



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:
AIDAN HOGAN
NANCY HITSCHFELD KAHLER

SANTIAGO DE CHILE
JULIO 2018

Resumen

La protección de la confidencialidad de la información manipulada por los programas computacionales es abordada a nivel del código fuente con distintas técnicas. Una de ellas es tipado de seguridad para el control de flujo, que controla el nivel de seguridad donde fluye la información agregando anotaciones a las variables tipadas.

La propiedad de seguridad fundamental de control de flujo es conocida como no-interferencia (*noninterference*), que establece que un observador público no puede obtener conocimiento desde datos confidenciales. A pesar de ser una propiedad muy atractiva, los sistemas reales la vulneran fácilmente, y necesitan mecanismos para desclasificar selectivamente alguna información.

En esta dirección, Cruz *et al.* proponen una forma de desclasificación basada en tipos (*type-based declassification*), en donde se utilizan las relaciones de subtipos del lenguaje para expresar las políticas de desclasificación de los datos que maneja el programa, en una forma simple y expresiva.

A pesar de que el fundamento teórico de la desclasificación basada en tipos está bien descrito, carece de una implementación que permita comprobar la utilidad práctica de la propuesta. En este trabajo, se implementa el análisis de la desclasificación basada en tipos para un subconjunto del lenguaje Dart, un lenguaje de programación de propósito general orientado a objetos desarrollado por Google.

Además, se implementó un sistema de inferencia de políticas de desclasificación y una extensión para ambientes de desarrollo, con el objetivo de facilitar el trabajo al programador y mejorar su experiencia.

A mis padres, por su apoyo incondicional.

Tabla de Contenido

Introducción	1
1.1. Objetivos	2
1.2. Organización del documento	2
2. Control de flujo de información	3
2.1. Tipado de seguridad	3
2.2. Desclasificación	5
3. Inferencia de tipos	9
3.1. Inferencia en lenguajes conocidos	9
3.1.1. Scala	9
3.1.2. Haskell	10
3.1.3. C++	10
3.2. Inferencia en sistemas Hindley-Milner	11
3.3. Inferencia en sistemas con subtipos	12
4. Inferencia de facetas públicas en Dart	16
4.1. Problema de inferencia de facetas públicas	16
4.2. Perspectiva general de la solución	17
4.3. Gramática de tipos	18
4.4. Conversión a facetas privadas	18
4.5. Generación de restricciones sobre subtipos	19
4.6. Resolución de restricciones sobre subtipos	22
4.6.1. Unificación	23
5. Implementación	28
5.1. Lenguaje Dart	28
5.1.1. Dart Analyzer	28
5.1.2. Analyzer Plugin	29
5.2. Implementación de sistema de inferencia	29
5.2.1. Representación de facetas públicas	29
5.2.2. Tipos de errores	30
5.2.3. Fase de generación de constraints	30
5.2.4. Fase de resolución de constraints	31
5.3. Implementación de plugin	33
5.3.1. Configuración del plugin	33

6. Validación	34
6.1. Programando con facetas públicas	34
6.2. Batería de tests	36
6.3. Repositorio de prueba	36
Conclusión	36

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.

Los sistemas computacionales seguros aplican múltiples técnicas de protección de confidencialidad de la información, las cuales pueden ser específicas a un nivel de comunicación de un programa. Por ejemplo, a nivel de red se utilizan protocolos de encriptación para cifrar mensajes confidenciales, y a nivel del sistema operativo el núcleo restringe el acceso a información manipulada por procesos independientes.

Existe un conjunto de técnicas de seguridad a nivel del código del programa, con el nombre de seguridad basada en el lenguaje (*language-based security*). Una de estas técnicas es control de acceso [14], que permite o deniega el acceso a recursos e información entre los distintos módulos de un programa. Estas técnicas pueden ser estáticas (análisis sintáctico) o dinámicas (análisis en tiempo de ejecución).

Una técnica de seguridad basada en el lenguaje se denomina tipado de seguridad, que consiste en clasificar la información manipulada por el programa agregando niveles de seguridad a los identificadores mediante anotaciones en el código fuente, lo que permite realizar un análisis estático o dinámico del flujo de la información. En este trabajo, nos centramos en el análisis estático del control de flujo de la información.

Los lenguajes con tipado de seguridad formalizan la protección de confidencialidad mediante una propiedad de no-interferencia (*noninterference*) [6], la cual puede ser muy restrictiva para aplicaciones reales y prácticas. Es por ello que los lenguajes con tipado de seguridad ofrecen mecanismos para desclasificar la información confidencial, y a su vez asegurar el cumplimiento de alguna propiedad de seguridad [13].

Uno de los mayores desafíos de los lenguajes con tipado 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. [3] recientemente propusieron la desclasificación basada en tipos (*type-based declassification*), una variante de tipado de seguridad que utiliza el sistema de tipos del lenguaje para controlar la desclasificación de la información.

El fundamento teórico de la desclasificación basada en tipos 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 inferencia de tipos, que consiste en asignar un tipo adecuado a las expresiones sin una anotación de tipo, con el fin de facilitar el trabajo al programador y mantener los beneficios de un lenguaje estáticamente tipado. En esta dirección, se han propuesto mecanismos de inferencia para tipos de seguridad [12], lo que motiva una proposición similar para la desclasificación basada en tipos.

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 ofrece herramientas para realizar análisis personalizado sobre el árbol sintáctico de un código fuente Dart. Estas herramientas pueden ser integradas a los ambientes de desarrollo (IDE) mediante extensiones, 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 la desclasificación basada en tipos. Dentro de los objetivos específicos del trabajo, podemos encontrar:

- **Inferencia y verificación estática de la desclasificación basada en tipos.** Se entiende como la implementación de un sistema de inferencia de facetas de desclasificación para la desclasificación basada en tipos, en el lenguaje de programación Dart.
- **Extensión para ambientes de desarrollo.** Mostrar al programador el resultado de la inferencia, por medio de una extensión para los ambientes de desarrollo 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 capítulo 5. En el último capítulo se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Control de flujo de información

El control de flujo de información es una técnica de seguridad que permite controlar las acciones que se pueden realizar con los valores que manipula un programa, y hacia dónde está permitido que fluyan. En este capítulo se presentan los antecedentes de tipado de seguridad para el control de flujo de información, y los conceptos importantes de la desclasificación basada en tipos.

2.1. Tipado de seguridad

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 un retículo¹ (*lattice*) de etiquetas de seguridad. Por ejemplo, con el retículo 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 de no-interferencia [6], esto es, que la información confidencial no fluya directa o indirectamente hacia canales públicos [15].

A modo de ejemplo, consideremos una aplicación móvil que permite encontrar un hotel cercano utilizando Google Maps, y luego reservar una habitación en ese hotel ingresando los datos de la tarjeta de crédito. La información confidencial involucrada en este caso son los datos de la tarjeta de crédito, que deben ser enviados al hotel para realizar la reserva. En el ejemplo 2.1 se muestra el código de una función `book` que realiza la reserva, y las funciones `sendToHotel` y `sendToGoogleMaps` que envían información a los respectivos servicios.

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

Ejemplo 2.1

```
String book(String username, int date, int cardNumber) {  
    return sendToHotel(username, date, cardNumber);  
}  
  
String sendToHotel(String username, int date, int cardNumber);  
String sendToGoogleMaps(String token, int xCoord, int yCoord);
```

La función `book` del ejemplo 2.1 no contiene fugas de información confidencial. Sin embargo, el programador puede cometer un error y llamar a la función `sendToGoogleMaps` en lugar de `sendToHotel`, con los mismos argumentos. El programa resultante estaría bien tipado, pero introduce una severa fuga de información confidencial al enviar el número de la tarjeta de crédito a Google. El uso de tipado de seguridad permitiría prevenir esta fuga de información, mediante la anotación del parámetro `cardNumber` de la función `book` con un nivel de seguridad de alta confidencialidad, y el parámetro `yCoord` de la función `sendToGoogleMaps` con un nivel de seguridad de baja confidencialidad. Esto se muestra en el ejemplo 2.2.

Ejemplo 2.2

```
String@L book(String@L username, int@L date, int@H cardNumber) {  
    return sendToGoogleMaps(username, date, cardNumber); /* error */  
}  
  
String@L sendToHotel(String@L username, int@L date, int@H cardNumber);  
String@L sendToGoogleMaps(String@H token, int@L xCoord, int@L yCoord);
```

El programa del ejemplo 2.2 es rechazado por el sistema de tipos, debido a un error causado por el llamado a la función `sendToGoogleMaps`, donde se disminuye el nivel de seguridad de `cardNumber` desde H a L, lo cual es una infracción a la propiedad de no-interferencia.

Los flujos de información que ocurren en la asignación directa de valores se denominan *flujos explícitos*. Para la detección de flujos explícitos que violan la propiedad de no-interferencia, los lenguajes con tipado de seguridad poseen distintas reglas que relacionan los niveles de seguridad involucrados. En una instrucción de asignación `x = y`, el nivel de seguridad de `y` debe ser igual o menor que el nivel de seguridad de `x`. En una instrucción de retorno `return y`, el nivel de seguridad de `y` debe ser igual o menor que el nivel de seguridad declarado como retorno de la función. En un llamado a una función, como ocurre en el ejemplo 2.2, el nivel de seguridad de cada argumento debe ser menor o igual que el nivel de seguridad del correspondiente parámetro.

Existen otros flujos de información que ocurren mediante la influencia indirecta que tienen algunas instrucciones o acciones del programa sobre su estado, denominados *flujos implícitos*. En el ejemplo 2.3 se muestra una función de `login` anotada con niveles de seguridad. Se considera que los valores literales son de baja confidencialidad.

Ejemplo 2.3

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

En el ejemplo 2.3 ocurre un flujo implícito que infringe con la propiedad de no-interferencia, debido a que un observador público puede obtener información del parámetro confidencial `password` observando cambios en el valor de retorno de la función, mediante el control de flujo del programa.

Es posible detectar un flujo implícito inválido considerando que las instrucciones de retorno y asignación de valores de baja confidencialidad, ocurren en un contexto de alta confidencialidad, determinado por la condición de la instrucción `if`. Para considerar el contexto de ejecución de una instrucción en las reglas del sistema de tipos, se utiliza el concepto de *contexto de seguridad* [8], usualmente llamado `pc` por *program counter*. Así, en el ejemplo 2.3 las instrucciones de retorno son inválidas, debido a que retornan valores de baja confidencialidad cuando el contexto de seguridad tiene un valor de alta confidencialidad.

2.2. Desclasificación

A pesar de que no-interferencia 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` del ejemplo 2.3 sea aceptado a pesar de no cumplir con la propiedad, pues de otra forma no tendríamos cómo realizar la autenticación.

Para solucionar este problema, los lenguajes de seguridad adicionan mecanismos de desclasificación que disminuyen el nivel de seguridad de un valor confidencial, implementados de diferentes formas [13]. Una de ellas, por ejemplo en Jif [9] es usar un operador `declassify`, que desclasifica un valor de alta confidencialidad retornando un valor de baja confidencialidad. En el ejemplo 2.4, se utiliza para desclasificar el resultado de la operación de comparación.

Ejemplo 2.4

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

A pesar de que este programa no cumple con no-interferencia, no representa 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 [13]. Por ejemplo, mediante la definición de políticas globales de desclasificación [7] se asegura una versión relajada de la propiedad de no-interferencia. En esta dirección, Cruz et al. [3] recientemente propusieron la desclasificación basada en tipos 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 la desclasificación basada en tipos, 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 $\text{StringEq} \triangleq [\text{eq} : \text{String} \rightarrow \text{Bool}]^2$ autoriza la operación `eq` sobre un `String`. Entonces se puede usar el tipo de dos facetas `String<StringEq`, donde `String` es la faceta privada y `StringEq` es la faceta pública, para controlar la operación de desclasificación de la igualdad sobre `password`, lo que se muestra en el ejemplo 2.5.

Ejemplo 2.5

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

En la desclasificación basada en tipos, se cumple que la faceta privada es subtipo de la faceta pública. En el ejemplo 2.5, `String` es subtipo de `StringEq`, relación que se escribe como `String <: StringEq`. Los tipos que cumplen con esta restricción se denominan bien formados (*well-formed*).

Al igual que en tipado de seguridad de dos o más niveles, las facetas de la desclasificación basada en tipos forman un retículo con relaciones de subtipos, lo que se ejemplifica en la figura 2.1. 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 del retículo. Cuando se quiere referir a una faceta pública vacía o que no autoriza ninguna operación, se usa `Top`, por encontrarse en la parte superior del retículo.

²La notación $[m1 : t_1 \rightarrow t_2, m2 : t_3 \rightarrow t_4]$ corresponde al tipo de un objeto que contiene a los métodos `m1` y `m2`, y la notación $t_1 \rightarrow t_2$ corresponde al tipo de una función con parámetro de tipo t_1 y retorno de tipo t_2

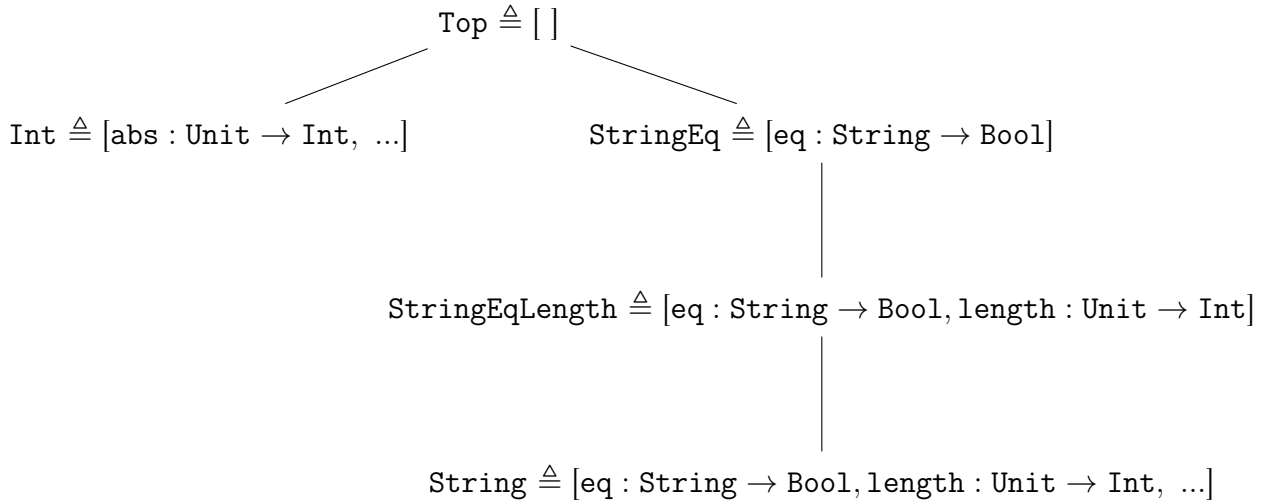


Figura 2.1: Retículo de subtipos

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 ≜ [eq : String<String → Bool<Bool]`.

Existen dos reglas principales para comprobar que un programa con tipos de dos facetas se encuentra bien tipado. Consideremos el siguiente ejemplo, en donde `StringHashEq ≜ [hash : Unit<Unit → String<StringEq]`.

Ejemplo 2.6

```

String<StringEq getHash(String<StringHashEq password) {
    return password.hash();
}

```

En el ejemplo 2.6, el valor de retorno de la invocación al método `hash` sobre el parámetro `password`, tiene faceta pública `StringEq`, debido a que fue declarado de esta forma en la faceta pública `StringHashEq`. A esta regla se le llama TmD y se muestra en la figura 2.2.

Ahora consideremos el siguiente ejemplo, en donde se cambia la faceta pública del parámetro `password` del ejemplo 2.6 por `StringEq`.

Ejemplo 2.7

```

String<Top getHash(String<StringEq password) {
    return password.hash();
}

```

En el ejemplo 2.7 se realiza una invocación al método **hash** sobre el parámetro **password**, que declara una faceta pública que no autoriza la operación. Cuando esto sucede, la faceta pública de retorno de la invocación es **Top**. A esta regla se le llama TmH y se muestra en la figura 2.2.

$$\frac{\Gamma \vdash e_1 : T < U \quad m \in U \quad \text{methsig}(U, m) = S_1 \rightarrow S_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2} \text{ (TmD)}$$

$$\frac{\Gamma \vdash e_1 : T < U \quad m \notin U \quad \text{methsig}(T, m) = S_1 \rightarrow T_2 < U_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 < \top} \text{ (TmH)}$$

Figura 2.2: Reglas principales del sistema de tipos de la desclasificación basada en tipos. Cada regla se compone de una serie de premisas en la parte superior, y una conclusión en la parte inferior. El juicio de la forma $\Gamma \vdash e : T$ indica que en el contexto Γ la expresión e tiene tipo T . El procedimiento auxiliar $\text{methsig}(O, m)$ entrega la firma del método m en el tipo de objeto O . Un tipo S es un tipo de dos facetas $T < U$.

La propiedad de seguridad que se demuestra para el sistema de tipos de la desclasificación basada en tipos es una forma de no-interferencia con políticas de desclasificación, denominada no-interferencia relajada (*relaxed noninterference*). Un lenguaje de seguridad que cumple con esta propiedad, garantiza que la información confidencial solo puede fluir hacia canales públicos de forma controlada, por medio de las políticas de desclasificación.

Resumen

En este capítulo se revisaron los conceptos de tipado de seguridad para el control de flujo de información, y las distintas reglas que permiten detectar la ocurrencia de infracciones a la propiedad de no-interferencia. Luego, se mostró la necesidad de contar con mecanismos de desclasificación, para finalizar explicando la desclasificación basada en tipos.

Es importante considerar que la implementación de un lenguaje de programación con el sistema de tipos de la desclasificación basada en tipos, requiere la anotación de los identificadores con facetas públicas y privadas, lo cual es poco práctico. Por este motivo es deseable contar con un sistema de inferencia de tipos para la desclasificación basada en tipos. En el siguiente capítulo se revisan los conceptos importantes de inferencia de tipos.

Capítulo 3

Inferencia de tipos

La inferencia de tipos es el proceso de determinar el tipo 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, y mantener los beneficios de un lenguaje estáticamente tipado. Además, aumenta la escalabilidad de los sistemas, ya que el costo de refactorizar código anotado es mayor al costo de refactorizar código no anotado.

3.1. Inferencia en lenguajes conocidos

Las características de los algoritmos de inferencia varían dependiendo de las características de los lenguajes de programación. A continuación se describe la inferencia de tipos en lenguajes conocidos.

3.1.1. Scala

Scala es un lenguaje de programación estáticamente¹ tipado que combina los paradigmas de la programación orientada a objetos y la programación funcional. Scala posee una inferencia de tipos local [11], que permite omitir las anotaciones de tipos en la declaración de variables locales, en el uso de clases y métodos polimórficos, y en el tipo de retorno de métodos no recursivos. Esto se muestra en el ejemplo 3.1, donde se infiere el tipo de los identificadores `p` y `q`. `MyPair` es una clase polimórfica especial que define un constructor con dos parámetros. `id` define la función identidad polimórfica.

¹Un lenguaje estáticamente tipado es aquel que comprueba en tiempo de compilación la correctitud de los tipos del programa

Ejemplo 3.1

```
case class MyPair[A, B](x: A, y: B);
object InferenceTest {
  def id[T](x: T) = x
  val p = MyPair(1, "scala") // type: MyPair[Int, String]
  val q = id(1)              // type: Int
}
```

3.1.2. Haskell

Haskell es un lenguaje de programación funcional estáticamente tipado, cuyo sistema de tipos se basa en el sistema de tipos Hindley-Milner (HM) [4]. Este sistema de tipos tiene inferencia global, es decir, es capaz de inferir los tipos principales² en un programa sin anotaciones de tipo. Haskell es capaz de inferir tipos con polimorfismo paramétrico, lo que significa que se pueden declarar funciones polimórficas como en Java y Scala sin necesidad de indicar un tipo genérico. En la parte superior del ejemplo 3.2 se muestra la definición de la función `map` en Haskell, y en la parte inferior el tipo inferido. Este tipo corresponde al tipo de una función, que dado una función de tipo $a \rightarrow b$ y una lista de tipo a , retorna una lista de tipo b .

Ejemplo 3.2

```
map f [] = []
map f (first:rest) = f first : map f rest

-----
map :: (a -> b) -> [a] -> [b]
```

3.1.3. C++

C++ es un lenguaje de programación orientado a objetos estáticamente tipado que posee herramientas para realizar programación de bajo nivel. C++ ofrece la posibilidad de omitir el tipo en la declaración de variables mediante los keywords `auto` y `decltype`, lo que se muestra en el ejemplo 3.3.

Ejemplo 3.3

```
auto n = 1;      // type: int
string s = "abc";
decltype(s) k;   // type: string
```

²Un tipo principal es el tipo más general que puede ser inferido para una expresión, en el sentido de que cualquier otro tipo posible es una especialización del tipo principal

3.2. Inferencia en sistemas Hindley-Milner

Un sistema de tipos de Hindley-Milner (HM) [4] es un sistema de tipos clásico del cálculo lambda con polimorfismo paramétrico. El algoritmo de inferencia del sistema de tipos HM, denominado algoritmo W, siempre calcula el tipo principal de las expresiones. En esta sección se explica el proceso de inferencia de tipos para un lenguaje sencillo de tipo HM, con el fin de ilustrar los conceptos importantes de inferencia de tipos.

El lenguaje a considerar posee operaciones aritméticas entre enteros, funciones y expresiones condicionales. La sintaxis se muestra en el ejemplo 3.4.

Ejemplo 3.4 `let g = (x,c) => if (c) then x + 5 else x;`

En el ejemplo 3.4, se asigna a `g` una función que retorna la suma entre `x` y 5 si `c` es `true`, o `x` en caso contrario. Notar que no se anotó el tipo de los identificadores `g`, `x` y `c`.

En la etapa de tipar un programa (*type checking*), los sistemas de inferencia asignan una variable de tipo a cada expresión sin un tipo conocido, y un tipo concreto³ (por ejemplo, `int`) a cada expresión con tipo conocido. Además, generan un conjunto de restricciones que se deben cumplir para que cada expresión esté bien tipada.

Una restricción representa una relación entre dos tipos. Esta relación puede ser de igualdad o de subtipos. El uso de restricciones permite presentar un algoritmo de inferencia de forma modular, como una fase de generación de restricciones, y una fase de resolución de restricciones.

La figura 3.1 muestra la asignación de tipos a cada una de las expresiones del ejemplo 3.4, y las restricciones que se generan.

Expresión	Tipo	Restricciones
<code>(x,c) => if (c) then x + 5 else x</code>	<code>X</code>	<code>X = (Y,Z) → W</code>
<code>x</code>	<code>Y</code>	-
<code>c</code>	<code>Z</code>	-
<code>if (c) then x + 5 else x</code>	<code>W</code>	<code>W = V, W = Y</code>
<code>c</code>	<code>Z</code>	<code>Z = bool</code>
<code>x + 5</code>	<code>V</code>	<code>Y = int, V = int</code>
<code>+</code>	<code>(int,int) → int</code>	-
<code>x</code>	<code>Y</code>	-
<code>5</code>	<code>int</code>	-
<code>x</code>	<code>Y</code>	-

Figura 3.1: Etapa de *type checking*

Las restricciones generadas para el ejemplo 3.4 representan la igualdad entre dos tipos. Las siguientes observaciones permiten derivar el conjunto de restricciones del programa.

³Un tipo concreto no tiene variables de tipo

1. $(x, c) \Rightarrow \text{if } (c) \text{ then } x + 5 \text{ else } x$ es una función anónima que debe tener tipo $(Y, Z) \rightarrow W$, donde Y y Z son los tipos de los parámetros y W es el tipo del cuerpo.
2. $\text{if } (c) \text{ then } x + 5 \text{ else } x$ es una expresión condicional, cuyo tipo W es igual al tipo de ambas ramas de la expresión.
3. La condición c de la expresión condicional debe tener tipo `bool`.
4. $x + 5$ es una aplicación de la función suma, por lo que su tipo debe coincidir con el tipo de retorno de la función. Además, el tipo de los argumentos debe coincidir con el tipo de los parámetros de la función.

Sintetizando, el conjunto de restricciones generado es el siguiente:

$$R_1 : \{X = (Y, Z) \rightarrow W, W = V, W = Y, Z = \text{bool}, Y = \text{int}, V = \text{int}\}$$

Una vez que se genera el conjunto de restricciones sobre el programa, se procede a encontrar una solución para las variables de tipo del conjunto. Cuando las restricciones representan relaciones de igualdad, se utiliza el algoritmo de unificación de Hindley-Milner [4]. Este algoritmo genera un diccionario de variables de tipo a tipos concretos, mediante substituciones desde la última restricción generada a la primera restricción generada (*bottom-up*). La figura 3.2 muestra la solución esperada del ejemplo 3.4.

Variable de tipo	Tipo concreto
X	$(\text{int}, \text{bool}) \rightarrow \text{int}$
Y	<code>int</code>
Z	<code>bool</code>
W	<code>int</code>
V	<code>int</code>

Figura 3.2: Solución del conjunto de restricciones

A lo largo del tiempo se han desarrollado diversas extensiones al sistema de tipos HM, para brindar características avanzadas manteniendo la completitud del algoritmo W. Odersky et al. [10] desarrollaron el framework HM(X), que entrega un algoritmo de inferencia genérico que calcula los tipos principales para cualquier sistema de tipos basado en restricciones, que cumpla con ciertas propiedades. Utilizando el framework HM(X), Pottier y Simonet [12] presentaron un análisis de control de flujo con inferencia de tipos de seguridad.

3.3. Inferencia en sistemas con subtipos

Los lenguajes de programación orientados a objetos como Java, Scala y OCaml permiten la definición de nuevos tipos mediante la creación de clases. Estos tipos se relacionan de forma explícita (nominal) o implícita (estructural), y se integran al retículo de subtipos definido previamente por el lenguaje de programación.

En una relación de subtipos nominal, el retículo de subtipos es definido explícitamente por el programador o el lenguaje de programación. El ejemplo 3.5 muestra una declaración explícita de una jerarquía de tipos, mediante el uso del *keyword* `extends`.

Ejemplo 3.5

```
class Animal {
    String getName();
}
class Duck extends Animal {
    void cuack();
}
class Cat extends Animal {
    void miau();
}
```

Cuando una relación de subtipos es estructural, el retículo de subtipos se define por la forma que tienen los tipos. El ejemplo 3.6 muestra la declaración de las clases que definen una relación de subtipos estructural.

Ejemplo 3.6

```
class Animal {
    String getName();
}
class Duck {
    String getName();
    void cuack();
}
class Cat {
    String getName();
    void miau();
}
```

Para realizar inferencia en sistemas con subtipos, se utilizan restricciones sobre subtipos en lugar de restricciones sobre igualdad. Desafortunadamente, el algoritmo de unificación de Hindley-Milner no funciona cuando las restricciones representan una relación de subtipos.

Una forma de resolver restricciones sobre subtipos es utilizar operaciones sobre el retículo de subtipos. Consideremos una extensión del lenguaje de la sección 3.2, en donde se introducen los tipos `Top`, `num` y `float`, tal que `num <: Top`, `bool <: Top`, `int <: num` y `float <: num`. En el ejemplo 3.7 se muestra una expresión condicional que retorna valores de distinto tipo en las ramas.

Ejemplo 3.7 `let g = (c) => if (c) then 1 else 0.5;`

Siguiendo la misma lógica de la inferencia en sistemas HM, se asigna una variable de tipo X a la expresión condicional y una variable de tipo Y a la condición, y se generan las siguientes restricciones sobre subtipos:

$$R_2 : \{X <: \text{int}, X <: \text{float}, Y <: \text{bool}\}$$

Para resolver este tipo de restricciones, se considera la relación de orden parcial entre los tipos, inducida por la relación de subtipos, que se muestra en la figura 3.3.

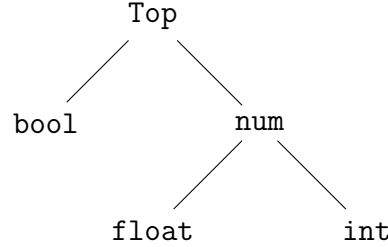


Figura 3.3: Orden parcial entre los tipos

Un orden parcial admite las operaciones **meet** y **join**, que corresponden a la máxima cota inferior (ínfimo) y a la mínima cota superior (supremo) entre dos tipos, respectivamente. Estas operaciones tienen solución si todo par de tipos en el orden parcial tiene un único supremo e ínfimo, lo que se define como retículo. En la figura 3.3, se debe agregar el tipo **Bot** para que el orden parcial sea un retículo.

Luego, mediante los tipos $t_1 \sqcap t_2$ y $t_1 \sqcup t_2$ introducidos por Cardelli [2], se representan las operaciones **meet** y **join** entre dos tipos, respectivamente.

En la fase de resolución de restricciones, se aplican las reglas de la proposición 3.8 para generar los tipos correspondientes. Luego, si al final del algoritmo de inferencia los tipos \sqcap y \sqcup relacionan tipos concretos, se puede materializar la operación sobre el retículo, es decir, encontrar el ínfimo o el supremo respectivamente.

Proposición 3.8 Si x , y y z pertenecen a una retículo de subtipos, se cumple lo siguiente:

- $x <: y, x <: z \implies x <: y \sqcap z$
- $y <: x, z <: x \implies y \sqcup z <: x$

En el ejemplo 3.7, las restricciones generadas se reducen con la aplicación de la proposición 3.8, lo que da como resultado las restricciones $\text{int} \sqcup \text{float} <: X$ y $Y <: \text{bool}$. Como int y float son tipos concretos, se puede materializar la operación **join**, lo que da como resultado **num**, por lo que se resuelve que X tiene tipo **num** y que Y tiene tipo **bool**.

La inferencia global al estilo HM en sistemas con subtipos ha sido estudiada por distintos autores [11]. Sin embargo, se reconoce que esta combinación no funciona bien en la práctica, debido a que el tamaño del conjunto de restricciones crece de forma exponencial, lo cual dificulta la identificación y despliegue de errores de tipo [11].

Resumen

En este capítulo se mencionaron los usos y beneficios de la inferencia de tipos, se mostraron las características generales de los sistemas de inferencia en Scala, Haskell y C++, se explicaron los conceptos básicos del proceso de inferencia utilizando un sistema Hindley-Milner, y se mostró una forma de resolver restricciones sobre subtipos. En el siguiente capítulo se presenta la propuesta de un sistema de inferencia para la desclasificación basada en tipos.

Capítulo 4

Inferencia de facetas públicas en Dart

En este trabajo se propone realizar la implementación de un sistema de inferencia de facetas públicas en el lenguaje Dart, que incluya el análisis de la desclasificación basada en tipos, y luego la realización de una extensión para ambientes de desarrollo. En este capítulo se detalla el problema de inferencia a resolver y las estrategias utilizadas para resolverlo.

4.1. Problema de inferencia de facetas públicas

Para la formulación del problema, es posible asumir que la información de las facetas privadas de la desclasificación basada en tipos se encuentra a disposición, debido a que se pueden convertir los tipos estáticos comunes (como `String`) de Dart en facetas privadas. Los detalles de esta conversión se explican en la sección 4.4.

Definición 4.1 (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 el programa sea bien tipado.*

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

Ejemplo 4.2

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

Recordemos que la regla TmH de la desclasificación basada en tipos indica que la invocación a un método no autorizado sobre el receptor de la llamada, retorna `Top`. Esto es lo que se quiere inferir para el tipo de retorno del método `login` en el ejemplo 4.2.

Ejemplo 4.3

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

En el ejemplo 4.3 ocurrió un encadenamiento de invocaciones a métodos sobre `password`. La faceta pública que se quiere inferir para `password` contiene al método `hash`, al cual se le quiere inferir una faceta pública de retorno que contiene al método `eq`. Si se declara una faceta pública de retorno para el método `eq`, entonces se debe inferir esa misma faceta para el retorno del método `login`, por la aplicación de la regla TmD de la desclasificación basada en tipos.

4.2. Perspectiva general de la solución

Para resolver el problema de inferencia, se decidió realizar la implementación de un algoritmo de inferencia global basado en restricciones sobre subtipos, con tipos estructurales. Como el objetivo de este trabajo no es formalizar un sistema de inferencia para la desclasificación basada en tipos, se implementa un algoritmo heurístico reutilizando enfoques e ideas de trabajos con características similares, lo cual fue presentado en el capítulo 3. La propuesta de solución se divide en los siguientes pasos:

1. Definición de los tipos utilizados por el sistema de inferencia mediante la gramática de tipos.
2. Integración de los tipos estáticos comunes de Dart con las facetas de la desclasificación basada en tipos.
3. Descripción del paso de generación de restricciones para un programa de Dart.
4. Descripción del paso de resolución de restricciones.

El subconjunto del lenguaje Dart a considerar en la implementación tendrá clases, métodos, variables, asignaciones e instrucciones condicionales.

Con respecto al control de flujo de información, como el subconjunto del lenguaje a considerar tiene instrucciones condicionales, se deben detectar flujos implícitos para preservar la propiedad de no-interferencia relajada, por lo que se utiliza el concepto de contexto de seguridad presentado en el capítulo 2.

4.3. Gramática de tipos

Por lo discutido en el capítulo 3, es necesario introducir variables de tipo para presentar un algoritmo de inferencia basado en restricciones. Además, se incluyen los constructores \sqcap y \sqcup que representan las operaciones **meet** y **join** sobre un orden parcial, y se definen los otros tipos que serán utilizados en el análisis. Esto se hace definiendo la gramática de tipos que usará el sistema de inferencia:

$$\tau := \alpha \mid \text{Obj}(\overline{1 : \tau}) \mid \overline{\tau} \rightarrow \tau \mid \tau \sqcap \tau \mid \tau \sqcup \tau \mid \text{Bot} \mid \text{Top}$$

Donde α es cualquier variable de tipo. La barra superior indica una lista de elementos. Por ejemplo, $\text{Obj}(\overline{1 : \tau})$ representa el tipo de un objeto con una lista de métodos de nombre 1.

4.4. Conversión a facetas privadas

Para que la implementación del sistema de tipos de la desclasificación basada en tipos se integre con Dart, se debe realizar una conversión de los tipos que son incluidos por defecto en todo programa de Dart (*core* de Dart), como **String** y **List**, a facetas privadas de la desclasificación basada en tipos.

Sea $[m_i : \overline{A_i} \rightarrow B_i]$ un tipo de objeto con métodos m_i . La operación **convert** se define como sigue:

$$\text{convert}([m_i : \overline{A_i} \rightarrow B_i]) = [m_i : \overline{A_i < P_{A_i}} \rightarrow B_i < P_{B_i}]$$

Lo que hace **convert** es reemplazar el tipo de objeto de Dart por una faceta privada, donde los tipos de los parámetros y retorno de cada método se convirtieron a tipos de dos facetas, con faceta pública P_{A_i} y P_{B_i} respectivamente.

Las facetas públicas P_{A_i} y P_{B_i} se pueden establecer según distintos criterios. En un sistema de tipos con polimorfismo, podemos considerarlas como variables de tipo cuantificadas, que pueden tomar distintos valores. Aquello está fuera del alcance de este trabajo, por lo que se debe tomar una decisión respecto al valor que tendrán las facetas públicas en la operación **convert**.

Hay dos opciones para tipar las facetas públicas: **Top** y **Bot**, debido a que son las únicas facetas públicas que se conocen de antemano. Primero, veamos las opciones para la faceta pública de retorno P_{B_i} . Si decidimos que P_{B_i} es igual a **Top**, entonces el sistema de inferencia terminará asignando **Top** a cada expresión que se asocie con un llamado a un método del *core* de Dart, lo cual se propaga rápidamente. Esto es poco útil, debido a que el sistema de inferencia detectará flujos inválidos cuando en la práctica no hay una fuga de información. Por lo tanto, la decisión por defecto es que P_{B_i} sea **Bot**.

Ahora, analicemos ambas posibilidades para P_{A_i} :

- **Top:** 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:** 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` tiene el mejor equilibrio entre utilidad y seguridad, por lo que es la opción por defecto considerada. Con esto, la versión utilizada de `convert` es:

$$\text{convert}([m_i : \overline{A_i} \rightarrow B_i]) = [m_i : \overline{A_i} < \perp \rightarrow B_i < \perp]$$

4.5. Generación de restricciones sobre subtipos

Como se mencionó en la sección 3.2, el uso de restricciones permite presentar un algoritmo de inferencia como una fase de generación de restricciones, y una fase de resolución de restricciones.

El algoritmo de generación de restricciones es un algoritmo recursivo que se ejecuta sobre un nodo del árbol de sintaxis abstracta (AST) del programa y retorna un tipo y un conjunto de restricciones. A continuación se explican informalmente las reglas de generación de restricciones, y luego se muestra el algoritmo en pseudocódigo.

Invocación a método. Cuando se procesa un nodo que representa una invocación a método, se genera la restricción `(target <: Obj(name: signature), exp)`, donde `target` es el tipo del receptor de la invocación, `Obj(name: signature)` es un tipo de objeto que contiene al método invocado, y `exp` es el tipo de la expresión de invocación. El tipo de la expresión se almacena en la restricción debido a la posible aplicación de la regla TmH de la desclasificación basada en tipos, lo cual se explica en la sección 4.6.1. Además, por cada argumento de la invocación, se genera la restricción `(arg <: par)`, donde `arg` es el tipo del argumento y `par` es el tipo del correspondiente parámetro del método.

Expresión de retorno. Cuando se procesa un nodo que representa una expresión de retorno, se genera la restricción `(exp <: ret)`, donde `exp` es el tipo de la expresión de retorno y `ret` es el tipo de retorno en la firma del método. Además, se genera la restricción `(pc <: ret)`, donde `pc` es el contexto de seguridad, debido a que la expresión de retorno puede ocurrir dentro del cuerpo de un condicional.

Expresión de asignación. Cuando se procesa un nodo que representa una expresión de asignación, se genera la restricción ($\text{right} <: \text{left}$), donde right es el tipo del lado derecho de la asignación y left es el tipo del lado izquierdo de la asignación. Además, se genera la restricción ($\text{pc} <: \text{left}$), por el mismo motivo que en la expresión de retorno.

Expresión condicional. Cuando se procesa un nodo que representa una expresión condicional (como `if` y `while`), se cambia el contexto de seguridad por la faceta pública de la condición, debido a que será utilizado por el algoritmo en el cuerpo del condicional.

El algoritmo 1 muestra la generación de restricciones para un nodo determinado del AST. Se asume la existencia de un almacén (*store*) que almacena los tipos de los identificadores. En la línea 8 se asigna una variable de tipo a `type`. La asignación de la línea 9 asigna el tipo retornado por la función `Constraint_Generation` a `targetType` y el conjunto de restricciones retornado por la misma función a `targetCS`.

En el ejemplo 4.4 se muestra un código parcialmente anotado con facetas públicas. En este ejemplo, se espera inferir una faceta pública para `password` que contenga a los métodos `eq` y `hash`. En esta sección se muestra el conjunto de restricciones generado para el ejemplo 4.4, mientras que en la sección 4.6 se muestra la resolución de este conjunto.

Ejemplo 4.4

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

1. ($\alpha <: \text{Obj}(\text{eq} : [\text{Bot}] \rightarrow \text{Bot})$, β)
2. ($\alpha <: \text{Obj}(\text{hash} : [] \rightarrow \text{Bot})$, γ)
3. ($\text{Bot} <: \text{Bot}$)
4. ($\text{Bot} <: \text{Bot}$)
5. ($\gamma <: \text{Bot}$)
6. ($\text{Bot} <: \text{Bot}$)

La variable de tipo α corresponde al tipo del parámetro `password`. La variable de tipo β corresponde al tipo de la condición en la expresión condicional. La variable de tipo γ corresponde al tipo de la expresión de la invocación a método `password.hash()`.

Las restricciones 1 y 2 se generan por las llamadas a métodos sobre el parámetro `password`. Las restricciones 3 y 4 se generan por la relación entre el contexto de seguridad (`Bot`) y el retorno del método (`Bot`) en ambas ramas de la expresión condicional, y las restricciones 5 y 6 se generan por la relación entre la expresión de retorno (γ y `Bot`) y el retorno del método en ambas ramas de la expresión condicional.

Algoritmo 1Generación de restricciones

```
1: function CONSTRAINT_GENERATION(node, pc)
2:   cs ← {}
3:   type ← Bot
4:   switch node do
5:     case Identifier(name)
6:       type ← Store.getType(name)
7:     case MethodInvocation(name, target, signature, arguments)
8:       type ←  $\alpha$ 
9:       targetType, targetCS ← Constraint_Generation(target, pc)
10:      cs.insert(targetType <: Obj(name: signature), type)
11:      cs.addAll(targetCS)
12:      for argument, parType in arguments do
13:        argType, argCS ← Constraint_Generation(argument, pc)
14:        cs.insert(argType <: parType)
15:        cs.addAll(argCS)
16:      end for
17:     case ReturnStatement(expression, methodReturn)
18:       expType, expCS ← Constraint_Generation(expression, pc)
19:       cs.insert(expType <: methodReturn)
20:       cs.insert(pc <: methodReturn)
21:       cs.addAll(expCS)
22:     case AssignmentExpression(leftHand, rightHand)
23:       leftType, leftCS ← Constraint_Generation(leftHand, pc)
24:       rightType, rightCS ← Constraint_Generation(rightHand, pc)
25:       cs.insert(rightType <: leftType)
26:       cs.insert(pc <: leftType)
27:       cs.addAll(leftCS, rightCS)
28:     case IfExpression(conditionExpression, thenExpression, elseExpression)
29:       condType, condCS ← Constraint_Generation(conditionExpression, pc)
30:       thenType, thenCS ← Constraint_Generation(thenExpression, condType)
31:       elseType, elseCS ← Constraint_Generation(elseExpression, condType)
32:       cs.addAll(condCS, thenCS, elseCS)
33:   return type, cs
34: end function
```

Además de la generación de restricciones, en este paso se verifica que las expresiones que tienen facetas públicas declaradas cumplan con la restricción de tipos bien formados. Esto se muestra en el algoritmo 2, para un nodo determinado del AST.

Algoritmo 2

Verificación de tipos bien formados

```
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
```

4.6. Resolución de restricciones sobre subtipos

El primer paso en la resolución de restricciones es la eliminación de restricciones *obvias*. Esto es, la eliminación de las restricciones $(\text{Bot} <: X)$ y $(X <: \text{Top})$, ya que no aportan información útil al algoritmo de inferencia.

Algoritmo 3

Simplificación de restricciones

```
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 conjunto de restricciones del ejemplo 4.4 se reduce a solo tres restricciones, lo que se muestra en el ejemplo 4.5.

Ejemplo 4.5

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

El siguiente paso es agrupar las restricciones sobre la misma variable de tipo, usando las reglas del teorema 3.8. Aplicando el algoritmo 4, el conjunto de restricciones del ejemplo 4.5 se reduce a solo dos restricciones, lo que se muestra en el ejemplo 4.6.

Ejemplo 4.6

1. $(\alpha <: \text{Obj}(\text{eq} : [\text{Bot}] \rightarrow \text{Bot}) \sqcap \text{Obj}(\text{hash} : [] \rightarrow \text{Bot}))$
2. $(\gamma <: \text{Bot})$

Algoritmo 4**Agrupación de restricciones**

```
1: function GROUP(cs, typeVariables)
2:   toRemove  $\leftarrow \{\}$ 
3:   for tvar in typeVariables do
4:      $c_1 \leftarrow \text{Constraint}(\text{tvar}, \sqcap())$ 
5:      $c_2 \leftarrow \text{Constraint}(\sqcup(), \text{tvar})$ 
6:     for constraint in cs do
7:       if constraint.left == tvar then
8:          $c_1.\text{right.insert}(\text{constraint.right})$ 
9:         toRemove.insert(constraint)
10:      end if
11:      if constraint.right == tvar then
12:         $c_2.\text{left.insert}(\text{constraint.left})$ 
13:        toRemove.insert(constraint)
14:      end if
15:    end for
16:    if  $c_1.\text{right.notEmpty}()$  then
17:      cs.insert( $c_1$ )
18:    end if
19:    if  $c_2.\text{left.notEmpty}()$  then
20:      cs.insert( $c_2$ )
21:    end if
22:  end for
23:  cs.removeAll(toRemove)
24: end function
```

4.6.1. Unificación

En este paso, se construyen tipos utilizando las operaciones **meet** y **join** sobre los tipos \sqcap y \sqcup que relacionan tipos concretos, lo que significa calcular la unión y la intersección respectivamente entre los tipos relacionados. Además, se comprueba la validez de las restricciones y se realizan substituciones de forma iterativa.

Meet y Join

Cuando se tiene un tipo $t_1 \sqcap t_2$ o $t_1 \sqcup t_2$, donde t_1 y t_2 no tienen variables de tipo, entonces se puede construir el tipo correspondiente. Este procedimiento se muestra en el algoritmo 5. La operación **meet** consiste en la unión de los métodos en t_1 y t_2 , y **join** consiste en la intersección de los métodos en t_1 y t_2 .

Si aplicamos el algoritmo 5 al conjunto de restricciones del ejemplo 4.6, se construye el tipo de la derecha en la restricción 1. Esto se muestra en el ejemplo 4.7.

Ejemplo 4.7

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

Algoritmo 5

Construcción de tipos

```
1: function PERFORMOPERATIONS(cs)
2:   for constraint in cs do
3:     switch constraint.right do
4:       case  $t_1 \sqcap t_2$ 
5:         constraint.right  $\leftarrow \text{Obj}(t_1.\text{methods} \cup t_2.\text{methods})$ 
6:       switch constraint.left do
7:         case  $t_1 \sqcup t_2$ 
8:           constraint.left  $\leftarrow \text{Obj}(t_1.\text{methods} \cap t_2.\text{methods})$ 
9:     end for
10: end function
```

Verificación de restricciones

Cuando una restricción representa una relación no válida, existen dos casos posibles, dependiendo del origen de la restricción:

1. Si la restricción 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 restricción, en el conjunto de restricciones, debido a la aplicación de la regla TmH de la desclasificación basada en tipos. En este caso no se debe reportar error.

En el ejemplo 4.2, las restricciones relevantes generadas son:

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

Donde β es la faceta pública de retorno de **login**. Como la restricción 1 representa una relación no válida y proviene de invocación a método, el tipo de expresión α debe substituirse por **Top** en el conjunto de restricciones. Luego, la restricción 2 cambia a $(\text{Top} <: \beta)$.

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

Algoritmo 6Verificación de restricciones

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

Algoritmo 7Substitución 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  $x \rightarrow y$ 
9:       return substitute(x,target,newType)  $\rightarrow$  substitute(y,target,newType)
10:    case  $x \sqcap y$ 
11:      return substitute(x,target,newType)  $\sqcap$  substitute(y,target,newType)
12:    case  $x \sqcup y$ 
13:      return substitute(x,target,newType)  $\sqcup$  substitute(y,target,newType)
14:    case  $\alpha$ 
15:      return  $\alpha$ 
16:   return source
17: end function
```

Substitución

Cuando se tiene una restricción que relaciona una variable de tipo con un tipo concreto, se debe substituir en el conjunto de restricciones toda aparición de la variable de tipo, por el tipo concreto. Como este proceso puede generar nuevas restricciones que sean candidatas a construcción de nuevos tipos, a verificación o a substitución, se debe iterar hasta que no queden restricciones candidatas. El algoritmo 8 muestra la substitución de restricciones resueltas, mientras que el algoritmo 9 muestra el procedimiento completo de unificación.

Algoritmo 8**Substitución de restricciones**

```
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(substueResolved(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 significa que cualquier faceta pública sirve para validar la expresión según las reglas del sistema de tipos.

Resumen

En este capítulo se describió la propuesta de un algoritmo de inferencia heurístico para la desclasificación basada en tipos. Este algoritmo se basa en la inferencia en sistemas Hindley-Milner, con variaciones en el paso de resolución de restricciones para tratar con restricciones sobre subtipos. En el próximo capítulo se describen detalles de la implementación en Dart.

Capítulo 5

Implementación

En esta sección se detalla la implementación de la propuesta de solución, 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.

5.1. 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.

5.1.1. Dart Analyzer

Dart Analyzer 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. Dart Analyzer utiliza el patrón Visitor para incorporar un nuevo análisis sobre el AST, y así facilitar su integración con otros análisis.

5.1.2. 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.

5.2. Implementación de sistema de inferencia

5.2.1. 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 5.1. 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.

5.2.2. 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. Este error almacena el nodo del AST en el cual la constraint fue generada, para informar al usuario con mayor precisión la ubicación y la causa del fallo. 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 carácter 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.

5.2.3. 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 5.1.

La clase encargada de procesar las clases declaradas en un archivo es **CompilationUnitVisitor**. 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 4.5.

La clase **Store** es la encargada de la generación de variables de tipo, y el almacenamiento en diccionarios del tipo de las expresiones y elementos. Una expresión es un ente sintáctico representado por un nodo en el AST, mientras que un elemento es un ente semántico que fue declarado con un nombre en algún lugar del programa.

Por ejemplo, el nodo **MethodInvocation** representa una expresión de invocación a método. Desde este nodo es posible obtener un **MethodElement**, que representa la declaración del método invocado.

El diccionario que almacena el tipo de las expresiones es necesario para almacenar in-

formación que no es posible acceder desde nodos que la necesitan. Por ejemplo, en el nodo `ReturnStatement` se genera una constraint entre la expresión de retorno y el retorno del método. La expresión de retorno es un hijo del nodo `ReturnStatement`, por lo que debe ser visitado antes de generar la constraint, para que guarde el tipo de la expresión en el diccionario y pueda ser obtenido en el nodo padre.

El diccionario que almacena el tipo de los elementos es necesario para la generación de constraints sobre identificadores. Por ejemplo, en una invocación a método sobre una variable, se debe obtener el tipo del `VariableElement` para generar la constraint de invocación a método. Además, al final del proceso de inferencia, los elementos incluidos en este diccionario deben ser marcados en el código fuente para informar el tipo inferido al programador.

En la fase de generación de constraints 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.

5.2.4. Fase de resolución de constraints

En esta fase, la clase `ConstraintSolver` se encarga de convertir el set de constraints en un mapeo entre variables de tipo y tipos concretos, implementando las operaciones descritas en la sección 4.6. Esta clase se muestra en el diagrama 5.1.

En el algoritmo de unificación, se pueden generar errores del tipo `FlowError`, al momento de verificar las constraints.

Al final del proceso de resolución, se extraen las facetas públicas desde el diccionario que almacena el tipo de los elementos, que serán informadas al programador mediante errores del tipo `InferredFacetInfo`. Si un elemento está asociado a una variable de tipo, se genera un error del tipo `UnableToResolveInfo`.

Los errores generados en esta fase son recolectados mediante el mismo `ErrorCollector` de la fase de generación de constraints.

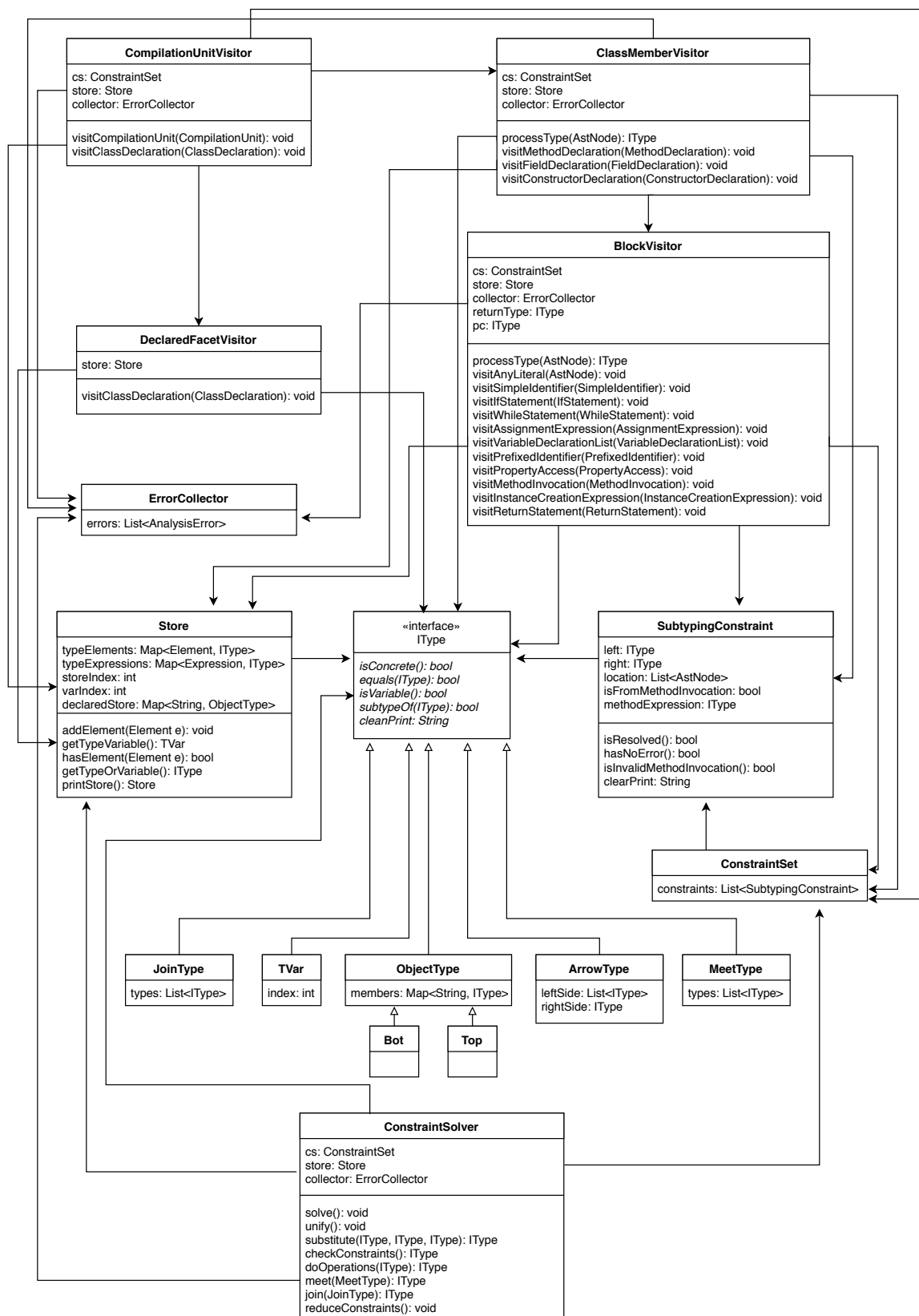


Figura 5.1: Diagrama de clases de sistema de inferencia

5.3. 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 [5].

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 5.2 muestra el diagrama de la secuencia de operaciones.

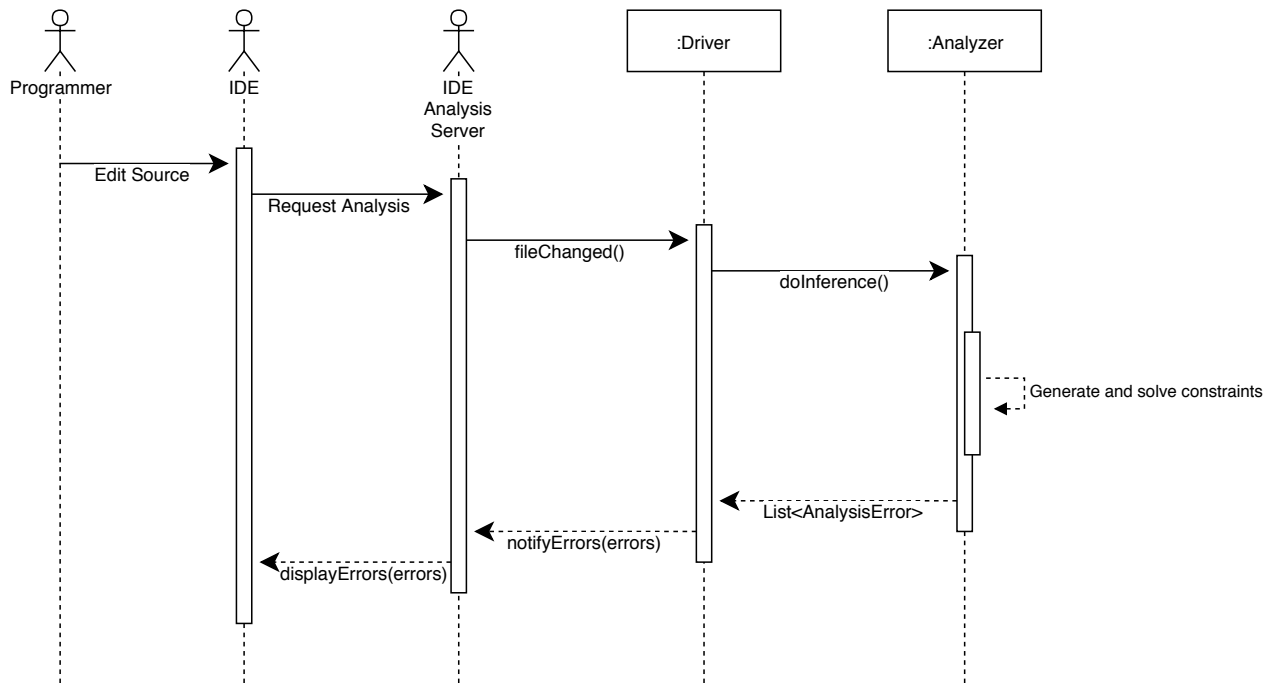


Figura 5.2: Diagrama de secuencia del plugin

5.3.1. 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 raíz 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 6

Validación

6.1. Programando con facetas públicas

Poseer un sistema de inferencia en conjunto con un plugin para IDEs tiene múltiples beneficios. Primero, evita la anotación de facetas públicas en identificadores no relevantes para el análisis. Segundo, provee comodidad al momento de programar, ya que se obtienen en tiempo real los resultados de la inferencia, sin tener que re-ejecutar un análisis. Por último, aumenta la escalabilidad de los sistemas, ya que el costo de refactorizar código anotado es mayor al costo de refactorizar código no anotado.

A continuación, se muestran algunas capturas de pantalla que ejemplifican la interacción entre el plugin y el programador.

La figura 6.1 ilustra la integración de los errores del plugin con los errores de Dart, en la misma ventana del IDE.

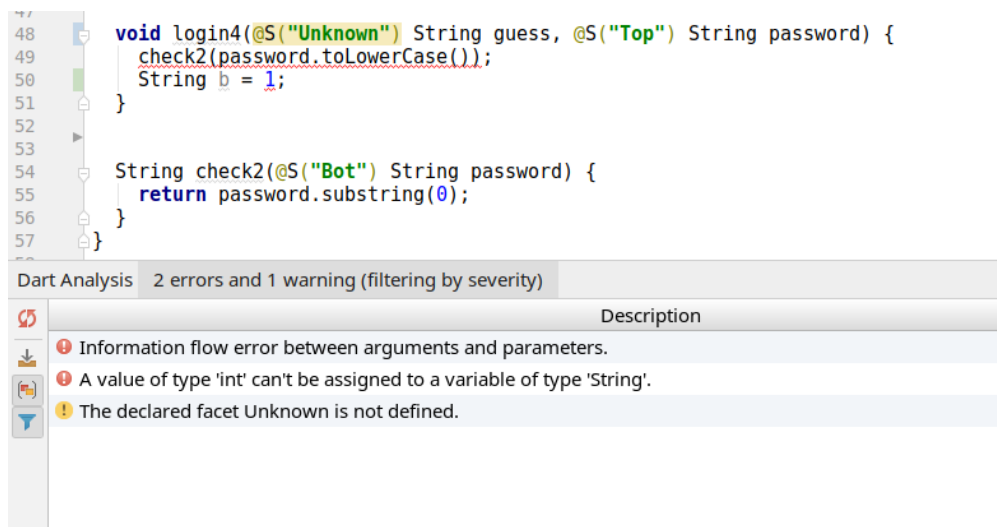
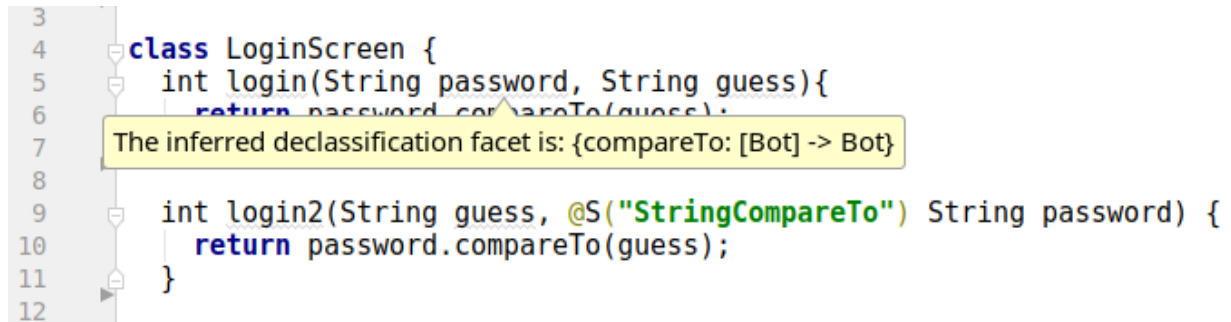


Figura 6.1: Errores integrados con errores de Dart

La figura 6.2 muestra la información inferida al ubicar el cursor sobre un identificador sin faceta pública declarada.

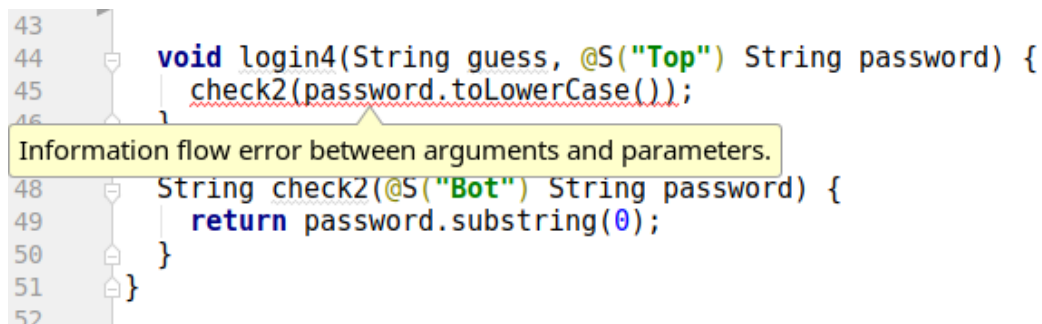


```
3
4 class LoginScreen {
5     int login(String password, String guess){
6         return password.compareTo(guess);
7     }
8
9     int login2(String guess, @S("StringCompareTo") String password) {
10        return password.compareTo(guess);
11    }
12 }
```

The inferred declassification facet is: {compareTo: [Bot] -> Bot}

Figura 6.2: Faceta pública inferida

La figura 6.3 muestra un error generado por un flujo no permitido.

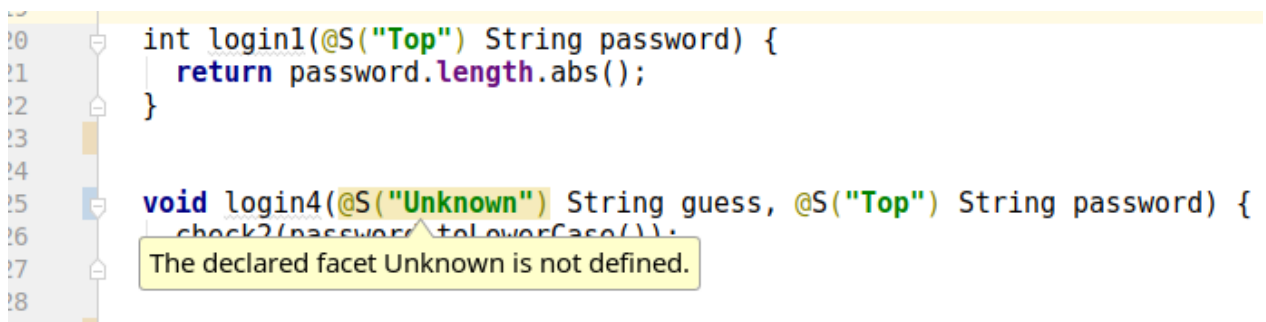


```
43
44 void login4(String guess, @S("Top") String password) {
45     check2(password.toLowerCase());
46 }
47
48 String check2(@S("Bot") String password) {
49     return password.substring(0);
50 }
51
52 }
```

Information flow error between arguments and parameters.

Figura 6.3: Error de flujo de información

La figura 6.4 muestra un error generado por la declaración de una faceta pública no definida.

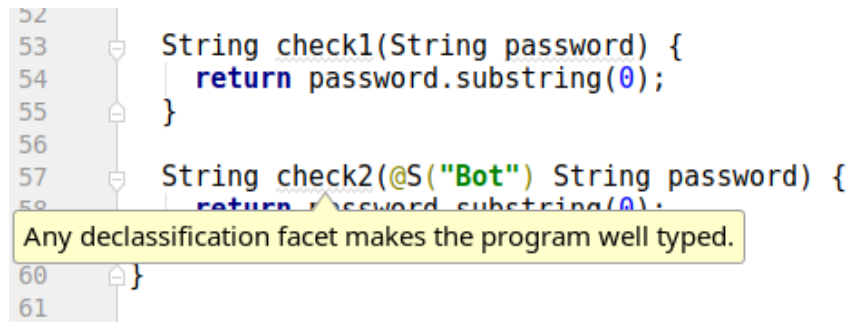


```
20 int login1(@S("Top") String password) {
21     return password.length.abs();
22 }
23
24
25 void login4(@S("Unknown") String guess, @S("Top") String password) {
26     check2(password.toLowerCase());
27 }
28
```

The declared facet Unknown is not defined.

Figura 6.4: Warning ante faceta pública no definida

La figura 6.5 muestra un identificador cuya faceta pública no pudo ser inferida con precisión, debido a que cualquier faceta genera un programa bien tipado.



```
52
53 String check1(String password) {
54     return password.substring(0);
55 }
56
57 String check2(@S("Bot") String password) {
58     return password.substring(0);
59 }
60
61
```

Any declassification facet makes the program well typed.

Figura 6.5: Toda faceta pública genera un programa bien tipado

6.2. Batería de tests

En conjunto con uno de los autores de type-based declassification, se validaron una serie de tests unitarios que ponen a prueba el cumplimiento de las reglas del sistema de tipos.

Además de proveer una forma de validación de este trabajo, los tests unitarios sirvieron como guía para la implementación de las distintas componentes.

Para escribir los tests se utilizó la librería estándar de testing en Dart.

6.3. Repositorio de prueba

Para demostrar el uso del análisis en un ejemplo realista, se creó un pequeño sistema de login web, programado con facetas públicas de type-based declassification. El código se encuentra en un repositorio de GitHub [1].

En este ejemplo se pudo constatar la utilidad del sistema de inferencia. De las 20 declaraciones de identificadores del código, solo 6 de ellas fueron anotadas con facetas públicas, y el resto fue inferido de acuerdo a las reglas del sistema de tipos.

Conclusión

Type-based declassification muestra una conexión entre las relaciones de subtyping de un lenguaje orientado a objetos, y las relaciones de orden que conforman los tipos de seguridad, para proponer un sistema de tipos que cumple una versión relajada de noninterference. Con esta propuesta, Cruz *et al.* abordan parcialmente el desafío de integrar los modelos de control de flujo de información con infraestructuras existentes. Este trabajo materializa aquella propuesta, con una implementación para un subconjunto del lenguaje Dart, en conjunto con un sistema de inferencia y un plugin para editores.

A pesar del foco de seguridad que tiene un trabajo de estas características, la formulación del problema de inferencia y el uso del plugin para integrar los resultados fueron concebidos teniendo al programador en mente, para facilitarle el trabajo y mejorar su experiencia programando, entre otros beneficios. Esta experiencia puede mejorar aún más, agregando nuevas características al plugin.

Trabajo futuro

Formalización de inferencia. En este trabajo se implmentó un sistema de inferencia sin demostrar que la extensión necesaria al sistema de tipos de type-based declassification preserva las propiedades del trabajo original, como son *type safety* y relaxed noninterference. En este sentido, es deseable la formalización de la inferencia de tipos de dos facetas, antes de realizar cualquier extensión a este trabajo, cuyo objetivo fue demostrar el uso práctico del enfoque propuesto por Cruz *et al.*

Extensión al subconjunto soportado. Este trabajo soporta un subconjunto pequeño del lenguaje Dart, lo que no permite probarlo en aplicaciones reales de mayor envergadura. Soportar características avanzadas del lenguaje, e implementar la herramienta en otros lenguajes de programación, permitiría posicionar a la herramienta como una alternativa competente de análisis de control de flujo para aplicaciones en producción.

Características del plugin. El plugin implementado en este trabajo solo muestra los resultados de la inferencia, pero no permite al programador tomar acciones automáticas al respecto. Por ejemplo, es posible asistir al usuario en la definición de una faceta pública que

ha sido declarada, navegar al lugar donde se define una faceta pública al ubicarse en la faceta declarada, definir y declarar una faceta pública basándose en el resultado de la inferencia, entre otros.

Bibliografía

- [1] Matías Meneses C. Secure login screen, programmed with type-based security types. <https://github.com/matiasimc/secure-login-test>.
- [2] Luca Cardelli. A semantics of multiple inheritance. In *Information and Computation*, pages 51–67. Springer-Verlag, 1988.
- [3] 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.
- [4] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.
- [5] Dart. Analyzer plugin: A framework for building plugins for the analysis server. https://github.com/dart-lang/sdk/tree/master/pkg/analyzer_plugin.
- [6] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.
- [7] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 158–170. ACM, 2005.
- [8] 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.
- [9] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006.
- [10] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, January 1999.

- [11] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *SIGPLAN Not.*, 36(3):41–53, January 2001.
- [12] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.
- [13] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
- [14] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 137–196, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [15] 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.