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.

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 [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 que el análisis estático de *type-based declassification* no es suficiente por sí solo, ya que el programador tendría que anotar completamente el código fuente con facetas de desclasificación.

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:

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 (que por convención es considerado de baja confidencialidad) en un contexto determinado por la operación de comparación en la condición del `if`, la cual retorna un valor de alta confidencialidad.

Para detectar un flujo implícito, se utiliza el concepto de *program counter* (PC) para seguridad [4], el cual permite considerar el contexto de ejecución de una instrucción en las

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

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 previene que la información confidencial tenga cualquier tipo de influencia en una salida observable de un programa. En efecto, el programa de `login` del ejemplo 2.1 rompe con la propiedad, debido a que un observador público puede aprender acerca de la variable confidencial `password` observando la salida del programa para distintas ejecuciones.

Sin embargo, deseamos que el programa anterior sea aceptado, 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`, como se indica en el siguiente ejemplo, *desclasificando* la comparación de igualdad del parámetro confidencial `password` con el parámetro público `guess`

```
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)` da conocimiento absoluto sobre el valor de la variable.

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 $\texttt{StringEq} \triangleq [\texttt{eq} : \texttt{String} \rightarrow \texttt{Bool}]$ 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`.

```
String<String login(String<StringEq password, String<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 desclasificación es parte de la jerarquía de tipos, la que forma una lattice como se muestra en la figura 2.1. Si la faceta pública coincide con la faceta privada, entonces 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.

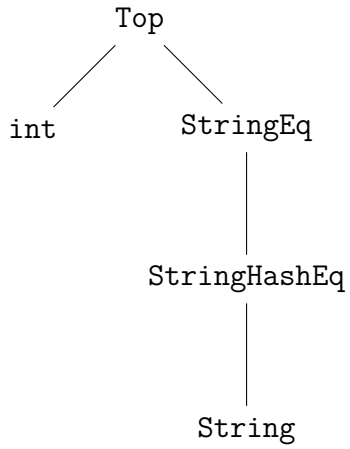

Figura 2.1: lattice de tipos de dos facetas

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 $\texttt{StringEq} \triangleq [\texttt{eq} : \texttt{String} < \texttt{String} \rightarrow \texttt{Bool} < \texttt{Bool}]$.

Existen dos reglas principales para comprobar que un programa con facetas de desclasificación se encuentra bien tipado. En primer lugar, la llamada a un método sobre un valor cuya faceta pública autoriza la operación, retorna la faceta pública que haya sido declarada como retorno para aquella operación. Por ejemplo, si tenemos un valor con faceta pública $\texttt{StringHashEq} \triangleq [\texttt{hash} : \texttt{String} < \texttt{String} \rightarrow \texttt{String} < \texttt{StringEq}]$, y llamamos al método `hash` sobre este valor, la faceta de retorno de esa llamada será $\texttt{String} < \texttt{StringEq}$. 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 `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

2.2.1. Objetivo y usos

La inferencia de tipos es el proceso de determinar los tipos para las expresiones de 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:

```
int foo(int a, int b) {
  int c = a + b;
  return c;
}
```

Supongamos que la suma solo está definida para números. Aquí, se considera redundante declarar el tipo de la variable `c`, debido a que la suma de dos números enteros siempre da como resultado un número entero. De la misma forma, podría ser considerado redundante declarar los tipos de los parámetros `a` y `b`, debido a que son utilizados como sumando en la asignación a la variable `c`, que fue declarada como `int`.

2.2.2. Variables de tipo

Los lenguajes de programación que tienen un mecanismo de inferencia de tipos, pueden expresar sus tipos mediante *type schemes*, los cuales incluyen variables de tipo y tipos concretos. Una variable de tipo es un tipo que no ha sido determinado.

Consideremos el mismo ejemplo, pero ahora omitiendo la declaración del tipo de la variable `c`.

```
int foo(int a, int b) {
  var c = a + b;
  return c;
}
```

Para tipar la variable `c`, el sistema de tipos le asignará una variable de tipo α, y generará *side conditions* al aplicar las reglas del sistema de tipos. En este caso, la *side condition* dice que el tipo de `c`, es decir, α, tiene que ser igual al tipo del lado derecho de la asignación, es decir, `int`.

2.2.3. Constraints

En el ejemplo anterior, se expresó la *side condition* con lenguaje natural. Como el objetivo es automatizar el proceso de inferencia, se utilizan constraints para expresar las *side conditions* generadas por la aplicación de una regla del sistema de tipos.

Las constraints sirven para expresar una relación entre dos tipos. Esta relación puede ser de igualdad o de subtyping. En el ejemplo anterior, la *side condition* puede ser representada mediante la constraint $\{\alpha = int\}$.

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. En el siguiente ejemplo, se supone que `num` es heredado por otros tipos numéricos.

```
foo(int a, float b, bool cond) {
  var c;
  if (cond) c = a;
  else c = b;
  return c;
}
```

El sistema de tipos asignará los siguientes tipos al programa:

```
α foo(int a, float b, bool cond) {
  β c;
  if (cond) c = a;
  else c = b;
  return c;
}
```

Y generará el set de constraints de subtyping $C : \{\beta <: \alpha, int <: \beta, float <: \beta\}$, de donde se deben resolver los tipos de las dos variables de tipo generadas. Notemos que β debe ser supertipo de `int` y `float`. El mejor tipo que cumple con ambas constraints es `num`.

2.2.4. Unificación

La unificación es el proceso de encontrar una substitución que hace iguales a dos términos. Aplicado a un set de constraints, se genera un mapeo de variables de tipo a tipos concretos, que satisfacen cada una de las constraints.

Si las constraints son de igualdad, la unificación consiste en realizar 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.

Por ejemplo, en el set de constraints generado en el ejemplo anterior, para resolver el tipo de β se debe realizar la operación `join` entre `int` y `float`, lo que da como resultado `num`. Luego, el set de constraints se reduce a $C : \{\beta <: \alpha, num <: \beta\}$, donde ahora se puede substituir β por `num`, lo que genera finalmente el mapeo $S : \{\beta \to num, \alpha \to num\}$.

2.2.5. Inferencia de tipos de seguridad

Como se vio en la sección **??**, los tipos de seguridad conforman una lattice con relaciones de subtyping, al igual que la jerarquía de tipos. En consecuencia, es posible formular un algoritmo de inferencia basado en constraints para tipos de seguridad.

En esta dirección, Pottier *et al.* [7] estudiaron un análisis de control de flujo basado en tipos para un lenguaje con referencias, excepciones y polimorfismo, con un sistema de tipos basado en constraints e inferencia de tipos decidible.

Capítulo 3

Propuesta

En este trabajo se propone realizar la implementación en Dart de un sistema de inferencia de facetas de desclasificación, 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 de desclasificación, y completamente tipado con facetas privadas, encontrar la faceta de desclasificación de las expresiones no tipadas, tal que se cumplan las reglas del sistema de tipos de type-based declassification.*

A continuación, se muestran algunos ejemplos de código parcialmente anotado con facetas de desclasificación, con el objetivo de ilustrar la solución esperada al problema de inferencia.

Ejemplo 3.2

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

En este caso, se quiere inferir que `password` tiene una faceta de desclasificación que contiene al método `eq`, y que tanto el retorno de `login` como el parámetro `guess` dependen de

la decisión que se tome sobre las facetas públicas por defecto que tendrán los métodos del *core* del lenguaje.

Ejemplo 3.3

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

Acá, se quiere inferir que el método `login` tiene a `Top` como faceta de desclasificación, debido a la aplicación de la regla `TmH` de *type-based declassification*.

Ejemplo 3.4

```
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 de desclasificación para `password` que resuelve el problema de inferencia contiene al método `hash`, al cual se le infiere una faceta de desclasificación de retorno que contiene al método `eq`. Si se declara una faceta de desclasificación de retorno para el método `eq`, entonces se infere esa misma faceta para el retorno del método `login`, por la aplicación de la regla `TmD` de *type-based declassification*.

Ejemplo 3.5

```
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

En el ejemplo 3.2, se mencionó sobre la decisión acerca de las facetas de desclasificación 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 de desclasificación de retorno del campo `length` es `Bot` o `Top` respectivamente.

Si decidimos que la faceta de desclasificación 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 de desclasificación de retorno para métodos del *core* del lenguaje sea `Bot`.

Ahora, analicemos ambas posibilidades para la faceta de desclasificación 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 de desclasificación de los argumentos utilizados a `Bot`, lo cual también podría ser considerado poco útil. Sin embargo, al retornar un valor con faceta de desclasificación `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 vió en la sección 2.2.2, 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.

Definición 3.6 (Gramática de tipos) $\tau := \alpha \mid \text{Obj}(\overline{l : \tau}) \mid \tau \rightarrow \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \text{Bot} \mid \text{Top}$

Donde α es cualquier variable de tipo, $\text{Obj}(\overline{l : \tau})$ representa el tipo de un objeto, l es un nombre de método, $\vee$ representa la operación `join` en la lattice, y $\wedge$ representa la operación `meet`.

3.4. Generación de constraints de subtyping

Como se mencionó en la sección 2.2.3, el uso de constraints permite presentar un algoritmo de inferencia como una fase de generación de constraints, y una fase de resolución de constraints. A continuación se muestran ejemplos de la generación de constraints para distintas expresiones, y luego un algoritmo en pseudo-lenguaje. De ahora en adelante, se usará el término *faceta* para referirse a las facetas de desclasificación.

Ejemplo 3.7

```
bool<α check(String<Top password, String<β guess) {
  return password == guess; /* γ */
}
```

1. $\gamma <: \alpha$
2. $\beta <:$ Bot
3. Top $<:$ Obj(==: Bot $\rightarrow$ Bot), γ

En este ejemplo, la constraint 1 se genera por la regla que indica que la faceta de la expresión de retorno del método (γ) debe ser subtipo de la faceta de retorno del método (α). La constraint 2 se genera por la regla que indica que la faceta del argumento en una llamada a método (β), debe ser subtipo de la faceta del parámetro correspondiente de ese método (Bot, ya que el método `==` es parte del *core* del lenguaje).

La constraint 3 se genera por la llamada al método `==`, indicando que la faceta del objetivo de la llamada (`Top`) debe ser subtipo de un objeto que contenga al método. Las constraints que se generan por llamadas a métodos, poseen además la faceta que corresponde a la expresión de esa llamada (γ), debido a la posible aplicación de la regla `TmH` en caso de que la relación de subtyping no sea válida. En este caso, la relación no es válida pues `Top` no es subtipo de Obj(==: Bot $\rightarrow$ Bot).

Ejemplo 3.8

```
class Person {
  bool<Top permission => true;
}

class Foo {
  String<α foo(Person<β p) {
    String<Bot ret = "denied";
    if (p.permission /* γ */) ret = "success";
    return ret;
  }
```

```
}
```

1. Bot <: Top
2. Bot <: Bot
3. β <: Obj(permission: Top), γ
4. Bot <: Bot
5. Top <: Bot
6. Bot <: α

En este ejemplo, la constraint 1 se genera por la relación entre la expresión de retorno y la faceta de retorno del método `permission`. Notar que el literal `true` tiene una faceta por defecto `Bot`. La constraint 2 se genera por la relación entre los lados izquierdo y derecho de la asignación a la variable `ret`. La constraint 3 se genera por la llamada al método `permission` sobre el parámetro `p`. Las constraints 4 y 5 se generan por la asignación a la variable `ret` en el cuerpo del condicional. La primera, expresa la relación entre el lado derecho de la asignación y el lado izquierdo, y la segunda expresa la relación entre el PC y el lado izquierdo de la asignación. La constraint 6 se genera por la relación entre la expresión de retorno del método y el retorno del método.

Ejemplo 3.9

```
int<α calc(num<β n) {
  int<Top ret = 1 + n.ceil() /* γ */;
  ret = ret + n.floor() /* δ */;
  return ret;
}
```

1. Bot <: Top
2. β <: Obj(ceil : [ ] $\rightarrow$ Bot), γ
3. Top <: Top
4. β <: Obj(floor : [ ] $\rightarrow$ Bot), δ
5. Top <: α

En este ejemplo, las constraints 1 y 3 se generan por la asignación a la variable `ret`. En cambio, las constraints 2 y 4 se generan por la llamada a los métodos `ceil` y `floor` del parámetro `n`. Este ejemplo es útil para mostrar el uso de la operación `meet` en el paso de resolución de constraints, debido a que la faceta de la variable β se debe resolver considerando la intersección entre dos facetas.

En la figura 3.1 se muestra el algoritmo para la generación de constraints de un nodo determinado del árbol de sintaxis abstracta.

```
function CONSTRAINT_GENERATOR(node, pc)
    cs ← {}
    if node is MethodInvocation then
        cs.insert(methodTarget <: Obj(methodName: methodSignature), callExpression)
        cs.insert(callExpression <: methodReturn)
        for argument← callArguments do
            cs.insert(argument <: correspondingParameter)
        end for
    else if node is ReturnStatement then
        cs.insert(returnExpression <: methodReturn)
        cs.insert(pc <: methodReturn)
    else if node is AssignmentExpression then
        cs.insert(rightHand <: LeftHand)
        cs.insert(pc <: LeftHand)
    else if node is IfExpression then
        pc ← conditionExpression
    end if
    return cs,pc
end function
```

Figura 3.1: Algoritmo de generación de constraints

3.5. Resolución de constraints

El siguiente paso del algoritmo de inferencia es la resolución del set de constraints obtenido en la fase de generación de constraints.

3.5.1. Simplificación y eliminación de constraints

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`. Así, en el ejemplo 3.8, las constraints resultantes son:

1. $\beta <:$ Obj(permission: Top), γ
2. Top <: Bot

Y para el ejemplo 3.9:

1. $\beta <:$ Obj(ceil : [ ] $\rightarrow$ Bot), γ
2. $\beta <:$ Obj(floor : [ ] $\rightarrow$ Bot), δ
3. Top <: α

3.5.2. Agrupación de constraints

El siguiente paso es agrupar las constraints sobre la misma variable de tipo, usando las siguientes reglas:

- `x <: y`, `x <: z` $\rightarrow$ `x <: y` $\wedge$ `z`
- `y <: x`, `z <: x` $\rightarrow$ `y` $\vee$ `z <: x`

Luego, el ejemplo 3.9 simplificado se reduce a dos constraints:

1. $\beta <:$ Obj(ceil : [ ] $\rightarrow$ Bot) $\wedge$ Obj(floor : [ ] $\rightarrow$ Bot)
2. Top $<: \alpha$

3.5.3. 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 $x \wedge y$ o $x \vee y$, donde x e y no tienen variables de tipo, entonces se puede materializar la operación sobre la lattice correspondiente.

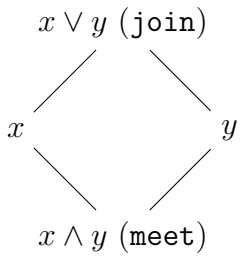

Figura 3.2: Operaciones sobre lattice

En el ejemplo 3.9 reducido, se materializa la operación `meet` del lado derecho de la constraint 1, por lo que las constraints quedan como sigue:

1. $\beta <:$ Obj(ceil : [ ] $\rightarrow$ Bot, floor : [ ] $\rightarrow$ Bot)
2. Top $<: \alpha$

La figura 3.3 muestra el algoritmo de materialización de operaciones sobre un set de constraints. Notar que se asume que MeetType aparece a la izquierda de las constraints, y JoinType a la derecha, debido a las reglas de agrupación.

```
function performOperations(cs)
    for c in cs do
        if c.right is MeetType and c.right.isConcrete() then
            c.right ← meet(c.right)
        end if
        if c.left is JoinType and c.left.isConcrete() then
            c.left ← join(c.left)
        end if
    end for
end function
```

Figura 3.3: Algoritmo de materialización de operaciones

Verificación de constraints

Cuando una constraint representa una relación no válida, existen dos posibilidades:

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

En el ejemplo 3.7, la constraint 3 proviene de invocación a método y representa una relación de subtyping no válida. Luego, se debe reemplazar la variable de tipo γ por `Top`:

1. Top $<:$ α
2. $\beta <:$ Bot
3. Top $<:$ Obj($==:$ Bot $\rightarrow$ Bot), γ

En cambio, en el ejemplo 3.8 simplificado, la constraint `Top <: Bot` representa una relación no válida que no proviene de invocación a método, por lo que un error debe ser reportado.

La figura 3.4 muestra el algoritmo de verificación de constraints. El método `substitueXForY(x,y)` busca `x` en todo el set de constraints, y lo substituye por `y`.

```
function checkConstraints(cs)
    for c in cs do
        if c.isNotValid() and c.isFromMethodInvocation then
            cs.substituteXForY(c.expressionType, Top)
        end if
        if c.isNotValid() and !c.isFromMethodInvocation then
            add SubtypingError
        end if
    end for
end function
```

Figura 3.4: Algoritmo de verificación de constraints

Substitución

Cuando se tiene una constraint que relaciona una variable de tipo y un tipo concreto (sin variables de tipo), 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 comprobación o a substitución, se debe iterar hasta que no queden constrainst candidatas.

La figura 3.5 muestra el algoritmo de substitución, mientras que la figura 3.6 muestra el algoritmo de unificación.

```
function performSubstitutions(cs)
    for c in cs do
        if c.right.isConcrete() and c.left.isVariable() then
            cs.substituteXForY(c.left, c.right)
        end if
        if c.left.isConcrete() and c.right.isVariable() then
            cs.substituteXForY(c.right, c.left)
        end if
    end for
end function
```

Figura 3.5: Algoritmo de substitución

```
function Unify(cs)
    while cs.hasOperationCandidates or cs.hasSubstCandidates do
        performOperations(cs)
        checkConstraints(cs)
        performSubstitutions(cs)
    end while
end function
```

Figura 3.6: Algoritmo de unificación

Al terminar el algoritmo, cada variable de tipo debe estar asociada a 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
- 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 un AST (Abstract Syntax Tree) dado 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. En efecto, 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. Con esta librería es posible mostrar errores, *warnings*, sugerencias de edición, sugerencias de navegación y resaltado de sintaxis.

4.2. Implementación de sistema de inferencia

4.2.1. Representación de facetas de desclasificación

Para declarar las facetas de desclasificación, se usarán 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 de desclasificación.

La definición de las facetas de desclasificación 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 de desclasificación, 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 de desclasificación 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.

- `SubtypingError`: 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.
- `UndefinedFacetWarning`: Se genera por la declaración de una faceta de desclasificación 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 de desclasificación declarada, con la información de la faceta inferida. Al igual que el error anterior, es de caracter informativo.

4.2.3. Fase de generación de constraints

Una vez que se procesan las facetas de desclasificación, 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 4.2.

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

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.

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.

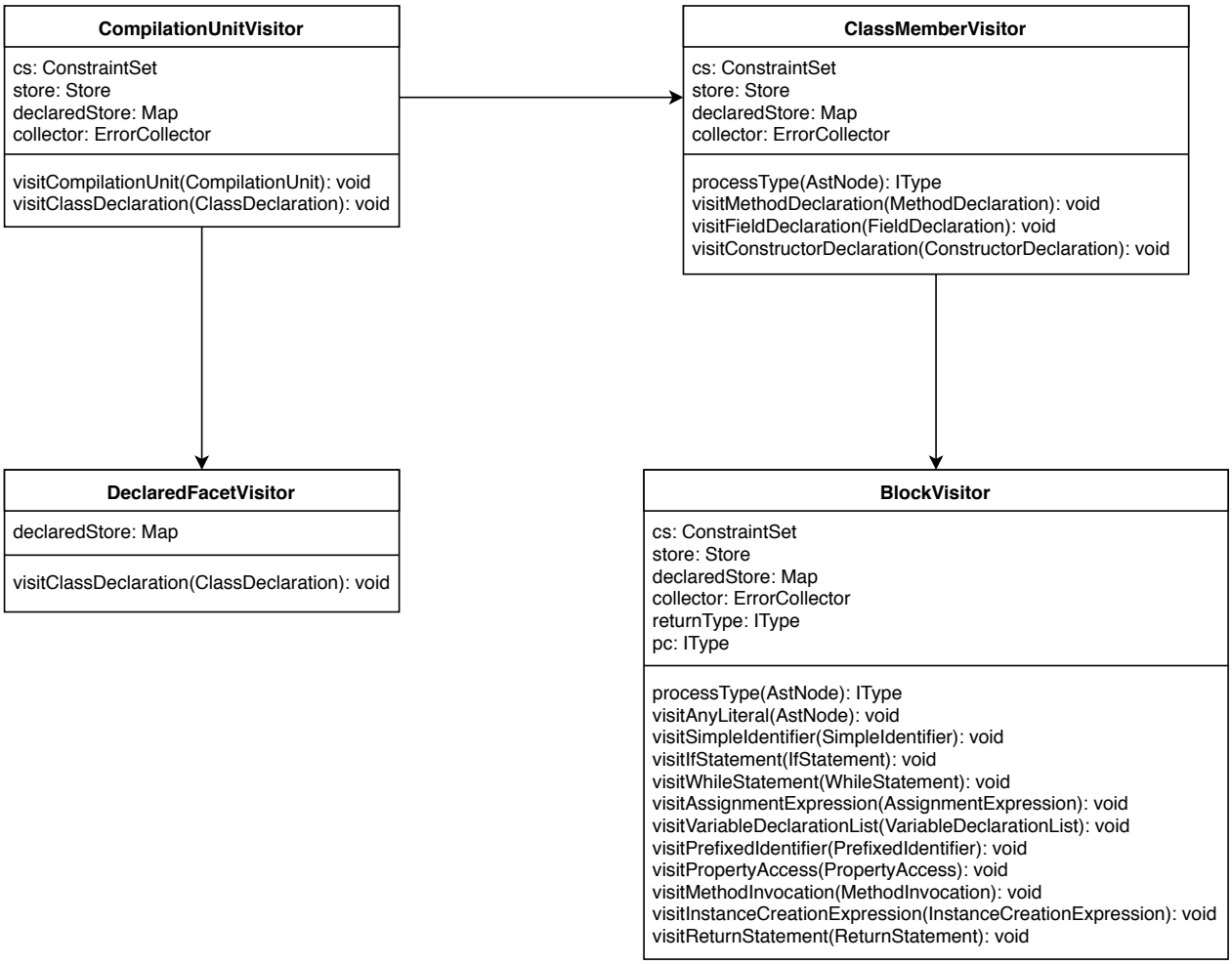

Figura 4.1: Diagrama de las clases encargadas de la generación de constraints

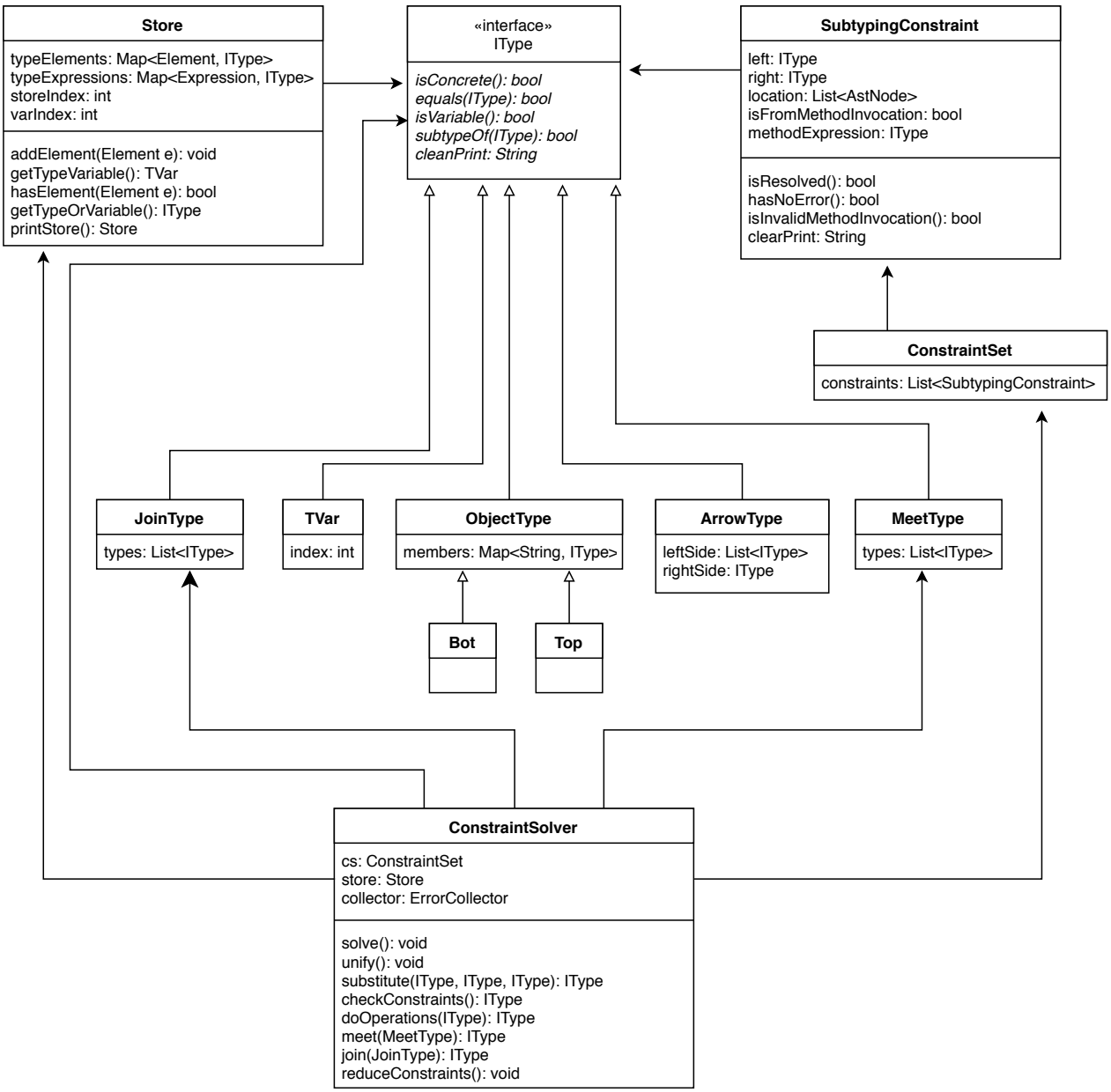

Figura 4.2: Diagrama de las clases relevantes del análisis interno y resolución de constraints

4.3. Implementación de plugin

4.3.1. Descripción general

El plugin implementa una API para establecer comunicación con el servidor de análisis de un IDE, respondiendo con el análisis de inferencia ante las peticiones recibidas desde el servidor.

Cuando el servidor de análisis detecta un cambio en un archivo, envía un mensaje al plugin, el cual gatilla la inferencia para todos los archivos del proyecto activo. Además, el plugin se encarga de recolectar los errores generados en las fases del proceso de inferencia, para enviárselos al servidor de análisis, el cual los despliega ante el usuario.

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

4.3.2. 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 de desclasificación 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.