束構造をもつセキュリティクラスに基づく情報フロー解析アルゴリズム

國信 茂太[†]　高田 喜朗[†]　関 浩之[†]　井上 克郎[‡†]
[†] 奈良先端科学技術大学院大学 情報科学研究科
〒 630-0101 生駒市高山町 8916-5
[‡] 大阪大学大学院 基礎工学研究科 情報数理系専攻
〒 560-8531 豊中市待兼山町 1-3
E-mail: {shige-ku,y-takata,seki}@is.aist-nara.ac.jp, inoue@ics.es.osaka-u.ac.jp

キーワード：情報フロー解析, セキュリティクラス, 不動点計算

内容梗概

プログラムの実行によって, 機密度の高い情報が, 不特定多数のユーザにアクセスされるような記憶域に書き出されないことを保証するためには, プログラムにある機密度の情報が入力されたとき, どのような機密度の情報が出力されるか解析できればよい. 従来の研究において, 機密度 (セキュリティクラス) を束でモデル化し, それに基づいて手続き型プログラムの情報フローを静的に解析する方法が提案されているが, 解析対象となるプログラムが単純で, 特に再帰手続きを考慮していなかった. そこで, 著者らは [K00] において, 一般に再帰を含む手続き型プログラムの情報フローを静的に解析するアルゴリズムを提案し, その健全性を証明した. 本稿では, [K00] で提案したアルゴリズムが分配性を満たすことを示し, これにより解析アルゴリズムの時間計算量を改善できることを述べる.

1　まえがき

不特定多数の人が利用するシステムにおいては, 情報の漏洩を防ぐことは重要なことである. 情報の漏洩を防ぐ手段として以下のようなアクセス制御法がよく用いられる (Mandatory Access Control と呼ばれる [P94]):「各データに, そのデータの機密度を表すセキュリティクラス (SC と略記, 例えば, topsecret, confidential, unclassified) を割り当てる. データ d の SC を $SC(d)$ と表す. 同様に, ユーザ (またはプロセス) には, どの機密度のデータまでアクセスできるかを表すクリアランスを割り当てる. ユーザ u のクリアランスを $clear(u)$ と表す. $clear(u) \geq SC(d)$ のときかつそのときのみ, ユーザ u はデータ d を読むことができる.」しかし, クリアランスが $SC(d)$ 以上のユーザプログラムがデータ d を読み込み, それを, 故意にまたは過失によって, クリアランスが $SC(d)$ より小さいユーザにもアクセスできる記憶域に書き出してしまうと, 望ましくない情報の漏洩が生じる. このような情報漏洩を防ぐためには, プログラム P と, P への入力となるデータの SC が与えられたとき, P がどの記憶域にどのような SC のデータを出力するのかを解析できることが望ましい.

Denning らは [D76][DD77] において, SC を束でモデル化し, それに基づいてプログラムの情報フローを静的に解析する方法を初めて提案した. [BBM94] [O95] [VS97] などでは, [DD77] の手法を理論的に再検討し, 手続き型プログラムの情報フロー解析法の一般化を試みている. [O95] では抽象解釈を利用してモデル化を行ない, 通常のプログラムの意味論に情報フローという概念を直接追加した instrumented semantics (作為的意味論) を定義し, その意味論の下で解析法の健全性を示している. [VS97] などでは, 型推論の手法を応用して,

解析法を形式的に記述し, 健全性の証明を行なっている. [VS97] では, [O95] における健全性の定義が必ずしも自然でないことを指摘し, 解析法の健全性を次のように定義している: 「解析アルゴリズムが『解析対象のプログラムにおける手続き f の SC は τ である』と答えたと仮定する. このとき, SC が τ 以下であるような f の実引数値 (入力) がすべて等しいならば, それ以外の入力が変化しても f の実行結果は不変である.」しかし, これらの研究では, 解析対象のプログラムが単純であり, 特に, 再帰手続きを考慮していなかった. 例えば, [VS97] で解析対象としているプログラムにおいて許される手続き呼び出しは再帰を含まないものに限られている. [M99] では, Java 言語に基づき, 静的情報フロー解析と動的 (実行時) 情報フローチェックの組合せ, declassification, label morphism などの機構をもつ JFLOW という言語を提案しているが, モデルが極めて複雑で, 解析法の健全性は示されていない. [AFG99] [HR98] [SV98] では, 分散環境における情報フロー解析法が提案されている. 一方, 関数型のプログラムを対象とする研究として, [HR98] では, 純ラムダ計算に基づく単純な関数型言語における情報フロー解析法を型推論を用いて提案している. また, 以上の研究ではいずれも, 解析アルゴリズムの時間計算量の評価は行われていなかった.

著者らは, [K00] において, 一般に再帰を含む手続き型プログラムの情報フローを解析する方法を提案した. この方法において, SC の集合は任意の有限分配束によって与えることができる. この方法では, まず, 解析対象プログラム中の全ての実行文について, その文の実行によって起こる情報フローを表すような関係式を定義する. その後, それらの関係式を同時に満たす最小解を最小不動点計算によって求める.

本稿では, [K00] で提案した解析アルゴリズムが分配性を満たすことを示し, これにより解析アルゴリズムの時間計算量を改善できることを述べる. 本稿の構成は以下の通りである. まず, 2 節で解析対象となるプログラムを定義する. プログラムは, 代入文, if 文, while 文, 関数呼出し (再帰的なものを含む) の機能をもつ通常の手続き型プログラムである. 3 節では, 抽象解釈を用いて解析アルゴリズムを形式的に記述する. そして, 4 節では, 分配性の定義を述べ, このアルゴリズムが分配性を満たすことを示す. また, アルゴリズムの時間計算量について考察する.

2 諸定義

2.1 プログラムの構文

解析対象とするプログラムの構文を以下のように定義する．このプログラミング言語は C 言語に類似した手続き型言語である. プログラムの形式的意味は 2.3節で定義する．

プログラム $Prog$ は関数定義の集合である．関数定義は以下の形式で与えられる．

$$f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \{P_f\}$$

ただし，f は関数名，$x_1, \ldots, x_n$ は仮引数，$y_1, \ldots, y_m$ は局所変数，P_f は関数本体である．

P_f の構文 P は以下の BNF 式で与えられる．ただし，c は定数，x は局所変数または仮引数，f はプログラム中で定義された関数，θ は加減算などの組み込み演算を表す．

$$\begin{array}{rcl} P & ::= & C \mid C_1; P \\ C & ::= & \text{if } M \text{ then } P \text{ else } P \text{ fi} \mid \text{return } M \\ P_1 & ::= & C_1 \mid C_1; P_1 \\ C_1 & ::= & x := M \mid \text{if } M \text{ then } P_1 \text{ else } P_1 \text{ fi} \mid \text{while } M \text{ do } P_1 \text{ od} \\ M & ::= & c \mid x \mid f(M, \ldots, M) \mid \theta(M, \ldots, M) \end{array}$$

プログラム $Prog$ の実行は，$Prog$ の要素のうち関数名が $main$ である関数を評価することで行われる．この関数を main 関数と呼ぶ．$Prog$ の入力は main 関数に与える実引数，出力は main 関数の返り値である．

2.2 値とセキュリティクラス

プログラム実行時にデータとして扱われる値 (定数，変数が格納する値，実引数，関数の返り値など) の型は区別せず，すべて val 型に属すとする．

解析時には，プログラムの入力となる値にそれぞれ，その機密度を表すセキュリティクラス (SC) が与えられる．SC の集合 SCs 上には半順序 $\sqsubseteq$ が定義され，SCs は $\sqsubseteq$ に関して分配束をなすと仮定する．SCs の最小元を $\bot$，最小上界演算を $\sqcup$ と表す．直観的には，$s \sqsubseteq t$ であるとは，クリアランスが t であるユーザは SC が s である値にアクセスできることを表す．以下は最も単純な SCs の例である．

$$SCs = \{low, high\}, \quad low \sqsubseteq high.$$

解析の目的は，入力となる各値の SC がわかっていると仮定した上で，出力となる各値の SC がどのようになるか調べることである．厳密には，3.1節で述べる健全性を満たすような解析結果を求めることが目的である．

2.3 プログラムの意味

次のような型を仮定する. ただし, $\times$ で直積, $+$ で分離和を表す.

val 型 (値を表す型) プログラムに現れる n 引数基本演算子 θ に対し, n 引数関数 $\theta_{\mathcal{I}} : val \times \cdots \times val \to val$ が定義されているとする.

$store$ 型 (記憶域を表す型) 2 つの関数

$$update : store \times var \times val \to store$$
$$lookup : store \times var \to val$$

があり, 以下の公理を満たすとする.

$$lookup(update(\sigma, x, v), y) =$$
$$\quad \text{if } x = y \text{ then } v \text{ else } lookup(\sigma, y)$$

読み易さのため, $\sigma(x) \equiv lookup(\sigma, x)$, $\sigma[x := v] \equiv update(\sigma, x, v)$ と略記する. また, $\bot_{store}$ で, すべての変数の値が未定義の記憶域を表す.

sc 型 (セキュリティクラス (**SC**) の型) .

メタ変数として以下を用いる.

$x, x_1, \ldots, y_1, \ldots : var$ $\quad$ $M, M_1, \ldots : expression$ $\quad$ $C : command$
$P, P_1, P_2 : cseq$ $\quad$ $\sigma, \sigma', \sigma'' : store$

プログラムの意味を与える意味関数を定義する. この関数は, 記憶域, (式, 文, 文の系列のいずれか) を引数に取り, 記憶域または値を返す.

$$\models : (store \to expression \to val) + (store \to command \to (store + val))$$
$$+ (store \to cseq \to (store + val))$$

- 「$\sigma \models M \Rightarrow v$」は, 記憶域 σ の下で式 M を評価した値が v であることを表す.
- 「$\sigma \models C \Rightarrow \sigma'$」は, 記憶域 σ の下で文 C を実行した直後の記憶域が σ' であることを表す.
- 「$\sigma \models C \Rightarrow v$」は, 記憶域 σ の下で文 C を実行した直後, 関数値 v が返されることを表す. これは, C が return M の場合のみあてはまる.
- 文の系列に対しても同様.

以下に意味関数を定義する公理と推論規則を与える.

(CONST) $\sigma \models c \Rightarrow c_{\mathcal{I}}$

(VAR) $\sigma \models x \Rightarrow \sigma(x)$

(PRIM) $$\frac{\sigma \models M_i \Rightarrow v_i \quad (1 \leq i \leq n)}{\sigma \models \theta(M_1, \ldots, M_n) \Rightarrow \theta_{\mathcal{I}}(v_1, \ldots, v_n)}$$

(CALL) $$\frac{\sigma \models M_i \Rightarrow v_i \quad (1 \leq i \leq n) \quad \sigma' \models P_f \Rightarrow v}{\sigma \models f(M_1, \ldots, M_n) \Rightarrow v} \left(\begin{array}{l} f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \quad P_f \\ \sigma' = \bot_{store}[x_1 := v_1] \cdots [x_n := v_n] \end{array} \right)$$

(ASSIGN) $$\frac{\sigma \models M \Rightarrow v}{\sigma \models x := M \Rightarrow \sigma[x := v]}$$

(IF1) $$\frac{\sigma \models M \Rightarrow \text{true} \quad \sigma \models P_1 \Rightarrow \sigma' \ (rsp.\ v)}{\sigma \models \text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi} \Rightarrow \sigma' \ (rsp.\ v)}$$

(IF2) $$\frac{\sigma \models M \Rightarrow \text{false} \quad \sigma \models P_2 \Rightarrow \sigma' \ (rsp.\ v)}{\sigma \models \text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi} \Rightarrow \sigma' \ (rsp.\ v)}$$

(WHILE1) $$\frac{\sigma \models M \Rightarrow \text{true} \quad \sigma \models P \Rightarrow \sigma' \quad \sigma' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}$$

(WHILE2) $$\frac{\sigma \models M \Rightarrow \text{false}}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma}$$

(RETURN) $$\frac{\sigma \models M \Rightarrow v}{\sigma \models \text{return } M \Rightarrow v}$$

(CONCAT1) $$\frac{\sigma \models C \Rightarrow \sigma' \quad \sigma' \models P \Rightarrow \sigma''}{\sigma \models C; P \Rightarrow \sigma''}$$

(CONCAT2) $$\frac{\sigma \models C \Rightarrow \sigma' \quad \sigma' \models P \Rightarrow v}{\sigma \models C; P \Rightarrow v}$$

3 解析アルゴリズム

著者らは, [K00] において解析アルゴリズムの概要を示し, その解析アルゴリズムを形式的に定義することにより健全性の証明を行なった. 本節では, 健全性の定義およびアルゴリズムの形式的定義を述べる.

3.1 健全性の定義

解析アルゴリズムとは, $\mathcal{A}^*[\![\cdot]\!] : program \rightarrow (fname \times sc \cdots \times sc) \rightarrow sc$ という型をもつ関数である. $\mathcal{A}^*[\![Prog]\!][f, \tau_1, \ldots, \tau_n] = \tau$ であるとき, $\mathcal{A}^*[\![\cdot]\!]$ は, $Prog$ で定義された n 引数関数 f に対して, 引数の SC が $\tau_1, \ldots, \tau_n$ であるとき, f の SC を τ と解析したことを表す.

定義 3.1 以下の条件が成り立つとき, 解析アルゴリズム $\mathcal{A}^*[\![\cdot]\!]$ は健全性 (**soundness**) を満たすという*:*

$Prog$ を任意のプログラム, $main$ を $Prog$ の main 関数とする.

$$\mathcal{A}^*[\![Prog]\!][main, \tau_1, \ldots, \tau_n] = \tau$$

であり, また,

$$\bot_{store} \models main(v_1, \ldots, v_n) \Rightarrow v$$
$$\bot_{store} \models main(v'_1, \ldots, v'_n) \Rightarrow v'$$
$$\forall i (1 \leq i \leq n) : \tau_i \sqsubseteq \tau.\ v_i = v'_i$$

であると仮定する. このとき, $v = v'$ が成り立つ. □

上の定義で, 解析アルゴリズムが健全性を満たすとは,「解析結果が τ であると仮定する. このとき, SC が τ 以下であるような実引数の値がすべて等しいならば, それ以外の実引数が変化しても実行結果の値は不変である」ことを表す. つまり, 解析アルゴリズムが「関数 f の SC は τ」と答えたときは, τ 以下でない実引数の情報は, 実行結果に流れ出ないと保証されることを表す.

3.2 情報フロー解析関数

2.3節で導入した型に加えて以下の型を用いる.

$\underline{store}$ 型 (記憶域の **SC**)

$$update : \underline{store} \times var \times sc \to \underline{store}$$
$$lookup : \underline{store} \times var \to sc$$

$\underline{store}$ 型でも, $store$ 型と同様の略記法を用いる. $\underline{\sigma}$ を $\underline{store}$ 型の元とするとき, $\underline{\sigma}(x)$ は, 変数 x の SC の解析 (途中) 結果を表す. sc で定義された半順序 $\sqsubseteq$ を型 $\underline{store}$ 上に次のように拡張することにより, $\underline{store}$ に束の構造を与えることができる. $\underline{\sigma}$, $\underline{\sigma}'$ を $\underline{store}$ 型の値とする.

$$\underline{\sigma} \sqsubseteq \underline{\sigma}' \Leftrightarrow \forall x \in var.\ \underline{\sigma}(x) \sqsubseteq \underline{\sigma}'(x)$$

$\underline{store}$ の最小元は $\forall x \in var.\ \underline{\sigma}(x) = \bot$ であるような $\underline{\sigma}$ であり, これを $\bot_{\underline{store}}$ と表す.

$\underline{fun}$ 型 (関数の **SC**) $\underline{store}$ 型と同様, 次の関数が定義されている.

$$update : \underline{fun} \times fname \times sc \times \cdots \times sc \times sc \to \underline{fun}$$
$$lookup : \underline{fun} \times fname \times sc \times \cdots \times sc \to sc$$

また,

$$F[f, \tau_1, \ldots, \tau_n] \equiv lookup(F, f, \tau_1, \ldots, \tau_n)$$
$$F[\langle f, \tau_1, \ldots, \tau_n \rangle := \tau] \equiv update(F, f, \tau_1, \ldots, \tau_n, \tau)$$

と略記する. $F[f, \tau_1, \ldots, \tau_n]$ は引数の SC が $\tau_1, \ldots, \tau_n$ であるときの関数 f の SC の解析 (途中) 結果を表す. sc の束構造を $\underline{store}$ 型に拡張したのと同様に, $\underline{fun}$ 型にも束構造を導入する. 最小元を $\bot_{\underline{fun}}$ と表す.

cv-$\underline{fun}$ 型 (**covariant** $\underline{fun}$ 型) $\underline{fun}$ 型の値 F のうち, 条件

$$(\tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_1, \ldots, \tau'_n) \text{ ならば } F[f, \tau_1, \ldots, \tau_n] \sqsubseteq F[f, \tau'_1, \ldots, \tau'_n]$$

を満たすものすべてからなる.

メタ変数として, 2.3節のものに加えて以下を用いる.

$$\underline{\sigma}, \underline{\sigma}', \underline{\sigma}'' : \underline{store} \qquad F, F_1, F_2 : \underline{fun}$$

以下では, 情報フローを解析する関数 $\mathcal{A}[\![\cdot]\!]$ を定義する. $\mathcal{A}$ は, 式, 関数の SC, 記憶域の SC を引数に取って SC を返し, 文または文の系列, 関数の SC, 記憶域の SC, implicit flow の SC を引数に取って SC または記

憶域の SC を返す. $\mathcal{A}[\![\cdot]\!]$ は $\underline{fun}$ 型 (関数の SC) 上の単調関数になるとは限らないが, その部分型である cv-$\underline{fun}$ 型上では単調関数となる [K00]. 後述のように, この性質を利用して, $\mathcal{A}[\![\cdot]\!]$ の不動点を計算することができる. ここで, implicit flow について説明する. 例として, 次のような if 文を考える.

$$\text{if } x = 0 \text{ then } y := 0 \text{ else } y := 1 \text{ fi}$$

この if 文内で, y への代入文の右辺には x は現れていないが, この if 文の実行直後において y の値が 0 か 1 かを調べることにより, x の値が 0 かどうかを知ることができる. すなわち, 変数 x の情報が変数 y に流れたと考えることができる. このように, if 文の条件部の情報が then および else 以下の文に流れること, および while 文の条件部の情報が do 以下の文に流れることを implicit flow と呼ぶ. 各文または文の系列について, それが属する if 文および while 文の条件部の SC を, その文 (の系列) での implicit flow の SC と定義する. $\mathcal{A}$ の計算においては, 各文 (の系列) が, どの if 文・while 文に属するかを記憶しておく代わりに, implicit flow の SC の解析結果を $\mathcal{A}$ の第 4 引数を用いて受け渡す.

$$\begin{aligned}\mathcal{A} : &(\mathit{expression} \times \underline{\mathit{fun}} \times \underline{\mathit{store}} \to \mathit{sc}) + (\mathit{command} \times \underline{\mathit{fun}} \times \underline{\mathit{store}} \times \mathit{sc} \to \underline{\mathit{store}}) \\ &+ (\mathit{cseq} \times \underline{\mathit{fun}} \times \underline{\mathit{store}} \times \mathit{sc} \to \underline{\mathit{store}})\end{aligned}$$

- 「$\mathcal{A}[\![M]\!](F, \underline{\sigma}) = \tau$」は, 各関数の SC の解析結果 F, 記憶域の SC $\underline{\sigma}$ の下で, 式 M の SC を τ と解析することを表す.
- 「$\mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu) = \underline{\sigma'}$」は, 各関数の SC の解析結果 F, 記憶域の SC $\underline{\sigma}$, implicit flow の SC ν の下で, 文 C を実行した直後の記憶域の SC を $\underline{\sigma'}$ と解析することを表す.
- 文の系列に対しても同様.

以下に $\mathcal{A}$ の定義を与える.

(CONST) $\mathcal{A}[\![c]\!](F, \underline{\sigma}) = \bot$

(VAR) $\mathcal{A}[\![x]\!](F, \underline{\sigma}) = \underline{\sigma}(x)$

(PRIM) $\mathcal{A}[\![\theta(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = \bigsqcup_{1 \le i \le n} \mathcal{A}[\![M_i]\!](F, \underline{\sigma})$

(CALL) $\mathcal{A}[\![f(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = F[f, \mathcal{A}[\![M_1]\!](F, \underline{\sigma}), \ldots, \mathcal{A}[\![M_n]\!](F, \underline{\sigma})]$

(ASSIGN) $\mathcal{A}[\![x := M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[x := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu]$

(IF) $\mathcal{A}[\![\text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi}]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P_1]\!](F, \underline{\sigma}, \nu \sqcup \tau) \sqcup \mathcal{A}[\![P_2]\!](F, \underline{\sigma}, \nu \sqcup \tau)$
ただし, $\tau = \mathcal{A}[\![M]\!](F, \underline{\sigma})$

(WHILE) $\mathcal{A}[\![\text{while } M \text{ do } P \text{ od}]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu \sqcup \mathcal{A}[\![M]\!](F, \underline{\sigma})) \sqcup \underline{\sigma}$

(RETURN) ret は関数の返り値を記憶する特別な局所変数とする.
$\mathcal{A}[\![\text{return } M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[\mathit{ret} := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu]$

(CONCAT) $\mathcal{A}[\![C; P]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu), \nu)$

プログラム Prog を入力として Prog で定義された各関数の情報フローを解析する関数 $\mathcal{A}[\![\cdot]\!] : \mathit{program} \to \underline{\mathit{fun}} \to \underline{\mathit{fun}}$ を次のように定義する:

$\mathit{Prog} \equiv \{f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \;\; P_f, \ldots\}$ とするとき,

$$\begin{aligned}&\mathcal{A}[\![\mathit{Prog}]\!](F) = \\ &\quad F[\langle f, \tau_1, \ldots, \tau_n\rangle := \big(\mathcal{A}[\![P_f]\!](F, \bot_{\underline{\mathit{store}}}[x_1 := \tau_1] \cdots [x_n := \tau_n], \bot)\big)(\mathit{ret}) \\ &\qquad \mid \mathit{Prog} \text{ で定義された各 } n \text{ 引数関数 } f \text{ と任意の SC } \tau_1, \ldots, \tau_n]\end{aligned} \tag{1}$$

束 $(S, \preceq)$ における関数 $f : S \to S$ に対して, f の最小不動点を $fix(f)$ で表す. プログラム $Prog$ に対して, $\mathcal{A}[\![Prog]\!]$ の最小不動点を与える関数

$$\mathcal{A}^*[\![Prog]\!] = fix(\lambda F.\mathcal{A}[\![Prog]\!](F))$$

が解析アルゴリズムである. $\mathcal{A}[\![Prog]\!]$ は有限の束 cv-$\underline{fun}$ 上の単調関数となるので [K00],

$$\mathcal{A}^*[\![Prog]\!] = \bigsqcup_{i \geq 0} \mathcal{A}[\![Prog]\!]^i(\bot_{\underline{fun}})$$

が成り立つ [M96]. ここで, $f^0(x) = x$, $f^{i+1}(x) = f(f^i(x))$. すなわち, $\mathcal{A}^*[\![Prog]\!]$ は, すべての内容が最小元であるような関数の SC から開始して, $\mathcal{A}[\![Prog]\!]$ を, 関数の SC が変化しなくなるまで繰り返し適用することにより計算できる.

定理 **3.2**[K00] 解析アルゴリズム $\mathcal{A}^*[\![\cdot]\!]$ は健全性を満たす. □

3.3 解析例

ここで, 次のような 2 関数からなるプログラムを例に, どのように解析が行なわれるのかを簡単に示す. $SCs = \{low, high\}$, $low \sqsubseteq high$ とする. ただし, 局所変数の宣言は省略している.

```
main(x)                          f(x)
{                                {
    while x > 0 do                   if x > 0 then
        y := x + 1;                      return x * f(x - 1)
        x := y - 4                   else
    od;                                  return 0
    return f(x)                      fi
}                                }
```

このプログラムを解析するためには, F の更新

$F[\langle main, \tau\rangle := \big(\mathcal{A}[\![P_{main}]\!](F, \bot_{\underline{store}}[x^{main} := \tau], \bot)\big)(ret)]$

$[\langle f, \tau\rangle := \big(\mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x^f := \tau], \bot)\big)(ret)]$ $for\ \forall \tau \in SCs$

を F が変化しなくなるまで繰り返せばよい. 以下の表は, この繰り返し計算による F の変化の過程を示したものである. ただし, 表中の L, H はそれぞれ, low, $high$ を表す. また, 第 i 列の SC は第 $i-1$ 列を用いて計算された値を表している.

	0	1	2	3
$F[main, low]$	L	L	L	L
$F[f, low]$	L	L	L	L
$F[main, high]$	L	L	H	H
$F[f, high]$	L	H	H	H

この表より, main 関数の実引数の SC が low であるとき, main 関数の返り値の SC は low となることがわかる. また, main 関数の実引数の SC が $high$ であるとき, main 関数の返り値の SC は $high$ となり得ることがわかる.

4 解析アルゴリズムの時間計算量

解析アルゴリズムが以下で定義する分配性を満たすことを証明する. また, 分配性を用いることにより解析アルゴリズムの時間計算量を改善できることを示す.

4.1 分配性

定義 4.1 解析アルゴリズム $\mathcal{A}^*[\![\cdot]\!]$ が分配性を満たすとは以下のことをいう.
「 $\mathcal{A}^*[\![Prog]\!][f, \tau_1, \ldots, \tau_n] = \tau$ かつ $\mathcal{A}^*[\![Prog]\!][f, \tau'_1, \ldots, \tau'_n] = \tau'$ であるとき,
$\mathcal{A}^*[\![Prog]\!][f, \tau_1 \sqcup \tau'_1, \ldots, \tau_n \sqcup \tau'_n] = \tau \sqcup \tau'$ となる. 」 □

つまり, 解析アルゴリズムが分配性を満たすと仮定すると, 入力が $\tau_1, \ldots, \tau_n$ であるときの解析結果 τ と 入力が $\tau'_1, \ldots, \tau'_n$ であるときの解析結果 τ' がわかっているとき, 入力が $\tau_1 \sqcup \tau'_1, \ldots, \tau_n \sqcup \tau'_n$ であるときの解析結果は, 解析アルゴリズムを実行しなくとも $\tau \sqcup \tau'$ であることがわかる.

補題 4.2 $F = \mathcal{A}^*[\![Prog]\!]$ とすると,
$\forall M, P, F, \underline{\sigma}_1, \underline{\sigma}_2, \nu_1, \nu_2$:

(a) $\mathcal{A}[\![M]\!](F, \underline{\sigma}_1 \sqcup \underline{\sigma}_2) = \mathcal{A}[\![M]\!](F, \underline{\sigma}_1) \sqcup \mathcal{A}[\![M]\!](F, \underline{\sigma}_2)$

(b) $\mathcal{A}[\![P]\!](F, \underline{\sigma}_1 \sqcup \underline{\sigma}_2, \nu_1 \sqcup \nu_2) = \mathcal{A}[\![P]\!](F, \underline{\sigma}_1, \nu_1) \sqcup \mathcal{A}[\![P]\!](F, \underline{\sigma}_2, \nu_2)$

*(*証明の方針*)* 不動点帰納法およびプログラムの構造に関する帰納法により示せる. □

定理 4.3 解析アルゴリズム $\mathcal{A}^*[\![\cdot]\!]$ は分配性を満たす.
*(*証明*)* 補題 4.2 から容易. □

系 4.4 $\forall f \in fname,\ \forall \tau_1, \ldots, \tau_n \in SCs$

$$\mathcal{A}^*[\![Prog]\!][f, \tau_1, \ldots, \tau_n] = \bigsqcup_{1 \leq i \leq n} \mathcal{A}^*[\![Prog]\!][f, \bot, \ldots, \tau_i, \ldots, \bot]$$

*(*証明*)* 定理 4.3 より明らか. □

4.2 時間計算量

この節では, まず [K00] で提案した解析アルゴリズムを直接用いて $\mathcal{A}^*[\![Prog]\!]$ を求める場合の時間計算量を述べ, その後, 前節で示した分配性を利用することによる時間計算量の改善について述べる.

プログラム $Prog$ の記述長を N, $Prog$ 内の関数の引数の個数の最大値を k, SC の個数を $|SC|$ とする. 3.2 節の定義に従って $\mathcal{A}^*[\![Prog]\!]$ を求める場合の計算量を考える. 与えられた F に対して $\mathcal{A}[\![Prog]\!](F)$ を計算するための時間計算量は $O(N \times |SC|^k)$ である. 一方, 一つの組 $\langle f, \tau_1, \ldots, \tau_n \rangle$ について $F[f, \tau_1, \ldots, \tau_n]$ のとり得る値は $|SC|$ 通りであり, 組 $\langle f, \tau_1, \ldots, \tau_n \rangle$ の総数は $O(N \times |SC|^k)$ である. 従って, F の更新が収束するまでの繰り返し回数は, 高々$O(N \times |SC|^{k+1})$ 回である. 以上より, 全体の時間計算量は $O(N^2 \times |SC|^{2k+1})$ となる. [SID98] の技法を用いて, $\mathcal{A}[\![Prog]\!](F)$ を求める際の重複計算を避けると, 時間計算量は $O(kN \times |SC|^{2k+1})$ に改善できる.

一方, 以下の補題が成り立つ.

補題 4.5 SCs を任意のセキュリティクラスの集合とし, $SCs^2 = \{\bot, \top\}, \bot \sqsubseteq \top$ とする.
SCs^2 に対して, $\mathcal{A}^*[\![Prog]\!][f, \bot, \ldots, \top, \ldots, \bot] = \top$ ならば,
任意の $\tau \in SCs$ に対して, $\mathcal{A}^*[\![Prog]\!][f, \bot, \ldots, \tau, \ldots, \bot] = \tau$
かつ,
SCs^2 に対して, $\mathcal{A}^*[\![Prog]\!][f, \bot, \ldots, \top, \ldots, \bot] = \bot$ ならば,
任意の $\tau \in SCs$ に対して, $\mathcal{A}^*[\![Prog]\!][f, \bot, \ldots, \tau, \ldots, \bot] = \bot$ □

系 4.4 および補題 4.5 を用いると次のようにして $\mathcal{A}^*[\![Prog]\!]$ を求めることができる. まず, $SCs^2 = \{\bot, \top\},\ \bot \sqsubseteq \top$ であるものを考え,

$$\mathcal{A}^*[\![Prog]\!][f, \top, \bot, \bot, \ldots, \bot], \mathcal{A}^*[\![Prog]\!][f, \bot, \top, \bot, \ldots, \bot], \ldots, \mathcal{A}^*[\![Prog]\!][f, \bot, \bot, \bot, \ldots, \top]$$

を解析アルゴリズムにより求める. 補題 4.5 からすぐに,

$$\mathcal{A}^*[\![Prog]\!][f,\tau_1,\bot,\bot,\ldots,\bot],\mathcal{A}^*[\![Prog]\!][f,\bot,\tau_2,\bot,\ldots,\bot],\ldots,\mathcal{A}^*[\![Prog]\!][f,\bot,\bot,\bot,\ldots,\tau_n]$$

が求まり, これに系 4.4 を用いると, $\mathcal{A}^*[\![Prog]\!][f,\tau_1,\tau_2,\ldots,\tau_n]$ を求めることができる. この方法を用いると, $\mathcal{A}^*[\![Prog]\!]$ を求めるための時間計算量は, $O(kN\times 2^{2k+1}+k\times|SC|^k)$ となる.

定理 4.6 プログラム $Prog$ の記述長を N, $Prog$ 内の関数の引数の個数の最大値を k, SC の個数を $|SC|$ とすると $\mathcal{A}^*[\![Prog]\!]$ を求める時間計算量は $O(kN\times 2^{2k+1}+k\times|SC|^k)$ である. □

5 あとがき

著者らは [K00] において, 一般に再帰を含む手続き型プログラムの情報フローを静的に解析するアルゴリズムを提案し, 本稿では, このアルゴリズムが分配性を満たすことを抽象解釈による解析アルゴリズムの形式的記述を用いて証明した . また, 解析アルゴリズムの時間計算量を解析し, 分配性を利用すると時間計算量が改善されることを示した. [Y00] では, 提案アルゴリズムの実装が行なわれている. いくつかの例プログラムに対する解析時間を以下の表に示す. ただし, この試作システムでは, 本稿で述べたアルゴリズムの高速化の手法は用いられていない. 全てのプログラム例において不動点に収束するまでの繰り返し回数は数回程度であった.

プログラム	行数	平均解析時間 (s)
簡単なチケット予約システム	419	0.050
ソートアルゴリズム	825	0.130
上記 2 つを含むライブラリ群	2471	2.270

今後は , 大域変数, 動的記憶域, 関数ポインタを含むような , より複雑なプログラムに対しても解析が行えるようにアルゴリズムを拡張することを考える .

謝辞

共同研究を通して日頃より御討論頂き, また, 解析時間に関するデータを提供して頂きました, 大阪大学大学院基礎工学研究科情報数理系専攻の大畑 文明氏, 横森 励士氏に深謝致します.

参考文献

[AFG99] M. Abadi, C. Fournet and G. Gonthier: “Secure Communications Processing for Distributed Languages,” IEEE Symp. on Security and Privacy, pp.74–88, 1999.

[BBM94] J. Banâtre, C. Bryce and D. Le Métayer: “Compile-Time Detection of Information Flow in Sequential Programs,” Proc. 3rd ESORICS, LNCS 875, pp.55–73, 1994.

[D76] D. E. Denning: “A Lattice Model of Secure Information Flow,” Communications of the ACM, Vol.19, No.5, pp.236–243, 1976.

[DD77] D. E. Denning and P. J. Denning: “Certifications of Programs for Secure Information Flow,” Communications of the ACM, Vol.20, No.7, pp.504–513, 1977.

[HR98] N. Heintze and J. G. Riecke: “The SLam Calculus: Programming with Secrecy and Integrity, ” Proc. 25th ACM POPL, pp.365–377, 1998.

[K00] 國信, 高田, 関, 井上: “束構造のセキュリティモデルに基づくプログラムの情報フロー解析,” 信学技報, SS 2000-30(2000-11).

[M96] J. Mitchell: “Foundations of Programming Languages,” The MIT Press, 1996.

[M99] A. C. Myers : “JFLOW: Practical Mostly-Static Information Flow Control,” Proc. 26th ACM POPL, pp.228-241, 1999.

[O95] P. Ørbæk: “Can You Trust Your Data?” Proc. TAPSOFT ’95, LNCS 915, pp.575–589, 1995.

[P94] G. Purnul: “Database Security,” Advances in Computers (M. Yovits Ed.), Vol.38, pp.1–72, 1994.

[SID98] H. Seki, Y. Ishihara and H. Dodo: "Testing Type Consistency of Method Schemas," IEICE Trans. on Information and Systems, Vol.E81-D, No.3, pp.278–287, 1998.

[SV98] G. Smith and D. Volpano: "Secure Information Flow in a Multi-Threaded Imperative Language," Proc. 25th ACM POPL, pp.355–364, 1998.

[VS97] D. Volpano and G. Smith: "A Type-Based Approach to Program Security," Proc. TAPSOFT '97, LNCS 1214, pp.607–621, 1997.

[Y00] 横森, 大畑, 高田, 関, 井上: "セキュリティ解析アルゴリズムの実現とオブジェクト指向言語への適用に関する一考察," 信学技報, SS 2000-29(2000-11).