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 [3].

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

Los lenguajes de seguridad formalizan la protección de confidencialidad mediante una propiedad de no-interferencia [2], la cual puede ser muy restrictiva para aplicaciones reales y prácticas. Es por ello que los lenguajes de seguridad ofrecen mecanismos para declasificar 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 declasificación 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 declasificación 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 declasificación.

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 [5], lo que motiva una

proposición similar para *type-based declassification*.

Las herramientas de análisis estático que no son parte del análisis en tiempo de compilación de un lenguaje de programación, son integradas a los entornos de desarrollo integrado (IDE) mediante plugins, lo que permite al usuario interactuar con estas herramientas en tiempo real.

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 declasificación 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

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

2.1. Protección de confidencialidad en computación

2.1.1. No-interferencia

Los sistemas seguros utilizan el concepto de no-interferencia (*noninterference*) [2] para referirse al cumplimiento de políticas de seguridad.

Si consideramos que un sistema maneja dos niveles de seguridad, *low* o de baja confidencialidad, y *high* o de alta confidencialidad, tenemos la siguiente definición de noninterference:

Definición 2.1 (Noninterference) *Un sistema cumple con la propiedad de noninterference si y solo las acciones realizadas por entes high no tienen efecto en lo que entes low pueden ver.*

2.1.2. Language-based security

Los sistemas computacionales abordan la protección de confidencialidad con diversas técnicas, tales como criptografía y control de acceso. Ninguna de estas técnicas garantiza efectivamente la protección de confidencialidad. En efecto, los esquemas de encriptación tienen que ser actualizados constantemente debido al descubrimiento de vulnerabilidades, y control de acceso no restringe la propagación de información [3].

Existe un conjunto de técnicas más efectivas para fortalecer la seguridad de un sistema,

utilizando propiedades de los lenguajes de programación. A este conjunto se le denomina *Language-based security* (LBS).

2.1.3. Tipado de seguridad y control de flujo

Una de las técnicas más efectivas de LBS con análisis estático es *tipado de seguridad* en un *lenguaje de seguridad*. En un lenguaje de seguridad, los valores y los tipos son anotados con niveles de seguridad para clasificar la información que el programa manipula. Dichos niveles de seguridad forman una *lattice*[1].

Figura 2.1: *lattice* de dos niveles de seguridad

Por ejemplo con la *lattice* de dos niveles de seguridad $L \sqsubseteq H$, se puede distinguir entre valores enteros públicos o de baja confidencialidad (Int_L) y valores enteros privados o de alta confidencialidad (Int_H). El sistema de tipos usa estos niveles de seguridad para prevenir que la información confidencial no fluya directa o indirectamente hacia canales públicos [7], técnica que en general se denomina *information flow control*.

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

En el código anterior, los tipos de los parámetros y el tipo de retorno de la función están etiquetados con niveles de seguridad.

Flujo explicito

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

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

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

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

Flujo implicito

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

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

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

2.1.4. Declasificación

Noninterference se considera una propiedad muy estricta, debido a que aplicaciones reales y prácticas la vulneran fácilmente. En efecto, el programa de `login` es un buen ejemplo.

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

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

Sin embargo, deseamos que el programa anterior sea aceptado a pesar de violar noninterference, pues de otra forma no tendríamos cómo realizar la autenticación. Para solucionar este problema, los lenguajes de seguridad adicionan mecanismos para *declasificar* la información confidencial, implementados de diferentes formas [6]. Una de ellas, por ejemplo en Jif (un lenguaje de seguridad) [4] es usar un operador `declassify`, como se indica en el siguiente

ejemplo, declasificando la comparación de igualdad del parámetro confidencial `password` con el parámetro público `guess`

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

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

2.2. Type-based declassification

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

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 declasificación sobre los valores de dicho tipo. Por ejemplo, el tipo StringEq $\triangleq$ [eq : String $\rightarrow$ Bool] autoriza la operación `eq` sobre un String. Entonces se puede usar el tipo de dos facetas String $<$ StringEq, en donde String es un subtipo de StringEq, para controlar la operación de declasificación de la igualdad sobre `password`.

```
String<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 declasificación es parte de la jerarquía de tipos, la que forma una *lattice* como se muestra en la figura 2.2. 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*.

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*.

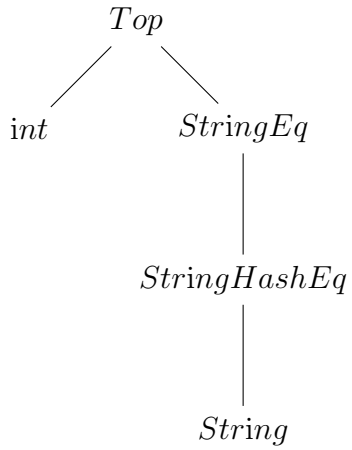

Figura 2.2: *lattice* de tipos de dos facetas

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

Existen dos reglas principales para comprobar que un programa con facetas de declasificación 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 el tipo StringHashEq $\triangleq$ [hash : String<String $\rightarrow$ String<StringEq] y llamamos al método hash sobre un valor que declara la faceta pública StringHashEq, la faceta de retorno de esa llamada será String<StringEq. A esta regla se le llama `TmD`.

La segunda regla expresa que la llamada a un método sobre un valor cuya faceta pública no autoriza la operación, retorna `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 declasificación, 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 declasificación.

2.3. Inferencia de tipos

2.3.1. 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`.

2.3.2. Variables de tipo

Los lenguajes de programación que tienen un mecanismo de inferencia de tipos, expresan sus tipos mediante *type schemes*, los cuales pueden incluir 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`.

2.3.3. 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. Veamos el siguiente ejemplo, suponiendo que el tipo `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`.

2.3.4. 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 $C : \{\beta <: \alpha, int <: \beta, float <: \beta\}$, 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 $S : \{\beta \to num, \alpha \to num\}$.

2.3.5. Decidibilidad

Los algoritmos de unificación no siempre son capaces de encontrar un tipo concreto para cada variable de tipo. Esto puede ser debido a alguna infracción de las reglas del sistema de tipos, o a la falta de tipos declarados suficientes para que el algoritmo de una respuesta correcta.

En efecto, es conocido que el problema de la inferencia completa de tipos es no decidible [8]. Veamos el siguiente ejemplo:

```
foo(x) {
  return x;
}
```

En este caso, no hay forma de saber cuál es el tipo de la variable `x` y del retorno del método `foo`. Lo único que se sabe es que tienen el mismo tipo.

Los lenguajes de programación manejan la no-decidibilidad de distintas formas. Por ejemplo, en Scala se exige que los tipos de los parámetros sean declarados, y también los tipos de retorno de métodos recursivos.

Otros lenguajes introducen cuantificadores universales para tipar las expresiones. En el ejemplo, se puede decir que el tipo de `x` y el tipo de retorno de `foo` es `t`, $\forall t$.

2.3.6. Inferencia de tipos de seguridad

Como vimos en la sección 2.1.3, 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 efecto, Pottier *et al.* [5] estudiaron un análisis de control de flujo basado en tipos para un lenguaje con referencias, excepciones y polimorfismo, en donde el sistema de tipos es basado en constraints y tiene 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 declasificación, que incluya el análisis de *Type-based Relaxed Noninterference*, mediante la realización de un plugin para entornos de desarrollo integrado (IDE). En este capítulo se detallan los problemas de inferencia a resolver, las estrategias utilizadas para resolverlos, los cambios al trabajo original y el subconjunto del lenguaje soportado.

3.1. Problema de inferencia

Se discute el problema de inferencia a resolver y ejemplos de lo que se busca del sistema a implementar.

3.2. Consideraciones de diseño

Se discuten las alternativas disponibles respecto a la decisión sobre las facetas de métodos que pertenecen al core del lenguaje, y por qué Bot ->Bot es la escogida. Explicar que de todas formas sería un parámetro configurable de la herramienta.

3.3. Generación de constraints de subtyping

Se muestra en palabras la generación de constraints para las distintas expresiones.

3.3.1. Declaración de métodos

3.3.2. Llamadas a métodos

Retorno

Argumentos

Encadenamiento de llamados

3.3.3. Expresión de retorno

3.3.4. Declaración y asignación a variables

3.3.5. Expresiones condicionales

3.4. Resolución de constraints

3.4.1. Simplificación y eliminación de constraints

3.4.2. Agrupación de constraints

Join

Meet

3.4.3. Unificación y substitución

3.5. Extensión de propuesta teórica

Mostrar los cambios y extensiones necesarias al trabajo de type-based declassification para ajustarse a un lenguaje como Dart, y el subconjunto de Dart soportado.

3.6. Interacción con el usuario

Cuál es la interacción con el usuario deseada.

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 los editores de texto más populares que integra el resultado de la inferencia.

4.1. Implementación de sistema de inferencia

4.1.1. Diagrama de componentes y descripción general

Se explica el funcionamiento general de la inferencia.

4.1.2. Dart Analyzer

Explicar funcionalidades utilizadas de la librería Dart Analyzer. Información contenida en AST, en especial inferencia de tipos. Limitaciones (si es que tiene alguna relevante).

4.1.3. Representación de tipos

Cómo se representaron los distintos tipos (type variables, arrow types, object types, Top, Bot, OrType).

4.1.4. Representación de constraints

En qué consisten las constraint de subtyping implementadas.

4.1.5. Representación de facetas de declasificación

Uso de anotaciones para indicar las facetas. Abstract classes de Dart en archivo sec.dart para declararlas. Mencionar el parsing de facetas declaradas a object types.

4.1.6. Almacenamiento de información relevante

Uso de diccionarios para llevar registro del tipo de ciertos elementos o expresiones.

4.1.7. Fase de generación de constraints

Uso del patrón visitor para recorrer el AST.

Recolección de errores

4.1.8. Fase de resolución de constraints

Recolección de errores

4.1.9. Testing

Cómo se testea la inferencia.

4.2. Implementación de plugin

4.2.1. Diagrama de componentes y descripción general

Explicar funcionamiento general e integración con sistema de inferencia.

4.2.2. Configuración inicial

Uso de la herramienta y creación del archivo sec.dart en primera ejecución del análisis.

4.2.3. Tipos de errores e información

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

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

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

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

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

[7] 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.

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