Universidad Nacional de Rosario

Departamento de Ciencias de la Computación

EDSL en Haskell para la programación segura respecto a la propiedad Delimited Release

Tesina de grado

Autor:
Gonzalo de Latorre

Directora:
Cecilia Manzino

11 de mayo de 2022

1. Resumen

La confidencialidad de la información manipulada por sistemas informáticos ha tomado mayor importancia con el uso creciente de aplicaciones a través de internet. Asegurar que no se filtre información confidencial se ha tornado un problema muy complejo debido a que cada vez es más sencillo el acceso a los datos por parte de los usuarios, y cada vez son más complejos los volúmenes de datos que manejan estos sistemas.

Los mecanismos de seguridad tradicionales como control de acceso o criptografía no proveen protección punta a punta [29] de los datos: funcionan eficientemente en limitar su acceso, pero no pueden hacer nada para evitar su propagación. Por ejemplo, mediante un mecanismo de control de acceso, una vez que un usuario tiene permiso para acceder a la información, no hay garantías acerca de lo que este usuario pueda hacer con ella. Similarmente, mediante algoritmos criptográficos, una vez que los datos son desencriptados, queda fuera de alcance de este la manipulación de los datos por parte del usuario.

Para complementar estos mecanismos de seguridad, surgen las técnicas de control de flujo de información (IFC, Information-Flow Control) [26] [2], las cuales permiten establecer garantías sobre la confidencialidad e integridad de los datos. Mediante estas técnicas se analiza cómo fluye la información dentro del programa que la utiliza. En este contexto surgen políticas de confidencialidad que garantizan que la información confidencial no puede ser inferida a partir de los datos públicos. No-interferencia [9] es un ejemplo de una política de seguridad. Lo interesante de esta propiedad es que puede ser chequeada de manera estática mediante un sistema de tipos [26][33], por lo tanto, cuando un programa tipa en ese sistema de tipos, significa que satisface la propiedad de seguridad. Esta propiedad se define como una condición semántica que asegura que no ocurrirá un flujo ilícito de información durante la ejecución de un programa, no permitiendo distinguir los resultados de dos computaciones que solo varían en sus argumentos confidenciales.

El estudio de lenguajes seguros para flujos de información tiene una historia extensa con resultados principalmente teóricos. Una de las dificultades del uso de sistemas de tipos que garantizan no-interferencia es que la mayor parte de los programas de uso masivo muchas veces necesitan revelar información confidencial como parte de su funcionalidad y la no-interferencia resulta ser demasiado restrictiva para estos casos. Para que los lenguajes de seguridad tengan utilidad práctica necesitamos mecanismos de desclasificación, en los cuales el flujo de información sea controlado y al mismo tiempo se permita liberar información confidencial a canales públicos, pero solo de manera permitida y controlada.

Existen muchos trabajos sobre diferentes mecanismos de desclasificación [5][24][20],

sin embargo, generalmente son pocas o nulas las garantías acerca de que es lo que realmente se está desclasificando, en consecuencia estos lenguajes son vulnerables a ataques en los que se explotan los mecanismos de desclasificación para mostrar más información de la que es liberada.

Sabelfeld y Myers [27] introducen una nueva propiedad de seguridad a la que denominan *delimited release*, que garantiza que la desclasificación no puede explotarse para construir ataques. Junto a la definición definen un sistema de tipos de seguridad que garantiza que cualquier programa tipable satisface la propiedad.

El objetivo de esta tesina es desarrollar un lenguaje de dominio específico [8][10] embebido en Haskell para escribir programas seguros respecto a la propiedad *delimited release*. Para la misma se usarán algunas extensiones de Haskell que permiten una forma de programar con tipos dependientes. Utilizando estas extensiones que posibilitan aumentar la expresividad del sistema de tipos de Haskell, es factible representar como invariante en los tipos de los términos del lenguaje embebido la propiedad *delimited release*, de manera que si un programa tipa en Haskell significa que es seguro respecto a la propiedad de seguridad.

Índice

2. Introducción

El objetivo principal de la tesina es la creación de un EDSL en Haskell para programar en un lenguaje imperativo simple, tipo *while* y seguro respecto a la propiedad *delimited release*. Para ello nos basaremos en el sistema de tipos presentado en [27] y utilizaremos *GADTs*, *type families* y otras extensiones de Haskell para codificar la información sobre los tipos de seguridad en los tipos de los términos del lenguaje embebido. Al codificar esta información en los tipos, los términos son seguros por construcción, ya que el sistema de tipos de Haskell solo tipará programas seguros. Lograr que un usuario de este lenguaje pueda programar de manera sencilla, sin más conocimientos que el dominio en sí del lenguaje, es una de las metas de este trabajo.

Teniendo en cuenta las ventajas de definir un lenguaje embebido en lugar de crear un lenguaje desde cero, claramente una solución más costosa, es que en esta tesina proponemos definir un EDSL en Haskell para la programación segura respecto de la propiedad. Creemos que este lenguaje es el mejor huésped para nuestros propósitos, por su poderoso sistema de tipos, el cual nos permitirá programar de manera cercana con tipos dependientes y por ser también un lenguaje de propósito general. Sumamos a las características estándares del lenguaje algunas extensiones que nos permitirán expresar invariantes dentro del tipo de los programas.

Esta tesina está estructurada de la siguiente manera: en la Parte I, haremos una breve introducción de las principales líneas de investigación relativas a seguridad y una breve descripción de las extensiones de Haskell que nos permiten simular tipos dependientes. En la Parte 2, incluimos nuestro aporte, las conclusiones y trabajos futuros. En el Anexo encontraremos las demostraciones e incluiremos la estructura de los archivos junto a una breve explicación de sus contenidos.

En este trabajo se incluyen extractos del código con sus explicaciones correspondientes. El código completo y ejemplos de programas pueden verse en `https://github.com/gonzalodelatorre/delimitedRelease`.

Parte I

Preliminares

3. Seguridad

Cómo proteger la privacidad de información dentro de un sistema es un problema muy estudiado desde hace muchos años, pero muy en auge en la actualidad, donde nunca la información fue tan accesible. Desde este punto de partida, el principal desafío es garantizar que la información sensible sea solo accedida por partes autorizadas y necesariamente para cumplir este objetivo se debe tener el control sobre la forma en que la información se propaga dentro del sistema.

Cabe destacar que realizar un análisis manual no es factible, hoy en día los volúmenes de datos manejados por los sistemas informáticos son masivos, así como también su complejidad. Bajo este escenario, ¿cómo podemos prevenir una filtración de datos privados?. Actualmente, si bien existen soluciones prometedoras, no existe una solución única capaz de garantizar la confidencialidad de los datos.

En esta sección presentaremos una breve introducción sobre las principales líneas de investigación abocadas a este tema.

3.1. Mecanismos de Seguridad Estándares

Existen diversos mecanismos para proteger la confidencialidad de los datos manipulada por sistemas de información. Dentro de ellos, quizás el mecanismo de control más conocido, es el Control de Acceso Discrecional (Discretionary Access Control, DAC [1]), que fue diseñado para restringir accesos no autorizados sobre recursos de un sistema, basado en que los usuarios requieren de privilegios para acceder a estos. Pese a poner restricciones a quienes intentan acceder a información confidencial, no controla su propagación: una vez que a un usuario le es concedido el acceso, está fuera de sus limites restringir lo que el usuario pueda hacer con esta.

Otros mecanismos como firewalls, encriptación y antivirus utilizados para este mismo propósito comparten su misma falencia. Por ejemplo, un firewall protege información sensible prohibiendo la comunicación de una parte del sistema con el exterior, no obstante, no controla como se propaga la información. De manera similar, la encriptación puede ser usada para asegurar la confidencialidad de los datos en un canal de comunicación entre sus dos extremos, sin embargo, una vez que los datos son desencriptados por el receptor, no existen garantías del uso que este le dé a los datos.

3.2. Seguridad Basada en el Flujo de Información

Podemos garantizar la privacidad de la información analizando cómo la información fluye dentro del sistema en tiempo de ejecución. Los pioneros en presentar un modelo

formal para regular el flujo de información de forma dinámica fueron Bell y LaPadula en su trabajo seminal [2], al que denominaron Control de Acceso Mandatorio (Mandatory Access Control, MAC). Este modelo funciona etiquetando a cada ítem de información y a usuarios con un nivel de seguridad. Con esta información extra, el flujo de información es controlado al momento de que un usuario solicita acceso a cierto ítem, al cual se le otorga solo si su nivel de seguridad es mayor al del ítem que trata de acceder. También se impide el flujo de información hacia niveles menores, por ejemplo cuando un usuario solicita el acceso a un ítem con nivel de seguridad menor, con el objetivo de editarlo.

Existen también enfoques estáticos en el control de flujo de información. Los primeros en observar que realizar un análisis estático de un programa puede usarse para este propósito, fueron Denning y Denning [6], con su propuesta llamada *certificación de programas*. Mientras que Volpano, Smith e Irvine [32] fueron los primeros en presentar un sistema de tipos que asegure que una propiedad de seguridad llamada no-interferencia se cumple en los programas. En este enfoque, se le asocia a cada expresión en un programa un *tipo de seguridad* que consiste en un tipo común, como `int`, y una etiqueta que describe como este valor debe usarse. Por ejemplo, la siguiente declaración: `x int : low`; nos indica que x es una variable de tipo `int` que contiene datos públicos. A los sistema de tipos que permiten asegurar que los programa tipados no contienen flujos de información de datos privados hacia datos públicos se los denomina sistema de tipos de flujo de información.

Existen dos tipos de flujo de información: los *flujos explícitos* que son causados por una asignación directa de datos confidenciales hacia datos públicos, mientras que los *flujos de información implícitos* se generan por la estructura de control del programa, los cuales son más difíciles de detectar.

El siguiente es un ejemplo de un programa que contiene un flujo implícito de un valor confidencial a un valor público. Primero clasificamos las variables en dos niveles de seguridad: *high* para datos confidenciales y *low* para datos públicos. En el siguiente ejemplo, donde la variable l tiene nivel de seguridad *low* y h tiene nivel *high*, ocurre un flujo implícito desde la variable h hacia l, dado que inspeccionando el valor de l podemos determinar si h almacena el valor 0 o no:

$l := 0$;
`if` $h = 0$ `then` $l := 1$ `else` *skip*

Para detectar flujos de información implícitos mediante un sistemas de tipos se introduce una etiqueta de seguridad global denominada *program counter* (*pc*), la cual se utiliza para fijar el nivel de seguridad del contexto en el cual se debe ejecutar una sentencia.

Este enfoque es el que utilizaremos en esta tesina para formalizar una propiedad de seguridad en un lenguaje imperativo simple. Entraremos más en profundidad en la siguiente sección.

Por último, aunque esté fuera del alcance de este trabajo, es importante mencionar que existen otros tipos de flujos implícitos además de los observables desde la estructura de control del programa, siendo en este caso a través de *canales encubiertos* (Covert Channels, [14]). Por ejemplo en el caso del ejemplo anterior un atacante puede inferir el valor de la variable h observando si el programa termina o no. Este caso se lo denomina *termination channel*.

3.3. Sistema de Tipos No-Interferente

Introducimos un lenguaje imperativo sencillo tipo *while* con la siguiente sintaxis abstracta:

$$\begin{array}{rcl} e & ::= & val \mid v \mid e_1 \,\mathtt{op}\, e_2 \\ c & ::= & \mathtt{skip} \mid v := e \mid c_1;c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c \end{array}$$

Denotaremos con los nombres h y l a variables con niveles de confidencialidad *high* y *low*, respectivamente.

La confidencialidad de la información puede asegurarse en tiempo de compilación a través de un sistema de tipos que garantice una propiedad de seguridad. Una política de seguridad que puede ser garantizada a través de un sistema de tipos es no-interferencia [9]. Esta propiedad formaliza el hecho de que el uso de datos sensibles o privados no interfiere con los datos públicos, es decir, si ejecutamos un programa con diferentes entradas que difieren en los valores de las variables confidenciales, los valores de las variables públicas de la salida no pueden ser diferentes.

En la Figura 1 mostramos un sistema de tipos de seguridad definido en [32]. Este sistema nos dice que para el caso de las expresiones, la notación $\vdash exp : \tau$ significa que la expresión exp tiene tipo de seguridad τ, mientras que para los comandos, la notación $[pc] \vdash C$ nos dice que C es *tipable* en el contexto de seguridad pc. El sistema nos asegurará que si son tipables, entonces cumplirán con la propiedad de no-interferencia. Por simplicidad se asumen solo dos niveles de seguridad: *high* y *low*.

La regla EXPH, nos dice que cualquier expresión puede tener tipo *high*. Mientras que según la regla EXPL, una expresión puede tener tipo *low* solamente si no tiene ocurrencias de variables de tipo *high*. Las reglas para los comandos SKIP y ASSH aseguran que los comandos **skip** y $h := exp$ son tipables en cualquier contexto. La regla ASSL previene los flujos explícitos asegurando que $l := exp$ solo es tipable si

exp tiene tipo *low*. Además esta regla es usada junto con la regla IF y WHILE para prevenir flujos implícitos, asegurando que $l := exp$ tipa solamente en un contexto de seguridad *low*.

La regla IF asegura que si la condición del comando **if** es de tipo *high*, entonces las ramas deben tipar en un contexto de seguridad *high*. Mientras que si la condición es de tipo *low*, las ramas pueden tipar en cualquier contexto, ya que según la regla SUB si un programa es tipable en un contexto *high*, también es tipable en *low*. La regla WHILE es similar a IF.

Expresiones

$$\frac{}{\vdash exp : high}\ \text{EXPH} \qquad \frac{h \notin Vars(exp)}{\vdash exp : low}\ \text{EXPL}$$

Comandos

$$\frac{}{[pc] \vdash \mathbf{skip}}\ \text{SKIP} \qquad \frac{}{[pc] \vdash h := exp}\ \text{ASSH}$$

$$\frac{\vdash exp : low}{[low] \vdash l := exp}\ \text{ASSL} \qquad \frac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1; C_2}\ \text{SEQ}$$

$$\frac{\vdash exp : pc \quad [pc] \vdash C}{[pc] \vdash \mathbf{while}\, exp\, \mathbf{do}\, C}\ \text{WHILE} \qquad \frac{\vdash exp : pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \mathbf{if}\, exp\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2}\ \text{IF}$$

$$\frac{[high] \vdash C}{[low] \vdash C}\ \text{SUB}$$

Figura 1: Sistema de tipos seguro en lenguaje imperativo simple.

3.4. Desclasificación

Aunque la mayoría de los trabajos en sistemas de tipos de seguridad modelan la propiedad de no-interferencia, esta suele ser muy restrictiva en la práctica. Algunos programas necesitan liberar información secreta en cierto grado como parte de su funcionalidad. Por ejemplo, si intentamos loguearnos en una aplicación, es posible

que una persona aprenda una pequeña cantidad de información acerca de la contraseña del usuario con la cual intenta acceder, ya que en cada intento se revela que la contraseña no es la que se ingresa. Otra clase de ejemplos importantes es la agregación de datos, consideremos por ejemplo un programa que calcula el promedio de salarios de empleados de una empresa: aunque cada salario sea información confidencial, cada empresa necesita tener un programa de estas características.

La no-interferencia falla en reconocer tales programas como seguros y son rechazados por los sistemas de tipos que modelan esta propiedad, por lo que muchos lenguajes seguros que modelan no-interferencia no son útiles en la práctica. Como consecuencia es necesario relajar la noción de no-interferencia y tener mecanismos para liberar de manera controlada información confidencial.

Sabelfeld y Sands [28] dan una clasificación de los objetivos básicos de los últimos trabajos sobre desclasificación, de acuerdo a los siguientes puntos a los que ellos denominan *dimensiones*:

- **qué** información es liberada.
- **quién** libera la información.
- **dónde** es liberada la información en el sistema
- **cuándo** la información puede ser liberada

Existen trabajos que definen distintas políticas de seguridad y sistemas de tipos que las garantizan en donde está permitido la liberación intencional de información. Por ejemplo, clasificada dentro de qué tipo de información es liberada, Sabelfeld y Myers [27] introducen una nueva política de seguridad que llaman *delimited release*, mediante esta se asegura que la desclasificación o liberación de información no puede ser usada para filtrar información de manera no deseada. Esta propiedad, como la no-interferencia, tiene el atractivo de que se puede definir un sistema de tipos, de manera que los programas tipables satisfacen la propiedad.

El lenguaje definido en [27], es un lenguaje imperativo simple con semántica estándar, que consiste solo de expresiones y comandos, similar al lenguaje definido en la sección anterior. La sintaxis abstracta se define como:

$$\begin{array}{lcl} e & ::= & val \mid v \mid e_1 \,\mathtt{op}\, e_2 \mid \mathtt{declassify}(e, l) \\ c & ::= & \mathtt{skip} \mid v := e \mid c_1;c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c \end{array}$$

En este lenguaje, las constantes puede tomar valores naturales o booleanos, las variables pertenecen a un conjunto de identificadores Var, y las operaciones son las aritméticas o booleanas estándar.

Los niveles de seguridad forman parte de un retículo de seguridad $\mathcal{L}$, donde el orden entre los elementos de este conjunto especifica una relación entre los distintos niveles de seguridad. Un ejemplo de un retículo de seguridad es el usado en la sección previa: $\mathcal{L}_{LH}$, que contiene dos elementos *high* y *low*, representando los niveles de confidencialidad públicos y privados respectivamente, con orden $low \sqsubseteq high$.

Los comandos son los mismos que en el lenguaje de la sección anterior, la única diferencia es la expresión `declassify`(e, l): su función es la de modificar el nivel de seguridad de la expresión e al nivel l. El nivel asignado a l puede variar dentro de los distintos niveles de seguridad del retículo $\mathcal{L}$. Las expresiones `declassify` no pueden ser anidadas.

Definimos la semántica del lenguaje en términos de transiciones entre configuraciones. Una configuración $\langle M, c\rangle$ consta de una memoria M y un comando o expresión c. La memoria la definimos como un mapa finito entre variables y valores: $M : Var \to Val$. Una transición desde la configuración $\langle M, c\rangle$ hacia otra configuración $\langle M', c'\rangle$, se denota: $\langle M, c\rangle \longrightarrow \langle M', c'\rangle$, mientras que una transición desde una configuración $\langle M, c\rangle$ hacia una configuración terminal con una memoria M', se denota como: $\langle M, c\rangle \longrightarrow M'$. La relación $\longrightarrow^*$ es la clausura reflexiva y transitiva de $\longrightarrow$. Decimos que la configuración $\langle M, c\rangle$ *termina* en M' si $\langle M, c\rangle \longrightarrow^* M'$, que también puede escribirse como: $\langle M, e\rangle \Downarrow M'$. Si las operaciones usadas en las expresiones son totales, las configuraciones para expresiones siempre terminan. Podemos escribirlas como: $\langle M, e\rangle \Downarrow val$.

La intención de los autores de la política de seguridad *delimited release* es que solo se permita liberar información mediante una expresión `declassify`. Por lo tanto, la definición de esta propiedad va a depender de las expresiones que aparecen bajo el operador `declassify`.

Antes de mostrar la definición de la propiedad veremos algunos conceptos que se utilizan en su definición.

Un *ambiente de seguridad*: $\Gamma : Var \to \mathcal{L}$, es un mapa finito entre variables y un retículo de seguridad. Dado un ambiente de seguridad Γ fijo, diremos que dos memorias M_1 y M_2 son *indistinguibles* a nivel l, y escribimos: $M_1 =_l M_2$, si $\forall v \,.\, \Gamma(v) \sqsubseteq l \implies M_1(v) = M_2(v)$. Mientras que el comportamiento de dos configuraciones de comandos $\langle M_1, c_1\rangle$ y $\langle M_2, c_2\rangle$ es *indistinguible* a nivel l, y escribimos: $\langle M_1, c_1\rangle \approx_l \langle M_2, c_2\rangle$, si cuando $\langle M_1, c_1\rangle \Downarrow M_1'$ y $\langle M_2, c_2\rangle \Downarrow M_2'$ para algún M_1' y M_2', entonces $M_1' =_l M_2'$. De manera similar el comportamiento de dos configuraciones de expresiones $\langle M_1, e_1\rangle$ y $\langle M_2, e_2\rangle$ es *indistinguible*, y escribimos: $\langle M_1, e_1\rangle \approx \langle M_2, e_2\rangle$, cuando $\langle M_1, e_1\rangle \Downarrow val$ y $\langle M_2, e_2\rangle \Downarrow val$ para algún val.

A continuación daremos la definición formal de la propiedad de seguridad *delimited*

release presentada en [27].

Definición 1 (Delimited release)**.** *Supongamos que el comando c contiene exactamente n expresiones* `declassify`*:* $\texttt{declassify}(e_1, l_1), ..., \texttt{declassify}(e_n, l_n)$*. Decimos que el comando c es seguro si para todos los niveles de seguridad l tenemos:*

$$\forall M_1, M_2 \;.\; (M_1 =_l M_2 \;\&\; \forall i \in \{i \mid l_i \sqsubseteq l\}.\langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle) \implies \langle M_1, c\rangle \approx_l \langle M_2, c\rangle$$

Formalmente la definición establece que para todo nivel de seguridad l y memorias M_1 y M_2, tal que $M_1 =_l M_2$, si podemos observar a nivel l una filtración a través de alguna de las puertas de escape (expresiones que ocurren bajo un operador `declassify`) $e_1, ..., e_n$, es decir: $\exists i \in \{i \mid l_i \sqsubseteq l\} \;.\; \langle M_1, e_i\rangle \not\approx \langle M_2, e_i\rangle$, entonces esta filtración está permitida. Si a nivel l, la diferencia entre M_1 y M_2 es invisible sobre todas las puertas de escape, es decir: $\forall i \in \{i \mid l_i \sqsubseteq l\} \;.\; \langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$, entonces esta diferencia debe ser invisible a nivel l sobre la ejecución de c, es decir: $\langle M_1, c\rangle \approx_l \langle M_2, c\rangle$.

Los siguientes ejemplos ayudarán a entender esta propiedad de seguridad.

3.4.1. Ejemplos

Ejemplo 1. Promedio de salarios.

Supongamos que tenemos 3 variables: h_1, h_2 y h_3 destinadas a guardar información confidencial y una variable pública avg. En estas variables se van a almacenar los salarios de 3 empleados y el promedio de ellas se guardará en la variable avg. Dado que las variables h_1, h_2 y h_3 son confidenciales se desclasificará el promedio de ellas para poder realizar la asignación. Con la sintaxis del lenguaje original podemos escribir este programa de la siguiente manera:

$$avg := (\texttt{declassify}(h_1 + h_2 + h_3)/3, \mathit{low}) \qquad \text{(Avg)}$$

Podemos observar que hay una dependencia desde las variables confidenciales h_1, h_2 y h_3 hacia la variable pública avg, por lo que este programa no satisface la propiedad de no-interferencia, sin embargo, aunque el usuario pueda aprender el promedio de variables confidenciales, no puede obtener más información sobre los salarios de los empleados. Por ejemplo, cualquier actualización que haya en los valores de los salarios, no va a ser visible al usuario que solo tiene acceso a variables públicas. Este programa sí es seguro de acuerdo a la definición de la propiedad delimited release.

Consideremos ahora el siguiente programa que filtra el salario del empleado almacenado en la variable h_2. Al actualizar los valores de h_1 y h_3, variables que luego

van a ser usadas en una expresión dentro de un operador `declassify`, el siguiente programa no satisface la propiedad delimited release:

$$\begin{array}{l} h_1 =: h_2; \\ h_3 =: h_2; \\ avg := (\texttt{declassify}(h_1 + h_2 + h_3)/3,\ low) \end{array} \qquad \text{(Avg-Attack)}$$

El programa falla al revelar información sobre la variable h_2. Para ver por qué este programa es rechazado por el modelo, consideremos la memoria M_1 con los valores: $M_1(h_1) = 2, M_1(h_2) = 3, M_1(h_3) = 5$ y $M_1(avg) = 0$, y M_2 con los valores $M_2(h_1) = 3, M_2(h_2) = 2, M_2(h_3) = 5$ y $M_2(avg) = 0$. Para $l = low$ tenemos $M_1 =_l M_2$ y $\langle M_1, (h_1+h_2+h_3)/3)\rangle \approx \langle M_2, (h_1+h_2+h_3)/3)\rangle$, por que ambas expresiones evalúan a 10/3. Sin embargo, $\langle M_1, \text{Avg-Attack}\rangle \not\approx_l \langle M_2, \text{Avg-Attack}\rangle$ por que el valor final de la variable pública avg es 3 y 2, respectivamente, violando la Definición 1.

Ejemplo 2. Billetera electrónica.

Consideremos ahora una billetera electrónica, la cual se quiere utilizar como medio de pago para realizar compras online. Para este caso, supongamos que la variable h almacena información confidencial sobre la cantidad de dinero que el usuario posee en la billetera. La variable l, la cantidad de dinero gastada durante la sesión, y k el costo del ítem que se quiere comprar. Estas últimas dos variables son públicas.

El siguiente fragmento es un programa escrito en el lenguaje original que satisface la propiedad de delimited release. En este, el programa verifica que la cantidad de dinero que queda en la billetera es suficiente, y en caso de serlo, transfiere la cantidad k de dinero desde la billetera electrónica del usuario hacia a la variable l:

$$\texttt{if declassify}(h \geq k,\ low)\ \texttt{then}\ (h := h - k;\ l := l + k)\ \texttt{else}\ skip \qquad \text{(Wallet)}$$

De la misma manera que el ejemplo anterior, este programa no satisface la propiedad de no-interferencia, pero sí la propiedad de delimited release.

Consideremos ahora el siguiente ataque a la integridad de la billetera:

$$\begin{array}{l} l =: 0; \\ \texttt{while}\ n \geq 0\ \texttt{do} \\ \quad k =: 2^{n-1}; \\ \quad \texttt{if declassify}(h \geq k, low)\ \texttt{then}\ (h := h - k;\ l := l + k)\ \texttt{else skip}; \\ \quad n =: n - 1 \end{array} \qquad \text{(Wallet-Attack)}$$

En este caso el observador puede llegar a conocer el valor de h a través de l. Si $h < 2^n$ se puede aprender el valor de h al quedar almacenado en l al finalizar el bucle. Este abuso en el uso de la expresión `declassify` será rechazado por el modelo.

$$\frac{}{\Gamma \vdash val \,:\, l,\, \emptyset}\ VAL$$

$$\frac{\Gamma(v) = l}{\Gamma \vdash v \,:\, l,\, \emptyset}\ VAR$$

$$\frac{\Gamma \vdash e \,:\, l,\, D_1 \quad \Gamma \vdash e' \,:\, l,\, D_2}{\Gamma \vdash e \,\mathbf{op}\, e' \,:\, l,\, D_1 \cup D_2}\ OP$$

$$\frac{\Gamma \vdash e \,:\, l,\, D}{\Gamma \vdash \mathbf{declassify}(e, l') \,:\, l',\, Vars(e)}\ DEC$$

$$\frac{\Gamma \vdash e \,:\, l,\, D \quad l \sqsubseteq l'}{\Gamma \vdash e \,:\, l',\, D}\ SUBE$$

$$\frac{}{\Gamma,\, pc \vdash \mathbf{skip} \,:\, \emptyset,\, \emptyset}\ SKIP$$

$$\frac{\Gamma \vdash e \,:\, l,\, D \quad l \sqcup pc \sqsubseteq \Gamma(v)}{\Gamma,\, pc \vdash v := e \,:\, \{v\},\, D}\ ASIG$$

$$\frac{\Gamma,\, pc \vdash c_1 \,:\, U_1,\, D_1 \quad \Gamma,\, pc \vdash c_2 \,:\, U_2,\, D_2 \quad U_1 \cap D_2 = \emptyset}{\Gamma,\, pc \vdash c_1; c_2 \,:\, U_1 \cup U_2,\, D_1 \cup D_2}\ SEQ$$

$$\frac{\Gamma \vdash e \,:\, l,\, D \quad \Gamma,\, l \sqcup pc \vdash c_1 \,:\, U_1,\, D_1 \quad \Gamma,\, l \sqcup pc \vdash c_2 \,:\, U_2,\, D_2}{\Gamma,\, pc \vdash \mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \,:\, U_1 \cup U_2,\, D \cup D_1 \cup D_2}\ IF$$

$$\frac{\Gamma \vdash e \,:\, l,\, D \quad \Gamma,\, l \sqcup pc \vdash c \,:\, U_1,\, D_1 \quad U_1 \cap (D \cup D_1) = \emptyset}{\Gamma,\, pc \vdash \mathbf{while}\, e\, \mathbf{do}\, c \,:\, U_1,\, D \cup D_2}\ WHILE$$

$$\frac{\Gamma,\, pc \vdash c \,:\, U,\, D \quad pc' \sqsubseteq pc}{\Gamma,\, pc' \vdash c \,:\, U,\, D}\ SUBC$$

Figura 2: Sistema de tipos seguro en lenguaje imperativo simple con desclasificación.

3.4.2. Sistema de Tipos Para la Propiedad Delimited Release

De la misma manera que en el sistema de tipos presentado en la sección anterior para no-interferencia, el sistema de tipos para delimited release también controla el flujo de información en las asignaciones y las sentencias de control. Además el sistema también colecta información acerca de cuales de las variables son usadas para desclasificar información y cuales son actualizadas.

El juicio de tipado $\Gamma \vdash e \,:\, l, D$, significa que una expresión e tiene tipo l y efecto D bajo el ambiente Γ. En D se almacenarán las variables que son desclasificadas.

Mientras que para comandos el juicio de tipado Γ, $pc \vdash c : U, D$ significa que un comando c es tipable con efectos U y D, bajo un ambiente Γ y un *program counter* pc. En U se guardarán las variables que son actualizadas y en D las variables que son desclasificadas.

Las reglas de tipado son presentadas en la Figura 2.

Además de controlar el flujo de información explícito e implícito, este sistema de tipos colecta información sobre las variables desclasificadas y actualizadas para garantizar que las variables que aparecen bajo una expresión `declassify`, no son actualizadas antes de desclasificarse. Cabe destacar que si un programa tipable en este lenguaje no contiene expresiones `declassify`, es seguro respecto a la propiedad de no-interferencia.

Por último es importante mencionar que todo programa tipable es seguro respecto a la propiedad de delimited release, pero no todo programa seguro respecto a esta propiedad es tipable. Por ejemplo, consideremos el siguiente programa:

$$h =: h \ \% \ 2 = 0;$$
$$l := \texttt{declassify}(h,\ low)$$

En donde h será destinada a almacenar valores confidenciales y l valores públicos. En este ejemplo, el programa es rechazado por el sistema de tipos porque h es actualizada antes de la desclasificación, sin embargo satisface la Definición 1.

Utilizaremos este sistema de tipos como punto de partida de esta tesina, presentando una variante dirigida por sintaxis que nos resultará más apropiada para su posterior implementación. Mostraremos también que ambos sistemas de tipos son equivalentes.

4. Programación con Tipos de Datos Dependientes en Haskell

Los tipos de datos dependientes son tipos cuya definición *depende* de valores de otros tipos. Por ejemplo en el tipo de vectores de tamaño n compuesto por elementos de tipo A: A^n, decimos que el tipo A^n depende del número n. Otro ejemplo similar es el tipo $A^{m \times n}$ de matrices $m \times n$, podemos decir en este caso que depende de los valores m y n. En los lenguajes de programación que poseen tipos de datos dependientes, como por ejemplo Agda, Coq o Epigram, los tipos de datos dependientes pueden ser usados para exprear propiedades acerca de los programas en el tipo de los mismos.

En este tipo de lenguajes no hay una clara separación entre el universo de los valores con el del universo de los tipos de datos (como sí sucede en lenguajes como ML o

Haskell). Los tipos son valores, pueden depender de otros valores, ser pasados como argumentos, o devuelto por funciones.

Como ejemplo de una definición de un tipo de datos dependiente consideremos el tipo de vectores de un tamaño dado que mencionamos al comienzo en Agda [22]:

```
data Vec (A : Set) : N -> Set where
 nil : Vec A zero
 cons : {n : N} -> A -> Vec A n -> Vec A (suc n)
```

donde `A` es el tipo de los elementos y `n` es su tamaño. En la versión estándar de Haskell podemos definir el tipo para vectores de un tamaño dado definiendo primero los valores del tipo `Nat` como tipos:

```
data Zero
data Succ n
```

Notar que estos dos tipos de datos son vacíos, ya que solo nos servirán como valores a nivel de tipos. Luego usamos GADTs (Ver 4.1) para definir el tipo `Vec`, donde el primer argumento es un número natural a nivel de tipos, que representa el tamaño del vector y el segundo argumento es el tipo de sus elementos:

```
data Vec :: * -> * -> * where
 Nil :: Vec Zero x
 Cons :: x -> Vec n x -> Vec (Succ n) x
```

Cabe destacar que a diferencia de la definición de vectores en Agda, donde el parámetro `n` de `Vec` tiene tipo `Nat`, en la definición en Haskell el parámetro `n` de `Vec` es de la forma `Zero` o `Succ n`, donde `n` tiene kind `*`, lo que nos posibilitará escribir tipos sin significado, como `Succ Bool`, `Succ Char`, etc. Más adelante veremos una extensión de Haskell para definir un kind `Nat` y de esta manera tipar los constructores de tipo `Zero` y `Suc`.

Aunque actualmente hay una extensión en desarrollo [7], Haskell todavía no es un lenguaje con tipos de datos dependientes, básicamente, al no permitir que aparezcan valores en los tipos. Sin embargo, existen algunas extensiones: GADTs, type families, datatype promotion y kind polymorphism, que nos permitirán simular, aunque de manera acotada, la programación con tipos dependientes.

A continuación haremos una breve introducción de algunas extensiones que se utilizarán en esta tesina.

4.1. Generalized Algebraic Data Types

Generalized algebraic data types (GADTs) [12] son una generalización de los tipos

algebraicos que usamos generalmente en Haskell. Básicamente nos permiten anotar explícitamente los tipos en los constructores, por ejemplo:

```
data Expr a where
    I   :: Int  -> Expr Int
    B   :: Bool -> Expr Bool
    Add :: Expr Int -> Expr Int -> Expr Int
    Mul :: Expr Int -> Expr Int -> Expr Int
    Eq  :: Eq a => Expr a -> Expr a -> Expr Bool
```

De esta manera, a diferencia de los tipos algebraicos, podemos aumentar la expresividad de los tipos al proveerlos de más información en su definición. En este ejemplo el tipo de retorno no es siempre el mismo. Esta extensión nos permite escribir evaluadores como el siguiente:

```
eval :: Expr a -> a
eval (I n) = n
eval (B b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Eq  e1 e2) = eval e1 == eval e2
```

Permitiéndonos poder incluir los `Int` y los `Bool` en una función de evaluación, que de sin hacer uso de esta extensión, no hubiese sido posible.

Los GADTs son muy útiles para definir tipos de datos dependientes en Haskell. Como adelantamos en la sección anterior donde se definió el tipo de vectores de tamaño fijo usando GADTs.

4.2. Promoted Data Types

En Haskell los valores son clasificados en tipos y los tipos son clasificados en kinds. Los tipos básicos como `Int`, `String`, etc, tienen kind `*`, mientras que los constructores de tipo, que toman un tipo de kind `v` y devuelven un tipo de kind `k`, tienen kind `v -> k`. Por ejemplo, `Maybe`, tiene kind `* -> *`.

A diferencia del lenguaje de los tipos, el lenguaje de kinds es muy restrictivo, lo que lleva a tener dos consecuencias negativas, por un lado, nos limita al momento de escribir programas más interesantes a nivel de tipos, y además puede llevar a errores involuntarios como el que veremos a continuación.

Como mencionamos al comienzo de esta sección, previamente a esta extensión [36], si uno deseaba usar los números naturales a nivel de tipos, generalmente se daban dos declaraciones `data` vacías para usarse como "valores a nivel de tipos":

```
data Zero
data Succ a
```

Ahora podemos considerar el tipo `Succ (Succ Zero)` como el número 2. Sin embargo, podemos escribir tipos sin sentidos como `Succ Bool` que, además de no expresar la intención inicial de programador, es propenso a errores.

Con la extensión promoted data types se puede ampliar el conjunto de kinds. Esta extensión permite elevar la definición de tipos realizada con el constructor `data`, elevando los valores a tipos y el tipo de estos valores a kind. Por ejemplo a partir de esta definición:

```
data Nat = Zero | Succ Nat
```

Si queremos usar los valores `Zero` y `Suc Zero`, etc, como tipos escribimos `'Zero` y `'Succ 'Zero`, etc, mientras que también podemos usar el tipo `Nat` como kind usando el mismo nombre, por ejemplo, ahora podríamos escribir una definición de vectores, diferente a la del ejemplo anterior, como:

```
data Vec :: Nat -> * -> * where
 Nil :: Vec Zero x
 Cons :: x -> Vec n x -> Vec (Succ n) x
```

4.3. Type Families

Las *type families* [30] son una de las herramientas más poderosas para programar a nivel de tipos en Haskell. Básicamente una familia de tipos es una función sobre tipos. El siguiente es un ejemplo sencillo de una función que reemplaza el tipo `Int` por `Bool` y `Char` por `Double`:

```
type family F a where
 F Int = Bool
 F Char = Double
```

La función `F` puede ser usada en el tipo de cualquier función común, por ejemplo:

```
useF :: F Int -> F Char
useF True = 1.0
useF False = -1.0
```

Un ejemplo más interesante es la concatenación de listas a nivel de tipos:

```
type family Append xs ys where
 Append '[]     ys = ys
 Append (x ': xs) ys = x ': Append xs ys
```

Esta definición es similar a la definición de concatenación a nivel de valores: `(++):[a]->[a]->[a]`, la diferencia es que los constructores de las listas `[]` y `(:)` aparecen primados en esta definición, ya que se refieren a los constructores de listas a nivel de tipo.

Tanto `F` como `Append` son *closed type families*, en estas, todas las ecuaciones que la definien están dadas en un solo lugar, de manera similar a como se escribe una función a nivel de valores. Haskell tambien soporta *open type families* donde las ecuaciones que las definen se pueden extender arbitrariamente. En esta tesina usaremos solamente las primeras.

4.4. Kind Polymorphism

El *kind polymorphism* [36] nos permite usar variables a nivel de kind, lo cual es necesario para escribir funciones a nivel de tipo polimórficas. Por ejemplo, la siguiente función a nivel de tipos calcula el largo de una lista:

```
type family Length (list :: [k]) :: Nat where
 Length '[] = 'Zero
 Length (x ': xs) = 'Succ (Length xs)
```

En esta definición `k` es una variable a nivel de kind, lo cual nos permite aplicar la función a listas de tipos de cualquier kind. Los constructores primados como `'[]` y `'(:)` también tienen variables en su kind, por ejemplo: `'():: a -> [a] -> [a]`.

4.5. Singletons

Los *singletons* son tipos de datos que tienen exactamente un habitante. Como explicamos anteriormente, la extensión *promoted data types* nos permite duplicar un tipo de datos común al nivel de kinds. Para el caso de `Nat` usado en secciones anteriores, a partir de su declaración de tipos, GHC genera un nuevo kind, también llamado `Nat`, formado por los tipos definidos a partir los constructores de datos `'Zero` y `'Succ`. Los valores a nivel de tipo son útiles para indexar otros tipos como por ejemplo, el caso de vectores de un tamaño dado.

Sin embargo los tipos de datos promocionados a veces no son suficientes para definir algunas funciones que necesitan inspeccionar el tipo de datos. En el siguiente ejemplo, tenemos definida la suma a nivel de tipos de los naturales, mediante la siguiente familia de tipos:

```
type family (m :: Nat) :+ (n :: Nat) :: Nat
 Zero :+ n = n
 Succ m :+ n = Succ (m :+ n)
```

Con la definición de esta función, puedo definir sin problemas la concatenación de vectores:

```
vappend :: Vec m x -> Vec n x -> Vec (m :+ n) x
vappend Nil ys = ys
vappend (Cons x xs) ys = Cons x (vappend xs ys)
```

Supongamos que queremos definir ahora una función que parta un vector en dos vectores:

```
vsplit :: Vec (m :+ n) x ->  (Vec m x, Vec n x)
```

A diferencia de `vappend`, en este caso necesitamos de `m` para poder definir la función, de lo contrario no tendríamos manera de saber donde cortar el vector. La solución estándar es definir primero un GADT `Natty m` el cual tiene exactamente un valor para cada `m`:

```
data Natty :: Nat -> * where
 Zy :: Natty Z
 Sy :: Natty n -> Natty (S n)
```

De esta manera tenemos dos representaciones de los objetos que manipulamos, una a nivel de valores y otra a nivel de tipos: cada valor a nivel de tipos `n` en el kind `Nat` tiene su representante a nivel de valores en el tipo `Natty`. Podemos usar pattern matching sobre un valor de tipo `Natty m` para conocer `m` en tiempo de ejecución. Ahora estamos en condiciones de definir `vsplit`:

```
vsplit :: Natty m -> Vec (m :+ n) x -> (Vec m x, Vec n x)
vsplit Zy xs = (Nil, xs)
vsplit (Sy m) (Cons x xs) = (Cons x ys, zs)
  where (ys, zs) = vsplit m xs
```

Parte II

Lenguaje Seguro Respecto a la Propiedad Delimited Release

5. Introducción

En esta sección introduciremos el lenguaje seguro que implementaremos como un EDSL. Comenzaremos describiendo el sistema de tipos que cumple la propiedad de seguridad, para luego mostrar la implementación de la sintaxis y el sistema de tipos en Haskell. Finalizaremos la sección con ejemplos representativos.

5.1. Sistema de Tipos

La Figura 3 presenta una alternativa al sistema de tipos de seguridad del lenguaje, la diferencia entre el que presentamos y el definido en la Figura 3, es que eliminamos las reglas $SUBE$ y $SUBC$, denominadas reglas de subsunción. Como resultado ahora tenemos un sistema dirigido por sintaxis que nos resultará más apropiado para la implementación.

Bajo este nuevo contexto, la forma de tipar una expresión es a través del juicio de tipado: $\Gamma \vdash_{sd} e : l, D$, que significa que una expresión e tiene tipo l y efecto D bajo el ambiente Γ. En el conjunto D se almacenan las variables que son desclasificadas. Mientras que un juicio de tipado de la forma Γ, $pc \vdash_{sd} c : U, D$, nos indica que el comando c es tipable con efectos U y D, bajo un ambiente Γ y un contexto de seguridad pc. El conjunto U guarda las variables que son actualizadas y en D se guardan las variables desclasificadas.

$$\frac{}{\Gamma \vdash_{sd} val : l\,,\, \emptyset}\ VAL_{sd}$$

$$\frac{\Gamma(v) = l}{\Gamma \vdash_{sd} v : l\,,\, \emptyset}\ VAR_{sd}$$

$$\frac{\Gamma \vdash_{sd} e : l\,,\, D_1 \quad \Gamma \vdash e' : l'\,,\, D_2}{\Gamma \vdash_{sd} e\ \mathbf{op}\ e' \ :\ l \sqcup l',\ D_1 \cup D_2}\ OP_{sd}$$

$$\frac{\Gamma \vdash_{sd} e : l\,,\, D}{\Gamma \vdash_{sd} \mathbf{declassify}(e, l') \ :\ l',\ Vars(e)}\ DEC_{sd}$$

$$\frac{}{\Gamma,\ pc \vdash_{sd} \mathbf{skip} \ :\ \emptyset,\ \emptyset}\ SKIP_{sd}$$

$$\frac{\Gamma \vdash_{sd} e \ :\ l,\ D \quad l \sqcup pc \sqsubseteq \Gamma(v)}{\Gamma,\ pc \vdash_{sd} v := e \ :\ \{v\},\ D}\ ASIG_{sd}$$

$$\frac{\Gamma,\ pc \vdash_{sd} c_1 \ :\ U_1,\ D_1 \quad \Gamma,\ pc' \vdash_{sd} c_2 \ :\ U_2,\ D_2 \quad U_1 \cap D_2 = \emptyset}{\Gamma,\ pc \sqcap pc' \vdash_{sd} c_1,\ c_2 \ :\ U_1 \cup U_2,\ D_1 \cup D_2}\ SEQ_{sd}$$

$$\frac{\Gamma \vdash_{sd} e \ :\ l,\ D \quad \Gamma,\ pc \vdash_{sd} c_1 \ :\ U_1,\ D_1 \quad \Gamma,\ pc' \vdash_{sd} c_2 \ :\ U_2,\ D_2 \quad l \sqsubseteq pc \sqcap pc'}{\Gamma,\ pc \sqcap pc' \vdash_{sd} \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \ :\ U_1 \cup U_2,\ D \cup D_1 \cup D_2}\ IF_{sd}$$

$$\frac{\Gamma \vdash_{sd} e \ :\ l,\ D \quad \Gamma,\ pc \vdash_{sd} c_1 \ :\ U_1,\ D_1 \quad l \sqsubseteq pc \quad U_1 \cap (D \cup D_1) = \emptyset}{\Gamma,\ pc \vdash_{sd} \mathbf{while}\ e\ \mathbf{do}\ c \ :\ U_1,\ D \cup D_2}\ WHILE_{sd}$$

Figura 3: Sistema de tipos dirigido por sintaxis que satisface Delimited Release.

El siguiente teorema establece una relación entre los dos sistemas de tipos:

Teorema 5.1. *Para cada expresión e, comando c y niveles de seguridad pc y l:*

(i) Si $\Gamma \vdash_{sd} e : l, D$ *entonces* $\Gamma \vdash e : l, D$.

(ii) Si $\Gamma \vdash e : l, D$ *entonces existe* l' *tal que* $\Gamma \vdash_{sd} e : l', D$ *y* $l' \sqsubseteq l$.

(iii) Si $\Gamma, pc \vdash_{sd} c : U, D$ *entonces* $\Gamma, pc \vdash c : U, D$.

(iv) Si $\Gamma, pc \vdash c : U, D$ *entonces existe un* pc', *tal que* $\Gamma, pc' \vdash_{sd} c : U, D$ *y* $pc \sqsubseteq pc'$.

Demostración. Ver Anexo. □

5.2. Implementación

En esta sección presentaremos un EDSL para la programación segura en Haskell. Para la construcción del mismo implementamos el lenguaje presentado en la sección 3.4 mediante GADTs, donde los constructores del GADT pueden verse como una traducción de las reglas de tipado de la Figura 3. A través de esta codificación, la propiedad de seguridad que es asegurada por el sistema de tipos, resulta chequeada por el sistema de tipos de Haskell.

Para representar los niveles de seguridad definimos el siguiente tipo:

```
data SType = Low | High
```

Por simplicidad se definieron sólo dos niveles de seguridad, pero se podría extender fácilmente el lenguaje a varios niveles. A partir de esta definición si quisiéramos usar los valores `Low` y `High` a nivel de tipo, usando la extensión promoted data types, tendríamos los tipos `'Low` y `'High`, los cuales tienen kind `SType`. Sin embargo, a veces necesitamos para cada nivel de seguridad a nivel de tipos una copia a nivel de valores, por lo tanto también usaremos la siguiente definición de `SeType`, definida como singletons:

```
data SeType (s :: SType) where
  L :: SeType 'Low
  H :: SeType 'High
```

Cuando necesitemos una copia del nivel de seguridad a nivel de valores, utilizaremos el tipo `SeType 'Low` en lugar de `'Low` y `SeType 'High` en lugar de `'High`. De la misma manera, definiremos los números naturales como singletons:

```
data Nat = Zero | Succ Nat

data SNat (n :: Nat) where
  SZero :: SNat 'Zero
  SSucc :: SNat n -> SNat ('Succ n)
```

Las siguientes familias de tipos serán usadas en la definición de los GADTs para expresiones y comandos del lenguaje. La familia de tipos `Meet` calcula el mínimo entre dos tipos de seguridad mediante la siguiente función a nivel de tipos:

```
type family Meet (st :: SType) (st' :: SType) :: SType where
  Meet 'Low  x = 'Low
  Meet 'High x = x
```

De manera similar definimos `Join` que nos calculará el máximo:

```
type family Join (st :: SType) (st' :: SType) :: SType where
  Join 'Low  x = x
  Join 'High x = 'High
```

Para las operaciones entre conjuntos, pero definidas a nivel de tipos, definimos `Elem`, que verificará si un elemento se encuentra o no en un conjunto:

```
type family Elem (x :: a) (xs :: [a]) :: Bool where
  Elem x '[]       = 'False
  Elem x (x ': xs) = 'True
  Elem x (y ': xs) = Elem x xs
```

Para operar con valores booleanos a nivel de tipos, definimos la familia de tipos `IfThenElse`, que es un operador if-else, a nivel de tipos:

```
type family IfThenElse (b :: Bool) (t :: a) (u :: a) :: a where
  IfThenElse 'True  t u = t
  IfThenElse 'False t u = u
```

La unión e intersección las definimos de la siguiente manera:

```
type family Union (xs :: [a]) (ys :: [a]):: [a] where
  Union '[]       ys = ys
  Union (x ': xs) ys = x ': (Union xs ys)

type family Intersection (xs :: [a]) (ys :: [a]) :: [a] where
  Intersection '[] ys = '[]
  Intersection xs '[] = '[]
  Intersection (x ': xs) ys = IfThenElse (Elem x ys)
                                         (x ': Intersection xs ys)
                                         (Intersection xs ys)
```

Para buscar un elemento en una lista a nivel de tipos, definimos la familia de tipos `Lookup`, que nos devolverá el valor asociado a una clave en la lista:

```
type family Lookup (env :: [(k,st)]) (n :: k) :: a where
  Lookup ('(n, st) ': env) n = st
  Lookup ('(m, st) ': env) n = Lookup env n
```

Por último, definimos la clase `LEq` para modelar condiciones como $pc \sqsubseteq pc'$, como la usada en la regla de tipado IF_{sd}:

```
class LEq (a :: SType) (b :: SType)
instance LEq 'Low x
instance LEq 'High 'High
```

Notar que esta clase no contiene métodos: si la condición $pc \sqsubseteq pc'$ es cierta para los tipos pc y pc', entonces existirá una instancia de esta clase para pc y pc', y el sistema de tipos de Haskell la encontrará.

Continuaremos con la definición del tipo `Exp` que representa las expresiones del lenguaje:

```
data Exp :: [(Nat, SType)] -> SType -> [Nat] -> [Nat] -> * where

 Var :: SNat (n :: Nat) -> Exp env (Lookup env n) '[] '[n]

 IntLit :: Int -> Exp env 'Low '[] '[]

 BoolLit :: Bool -> Exp env 'Low '[] '[]

 Ope :: Op ->
        Exp env st d var1 ->
        Exp env st' d' var2 ->
        Exp env (Join st st') (Union d d') (Union var1 var2)

 Declassify :: Exp env l' d vars -> SeType l -> Exp env l vars vars
```

donde el tipo `Op` define las operaciones enteras y booleanas del lenguaje:

```
data Op = Plus | Minus | Mult | Div | Exp | Mod
          | And | Or | Gt | GtE | Lt | LtE | Eq | NotEq
```

El tipo `Exp` está indexado por el entorno con los tipos de seguridad de las variables, la cual tiene kind `[(Nat, SType)]`, el nivel de seguridad de la expresión `SType`, la lista de las variables que fueron desclasificadas y la lista de variables usadas en la expresión, ambas de kind `[Nat]`.

En nuestro sistema de tipos, el juicio de tipado `exp :: Exp env l D  D'` en Haskell se corresponde con el juicio $env \vdash_{sd} exp : l, D$, para cierto D. El parámetro `D'` no tiene su correspondencia en el sistema de tipos, ya que fue agregado para la implementación, donde es utilizado para recolectar las ocurrencias de las variables en las expresiones.

Dado que cada constructor del tipo Exp se corresponde con una regla del sistema de tipos, vemos aquí la necesidad de que el sistema de tipos sea dirigido por sintaxis.

Para modelar los comandos definimos un GADT también parametrizado por el nivel de seguridad, pero en este caso representa el nivel de seguridad del contexto en el que es ejecutado, es decir, el program counter:

```
data Stm :: [(Nat, SType)] -> SType -> [Nat] -> [Nat] -> * where

 Skip :: Stm env 'High '[] '[]

 Ass :: LEq st (Lookup env n) =>
        SNat (n :: Nat) ->
        Exp env st d var ->
        Stm env (Lookup env n) '[n] d

 Seq :: Intersection u1 d2 ~ '[] =>
        Stm env pc u1 d1 ->
        Stm env pc' u2 d2 ->
        Stm env (Meet pc pc') (Union u1 u2) (Union d1 d2)


 If  :: LEq st (Meet pc pc') =>
        Exp env st d vars ->
        Stm env  pc u1 d1 ->
        Stm env  pc' u2 d2 ->
        Stm env (Meet pc pc') (Union u1 u2) (Union d (Union d1 d2))

 While :: (Intersection u1 (Union d d1) ~ '[], LEq st pc) =>
          Exp env st d vars ->
          Stm env pc u1 d1 ->
          Stm env pc u1 (Union d d1)
```

El tipo `Stm` está indexado por el entorno de seguridad, el cual, como en el caso de las expresiones, tiene kind `[(Nat, SType)]`, el contexto de seguridad de kind `SType`, la lista de las variables que fueron actualizadas y la lista de variables desclasificadas, ambas de kind `[Nat]`.

Cada constructor se corresponde con una regla de la Figura 3. Para el caso de los comandos, el juicio de tipado `c :: Stm env pc U D` en Haskell se corresponde con el juicio: $env, pc \vdash_{sd} c : U, D$. En este caso D recolecta las variables desclasificadas, y U las variables que han sido actualizadas. Llegado a este punto cabe destacar que al tener todas las reglas de tipado codificadas dentro de los términos del GADT, ahora solo es posible escribir programas seguros, ya que los programas inseguros serán rechazados por el compilador.

Creemos importante destacar cómo funcionan las restricciones del while en el sistema de tipos:

```
(Intersection u1 (Union d d1) ~ '[], LEq st pc)
```

La primera de las restricciones nos dice que para que un programa que contenga un while sea aceptado por el sistema de tipos, necesariamente la intersección entre las variables desclasificadas no deben coincidir con las que fueron actualizadas previamente. Observamos esta misma restricción en la composición secuencial. La segunda restricción, también presente en un programa que contenga un if, es la que nos permite evitar flujos implícitos de información. Como mencionamos en la sección Preliminares, la manera de evitarlos es imponer una restricción entre el nivel de seguridad de la condición y el contexto en el cual se ejecuta el bucle.

5.3. Constructores del EDSL

A continuación presentaremos los constructores del lenguaje que se utilizarán para construir programas en el EDSL, para ello comenzaremos definiendo el tipo `HList`, el cual necesitaremos para formalizar el entorno de seguridad de las variables definidas en nuestro programa. Formalmente lo definimos de la siguiente manera:

```
infixr 5 :-:
data HList :: [(Nat , SType)] -> * where
  Nil :: HList '[]
  (:-:) :: (SNat n , SeType s) ->
                              HList xs ->
                                  HList ('(n , s) ': xs)
```

Mediante este tipo estamos definiendo una lista heterogénea de tipos, donde los elementos que componen la lista pueden ser de distintos tipos siempre que sean de kind: `(Nat, SType)`. Para nuestro propósito usaremos esta lista de pares definida a nivel de tipos para representar el entorno de seguridad, donde el primer elemento lo ocupará un número asociado a una variable, y su segundo elemento el nivel de seguridad asociado a ella. Para ejemplificar su uso, definiremos a continuación un entorno de seguridad de tres variables:

```
type Zero = 'Zero
type One = 'Succ 'Zero
type Two = 'Succ One

zero = SZero
one = SSucc zero
two = SSucc one

env = (zero, L) :-:
```

```
                (one, H) :-:
                              (two, H) :-: Nil
```

Continuando con los constructores, para declarar una variable utilizaremos el siguiente constructor:

```
var
  :: HList env
     -> SNat (n :: Nat)
     -> Exp env (Lookup env n) '[] (n ': '[])
var en n = Var n
```

Es decir, para poder declarar una variable, deberemos declarar primero el entorno de seguridad que utilizaremos a lo largo de todo el programa, el cual pasaremos como primer argumento. La declaración se completa agregando como segundo argumento el número que corresponde a la variable en la declaración del ambiente. Notar que para que esta expresión tipe el número asociado a la variable debe pertenecer al entorno. Como ejemplo, declararemos la variable pública `l`, en el mismo entorno definido previamente:

```
l = var env zero
```

Como mencionamos en la introducción, el lenguaje admite valores enteros y booleanos. Para poder utilizarlos, los definimos mediante los siguientes constructores:

```
infixr 6 `int`, `bool`
int :: Int -> Exp env 'Low '[] '[]
int = IntLit

bool :: Bool -> Exp env 'Low '[] '[]
bool = BoolLit
```

Con esta definición podremos escribir los literales en el EDSL como `int 3`, o `bool True`.

El constructor para las asignaciones es el siguiente:

```
infixr 2 =:
(=:)
  :: LEq st (Lookup env n) =>
     SNat (n : Nat)
     -> Exp env st d var
     -> Stm env (Lookup env n) '[n] d
(=:) n exp = Ass n exp
```

De esta manera, podemos asignar a la variable `l`, el literal entero 3, con la siguiente

declaración:

```
correctAssigment = zero =: int 3
```

Notar que en el lado izquierdo de una asignación usamos el número natural que representa a una variable en el entorno de seguridad, mientras que del lado derecho usamos el constructor `var` para referirnos a una variable.

La lista de operadores del lenguaje es la siguiente:

```
plus  = Ope Plus
minus = Ope Minus
mult  = Ope Mult
gt    = Ope Gt
gte   = Ope GtE
lt    = Ope Lt
lte   = Ope LtE
eq    = Ope Eq
neq   = Ope NotEq
expo  = Ope Exp
(//.) = Ope Div
(%.)  = Ope Mod
(&.)  = Ope And
(|.)  = Ope Or
```

Por último, los constructores para expresiones `declassify`, composición secuencial, skip, if y while son los siguientes:

```
declassify
  :: Exp Env l' d vars
     -> SeType l
     -> Exp Env l vars vars
declassify e l = Declassify e l

(\.)
  :: (Intersection u1 d2 ~ '[]) =>
     Stm Env pc u1 d1
     -> Stm Env pc' u2 d2
     -> Stm Env (Meet pc pc') (Union u1 u2) (Union d1 d2)
(\.) c1 c2 = Seq c1 c2

skip :: Stm env 'High '[] '[]
skip = Skip
```

```
iff
  :: LEq st (Meet pc pc') =>
     Exp env st d vars
     -> Stm env pc u1 d1
     -> Stm env pc' u2 d2
     -> Stm env (Meet pc pc') (Union u1 u2) (Union d (Union d1 d2))
iff c e1 e2 = If c e1 e2

while
  :: (LEq st pc,
     Intersection u1 (Union d d1) ~ '[]) =>
     Exp env st d vars
     -> Stm env pc u1 d1
     -> Stm env pc u1 (Union d d1)
while c e1 = While c e1
```

Daremos ejemplos de su uso en la siguiente sección. Finalmente, la función de evaluación para expresiones y comandos podemos encontrarla en el repositorio del proyecto.

5.4. Ejemplos

Implementaremos en nuestro lenguaje los programas mencionados en la sección 3.4.1.

Ejemplo 1. Promedio de salarios.

Para reescribir este mismo programa en nuestro lenguaje, primero debemos proveer un estado inicial y la declaración de las variables. Para ello declaramos el entorno de seguridad, es decir el natural que hace referencia a las variables junto al nivel de seguridad asociado a cada una de ellas, sus valores iniciales y luego les asociamos un nombre. Finalmente, el programa podría escribirse de la siguiente manera:

```
securityEnvironment = (zero, L) :-:
                        (one, H) :-:
                           (two, H) :-:
                             (three, H) :-:
                               (four, H) :-: Nil

memory = M.insert 0 2
                    (M.insert 1 3000
                        (M.insert 2 6000
                            (M.insert 3 3000 initMemory)))
```

```
avg = var securityEnvironment zero
h1 = var securityEnvironment one
h2 = var securityEnvironment two
h3 = var securityEnvironment three

averageSalaries = zero =: declassify ((h1 +. h2 +. h3) //. int 3) L
```

Si evaluamos el programa, podemos observar el resultado de avg almacenado en el primer par:

```
*Main> evalStmWithEnviroment averageSalaries memory
fromList [(0,4000),(1,3000),(2,6000),(3,3000)]
```

Para el caso donde se filtra información de manera no deseada, el programa quedaría escrito de la siguiente manera en nuestro EDSL:

```
avgAttack = one =: h2 \.
            three =: h2 \.
            averageSalaries
```

El programa es rechazado por nuestro sistema de tipos al filtrarse el salario de un empleado.

Ejemplo 2. Billetera electrónica.

Para reescribir este mismo programa en nuestro lenguaje, procedemos de la misma manera que el ejemplo anterior, proveemos un entorno de seguridad, un estado inicial, y la declaración de las variables:

```
securityEnvironment = (zero, H) :-:
                        (one, L) :-:
                            (two, L) :-: Nil

memory = M.insert 0 500
            (M.insert 1 0
                (M.insert 2 45 initMemory))


h = var securityEnvironment zero
l = var securityEnvironment one
k = var securityEnvironment two
```

```
secureElectronicWallet = iff (declassify (h  >. k) L)
                             (zero =: h -. k  \. one =: l +. k)
                                    skip
```

Si evaluamos el programa, podemos observar el resultado de las variables:

```
*Main> evalStmWithEnviroment secureElectronicWallet memory
fromList [(0,455),(1,45),(2,45)]
```

El ejemplo Wallet-Attack lo podemos reescribir de la siguiente manera:

```
walletAttack =
    one =: int 0 \.
    (while (n >. int 0)
           (two =: int 2 ^. n -. int 1 \.
           iff (declassify (h >=. k) L)
                           (zero =: h -. k \. one =: l +. k)
                            skip \.
                 three =: n -. int 1 \. skip))
```

Este programa va a ser rechazado por el compilador.

Ejemplo 3. Verificador de Contraseñas.

Consideremos por último un programa para verificar contraseñas. Basaremos nuestra implementación en una solución tipo UNIX, donde el usuario y la contraseña se guardan como un hash en una base datos. De esta manera, para que el usuario y la contraseña ingresada sea la correcta, estos datos deben coincidir con el hash guardado en la base de datos.

Para este programa, consideramos ahora la variable confidencial pw, que almacenará la contraseña ingresada por el usuario, $userId$, variable pública, que almacenará el nombre del usuario, $pwUserIdHash$ el hash del usuario y contraseña, y utilizaremos $newPw$ para utilizar en un programa que permita actualizar la contraseña. No explicaremos en detalle el resto de las variables ya que son de uso interno del programa.

Con todo lo dicho anteriormente, nuestro ambiente de seguridad, la base de datos y la declaración de variables son las siguientes:

```
env =
    (zero, H) :-:
        (one, L) :-:
            (two, H) :-:
                (three, H) :-:
```

```
                    (four, H) :-: Nil

database = M.insert 0 23
            (M.insert 1 45
                (M.insert 2 2391
                    (M.insert 3 0
                        (M.insert 4 67 initMemory))))

pw = var env zero
userId = var env one
pwUserIdHash = var env two
tmp = var env three
newPw = var env four
```

Para construir el hash a partir de la contraseña y el nombre de usuario utilizaremos la función de emparejamiento de Cantor. La siguiente expresión nos dará un número único, el cual nos permitirá utilizarlo como un hash, utilizando como argumentos el usuario y la contraseña:

```
buildHash =
    ((pw +. userId) *. (pw +. userId +. int 1)) //.
        int 2  +. userId
```

La expresión `hash` utiliza `buildHash` para construir el hash, desclasificando todo el resultado a nivel *low*:

```
hash = declassify buildHash L
```

La siguiente expresión o programa actualiza el hash generado por la contraseña antigua. El programa comienza verificando si hay una igualdad entre el hash generado por los datos que ingresa el usuario, con lo almacenado en la base de datos. En base a esto, si existe una coincidencia, un nuevo hash es generado a partir de la nueva contraseña:

```
update =
    iff (pwUserIdHash =. hash)
        (two =: declassify
            (((newPw +. userId) *.
                (newPw +. userId +. int 1))
            //. int 2  +. userId)
        L)
        skip
```

Si ejecutamos el programa, una nueva contraseña se almacenará en la variable 2:

```
evalStmWithEnviroment update database
fromList [(0,23),(1,45),(2,6373),(3,0),(4,67)]
```

Como vemos, todos los programas escritos son tipables y por ende seguros por construcción. Consideremos el siguiente programa, similar al del ejemplo del ataque a la seguridad de la billetera electrónica:

$l =: 0;$
`while` $n \geq 0$ `do`
$k =: 2^{n-1};$
`if` `hash`$(\texttt{sign}(h - k + 1), 0) = \texttt{hash}(1, 0)$ `then` $(h := h - k;\ l := l + k)$ `else`
`skip`;
$n =: n - 1$

Donde `hash` es una expresión que calcula un número hash entre dos números, similar a la explicada previamente, y `sign` la función signo.

Este programa puede filtrar información sobre la variable confidencial h, por lo que el programa será rechazado por el *type checker* de Haskell.

Para una posible implementación del programa en nuestro EDSL, comenzamos por el ambiente de seguridad, las variables, las expresiones que calculan los valores hash y algunas definiciones:

```
env =
    (zero, L) :-:
        (one, H) :-:
            (two, L) :-:
                (three, L) :-:
                    (four, L) :-: Nil

l = var env zero
h = var env one
k = var env two
n = var env three
temporalSignal = var env four

sign =
    iff ((declassify h L) -. k +. int 1 >. int 0)
        (four =: int 1)
        ((iff ((declassify h L) -. k +. int 1 <. int 0))
        (four =: int (-1))
        (four =: int 0))
```

```
hashSignal =
    ((temporalSignal +. int 0) *.
    (temporalSignal +. int 0 +. int 1)) //. int 2  +. int 1

hash10 =
    ((int 1 +. int 0)  *. (int 1 +. int 0 +. int 1))
            //. int 2  +. int 1
```

Finalmente, el programa en nuestro EDSL podríamos escribirlo de la siguiente manera:

```
pwAttack =
    zero =: int 0 \.
    while (n >. int 0)
        (sign  \.
        two =:  int 2 ^. (n -. int 1)   \.
             (iff (hashSignal  =. hash10 )
                (one =: h -. k \. zero =: l +. k)
                skip)
            \.
        three =:  n -. int 1)
```

Este programa también será rechazado por el *type checker* . En este caso, la variable h que es actualizada en el cuerpo del bucle, ocurre previamente en una expresión `declassify`.

6. Conclusión

6.1. Aportes

En los últimos años surgieron muchas investigaciones sobre cómo garantizar la confidencialidad de la información a través de sistemas de tipos que garanticen alguna propiedad de seguridad. Probablemente la propiedad más conocida sea la de no-interferencia. Sin embargo esta propiedad suele ser muy restrictiva en la práctica. La mayoría de los programas necesitan liberar información secreta como parte natural de su funcionalidad. La propiedad *delimited release*, propuesta por Sabelfeld y Myers [27], soluciona esta problemática. En su trabajo definen un modelo donde se permite liberar información mediante una expresión `declassify`, pero aseguran que esta liberación de información sea de manera controlada, es decir, que no se pueda utilizar la expresión `declassify` para liberar más información de la que se pretende. Los autores llaman a este tipo de abusos del lenguaje *laundering attacks*.

En esta tesina presentamos un EDSL en Haskell para programar en un lenguaje imperativo simple, tipo *while* y seguro respecto a la propiedad *delimited release*.

Para implementar el EDSL en Haskell primero modificamos el sistema de tipos original para que sea dirigido por sintaxis. Esto nos permitió codificar la sintaxis de los términos del lenguaje embebido junto con las reglas de tipado mediante GADTs.

Como necesitamos los tipos de datos dependientes para representar en los tipos de nuestro programa la propiedad de seguridad, utilizaremos también de algunas extensiones de Haskell que nos permiten programar, aunque de manera acotada, con tipos de datos dependientes.

Elegimos Haskell como huésped para nuestro EDSL porque es un lenguaje de propósito general y por la expresividad de su sistema de tipos. Si bien no es un lenguaje con tipos dependientes, sus extensiones nos permitieron emular los tipos de datos dependientes, los cuales fueron necesarios para la codificación de la propiedad de seguridad en el tipo de los términos del lenguaje.

Por último es necesario aclarar que a través de esta codificación, la propiedad de seguridad garantizada por el sistema de tipos también resulta chequeada por el type checker de Haskell. Al codificar esta información en los tipos, los términos son seguros por construcción, el sistema de tipos de Haskell solo tipará programas seguros.

El EDSL propuesto nos permite escribir programas seguros usando un conjunto de constructores simples, que el usuario puede utilizar sin necesidad de conocer algunos conceptos complejos de Haskell que requieran de cierta experiencia para ser aplicados correctamente.

La facilidad de su uso se ve reflejada en los distintos programas presentados en la tesina, como por ejemplo los de la billetera electrónica. Veremos una gran similitud si comparamos el pseudo-código presentado en el ejemplo 2 de la sección 3.4.1 con el mismo programa escrito en el EDSL.

6.2. Trabajo Relacionado

El ejemplo más conocido de un lenguaje de programación seguro con políticas de desclasificación, actualmente activo, es Jif [20]. Jif es un lenguaje que extiende Java, añadiendo etiquetas que expresan restricciones acerca de cómo la información es usada. Aunque este lenguaje ofrece las garantías de seguridad deseadas, introducir nuevas etiquetas a un lenguaje preexistente puede ser una solución demasiado costosa para regular el flujo de información.

Li y Zdancewic [16][15] proponen un enfoque diferente, en lugar de un nuevo lenguaje, definen una librería en Haskell para escribir código seguro mediante el control del flujo de información. En la implementación de la misma utilizan *arrows* y requiere que los programadores estén familiarizados con este concepto.

Un enfoque similar es el propuesto por Russo et al. en [25]. En este trabajo se presenta una librería también en Haskell para proveer seguridad mediante el control de flujo de información. En este caso, en lugar de arrows se utilizan mónadas, argumentando que las mónadas son más simples de usar que los arrows. Esta librería, a diferencia de la anterior incluye operaciones seguras para el manejo de archivos, permitiendo controlar los efectos de E/S para garantizar la seguridad de los programas. También incluye varias políticas de desclasificación, en este caso, realizada a través de funciones. Una diferencia con la librería anterior es que como los niveles de seguridad son representados con tipos, estos deben conocerse en tiempo de compilación y no de ejecución.

La principal diferencia de estos dos enfoques con el nuestro es que en ambos el control del flujo de información es realizado sobre el lenguaje Haskell, mientras que en nuestro EDSL el control de flujo de información se realiza sobre un lenguaje imperativo, el cual está embebido en Haskell.

6.3. Trabajo Futuro

A continuación detallamos algunas líneas de investigación que nos resultan las más prometedoras.

- Existen en la literatura científica varios lenguajes de programación seguros en donde aplicar nuestro enfoque, en particular podemos aplicar los mismos resultados que logramos en esta tesina para modelar otra propiedad de seguridad

con desclasificación: *robust declassification*. En esta propiedad, a diferencia de la que modelamos, en la que se controla qué se esta desclasificando, asegura que un atacante no puede controlar decisiones acerca de cuando la información es desclasificada.

Myers et al. [21] presentan un modelo similar, y de cierta manera ortogonal, al propuesto por Sabelfeld y Myers [27], del cual se basa este trabajo. Vemos una posible implementación de nuestros resultados en este modelo.

- Algunas extensiones al EDSL que podrían considerarse son:
    - Agregar un constructor al lenguaje que permita agregar variables con un nivel de seguridad dado, de esta manera el entorno de tipos de seguridad no sería fijo.
    - Agregar operadores que enriquezcan el lenguaje imperativo implementado como EDSL.
    - Podemos extender el lenguaje para contar con muchos niveles de seguridad.

7. Anexo

7.1. Demostración

Propiedad (i) Si $\Gamma \vdash_{sd} e : l, D$ entonces $\Gamma \vdash e : l, D$.

Demostración. La demostración es por inducción en la estructura de las sentencias. Los casos correspondientes a los valores, variables y sentencias `declassify` son inmediatos.

- Caso $e = e_1 \; op \; e_2$

  Si $e_1 \; op \; e_2$ es tipable, entonces, por regla OP_{sd} tenemos:

  - $\Gamma \vdash_{sd} e_1 : l, D_1$
  - $\Gamma \vdash_{sd} e_2 : l', D_2$
  - $\Gamma \vdash_{sd} e_1 \; op \; e_2 : l \sqcup l', D_1 \cup D_2$

  Por hipótesis de inducción tenemos: $\Gamma \vdash e_1 : l, D_1$ y $\Gamma \vdash e_2 : l', D_2$.

  Como $l \sqsubseteq l \sqcup l'$ y $l' \sqsubseteq l \sqcup l'$, podemos probar usando la regla $SUBS$ que $\Gamma \vdash e_1 : l \sqcup l', D_1$ y $\Gamma \vdash e_2 : l \sqcup l', D_2$. Entonces por regla OP podemos afirmar que $\Gamma \vdash e_1 \; op \; e_2 : l \sqcup l', D_1 \cup D_2$

□

Propiedad (ii) Si $\Gamma \vdash e : l, D$ entonces existe l' tal que $\Gamma \vdash_{sd} e : l', D$ y $l' \sqsubseteq l$.

Demostración. La demostración es por inducción en la derivación de $\Gamma \vdash e : l, D$.

- Si la última regla usada es alguna de las reglas VAL, EXP o DEC la prueba es inmediata.
- Si la última regla usada es OP, tenemos que:

  - $\Gamma \vdash e_1 : l, D_1$
  - $\Gamma \vdash e_2 : l, D_2$
  - $\Gamma \vdash e_1 \; op \; e_2 : l, D_1 \cup D_2$

  Por hipótesis de inducción existen l y l' tal que $\Gamma \vdash_{sd} e_1 : l', D_1$, $\Gamma \vdash_{sd} e_2 : l'', D_2$ con $l' \sqsubseteq l$ y $l'' \sqsubseteq l$. Además como $l' \sqsubseteq l$ y $l'' \sqsubseteq l$, $l' \sqcup l'' \sqsubseteq l$. Por la regla OP_{sd} concluimos que:

$$\Gamma \vdash_{sd} e_1 \ op \ e_2 : l' \sqcup l'', D_1 \cup D_2$$

- Si la última regla usada es $SUBE$, tenemos que:
  - $\Gamma \vdash e : l, D$
  - $l \sqsubseteq l'$
  - $\Gamma \vdash e : l', D$

  Por hipótesis de inducción, existe l'', tal que $\Gamma \vdash_{sd} e : l'', D$ con $l'' \sqsubseteq l$. Como $l \sqsubseteq l'$, $l'' \sqsubseteq l'$ por transitividad.

□

Propiedad (iii) Si $\Gamma, pc \vdash_{sd} c : U, D$ entonces $\Gamma, pc \vdash c : U, D$

Demostración. La demostración es por inducción en la estructura de los comandos. Los casos correspondientes a las asignaciones y las expresiones skip son inmediatos.

- Caso $c = if \ e \ then \ c_1 \ else \ c_2$

  Si $if \ e \ then \ c_1 \ else \ c_2$ es tipable, entonces, por regla IF_{sd} tenemos:
  - $\Gamma \vdash_{sd} e : l, D$
  - $\Gamma, pc \vdash_{sd} c_1 : U_1, D_1$
  - $\Gamma, pc' \vdash_{sd} c_2 : U_2, D_2$
  - $\Gamma, pc \sqcap pc' \vdash_{sd} if \ e \ then \ c_1 \ else \ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2$
  - $l \sqsubseteq pc \sqcap pc'$

  Por propiedad (i) sabemos que $\Gamma \vdash e : l, D$, y por hipótesis de inducción tenemos $\Gamma, pc \vdash c_1 : U_1, D_1$ y $\Gamma, pc' \vdash c_2 : U_2, D_2$.

  Como $pc \sqcap pc' \sqsubseteq pc$ y $pc \sqcap pc' \sqsubseteq pc'$, podemos probar usando la regla $SUBC$ que $\Gamma, pc \sqcap pc' \vdash c_1 : U_1, D_1$ y $\Gamma, pc \sqcap pc' \vdash c_2 : U_2, D_2$. Por regla IF concluimos que $\Gamma, pc \sqcap pc' \vdash if \ e \ then \ c_1 \ else \ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2$.

- Caso $c = while \ e \ do \ c'$

  Si $while \ e \ do \ c'$ es tipable, entonces, por regla $WHILE_{sd}$ tenemos:
  - $\Gamma \vdash_{sd} e : l, D$
  - $\Gamma, pc \vdash_{sd} c_1 : U_1, D_1$

- $l \sqsubseteq pc$
- $U_1 \cap (D \cup D_1) = \emptyset$
- $\Gamma, pc \vdash_{sd} while\ e\ do\ c_1 : U_1, D \cup D_1$

Por propiedad (i) sabemos que $\Gamma \vdash e : l, D$, y por hipótesis de inducción tenemos $\Gamma, pc \vdash c_1 : U_1, D_1$.

Como $l \sqsubseteq pc$, podemos afirmar que $l \sqcup pc \sqsubseteq pc$, y por regla $SUBC$ $\Gamma, l \sqcup pc \vdash c_1 : U_1, D_1$.

Por regla $WHILE$ concluimos que $\Gamma, pc \vdash while\ e\ do\ c_1 : U_1, D \cup D_1$

- Caso $c = c_1; c_2$

Si $c_1; c_2$ es tipable, entonces, por regla SEQ_{sd} tenemos:

- $\Gamma, pc \vdash_{sd} c_1 : U_1, D_1$
- $\Gamma, pc' \vdash_{sd} c_2 : U_2, D_2$
- $U_1 \cap D_2 = \emptyset$
- $\Gamma, pc \sqcap pc' \vdash_{sd} c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2$

Por hipótesis de inducción tenemos $\Gamma, pc \vdash c_1 : U_1, D_1$ y $\Gamma, pc' \vdash_{sd} c_2 : U_2, D_2$.

Como $pc \sqcap pc' \sqsubseteq pc$ y $pc \sqcap pc' \sqsubseteq pc'$, podemos probar usando la regla $SUBC$ que $\Gamma, pc \sqcap pc' \vdash c_1 : U_1, D_1$ y $\Gamma, pc \sqcap pc' \vdash c_2 : U_2, D_2$. Por regla SEQ concluimos que $\Gamma, pc \sqcap pc' \vdash c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2$.

□

Propiedad (iv) Si $\Gamma, pc \vdash c : U, D$ entonces existe un pc', tal que $\Gamma, pc' \vdash_{sd} c : U, D$ y $pc \sqsubseteq pc'$

Demostración. La demostración es por inducción en la derivación $\Gamma, pc \vdash c : U, D$. Cuando la última regla usada en la derivación es $ASIG$ o $SKIP$ el resultado es inmediato.

- Cuando la última regla usada en la derivación es SEQ tenemos:
  - $\Gamma, pc \vdash c_1 : U_1, D_1$
  - $\Gamma, pc \vdash c_2 : U_2, D_2$
  - $U_1 \cap D_2 = \emptyset$
  - $\Gamma, pc \vdash c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2$

Por hipótesis de inducción existe pc' tal que $\Gamma, pc' \vdash_{sd} c_1 : U_1, D_1$ y pc'' tal que $\Gamma, pc'' \vdash_{sd} c_2 : U_2, D_2$ con $pc \sqsubseteq pc'$ y $pc \sqsubseteq pc''$.

Como $pc \sqsubseteq pc' \sqcap pc''$, por regla SEQ_{sd} concluimos que:

$$\Gamma, pc' \sqcap pc'' \vdash_{sd} c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2$$

- Cuando la última regla usada en la derivación es IF tenemos:
  - $\Gamma \vdash e : l, D$
  - $\Gamma, pc \sqcup l \vdash c_1 : U_1, D_1$
  - $\Gamma, pc \sqcup l \vdash c_2 : U_2, D_2$
  - $\Gamma, pc \vdash if\ e\ then\ c_1\ else\ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2$

  Por (ii) sabemos que $\Gamma \vdash_{sd} e : l', D$, con $l' \sqsubseteq l$.

  Por hipótesis de inducción tenemos que existe un pc' tal que $\Gamma, pc' \vdash_{sd} c_1 : U_1, D_1$ con $pc \sqcup l \sqsubseteq pc'$ y existe un pc'' tal que $\Gamma, pc'' \vdash_{sd} c_2 : U_2, D_2$ con $pc \sqcup l \sqsubseteq pc''$. Además como $pc \sqcup l \sqsubseteq pc' \sqcap pc''$, concluimos que $pc \sqsubseteq pc' \sqcup pc''$.

  Como $l' \sqsubseteq l$ y $pc \sqcup l \sqsubseteq pc'$, concluimos que $pc \sqcup l' \sqsubseteq pc'$. Similarmente concluimos que $pc \sqcup l' \sqsubseteq pc''$, con lo cual $pc \sqcup l' \sqsubseteq pc' \sqcap pc''$ y por lo tanto $l' \sqsubseteq pc' \sqcap pc''$. Por regla IF_{sd} concluimos que:

$$\Gamma, pc' \sqcap pc'' \vdash_{sd} if\ e\ then\ c_1\ else\ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2$$

- Cuando la última regla usada en la derivación es $WHILE$ tenemos:
  - $\Gamma \vdash e : l, D$
  - $\Gamma, pc \sqcup l \vdash c_1 : U_1, D_1$
  - $U_1 \cap (D \cup D_1) = \emptyset$
  - $\Gamma, pc \vdash while\ e\ do\ c_1 : U_1, D \cup D_1$

  Por (ii) sabemos que $\Gamma \vdash_{sd} e : l', D$, con $l' \sqsubseteq l$.

  Por hipótesis de inducción tenemos que existe un pc' tal que $\Gamma, pc' \vdash c_1 : U_1, D_1$ y $pc \sqcup l \sqsubseteq pc'$. Concluyo que $pc \sqcup l' \sqsubseteq pc'$ y por lo tanto que $l' \sqsubseteq pc'$. Además como $pc \sqcup l' \sqsubseteq pc'$, concluimos que $pc \sqsubseteq pc'$.

  Por regla $WHILE_{sd}$ concluimos que:

$$\Gamma, pc' \vdash_{sd} while\ e\ do\ c_1 : U_1, D \cup D_1$$

- Si la última regla usada es $SUBC$, tenemos que:
  - $\Gamma, pc \vdash c : U, D$
  - $pc' \sqsubseteq pc$
  - $\Gamma, pc' \vdash c : U, D$

  Procedemos por inducción en la estructura de los comandos. Los casos para comandos skip y asignación son inmediatos.

  - Caso $c = c_1; c_2$

    La única regla que puede ser usada en la derivación de $c_1; c_2$ es la regla SEQ, en este caso tenemos:

    - $\Gamma, pc \vdash c_1 : U_1, D_1$
    - $\Gamma, pc \vdash c_2 : U_2, D_2$
    - $\Gamma, pc \vdash c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2$
    - $U_1 \cap D_2 = \emptyset$

    Por hipótesis de inducción tenemos que existe pc_1 tal que $\Gamma, pc_1 \vdash_{sd} c_1 : U_1, D_1$ con $pc \sqsubseteq pc_1$ y existe pc_2 tal que $\Gamma, pc_2 \vdash_{sd} c_2 : U_2, D_2$ con $pc \sqsubseteq pc_2$.

    Por regla SEQ_{sd} concluimos que:

    $$\Gamma, pc_1 \sqcap pc_2 \vdash_{sd} c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2$$

    donde $pc \sqsubseteq pc_1 \sqcap pc_2$.

  - Caso $c = if\ e\ then\ c_1\ else\ c_2$

    La única regla que puede ser usada en la derivación de $if\ e\ then\ c_1\ else\ c_2$ la regla IF, en este caso tenemos:

    - $\Gamma \vdash e : l, D$
    - $\Gamma, l \sqcup pc \vdash c_1 : U_1, D_1$
    - $\Gamma, l \sqcup pc \vdash c_2 : U_2, D_2$
    - $\Gamma, pc \vdash if\ e\ then\ c_1\ else\ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2$

Por propiedad (ii) sabemos que $\Gamma \vdash_{sd} e : l', D$, $l' \sqsubseteq l$ y por hipótesis de inducción tenemos existe pc_1 tal que $\Gamma, pc_1 \vdash_{sd} c_1 : U_1, D_1$ con $l \sqcup pc \sqsubseteq pc_1$ y existe pc_2 tal que $\Gamma, pc_2 \vdash_{sd} c_2 : U_2, D_2$ con $l \sqcup pc \sqsubseteq pc_2$.

Como $l' \sqsubseteq l$ y $pc \sqcup l \sqsubseteq pc_1$, concluimos que $pc \sqcup l' \sqsubseteq pc_1$. Similarmente concluimos que $pc \sqcup l' \sqsubseteq pc_2$, con lo cual $pc \sqcup l' \sqsubseteq pc_1 \sqcap pc_2$. Por lo tanto $l' \sqsubseteq pc_1 \sqcap pc_2$ y $pc \sqsubseteq pc_1 \sqcap pc_2$. Por regla IF_{sd} concluimos que:

$$\Gamma, pc_1 \sqcup pc_2 \vdash_{sd} if\ e\ then\ c_1\ else\ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2$$

- El caso $c = while\ e\ do\ c'$ es análogo al anterior.

□

7.2. Roadmap

En `https://github.com/gonzalodelatorre/delimitedRelease` encontraremos el repositorio Git del proyecto. El mismo está organizado como se muestra en la Figura 4. Bajo la carpeta `Source` vamos a encontrar el código principal del EDSL implementado en esta tesina. A continuación describiremos brevemente el contenido de los archivos más relevantes. En `DelimitedRelease.hs` se encuentra la definición del lenguaje seguro mediante GADTs y type families, en `Constructors.hs` están los constructores que podrá utilizar un usuario del lenguaje y en `Environment.hs` vamos a encontrar la definición del tipo del ambiente de seguridad..

En el resto de los archivos podemos encontrar ejemplos de programas escritos en el EDSL, incluidos los presentados en esta tesina.

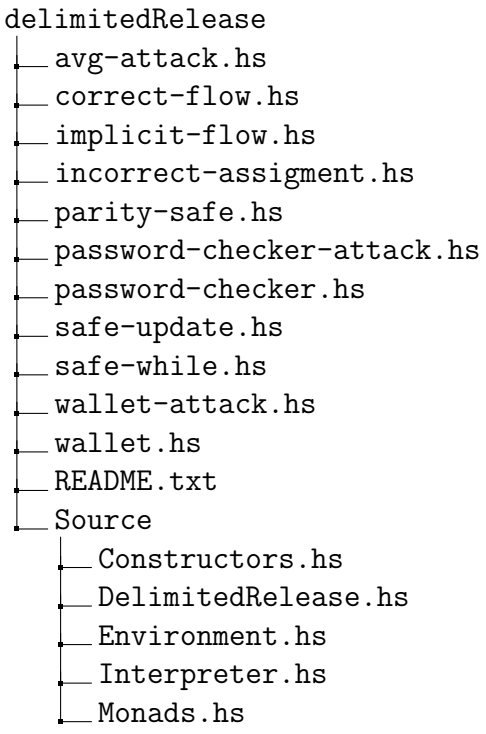

Figura 4: Estructura del directorio del proyecto.

8. Referencias

[1] DoD 5200.28-STD. *Trusted Computer System Evaluation Criteria*. Dod Computer Security Center, December 1985.

[2] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation, 1976.

[3] Ana Bove and Peter Dybjer. Dependent types at work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer, 2008.

[4] Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. Associated type synonyms. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 241–253. ACM, 2005.

[5] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick D. McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 198–209. ACM, 2004.

[6] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[7] Richard A. Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, abs/1610.07978, 2016.

[8] Andy Gill. Domain-specific languages and code synthesis using haskell. *Commun. ACM*, 57(6):42–49, 2014.

[9] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[10] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[11] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[12] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61. ACM, 2006.

[13] Dexter Kozen. Language-based security. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 1999.

[14] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[15] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, page 16. IEEE Computer Society, 2006.

[16] Peng Li and Steve Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci.*, 411(19):1974–1994, 2010.

[17] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 81–92. ACM, 2013.

[18] Cecilia Manzino and Alberto Pardo. A security types preserving compiler in haskell. In Fernando Magno Quintão Pereira, editor, *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings*, volume 8771 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2014.

[19] Conor McBride. Faking it: Simulating dependent types in haskell. *J. Funct. Program.*, 12(4&5):375–392, 2002.

[20] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241. ACM, 1999.

[21] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 172–186. IEEE Computer Society, 2004.

[22] Ulf Norell. Towards a practical programming language based on dependent type theory, 2007.

[23] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer, 2004.

[24] François Pottier and Sylvain Conchon. Information flow inference for free. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 46–57. ACM, 2000.

[25] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In Andy Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 13–24. ACM, 2008.

[26] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.

[27] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[28] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, 2009.

[29] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. In *Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, France, 1981*, pages 509–512. IEEE Computer Society, 1981.

[30] Tom Schrijvers, Simon L. Peyton Jones, Manuel M. T. Chakravarty, and Martin Sulzmann. Type checking with open type functions. In James Hook and

Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 51–62. ACM, 2008.

[31] Tim Sheard. Languages of the future. *ACM SIGPLAN Notices*, 39(12):119–132, 2004.

[32] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2/3):167–188, 1996.

[33] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/-FASE, Lille, France, April 14-18, 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer, 1997.

[34] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989.

[35] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

[36] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In Benjamin C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 53–66. ACM, 2012.