情報流解析のためのJavaアノテーション

Java Annotations for Information Flow Analysis

吉田 真也* 桑原 寛明† 國枝 義敏‡

Summary. In this paper, we propose Java annotations for information flow analysis of Java programs. Information flow analysis is useful to detect invalid information leaks. Since information flow analysis requires a lattice of secrecy and the secrecy of each data in programs, we define annotations to describe a secrecy lattice and give a secrecy to each data. Additionally, we implement information flow analysis for annotated Java programs to Java compiler of OpenJDK.

1 はじめに

個人情報などの機密情報を扱うプログラムは，機密情報を外部に漏洩しないことが求められる．機密情報の流出の可能性をプログラムの静的解析によって検出する手法として，型検査に基づく情報流解析が提案されている [1] [2] [3] [4]．型検査に基づく情報流解析では，プログラム中のデータの機密度を型とみなし，プログラムが非干渉性を満たすことを検査する型システムを構築する．非干渉性とは機密度の低いデータが機密度の高いデータに依存しないことを保証する性質である．不正な情報流が存在するとき，型エラーとして検出される．

Java 言語にはアノテーションと呼ばれるプログラム要素を注釈する機能がある．Java SE 8 で型を注釈する型アノテーションが導入された [5]．アノテーションを用いる様々な静的検査が提案されている [6] [7] [8]．例えば，CheckerFramework [7] [8] は型アノテーションを用いて静的検査を行うツールである．

本稿では，Javaプログラムに対する情報流解析を実現するためのアノテーションを提案し，プログラムのコンパイル時にアノテーションに基づいて情報流解析を行う検査器を実装する．Javaアノテーションに基づく情報流解析の実現可能性を検討するために十分な構文要素を含む Java 言語のサブセットを対象とする．情報流解析のためのアノテーションとして，機密度束を定義するためのアノテーションと，プログラムの要素に機密度を与えるためのアノテーションを定義する．後者のアノテーションは型アノテーションである．アノテーションを用いることで，構文を拡張せずに情報流解析に必要な機密度を記述できる．構文を拡張しないため既存の開発ツールを利用でき，構文を拡張する手段と比較して実用に際してのコストは低くなると期待できる．

本稿の構成は以下のとおりである．2 章で情報流解析と対象とする言語について述べる．3 章では Java 言語のアノテーションについて述べる．4 章で情報流解析のための Javaアノテーションを提案する．5 章で検査器の実装について説明し，簡単な適用例を示す．6 章で関連研究を挙げ，7 章でまとめる．

2 情報流解析

情報流解析では，データの機密度に基づき不正な情報流の有無を検査する．低い機密度のデータが高い機密度のデータに直接あるいは間接的に依存するときに不正な情報流が存在するとみなす．機密度は機密度束 $(\mathcal{H}, \sqsubseteq)$ の元であるとする [9]．機密

*Shinya Yoshida, 立命館大学 大学院 情報理工学研究科

†Hiroaki Kuwabara, 南山大学 情報センター

‡Yoshitoshi Kunieda, 立命館大学 情報理工学部

$$
\begin{array}{rcl}
T & ::= & \mathrm{Bool} \mid \mathrm{Unit} \mid \mathrm{Null} \mid C \\
\tau & ::= & (T,\ \eta) \\
CL & ::= & \mathrm{class}\ C\ \eta\ \mathrm{extends}\ C\ \{\overline{\tau\ f};\ \overline{M}\} \\
M & ::= & \tau\ m(\overline{\tau\ x})\ \eta\ \mathrm{throws}\ \overline{E}\ B \\
B & ::= & \{\overline{\tau\ x};\ \overline{S};\} \\
S & ::= & x\ =\ e \mid e.f\ =\ e \mid x\ =\ \mathrm{new}\ C \mid x =\ e.m(\overline{e}) \mid \mathrm{throw\ new}\ E \\
 & \mid & \mathrm{if}\ (e)\ B\ \mathrm{else}\ B \mid \mathrm{try}\ B\ \mathrm{catch}(E\ x)\ B\ ...\mathrm{catch}(E\ x)\ B \\
e & ::= & x \mid \mathrm{this} \mid \mathrm{null} \mid \mathrm{true} \mid \mathrm{false} \mid \mathrm{it} \mid e.f \mid e\ ==\ e \mid (C)e \mid e\ \mathrm{is}\ C
\end{array}
$$

図 1　対象とする言語の構文

度 η_L, η_H について，$\eta_L \sqsubseteq \eta_H$ のとき η_L は η_H と同じかそれより低い機密度である．

型検査に基づく情報流解析では，データの型として機密度を与え，プログラムが非干渉性を満たすことを検査する型システムを構築する．コンパイラは型システムに基づき型検査することで，不正な情報流を発見できる．

以下では，[4] に基づき例外処理付きオブジェクト指向言語を対象とする情報流解析を概説する．対象とする言語の構文を図 1 に示す．図中の $\overline{A}$ は長さが 0 以上の有限リストを表す．τ は型情報であり，データの値型 T と機密度 η の組である．機密度束 $(\mathcal{H}, \sqsubseteq)$ の存在を仮定し，機密度 η はその元を指す．CL はクラス定義であり，0 個以上のフィールド宣言と 0 個以上のメソッド宣言からなる．CL の η はクラスの機密度を表す．M はメソッド宣言であり，戻り値の型とメソッド名 m，仮引数に加えて，メソッドのヒープエフェクト η，投げられる例外のリスト $\overline{E}$ が指定される．投げられる例外が無い場合は throws $\overline{E}$ を省略できる．ヒープエフェクトはインスタンス生成やフィールド代入によって変更されるヒープ上のデータの機密度である．対象とする言語ではインスタンスはヒープ上に生成される．B はブロックを表し，0 個以上のローカル変数宣言と 0 個以上の文からなる．S は文であり，e は式である．情報流解析のために，ローカル変数，仮引数，フィールド，メソッドの戻り値それぞれの機密度，メソッドのヒープエフェクト，クラスの機密度をそれぞれの宣言において指定できる．機密度の指定が不要な式などは Java 言語に比べて簡略化されている．

クラス定義，メソッド定義，文の機密度に関する型付け規則を図 2 に示す．この型付け規則はプログラムが非干渉性を満たすかを検査できる．CDEC はクラス定義，MDEC はメソッド定義の型付け規則であり，それ以外は文の型付け規則である．$level(C)$ はクラス C の機密度を返す関数であり，$smtype(C)$ はクラス C のメソッドのシグネチャを返す関数である．$sfields(C)$ はクラス C のフィールド宣言の集合を返す関数である．$result$ はメソッドの戻り値を表す．式の型付け規則は [2] を参照されたい．直感的には，機密度の高い値を機密度の低い変数に代入できないことと，レシーバの機密度より低いフィールドに影響を与えてはいけないことを表す．

文の型判定式 $\Delta \vdash S : (\eta_s, \eta_h, P)$ は型環境 Δ のもとで，以下を満たすことを表す．

- S で代入される変数やパラメータの機密度は η_s 以上
- S によるヒープエフェクトは η_h 以上
- S で投げられる例外の集合が P
- S 内の情報流が安全である

(η_s, η_h, P) が文の型である．型環境 Δ は変数名から変数の型 τ を返す関数である．S 内の情報流が安全であるとは，x から y への情報流が存在するとき，x の機密度 η_x と y の機密度 η_y は $\eta_x \sqsubseteq \eta_y$ を満たすことを指す．

$$\frac{\mathit{level}(D) \sqsubseteq \eta \quad C\ \eta \text{ extends } D \vdash M \text{ for each } M \in \overline{M} \quad \mathit{level}(D) \neq \eta \Rightarrow \text{ every } m \text{ with } \mathit{smtype}(m, D) \text{ defined is overridden in } C}{\vdash C\ \eta \text{ extends } D\ \{\overline{\tau\ f};\ \overline{M}\}}\ [\text{CDEC}]$$

$$\frac{\overline{x : (T_x, \eta_x)},\ \mathit{this} : (C, \eta_c),\ \mathit{result} : (T_r, \eta_r) \vdash B : (\eta_s, \eta_h', \overline{E}) \quad \mathit{smtype}(m, D) \text{ is defined } \Rightarrow \mathit{smtype}(m, D) = \overline{x : (T_x, \eta_x)} \xrightarrow{\eta_h} (T_r, \eta_r); \overline{E} \quad \eta_h \sqsubseteq \eta_h'}{C\ \eta_c \text{ extends } D \vdash (T_r, \eta_r)\ m(\overline{(T_x, \eta_x)\ x})\ \eta_h \text{ throws } \overline{E}\ B}\ [\text{MDEC}]$$

$$\frac{\Delta,\ \overline{x : (T_x, \eta_x)} \vdash S_i : (\eta_{s_i}, \eta_{h_i}, P_i) \quad \eta_s \sqsubseteq \eta_{s_i} \quad \eta_h \sqsubseteq \eta_{h_i} \quad \forall E \in P_i.\ \forall j > i.\ \mathit{level}(E) \sqsubseteq \eta_{s_j}, \mathit{level}(E) \sqsubseteq \eta_{h_j} \quad i \in \{1, \ldots, n\} \quad P = \bigcup_i P_i}{\Delta \vdash \{\overline{(T_x, \eta_x)\ x};\ S_1; \ldots S_n;\} : (\eta_s, \eta_h, P)}\ [\text{BLOCK}]$$

$$\frac{\Delta \vdash x : (T_x, \eta_x) \quad \Delta \vdash e : (T_e, \eta_e) \quad \eta_e \sqsubseteq \eta_x \quad \eta_s \sqsubseteq \eta_x}{\Delta \vdash x = e : (\eta_s, \eta_h, \emptyset)}\ [\text{ASSIGN1}]$$

$$\frac{\Delta \vdash e_1 : (T_1, \eta_1) \quad \Delta \vdash e_2 : (T_2, \eta_2) \quad (T_f, \eta_f)\ f \in \mathit{sfields}(T_1) \quad \eta_1 \sqsubseteq \eta_f \quad \eta_2 \sqsubseteq \eta_f \quad \eta_h \sqsubseteq \eta_f}{\Delta \vdash e_1.f = e_2 : (\eta_s, \eta_h, \emptyset)}\ [\text{ASSIGN2}]$$

$$\frac{\Delta \vdash x : (T_x, \eta_x) \quad \mathit{level}(C) \sqsubseteq \eta_x \quad \eta_s \sqsubseteq \eta_x \quad \eta_h \sqsubseteq \mathit{level}(C)}{\Delta \vdash x = \text{new } C : (\eta_s, \eta_h, \emptyset)}\ [\text{NEW}]$$

$$\frac{\Delta \vdash x : (T_x, \eta_x) \quad \Delta \vdash e : (T_e, \eta_e) \quad \Delta \vdash e_i : (T_{e_i}, \eta_{e_i}) \quad y_1 : (T_{y_1}, \eta_{y_1}), \ldots, y_n : (T_{y_n}, \eta_{y_n}) \xrightarrow{\eta_h'} (T_r, \eta_r); E_1, \ldots, E_k = \mathit{smtype}(m, T_e) \quad \forall i \in \{1, \ldots, n\}.\ \eta_{e_i} \sqsubseteq \eta_{y_i} \quad \eta_e \sqsubseteq \eta_x \quad \eta_r \sqsubseteq \eta_x \quad \eta_s \sqsubseteq \eta_x \quad \eta_e \sqsubseteq \eta_h' \quad \eta_h \sqsubseteq \eta_h' \quad P = \bigcup_i E_i \quad \forall E \in P.\ \eta_x \sqsubseteq \mathit{level}(E)}{\Delta \vdash x = e.m(e_1, \ldots, e_n) : (\eta_s, \eta_h, P)}\ [\text{CALL}]$$

$$\frac{\eta_s \sqsubseteq \mathit{level}(E) \quad \eta_h \sqsubseteq \mathit{level}(E)}{\Delta \vdash \text{throw new } E : (\eta_s, \eta_h, \{E\})}\ [\text{THROW}]$$

$$\frac{\Delta \vdash e : (\mathit{Bool}, \eta_e) \quad \Delta \vdash B : (\eta_1, \eta_2, P) \quad \Delta \vdash B' : (\eta_1', \eta_2', P') \quad \eta_e \sqsubseteq \eta_s \quad \eta_e \sqsubseteq \eta_h \quad \eta_s \sqsubseteq \eta_1 \quad \eta_s \sqsubseteq \eta_1' \quad \eta_h \sqsubseteq \eta_2 \quad \eta_h \sqsubseteq \eta_2'}{\Delta \vdash \text{if } (e)\ B \text{ else } B' : (\eta_s, \eta_h, P \cup P')}\ [\text{IF}]$$

$$\frac{\Delta \vdash B : (\eta_s', \eta_h', P') \quad \Delta, x : (E_i, \mathit{level}(E_i)) \vdash B_i : (\eta_{s_i}, \eta_{h_i}, P_i) \quad i \neq j \Rightarrow E_i \neq E_j \quad \eta_s \sqsubseteq \eta_s' \quad \eta_s \sqsubseteq \eta_{s_i} \quad \eta_h \sqsubseteq \eta_h' \quad \eta_h \sqsubseteq \eta_{h_i} \quad \mathit{level}(E_i) \sqsubseteq \eta_{s_i} \quad \mathit{level}(E_i) \sqsubseteq \eta_{h_i} \quad i \in \{1, \ldots, n\} \quad P = (P' - \bigcup_i E_i) \cup \bigcup_i P_i}{\Delta \vdash \text{try } B\ \text{catch}(E_1\ x_1)\ B_1\ \ldots \text{catch}(E_n\ x_n)\ B_n : (\eta_s, \eta_h, P)}\ [\text{CATCH}]$$

図 2　対象言語の型付け規則

3　アノテーション

Java 言語にはプログラム内の宣言や型に対し，ツールやライブラリのための情報を注釈する言語機能としてアノテーションがある．アノテーションはコードの自動生成や静的検査，依存性の注入など様々な目的で用いられる．Java SE 8 では，型

に対して情報を注釈する型アノテーションと呼ばれるアノテーションが導入された．アノテーションの引数は名前と値の組の列であり，引数の値としてリテラル，列挙型の定数，クラスオブジェクト，アノテーション及びそれらの一次元配列を取ることができる．引数の名前が省略された場合は暗黙に value を指定したとみなされる．引数を取らないアノテーションはマーカアノテーションと呼ばれる．

表 1　ElementType の値と意味

値	意味
TYPE	クラスやインターフェースなどの型の宣言
FIELD	フィールド宣言
CONSTRUCTOR	コンストラクタ宣言
METHOD	メソッド宣言
PARAMETER	引数宣言
LOCAL_VARIABLE	ローカル変数宣言
PACKAGE	パッケージ宣言
TYPE_PARAMETER	型引数
TYPE_USE	型使用

アノテーションは`@interface`キーワードを用いて定義される．アノテーションの引数はメソッド宣言文を用いて定義される．メソッドの戻り値の型が引数の型とみなされ，メソッド名が引数名とみなされる．アノテーションの定義を注釈するメタアノテーションとして Target アノテーションがある．Target アノテーションにより，定義されるアノテーションを記述できる箇所を列挙型 ElementType を用いて指定する．表 1 に ElementType の値とそれぞれの意味を示す．TYPE_PARAMETER か TYPE_USE が指定されるアノテーションが型アノテーションである．

任意のアノテーションによる注釈を対象として任意の処理を行うための仕組みとして Pluggable Annotation Processing API(以下，PAP) が提供されている．PAP を利用するためには javax.annotation.processing.Processor を実装してアノテーションプロセッサクラスを作成し，そのクラスをコンパイラに指定する．SupportedAnnotationTypes アノテーションでクラス宣言を注釈して処理対象のアノテーションを指定する．図 3 に Override アノテーションによる注釈を対象として処理を行うアノテーションプロセッサを示す．コンパイラはパース処理とシンボルテーブルへの名前登録処理が終了した後，指定されたアノテーションを発見すると，アノテーションプロセッサクラスの process メソッドを実行する．コンパイラは全てのアノテーションプロセッシングが終了した後，通常のコンパイル処理を行う．

```
@SupportedAnnotationTypes("java.lang.Override")
@SupportedSourceVersion(RELEASE_8)
public class MyAnnotationProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment roundEnv) {
        /* 任意の処理 */
        return true;
    }
}
```

図 3　Override アノテーションを対象とするアノテーションプロセッサクラス

4　情報流解析のための Java アノテーション

4.1　アノテーションの定義

Java プログラムを対象とする情報流解析を実現するには，Java プログラム内に機密度束の定義と，データの型としての機密度，メソッドのヒープエフェクト，クラスの機密度を記述できる必要がある．一般にプログラムによって前提とする機密度束は異なるため，任意の機密度束を定義するための仕組みを用意する．データの型としての機密度は，変数や引数，メソッドの戻り値の値型と共に機密度を指定することで記述する．

機密度束を定義するためには，束の元である機密度を列挙し機密度間の順序関係を記述できればよいため，各機密度を定数として持つ列挙型として機密度束を定義する．列挙型は定数を定義するための機能であり，各定数が各機密度に対応する．機密度を列挙型の定数として定義することで，指定される機密度が機密度束の元であるかをコンパイラが検査できる．機密度束を表す列挙型を他の列挙型と区別するために，列挙型が機密度束であることを表すアノテーションとして SecrecyLattice アノテーションを定義する．機密度間の順序関係については，各機密度に対して最小上界を指定できればよいため，そのためのアノテーションとして列挙型の定数を注釈する Top アノテーションと LowerOf アノテーションを定義する．Top アノテーションは注釈される定数が機密度束の最大元，つまり最も高い機密度であることを示す．列挙型の先頭の定数に対して Top アノテーションを付与する．LowerOf アノテーションは機密度間の順序を指定するアノテーションであり，注釈される機密度の最小上界である機密度の集合が引数として与えられる．

機密度の指定にもアノテーションを利用する．データの型としての機密度は型アノテーション，メソッドのヒープエフェクトはメソッド宣言に対するアノテーション，クラスの機密度はクラス宣言に対するアノテーションを用いて指定する．Java 言語におけるアノテーションの構文上の制約により，メソッドの戻り値の機密度を指定する型アノテーションとヒープエフェクトを指定するアノテーションを区別するには，それぞれを異なる名前のアノテーションとする必要がある．そこで，メソッドの戻り値を含むデータの型としての機密度およびクラスの機密度を指定するための Secrecy アノテーションと，メソッドのヒープエフェクトを指定するための HeapEffect アノテーションを定義する．いずれのアノテーションも指定する機密度として機密度束に対応する列挙型の定数を引数に取るようにする．Secrecy アノテーションは，変数や引数，メソッドの戻り値の型としての機密度を指定するために，これらの値型を注釈する型アノテーションとする．同時に，クラスの機密度を指定するためにクラス宣言を注釈するアノテーションともする．Secrecy アノテーションにより変数宣言 `(Bool, H) data` は`@Secrecy(H) Bool data` と書ける．HeapEffect アノテーションにより，機密度 H のヒープエフェクトが存在するメソッド m は`@HeapEffect(H) void m() {}`と定義できる．

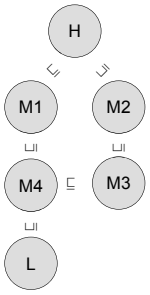

図 4　機密度束 MySecrecy のグラフ

例として，図 4 の機密度束 MySecrecy に対応する列挙型の定義を図 5 に示す．機密度は SecrecyLattice アノテーションで注釈される列挙型 MySecrecy の定数として表される．H は機密度束 MySecrecy の最大元であるため，Top アノテーションで注釈する．M1 と M2 は H が最小上界であるため，引数が H の LowerOf アノテーションで注釈する．M3 は M2 が最小上界であり，M4 は M1 と M3 が最小上界であるため，引数がそれぞれの最小上界の LowerOf アノテーションで注釈する．L は最小元であり，M4 を最小上界とするため，引数が M4 の LowerOf アノテーションで

注釈する．複数の機密度束が同一パッケージ内で定義されても，LowerOf，Secrecy，HeapEffect アノテーションの名前が衝突しないように，これらのアノテーションは列挙型内で定義する．図 5 内の SecrecyLattice アノテーションと Top アノテーションは informationflows.annotations パッケージで定義される．SecrecyLattice アノテーションと Top アノテーションの定義を図 6 と図 7 に示す．

```
package sample;
import java.lang.annotation.*;
import informationflows.annotations.*;

@SecrecyLattice
public enum MySecrecy {
    @Top H,
    @LowerOf(H) M1,
    @LowerOf(H) M2,
    @LowerOf(M2) M3,
    @LowerOf({M1, M3}) M4,
    @LowerOf(M4) L,;

    @Target(ElementType.FIELD)
    public @interface LowerOf {
        MySecrecy[] value();
    }

    @Target({ElementType.TYPE_USE, ElementType.TYPE})
    public @interface Secrecy {
        MySecrecy value();
    }

    @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
    public @interface HeapEffect {
        MySecrecy value();
    }
}
```

図 5　機密度束 MySecrecy の定義

```
package informationflows.annotations;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
public @interface SecrecyLattice {}
```

図 6　SecrecyLattice アノテーションの定義

```
package informationflows.annotations;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
public @interface Top {}
```

図 7　Top アノテーションの定義

4.2 アノテーション定義の自動生成

図 5 の LowerOf，Secrecy，HeapEffect アノテーションは引数として機密度である MySecrecy の値を取るため，value の戻り値の型は MySecrecy あるいは MySecrecy の配列となっている．任意の列挙型の値を取るジェネリックなアノテーションは定義できないため，定義される機密度束ごとに value の戻り値の型を変え，これらのアノテーションを定義する必要があり，面倒である．そこで，value の戻り値の型は対応する機密度束の列挙型名であることと，機密度束の列挙型には SecrecyLattice アノテーションが付与されることから，PAP を用いてこれらのアノテーションの定義を自動生成する．同一パッケージに複数の機密度束が定義されてもアノテーションの名前の衝突が起きないように，自動生成されるアノテーションは機密度束ごとに 1 つのクラス中に定義する．

機密度束 MySecrecy の定義と自動生成されたソースコードをそれぞれ図 8 と図 9 に示す．図 9 が自動生成されるため，機密度束を定義するには図 8 を記述するだけでよい．MySecrecyAnnotations クラス中に LowerOf，Secrecy，HeapEffect アノテーションの定義が含まれるように自動生成する．SecrecyAnnotationClass アノテーションは，注釈されるクラスが自動生成されるアノテーションを囲むクラスであることを示す．LowerOfAnnotation アノテーションはメタアノテーションであり，注釈されるアノテーションが図 5 の LowerOf アノテーションに相当する．SecrecyAnnotation，HeapEffectAnnotation アノテーションも同様である．機密度束の定義や機密度を指定するソースファイルでは機密度束に対応する SecrecyAnnotationClass アノテーションが注釈されるクラスをインポートすればよい．

```
package sample;
import java.lang.annotation.*;
import informationflows.annotations.*;
import sample.MySecrecyAnnotations.*;

@SecrecyLattice
public enum MySecrecy {
    @Top H,
    @LowerOf(H) M1,
    @LowerOf(H) M2,
    @LowerOf(M2) M3,
    @LowerOf({M1, M3}) M4,
    @LowerOf(M4) L,;
}
```

図 8　自動生成向けの MySecrecy の定義

5　検査器の実装

提案するアノテーションを用いて機密度が記述されたプログラム中に不正な情報流が存在しないか検査する検査器を OpenJDK の Java コンパイラを拡張して実装する．情報流解析を行う Java コンパイラの処理の流れを図 10 に示す．parse はパース処理，enter はシンボルテーブルへの名前登録処理，process はアノテーションプロセッサの実行処理，analyze はプログラムの意味解析処理，generate はバイトコード生成処理を表す．プログラムの意味解析が終了し，メソッドやフィールド，変数への参照が解決され，型が決定された後に情報流解析を行う．プログラムの意味解析で値型に関する全ての型エラーは発見されるため，情報流解析では機密度に関する型エラーのみを検査する．

機密度に関する型エラーの検出は制約条件集合の充足問題として実装する．制約条件集合とはプログラム要素が型付け可能となる制約条件の集合であり，[4] のアル

```
package sample;
import javax.annotation.Generated;
import java.lang.annotation.*;
import informationflows.annotations.*;

@Generated("MySecrecy")
@SecrecyAnnotationClass(clazz=MySecrecy.class, className="MySecrecy")
public final class MySecrecyAnnotations {
    private MySecrecyAnnotations() {}

    @Target(ElementType.FIELD)
    @LowerOfAnnotation public @interface LowerOf {
        MySecrecy[] value();
    }

    @Target({ElementType.TYPE_USE, ElementType.TYPE})
    @SecrecyAnnotation public @interface Secrecy {
        MySecrecy value();
    }

    @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
    @HeapEffectAnnotation public @interface HeapEffect {
        MySecrecy value();
    }
}
```

図 9　MySecrecy を元に自動生成されたソースコード

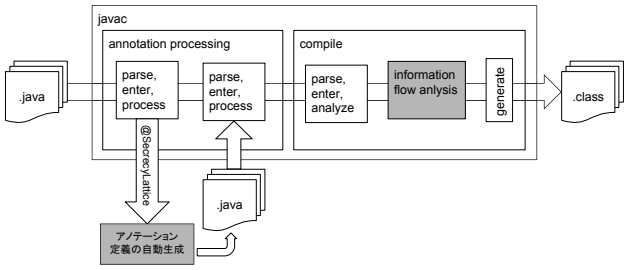

図 10　情報流解析を行う Java コンパイラの処理の流れ

ゴリズムに基づいて生成する．生成される制約条件集合が充足可能であれば型付け可能であり，充足不可能であれば型付けできず，型エラーが存在する．情報流解析を行う Java コンパイラはメソッド定義ごとに，プログラム中のアノテーションから変数やクラスの機密度やヒープエフェクトを求め，制約条件集合を生成し，その充足可能性を判定する．生成された制約条件集合の充足可能性の判定には Cream [10] を用いる．Cream は制約プログラミングのためのライブラリであり，制約条件集合の解を発見するアルゴリズムを提供する．Cream が解を発見できないとき，その制約条件集合は充足不可能であり，不正な情報流が存在するため，エラーを出力する．今回の実装は [4] [11] に基づくため，対応する構文要素はクラス宣言と例外を投げうるメソッド宣言，if 文や代入，例外の送出，インスタンス生成などに限られる．

提案するアノテーションが記述された簡単なプログラムに対して実装した検査器を適用した例を示す．図 11 は不正な情報流が存在するプログラムである．このプログラムで用いられる機密度束の定義は図 8 である．if 文の条件式の機密度は H であるが，if 文内で変更される変数の最低の機密度は L であるため，不正な情報流が存在する．情報流解析を行う Java コンパイラに図 8 と図 11 のソースファイルを指定して実行したときの実行結果を図 12 に示す．`ifjavac.jar` が情報流解析を行う Java コンパイラであり，

依存するライブラリやアノテーションの自動生成のためのアノテーションプロセッサへのクラスパスが指定されてパッケージングされている．sample/MySecrecy.java が図 8 のファイル名であり，sample/Sample.java が図 11 のファイル名である．sample/Sample.java の 7 行目で定義される illegalMethod() メソッド内に不正な情報流が存在することを示すエラーメッセージが表示される．メソッドごとに生成した制約条件集合の充足可能性を判定することで不正な情報流を発見するが，その際に位置情報を保存しないため，メソッド内に不正な情報流が存在することだけがわかる．

```
package sample;
import sample.MySecrecyAnnotations.*;
import static sample.MySecrecy.*;

@Secrecy(L) public class Sample {
    @HeapEffect(L) public void illegalMethod() {
        @Secrecy(H) boolean cond;
        @Secrecy(L) int ok;

        cond = true;
        if (cond) {
            ok = 1;
        } else {
            ok = 0;
        }
    }
}
```

図 11　不正な情報流が存在する簡単なプログラム

```
$ java -jar ifjavac.jar sample/MySecrecy.java sample/Sample.java
sample/Sample.java:7: エラー: Sample
    public void illegalMethod() {
                ^
  不正な情報流が存在します
エラー 1 個
```

図 12　コンパイル結果

6　関連研究

情報流解析可能なオブジェクト指向言語として，Myers らは JFlow [12] や Jif [13] を提案している．これらの言語は Java 言語の文法を拡張した言語であり，データの値型の後に機密情報を指定する．既存の言語の文法を拡張すると，既存の言語のための統合開発環境や静的解析ツールなどの既存の開発ツールが利用できないため，実用に問題がある．本稿の提案手法では Java 言語のアノテーションを用いてクラスや型に機密度を指定するため，Java 言語の文法を拡張せずに情報流解析を実現できる．Java 言語の文法を拡張しないため，既存の開発ツールを利用できる．

アノテーションを用いる静的検査ツールとして Papi らは CheckerFramework [7] [8] を提案している．CheckerFramework は予め定義された型アノテーションと検査器に基づいて，Java プログラムを検査する．いくつかの型アノテーションと対応する検査器は CheckerFramework に実装されている．例えば，null が代入されないことを示す NonNull アノテーションと NonNull アノテーションが注釈される変数や

引数に null が代入されないことを検査する検査器がある．本稿の提案手法では，開発者が任意に定義した機密度束に対し，プログラム要素の機密度を指定するためのアノテーションを自動生成し，そのアノテーションによる注釈に基づき情報流解析を行う．

7 おわりに

本稿では，Java プログラムを対象とする型検査に基づく情報流解析のためのアノテーションを提案した．機密度束は提案するアノテーションを用いて列挙型により定義する．機密度の指定は型アノテーションを用い，メソッドのヒープエフェクトの指定やクラスの機密度の指定にはアノテーションを用いる．記述を簡単にするために PAP によって機密度束の定義から必要なアノテーションの定義を自動生成する．不正な情報流の検査は，型付け規則に基づいて機密度に関する制約条件集合を生成し，その充足可能性を判定することで行う．検査器は OpenJDK の javac を拡張して実装した．

本稿では，Java 言語でのアノテーションを用いた情報流解析の可能性を検討するために，[4] に基づく Java 言語のサブセットを対象とした．今後の課題として，実用的なプログラムの情報流解析のために，本稿では対象としなかったジェネリクスやインターフェースなどの構文要素への対応や，ライブラリとユーザプログラムそれぞれに対する情報流解析の連携が挙げられる．規模の大きなプログラムに対して提案するアノテーションと検査器を適用し，コンパイラ拡張として実装された情報流解析のオーバヘッドを評価することは今後の課題である．

謝辞 本研究の一部は JSPS 科研費 15K00112 および 2016 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.

[2] Anindya Banerjee and David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 253–267. IEEE Computer Society Press, 2002.

[3] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[4] 翔 黒川, 寛明 桑原, 晋一郎 山本, 俊樹 坂部, 正彦 酒井, 圭一朗 草刈, and 直樹西田. 例外処理付きオブジェクト指向言語における情報流の安全性解析. **電子情報通信学会技術研究報告**. *SS,* **ソフトウェアサイエンス**, 106(324):13–18, 2006.

[5] Michael D. Ernst. Type Annotations Specification (JSR 308). 2013.

[6] Daniel Tang, Ales Plsek, and Jan Vitek. Static Checking of Safety Critical Java Annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 148–154, 2010.

[7] The Checker Framework. `http://types.cs.washington.edu/checker-framework/`.

[8] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, 2008.

[9] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976.

[10] Cream: Class Library for Constraint Programming in Java. `http://bach.istc.kobe-u.ac.jp/cream/`.

[11] 寛明 桑原. 型検査に基づく情報流解析における型エラースライシング. In **ソフトウェア工学の基礎** *XVI(FOSE2009)*, pages 49–60. 近代科学社, 2009.

[12] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, 1999.

[13] Jif: Java + information flow. `http://www.cs.cornell.edu/jif/`.