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

TYPE-BASED DECLASSIFICATION EN DART: IMPLEMENTACIÓN Y ELABORACIÓN DE HERRAMIENTAS DE INFERENCIA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS IGNACIO MENESES CORTÉS

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:

SANTIAGO DE CHILE
ABRIL 2018

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: MATÍAS IGNACIO MENESES CORTÉS
FECHA: ABRIL 2018
PROF. GUÍA: ÉRIC TANTER

TYPE-BASED DECLASSIFICATION EN DART: IMPLEMENTACIÓN Y ELABORACIÓN DE HERRAMIENTAS DE INFERENCIA

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Una dedicatoria corta. Por ejemplo, A los creadores de U-Campus

Agradecimientos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Introducción

La protección de la confidencialidad de la información manipulada por los programas computacionales es un problema cuya relevancia se ha incrementado en el último tiempo, a pesar de tener varias décadas de investigación. Por ejemplo, una aplicación web (o móvil) que como parte de su funcionamiento debe interactuar con servicios de terceros y por tanto debe proteger que su información sensible no se escape durante la ejecución de la aplicación a canales públicos.

Muchas de las técnicas de seguridad convencionales como *control de acceso* tienen deficiencias para proteger la confidencialidad de un programa, por ejemplo no restringen la propagación de información[3].

Formas más expresivas y efectivas de proteger la confidencialidad se basan en un análisis estático sobre el código del programa, y se categorizan dentro de *language-based security*. Una de las técnicas más efectivas se denomina *tipado de seguridad* en un *lenguaje de seguridad*, donde los tipos son anotados con niveles de seguridad para clasificar la información manipulada por el programa.

Uno de los mayores desafíos de los lenguajes de seguridad es facilitar el trabajo del programador, utilizando técnicas más expresivas. En esta dirección, Cruz et al. [1] recientemente propusieron *type-based declassification*, una variación de tipado de seguridad que utiliza el sistema de tipos del lenguaje para controlar la declasificación de la información.

Type-based declassification presenta limitaciones en cuanto a su implementación, debido a que el análisis teórico se realiza sobre un lenguaje minimalista que no incluye características básicas de los lenguajes de programación, como instrucciones condicionales y mutabilidad. En este trabajo, se propone una extensión al lenguaje minimalista utilizado en *type-based declassification*, y la implementación en el lenguaje de programación Dart, desarrollando un plugin para los editores de texto más populares, con el objetivo de proporcionar una experiencia interactiva e intuitiva al usuario.

1.1. Motivación

El tema escogido se considera interesante debido a su importancia teórica y práctica en el campo de la seguridad en lenguajes de programación. Los fundamentos teorícos de *type-*

based declassification están bien descritos [1], pero no así su realización práctica, siendo esto reconocido por sus autores.

Se considera que la realización de este trabajo es relevante para demostrar y materializar en un lenguaje de uso general la investigación de type-based declassification. Dicha materalización constituye un desafío importante en términos de complejidad, al tener que entender la teoría subyacente de tipado de seguridad, así como extender un lenguaje real y sus herramientas de análisis estático con nuevas funcionalidades.

1.2. Objetivos

El objetivo de la memoria es realizar la implementación de type-based declassification en Dart. Dentro de los objetivos específicos del trabajo, podemos encontrar:

- **Verificación estática de type-based declassification**. Se entiende coma la implementación del sistema de tipos de *type-based declassification* en un herramienta de análisis estático para un subconjunto del lenguaje Dart.
- **Inferencia de tipos de declasificacion**. Desarrollar una herramienta, que dado un código fuente de Dart, realice una inferencia de las facetas de declasificación, evitando así que el programador tenga que anotarlas inicialmente. Luego del proceso de inferencia se generara un código equivalente con los tipos inferidos. Sobre este nuevo código el programador podrá realizar las modificaciones que considere pertinentes y usar las herramienta de verificación.
- **Plugin para editores**. Integrar los objetivos anteriores mediante un plugin para los editores de texto que soporten servidores de análisis estático de Dart, con el objetivo de mostrar al usuario el resultado del análisis estático de type-based declassification, y ofrecerle acciones al respecto.

1.3. Resultados y Organización del documento

En términos concretos este trabajo presenta el diseño de un plugin que realiza el análisis de type-based declassification. Los antecedentes teóricos necesarios para entender este trabajo se abordan en el capítulo 2, mientras que la propuesta de solución es desarrollada en el capítulo 3.

Los detalles de diseño de implementación de la propuesta son revisados en el capítulo 4.

Por terminar.

Capítulo 2

Antecedentes

En este capítulo se discuten los antecedentes y el marco teórico necesario para poder entender este trabajo. Además, se muestran las definiciones formales utilizadas en el resto del documento.

2.1. Information flow control

El problema de la protección de confidencialidad de la información ha sido enfrentado principalmente mediante *control de acceso*, método que tiene deficiencias debido a que no restringe la propagación de información. [3].

Un mecanismo más expresivo para la protección de la confidencialidad e integridad de la información se denomina *information flow control*. Mientras control de acceso restringe qué datos pueden ser accedidos, *information flow control* restringe el flujo de estos datos.

2.1.1. Tipado de seguridad

El análisis de *information flow control* puede ser realizado sobre el código del programa, de forma estática o dinámica. Una de las técnicas más efectivas de *information flow control* con análisis estático es *tipado de seguridad* en un *lenguaje de seguridad*. En un lenguaje de seguridad, los valores y los tipos son anotados con niveles de seguridad para clasificar la información que el programa manipula. Dichos niveles de seguridad forman una *lattice*[1].

Por ejemplo con la *lattice* de dos niveles de seguridad $L \sqsubseteq H$, se puede distinguir entre valores enteros públicos o de baja confidencialidad (Int_L) y valores enteros privados o de alta confidencialidad (Int_H). El sistema de tipos usa estos niveles de seguridad para prevenir que la información confidencial no fluya directa o indirectamente hacia canales públicos[8].

[1]Un orden parcial, donde todo par de elementos tiene un único supremo e ínfimo

Figura 2.1: *lattice* de dos niveles de seguridad

```
String @H login(String @L guess, String @H password) {
    if (password == login) return "Login successful";
    else return "Login failed";
}
```

En el código anterior, los tipos de los parámetros y el tipo de retorno de la función están etiquetados con niveles de seguridad. En este ejemplo, la operación `password == login` tiene un nivel de seguridad `@H`, debido a que las operaciones binarias tienen un nivel de seguridad que corresponde a la operación `join` en la *lattice* a la cual pertenecen los elementos. Como el nivel de seguridad se propaga desde la condición del `if` hacia ambas ramas, el tipo de retorno de la función también tiene un nivel de seguridad `@H`.

2.1.2. Flujo explícito

Se denomina flujo explícito a las instrucciones del programa que directamente asignan un valor a una variable con distintos niveles de seguridad. El siguiente programa ilustra el flujo explícito.

```
void @L foo(int @H highVar, int @L lowVar) {
        int @H v1 = lowVar;
        int @L v2 = highVar;
}
```

En el ejemplo, la primera asignación no representa un riesgo de seguridad, puesto que se asigna un valor público a una variable confidencial. Por otra parte, la segunda asignación es insegura, debido a que se asigna un valor confidencial a una variable pública, que luego puede ser utilizada en contextos no deseados.

2.1.3. Flujo implícito

Se denomina flujo implícito a las instrucciones del programa que indirectamente dan conocimiento de algún aspecto de una variable, usualmente mediante instrucciones condicionales. Podemos adaptar el mismo programa de `login` visto anteriormente para ilustrar el flujo implícito.

```
String @L login(String @L guess, String @H password) {
```

```
    String @L ret;
    if (password == login) ret = "Login successful";
    else ret = "Login failed";
    return ret;
}
```

En este ejemplo, las instrucciones de asignación a la variable `ret` son seguras por si solas, pero no lo son considerando que su ejecución depende del valor de una variable confidencial, en este caso `password`.

2.1.4. No-interferencia

La propiedad para formalizar y caracterizar la seguridad y confidencialidad en los lenguajes con tipado de seguridad se denomina *no-interferencia (noninterference)*. A grandes rasgos noninterference expresa que para dos ejecuciones realizadas por el adversario de un programa seguro, con valores confidenciales equivalentes, las salidas del programa deben ser equivalentes para el adversario. Esto caracteriza que el adversario no aprende nada sobre los valores confidenciales de un programa.

El programa de `login` es un buen ejemplo para ilustrar el concepto de noninterference.

```
String @H login(String @L guess, String @H password) {
    if (password == login) return "Login successful";
    else return "Login failed";
}
```

Este programa no cumple con noninterference, debido a que el adversario puede aprender sobre la variable confidencial `password` observando el valor de retorno del método para distintas ejecuciones.

2.1.5. Declasificación

En una aplicación real y práctica deseamos que el programa anterior sea aceptado a pesar de violar la propiedad de no-interferencia, pues de otra forma no tendríamos cómo realizar la autenticación. Para solucionar este problema, los lenguajes de seguridad adicionan mecanismos para *declasificar* la información confidencial, implementados de diferentes formas[6]. Una de ellas, por ejemplo en Jif (un lenguaje de seguridad)[4] es usar un operador `declassify`, como se indica en el siguiente ejemplo, declasificando la comparación de igualdad del parámetro confidencial `password` con el parámetro público `guess`

```
String login(String password, String guess) {
    if (declassify(password == guess)) return "Login Successful";
    else return "Login failed";
}
```

Esto no corresponde a una amenaza de seguridad, debido a que el resultado de la operación de comparación es negligible con respecto al parámetro privado `password`. Sin embargo, usos arbitrarios del operador `declassify` pueden resultar en serias fugas de información, como por ejemplo `declassify(password)`.

2.2. Type-based declassification

Varios mecanismos se han explorado para controlar el uso de declasificación, y poder asegurar además una propiedad de seguridad para el programa[6]. En esta dirección, Cruz et al.[1] recientemente propusieron *type-based declassification* como un mecanismo de declasificación que conecta la abstracción de tipos con una forma controlada de declasificación, en una manera intuitiva y expresiva, proveyendo garantías formales sobre la seguridad del programa.

2.2.1. Sistema de tipos

En *type-based declassification* los tipos tienen dos facetas, una que refleja el tipo de implementación y otro tipo que refleja las operaciones de declasificación sobre los valores de dicho tipo. Por ejemplo, el tipo StringEq $\triangleq$ [eq : String $\rightarrow$ Bool] autoriza la operación `eq` sobre un String. Entonces se puede usar el tipo de dos facetas String $<$ StringEq, en donde String es un subtipo de StringEq, para controlar la operación de declasificación de la igualdad sobre `password`.

```
String login(String<StringEq password, String guess) {
    if (password.eq(guess)) return "Login successful";
    else return "Login failed";
}
```

Al igual que en tipado de seguridad con etiquetas (@L y @H), la faceta de declasificación es parte de la jerarquía de tipos, la que forma una *lattice* como se muestra en la figura siguiente.

De la misma forma, el tipo de las operaciones binarias se resuelve utilizando el operador `join` sobre la *lattice*.

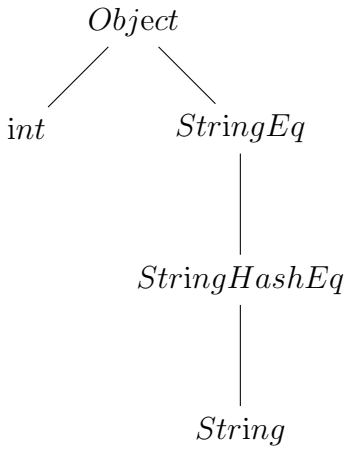

Figura 2.2: *lattice* de tipos de dos facetas

2.2.2. Formalización

Para formalizar y demostrar propiedades del sistema de tipos, se utilizó el lenguaje Ob_{SEC}, que se muestra en la figura.

$$
\begin{array}{rcll}
e & ::= & v \mid e.m(e) \mid x & \text{(terms)} \\
v & ::= & \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] & \text{(values)} \\
T, U & ::= & O \mid \alpha & \text{(types)} \\
O & ::= & \mathbf{Obj}(\alpha).\ \left[\overline{m : S \to S}\right] & \text{(object type)} \\
S & ::= & T \triangleleft U & \text{(security type)}
\end{array}
\qquad
\begin{array}{rl}
x, y, z & \text{(variables)} \\
\alpha, \beta & \text{(type variables)} \\
m & \text{(method labels)}
\end{array}
$$

$$T_{\mathsf{L}} \triangleq T \triangleleft T \qquad T_{\mathsf{H}} \triangleq T \triangleleft \top$$

Figura 2.3: Sintaxis de Ob_{SEC}

Ob_{SEC} es un lenguaje minimalista, y por tanto no soporta características comunes de lenguajes de programación, como asignaciones a variables y condicionales. La justificación de su uso se debe a que es suficiente para formular y demostrar la proposición.

Type Safety

Se demostró que programas de Ob_{SEC} bien tipados son *safe*[2].

Relaxed noninterference

La propiedad de seguridad que se demuestra para Ob_{SEC} es una forma de no-interferencia con políticas de declasificación, denominada *Relaxed noninterference*. Un lenguaje de seguridad que tiene esta propiedad, garantiza que la información confidencial solo puede fluir hacia canales públicos de una forma controlada, por medio de las políticas de declasificación[1].

[2]Sin errores de tipos en tiempo de ejecución

2.3. Inferencia de tipos de seguridad

La inferencia de tipos se refiere a la detección automática del tipo de una expresión en un lenguaje de programación. En *Security Typing*, las expresiones también poseen un tipo de seguridad, el cual puede estar implícito o no especificado, por lo que es posible desarrollar un algoritmo de inferencia para tipos de seguridad.

En esta dirección, se han desarrollado modelos de inferencia de tipos de seguridad, para lenguajes orientados a objetos[7] y lenguajes funcionales[5], con el objetivo de extender los modelos simplificados para los cuales se demuestran las propiedades de seguridad relevantes.

Para llevar registro del nivel de seguridad en cada instrucción del programa, se utiliza el concepto de *program counter* (PC) para seguridad[2]. Este registro es utilizado por los modelos de inferencia de tipos de seguridad, y para realizar el análisis de seguridad que detecta la existencia de fugas de información.

2.4. Lenguaje Dart

Dart es un lenguaje de programación de propósito general, orientado a objetos y de código abierto desarrollado por Google. Es usado para construir aplicaciones web, móviles y dispositivos IoT (Internet of Things).

Dart es un lenguaje de creciente adopción. Actualmente, la interfaz de Google Adwords está construida sobre Dart y Angular2. Además, Dart es usado en el framework de desarollo multiplataforma *Flutter*.

El lenguaje Dart fue escogido porque los investigadores que realizaron el trabajo de *type-based declassification* estudian este lenguaje como parte de un proyecto de investigación mayor en el área de seguridad. Es factible implementar el enfoque de *type-based declassification* en Dart, dado que fue formalizado considerando un lenguaje minimalista orientado a objetos (Ob_{SEC}).

Se utilizaron de forma intensiva dos librerías en el desarrollo de este trabajo.

2.4.1. Dart Analyzer

La herramienta *Dart Analyzer* sirve para realizar análisis estático de código Dart. Al usarse como librería, se puede obtener el *Abstract Syntax Tree* (AST)[3] del código, el cual contiene, en sus nodos, la información de los distintos elementos que conforman la sintaxis, y también la inferencia de tipos resuelta.

Por ejemplo, para el siguiente código en Dart

[3]Representación de árbol de la estructura sintáctica abstracta del código fuente

```
class Bar {
    inf foo(int a) {
        var b = 2*a;
        return b;
    }
}
```

Dart Analyzer genera el siguiente AST, en donde se indica el tipo de nodo y la expresión al cual corresponde

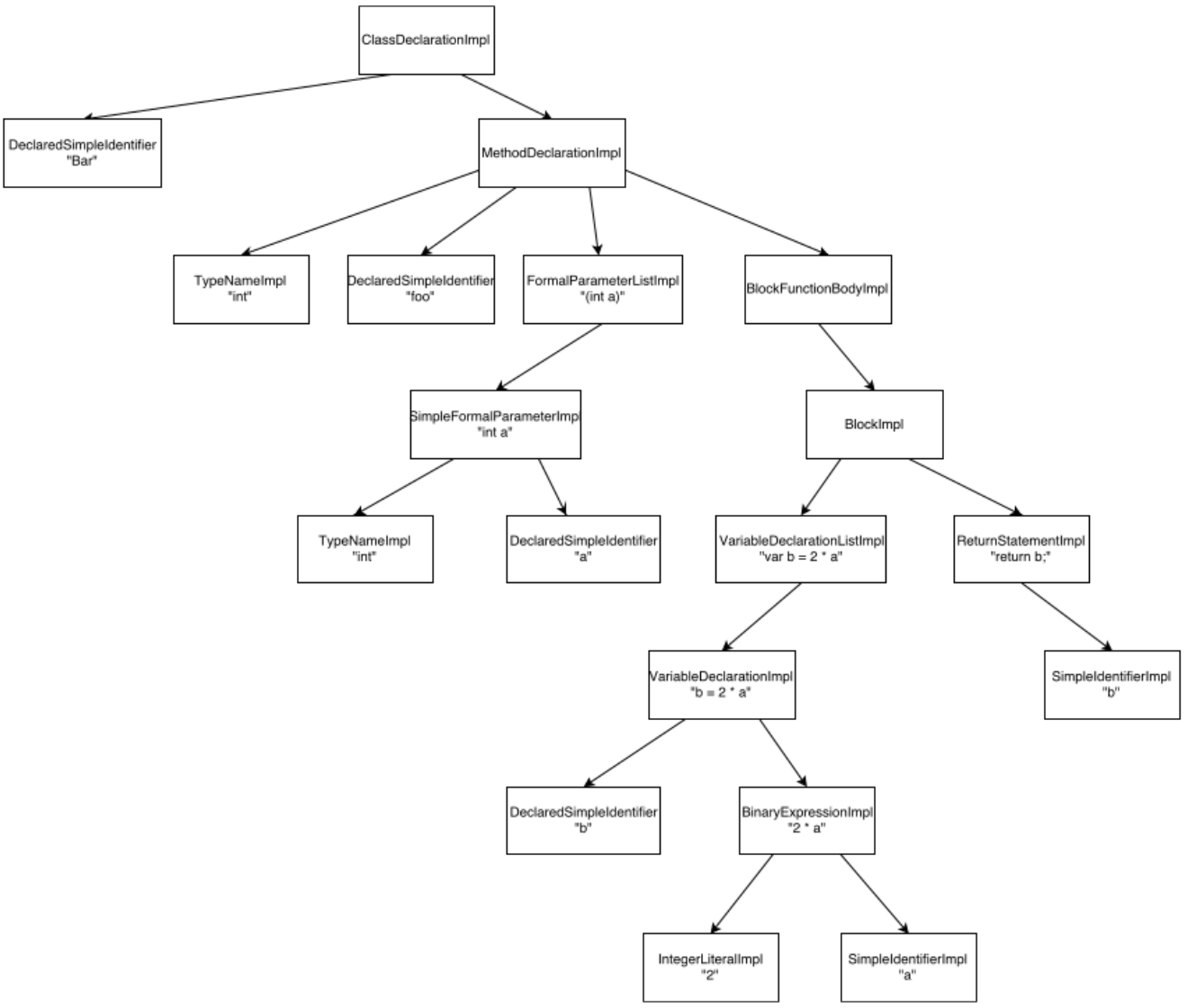

Figura 2.4: Representación del AST generado por Dart Analyzer

Con Dart Analyzer es posible realizar análisis personalizado sobre el AST, utilizando el patrón de diseño *Visitor*.

2.4.2. Analyzer Plugin

La herramienta *Analyzer Plugin* sirve para integrar un análisis personalizado sobre el AST generado por *Dart Analyzer* con los editores de texto más populares, como IntelliJ, Eclipse, Atom, entre otros. Con esta librería es posible mostrar errores, *warnings*, sugerencias de edición y resaltado de sintaxis.

Capítulo 3

Propuesta

En este trabajo se propone realizar la implementación del análisis de *Type-based Relaxed Noninterference* en Dart, y una herramienta de inferencia de facetas de declasificación, mediante la realización de un plugin para Dart Analyzer. En este capítulo se detallan los problemas de inferencia a resolver, las estrategias utilizadas para resolverlos, las extensiones al trabajo original y el subconjunto del lenguaje soportado.

3.1. DRAFT de ejemplos

Inferencia completa basado en uso de distintas variables.

```
int foo(String a, String b) {
  return a.length + b.indexOf("a");
}
->
@IntPlus int foo(@StringLength String a, @StringIndexOf String b) {
  return a.length + b.indexOf("a");
}
```

¿Qué pasa si se llama a un metodo que retorna High?

```
class int {
  @High int operator +(int b);
   ...
}

int foo(String a, String b) {
  return a.length + b.indexOf("a");
}
->
```

```
@High int foo(@StringLength String a, @StringIndexOf String b) {
  return a.length + b.indexOf("a");
}
```

Inferencia en condicionales.

```
int foo(bool a) {
 bool b = !a;
 int ret = 0;
 if (a) {
   ret = 4;
 }
 else {
   ret = 5;
 }
 return ret;
}

@High int foo(@BoolNot bool a) {
 @High bool b = !a;
 @High int ret = 0;
 if (a) {
   ret = 4;
 }
 else {
   ret = 5;
 }
 return ret;
}
```

Si la variable retornada no tiene ninguna llamada a método, retornará High. En este caso las asignaciones son válidas, pues se puede asignar cualquier cosa a High (si los tipos de implementación son válidos).

```
 @High int foo(@Bool bool a) {
  @Int int ret = 0;
  if (a) {
    ret = 4;
  }
  else {
    ret = 5;
  }
  return ret;
 }
```

Si forzamos la declaración de `ret` como pública, la asignación será inválida, puesto que el contexto de seguridad del `if` es `BoolNot`, lo cual no tiene una relación de subtyping válida con `int` para realizar la asignación.

Conclusión

Mauris ac ipsum. Duis ultrices erat ac felis. Donec dignissim luctus orci. Fusce pede odio, feugiat sit amet, aliquam eu, viverra eleifend, ipsum. Fusce arcu massa, posuere id, nonummy eu, pulvinar ut, wisi. Sed dui. Vestibulum nunc nisl, rutrum quis, pharetra eget, congue sed, dui. Donec justo neque, euismod eget, nonummy adipiscing, iaculis eu, leo. Duis lectus. Morbi pellentesque nonummy dui.

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.

Nunc euismod, mauris luctus adipiscing pellentesque, augue ligula pellentesque lectus, vitae posuere purus velit a pede. Phasellus leo mi, egestas imperdiet, blandit non, sollicitudin pharetra, enim. Nullam faucibus tellus non enim. Sed egestas nunc eu eros. Nunc euismod venenatis urna. Phasellus ullamcorper. Vivamus varius est ac lorem. In id pede eleifend nibh consectetuer faucibus. Phasellus accumsan euismod elit. Etiam vitae elit. Integer imperdiet nibh. Morbi imperdiet orci euismod mi.

Figura 3.1: Logo de la Facultad

Donec tincidunt tempor metus. Aenean egestas cursus nulla. Fusce ac metus at enim viverra lacinia. Vestibulum in magna non eros varius suscipit. Nullam cursus nibh. Mauris neque. In nunc quam, convallis vitae, posuere in, consequat sed, wisi. Phasellus bibendum consectetuer massa. Curabitur quis urna. Pellentesque a justo.

In sit amet dui eget lacus rutrum accumsan. Phasellus ac metus sed massa varius auctor. Curabitur velit elit, pellentesque eget, molestie nec, congue at, pede. Maecenas quis tellus non lorem vulputate ornare. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Etiam magna arcu, vulputate egestas, aliquet ut, facilisis ut, nisl.

Donec vulputate wisi ac dolor. Aliquam feugiat nibh id tellus. Morbi eget massa sit amet purus accumsan dictum. Aenean a lorem. Fusce semper porta sapien.

Campo 1	Campo 2
Valor 1	Valor2

Tabla 3.1: Tabla 1

Curabitur sit amet libero eget enim eleifend lacinia. Vivamus sagittis volutpat dui. Suspendisse potenti. Morbi a nibh eu augue fermentum posuere. Curabitur elit augue, porta quis, congue aliquam, rutrum non, massa. Integer mattis mollis ipsum. Sed tellus enim, mattis id, feugiat sed, eleifend in, elit. Phasellus non purus sed elit viverra rhoncus. Vestibulum id tellus vel sem imperdiet congue. Aenean in arcu. Nullam urna justo, imperdiet eget, volutpat vitae, semper eu, quam. Sed turpis dui, porttitor ut, egestas ac, condimentum non, wisi. Fusce iaculis turpis eget dui. Quisque pulvinar est pellentesque leo. Ut nulla elit, mattis vel, scelerisque vel, blandit ut, justo. Nulla feugiat risus in erat.

Bibliografía

[1] Raimil Cruz, Tamara Rezk, Bernard Serpette, and Éric Tanter. Type abstraction for relaxed noninterference. In Peter Müller, editor, *Proceedings of the 31st European Conference on Object-oriented Programming (ECOOP 2017)*, Barcelona, Spain, June 2017. Dagstuhl LIPIcs. To appear.

[2] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, January 1999.

[4] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.

[5] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[6] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[7] Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In Roberto Giacobazzi, editor, *Static Analysis*, pages 84–99, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[8] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.