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

INFORME FINAL DE TEMA DE MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

LUIS PINOCHET

PROFESOR GUÍA:
FEDERICO OLMEDO

SANTIAGO DE CHILE
2021

1. Introducción

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. En áreas como *machine learning* o criptografía se ocupa activamente este tipo de algoritmos.

Estos algoritmos son representados a través de programas probabilísticos, para luego poder ser ejecutados por un computador. Un programa probabilístico es simplemente un programa descrito con las construcciones estándar de los lenguajes de programación, a las cuales se le agrega una construcción adicional que permiten tomar muestras de distribuciones de probabilidad [5].

Un ejemplo de programa probabilístico es el mostrado en la Figura 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, esta representa a una muestra de una distribución Bernoulli con parámetro p. Le asigna la probabilidad de $1/2$ a true y $1/2$ a false(lo que se representa como $1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle$). Si el resultado observado es true el programa termina inmediatamente, asignándole el valor true 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 true a la variable succ; por el contrario, si el resultado observado es false se le asigna ese valor a succ terminando el programa.

Figura 1: Programa probabilístico C_{trunc} que representa una distribución geométrica truncada (hasta el lanzamiento de la segunda moneda).

```
if (1/2 · ⟨true⟩ + 1/2 · ⟨false⟩)
      {succ := true}
else {
      if (1/2 · ⟨true⟩ + 1/2 · ⟨false⟩)
            {succ := true}
      else
            {succ := false}
      }
```

Dada la relevancia de los programas probabilísticos, es fundamental poder estimar de manera eficiente su tiempo de ejecución. Para lograr este objetivo se han propuesto diversos enfoques. Por ejemplo, Chakarov y Sankaranarayanan [3] abordan el problema basándose en teoría de martingalas. En este trabajo se sigue la técnica de la transformada $\mathsf{ert}[\cdot]$ [6]. 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 es más expresivo y permite razonar con mayor precisión sobre los programas que la técnica basada en martingalas.

En este modelo el tiempo de ejecución se representa a través de una función $f\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, dónde Σ es el conjunto de los estados 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, $\mathsf{ert}[\mathsf{C}](f)$ 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 $f \equiv \mathbf{0}$.

Antes de ver un ejemplo del uso de la transformada $\mathsf{ert}[\cdot]$, presentamos el modelo de costo adoptado: Asumiremos que cada asignación y la evaluación de cada guarda (tanto de un condicional `if-then-else` como de un bucle `while`) consume 1 unidad de tiempo.

De manera ilustrativa, presentamos el cálculo de $\mathsf{ert}[C_{trunc}\,](\mathbf{0})$ para determinar el tiempo de ejecución de C_{trunc}:

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

Se llega a la conclusión que C_{trunc} tiene un tiempo de ejecución promedio de 2.5 unidades de tiempo. Tal como se aprecia, esta técnica sigue de forma natural la estructura del código.

Como se menciona anteriormente los tiempos de ejecución según este modelo son funciones. En el caso desarrollado arriba $\frac{\mathbf{5}}{\mathbf{2}}$ es una función constante (ya que el tiempo de ejecución del programa es independiente del estado inicial del cual se ejecuta).

2. Situación Actual

En la actualidad se encuentra establecida la teoría de la técnica. Esta incluye principalmente, la definición del lenguaje pProg, la definición de la transformada ert[·] y por último la definición de diferentes técnicas que permiten sobrellevar el problema de calcular la transformada para un ciclo `while`. El problema de los ciclos será mencionado más adelante junto a la teoría e inconvenientes de la técnica que estamos trabajando.

La sintaxis del lenguaje pProg está definida de la siguiente forma:

Figura 2: Sintaxis del lenguaje probabilístico pProg.

C :=	`empty`	programa vacío
	x:≈ μ	asignación probabilista
	C ; C	composición secuencial
	C □ C	elección no - determinista
	`if` (ξ) {C} `else` {C}	condicional
	`while` (ξ) {C}	ciclo

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

El principal inconveniente de esta definición es que que agrupa en una misma estructura a las guardas probabilistas y deterministas, como consecuencia de esto se agrupa también en un mismo constructor los `if` (respectivamente `while`) deterministas y probabilistas. Se retomará este punto después de definir la transformada ert[·]:

Tabla 1: Definición de la transformada ert[·].

C	ert[C](f)
`empty`	f
x:≈ μ	$\mathbf{1} + \lambda\sigma.\mathbf{E}_\mu(\lambda \mathsf{v}.\ f\ [\mathsf{x}/\mathsf{v}](\sigma))$
C_1; C_2	ert[C_1](ert[C_2](f))
C_1 □ C_2	max{ert[C_1](f), ert[C_2](f)}
`if` (ξ) (C_1) `else` (C_2)	$\mathbf{1}$ + [ξ:true]·ert[C_1](f) + [ξ:false]·ert[C_2](f)
`while` (ξ) {C}	lfp$X.\mathbf{1} + [\xi : \mathsf{false}] \cdot f + [\xi : \mathsf{true}] \cdot$ ert[C](f)

Recordemos que $\mathsf{ert}[\mathsf{C}](f)$ 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.

En el cálculo 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 $\sigma \in \Sigma, f[\mathsf{x}/\mathsf{v}](\sigma) \triangleq f(\sigma[\mathsf{x}/\mathsf{v}])$, dónde $\sigma[\mathsf{x}/\mathsf{v}]$ 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.

En 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 C_2).

En la transformada de elección no-determinista se toma un punto de vista demoniaco. Esto significa que se toma un camino pesimista dónde se elige el mayor de los dos tiempos de ejecución de las dos ramas.

En la condiciones se suma **1** por la evaluación de la guarda. La notación $[\xi{:}\mathsf{true}]$ 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 $\mathsf{ert}[\mathsf{while}](\cdot)$, $\mathbf{lfp}X\mathbf{.}\ F(X)$ representa el menor punto fijo de la transformada $F : \mathbb{T} \to \mathbb{T}$.

Retomando el problema mencionado después de la definición del lenguaje pProg, si ξ es una guarda probabilista (lanzamiento de una moneda), la interpretación de ξ es directa y es el valor p de la distribución Bernoulli, pero si ξ es determinista, por ejemplo, $\mathsf{x} + \mathsf{y} = \mathsf{0}$, se interpreta como una indicatriz que toma los valores de 0 o 1 (dependiendo si la expresión es verdadera o falsa). Dado estos comportamiento tan dispares es claro que se debe pensar en dos estructuras diferentes para las guardas probabilistas y las deterministas.

Otra limitación es que debido a la definición inductiva de la transformada, los cálculos de esta se vuelven engorrosos y largos. Un ejemplo de lo anterior es la transformada de la composición secuencial de dos programas la cual crece rápidamente en tamaño debido a que la estructura esencial de un programa es la ejecución secuencial de sentencias (una asignación, seguida de una condición `if`, seguido de un ciclo `while`, etc.), por lo que es fundamental poder automatizar este proceso para que esta técnica sea usada en la práctica.

Por otro lado los ciclos representan en sí un desafío especial, ya que como se menciona en la anteriormente la definición de la transformada de un `while` implica el cálculo del menor punto fijo de un transformador de tiempos de ejecución, por lo que obtener el resultado buscado 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 bucles. Estos invariantes son tiempos de ejecución $\in \mathbb{T}$ 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 $f \leq f'$. A continuación se introduce la definición formal de invariante:

Definicion 1 Sea $f \in \mathbb{T}$, $\mathsf{C} \in \mathsf{pProg}$ y ξ una distribución sobre los booleanos. Se dice que $I \in \mathbb{T}$ es *invariante* del ciclo $\texttt{while}\ (\xi)(\mathsf{C})$ con respecto a f si y solo si

$$1 + [\xi : \mathsf{false}] \cdot f + [\xi : \mathsf{true}] \cdot \mathsf{ert}[\mathsf{C}](I) \ \leq \ I$$

A la condición anterior se le una el siguiente teorema:

Teorema 1 Sea $f \in \mathbb{T}$, $\mathsf{C} \in \mathsf{pProg}$ y ξ una distribución sobre los booleanos. Si $I \in \mathrm{T}$ es un invariante de $\texttt{while}\ (\xi)(\mathsf{C})$ con respecto a f entonces

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

Con el condición y teorema antes introducidos podemos establecer las cotas a los tiempos de ejecución de los ciclos. Esto agrega un nivel de complejidad adicional al cálculo de tiempo de ejecución de los programas, ya que se requiere asociarle al ciclo un *invariante* externo al cálculo.

Por último agregar que estos *invariantes* deben ser propuestos (son un *input* del análisis) y aunque el trabajo original describe brevemente una técnica basada en plantillas para la síntesis de invariantes, estas siguen requiriendo de significativo esfuerzo. Dado lo anterior es conveniente una herramienta que automatice la generación de invariantes de bucles. Este punto sin embargo no se abordará en la memoria, sino que se dejará para un trabajo futuro.

Dado los puntos anteriores es clara la necesidad de mecanizar de alguna manera este proceso, para permitir alguna forma de automatización del cálculo del tiempo de ejecución de programas probabilísticos.

3. Objetivos

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 presentada por Kaminski et al[6]. . En particular, la herramienta va a tomar un programa donde cada bucle 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 bucles se espera calcular con exactitud el tiempo estimado de ejecución del mismo.

Objetivos Específicos

Generación del conjunto de restricciones. Dado un programa con las anotaciones de *invariantes* respectivas y un tiempo de ejecución candidato f, se busca generar el conjunto de restricciones que debe satisfacer f para que sea efectivamente una cota superior del tiempo de ejecución del programa.

Verificación del conjunto de restricciones. En este punto se busca verificar que el tiempo de ejecución f propuesto efectivamente satisfaga el conjunto de restricciones generados en el apartado anterior. Para ello se plantea el uso de *SMT solvers*.

Síntesis de los tiempos de ejecución. En este punto se busca dar el siguiente paso y poder sintetizar el menor tiempo de ejecución f que sea solución de las restricciones generadas en el primer apartado.

Validación. Hacia el final se propone validar que la solución desarrollada funcione como un conjunto y cumpla el objetivo general propuesto anteriormente. Se propone hacer diversas comparaciones entre el tiempo de ejecución retornado por la herramienta para un programa C y el tiempo de ejecución esperado teórico $\mathsf{ert}[\mathsf{C}](f)$. En un comienzo se probará programas más bien minimalistas y progresivamente se aumentará la complejidad de estos. Entre los programas a analizar estarán los casos de estudio desarrollados en los trabajos que serán revisados.

4. Plan de Trabajo

Una forma de ver el problema planteado es interpretándolo como uno de optimización:

$$\begin{aligned} &\text{minimizar} \quad h \\ &\text{sujeto a} \quad f_i \leq g_i, \qquad i = 1, \ldots, m \ , \end{aligned}$$

donde $f_i, g_i, h \in \mathbb{T}$ son tiempos de ejecución, es decir, funciones que mapean estados de programa a reales no-negativos y se obtienen como resultado de la aplicación del transformador $\mathsf{ert}[\cdot]$, a partir de las anotaciones de invariantes de ciclo del programa original.

Para alcanzar la solución de este problema se propone la utilización de diferentes métodos y herramientas que serán descritos a continuación.

En un comienzo se debe revisar en extenso más ejemplos de la aplicación de la transformada, para esto se estudiará de forma profunda la versión extendida del trabajo original[7] y diversos artículos en dónde se presenta la técnica. Fruto de estos estudios se espera adquirir las habilidades necesarias para identificar una representación adecuada para los cálculos de $\mathsf{ert}[\cdot]$. En particular se desea estudiar más ejemplos relacionados a los ciclos `while`, ya que representan el tópico más desafiante del trabajo.

En cuanto a la implementación del código que resolverá el problema, es claro que se debe invertir esfuerzos en la investigación de las distintas herramientas. En particular, se propone utilizar *SMT solvers* para este propósito. Estas son herramientas que dado una teoría modular subyacente, verifican si una serie de restricciones tienen una solución admisible. Existen varias opciones en este campo, por ejemplo `CVC4`[2]o `Z3`[4]. Durante esta etapa del trabajo se ocupó `Z3` en su versión SMT-LIB 2.0[1], si bien se lograron los resultados esperados, sería conveniente considerar las API's que ofrece para distintos lenguaje imperativos. Esto le podría dar un grado extra de flexibilidad que podría ser útil según progrese el desarrollo y aparezcan diferentes problemas.

Con respecto a las restricciones, una posibilidad es llevar a cabo la síntesis en el mismo *solver*, mientras otra posibilidad es hacerlo a través de una herramienta programada en un lenguaje de propósito general (dada la estructura inductiva de la transformada $\mathbf{ert}[\cdot]$, resulta conveniente el uso de un lenguaje funcional, por ejemplo `Haskell`, dado el conocimiento previo del memorista).

En cuanto a la verificación de estas restricciones, es claro que los *solvers* antes mencionados son la principal opción. Para poder usar estos *solvers* se debe primero elegir que teoría ocupar. Durante este trabajo se ha investigado y ocupado la teoría de los reales. Con ella en mente se debe poder transformar todo tiempo de ejecución en una expresión de tipo real , de esta manera se podrán comparar fácilmente y como consecuencia se decidirá si un tiempo de ejecución cumple o no las restricciones impuestas.

Un punto relacionado con la anterior es la estructura que puede tomar un tiempo de ejecución. En principio estos pueden tomar una forma arbitraria. Es claro que se deben invertir esfuerzos en acotar la clase de tiempo de ejecución que vamos a soportar, de tal manera de que sea factible razonar sobre ellos de manera automática, a través de las teorías provistas por los *solvers*. En particular, gran parte de los algoritmos implementados en los *solvers* responden bien a la aritmética sobre expresiones lineales, por lo que creemos que esa debería ser la forma óptima. En este trabajo se creó una gramática con la cual se les dio a los tiempos de ejecución una forma cercana a la forma lineal.

Llegando al punto de la síntesis del menor tiempo de ejecución, lo mas factible es seguir usando el poder de los *solver*. Existen implementaciones orientadas a la optimización en `Z3`. Es de esperar que estas implementaciones sean de ayuda sobre todo si se sigue trabajando con expresiones lineales.

Finalmente, con el fin de simplificar el trabajo se considerará un lenguaje con distribuciones de probabilidad sobre los reales y de soporte finito , y posteriormente se analizará la factibilidad de soportar distribuciones discretas de soporte (infinito) numerable.

5. Trabaja Adelantado

El trabajo adelantado consiste en la resolución de un problema en concreto:
Calcular $\mathsf{ert}[C_{trunc}\,](\mathbf{0})$ el tiempo de ejecución del programa C_{trunc} de la Figura 1, y verificar que esté debajo de una cota superior f dada.

Si bien C_{trunc} no posee ciclos, esto no significa que el desarrollo sea directo, ya que se debe resolver una serie de problemas antes de poder calcular su transformada y luego compararla con otro tiempo de ejecución.

Para llegar a la solución del problema planteado, se debió primero acotar la forma de todas las estructuras involucradas (expresiones aritméticas, programas y tiempos de ejecución), de tal manera que fuese posible razonar sobre ellas de manera automática, usando las teorías provistas por los SMT *solvers*, además se investigó y utilizó las principales funcionalidades de `Z3`, el SMT *solver* elegido, llegando a tener una representación de la estructuras antes mencionadas.

El primer paso es adaptar la sintaxis de los lenguajes probabilistas. Dado las observaciones hechas anteriormente en la Sección 2 se decide separar tanto a los ciclos `while` como a las condiciones `if` en sus versiones deterministas y probabilistas.

Figura 3: Sintaxis refinada del lenguaje probabilístico pProg.

C :=	`empty`	programa vacío
	x:≈ μ	asignación probabilista
	C ; C	composición secuencial
	C □ C	elección no - determinista
	`if` $(d\xi)${C}`else` {C}	condicional determinista
	`pif` $(p\xi)${C}`else` {C}	condicional probabilista
	`while` $(d\xi)${C}	ciclo determinista
	`pwhile` $(p\xi)${C}	ciclo probabilista

La guarda probabilista $p\xi$ representa una muestra de la distribución Bernoulli con probabilidad `p` de `true`, $d\xi$ a una expresión booleanas determinista que involucra expresiones aritméticas y μ representa las expresiones aritméticas probabilistas, es decir, un dado de `n` caras con distribución sobre los reales.

El siguiente paso hecho fue definir el lenguaje de las expresiones aritméticas. Este lenguaje es la base de las de las demás estructuras, por lo que es importante tener una representación clara de la misma.

Figura 4: Gramática de expresiones aritméticas.

$\mathsf{Arit} :=$	n	número real
	x	variable
	$\mathsf{n} * \mathsf{Arit}$	ponderación por constante
	$\mathsf{Arit} + \mathsf{Arit}$	suma de expresiones aritméticas
	$\mathsf{Arit} - \mathsf{Arit}$	resta de expresiones aritméticas

Como se aprecia las expresiones generadas serán lineales con respecto a las variables. Esto tienen directa relación con el hecho de ocupar Z3, ya que como se mencionó antes, sus teorías presentan soporte para el razonamiento sobre expresiones lineales.

Con las expresiones aritméticas ya definidas, se puede definir las expresiones boolenas deterministas:

Figura 5: Gramática de expresiones booleanas deterministas.

$d\xi :=$	$\mathsf{true}, \mathsf{false}$	constantes booleanas
	$\mathsf{Arit} = \mathsf{Arit}$	igualdad de expresiones aritméticas
	$\mathsf{Arit} \leq \mathsf{Arit}$	desigualdad de expresiones aritméticas
	$\neg d\xi$	negación
	$d\xi \wedge d\xi$	conjunción

Por otro lado las expresiones booleanas probabilistas son más simples, como se ha mencionada antes es una muestra una distribución Bernoulli.

Figura 6: Gramática expresiones booleanas probabilistas.

$p\xi :=$	$\mathsf{Ber}(\mathsf{p})$	distribución Bernoulli con $\mathsf{p} \in [0, 1]$

Para las asignaciones se define las expresiones aritméticas probabilistas, en este trabajo se consideró distribuciones sobre los reales.

Figura 7: Gramática expresiones aritméticas probabilistas.

$\mu :=$	$\Sigma_1^{\mathsf{n}} \mathsf{p}_i \cdot \mathsf{a}_i$	dado de n caras, con $\mathsf{p}_i \in [0, 1]$, $\Sigma_1^n \mathsf{p}_i = 1$ y a_i constante $\in \mathbb{R}$

El último lenguaje definido a continuación es el de los tiempos de ejecución; recordar que estos son funciones que toman como input un estado σ y retornan un real positivo.

Figura 8: Gramática de tiempos de ejecución.

RunTime :=	Arit	expresión aritmética
	$\mathsf{n} * \mathsf{RunTime}$	ponderación por constante numérica n
	$\mathbb{1}_{d\xi} \cdot \mathsf{RunTime}$	multiplicación por una indicatriz
	$\mathsf{RunTime} + \mathsf{RunTime}$	suma de expresiones aritméticas
	$\mathsf{RunTime} - \mathsf{RunTime}$	resta de expresiones aritméticas
	$\mathsf{RunTime}\ [\mathsf{x}/\mathsf{n}]$	sustitutción de variable por constante numérica

Con todos los lenguajes ya definidos e implementados están las condiciones para definir la transformada de tiempos de ejecución para las instrucciones en C_{trunc}:

Tabla 2: Definición de la transformada $\mathsf{ert}[\cdot]$ para las instrucciones presentes en C_{trunc}.

C	$\mathbf{ert}[\mathrm{C}](f)$
`empty`	f
$\mathsf{x} :\approx \Sigma_1^{\mathsf{n}} \mathsf{p}_i \cdot \mathsf{a}_i$	$\mathbf{1} + \Sigma_1^{\mathsf{n}} \mathsf{p}_i * f\ [\mathsf{x}/\mathsf{a}_i]$
`pif(Ber(p)) {`C_1`} else {`C_2`}`	$\mathbf{1} + \mathsf{p} * \mathsf{ert}[\mathsf{C}_1](f) + (1 - \mathsf{p}) * \mathsf{ert}[\mathsf{C}_2](f)$

Si bien estas fueron las transformadas ocupadas vale la pena mencionar el caso del condicional determinista.

$$\mathsf{ert}[\texttt{if}(d\xi)\ \{C_1\}\ \texttt{else}\ \{C_2\}](f) \ = \ \mathbf{1} + \mathbb{1}_{d\xi} \cdot \mathsf{ert}[\mathsf{C}_1](f) + \mathbb{1}_{\neg d\xi} \cdot \mathsf{ert}[\mathsf{C}_2](f)$$

Como se mencionó en la Sección 2 a la hora de aplicar la transformada el comportamiento de esta, sobre los condiciones cambia totalmente según su tipo. Para el caso probabilista la expresión $[\xi]$ que se encontraba en la definición original de la transformada $\mathsf{ert}[\cdot]$, tiene una interpretación como directa como una constante. Para el caso probabilista se interpreta como una indicatriz (de ahí el cambio de notación).

Ya con estas definiciones hechas, se procedió a a estudiar la sintaxis de los smt-solvers. De todas las opciones disponibles se escogió `Z3`, debido a la gran cantidad de material disponible. En particular de todas las posibles implementaciones de `Z3` se eligió la que usa notación SMT 2.0, debido a que es una notación estandar y sería posible de ejecutar en otro *solver*. Una vez familiarizado con la herramienta, se implementaron los lenguajes antes mencionados.

En este punto queda encontrar el mecanismo para poder comparar este resultado con otro tiempo de ejecución. Como se mencionó antes un *smt-solver* tiene asociado una o varias teorías subyacentes que permiten al usuario encontrar soluciones a cierto tipo de problemas. Es claro que la teoría mas cercana es la de los Reales y como se mencionó antes, esta teoría tiene gran alcance cuando los problemas son de tipo lineal.

Para el caso del programa C_{trunc}, si se logra convertir un tiempo de ejecución a una expresión real, se podrá comprar con otro tiempo de ejecución y decidir si uno es cota de otro.

Este procedimiento se logra llevando las expresiones de tipo `RunTime` a tipo `Arit`, para finalmente llevarlo a una constante de tipo real. Notar que para programas cuyo tiempo de

ejecución dependan del estado inicial, los tiempos de ejecución se deberán llevar a expresiones reales simbólicas, donde las variables de programa estarán representadas por variables simbólicas (variables "matemáticas").

Este es el punto en el que se encuentra el trabajo, se puede calcular el tiempo de ejecución del programa C_{trunc} y de otros programas similares. La implementación en `Z3` puede descargarse desde el siguiente link. La principal limitante de la implementación actual para que sea una solución completa es tener no la transformada completamente definida, ya que se deja fuera a los ciclos debido a su dificultad y al `if` determinista debido a que tiene la forma de $\mathbb{1}_{d\xi} \cdot \mathsf{RunTime}$, lo que es una expresión no lineal. Claramente se debe buscar una forma de *linealizar* el problema.

Si bien existen estas limitantes, es claro que se pudo plasmar la idea principal de la técnica al computador. El poder tomar programas y convertirlos en reales es muy representativo de la intención inicial del trabajo original, por lo hay indicios de que este trabajo será factible.

6. Cronograma

El cronograma está pensado en trabajar constantemente en el escrito final, pero despejando las semanas finales para concentrarse en el informe final.

Generación del conjunto de restricciones Dado que es el hito inicial, seguramente se necesitaŕa corregir algunos puntos del semestre anterior y además contiene el punto más difícil que el trato de los ciclos `while`. Se estima terminar el hito en 6 semanas.

Verificación del conjunto de restricciones En este punto debería ser menos exigente que el anterior, ya que la tarea principal es poder comparar dos tiempos de ejecución. Se estima terminar el hito en 2 semanas.

Síntesis de los tiempos de ejecución En este punto se espera unir la implementación propia con los diferentes algoritmos de optimización disponibles en los solvers, es posible que aparezcan inconsistencias. Se estima terminar el hito en 3 semanas.

Validación Se espera no encontrar *bugs* importantes, por lo que el trabajo se centraría en arreglar problemas menores y mejorar el código que se tenga. Se estima terminar en 2 semanas.

Redacción Se espera tener el código terminado, se espera ocupar el tiempo en terminar el informe y documentar lo implementado. Se estima terminar este hito en 2 semanas *body*.

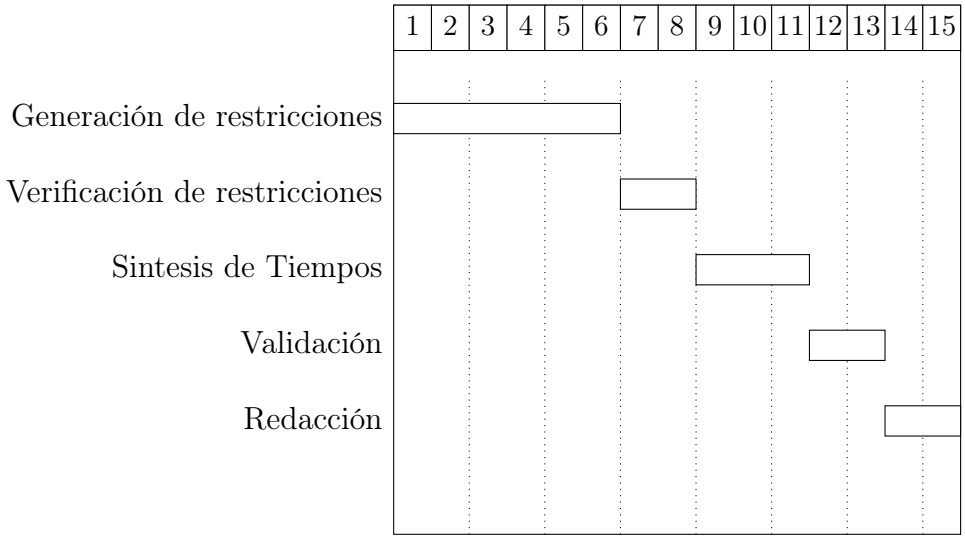

Figura 9: Cronograma

Referencias

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

[2] Barrett, Clark W., Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds y Cesare Tinelli: *CVC4*. En Gopalakrishnan, Ganesh y Shaz Qadeer (editores): *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volumen 6806 de *Lecture Notes in Computer Science*, páginas 171–177. Springer, 2011. `https://doi.org/10.1007/978-3-642-22110-1_14`.

[3] Chakarov, Aleksandar y Sriram Sankaranarayanan: *Probabilistic Program Analysis with Martingales*. En Sharygina, Natasha y Helmut Veith (editores): *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volumen 8044 de *Lecture Notes in Computer Science*, páginas 511–526. Springer, 2013. `https://doi.org/10.1007/978-3-642-39799-8_34`.

[4] De Moura, Leonardo y Nikolaj Bjørner: *Z3: An Efficient SMT Solver*. En *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, página 337–340, Berlin, Heidelberg, 2008. Springer-Verlag, ISBN 3540787992.

[5] Gordon, Andrew D., Thomas A. Henzinger, Aditya V. Nori y Sriram K. Rajamani: *Probabilistic Programming*. En *Future of Software Engineering Proceedings*, FOSE 2014, página 167–181, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 9781450328654. `https://doi.org/10.1145/2593882.2593900`.

[6] Kaminski, Benjamin Lucien, Joost Pieter Katoen, Christoph Matheja y Federico Olmedo: *Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs*. En Thiemann, Peter (editor): *Programming Languages and Systems*, páginas 364–389, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg, ISBN 978-3-662-49498-1.

[7] Kaminski, Benjamin Lucien, Joost-Pieter Katoen, Christoph Matheja y Federico Olmedo: *Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs*. CoRR, abs/1601.01001, 2016. `http://arxiv.org/abs/1601.01001`.