExp

$$\begin{array}{ll} \mathtt{sustAExp}(\mathtt{y}, \mathtt{q}) & = \mathtt{q} \\ \mathtt{sustAExp}(\mathtt{y}, \mathtt{x}) & = \phi(\mathtt{x} == \mathtt{y}, \mathtt{y}, \mathtt{x}) \\ \mathtt{sustAExp}(\mathtt{y}, \mathtt{q} * \mathtt{a_1}) & = \mathtt{q} * \mathtt{sustAExp}(\mathtt{y}, \mathtt{a_1}) \\ \mathtt{sustAExp}(\mathtt{y}, \mathtt{a_1} + \mathtt{a_2}) & = \mathtt{sustAExp}(\mathtt{y}, \mathtt{a_1}) + \mathtt{sustAExp}(\mathtt{y}, \mathtt{a_2}) \end{array}$$

Tabla B.2: Definición función sustBExp

$$
\begin{array}{ll}
\texttt{sustBExp}(\texttt{y}, \texttt{true}) & = \texttt{true} \\
\texttt{sustBExp}(\texttt{y}, \texttt{false}) & = \texttt{false} \\
\texttt{sustBExp}(\texttt{y}, \texttt{a}_1 == \texttt{a}_2) & = \texttt{sustAExp}(\texttt{y}, \texttt{a}_1) == \texttt{sustAExp}(\texttt{y}, \texttt{a}_2) \\
\texttt{sustBExp}(\texttt{y}, \texttt{a}_1 <= \texttt{a}_2) & = \texttt{sustAExp}(\texttt{y}, \texttt{a}_1) <= \texttt{sustAExp}(\texttt{y}, \texttt{a}_2) \\
\texttt{sustBExp}(\texttt{y}, \texttt{b}_1 \&\& \texttt{b}_2) & = \texttt{sustBExp}(\texttt{y}, \texttt{a}_1) \;\&\&\; \texttt{sustBExp}(\texttt{y}, \texttt{b}_2) \\
\texttt{sustBExp}(\texttt{y}, \texttt{b}_1 || \texttt{b}_2) & = \texttt{sustBExp}(\texttt{y}, \texttt{a}_1) \;||\; \texttt{sustBExp}(\texttt{y}, \texttt{b}_2) \\
\texttt{sustBExp}(\texttt{y}, !\texttt{b}) & = !\texttt{sustBExp}(\texttt{y}, \texttt{b})
\end{array}
$$

Tabla B.3: Definición función sustRuntime

$$
\begin{array}{ll}
\texttt{sustRuntime}(\texttt{y}, \texttt{a}) & = \texttt{sustAExp}(\texttt{y}, \texttt{a}) \\
\texttt{sustAExp}(\texttt{y}, [\texttt{b}] \cdot \texttt{runt}) & = [\texttt{sustBExp}(\texttt{y}, \texttt{b})] \cdot \texttt{sustRuntime}(\texttt{y}, \texttt{runt}) \\
\texttt{sustAExp}(\texttt{y}, \texttt{q} * \texttt{runt}) & = \texttt{q} * \texttt{sustRuntime}(\texttt{y}, \texttt{runt}) \\
\texttt{sustAExp}(\texttt{y}, \texttt{runt}_1 + \texttt{runt}_2) & = \texttt{sustAExp}(\texttt{y}, \texttt{runt}_1) + \texttt{sustAExp}(\texttt{y}, \texttt{runt}_2)
\end{array}
$$

B.1.1. Función getBExp

Tabla B.4: Definición función `getBExp`

$$
\begin{array}{ll}
\texttt{getBExp}(\texttt{a}) & = \emptyset \\
\texttt{getBExp}([!\texttt{b}] \cdot \texttt{runt}) & = \texttt{b} \cup \texttt{getBExp}(\texttt{runt}) \\
\texttt{getBExp}([\texttt{b}] \cdot \texttt{runt}) & = \texttt{b} \cup \texttt{getBExp}(\texttt{runt}) \\
\texttt{getBExp}(\texttt{q} * \texttt{runt}) & = \texttt{getBExp}(\texttt{runt}) \\
\texttt{getBExp}(\texttt{runt}_1 + \texttt{runt}_2) & = \texttt{getBExp}(\texttt{runt}_1) \cup \texttt{getBExp}(\texttt{runt}_1)
\end{array}
$$

B.2. Cerradura de la sintáxis abstracta RunTime

Teorema B.1 *Para todo tiempo de ejecución calculado a través de la transformada* $\mathsf{ert}[\![\texttt{C}]\!](f)$ *(ver Tabla 3.1), con la excepción de tiempo asociado al ciclo* `while`*, existe un elemento equivalente en el conjunto* `RunTime` *si* $f \in \texttt{RunTime}$*.*

Demostración. Como primer paso se debe notar que, por definición, toda expresión $\texttt{sustRuntime}(\texttt{y}, f) \in \texttt{RunTime}$ si $f \in \texttt{RunTime}$. De lo contrario, la expresión $\texttt{sustRuntime}(\texttt{y}, f)$ no está definida. La demostración procede por inducción sobre el conjunto pProg.

- Caso $\mathsf{ert}[\![\texttt{empty}]\!](f) = f$. Se tiene ya que $f \in \texttt{RunTime}$.

- Caso $\mathsf{ert}[\mathtt{skip}](f) = 1+f$. Es posible representar la constante 1, y es posible representar la suma con el constructor + . Si $f \in \mathtt{RunTime}$ entonces $1 + f \in \mathtt{RunTime}$.
- Caso $\mathsf{ert}[\mathtt{x} :\approx \mu](f) = 1 + \lambda\sigma.\mathbf{E}_\mu(\lambda \mathrm{v}.\ f[\mathrm{x/v}](\sigma))$. La expresión $\lambda\sigma.\mathbf{E}_\mu(\lambda \mathrm{v}.\ f[\mathrm{x/v}](\sigma))$ es una suma ponderada de sustituciones. Para la sustitución se ocupa la función `sustRuntime`, para la ponderación existe el constructor $*$ y para la suma existe constructor +. Si $f \in \mathtt{RunTime}$ entonces $1 + \lambda\sigma.\mathbf{E}_\mu(\lambda \mathrm{v}.\ f[\mathrm{x/v}](\sigma) \in \mathtt{RunTime}$.
- Caso $\mathsf{ert}[\mathtt{C}_1; \mathtt{C}_2](f) = \mathsf{ert}[\mathtt{C}_1](\mathsf{ert}[\mathtt{C}_2](f))$. La expresión $\mathsf{ert}[\mathtt{C}_2](f) \in \mathtt{RunTime}$ por hipótesis inductiva, misma situación con $\mathsf{ert}[\mathtt{C}_1](\mathsf{ert}[\mathtt{C}_2](f))$.
- $\mathtt{if}\ (\xi)\{\mathtt{C}_1\}\ \mathtt{else}\ \{\mathtt{C}_2\} = 1 + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}_1](f) + [\![\neg\xi]\!] \cdot \mathsf{ert}[\mathtt{C}_2](f)$. Por hipótesis inductiva, $\mathsf{ert}[\mathtt{C}_1](f) \in \mathtt{RunTime}$ y $\mathsf{ert}[\mathtt{C}_2](f) \in \mathtt{RunTime}$. Si ξ es una guarda determinista, entonces las expresiones $[\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}_1](f)$ y $[\![\neg\xi]\!] \cdot \mathsf{ert}[\mathtt{C}_2](f)$ son representables mediante la multiplicación por una indicatriz. En el caso que una guarda probabilista se puede representar con la ponderación.
- $\mathtt{while}\ (\xi)\ \{\mathtt{C}'\} = \mathtt{lfp} X.1 + [\![\neg\xi]\!] \cdot f + [\![\xi]\!] \cdot \mathsf{ert}[\mathtt{C}'](X)$. Para este caso se usan invariantes pertenecientes a `RunTime`.

□

Anexo C

Tests y demostraciones asociados a la Validación

C.1. *Tests* usados en la Validación, no incluidos en el texto principal

C.1.1. Programa determinista, tiempo de ejecución variable y sin ciclo ($\mathtt{C}_{\mathtt{dvs}}$)

```
1  x := x − 1;
2  if (x >= y)
3        { skip;
4         y := 2 ∗ x }
5  else {
6         if (w >= 8){
7               z := 3;
8               x := w + x }
9         else {
10              y := 5 }}
```

Figura C.1: Programa $\mathtt{C}_{\mathtt{dvs}}$

El programa analizado se presenta en la Figura C.1. El tiempo de ejecución resultante es dependiente del estado. Inicia disminuyendo el valor de `x` en una unidad. Luego ejecuta un condicional `if` cuya guarda es igual `x >= y`. La rama `true` ejecuta el programa `skip` y duplica el valor de `y`, terminando el programa. En la rama `false` se define otro condicional `if`. La guarda es la expresión `w >= 8`, cuya rama `true` ejecuta la modificación de las variables `z` y `x`, finalizando el programa. En la otra rama se cambia el valor de `y`, finalizando el programa.

◇ $2 + [\![(x-1) >= y]\!] \cdot 2 + [\![\neg(x-1) >= y]\!] \cdot (1 + [\![w >= 8]\!] \cdot 2 + [\![\neg w >= 8]\!])$
◇ $1 + 1 + [\![(x-1) >= y]\!] \cdot 2 + [\![\neg(x-1) >= y]\!] \cdot (1 + [\![w >= 8]\!] \cdot 2 + [\![\neg w >= 8]\!])$
1 x ≔ x − 1;
◇ $1 + [\![x >= y]\!] \cdot 2 + [\![\neg x >= y]\!] \cdot (1 + [\![w >= 8]\!] \cdot 2 + [\![\neg w >= 8]\!])$
2 if (x >= y)
◇ 2
◇ $1 + 1 + 0$
3 { skip;
◇ $1 + 0$
4 y ≔ 2 ∗ x }
◇ 0
5 else {
◇ $1 + [\![w >= 8]\!] \cdot 2 + [\![\neg w >= 8]\!]$
◇ $1 + [\![w >= 8]\!] \cdot 2 + [\![\neg w >= 8]\!] \cdot 1$
6 if (w >= 8){
◇ 2
◇ $1 + 1 + 0$
7 z ≔ 3;
◇ $1 + 0$
8 x ≔ w + x }
◇ 0
9 else {
◇ 1
◇ $1 + 0$
10 y ≔ 5 }}
◇ 0
◇ 0

Figura C.2: Cálculo ert[·] manual - Programa $\mathtt{C}_{\mathtt{dvs}}$

```
Programa Analizado:
x:=x-1; if(x>=y){skip; y:= 2*x}
        else{if(w>=8){w:= 3; x:=w+x} else{y:=5}}

Tiempo de ejecución calculado:
2 ++ [y <= -1 + x]<>2
  ++ [!(y <= -1 + x)]<>(1 ++ [8 <= w]<>2 ++ [!(8 <= w)])

El tiempo de ejecución calculado es válido
porque no hay obligaciones de prueba asociadas, ya que el programa no
 contiene ciclos.

Análisis Finalizado.
```

Listing C.1: Cálculo ert[·] automático - $\mathtt{C}_{\mathtt{dvs}}$

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta logra representar los cálculos manuales de manera correcta. Existen diferencias

en la representación de algunas expresiones, por ejemplo, la expresión aritmética $x - 1$ se representa con $-1 + \texttt{x}$ debido al proceso de simplificación que se ejecuta sobre el tiempo de ejecución. En cuanto a la expresión booleana $w >= 8$, esta se representa con $\texttt{8} <= \texttt{w}$ debido a que la operación mayor o igual(>=) es azúcar sintáctica del menor o igual (<=). Pese a estas diferencias visuales el *test* se considera aprobado.

C.1.2. Programa determinista, tiempo de ejecución constante, con ciclo e invariante correcto ($\texttt{C}_{\texttt{dkc+}}$)

```
1  x := 3;
2  while (x > 0)
3        { inv = 1 ++ 2 ** [x > 0] <> (x + 1)}
4        { x := x - 1 }
```

Figura C.3: Programa determinista, con ciclo, tiempo de ejecución constante e invariante correcto

◇ $\big(10,\ \{[\![x > 0]\!] \cdot (1 + [\![x - 1 > 0]\!] \cdot x) \leq [\![x > 0]\!] \cdot (x + 1)\}\big)$
◇ $\big(1 + 1 + 2 \cdot [\![3 > 0]\!] \cdot (3 + 1),\ \{[\![x > 0]\!] \cdot (1 + [\![x - 1 > 0]\!] \cdot x) \leq [\![x > 0]\!] \cdot (x + 1)\}\big)$
1 `x := 3`
◇ $\big(1 + 2 \cdot [\![x > 0]\!] \cdot (x + 1),\ \{[\![x > 0]\!] \cdot (1 + [\![x - 1 > 0]\!] \cdot x) \leq [\![x > 0]\!] \cdot (x + 1)\}\big)$
◇ $\big(1 + 2 \cdot [\![x > 0]\!] \cdot (x + 1),\ \{1 + [\![x > 0]\!] \cdot (2 + 2 \cdot [\![x - 1 > 0]\!] \cdot x) \leq 1 + 2 \cdot [\![x > 0]\!] \cdot (x + 1)\}\big)$
2 `while (x > 0)`
3 `      { inv = 1 ++ 2 ** [x > 0] <> (x + 1)}`
◇ $\big(2 + 2 \cdot [\![x - 1 > 0]\!] \cdot x,\ \emptyset\big)$
◇ $\big(1 + 1 + 2 \cdot [\![x - 1 > 0]\!] \cdot x,\ \emptyset\big)$
4 `      { x := x - 1 }`
◇ $\big(1 + 2 \cdot [\![x > 0]\!] \cdot (x + 1),\ \emptyset\big)$
◇ $\big(0,\ \emptyset\big)$

Figura C.4: Cálculo $\mathsf{vcg}[\cdot]$ manual - Programa $\texttt{C}_{\texttt{dkc+}}$

El conjunto atómico generado es:

$$\mathcal{A}_{[\![x>0]\!]\cdot(1+[\![x-1>0]\!]\cdot x)\leq[\![x>0]\!]\cdot(x+1)} = \{(x > 0), (x - 1 > 0)\}$$

1.

Contexto	$= (x > 0) \land (x - 1 > 0)$
Restricción derivada	$= 1 \cdot (1 + 1 \cdot x) \leq 1 \cdot (x + 1)$
	$= 1 + (x - 1) \leq x + 1$
	$= 0 \leq 1$
Problema Lineal	$= (x > 0) \land (x - 1 > 0) \land \neg(0 \leq 1)$
	$= (x > 0) \land (x - 1 > 0) \land (0 > 1)$
	$= (x > 0) \land (x - 1 > 0) \land \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

2.

Contexto	$= \neg(x > 0) \land (x - 1 > 0)$
Restricción derivada	$= 0 \cdot (1 + 1 \cdot x) \leq 0 \cdot (x + 1)$
	$= 0 \leq 0$
Problema Lineal	$= \neg(x > 0) \land (x - 1 > 0) \land \neg(0 \leq 0)$
	$= \neg(x > 0) \land (x - 1 > 0) \land (0 > 0)$
	$= \neg(x > 0) \land (x - 1 > 0) \land \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

3.

Contexto	$= (x > 0) \land \neg(x - 1 > 0)$
Restricción derivada	$= 1 \cdot (1 + 0 \cdot x) \leq 1 \cdot (x + 1)$
	$= 1 \leq (x + 1)$
	$= 0 \leq x$
Problema Lineal	$= (x > 0) \land \neg(x - 1 > 0) \land \neg(0 \leq x)$
	$= (x > 0) \land (x \leq 1) \land (x < 0)$
	$= (x > 0) \land (x < 0) \land (x \leq 1)$
	$= \texttt{false} \land (\texttt{x} < \texttt{1})$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

4.

Contexto	$= \neg(x > 0) \land \neg(x - 1 > 0)$
Restricción derivada	$= 0 \cdot (1 + 0 \cdot x) \leq 0 \cdot (x + 1)$
	$= 0 \leq 0$
Problema Lineal	$= \neg(x > 0) \land \neg(x - 1 > 0) \land \neg(0 \leq 0)$
	$= (x \leq 0) \land (x - 1 \leq 0) \land (0 > 0)$
	$= (x \leq 0) \land (x \leq 1) \land \texttt{false}$
	$= \texttt{false}$
Modelo	= La fórmula no es satisfacible, no existe modelo válido.

Conclusión No existe un contraejemplo para la obligación de prueba. La obligación de prueba asociada al invariante $1 + 2 \cdot [\![x > 0]\!] \cdot (x + 1)$, es válida. Por ende, el invariante es

correcto.

```
Programa Analizado:
while(x > 0){inv = 1 ++ 2**[x>0]<>(x + 1) }{x:= x-1}

Tiempo de ejecución calculado:
10

Obligaciones de prueba asociadas:
[1]
1 ++ [!(x <= 0)]<>(2 ++ 2**([!(-1 + x <= 0)]<>x))
:!<=:
1 ++ 2**([!(x <= 0)]<>(1 + x))
Es válida

El tiempo de ejecución calculado es válido
porque las obligaciones de prueba son válidas.

Análisis Finalizado.
```

Listing C.2: Cálculo vcg[·] automático - Programa $\mathtt{C}_{\mathtt{dkc}+}$

Comparación entre los cálculos manuales y la herramienta desarrollada. Existen diferencias en la representación de algunas estructuras y la simplificación de la obligación de prueba no es óptima. Pese a lo anterior, la herramienta logra validar la obligación de prueba y retornar el tiempo de ejecución correcto. El *test* se considera aprobado.

C.1.3. Programa determinista, tiempo de ejecución variable, con ciclo e invariante correcto ($\mathtt{C}_{\mathtt{dvc}+}$)

$$
\begin{array}{ll}
1 & \mathtt{while}\ (\mathtt{y} <= \mathtt{x}\ \&\&\ \mathtt{x} <= \mathtt{z}) \\
2 & \qquad \{\ \mathtt{inv} = 1\ \ ++\ 2\ \ **\ [\mathtt{y} <= \mathtt{x}\ \&\&\ \mathtt{x} <= \mathtt{z}] <> (2 * (\mathtt{z} - \mathtt{x} + 1))\} \\
3 & \qquad \{\ \mathtt{x} := \mathtt{x} + 1/2\ \}
\end{array}
$$

Figura C.5: Programa $\mathtt{C}_{\mathtt{dvc}+}$

En este apartado se analiza un programa con tiempo de ejecución variable. El programa analizado es un ciclo `while` que itera mientras el valor de la variable x se encuentre entre y y z. La intuición del invariante es la misma que la ocupada en la Sección 5.4.2, por lo que no se profundiza en ello. Lo verdaderamente interesante es la complejidad de la condición. En esta oportunidad se tiene un disyunción de dos expresiones booleanas, y además se trabaja con 3 variables y no con una como en la mayoría de los *tests*.

$\diamond$ $\big(1 + 4 \cdot [\![y <= x \;\&\&\; x <= z]\!] \cdot (z - x + 1),$
$\{[\![y <= x \;\&\&\; x <= z]\!] \cdot \big(1 + 4 \cdot [\![y <= (x + \frac{1}{2}) \;\&\&\; (x + \frac{1}{2}) <= z]\!] \cdot (z - x + \frac{1}{2})\big)$
$\leq$
$4 \cdot [\![y <= x \;\&\&\; x <= z]\!] \cdot (z - x + 1)\}\big)$

$\diamond$ $\big(1 + 4 \cdot [\![y <= x \;\&\&\; x <= z]\!] \cdot (z - x + 1),$
$\{1 + [\![y <= x \;\&\&\; x <= z]\!] \cdot \big(1 + 4 \cdot [\![y <= (x + \frac{1}{2}) \;\&\&\; (x + \frac{1}{2}) <= z]\!] \cdot (z - x + \frac{1}{2})\big)$
$\leq$
$1 + 4 \cdot [\![y <= x \;\&\&\; x <= z]\!] \cdot (z - x + 1)\}\big)$

```
1  while (y <= x &&  x <= z)
2        { inv = 1 + +2 * *[y <= x &&  x <= z] <> (2 * (z - x + 1)) }
```

$\diamond$ $\big(1 + 4 \cdot [\![y <= (x + \frac{1}{2}) \;\&\&\; (x + \frac{1}{2}) <= z]\!] \cdot (z - x + \frac{1}{2}),\ \emptyset\big)$

$\diamond$ $\big(1 + 2 \cdot [\![y <= (x + \frac{1}{2}) \;\&\&\; (x + \frac{1}{2}) <= z]\!] \cdot 2 \cdot (z - (x + \frac{1}{2}) + 1),\ \emptyset\big)$

```
3        { x := x + 1/2 }
```

$\diamond$ $\big(1 + 2 \cdot [\![y <= x \;\&\&\; x <= z]\!] \cdot 2 \cdot (z - x + 1),\ \emptyset\big)$

$\diamond$ $\big(0,\ \emptyset\big)$

Figura C.6: Cálculo vcg[·] manual - Programa $\mathtt{C}_{\mathtt{dvc+}}$

El programa genera una obligación de prueba, a la cual se le denota como `O`. Su conjunto atómico es :

$$\mathcal{A}_{\mathtt{O}} = \{(y <= x \;\&\&\; x <= z), (y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z)\}$$

1.

Contexto	$= (y <= x \;\&\&\; x <= z) \wedge \big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big)$
Restricción derivada	$= 1 \cdot (1 + 4 \cdot 1 \cdot (z - x + \frac{1}{2})) \leq 4 \cdot\ 1 \cdot (z - x + 1).$
	$= 1 + 4 \cdot z - 4 \cdot x + 2 \leq 4 \cdot z - 4 \cdot x + 4$
	$= 0 \leq 1$
Problema Lineal	$= (y <= x \;\&\&\; x <= z) \wedge$
	$\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \neg(0 \leq 0)$
	$= (y <= x \;\&\&\; x <= z) \wedge$
	$\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge (0 > 0)$
	$= (y <= x \;\&\&\; x <= z) \wedge$
	$\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \mathtt{false}$
	$= \mathtt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

2.

Contexto	$= \neg(y <= x \;\&\&\; x <= z) \wedge \big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big)$
Restricción derivada	$= 0 \cdot (1 + 4 \cdot 1 \cdot (z - x + \frac{1}{2})) \leq 4 \cdot 0 \cdot (z - x + 1).$
	$= 0 \leq 0$
Problema Lineal	$= \neg(y <= x \;\&\&\; x <= z) \wedge$
	$\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \neg(0 \leq 0)$
	$= \neg(y <= x \;\&\&\; x <= z) \wedge$
	$\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge (0 > 0)$
	$= \neg(y <= x \;\&\&\; x <= z) \wedge$
	$\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

3.

Contexto	$= (y <= x \;\&\&\; x <= z) \wedge \neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big)$
Restricción derivada	$= 1 \cdot (1 + 4 \cdot 0 \cdot (z - x + \frac{1}{2})) \leq 4 \cdot 1 \cdot (z - x + 1).$
	$= 1 \leq 4 \cdot z - 4 \cdot x + 4$
	$= x \leq z + \frac{3}{4}$
Problema Lineal	$= (y <= x \;\&\&\; x <= z) \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \neg(x \leq z + \frac{3}{4})$
	$= (y <= x) \wedge (x <= z) \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge (x > z + \frac{3}{4})$
	$= (y <= x) \wedge \big(x <= z \wedge x > z + \frac{3}{4}\big) \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big)$
	$= (y <= x) \wedge \texttt{false} \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big)$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

4.

Contexto	$= \neg(y <= x \;\&\&\; x <= z) \wedge \neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big)$
Restricción derivada	$= 0 \cdot (1 + 4 \cdot 0 \cdot (z - x + \frac{1}{2})) \leq 4 \cdot 0 \cdot (z - x + 1).$
	$= 0 \leq 0$
Problema Lineal	$= \neg(y <= x \;\&\&\; x <= z) \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \neg(0 \leq 0)$
	$= \neg(y <= x \;\&\&\; x <= z) \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge (0 > 0)$
	$= \neg(y <= x \;\&\&\; x <= z) \wedge$
	$\neg\big(y <= \big(x + \frac{1}{2}\big) \;\&\&\; \big(x + \frac{1}{2}\big) <= z\big) \wedge \texttt{false}$
	$= \texttt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

Conclusión No existe un contraejemplo para la obligación de prueba. La obligación de prueba asociada al invariante $1 + 2 \cdot [\![y <= x \;\&\&\; x <= z]\!] \cdot 2 \cdot (z - x + 1)$, es válida, por ende, el invariante es correcto.

```
Programa Analizado:
while(y <= x && x <= z){ inv = 1 ++ 2**[y<=x && x<=z]<>(2*(z - x + 1))}
```

```
                        {x:= x+ 1/2}

Se calcula la transformada con respecto a:
0

Tiempo de ejecución calculado:
1 ++ 2**([y <= x && x <= z]<>(2 + -2*x + 2*z))


Obligaciones de prueba asociadas:
[1]
1 ++ [y <= x && x <= z]<>
     (2 ++ 2**([y <= 1/2 + x && 1/2 + x <= z]<>(1 + -2*x + 2*z)))
:!<=:
1 ++ 2**([y <= x && x <= z]<>(2 + -2*x + 2*z))
Es válida

El tiempo de ejecución calculado es válido
porque las obligaciones de prueba son válidas.

Análisis Finalizado.
```

Listing C.3: Cálculo vcg[·] automático - Programa $\mathtt{C}_{\mathtt{dvc}+}$

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta logra procesar una expresión booleana compleja, pudiendo entregar la obligación de prueba correcta y tiempo de ejecución buscado. El *test* se considera aprobado.

C.1.4. Programa determinista, tiempo de ejecución variable, con ciclo e invariante incorrecto ($\mathtt{C}_{\mathtt{dvc}-}$)

```
1  while (false)
2        { inv = [x >= 0] <> x}
3        { skip }
```

Figura C.7: Programa determinista, con ciclo, tiempo de ejecución constante e invariante correcto

El programa analizado es un caso borde. Se trata de un ciclo `while` cuya guarda es la constante `false`. Se usa como invariante el tiempo de ejecución $[\![x >= 0]\!] \cdot x$, el que es incorrecto. Un invariante correcto tiene la forma de $1 + f$ (recordar que por definición, todo tiempo de ejecución es mayor o igual a 0). La intuición tras esta afirmación es simple, el programa evalúa su guarda y luego finaliza, por lo tanto, consume siempre una unidad de tiempo y cualquier invariante mayor a este número sería correcto, por ejemplo, la constante 2 o el tiempo $1 + [\![x >= 0]\!] \cdot x$.

◇ $(\llbracket x >= 0 \rrbracket \cdot x,\ \{1 \leq \llbracket x >= 0 \rrbracket \cdot x\})$
◇ $(\llbracket x >= 0 \rrbracket \cdot x,\ \{10 \cdot \llbracket \texttt{true} \rrbracket + \llbracket \texttt{false} \rrbracket \cdot (1 + \llbracket \texttt{x} >= 0 \rrbracket \cdot \texttt{x}) \leq \llbracket \texttt{x} >= 0 \rrbracket \cdot \texttt{x}\})$

```
1 while (false)
2       { inv = [x >= 0] <> x}
```

◇ $(1 + \llbracket x >= 0 \rrbracket \cdot x,\ \emptyset)$

```
3       { skip }
```

◇ $(\llbracket x >= 0 \rrbracket \cdot x,\ \emptyset)$
◇ $(0,\ \emptyset)$

Figura C.8: Programa $\texttt{C}_{\texttt{dvc}-}$

El programa tiene 1 obligación de prueba asociada. Su conjunto atómico es:

$$\mathcal{A}_{1 \leq \llbracket x >= 0 \rrbracket \cdot x} = \{x >= 0\}$$

1.

Contexto	$= x >= 0$
Restricción derivada	$= 1 \leq 1 \cdot x$
	$= 1 \leq x$
Problema Lineal	$= (x >= 0) \wedge \neg(1 \leq x)$
	$= (x >= 0) \wedge (1 > x)$
	$= \texttt{true}$
Modelo	= El problema es satisfacible, $x = 0$ es un modelo válido.

2.

Contexto	$= \neg(x >= 0)$
Restricción derivada	$= 1 \leq 0 \cdot x$
	$= 1 \leq 0$
Problema Lineal	$= \neg(x >= 0) \wedge \neg(1 \leq 0)$
	$= \neg(x >= 0) \wedge (1 > 0)$
	$= \neg(x >= 0) \wedge \texttt{true}$
	$= \texttt{true}$
Modelo	= El problema es satisfacible, $x = -1$ es un modelo válido.

```
Programa Analizado:
while(false){inv = [x >= 0]<>x}{skip}

Tiempo de ejecución calculado:
[0 <= x]<>x

Obligaciones de prueba asociadas:
[1]  1 :!<=: [0 <= x]<>x, No es válida

**********************************************************************

Obligación de prueba [1]
1 :!<=: [0 <= x]<>x
```

```
  La obligación de prueba tiene asociada 2 restricciones derivadas
   diferentes.

------------------------------------------------------------------------

  Restricción derivada [1, 1]
  [!(0 <= x)]   |-  1 :!<=: 0

  La restricción no es válida.

  Un contraejemplo encontrado es:
  x = -1 Racional
------------------------------------------------------------------------

------------------------------------------------------------------------

  Restricción derivada [1, 2]
  [0 <= x]   |-  1 :!<=: x

  La restricción no es válida.

  Un contraejemplo encontrado es:
  x = 0 Racional
------------------------------------------------------------------------

  El tiempo de ejecución calculado no es válido
  porque alguna obligación de prueba no es válida.
  Ajuste los invariantes de ciclo y vuelva a realizar el análisis.

  Análisis Finalizado.
```

Listing C.4: Cálculo vcg[·] automático - Programa $\mathtt{C}_{\mathtt{dvc}-}$

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta logra razonar correctamente sobre el tiempo de ejecución y la obligación de prueba. Logra comprobar la invalidez de las obligaciones de prueba y entregar un contraejemplo para ambas. Debido a esto, el *test* se considera aprobado.

C.1.5. Programa probabilístico, con tiempo de ejecución constante y sin ciclos ($\mathtt{C}_{\mathtt{pks}}$)

```
1 pif (< 1/2 >)
2       {succ :~ 5/100 * < 0 > + 95/100 * < 1 >}
3 pelse {
4       pif (< 1/2 >){
5             {succ :~ 5/100 * < 0 > + 95/100 * < 1 >}
6       pelse {
7             {succ :~ 95/100 * < 0 > + 5/100 * < 1 >}}
```

Figura C.9: Programa $\mathtt{C}_{\mathtt{pks}}$

Este programa probabilístico para analizar es una versión "completamente"probabilista de programa C_{trunc} en la Figura 1.1. La diferencia radica en la asignación. En el programa C_{trunc} se usa la asignación determinista (:=) para establecer el valor de la variable `succ`, en cambio en este *test*, se usa una asignación probabilista (:∼), dándole una probabilidad de 0.95 al valor booleano que el programa original establece como valor de `succ`.

```
◇  5/2
◇  1 + 1/2 · 1 + 1/2 · 2
1  pif (< 1/2 >)
◇        1
◇        1 + 5/100 · 0 + 95/100 · 0
2        {succ :∼ 5/100 ∗ < 0 > + 95/100 ∗ < 1 >}
◇        0
3  pelse {
◇        2
◇        1 + 1/2 · 1 + 1/2 · 1
4        pif (< 1/2 >){
◇              1
◇              1 + 5/100 · 0 + 95/100 · 0
5              {succ :∼ 5/100 ∗ < 0 > +95/100 ∗ < 1 >}
◇              0
6        pelse {
◇              1
◇              1 + 95/100 · 0 + 5/100 · 0
7              {succ :∼ 95/100 ∗ < 0 > + 5/100 ∗ < 1 >}}
◇              0
◇  0
```

Figura C.10: Cálculo ert[·] manual - Programa $\mathtt{C}_{\mathtt{pks}}$

```
Programa Analizado:
pif(<1/2>){succ:~ 5/100* <0> + 95/100* <1>}
pelse {pif(<1/2>) {succ:~  5/100* <0> + 95/100* <1>}
       pelse{succ:~ 95/100* <0> + 5/100* <1>}}


Tiempo de ejecución calculado:
5/2

El tiempo de ejecución calculado es válido
porque no hay obligaciones de prueba asociadas,
ya que el programa no contiene ciclos.

Análisis Finalizado.
```

Listing C.5: Cálculo ert[·] automático - Programa $\mathtt{C}_{\mathtt{pks}}$

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta logra representar los cálculos manuales sin inconveniente alguno. El *test* se considera

aprobado.

C.1.6. Programa probabilístico, con tiempo de ejecución variable y sin ciclos ($\texttt{C}_{\texttt{pvs}}$)

```
1  pit (< 9/10 >) {
2        if (x > 10)
3             { skip }
4        else
5             {y := y + 1;
6             x := x − 1 }}
```

Figura C.11: Programa $\texttt{C}_{\texttt{pvs}}$

En esta oportunidad se estudia un condicional probabilista `pit`. Como se mencionó anteriormente, el programa `pit` es azúcar sintáctica para un condicional `pif` que tenga el programa `empty` en su rama `false`. En el cuerpo del programa `pit` se encuentra un condicional `if` con guarda igual a `x >= y`. Si la guarda es verdadera ejecuta un `skip` y termina, en cambio si es falsa, disminuye en una unidad del valor de `x` y termina.

```
◇  1 + 9/10 · (1 + [[x > 10]] + [[¬x > 10]] · 2)
◇  1 + 9/10 · (1 + [[x > 10]] + [[¬x > 10]] · 2) + 0
1  pit (< 9/10 >) {
◇        1 + [[x > 10]] · 1 + [[¬x > 10]] · 2
2        if (x > 10)
◇             1
◇             1 + 0
3             { skip }
◇             0
4        else
◇             2
◇             2 + 0
5             {y := y + 1;
◇             1
◇             1 + 0
6             x := x − 1 }}
◇             0
◇  0
```

Figura C.12: Cálculo ert[·] manual - Programa $\texttt{C}_{\texttt{pvs}}$

```
Programa Analizado:
pit(<9/10>){ if(x > 10){skip} else{ x:= x-1}}

Tiempo de ejecución calculado:
```

```
1 ++ 9/10**(1 ++ [!(x <= 10)] ++ [x <= 10]<>2)

El tiempo de ejecución calculado es válido
porque no hay obligaciones de prueba asociadas,
ya que el programa no contiene ciclos.

Análisis Finalizado.
```

Listing C.6: Cálculo $\mathsf{ert}[\cdot]$ manual - Programa $\mathtt{C}_{\mathtt{pvs}}$

Comparación entre los cálculos manuales y la herramienta desarrollada. Al igual que en casos anteriores, la herramienta muestra diferencias al momento de visualizar. La operación mayor estricto ($>$) es azúcar sintáctica para la negación del menor o igual ($\neg <=$), pese a esto los cálculos son representados de manera correcta. El *test* se considera aprobado.

C.1.7. Programa probabilístico, tiempo de ejecución constante, con ciclo e invariante incorrecto ($\mathtt{C}_{\mathtt{pkc-}}$)

```
1 for(3){pwhile (< 1/2 >) }
```

Figura C.13: Programa $\mathtt{C}_{\mathtt{pkc-}}$.

En esta oportunidad se estudia un programa probabilístico con invariantes incorrectos. Se ocupa la azúcar sintáctica del ciclo `for`. La interpretación de este programa es componer secuencialmente el programa de su cuerpo, en este caso, se compone 3 veces el ciclo de su cuerpo. En los cálculos manuales se presenta el programa con la composición hecha.

◇ $(3,\ \{3 \leq 3, \frac{9}{2} \leq 3, \frac{9}{2} \leq 3\})$
◇ $(3,\ \{3 \leq 3, \frac{9}{2} \leq 3, 1 + \frac{1}{2} \cdot 3 + \frac{1}{2} \cdot 3 \leq 3\})$
1 pwhile $(< 1/2 >)$
2 $\{$ pinv $= 3\}$
◇ $(4,\ \{3 \leq 3, \frac{9}{2} \leq 3\})$
◇ $(3+1,\ \{3 \leq 3, \frac{9}{2} \leq 3\})$
3 $\{$ skip $\}$;
◇ $(3,\ \{3 \leq 3, \frac{9}{2} \leq 3\})$
◇ $(3,\ \{3 \leq 3, \frac{9}{2} \leq 3\})$
◇ $(3,\ \{3 \leq 3, 1 + \frac{1}{2} \cdot 3 + \frac{1}{2} \cdot 4 \leq 3\})$
4 pwhile $(< 1/2 >)$
5 $\{$ pinv $= 3\}$
◇ $(4,\ \{3 \leq 3\})$
◇ $(3+1,\ \{3 \leq 3\})$
6 $\{$ skip $\}$;
◇ $(3,\ 3 \leq 3)$
◇ $(3,\ \{3 \leq 3\})$
◇ $(3,\ \{1 + \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 4 \leq 3\})$
7 pwhile $(< 1/2 >)$
8 $\{$ pinv $= 3\}$
◇ $(4,\ \emptyset)$
◇ $(3+1,\ \emptyset)$
9 $\{$ skip $\}$
◇ $(3,\ \emptyset)$
◇ $(0,\ \emptyset)$

Figura C.14: Cálculo vcg[·] manual - Programa $\mathtt{C}_{\mathtt{pkc-}}$

Claramente, la obligación $\frac{9}{2} \leq 3$ no es válida y la obligación $3 \leq 3$ si lo es. A continuación los cálculos automatizados.

```
  Programa Analizado:
  for(3){pwhile(<1/2>){pinv = 3}{skip}}

  Tiempo de ejecución calculado:
  3

  Obligaciones de prueba asociadas:
  [1]  9/2 :!<=: 3, No es válida
  [2]  9/2 :!<=: 3, No es válida
  [3]  3 :!<=: 3, Es válida

************************************************************

  Obligación de prueba [1]
  9/2 :!<=: 3

  La obligación de prueba tiene asociada 1 restricciones derivadas
   diferentes.

  ------------------------------------------------------------
```

```
Problema lineal [1, 1]
[]   |-  9/2 :!<=: 3

La restricción no es válida.


------------------------------------------------------------
************************************************************

Obligación de prueba [2]
9/2 :!<=: 3

La obligación de prueba tiene asociada 1 restricciones derivadas
 diferentes.

------------------------------------------------------------

Problema lineal [2, 1]
[]   |-  9/2 :!<=: 3

La restricción no es válida.


------------------------------------------------------------
El tiempo de ejecución calculado no es válido
porque alguna obligación de prueba no es válida.
Ajuste los invariantes de ciclo y vuelva a realizar el análisis.

Análisis Finalizado.
```

Listing C.7: Cálculo $\mathsf{vcg}[\cdot]$ automático - Programa $\mathtt{C}_{\mathtt{pkc}-}$

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta logra representar de manera exacta los cálculos manuales. Es interesante comprobar que la herramienta puede manejar el caso de un contexto vacío. A pesar de que, no es un problema de alta dificultad, si es un caso borde que debe ser explorado. El *test* se considera aprobado.

C.1.8. Programa probabilístico, tiempo de ejecución variable y con ciclo ($\mathtt{C}_{\mathtt{pvc}}$)

```
1  pwhile (< 9/10 >)
2        { pinv = I_1}
3        {while (c == 1)
4              { inv = I_2}
5              {c :~ 1/2 * < 0 > +1/2 * < 1 > }}
```

Figura C.15: Programa probabilístico, con ciclo y tiempo de ejecución variable

Este último *test* se propone estudiar el programa de la Figura C.15. Este programa es una anidación de un ciclo `pwhile`, en cuyo cuerpo se define un ciclo `while`. Al igual que el *test* precedente se propondrá un método para sintetizar los invariantes, pero, al contrario de la situación anterior, el método del punto fijo de Kleene no es un alternativa adecuada, ya que, debido a la definición de $\mathtt{vcg}[\cdot]$, los invariantes se relacionan de una forma menos amigable que sus homólogos de la Figura 5.6.

$$F_1(\mathtt{I_1}) = \mathtt{1} + \tfrac{9}{10} \cdot \mathtt{I_2}$$
$$F_2(\mathtt{I_2}) = \mathtt{1} + [\![\neg \mathtt{c} == \mathtt{1}]\!] \cdot \mathtt{I_1} + [\![\neg c == \mathtt{1}]\!] \cdot (\mathtt{1} + \mathtt{I_2}\lceil c/\mathtt{0}\rceil + \mathtt{I_2}\lceil c/\mathtt{0}\rceil)$$

Figura C.16: Funciones características de los ciclos del programa de la Figura C.15

$$1 + \tfrac{9}{10} \cdot \mathtt{I_2} \leq \mathtt{I_1}$$
$$1 + [\![\neg c == 1]\!] \cdot \mathtt{I_1} + [\![\neg c == \mathtt{1}]\!] \cdot (\mathtt{1} + \mathtt{I_2}\lceil c/\mathtt{0}\rceil + \mathtt{I_2}\lceil c/\mathtt{0}\rceil) \leq \mathtt{I_2}$$

Figura C.17: Obligaciones de prueba del programa de la Figura C.15

Para encontrar los invariantes, se buscará un punto fijo para cada función característica. Esto se logrará planteando la igualdad $F(\mathtt{I}) = \mathtt{I}$ y despejando $\mathtt{I}$. Por el Corolario 3.5 se obtendría un invariante correcto. La desventaja que tiene este enfoque es que no puede asegurar que este invariante sea el más ajustado posible, pero para los objetivos de esta investigación este es un inconveniente menor.

La ventaja de este enfoque es que el punto fijo de F_1 es sencillo. Al ser una función constante con respecto a $\mathtt{I_1}$ el único punto fijo posible es $1 + \frac{9}{10} \cdot \mathtt{I_2}$. El inconveniente de este enfoque es que, en principio, no es conocido el valor de la expresión $1 + \mathtt{I_2}\lceil c/\mathtt{0}\rceil + \mathtt{I_2}\lceil c/\mathtt{0}\rceil$. Para avanzar en este punto se necesita tomar alguna hipótesis. La hipótesis elegida es que la expresión anterior es igual a una constante racional K.

$$K = 1 + \mathtt{I_2}\lceil c/\mathtt{0}\rceil + \mathtt{I_2}\lceil c/\mathtt{0}\rceil \tag{C.1}$$

Al hacer esta suposición se está asumiendo que la única variable libre del invariante $\mathtt{I_2}$ es la variable c. Dado que no hay otra variable en la definición del programa, esta hipótesis cae dentro de lo razonable. Considerando este supuesto, se obtiene el siguiente sistema de ecuaciones no lineales:

$$\begin{cases} 1 + \frac{9}{10} \cdot \mathtt{I_2} & = \mathtt{I_1} \\ 1 + [\![\neg c == 1]\!] \cdot \mathtt{I_1} + [\![\neg c == \mathtt{1}]\!] \cdot K & = 10 \end{cases}$$

Figura C.18: Sistema de ecuaciones para encontrar los invariantes

Al resolver el sistema anterior se pueden obtener los invariantes buscados en función de la variable K, pero para ocupar los invariantes este valor debe ser conocido. Para encontrar el valor de esta variable hay que recurrir a la ecuación C.1 y al valor encontrado del invariante

$\mathtt{I_2}$

$$\mathtt{I_2} = \frac{1 + [\![\neg c == 1]\!] + K \cdot [\![c == 1]\!]}{\left(1 - \frac{9}{10} \cdot [\![\neg c == 1]\!]\right)}$$

Primero se debe evaluar los dos valores posibles de c en el invariante. Para $c = 0$ se tiene que

$$\mathtt{I_2}\lceil \mathtt{c/0} \rceil = \frac{1 + [\![\neg 0 == 1]\!] + K \cdot [\![0 == 1]\!]}{1 - \frac{9}{10} \cdot [\![\neg 0 == 1]\!]} = 20$$

y para $c = 1$ se tiene que

$$\mathtt{I_2}\lceil \mathtt{c/1} \rceil = \frac{1 + [\![\neg 1 == 1]\!] + K \cdot [\![1 == 1]\!]}{1 - \frac{9}{10} \cdot [\![\neg 1 == 1]\!]} = 1 + K$$

Por último, reemplazar en la ecuación C.1 y despejar K.

$$K = 1 + \frac{1}{2} \cdot 20 + \frac{1}{2}(K + 1) \Longrightarrow K = 23$$

Una vez encontrado el valor de K es posible obtener los valores de los invariantes buscados, pero en este punto surge otro problema. Al reemplazar el valor de K, el invariante $\mathtt{I_2}$(es análogo para $\mathtt{I_1}$) tiene el valor de

$$\mathtt{I_2} = \frac{1 + [\![\neg c == 1]\!] + 23 \cdot [\![c == 1]\!]}{1 - \frac{9}{10} \cdot [\![\neg c == 1]\!]}$$

Con la sintaxis abstracta actual del conjunto `RunTime` no es posible representar a los invariantes, ya que no está definido el constructor de la división (/). Esta situación provoca tener que simplificar de alguna manera los invariantes. Para hacer lo anterior, no hay una sola manera de proceder y a continuación se presentará un enfoque junto a su justificación.

Para simplificar los siguientes pasos se trabajará solo sobre $\mathtt{I_2}$ y luego, usando la primera igualdad de la Figura C.18 se obtendrá $\mathtt{I_1}$. Junto con enfocarse en un solo invariante, esta manera de proceder permite que la obligación de prueba asociada siempre sea válida. Esto último se tiene, ya que la expresión $1 + \frac{9}{10} \cdot \mathtt{I_2}$ es el punto fijo de F_1 y por el Corolario 3.5 es un invariante correcto.

Con la consideración anterior hecha, están las condiciones para seguir. La clave de la simplificación es notar que el denominador de $\mathtt{I_2}$ puede tener dos valores 1 o $\frac{1}{10}$. Dado esto, se obtendrán dos simplificaciones de $\mathtt{I_2}$, las que son presentadas en la Figuras C.19, C.20. Se insiste nuevamente que esta forma de proceder no es única. Se pudo intentar simplificar el invariante a través de constantes o tomar el valor esperado para aproximar el denominador etc. Se eligió la forma presentada, ya que genera dos invariantes interesantes y "respeta"la forma del invariante original.

$$\mathtt{I_2^{min}} = 1 + [\![\neg c == 1]\!] + 23 \cdot [\![c == 1]\!]$$
$$\mathtt{I_1^{min}} = \tfrac{19}{10} + \tfrac{9}{10} \cdot [\![\neg c == 1]\!] + \tfrac{207}{10} \cdot [\![c == 1]\!]$$

Figura C.19: Invariantes obtenidos al aproximar el denominador como 1

$$\mathtt{I}_2^{\mathtt{max}} = 10 + 10 \cdot [\![\neg c == 1]\!] + 230 \cdot [\![c == 1]\!]$$
$$\mathtt{I}_1^{\mathtt{max}} = 20 + 9 \cdot [\![\neg c == 1]\!] + 207 \cdot [\![c == 1]\!]$$

Figura C.20: Invariantes obtenidos al aproximar el denominador como $\frac{1}{10}$

Ya con la propuesta de invariantes hechas, se inicia el proceso de cálculo de tiempo de ejecución. Inicia analizando los invariantes $\mathtt{I}_\mathtt{i}^{\mathtt{max}}$, los que son correctos. Finaliza analizando con los invariantes $\mathtt{I}_\mathtt{i}^{\mathtt{min}}$. Uno de estos invariantes es incorrecto ($\mathtt{I}_2^{\mathtt{min}}$), lo que se demuestra con la herramienta desarrollada y con los cálculos manuales.

Con el fin de no extender innecesariamente el tamaño de los cálculos, la obligación de prueba asociada al invariante $\mathtt{I}_1$ es reemplazada por su equivalente $0 \leq 0$. Debido a la forma en que se procede la primera obligación siempre será $1 + \frac{9}{10} \cdot \mathtt{I}_2 \leq 1 + \frac{9}{10} \cdot \mathtt{I}_2$, por lo mismo, se prefiere presentar la versión minimal antes mencionada.

Programa probabilístico, tiempo de ejecución variable, con ciclo e invariante correcto ($\mathtt{C}_{\mathtt{pvc+}}$)

```
1 pwhile (< 9/10 >)
2       { pinv = 10 ++ 9 * *[c! = 1] ++ 207 ** [c == 1]}
3       {while (c == 1)
4            { inv = 10 ++ 10 ** [c! = 1] ++ 230 ** [c == 1]}
5            {c :~ 1/2 * < 0 > +1/2 * < 1 > }}
```

Figura C.21: Programa $\mathtt{C}_{\mathtt{pvc+}}$

◇ $(10 + 9 \cdot [\![\neg c == 1]\!] + 207 \cdot [\![c == 1]\!],$
$\{0 \leq 0,$
$1 + [\![\neg c == 1]\!] \cdot (10 + 9 \cdot [\![\neg c == 1]\!] + 207 \cdot [\![c == 1]\!]) + 131 \cdot [\![c == 1]\!]$
$\leq$
$10 + 10 \cdot [\![\neg c == 1]\!] + 230 \cdot [\![c == 1]\!]\})$

1 `pwhile` $(< 9/10 >)$

2 $\{$ `pinv` $= 10 \ +\!+ \ 9 \ **$ `[c!` $= 1] \ +\!+ \ 207 \ **$ `[c` $== 1]\}$

◇ $(10 + 10 \cdot [\![\neg c == 1]\!] + 230 \cdot [\![c == 1]\!],$
$\{1 + [\![\neg c == 1]\!] \cdot (10 + 9 \cdot [\![\neg c == 1]\!] + 207 \cdot [\![c == 1]\!]) + 131 \cdot [\![c == 1]\!]$
$\leq$
$10 + 10 \cdot [\![\neg c == 1]\!] + 230 \cdot [\![c == 1]\!]\})$

3 $\{$`while` $(c == 1)$

4 $\{$ `inv` $= 10 \ +\!+ \ 10 \ **$ `[c!` $= 1] \ +\!+ \ 230 \ **$ `[c` $== 1]\}$

◇ $(131, \ \emptyset)$

◇ $(1 + \frac{1}{2} \cdot (10 + 10 \cdot [\![\neg 0 == 1]\!] + 230 \cdot [\![0 == 1]\!])$
$+ \frac{1}{2} \cdot (10 + 10 \cdot [\![\neg 1 == 1]\!] + 230 \cdot [\![1 == 1]\!]), \ \emptyset)$

4 $\{c :\sim 1/2 * < 0 > +1/2 * < 1 > \}\}$

◇ $(10 + 10 \cdot [\![\neg c == 1]\!] + 230 \cdot [\![c == 1]\!], \ \emptyset)$

◇ $(0, \ \emptyset)$

Figura C.22: Cálculo `vcg[·]` manual - Programa $\mathtt{C}_{\mathtt{pvc+}}$

La primera obligación es $0 \leq 0$, esta es claramente válida, por lo mismo sólo se analiza la segunda obligación. Si de denota la segunda obligación como $\mathtt{O}^{\mathtt{max}}$ Su conjunto atómico es :

$$\mathcal{A}_{\mathtt{O}^{\mathtt{max}}} = \{c == 1\}$$

1.

Contexto	$= c == 1$
Restricción derivada	$= 1 + 0 \cdot (10 + 9 \cdot 0 + 207 \cdot 1) + 131 \cdot 1 \leq 10 + 10 \cdot 0 + 230 \cdot 1$
	$= 131 \leq 240$
Problema Lineal	$= (c == 1) \wedge \neg(131 \leq 240)$
	$= (x > 0) \wedge (x - 1 > 0) \wedge (131 > 240)$
	$= (c == 1) \wedge \mathtt{false}$
	$= \mathtt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

2.

Contexto	$= \neg c == 1$
Restricción derivada	$= 1 + 1 \cdot (10 + 9 \cdot 1 + 207 \cdot 0) + 131 \cdot 0 \leq 10 + 10 \cdot 1 + 230 \cdot 0$
	$= 20 \leq 20$
Problema Lineal	$= \neg(c == 1) \wedge \neg(0 \leq 0)$
	$= \neg(c == 1) \wedge (20 > 20)$
	$= \neg(c == 1) \wedge \mathtt{false}$
	$= \mathtt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

Conclusión No existe un contraejemplo para la obligación de prueba, por ende, el invariante $\mathtt{I}_2^{\mathtt{max}}$ es correcto.

```
Programa Analizado:
pwhile(<9/10>) {pinv = 10 ++ 9**[c!=1] ++ 207**[c==1]}{
      while(c == 1){inv = 10 ++ 10**[c!=1] ++ 230**[c==1]} {
            c:~ 1/2* <0> + 1/2* <1>}}

Se calcula la transformada con respecto a:
0

Tiempo de ejecución calculado:
10 ++ 9**([!(c == 1)]) ++ 207**([c == 1])

Obligaciones de prueba asociadas:

[1]
1 ++ 9/10**(10 ++ 10**([!(c == 1)]) ++ 230**([c == 1]))
:!<=:
10 ++ 9**([!(c == 1)]) ++ 207**([c == 1])
Es válida

[2]
1 ++ [!(c == 1)]<>(10 ++ 9**([!(c == 1)]) ++ 207**([c == 1]))
  ++ [c == 1]<>131
:!<=:
10 ++ 10**([!(c == 1)]) ++ 230**([c == 1]),
Es válida

El tiempo de ejecución calculado es válido
porque las obligaciones de prueba son válidas.

Análisis Finalizado.
```

Listing C.8: Cálculo $\mathsf{vcg}[\cdot]$ automatizado - Programa $\mathtt{C}_{\mathtt{pvc}+}$

Comparación entre los cálculos manuales y la herramienta desarrollada. La herramienta logra validar y calcular correctamente las obligaciones de prueba asociadas al programa. Al igual que en casos anteriores la simplificación sobre las obligaciones de prueba no es óptima, pese a esto, el *test* se considera aprobado.

Programa probabilístico, tiempo de ejecución variable, con ciclo e invariante correcto ($\mathtt{C}_{\mathtt{pvc-}}$)

```
1  pwhile (< 9/10 >)
2        { pinv = 19/10 ++ 9/10 ** [c! = 1] ++ 207/10 ** [c == 1]}
3        {while (c == 1)
4              { inv = 1 ++ [c! = 1] ++ 23 **[c == 1]}
5              {c :~ 1/2 * < 0 > +1/2 * < 1 > }}
```

Figura C.23: Programa $\mathtt{C}_{\mathtt{pvc-}}$

```
◇      (19/10 + 9/10 · [[¬c == 1]] + 207/10 · [[c == 1]],
       {0 ≤ 0,
       {1 + [[¬c == 1]] · (19/10 + 9/10 · [[¬c == 1]] + 207/10 · [[c == 1]]) + 14 · [[c == 1]]
       ≤
      1 + [[¬c == 1]] + 23 · [[c == 1]]})
1  pwhile (< 9/10 >)
2        { pinv = 19/10 ++ 9/10 **[c! = 1] ++ 207/10 **[c == 1]}
◇      (1 + [[¬c == 1]] + 23 · [[c == 1]],
       {1 + [[¬c == 1]] · (19/10 + 9/10 · [[¬c == 1]] + 207/10 · [[c == 1]]) + 14 · [[c == 1]]
       ≤
      1 + [[¬c == 1]] + 23 · [[c == 1]]})
3        {while (c == 1)
4              { inv = 1 ++ [c! = 1] +! + 23 * *[c == 1]}
◇              (14, ∅)
◇              (1 + 1/2 · (1 + [[¬0 == 1]] + 23 · [[0 == 1]])
                 + 1/2 · (1 + [[¬1 == 1]] + 23 · [[1 == 1]]), ∅)
4              {c :~ 1/2 * < 0 > +1/2 * < 1 > }}
◇              (1 + [[¬c == 1]] + 23 · [[c == 1]], ∅)
◇  (0, ∅)
```

Figura C.24: Cálculo vcg[·] manual - Programa $\mathtt{C}_{\mathtt{pvc-}}$

El conjunto atómico coincide con el del programa anterior, a continuación la validación de la segunda obligación de prueba.

1.

Contexto	$= c == 1$
Restricción derivada	$= 1 + 0 \cdot (\frac{19}{10} + \frac{9}{10} \cdot 0 + \frac{207}{10} \cdot 1) + 14 \cdot 1 \leq 1 + 0 + 23 \cdot 0$
	$= 14 \leq 24$
Problema Lineal	$= (c == 1) \wedge \neg(14 \leq 24)$
	$= (c ==!) \wedge (14 > 24)$
	$= (c == 1) \wedge \mathtt{false}$
	$= \mathtt{false}$
Modelo	= El problema no es satisfacible, no existe modelo válido.

	Contexto	$= \neg c == 1$
	Restricción derivada	$= 1 + 1 \cdot (\frac{19}{10} + \frac{9}{10} \cdot 1 + \frac{207}{10} \cdot 0) + 14 \cdot 0 \leq 1 + 1 + 23 \cdot 0$
		$= \frac{38}{10} \leq 2$
	Problema Lineal	$= \neg(c == 1) \wedge \neg(\frac{38}{10} \leq 22)$
2.		$= \neg(c == 1) \wedge (20 > 20)$
		$= \neg(c == 1) \wedge \texttt{true}$
		$= \texttt{true}$
	Modelo	= El problema es satisfacible, $c = 2$ es un modelo válido.

Conclusión El invariante $\mathtt{I}_2^{\mathtt{min}}$ no es correcto, ya que la obligación de prueba asociada no es válida. Un contraejemplo es $c = 2$.

$$
\begin{array}{rl}
 & 1 + [\![\neg 2 == 1]\!] \cdot (\frac{19}{10} + \frac{9}{10} \cdot [\![\neg 2 == 1]\!] + \frac{207}{10} \cdot [\![2 == 1]\!]) + 14 \cdot [\![2 == 1]\!] \leq \\
 & 1 + [\![\neg 2 == 1]\!] + 23 \cdot [\![2 == 1]\!] \\
= & 1 + 1 \cdot (\frac{19}{10} + \frac{9}{10} \cdot 1 + \frac{207}{10} \cdot 0) + 14 \cdot 0 \leq 1 + 1 + 23 \cdot 0 \\
= & \frac{38}{10} \leq 2 \\
= & \texttt{false}
\end{array}
$$

```
Programa Analizado:
pwhile(<9/10>) {pinv = 19/10 ++ 9/10**[c!=1] ++ 207/10**[c==1]}{
   while(c == 1){inv = 1 ++ [c!=1] ++ 23**[c==1]} {
         c:~ 1/2* <0> + 1/2* <1>}}

Se calcula la transformada con respecto a:
0

Tiempo de ejecución calculado:
19/10 ++ 9/10**([!(c == 1)]) ++ 207/10**([c == 1])

Obligaciones de prueba asociadas:
[1]
1 ++ 9/10**(1 ++ [!(c == 1)] ++ 23**([c == 1]))
:!<=:
19/10 ++ 9/10**([!(c == 1)]) ++ 207/10**([c == 1]),
Es válida
[2]
1 ++ [!(c == 1)]<>(19/10 ++ 9/10**([!(c == 1)]) ++ 207/10**([c == 1]))
  ++ [c == 1]<>14
:!<=:
1 ++ [!(c == 1)] ++ 23**([c == 1]),
No es válida

**************************************************************************

Obligación de prueba [2]
1 ++ [!(c == 1)]<>(19/10 ++ 9/10**([!(c == 1)]) ++ 207/10**([c == 1]))
  ++ [c == 1]<>14
:!<=:
1 ++ [!(c == 1)] ++ 23**([c == 1]),
La obligación de prueba tiene asociada 2 restricciones derivadas
 diferentes.
```

```
------------------------------------------------------------------------------

  Restricción derivada [2, 1]
  [!(c == 1)] |-  19/5 :!<=: 2

  La restricción no es válida.

  Un contraejemplo encontrado es:
  c = 2 Racional
------------------------------------------------------------------------------

  El tiempo de ejecución calculado no es válido
  porque alguna obligación de prueba no es válida.
  Ajuste los invariantes de ciclo y vuelva a realizar el análisis.

  Análisis Finalizado.
```

Listing C.9: Cálculo vcg[·] automatizado - Programa $\mathtt{C}_{\mathtt{pvc}-}$

Comparación entre los cálculos manuales la herramienta desarrollada. Al igual que en el caso anterior, existen diferencias en la representación de las obligaciones de prueba. Pese a esta situación, la herramienta logra encontrar un contraejemplo para la obligación. El *test* se considera aprobado.

C.1.9. Demostraciones de punto fijo

Demostración. La expresión que modela la iteraciones de punto fijo de $\mathtt{I}_2^{\mathtt{k}}$ es $\frac{3\cdot(2^k-1)}{2^{k-1}}$

- Para el caso base $\mathtt{k} = 0$: $\mathtt{I}_2^0 = \frac{3\cdot(2^0-1)}{2^{0-1}} = 0$.

- Para el caso inductivo $\mathtt{k} > 0$:

$$\begin{aligned}
F(\mathtt{I}_2^{\mathtt{k}}) &= \frac{5}{2} + \frac{1}{2}\cdot\left(1 + \frac{3\cdot(2^k-1)}{2^{k-1}}\right) \\
&= \frac{5}{2} + \frac{2^{k-1} + 3\cdot 2^k - 3}{2^k} \\
&= \frac{5\cdot 2^{k-1} + 2^{k-1} + 3\cdot 2^k - 3}{2^k} \\
&= \frac{6\cdot 2^{k-1} + 6\cdot 2^{k-1} - 3}{2^{k+1}} \\
&= \frac{12\cdot 2^{k-1} - 3}{2^{k+1}} \\
&= \frac{3\cdot 2^{k+1} - 3}{2^{k+1}} \\
&= \frac{3(\cdot 2^{k+1} - 1)}{2^{k+1}} \\
&= \mathtt{I}_2^{\mathtt{k+1}}
\end{aligned}$$

□

Demostración. La expresión que modela la iteraciones de punto fijo de $\mathtt{I}_1^{\mathtt{k}}$ es $\frac{9\cdot(2^k-1)}{2^k}$

- Para el caso base $\mathtt{k} = \mathtt{0}$: $\mathtt{I}_1^{\mathtt{0}} = \frac{9\cdot(2^0-1)}{2^0} = \mathtt{0}$.

- Para el caso inductivo $\mathtt{k} > \mathtt{0}$:

$$\begin{aligned}
(\mathtt{I}_1^{\mathtt{k}}) &= 4 + \frac{1}{2}\cdot\left(1 + \frac{9\cdot(2^k-1)}{2^k}\right) \\
&= 4 + \frac{2^k + 9\cdot 2^k - 9}{2^{k+1}} \\
&= \frac{8\cdot 2^k + 2^k + 9\cdot 2^k - 9}{2^{k+1}} \\
&= \frac{9\cdot 2^{k+1} - 9}{2^{k+1}} \\
&= \frac{9\cdot(2^{k+1}-1)}{2^{k+1}} \\
&= \mathtt{I}_1^{\mathtt{k+1}}
\end{aligned}$$

□