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.

A mis padres, por su apoyo incondicional.

Tabla de Contenido

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 [5].

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.

Los lenguajes de seguridad formalizan la protección de confidencialidad mediante una propiedad de no-interferencia [3], la cual puede ser muy restrictiva para aplicaciones reales y prácticas. Es por ello que los lenguajes de seguridad ofrecen mecanismos para desclasificar la información, y a su vez asegurar el cumplimiento de la propiedad.

Uno de los mayores desafíos de los lenguajes de seguridad es ofrecer mecanismos de desclasificación utilizando técnicas más expresivas, y de esta forma facilitar el trabajo del programador. 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 desclasificación de la información.

El fundamento teórico de type-based declassification está bien descrito, pero carece de una implementación que permita comprobar la utilidad práctica de la propuesta. Además, se considera poco viable la implementación en su estado actual, ya que el programador tendría que agregar muchas anotaciones innecesarias al código para poder efectuar el análisis de control de flujo.

Un problema similar es el que resuelven los lenguajes de programación utilizando mecanismos de inferencia de tipos, con el fin de facilitar el trabajo al programador. En esta dirección, se han propuesto mecanismos de inferencia para tipos de seguridad [7], lo que motiva una

proposición similar para type-based declassification.

Dart es un lenguaje de programación multipropósito que ofrece herramientas para realizar análisis personalizado sobre el árbol sintáctico de un código fuente Dart. Estas herramientas pueden ser integradas a los entornos de desarrollo integrado (IDE) mediante plugins, lo que permite al usuario analizar sus programas de forma interactiva.

1.1. Objetivos

El objetivo de la memoria es realizar la implementación de un sistema de inferencia para type-based declassification. Dentro de los objetivos específicos del trabajo, podemos encontrar:

- **Inferencia y verificación estática de type-based declassification**. Se entiende coma la implementación de un sistema de inferencia de facetas de desclasificación para type-based declassification, en el lenguaje de programación Dart. Dentro de la inferencia se incluye la verificación de las reglas del sistema de tipos de type-based declassification.
- **Plugin para editores**. Mostrar al programador el resultado de la inferencia, por medio de un plugin para los IDE que soporten servidores de análisis estático de Dart, ofreciéndole acciones al respecto.

1.2. Organización del documento

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, y la validación del trabajo es discutida en el capitulo 5. En el último capítulo se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Antecedentes

2.1. Control de flujo de información

Los lenguajes con tipado de seguridad para el control del flujo de la información clasifican los valores de un programa con respecto a sus niveles de confidencialidad, expresado mediante una *lattice*[1] de etiquetas de seguridad. Por ejemplo, con la lattice de dos niveles de seguridad `L` $\sqsubseteq$ `H` se puede distinguir entre valores públicos o de baja confidencialidad (`L`) y valores privados o de alta confidencialidad (`H`). Un sistema de tipos con control de flujo asegura de forma estática el cumplimiento de la propiedad *noninterference* [3], esto es, que la información confidencial no fluya directa o indirectamente hacia canales públicos [9].

En el siguiente ejemplo se muestra un código anotado con niveles de seguridad, en donde el parámetro `guess` y el retorno del método se declaran de baja confidencialidad, y el parámetro `password` se declara de alta confidencialidad.

Ejemplo 2.1

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

Se dice que ocurrió un *flujo implícito* cuando un programa da conocimiento de una variable de baja confidencialidad, en un contexto de alta confidencialidad. En el ejemplo 2.1, se da conocimiento de un literal[2] en un contexto determinado por la operación de comparación del condicional, la cual retorna un valor de alta confidencialidad.

[1] Un orden parcial, donde todo par de elementos tiene un único supremo e ínfimo
[2] Un literal es considerado de baja confidencialidad

La ocurrencia de un flujo implícito significa una infracción a noninterference. Para su detección, se utiliza el concepto de *program counter* (`pc`) para seguridad [4], el cual permite considerar el contexto de ejecución de una instrucción en las reglas del sistema de tipos. En el ejemplo 2.1, las instrucciones de retorno se ejecutan con un `pc` igual al nivel de seguridad de retorno de la condición.

A pesar de que noninterference es una propiedad atractiva para la especificación de sistemas seguros, se considera muy estricta en la práctica, debido a que impide que la información confidencial tenga cualquier tipo de influencia en una salida observable de un programa. En efecto, queremos que el programa de `login` sea aceptado a pesar de infringir noninterference, pues de otra forma no tendríamos cómo realizar la autenticación.

Para solucionar este problema, los lenguajes de seguridad adicionan mecanismos para disminuir el nivel de seguridad de un valor confidencial, implementados de diferentes formas [8]. Una de ellas, por ejemplo en Jif [6] es usar un operador `declassify`, que *desclasifica* un valor de alta confidencialidad retornando un valor de baja confidencialidad. En el siguiente ejemplo, se utiliza para desclasificar el resultado de la operación de comparación:

Ejemplo 2.2

```
String@L login(String@L guess, String@H password) {
  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. Por ejemplo, `declassify(password)` puede dar conocimiento absoluto sobre el valor de la variable a un observador público.

Varios mecanismos se han explorado para controlar el uso de desclasificación, y poder asegurar además una propiedad de seguridad para el programa [8]. En esta dirección, Cruz et al. [1] recientemente propusieron *type-based declassification* como un mecanismo de desclasificación que conecta la abstracción de tipos con una forma controlada de desclasificación, en una manera intuitiva y expresiva, proveyendo garantías formales sobre la seguridad del programa.

En type-based declassification los tipos tienen dos facetas; la faceta privada, que refleja el tipo de implementación, y la faceta pública, que refleja las operaciones de desclasificación sobre los valores de dicho tipo. Por ejemplo, el tipo $\texttt{StringEq} \triangleq [\texttt{eq} : \texttt{String} \to \texttt{Bool}]$ autoriza la operación `eq` sobre un `String`. Entonces se puede usar el tipo de dos facetas $\texttt{String} < \texttt{StringEq}$, en donde `String` es un subtipo de `StringEq`, para controlar la operación de desclasificación de la igualdad sobre `password`, lo que se muestra en el ejemplo 2.3

Ejemplo 2.3

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

El cumplimiento de la relación de subtyping entre la faceta pública y la faceta privada es requisito para el análisis de type-based declassification. Los tipos que cumplan con esta relación se denominan *well-formed*. Esto permite considerar a las facetas públicas dentro de una lattice de subtyping, como se ejemplifica en la figura 2.1, donde `StringEq` $\triangleq$ [`eq` : `String` $\rightarrow$ `Bool`] y `StringEqLength` $\triangleq$ [`eq` : `String` $\rightarrow$ `Bool`, `length` : `Unit` $\rightarrow$ `Int`].

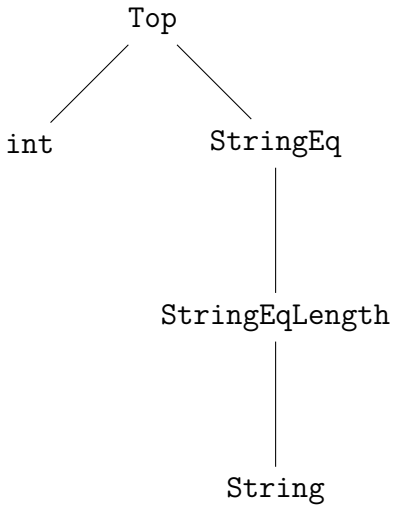

Figura 2.1: lattice de subtyping

Si la faceta pública coincide con la faceta privada, toda operación sobre el valor estará autorizada. Cuando esto sucede, se refiere usualmente a la faceta pública con `Bot`, por encontrarse siempre en la parte inferior de la lattice. Cuando se quiere referir a una faceta pública vacía o que no autoriza ninguna operación, se usa `Top`, por encontrarse en la parte superior de la lattice.

En estricto rigor, los métodos declarados en la faceta pública también poseen tipos de dos facetas en sus firmas. Así, el tipo `StringEq` visto anteriormente se define como `StringEq` $\triangleq$ [`eq` : `String` $<$ `String` $\rightarrow$ `Bool` $<$ `Bool`].

Existen dos reglas principales para comprobar que un programa con facetas públicas se encuentra bien tipado. En primer lugar, la llamada a un método sobre un valor cuya faceta pública autoriza la operación, retorna un valor del tipo declarado como retorno para aquella operación en la faceta pública. Por ejemplo, si tenemos un valor con faceta pública `StringHashEq` $\triangleq$ [`hash` : `Unit` $<$ `Unit` $\rightarrow$ `String` $<$ `StringEq`], y llamamos al método `hash` sobre este valor, el tipo de retorno de esa llamada será `String` $<$ `StringEq`. A esta regla se

le llama `TmD`.

La segunda regla expresa que la llamada a un método sobre un valor cuya faceta pública no autoriza la operación, retorna un valor con faceta pública `Top`. Esto ocurre, por ejemplo, si llamamos al método `hash` sobre un valor que declara la faceta pública `StringEq`. A esta regla se le llama `TmH`.

La propiedad de seguridad que se demuestra para el sistema de tipos de type-based declassification es una forma de noninterference con políticas de desclasificación, denominada *Relaxed noninterference*. Un lenguaje de seguridad que cumple esta propiedad, garantiza que la información confidencial sólo puede fluir hacia canales públicos de forma controlada, por medio de las políticas de desclasificación.

2.2. Inferencia de tipos

La inferencia de tipos es el proceso de determinar los tipos de las expresiones en un programa, basado en cómo son usadas. Tener un mecanismo de inferencia en un lenguaje de programación puede ser muy útil, debido a que da la posibilidad al programador de omitir las declaraciones de tipo para algunos identificadores.

Consideremos el siguiente ejemplo, donde se quita la anotación de la faceta pública del parámetro `password` del ejemplo 2.3:

Ejemplo 2.4

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

El sistema de tipos podría *inferir* que la faceta pública ausente contiene *al menos* el método `eq`. En efecto, `Bot` también es una faceta pública válida para `password`, pero no es la faceta pública que más se ajusta al uso del parámetro.

Para razonar acerca de la faceta pública del parámetro `password`, el sistema de tipos le asigna una *variable de tipo* α.

2.2.1. Constraints

Cuando un sistema de tipos aplica una determinada regla para tipar una expresión, puede imponer condiciones que los tipos deben cumplir para que la expresión esté bien tipada. En el ejemplo 2.4, se puede decir que `password` tiene una faceta pública α si y solo si α posee el

método `eq`.

Para razonar acerca de estas condiciones, el sistema de tipos las representa mediante *constraints*, que expresan una relación entre dos tipos. En el ejemplo 2.4, la condición sobre la faceta pública de `password` puede ser representada mediante la constraint de subtyping $\{\alpha <: [\texttt{eq} : \texttt{String} < \texttt{String} \rightarrow \texttt{Bool} < \texttt{Bool}]\}$.

El uso de constraints permite la presentación de un algoritmo de inferencia de forma modular, como un generador de constraints y un solucionador de constraints. El *set de constraints* generado se asemeja a un sistema de ecuaciones que siempre tendrá solución dependiendo de las características del lenguaje de programación. Por ejemplo, System F [10] es un lenguaje con inferencia de tipos completa no decidible.

2.2.2. Unificación

La unificación es el proceso de encontrar una solución a las variables de tipo del set de constraints. Si las constraints son de igualdad, la unificación realiza substituciones sucesivas hasta resolver cada uno de los tipos. En cambio, si las constraints son de subtyping, se deben realizar las operaciones `meet` (el ínfimo entre dos elementos, $a \wedge b$) y `join` (el supremo entre dos elementos, $a \vee b$) sobre la lattice que conforma la jerarquía de tipos, cuando sea pertinente.

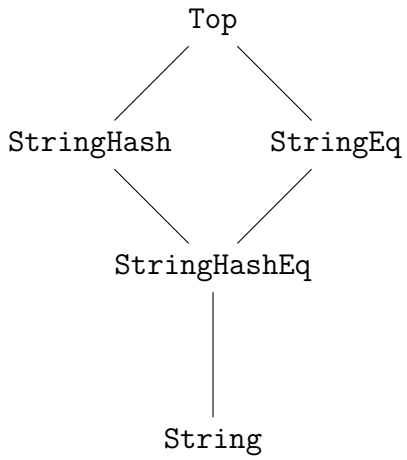

Figura 2.2: Operación `meet` entre `StringEq` y `StringHash`

Consideremos el siguiente ejemplo, en donde se modifica una expresión de retorno del ejemplo 2.4 con otro uso del parámetro `password`.

Ejemplo 2.5

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

Supongamos que `hash` retorna un `String`. Ahora se generan dos constraints sobre `password`, $\{\alpha <: [\texttt{eq} : \texttt{String} < \texttt{String} \to \texttt{Bool} < \texttt{Bool}]\}$ y $\{\alpha <: [\texttt{hash} : \texttt{Unit} < \texttt{Unit} \to \texttt{String} < \texttt{String}]\}$. El tipo α se resuelve con la operación `meet` entre ambos tipos de objeto, cuyo resultado $\texttt{StringHashEq} \triangleq [\texttt{eq} : \texttt{String} < \texttt{String} \to \texttt{Bool} < \texttt{Bool}, \texttt{hash} : \texttt{Unit} < \texttt{Unit} \to \texttt{String} < \texttt{String}]$ se muestra en la figura 2.2. El teorema 2.6 muestra las reglas utilizadas para la aplicación de operaciones sobre la lattice.

Teorema 2.6 Si `x`, `y` y `z` pertenecen a una lattice de subtyping, se cumple lo siguiente:

- $\texttt{x} <: \texttt{y}, \texttt{x} <: \texttt{z} \implies \texttt{x} <: \texttt{y} \wedge \texttt{z}$
- $\texttt{y} <: \texttt{x}, \texttt{z} <: \texttt{x} \implies \texttt{y} \vee \texttt{z} <: \texttt{x}$

Capítulo 3

Propuesta

En este trabajo se propone realizar la implementación en Dart de un sistema de inferencia de facetas públicas, que incluya el análisis de *Type-based declassification*, mediante la realización de un plugin para entornos de desarrollo integrado (IDE). En este capítulo se detalla el problema de inferencia a resolver y las estrategias utilizadas para resolverlo.

3.1. Problema de inferencia

Para la formulación del problema, es posible asumir que la información de las facetas privadas de type-based declassification se encuentra a disposición, debido a que algunos lenguajes de programación poseen herramientas para obtener dicha información.

Definición 3.1 (Problema de inferencia) *Dado un programa parcialmente tipado con facetas públicas, y completamente tipado con facetas privadas, encontrar la faceta pública de las expresiones no tipadas que más se ajuste al uso de las expresiones, tal que se cumplan las reglas del sistema de tipos de type-based declassification.*

A continuación, se muestran algunos ejemplos con casos que no fueron cubiertos en el capítulo 2, con el objetivo de ilustrar la solución esperada al problema de inferencia.

Ejemplo 3.2

```
bool login(String<Top password, String<String guess) {
  return password.eq(guess);
}
```

En este ejemplo, se quiere inferir que el método `login` tiene a `Top` como faceta pública, debido a la aplicación de la regla `TmH` de type-based declassification.

Ejemplo 3.3

```
bool login(String password, String guess) {
  return password.hash().eq(guess);
}
```

En este caso ocurrió un encadenamiento de llamadas a métodos sobre `password`. La faceta pública para `password` que resuelve el problema de inferencia contiene al método `hash`, al cual se le infiere una faceta pública de retorno que contiene al método `eq`. Si se declara una faceta pública de retorno para el método `eq`, entonces se infiere esa misma faceta para el retorno del método `login`, por la aplicación de la regla `TmD` de type-based declassification.

Ejemplo 3.4

```
void check(String<Bot s);

bool<Top login(String<Top password, String guess) {
  check(password);
  return password.eq(guess);
}
```

En este caso, se debe reportar un error de flujo en el llamado a la función `check`, debido a que la faceta del argumento debe ser subtipo de la faceta del parámetro, esto es, `Top <: Bot` es una relación no válida.

3.2. Consideraciones de diseño

Se debe tomar una decisión acerca de las facetas públicas de los métodos que pertenecen al *core* de un lenguaje de programación. Para ilustrar la necesidad de esta decisión, veamos el siguiente ejemplo:

```
int<Bot getLength(String password) {
  return password.length
}
```

Este método será aceptado o rechazado por las reglas del sistema de tipos, si la faceta pública de retorno del campo `length` es `Bot` o `Top` respectivamente.

Si decidimos que la faceta pública de retorno para métodos del `core` del lenguaje es `Top`, entonces cualquier operación que realicemos sobre el valor de retorno, retornará `Top`, lo cual

es poco útil. Por lo tanto la decisión por defecto es que la faceta pública de retorno para métodos del *core* del lenguaje sea `Bot`.

Ahora, analicemos ambas posibilidades para la faceta pública por defecto de los parámetros:

- `Top` $\rightarrow$ `Bot`: Supongamos que el *core* del lenguaje posee un método `identity`, que dado un `x`, retorna `x`. Si tomamos esta decisión, entonces el método `identity` podrá ser usado como desclasificador universal, como por ejemplo `identity(password)`.
- `Bot` $\rightarrow$ `Bot`: Esta elección restringe las facetas públicas de los argumentos utilizados a `Bot`, lo cual también podría ser considerado poco útil. Sin embargo, al retornar un valor con faceta pública `Bot`, cualquier operación podrá ser utilizada sobre ese valor.

Haciendo un balance, se considera que la opción `Bot` $\rightarrow$ `Bot` tiene el mejor equilibrio entre utilidad y seguridad, por lo que es la opción por defecto considerada. Sin embargo, es deseable que la herramienta se pueda configurar para elegir otra alternativa.

3.3. Gramática de tipos

Como se vio en la sección 2.2, es necesario introducir variables de tipo para presentar un algoritmo de inferencia basado en constraints. Además, se deben definir los otros tipos que serán utilizados internamente en el análisis. Esto se hace definiendo la gramática de tipos que usará el sistema de inferencia:

$\tau := \alpha \mid \texttt{Obj}(\overline{\texttt{l} : \tau}) \mid [\overline{\tau}] \rightarrow \tau \mid \texttt{Join}(\overline{\tau}) \mid \texttt{Meet}(\overline{\tau}) \mid \texttt{Bot} \mid \texttt{Top}$

Donde α es cualquier variable de tipo, $\texttt{Obj}(\overline{\texttt{l} : \tau})$ representa el tipo de un objeto y `l` es un nombre de método.

3.4. Generación de constraints de subtyping

Como se mencionó en la sección 2.2.1, el uso de constraints permite presentar un algoritmo de inferencia como una fase de generación de constraints, y una fase de resolución de constraints. En el algoritmo 1 se muestra la generación de constraints para un nodo determinado del árbol de sintaxis abstracta (AST).

Es importante notar que la constraint generada por la invocación a un método almacena el tipo de la expresión, debido a la posible aplicación de la regla `TmH` del sistema de tipos de type-based declassification.

Algoritmo 1
Generación de constraints

```
1: function CONSTRAINT_GENERATION(node, pc)
2:     cs ← {}
3:     switch node do
4:         case MethodInvocation(name, target, signature, expression, arguments)
5:             cs.insert(target <: Obj(name: signature), expression)
6:             for argument, correspondingParameter in arguments do
7:                 cs.insert(argument <: correspondingParameter)
8:             end for
9:         case ReturnStatement(expression, methodReturn)
10:            cs.insert(expression <: methodReturn)
11:            cs.insert(pc <: methodReturn)
12:        case AssignmentExpression(leftHand, rightHand)
13:            cs.insert(rightHand <: LeftHand)
14:            cs.insert(pc <: LeftHand)
15:        case IfExpression(conditionExpression)
16:            pc ← conditionExpression
17:    return cs, pc
18: end function
```

Ejecutando la función del algoritmo 1 en todos los nodos del AST y uniendo los resultados, se obtiene el set de constraints.

A modo de ejemplo, se muestran las constraints generadas para el ejemplo 2.5.

Ejemplo 3.5

1. $\{\alpha <: \texttt{Obj}(\texttt{eq} : [\texttt{Bot}] \to \texttt{Bot}),\ \beta\}$
2. $\{\alpha <: \texttt{Obj}(\texttt{hash} : [\,] \to \texttt{Bot}),\ \gamma\}$
3. $\{\texttt{Bot} <: \texttt{Bot}\}$
4. $\{\texttt{Bot} <: \texttt{Bot}\}$
5. $\{\gamma <: \texttt{Bot}\}$
6. $\{\texttt{Bot} <: \texttt{Bot}\}$

Las constraints 1 y 2 se generan por las llamadas a métodos sobre el parámetro `password` (α). Las constraints 3 y 4 se generan por la relación entre el `pc` (`Bot`) y el retorno del método (`Bot`), y las constraint 5 y 6 se generan por la relación entre la expresión de retorno (γ y `Bot`) y el retorno del método.

Además de la generación de constraints, en este paso se realiza la verificación de well-formed en las expresiones que tienen facetas públicas declaradas. Esto se muestra en el algoritmo 2, para un nodo determinado del AST.

Algoritmo 2
Verificación de well-formed

```
1: function isWellFormed(node)
2:     if node.hasDeclaredPublicFacet() then
3:         return node.privateFacet.subtypeOf(node.publicFacet)
4:     end if
5:     return true
6: end function
```

3.5. Resolución de constraints de subtyping

El primer paso en la resolución de constraints es la eliminación de constraints *obvias*. Esto es, la eliminación de las constraints `Bot <: X` y `X <: Top`, ya que no aportan información útil al algoritmo de inferencia.

Algoritmo 3
Simplificación de constraints

```
1: function Simplify(cs)
2:     for constraint in cs do
3:         if constraint.left is Bot then
4:             cs.remove(constraint)
5:         else if constraint.right is Top then
6:             cs.remove(constraint)
7:         end if
8:     end for
9: end function
```

Aplicando el algoritmo 3, el set de constraints del ejemplo 3.5 se reduce a solo tres constraints.

1. $\{\alpha <: \texttt{Obj}(\texttt{eq} : [\texttt{Bot}] \to \texttt{Bot}),\ \beta\}$
2. $\{\alpha <: \texttt{Obj}(\texttt{hash} : [\,] \to \texttt{Bot}),\ \gamma\}$
3. $\{\gamma <: \texttt{Bot}\}$

El siguiente paso es agrupar las constraints sobre la misma variable de tipo, usando las reglas del teorema 2.6. Aplicando el algoritmo 4, el set de constraints del ejemplo 3.5 se reduce a solo dos constraints.

Ejemplo 3.6

1. $\{\alpha <: \texttt{Meet}(\texttt{Obj}(\texttt{eq} : [\texttt{Bot}] \to \texttt{Bot}),\ \texttt{Obj}(\texttt{hash} : [\,] \to \texttt{Bot}))\}$
2. $\{\gamma <: \texttt{Bot}\}$

Algoritmo 4
Agrupación de constraints

```
1: function GROUP(cs, typeVariables)
2:     toRemove ← {}
3:     for tvar in typeVariables do
4:         c1 ← Constraint(tvar, Meet())
5:         c2 ← Constraint(Join(), tvar)
6:         for constraint in cs do
7:             if constraint.left == tvar then
8:                 c1.right.insert(constraint.right)
9:                 toRemove.insert(constraint)
10:            end if
11:            if constraint.right == tvar then
12:                c2.left.insert(constraint.left)
13:                toRemove.insert(constraint)
14:            end if
15:        end for
16:        if c1.right.notEmpty() then
17:            cs.insert(c1)
18:        end if
19:        if c2.left.notEmpty() then
20:            cs.insert(c2)
21:        end if
22:    end for
23:    cs.removeAll(toRemove)
24: end function
```

3.5.1. Unificación

En este paso, se materializan las operaciones `meet` y `join`, se comprueba la validez de las constraints y se realizan substituciones de forma iterativa.

Meet y Join

Cuando se tiene un tipo $\texttt{Meet}(\overline{\tau})$ o $\texttt{Join}(\overline{\tau})$, donde τ no tiene variables de tipo, entonces se puede materializar la operación sobre la lattice correspondiente. Este procedimiento se muestra en el algoritmo 5. La operación `meet` consiste en la unión de los miembros de cada tipo de objeto en el tipo `Meet`, y `join` consiste en la intersección de los miembros de cada tipo de objeto en el tipo `Join`.

Si aplicamos el algoritmo 5 al ejemplo 3.6, se materializa la operación de la constraint 1.

1. $\{\alpha <: \texttt{Obj}(\texttt{eq} : [\texttt{Bot}] \to \texttt{Bot}, \texttt{hash} : [\,] \to \texttt{Bot})\}$
2. $\{\gamma <: \texttt{Bot}\}$

Algoritmo 5
Materialización de operaciones

```
1: function performOperations(cs)
2:     for constraint in cs do
3:         if constraint.right is Meet and constraint.right.isConcrete() then
4:             constraint.right ←
5:                 constraint.right.reduce((t1, t2) => Obj(t1.members ∪ t2.members))
6:         end if
7:         if constraint.left is Join and constraint.left.isConcrete() then
8:             constraint.left ←
9:                 constraint.left.reduce((t1, t2) => Obj(t1.members ∩ t2.members))
10:        end if
11:    end for
12: end function
```

Verificación de constraints

Cuando una constraint representa una relación no válida, existen dos casos posibles:

1. Si la constraint no proviene de una invocación a método, se debe reportar un error.
2. En caso contrario, se debe reemplazar por `Top` toda aparición del tipo de expresión de la constraint, en el set de constraints. En este caso no se debe reportar error.

En el ejemplo 3.2, las constraints relevantes generadas son:

1. $\{\texttt{Top} <: \texttt{Obj}(\texttt{eq} : \texttt{Bot} \to \texttt{Bot}),\ \alpha\}$
2. $\{\alpha <: \beta\}$

Donde β es la faceta pública de retorno de `login`. Como la constraint 1 representa una relación no válida y proviene de invocación a método, el tipo de expresión α debe substituirse por `Top` en el set de constraint. Luego, la constraint 2 cambia a $\{\texttt{Top} <: \beta\}$.

El algoritmo 6 muestra la verificación de constraints. Este algoritmo usa la función `substitute`, que se muestra en el algoritmo 7.

Substitución

Cuando se tiene una constraint que relaciona una variable de tipo con un tipo concreto[1], se debe substituir en el set de constraint toda aparición de la variable de tipo, por el tipo concreto. Como este proceso puede generar nuevas constraints que sean candidatas a materialización de operaciones, a verificación o a substitución, se debe iterar hasta que no queden constrainst candidatas. El algoritmo 8 muestra la substitución de constraints resueltas, mientras que el algoritmo 9 muestra el procedimiento completo de unificación.

[1]Un tipo concreto no tiene variables de tipo

Algoritmo 6
Verificación de constraints

```
1: function checkConstraints(cs,errorCollector)
2:     for constraint in cs do
3:         if constraint.isNotValid() and constraint.isFromMethodInvocation() then
4:             for c in cs do
5:                 c.left ← substitute(c.left, constraint.expressionType, Top)
6:                 c.right ← substitute(c.right, constraint.expressionType, Top)
7:             end for
8:         else if constraint.isNotValid() then
9:             errorCollector.insert(SubtypingError)
10:        end if
11:    end for
12: end function
```

Algoritmo 7
Substitución dentro de tipos

```
1: function substitute(source,target,newType)
2:     if source == target then
3:         return newType
4:     end if
5:     switch source do
6:         case Obj(members)
7:             return Obj(members.map((l,s) =>l: substitute(s,target,newType)))
8:         case x → y
9:             return substitute(x, target, newType) → substitute(y, target, newType)
10:        case Meet(types)
11:            return Meet(Meet.map((t) =>substitute(t,target,newType))
12:        case Join(types)
13:            return Join(Join.map((t) =>substitute(t,target,newType)))
14:        case Bot
15:            return Bot
16:        case Top
17:            return Top
18:        case α
19:            return α
20: end function
```

Algoritmo 8
Substitución de constraints

```
1: function substituteResolved(cs)
2:     m ← {}
3:     for constraint in cs do
4:         if constraint.right.isConcrete() and constraint.left.isVariable() then
5:             for c in cs, c != constraint do
6:                 c.left ← substitute(c.left, constraint.left, constraint.right)
7:                 c.right ← substitute(c.right, constraint.left, constraint.right)
8:             end for
9:             m[constraint.left] ← constraint.right
10:            cs.remove(constraint)
11:        end if
12:        if constraint.left.isConcrete() and constraint.right.isVariable() then
13:            for c in cs, c != constraint do
14:                c.left ← substitute(c.left, constraint.right, constraint.left)
15:                c.right ← substitute(c.right, constraint.right, constraint.left)
16:            end for
17:            m[constraint.right] ← constraint.left
18:            cs.remove(constraint)
19:        end if
20:    end for
21:    return m
22: end function
```

Algoritmo 9
Unificación

```
1: function Unify(cs,errorCollector)
2:     m ← {}
3:     while cs.hasOperationCandidates() or cs.hasSubstCandidates() do
4:         performOperations(cs)
5:         checkConstraints(cs, errorCollector)
6:         m.addAll(substitueResolved(cs))
7:     end while
8:     return m
9: end function
```

La función de unificación retorna un mapeo entre cada variable de tipo y un tipo concreto. El caso de que queden variables de tipo sin resolver puede significar dos cosas:

- Falta información para determinar el tipo concreto de una expresión, lo que no significa errores de tipos
- Cualquier faceta sirve para validar la expresión según las reglas del sistema de tipos

Si se informa al usuario un error debido a la ocurrencia del primer caso, esto obliga la anotación de facetas que no son importantes, por lo que se considera al segundo caso una mejor opción.

Capítulo 4

Implementación

En esta sección se detalla la implementación de este trabajo, que se dividió en dos componentes principales. Primero, se implementó un sistema de inferencia para type-based declassification. Segundo, se elaboró un plugin para editores de texto que integra el resultado de la inferencia.

4.1. 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).

La implementación de este trabajo fue realizada en Dart, debido a que proporciona las herramientas necesarias para realizar el análisis requerido, como el AST (Abstract Syntax Tree) resuelto con la información completa de tipos. Además, 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.

4.1.1. Dart Analyzer

Dart Analyer es una herramienta incluida en Dart, que permite realizar análisis estático de código Dart. Entre otros servicios, esta herramienta permite obtener el AST de un código Dart. Dicho AST contiene la información relevante del programa, incluyendo el resultado del análisis de tipos.

Análisis personalizados de programas en Dart pueden ser realizados usando la información del AST. Para facilitar su integración, Dart Analyzer utiliza el patrón Visitor para incorporar un nuevo análisis sobre el AST.

4.1.2. Analyzer Plugin

La herramienta *Analyzer Plugin* sirve para integrar un análisis personalizado sobre el AST generado por Dart Analyzer, con los IDE que tengan soporte para servidores de análisis estático de Dart, como IntelliJ, Eclipse, Atom, entre otros.

Un plugin de Dart Analyzer se implementa en Dart, y utiliza una API para comunicarse con el servidor de análisis. La API consiste en tres tipos de comunicación:

1. Cuando el servidor de análisis necesita enviar información al plugin, o necesita pedir información al plugin, envía un *request*.
2. El plugin responde a toda request del servidor de análisis con un *response*.
3. El plugin puede enviar *notificaciones* al servidor de análisis con información.

Mediante el envío de notificaciones o respondiendo a peticiones del servidor de análisis, el plugin puede enviar información respecto a errores, resaltado de sintaxis, sugerencias de navegación, sugerencias de edición y marcado de ocurrencias, las cuales serán desplegadas en la interfaz del IDE.

4.2. Implementación de sistema de inferencia

4.2.1. Representación de facetas públicas

Para declarar las facetas públicas, se usan las anotaciones de Dart. Por ejemplo,

```
@S("Top") bool check(@S("StringCompareTo") String password);
```

es una declaración de un método de Dart anotado con facetas públicas.

La definición de las facetas públicas se hace mediante clases abstractas de Dart. Por ejemplo, la faceta `StringCompareTo` se define mediante la clase abstracta del mismo nombre:

```
abstract class StringCompareTo {
  int compareTo(String other);
}
```

Antes de la generación de constraints sobre un archivo, se realiza una etapa de *parsing* de facetas públicas, en donde se leen las clases abstractas del archivo. Esto se implementa mediante el *visitor* `DeclaredFacetVisitor`, que se muestra en el diagrama de la figura 4.1. Las facetas públicas procesadas se almacenan en el diccionario `declaredStore`, en donde se asocia el nombre de la faceta con su tipo de objeto correspondiente.

4.2.2. Tipos de errores

Durante el proceso de inferencia, se pueden generar varios tipos de errores, los cuales difieren en el mensaje que será desplegado en la interfaz de usuario, y el resaltado que aplicarán en la ubicación correspondiente del código fuente.

- `FlowError`: Se genera por la presencia de una constraint con una relación de subtyping no válida, que no proviene de invocación a método. Es un error, por lo que aplica un resaltado de color rojo en la ubicación correspondiente.
- `WellFormedTypeError`: Se genera por la detección de tipos que no son well-formed. Es un error, por lo que aplica un resaltado de color rojo en la ubicación correspondiente.
- `UndefinedFacetWarning`: Se genera por la declaración de una faceta pública que no ha sido definida. Es un warning, por lo que aplica un resaltado de color amarillo en la ubicación correspondiente.
- `UnableToResolveInfo`: Se genera por la incapacidad de inferir un tipo concreto para una variable de tipo. Es de caracter informativo, por lo que solo aplica un leve resaltado de sintaxis en el código, y muestra un mensaje cuando el cursor se posiciona sobre la ubicación correspondiente.
- `InferredFacetInfo`: Se genera en toda expresión que no posee una faceta pública declarada, con la información de la faceta inferida. Al igual que el tipo de error anterior, es de carácter informativo.

4.2.3. Fase de generación de constraints

Una vez que se procesan las facetas públicas, se procede a la generación de constraints. Esto se realiza implementando varios visitors mostrados en el diagrama de la figura 4.1.

La clase encargada de procesar un archivo es `CompilationUnitVisitor`, en donde se procesa cada clase declarada en el archivo. Mediante el visitor `ClassMemberVisitor`, se procesa cada método, campo y constructor de cada clase. Finalmente, el visitor implementado para procesar el cuerpo de cada miembro es `BlockVisitor`, en donde se procesa cada expresión relevante para el algoritmo de generación de constraints de la sección 3.4.

La clase `Store` es la encargada de la generación de variables de tipo, y el almacenamiento en diccionarios del tipo de las expresiones. Cada visita a los nodos del AST puede agregar constraints al set de constraints, y agregar o actualizar elementos en el store. Ambos se muestran en el diagrama 4.1.

En esta fase se pueden generar errores de tipo `WellFormedTypeError` y `UndefinedFacetWarning`, los cuales son recolectados mediante un `ErrorCollector`, el cual será utilizado para el despliegue de la información mediante el plugin.

4.2.4. Fase de resolución de constraints

En esta fase, la clase `ConstraintSolver`, que se muestra en el diagrama 4.1, se encarga de convertir el set de constraints en un mapeo entre variables de tipos y tipos concretos, implementando las operaciones descritas en la sección 3.5.

Los errores que son generados pueden ser de los tipos `SubtypingError`, `UnableToResolveInfo` y `InferredFacetInfo`, los cuales son recolectados mediante el mismo `ErrorCollector` de la fase de generación de constraints.

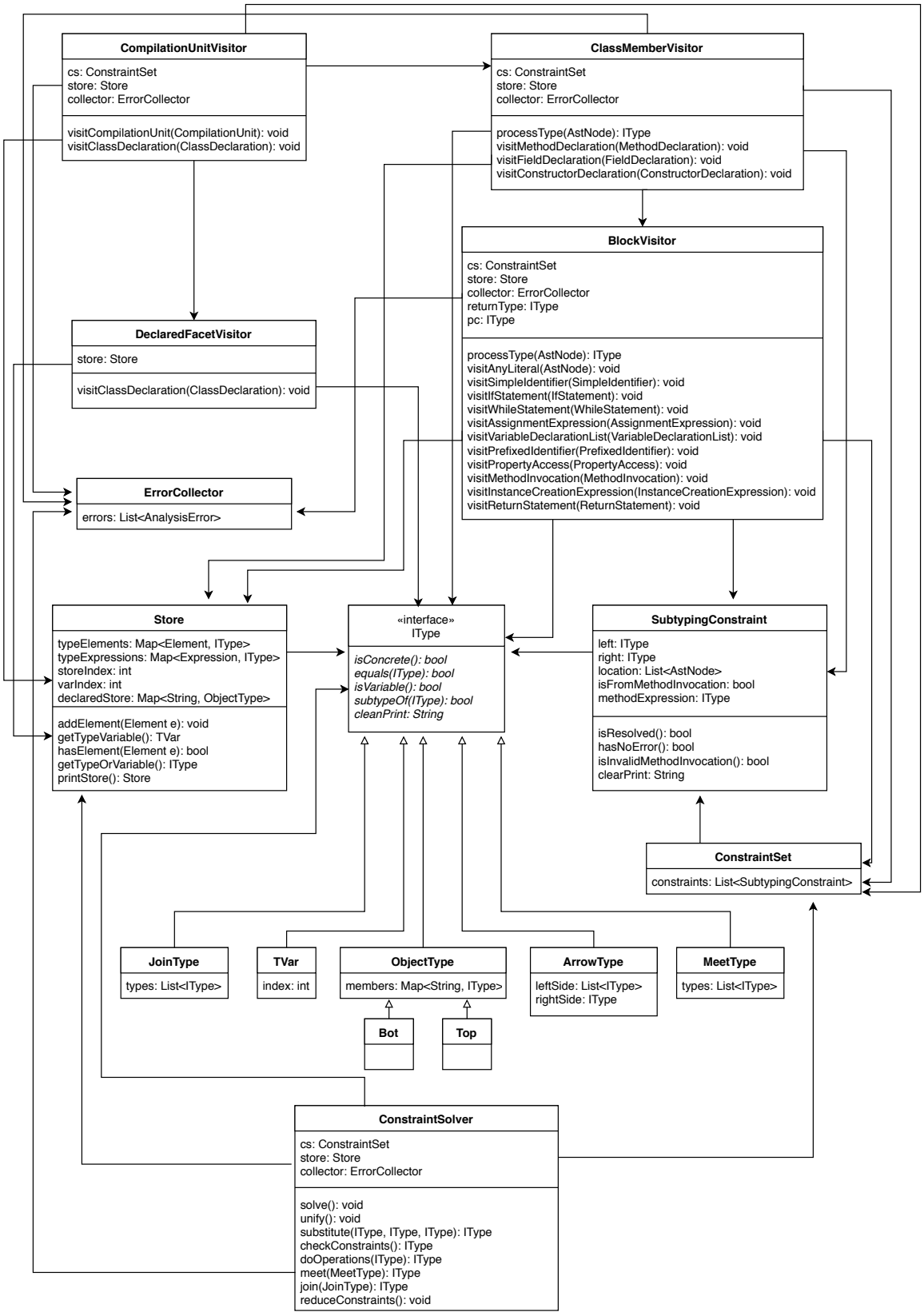

Figura 4.1: Diagrama de clases de sistema de inferencia

4.3. Implementación de plugin

Para la implementación del plugin, se siguió el tutorial oficial de la herramienta Analyzer Plugin, presente en el repositorio de GitHub oficial del lenguaje Dart [2].

La comunicación entre el plugin y el sistema de inferencia se realiza mediante la implementación de un *driver*, que administra los archivos que han sido modificados y solicita los resultados del análisis de inferencia a la clase `Analyzer`, que consiste en una lista de errores. Luego de obtener el resultado, el driver notifica al servidor de análisis para que pueda desplegar los errores en el IDE. La figura 4.2 muestra el diagrama de la secuencia de operaciones.

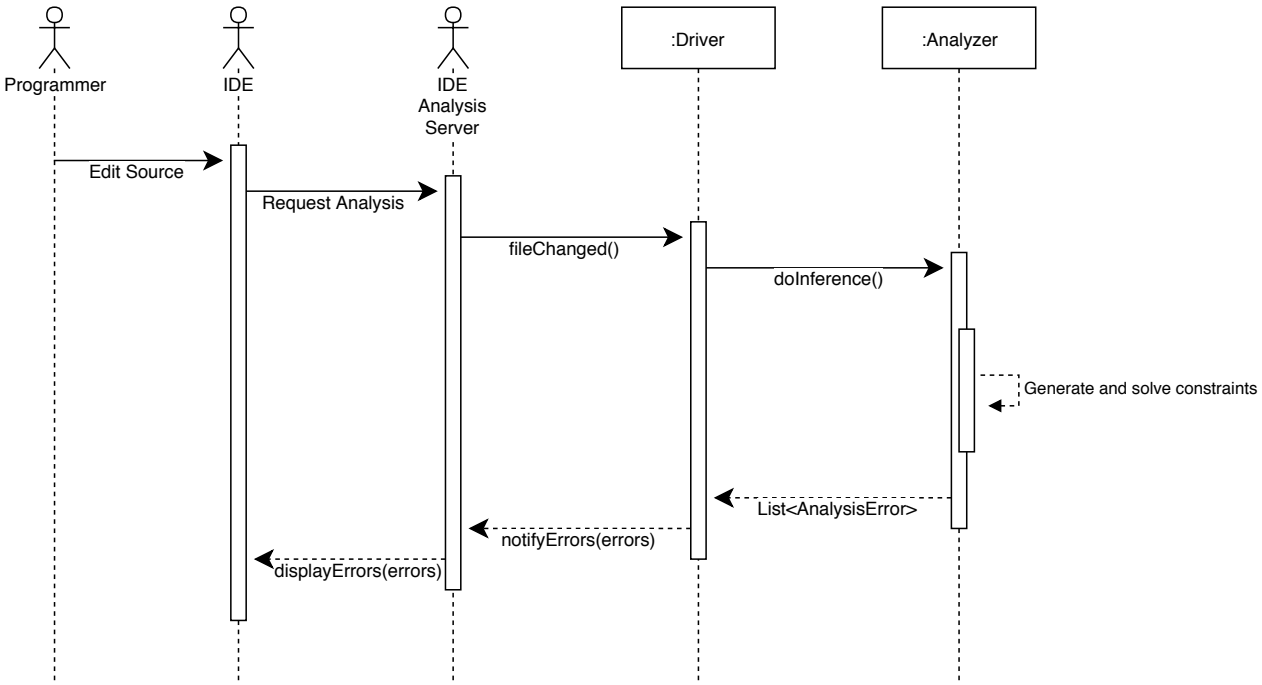

Figura 4.2: Diagrama de secuencia del plugin

4.3.1. Configuración del plugin

Para activar el análisis sobre un proyecto, se debe agregar el paquete del plugin como dependencia al proyecto, y agregar el plugin al archivo de configuración del análisis del proyecto `analysis_options.yaml`, ubicado en la raiz del proyecto.

```
analyzer:
  plugins:
    TRNIdart:
      default_core_return: Bot
      default_core_parameter: Bot
```

Las opciones `default_core_return` y `default_core_parameter` corresponden a las facetas públicas por defecto que tendrán los métodos del *core* de Dart.

Capítulo 5

Validación y Discusión

5.1. Batería de tests

Se ponen a prueba las reglas del sistema de tipos y la inferencia.

5.2. Repositorio de prueba

Pequeña aplicación segura que usa faceted types y el sistema de inferencia.

5.3. Usabilidad

Comportamiento, performance del plugin.

Conclusión

(Algo de conclusión)

Proyecciones y trabajo futuro

Formalización de inferencia

Extensión del subconjunto soportado

Sugerencias de edición, navegación y completación de código

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] Dart. Analyzer plugin: A framework for building plugins for the analysis server. `https://github.com/dart-lang/sdk/tree/master/pkg/analyzer_plugin`.

[3] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.

[4] 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 Dong Ho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005*, pages 156–168, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

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

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

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

[9] 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.

[10] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111 – 156, 1999.