Control Flow Obfuscation for FJ using Continuation Passing

Extended Version

Kenny Zhuo Ming Lu
School of Information Technology
Nanyang Polytechnic
luzhuomi@gmail.com
Information Systems Technology and Design
Singapore University of Technology and Design
kenny_lu@sutd.edu.sg

Abstract

Control flow obfuscation deters software reverse engineering attempts by altering the program's control flow transfer. The alternation should not affect the software's run-time behaviour. In this paper, we propose a control flow obfuscation approach for FJ with exception handling. The approach is based on a source to source transformation using continuation passing style (CPS). We argue that the proposed CPS transformation causes malicious attacks using context insensitive static analysis and context sensitive analysis with fixed call string to lose precision.

CCS Concepts • **Software and its engineering** → *Semantics*; • **Security and privacy** → *Software and application security*.

Keywords Control flow obfuscation, program transformation, continuation passing style

ACM Reference Format:
Kenny Zhuo Ming Lu. 2020. Control Flow Obfuscation for FJ using Continuation Passing: Extended Version. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Java applications are ubiquitous thanks to the wide adoption of android devices. Since Java byte-codes are close to their source codes, it is easy to decompile Java byte-codes back to source codes with tools. For example, `javap` shipped with JVM [14] can be used to decompile Java class files back to Java source. This makes the Man-At-The-End attack as one of the major security threat to Java applications. Code obfuscation is one of the effective mechanism to deter malicious attack through decompilation. There are many obfuscation techniques operating on the level of byte-codes [5, 16, 19]. In the domain of source code obfuscation, we find solutions such as [8] applying layout obfuscation. We put our interest in control flow obfuscation techniques, which include control flow flattening [3, 10, 20] and continuation passing [11]. Note that the difference between bytecode obfuscation and source code obfuscation is insignificant, because of the strong correlation between the Java bytecodes and source codes. In this paper, we propose an extension to the continuation passing approach to obfuscate FJ with exception handling.

We assume the attackers gain access to the byte-codes to which layout obfuscation has been applied. The attackers decompile the byte-codes into source codes and attempt to extract secret information by running control flow analysis on the decompiled code. Our goal here is to cause the control flow analysis become imprecise or more costly in computation.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Motivating Example

Example 1. To motivate the main idea, let's consider the following Java code snippet

```
class FibGen {
  int f1, f2, lpos;
  FibGen() {
    f1 = 0; f2 = 1; lpos = 1;
  }
  int get(int x) {
    int i = lpos;                    // (1)
    int r = -1;
    try {                            // (2)
      if (x < i) {                   // (3)
        throw new Exception();       // (4)
      } else {
        while (i < x) {              // (5)
          int t = f1 + f2;           // (6)
          f1 = f2; f2 = t; i++;
        }
```

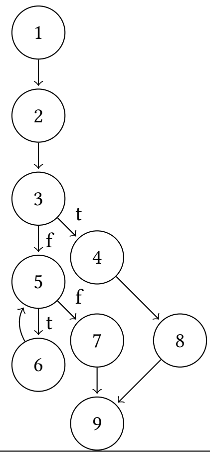

Figure 1. CFG of `get`

```
      }
      lpos = i;                          // (7)
      r = f2;
    } catch (Exception e) {              // (8)
      println("the input should be greater than "
          + i + ".");
    }
    return r;                            // (9)
  }
}
```

In the above we define a Fibonacci number generator in class `FibGen`. In the method `get`, we compute the Fibonacci number given the position as the input. Note that the generator maintains a state, in which we record the last two computed Fibonacci numbers, namely, `f1` and `f2` and the last computed position `lpos`. In method `get` lines 10 and 11, we raise an exception if the given input is smaller than `i` which has been initialized to `lpos`. Towards the end of the method, we catch the exception and print out the error message.

The number comment on the right of each statement indicates the code block to which the statement belongs. In Figure 1, we represent the function `get`'s control flow as a graph. Each circle denotes a code block from the source program. □

Inspired by the approach [11], our main idea is to translate control flow constructs, such as sequence, if-else, loop into CPS combinators. In the context of FJ with exception handling, we translate try-catch statements into CPS combinators as well.

In Figure 2 we find the obfuscated code snippet of `get` method in CPS style. The obfuscated code is in a variant of FJ, named FJ_λ, which is FJ with higher order functions, nested function declaration and mutable variables in function closures. `void => void` denotes a function type whose values accept no argument and return no result. `Exception => void` denotes a function type that accepts an exception and returns no result. `type NmCont = void => void` defines a type alias. `(void n) -> {return i < x};` defines an anonymous function whose input is of type `void` and the body returns a boolean value. For brevity, we omit the type annotations of the formal arguments where there is no confusion. The `return` key word is omitted when there is only one statement in the function body. We omit curly brackets in curried expressions, e.g. `x -> raise -> k -> { ... }` is the same as `x -> { raise -> { k -> { ... } } }`, where `x`, `raise` and `k` are formal arguments for the lambda abstractions. For convenience, we treat method declaration and lambda declaration as interchangeable. For instance, the lambda declaration

```
int => int => int f = x -> y -> { x + y }
```

is equivalent to the following method declaration

```
int f (int x, int y) { return x + y;}
```

In the last section, we mentioned that the layout obfuscation such as identifier renaming should have been applied to the obfuscated code; however in this paper we keep all the identifiers in the obfuscated code unchanged for the ease of reasoning. For the sake of assessing the obfuscation potency, we "flatten" the nested function calls into sequences of assignment statements. For example, let `x` and `y` be variables of type `int`, let `f` be a function of type `int => int => int` and `g` be a function of type `int => int`; instead of `int r = f(x)(g(y));` , we write:

```
int => int f_x = f(x);
int g_y = g(y);
int r = f_x(g_y);
```

As we observe in Figure 2, all the building blocks are continuation functions with type `CpsFunc`. The simple code blocks (1), (6), (7) and (8) from the original source code, which contain no control flow branching statements, are translated into nested CPS functions `get1`, `get6`, `get7` and `get8`. Block (4) containing a `throw` statement is translated into `get4` which applies the exception object to the exception handling continuation `raise` of type `ExCont`. Block (9) has a `return` statement, which is translated into a function in which we assign the variable being returned `r` to the `res` variable and call the normal continuation `k` of type `NmCont`. Block (2) is a try catch statement which is encoded as a call to the `trycatch` combinator in line 24. Similarly block (3) the if-else statement is encoded as a call to the `ifelse` combinator and block (5) the while loop is encoded as a call to the `loop` combinator.

In Figure 3, we present the definitions of the CPS combinators used in the obfuscation. Combinator `loop` accepts a condition test `cond`, a continuation executor `visitor` to be executed when the condition is satisfied, a continuation executor `exit` to be activated when the condition is not satisfied. Combinator `seq` takes two continuation executors and executes them in sequence. Combinator `trycatch` takes a continuation executor `tr` and an exception handling continuation `hdl`. It executes `tr` by replacing the current exception continuation with `ex_hdl`. Combinator `ifelse` accepts a condition test `cond`, a continuation for the then-branch `th` to be executed when the condition is satisfied, a continuation

```
type ExCont = Exception => void;
type NmCont = void => void;
type CpsFunc = ExCont => NmCont => void;
int get(int x) {
  int i, t, r, res; Exception ex;
  int => ExCont => (int => void) => void get_cps =
  x -> raise -> k -> {
    void => bool cond5 = n-> { i < x};
    void => bool cond3 = n -> {x < i};
    CpsFunc get5 = loop(cond5, get6, get7)
    CpsFunc get3 = ifelse(cond3,get4,get5);
    CpsFunc get1_2 = seq(get2, get9);
    CpsFunc pseq = seq(get1, get1_2);
    NmCont => void pseq_raise = pseq(raise);
    NmCont nk_res = n->k(res);
    return pseq_raise(nk_res);
  }
  CpsFunc get1 = (ExCont raise) -> (NmCont k) -> {
    i = this.lpos; r = -1; return k();
  }

  Exception => CpsFunc hdl =
    e -> {ex = e; return get8;}
  CpsFunc get2 = trycatch( get3, hdl);

  CpsFunc get4 = raise -> k
    -> raise(new Exception());
  CpsFunc get6 = raise -> k
    -> { t = this.f1 + this.f2; this.f1 = this.f2;
         this.f2 = t; i = i + 1; return k();}
  CpsFunc get7 = raise -> k
    -> { this.lpos = i; r = this.f2; return k();}
  CpsFunc get8 = raise -> k
    -> { System.out.println("..."); return k();}
  CpsFunc get9 = raise -> k
    -> { res = r; return k(); }
  NmCont id_bind = i -> { res = i; return; };
  CpsFunc get_x = get_cps(x);
  NmCont => void get_x_raise = get_x(id_raise);
  void ign = get_x_raise(id_bind);
  return res;
}
void id_raise(Exception e) { return ;}
```

Figure 2. `get` in CPS (flatten) (Line 1-43)

executor for the else-branch `el` to be activated when the condition is not satisfied.

To assess the potency of the obfuscation technique, let's put on the hat of the attackers and apply some static analysis to the obfuscated source code. The goal of the attack is to reconstruct the control flow graph from the obfuscated source. We apply an inter-procedural data flow analysis to the obfuscated code. For each variable or formal argument in the code, the analysis tries to approximate the set of possible lambda expressions which the variable/argument may capture during the execution. From the approximation we re-create the (global) control flow graph as presented in Figure 4. We give names to anonymous functions as λ_l where l refers to the

```
CpsFunc loop(void => Boolean cond,
  CpsFunc visitor, CpsFunc exit) {
  return raise -> k -> {
    if (cond()) {
      NmCont => void visitor_raise = visitor(raise
          );
      NmCont nloop = n -> {
        CpsFunc ploop = loop(cond, visitor, exit);
        NmCont => void = ploop_raise = ploop(raise
            );
        return ploop_raise(k);
      };
      return visitor_raise(nloop);
    } else {
      NmCont => void exit_raise = exit(raise);
      return exit_raise(k);
    }
  }
}
CpsFunc seq(CpsFunc first, CpsFunc second) {
  return raise -> k -> {
    NmCont => void first_raise = first(raise);
    NmCont n_second = n -> {
      NmCont => void second_raise = second(raise);
      return second_raise(k);
    };
    return first_raise(n_second);
  }
}
CpsFunc trycatch(CpsFunc tr, Exception => CpsFunc
    hdl) {
  return raise -> k -> {
    ExCont ex_hdl = ex -> {
      CpsFunc hdl_ex = hdl(ex);
      NmCont => void hdl_ex_raise = hdl_ex(raise);
      return hdl_ex_raise(k);
    }
    NmCont => void tr_hdl = tr(ex_hdl);
    return tr_hdl(k);
  }
}
CpsFunc ifelse(void => Boolean cond,
  CpsFunc th, CpsFunc el) {
  return raise -> k -> {
    if (cond()) {
      NmCont => void th_raise = th(raise);
      return th_raise(k);
    } else {
      NmCont => void el_raise = el(raise);
      return el_raise(k);
    }
  }
}
```

Figure 3. CPS Combinators (flatten) (Line 50-99)

line number appearing in Figures 2 and 3. In case that there are more than one anonymous functions introduced in line l. We use λ_i to denote the first one, λ'_i to denote the second

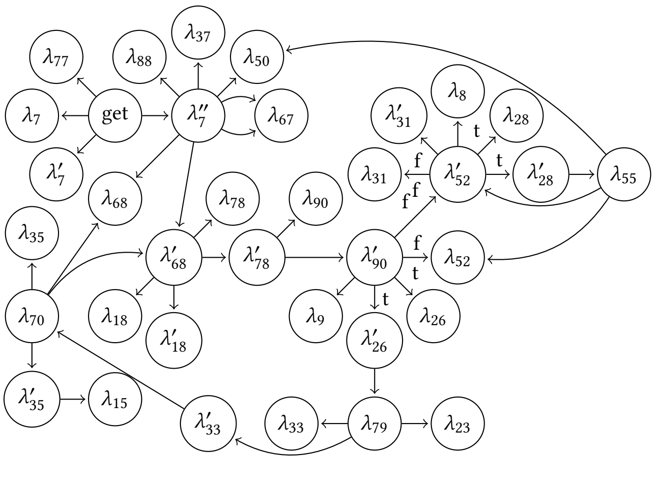

Figure 4. Reconstructed CFG of `get`

one and λ''_i to denote the third one. Compared to the CFG of the original source, the reconstructed CFG of the obfuscated code are far more complex. For instance, there exist more than one loop in the obfuscated CFG in Figure 4, namely,

1. $\lambda'_{52}, \lambda'_{28}, \lambda_{55}, \lambda'_{52}$,
2. $\lambda'_{68}, \lambda'_{78}, \lambda'_{90}, \lambda'_{26}, \lambda_{79}, \lambda'_{33}, \lambda_{70}, \lambda'_{68}$,

whereas there is clearly only one loop in the original CFG in Figure 1. The loss of precision is due to the fact that the attack which we simulate is using a context insensitive data flow analysis, which is known to be incomplete in the presence of multiple calls to the same function. For example, in Figure 2, lines 12 and 13, we call the combinator `seq` twice with different actual arguments. The analysis ignores the context and union the two sets of actual arguments into sets. These approximation are propagated along to the rest of the analysis. A similar observation is applicable to context sensitive analysis with a fixed size call string, in the presence of multiple calls to recursive combinators such as `loop`. Attackers may choose to use a context sensitive analysis, however to achieve a better approximation, the analysis will be much more costly and often not practical.

For the ease of establishing correctness result, we use an extension of the Single Static Assignment form for FJ with exception handling (SSAFJ-EH) as the source language of the translation. The construction of SSAFJ-EH can be extended from the work found in the literature [1], which is not the focus of this paper, hence we omit the details.

The contributions of this paper include,

- We formalize the single static assignment form of FJ with Exception Handling.
- We develop a control flow obfuscation algorithm by translating SSAFJ-EH to FJ$_\lambda$using continuation passing.
- We show that CPS based control flow obfuscation is effective against static analysis, in particular context insensitive control flow analysis.

The rest of the paper is organized as follows, In Section 3, we formalize SSAFJ-EH's syntax and semantics. In Section 4, we define the syntax of FJ$_\lambda$as well as its semantics. We formalize the source-to-source translation from SSAFJ-EH to FJ$_\lambda$. In Section 5, we discuss in details about the potency assessment of our obfuscation technique against static analyses. We discuss about related works in Section 6 and conclude in Section 7.

3 Single Static Assignment Form for FJ with Exception Handling

3.1 Syntax of SSAFJ-EH

We extend the syntax of SSAFJ [1] with exception handling,

(ClassDecl)	cd	::=	class C $\{\overline{fd}; \overline{md}\}$
(FieldDecl)	fd	::=	$t\ f$
(MethodDecl)	md	::=	$t\ m\ (t\ x)\ \{\overline{vd}; \overline{b}\}$
(VarDecl)	vd	::=	$t\ x$
(Block)	b	::=	$l : \{s\}$
(Statement)	s	::=	$\overline{a}$ | return e | throw e | $x = e.m(e)$
			| try$\{\overline{b}\}$ join $\{\overline{\phi}\}$ catch $(t\ x)$ $\{\overline{b}\}$ join $\{\overline{\phi}\}$
			| join $\{\overline{\phi}\}$ while e $\{\overline{b}\}$
			| if e $\{\overline{b}\}$ else $\{\overline{b}\}$ join $\{\overline{\phi}\}$
(Assignment)	a	::=	$x\ =\ e \vert e.f\ =\ e$
(Phi)	ϕ	::=	x = phi$(\overline{l : x})$
(Label)	l	::=	$L_0 \mid L_1 \mid L_2 \mid ...$
(Expression)	e	::=	$v \mid x \mid e.f \mid$ new $t()$ | this | $e\ op\ e$
(Operator)	op	::=	$+ \mid - \mid > \mid < \mid == \mid ...$
(Type)	t	::=	$int \mid bool \mid void \mid C$
(Value)	v	::=	$c \mid loc \mid$ null
(MemLoc)	loc	::=	`loc(0)` | `loc(1)` | ...

class C $\{\overline{fd}; \overline{md}\}$ defines a class. C denotes a class name. $\overline{fd}$ denotes a sequence of field declarations, $fd_1; ...; fd_n$. Likewise for $\overline{md}$ denotes a sequence of method declarations. For simplicity, we do not consider class inheritance, class constructors and method modifiers. Implicitly we assume each class comes with a default constructor and all field declarations are public and non-static. $t\ m\ (t\ x)\ \{\overline{vd}; \overline{b}\}$ defines a method declaration. For simplicity, we restrict the language to single argument methods. m denotes a method name. x, y and z denote variables. $\overline{b}$ defines a sequence of blocks. Each block is associated with a label l. Labels are unique within the method body. Reference to labels is restricted to the method's local scope. Each block consists of a sequence of assignment statements or a control flow statement. The last block in a method must contain a return statement. Note that all control flow statements potentially alter the default top-down execution order. The SSA form ensures the definition of a variable through assignment must dominate all the uses of this variable. Unlike the work [11], which uses low-level SSA structure with goto statements, the SSA form introduced in this paper is in a high-level structured form. That is, only certain control flow statements, such as if-else, try-catch and while, may carry one or more ϕ clauses. There is no goto statement. A ϕ assignment x = phi$(\overline{l : x})$ selects the right labeled argument $l_i : x_i$ to assign to the left hand

```
int get(int x) {
    int i_1, i_2, i_5, i_6, t_6, r_1, r_2, r_7;
L1: i_1 = this.lpos;
    r_1 = -1;
L2: try {
    L3: if (x < i_1) {
        L4: throw new Exception();
        } else {
        L5: join {i_5=phi(L3:i_1, L6:i_6)} while (
            i_5 < x) {
            L6: t_6 = this.f1 + this.f2;
                this.f1 = this.f2;
                this.f2 = t_6;
                i_6 = i_5 + 1;
            }
        L7: this.lpos = i_5;
            r_7 = this.f2;
        } join {i_3 = phi(L4:i_1, L7:i_5)}
    } join {i_2 = phi(L4:i_1)}
      catch (Exception e) {
    L8: System.out.println("the input..." + i_2 +
        ".");
    } join {r_2 = phi(L3:r_7, L8:r_1)};
L9: return r_2;
}
```

Figure 5. Method get in Single Static Assignment Form

side variable, based on the label of the preceding statement. For if-else statement, the ϕ assignment is inserted right after the then- and else-branches, which merges the possible different set of values from the branches into a new set of variables. In the while loop, the ϕ assignment is located before the loop-condition. In try-catch statement, we find two sets of ϕ assignments. The ϕ assignments located after the try block and catch block has a functionality similar to the one in if-else statement. The other one is located between the try block and the catch block. It is to merge the different sets of values that are arising in various parts of the try block due to exception being raised. We will discuss more in details in the semantics of SSAFJ-EH. t denotes a type. A type t can be basic types such as *int*, *void* or a class type C. A value v is either a constant, a memory location or null. The formal details will be elaborated in the upcoming subsection. Syntax of assignments and expressions is standard. For instance, the corresponding SSA form of the method `get` from the class `FibGen` in Example 1 is in Figure 5.

3.2 Semantics of SSAFJ-EH

We report a call-by-value semantics of SSAFJ-EH in Figures 6 and 7. We adopt the standard denotational semantics notation found in [15]. `GEnv` denotes a constant global environment which maps class names to field declarations and class names and method names to method declarations. We assume that the given program is free of type errors and there is no null pointer reference error. `LEnv` denotes a local variable environment which maps variables to values. `Store`

$$
\begin{array}{rrcl}
\text{(Global Decl Env)} & \mathtt{GEnv} & \subseteq & (\mathtt{Class} \times \overline{\mathtt{FieldDecl}}) \cup \\
 & & & ((\mathtt{Class} \times \mathtt{MethodName}) \times \mathtt{MethodDecl}) \\
\text{(Local Decl Env)} & \mathtt{LEnv} & \subseteq & (\mathtt{Variable} \times \mathtt{Value}) \\
\text{(Memory Store)} & \mathtt{Store} & \subseteq & (\mathtt{MemLoc} \times \mathtt{Object}) \\
\text{(Object)} & \mathrm{obj} & ::= & \mathrm{obj}(t, \rho) \\
\text{(Exception)} & ex & ::= & \mathrm{exception}(v, \mathsf{LEnv}, \mathsf{Store}, l) \\
\text{(Object Field Map)} & \rho & \subseteq & (\mathtt{FieldName} \times \mathtt{Value})
\end{array}
$$

$$
\begin{array}{l}
\mathbb{MD}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: \mathtt{MethodDecl} \rightarrow \mathtt{Value} \rightarrow \mathtt{Value} \rightarrow \mathtt{GEnv} \rightarrow \mathtt{Store} \rightarrow \\
\qquad\qquad\qquad\qquad (\mathtt{Value}, \mathtt{Store})_{\mathtt{ex}} \\
\mathbb{MD}_{\mathsf{ssa}}\llbracket(t'\ m(t\ x)\{\overline{vd}; \overline{b}\})\rrbracket\ v_o\ v_x\ genv\ st = \\
\quad \text{let } lenv' = \mathbb{VD}_{\mathsf{ssa}}\llbracket\overline{vd}\rrbracket\ \{(this, v_o), (x, v_x)\} \\
\quad \text{in case } \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\overline{b}\rrbracket\ L_0\ genv\ lenv'\ st \text{ of} \\
\qquad \mathrm{exception}(v, lenv'', st'.l') \rightarrow \mathrm{exception}(v, lenv', st', L_0) \\
\qquad (v, lenv'', st', l') \rightarrow (v, st')
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: [\mathtt{Block}] \rightarrow \mathtt{Label} \rightarrow \mathtt{GEnv} \rightarrow \mathtt{LEnv} \rightarrow \mathtt{Store} \rightarrow \\
\qquad\qquad\qquad\qquad (\mathtt{Value}, \mathtt{LEnv}, \mathtt{Store}, \mathtt{Label})_{\mathtt{ex}} \\
\overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket b\rrbracket\ l\ genv\ lenv\ st \ = \quad \mathbb{B}_{\mathsf{ssa}}\llbracket b\rrbracket\ l\ genv\ lenv\ st \\
\overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket b; \overline{b}\rrbracket\ l\ genv\ lenv\ st \ = \quad \text{case } \mathbb{B}_{\mathsf{ssa}}\llbracket b\rrbracket\ l\ genv\ lenv\ st \text{ of} \\
\qquad\qquad \mathrm{exception}(v, lenv', st', l') \rightarrow \mathrm{exception}(v, lenv', st', l) \\
\qquad\qquad (v, lenv', st', l') \rightarrow \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\overline{b}\rrbracket\ l'\ genv\ lenv'\ st'
\end{array}
$$

$$
\begin{array}{l}
\mathbb{B}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: \mathtt{Block} \rightarrow \mathtt{Label} \rightarrow \mathtt{GEnv} \rightarrow \mathtt{LEnv} \rightarrow \mathtt{Store} \rightarrow \\
\qquad\qquad (\mathtt{Value}, \mathtt{LEnv}, \mathtt{Store}, \mathtt{Label})_{\mathtt{ex}} \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\text{if } (e)\ \{\overline{b_1}\} \text{ else } \{\overline{b_2}\} \text{ join}\{\overline{\phi}\}\}\rrbracket\ l_p\ genv\ lenv\ st = \\
\qquad \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket\{\rrbracket e\}\ genv\ lenv\ st \text{ of} \\
\qquad (true, st') \rightarrow \text{case } \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\overline{b_1}\rrbracket\ l\ genv\ lenv\ st' \text{ of} \\
\qquad\qquad \mathrm{exception}(v, lenv', st'', l') \rightarrow \mathrm{exception}(v, lenv, st'', l) \\
\qquad\qquad (v, lenv', st'', l') \rightarrow (v, \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi}\rrbracket\ l'\ lenv', st'', l) \\
\qquad (false, st') \rightarrow \text{case } \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\overline{b_2}\rrbracket\ l\ genv\ lenv\ st' \text{ of} \\
\qquad\qquad \mathrm{exception}(v, lenv', st'', l') \rightarrow \mathrm{exception}(v, lenv, st'', l) \\
\qquad\qquad (v, lenv', st'', l') \rightarrow (v, \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi}\rrbracket\ l'\ lenv', st'', l) \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\text{return } e\}\rrbracket\ l_p\ genv\ lenv\ st = \ \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad\qquad (v, st') \rightarrow (v, lenv, st', l) \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\text{throw } e\}\rrbracket\ l_p\ genv\ lenv\ st = \ \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad\qquad (v, st') \rightarrow \mathrm{exception}(v, lenv, st', l) \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\text{try}\{\overline{b}\} \text{ join}\{\overline{\phi_r}\} \text{ catch}(t\ x)\ \{\overline{b'}\} \text{ join } \{\overline{\phi_k}\}\rrbracket\ l_p\ genv\ lenv\ st = \\
\quad \text{case } \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\{\overline{b}\}\rrbracket\ genv\ lenv\ st \text{ of} \\
\quad (v, lenv', st', l') \rightarrow (v, \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi_k}\rrbracket\ l'\ lenv', st', l') \\
\quad \mathrm{exception}(v, lenv', st', l') \rightarrow \\
\qquad \text{let } lenv'' = \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi_r}\rrbracket\ l'\ lenv' + (x, v) \\
\qquad \text{in case } \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\overline{b'}\rrbracket\ l'\ genv\ lenv''\ st' \text{ of} \\
\qquad\quad (v', lenv''', st'', l'') \rightarrow (v', \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi_k}\rrbracket\ l''\ lenv''', st'', l'') \\
\qquad\quad \mathrm{exception}(v', lenv''', st'', l'') \rightarrow \mathrm{exception}(v', lenv''', st'', l) \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\text{join } \{\overline{\phi}\} \text{ while}(e)\ \{\overline{b}\}\}\rrbracket\ l_p\ genv\ lenv\ st = \\
\quad \text{let } lenv' \ = \ \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi}\rrbracket\ l_p\ lenv \\
\quad \text{in case } \mathbb{E}_{\mathsf{ssa}}\llbracket e\rrbracket\ genv\ lenv'\ st \text{ of} \\
\quad (false, st') \rightarrow (null, lenv', st', l) \\
\quad (true, st') \rightarrow \text{case } \overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\overline{b}\rrbracket\ l\ genv\ lenv'\ st' \text{of} \\
\qquad \mathrm{exception}(v, lenv'', st'', l') \rightarrow \mathrm{exception}(v, lenv', st'', l) \\
\qquad (v, lenv'', st'', l') \rightarrow \\
\qquad\quad \mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\text{ join } \{\overline{\phi}\} \text{ while}(e)\ \{\overline{b}\}\}\rrbracket\ l'\ genv\ lenv''\ st'' \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{x = e_1.m(e_2)\}\rrbracket\ l_p\ genv\ lenv\ st = \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e_1\rrbracket\ genv\ lenv\ st \text{ of} \\
\quad (\mathtt{loc}(n), st') \rightarrow \text{case } st'(\mathtt{loc}(n)) \text{ of} \\
\qquad \mathrm{obj}(t, \rho) \rightarrow \text{case } genv(t, m) \text{ of} \\
\qquad\quad md \rightarrow \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e_2\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad\qquad (v'', st'') \rightarrow \text{case } \mathbb{MD}_{\mathsf{ssa}}\llbracket md\rrbracket\ \mathtt{loc}(n)\ v''\ genv\ st'' \text{ of} \\
\qquad\qquad\quad (v''', st''') \rightarrow (null, lenv + (x, v'''), st''', l) \\
\qquad\qquad\quad \mathrm{exception}(v''',, st''',_)) \rightarrow \mathrm{exception}(v''', lenv, st''', l) \\
\mathbb{B}_{\mathsf{ssa}}\llbracket l : \{\overline{a}\}\rrbracket\ l_p\ genv\ lenv\ st = \text{case } \mathbb{A}_{\mathsf{ssa}}\llbracket\overline{a}\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad (lenv', st') \rightarrow (null, lenv', st', l)
\end{array}
$$

Figure 6. Denotational Semantics of SSAFJ-EH (Part 1)

$$
\begin{array}{l}
\mathbb{VD}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: [\mathtt{VarDecl}] \rightarrow \mathtt{LEnv} \rightarrow \mathtt{LEnv} \\
\mathbb{VD}_{\mathsf{ssa}}\llbracket[\,]\rrbracket\ lenv = \qquad lenv \\
\mathbb{VD}_{\mathsf{ssa}}\llbracket(t\ x); \overline{vd}\rrbracket\ lenv = \quad \mathbb{VD}_{\mathsf{ssa}}\llbracket\overline{vd}\rrbracket\ (lenv + (x, null))
\end{array}
$$

$$
\begin{array}{l}
\mathbb{F}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: [\mathtt{Phi}] \rightarrow \mathtt{Label} \rightarrow \mathtt{LEnv} \rightarrow \mathtt{LEnv} \\
\mathbb{F}_{\mathsf{ssa}}\llbracket[\,]\rrbracket\ l\ lenv\ = lenv \\
\mathbb{F}_{\mathsf{ssa}}\llbracket x = \mathsf{phi}(l_1 : x_1, , ..., l_i : x_i, ..., l_n : x_n); \overline{\phi}\rrbracket\ l_i\ lenv = \\
\qquad \mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi}\rrbracket\ l_i\ lenv + (x, x_i)
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathbb{A}}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: [\mathtt{Assignment}] \rightarrow \mathtt{GEnv} \rightarrow \mathtt{LEnv} \rightarrow \mathtt{Store} \rightarrow (\mathtt{LEnv}, \mathtt{Store}) \\
\overline{\mathbb{A}}_{\mathsf{ssa}}\llbracket[\,]\rrbracket\ genv\ lenv\ st = (lenv, st) \\
\overline{\mathbb{A}}_{\mathsf{ssa}}\llbracket a; \overline{a}\rrbracket\ genv\ lenv\ st = \text{case } \mathbb{A}_{\mathsf{ssa}}\llbracket a\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad (lenv', st') \rightarrow \overline{\mathbb{A}}_{\mathsf{ssa}}\llbracket\overline{a}\rrbracket\ genv\ lenv' st'
\end{array}
$$

$$
\begin{array}{l}
\mathbb{A}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: \mathtt{Assignment} \rightarrow \mathtt{GEnv} \rightarrow \mathtt{LEnv} \rightarrow \mathtt{Store} \rightarrow (\mathtt{LEnv}, \mathtt{Store}) \\
\mathbb{A}_{\mathsf{ssa}}\llbracket x = e\rrbracket\ genv\ lenv\ st = \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad (v', st') \rightarrow (lenv + (x, v'), st') \\
\mathbb{A}_{\mathsf{ssa}}\llbracket e.f = e'\rrbracket\ genv\ lenv\ st = \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad (\mathtt{loc}(n), st') \rightarrow \text{case } st'(\mathtt{loc}(n)) \text{ of} \\
\qquad\qquad \mathsf{obj}(t, \rho) \rightarrow \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e'\rrbracket\ genv\ lenv\ st' \text{ of} \\
\qquad\qquad\qquad (v, st'') \rightarrow (lenv, st'' + (\mathtt{loc}(n), \mathsf{obj}(t, \rho + (f, v))))
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}_{\mathsf{ssa}}\llbracket\cdot\rrbracket :: \mathtt{Expression} \rightarrow \mathtt{GEnv} \rightarrow \mathtt{LEnv} \rightarrow \mathtt{Store} \rightarrow (\mathtt{Value}, \mathtt{Store}) \\
\mathbb{E}_{\mathsf{ssa}}\llbracket v\rrbracket\ genv\ lenv\ st = (v, st) \\
\mathbb{E}_{\mathsf{ssa}}\llbracket x\rrbracket\ genv\ lenv\ st = (lenv(x), st) \\
\mathbb{E}_{\mathsf{ssa}}\llbracket\mathsf{this}\rrbracket\ genv\ lenv\ st = (lenv(\mathsf{this}), st) \\
\mathbb{E}_{\mathsf{ssa}}\llbracket e.f\rrbracket\ genv\ lenv\ st = \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad (\mathtt{loc}(n), st') \rightarrow \text{case } st'(\mathtt{loc}(n)) \text{ of} \\
\qquad\qquad \mathsf{obj}(t, \rho) \rightarrow (\rho(f), st') \\
\mathbb{E}_{\mathsf{ssa}}\llbracket\mathsf{new}\ t()\rrbracket\ genv\ lenv\ st = \text{let } n = maxloc(st) \\
\qquad\quad \rho = \{(f, \mathsf{null}) | f \in genv(t)\} \\
\qquad \text{in } (\mathtt{loc}(n+1), st + (\mathtt{loc}(n+1), \mathsf{obj}(t, \rho))) \\
\mathbb{E}_{\mathsf{ssa}}\llbracket e_1\ op\ e_2\rrbracket\ genv\ lenv\ st = \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e_1\rrbracket\ genv\ lenv\ st \text{ of} \\
\qquad (v_1, st_1) \rightarrow \text{case } \mathbb{E}_{\mathsf{ssa}}\llbracket e_2\rrbracket\ genv\ lenv\ st_1 \text{ of} \\
\qquad\qquad (v_2, st_2) \rightarrow (apply(op, v_1, v_2), st_2)
\end{array}
$$

Figure 7. Denotational Semantics of SSAFJ-EH (Part 2)

defines a memory environment that maps memory locations to objects.

As a convention, we write $m(a)$ to refer to the object b associated with the key a in a mapping m, i.e. $(a, b) \in m$, given that all keys in m are unique. We use $m + (a, b)$ to denote an "update if exists – insert otherwise" operation, i.e. $m + (a, b) = \{(x, y) \in m | a \neq x\} \cup \{(a, b)\}$.

In this paper, we are only interested in the obfuscation of methods, hence we omit the semantics for class declaration and field declaration. $\mathbb{MD}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ defines the semantics of a method as a function expecting a reference to the current object, a value as the actual argument, a global environment and a memory store and returns a pair of value and memory store as result. Given a domain D, we write D_{ex} to denote $D \cup \mathtt{Exception}$. $\mathbb{VD}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ takes a list of variable declarations and a local declaration environment as inputs then registers each variable in the declaration environment. Note that we use Haskell's style of let-binding to introduce temporary variables and case expression for pattern matching. For breivity we omitted data constructors in the patterns when there is no ambiguity.

We adopt Haskell's style list syntax. [] denotes an empty list. $x : xs$ denotes a non-empty list where x refers to the head and xs refers to the tail. We assume there exists an implicit conversion from a sequence $b_1; b_2; ...; b_n$ to a list $b_1 : b_2 : ... : b_n : [\,]$.

$\mathbb{B}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ evaluates a block with respect to the context, i.e. the label of the preceding block, the local environment and the memory store. As the output, it returns a tuple of four items, namely, the value of the evaluation, the updated local environment, the updated memory store and the label from the exiting block if there is no exception occurred, otherwise an exception is returned. $\overline{\mathbb{B}}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ evaluates a sequence of blocks by applying $\mathbb{B}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ to each block in order, and propagates the resulting environments if there is no exception, otherwise the exception is propagated.

We highlight the a few interesting cases of $\mathbb{B}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$. In case of if-else statement, we evaluate either the then-branch $\overline{b_1}$ or the else-branch $\overline{b_2}$ depending on the result of the condition expression e. Given the label of the exiting block, either from $\overline{b_1}$ or $\overline{b_2}$, we apply $\mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi}\rrbracket$ to update the local environment in the result. In case of a try-catch statement, we first evaluate the try block. If the evaluation is successful, we compute the result by updating the local environment with $\mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi_k}\rrbracket$. If some exception arises from the evaluation of the try block, we generate a local environment with $\mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi_r}\rrbracket$ depending on the location from which the exception is raised. Next we evaluate the catch block under this local environment. Finally we update the output local environment with $\mathbb{F}_{\mathsf{ssa}}\llbracket\overline{\phi_k}\rrbracket$. In case of a method invocation, we evaluate the object expression into a memory location, from which we look up the memory store to retrieve the actual object and its type. From the global environment, we retrieve the method declaration based on the method name m. We call $\mathbb{MD}_{\mathsf{ssa}}\llbracket md\rrbracket$ with the actual arguments to compute the result of the right hand side. Finally, we return a tuple consists of a null value, a updated local environment with the updated binding of the left hand side x as well as the updated memory store. The remaining cases are trivial.

$\mathbb{F}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ walks through the list of ϕ assignments. For each ϕ of shape $x = \mathsf{phi}(l_1 : x_1, ..., l_i : x_i, ..., l_n : x_n)$, it searches for the label matching with the incoming label l_i. The value of x_i will be assigned to the variable x.

The definitions of $\overline{\mathbb{A}}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$, $\mathbb{A}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ and $\mathbb{E}_{\mathsf{ssa}}\llbracket\cdot\rrbracket$ are straightforward and we omit the details.

4 SSAFJ-EH to FJ$_\lambda$Translation

4.1 Syntax of FJ$_\lambda$

We consider the valid syntax of our target language FJ$_\lambda$

$$
\begin{array}{lrcl}
\text{(CLASSDECL)} & CD & ::= & class\ C\ \{\overline{FD}; \overline{MD}\} \\
\text{(FIELDDECL)} & FD & ::= & T\ F \\
\text{(METHODDECL)} & MD & ::= & T\ M\ (T\ X)\{\overline{VD}; \overline{S}\} \\
\text{(VARDECL)} & VD & ::= & T\ X \mid K\ X = \lambda \\
\text{(STATEMENT)} & S & ::= & A \mid \text{return}\ E \mid \text{if}\ (E)\ \{\overline{S}\}\ \text{else}\ \{\overline{S}\} \\
\text{(ASSIGNMENT)} & A & ::= & X = E \mid E.F = E \\
\text{(EXPRESSION)} & E & ::= & V \mid X \mid \text{this} \mid X(\overline{E}) \mid E.M(E) \mid E.F \\
 & & & \mid E\ op\ E \mid \text{new}\ T() \\
\text{(BASIC TYPE)} & T & ::= & int \mid bool \mid void \mid C \\
\text{(FUNCTION TYPE)} & K & ::= & T \mid K \Rightarrow K \\
\text{(VALUE)} & V & ::= & c \mid \lambda \mid LOC \mid \text{null} \\
\text{(MEMLOC)} & LOC & ::= & \texttt{loc}(0) \mid \texttt{loc}(1) \mid ... \\
\text{(LAMBDA)} & \lambda & ::= & \overline{(K\ X)} \rightarrow \{\overline{S}\}
\end{array}
$$

FJ$_\lambda$is an extension of FJ with the support of anonymous functions, i.e. lambda abstraction. FJ$_\lambda$differs from SSAFJ-EH as follows. Labels, while loop, try-catch and throw statements are excluded. FJ$_\lambda$supports a limited form of higher order functions. $K\ X = \lambda$ defines a local constant variable whose value is initialized to a lambda abstraction λ. Lambda abstraction $\overline{(K\ X)} \rightarrow \{\overline{S}\}$ denotes an anonymous function that expects zero or more parameters. Lambda functions do not introduce local variables within their own scopes. Lambda functions are nested in a top-level method. [1] Note that X and Y denote variables. Variables declared in a method are accessible within its nested functions. $X(\overline{E})$ denotes a function application where X is a variable bound to a lambda abstraction. $E.M(E)$ denotes a method application. The value V in the target language includes constants, lambda abstraction and memory locations.

4.2 Semantics of FJ$_\lambda$

In Figure 8 we describe the denotation semantics of FJ$_\lambda$. We use the upper case symbols GENV, LENV and STORE to capture the run-time bindings. They are similar to the counter-parts found in the SSAFJ-EH.

$\mathbb{MD}_{\texttt{fj}\lambda}[\![\cdot]\!]$ is similar to $\mathbb{MD}_{\texttt{ssa}}[\![\cdot]\!]$ except that it does not keep track of labels and exceptions. $\mathbb{VD}_{\texttt{fj}\lambda}[\![\cdot]\!]$ is nearly identical to $\mathbb{VD}_{\texttt{ssa}}[\![\cdot]\!]$ except that it handles an extra case of local function declaration.

$\mathbb{S}_{\texttt{fj}\lambda}[\![\cdot]\!]$ is a simplified version of $\overline{\mathbb{B}}_{\texttt{ssa}}[\![\cdot]\!]$ without the need of keeping track of the labels and the exceptions.

$\mathbb{A}_{\texttt{fj}\lambda}[\![\cdot]\!]$ is nearly identical to $\mathbb{A}_{\texttt{ssa}}[\![\cdot]\!]$, hence its definitions are omitted.

$\mathbb{E}_{\texttt{fj}\lambda}[\![\cdot]\!]$ differs from $\mathbb{E}_{\texttt{ssa}}[\![\cdot]\!]$ in case of function/method application. There are two different scenarios. (I) In case of $\mathbb{E}_{\texttt{fj}\lambda}[\![X(\overline{E})]\!]$ where $lenv(X)$ yields a lambda abstraction. We evaluate all the actual parameters E_1 to E_n into V_1 to V_n with the memory store being updated and propagated. We create an extended local environment by binding X_is to V_is. Finally we proceed with the evaluation the body of the lambda abstraction under the new environment and memory store. (II) In case of function application $\mathbb{E}_{\texttt{fj}\lambda}[\![E_1.M(E_2)]\!]$, we evaluate E_1 into a memory location $\texttt{loc}(n)$ with an updated memory store st'. By looking up $st'(\texttt{loc}(n))$ we retrieve the definition of the method associated with name M. We then evaluate E_2 and apply the resulting value to the method.

[1]The examples given in Section 2 Figures 2 and 3 seem to be violating this restriction. The violation is due to the "flattening" effect, which can be undone.

$$
\begin{array}{rrcl}
\text{(Global Decl Env)} & \textsf{GENV} & \subseteq & (\textsf{CLASS} \times \overline{\textsf{FIELDDECL}}) \cup \\
 & & & ((\textsf{CLASS} \times \textsf{METHODNAME}) \times \textsf{METHODDECL}) \\
\text{(Local Decl Env)} & \textsf{LENV} & \subseteq & (\textsf{VARIABLE} \times \textsf{VALUE}) \\
\text{(Memory Store)} & \textsf{STORE} & \subseteq & (\textsf{MEMLOC} \times \textsf{OBJECT}) \\
\text{(OBJECT)} & \text{obj} & ::= & \text{obj}(T, \rho)
\end{array}
$$

$$
\begin{array}{l}
\mathbb{MD}_{\texttt{fj}\lambda}[\![\cdot]\!] :: \textsf{METHODDECL} \rightarrow \textsf{VALUE} \rightarrow \textsf{VALUE} \rightarrow \textsf{GENV} \rightarrow \textsf{STORE} \\
\qquad \rightarrow (\textsf{VALUE}, \textsf{STORE}) \\
\mathbb{MD}_{\texttt{fj}\lambda}[\![(T'\ M(T\ X)\{\overline{VD}; \overline{S}\})]\!]\ V_0\ V_x\ genv\ st = \\
\quad \text{let}\ lenv = \mathbb{VD}_{\texttt{fj}\lambda}[\![\overline{VD}]\!]\ \{(this, V_0), (X, V_x)\} \\
\quad \text{in case}\ \mathbb{S}_{\texttt{fj}\lambda}[\![\overline{S}]\!]\ genv\ lenv\ st\ \text{of}\ (V, \_, st') \rightarrow (V, st')
\end{array}
$$

$$
\begin{array}{l}
\mathbb{VD}_{\texttt{fj}\lambda}[\![\cdot]\!] :: [\textsf{VARDECL}] \rightarrow \textsf{LENV} \rightarrow \textsf{LENV} \\
\mathbb{VD}_{\texttt{fj}\lambda}[\![K\ X = \lambda; \overline{VD}]\!]\ lenv = \mathbb{VD}_{\texttt{fj}\lambda}[\![\overline{VD}]\!]\ (lenv + (X, \lambda))
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}_{\texttt{fj}\lambda}[\![\cdot]\!] :: \textsf{STATEMENT} \rightarrow \textsf{GENV} \rightarrow \textsf{LENV} \rightarrow \textsf{STORE} \rightarrow (\textsf{VALUE}, \textsf{STORE}) \\
\mathbb{S}_{\texttt{fj}\lambda}[\![[]]\!]\ genv\ lenv\ st = (\text{null}, lenv, st) \\
\mathbb{S}_{\texttt{fj}\lambda}[\![A; \overline{S}]\!]\ genv\ lenv\ st = \text{let}\ (V, lenv', st') = \mathbb{A}_{\texttt{fj}\lambda}[\![A]\!]\ genv\ lenv\ st \\
\quad \text{in}\ \mathbb{S}_{\texttt{fj}\lambda}[\![\overline{S}]\!]\ genv\ lenv'\ st' \\
\mathbb{S}_{\texttt{fj}\lambda}[\![\text{return}\ E]\!]\ genv\ lenv\ st = \text{case}\ \mathbb{E}_{\texttt{fj}\lambda}[\![E]\!]\ genv\ lenv\ st\ \text{of} \\
\quad (V, st') \rightarrow (V, lenv, st') \\
\mathbb{S}_{\texttt{fj}\lambda}[\![\text{if}\ (E)\ \{\overline{S_1}\}\text{else}\{\overline{S_2}\}]\!]\ genv\ lenv\ st = \text{case}\ \mathbb{E}_{\texttt{fj}\lambda}[\![E]\!]\ genv\ lenv\ st\ \text{of} \\
\quad (true, st') \rightarrow \mathbb{S}_{\texttt{fj}\lambda}[\![\overline{S_1}]\!]\ genv\ lenv\ st' \\
\quad (false, st') \rightarrow \mathbb{S}_{\texttt{fj}\lambda}[\![\overline{S_2}]\!]\ genv\ lenv\ st'
\end{array}
$$

$$
\mathbb{A}_{\texttt{fj}\lambda}[\![\cdot]\!] :: \textsf{ASSIGNMENT} \rightarrow \textsf{GENV} \rightarrow \textsf{LENV} \rightarrow \textsf{STORE} \rightarrow (\textsf{VALUE}, \textsf{LENV}, \textsf{STORE})
$$

$$
\begin{array}{l}
\mathbb{E}_{\texttt{fj}\lambda}[\![\cdot]\!] :: \textsf{EXPRESSION} \rightarrow \textsf{GENV} \rightarrow \textsf{LENV} \rightarrow \textsf{STORE} \rightarrow (\textsf{VALUE}, \textsf{STORE}) \\
\mathbb{E}_{\texttt{fj}\lambda}[\![X(\overline{E})]\!]\ genv\ lenv\ st = \text{case}\ lenv(X)\ \text{of} \\
\quad (\overline{(K\ X)} \rightarrow \{\overline{S}\}) \rightarrow \text{case}\ \mathbb{S}_{\texttt{fj}\lambda}[\![\overline{S}]\!]\ genv\ lenv'\ st_n\ \text{of} \\
\qquad (V, st') \rightarrow (V, st') \\
\quad \text{where}\ (V_1, st_1) = \mathbb{E}_{\texttt{fj}\lambda}[\![E_1]\!]\ genv\ lenv\ st \\
\qquad ... \\
\qquad (V_n, st_n) = \mathbb{E}_{\texttt{fj}\lambda}[\![E_n]\!]\ genv\ lenv\ st_{n-1} \\
\qquad lenv' = lenv + (X_1, V_1) + ... + (X_n, V_n) \\
\mathbb{E}_{\texttt{fj}\lambda}[\![E_1.M(E_2)]\!]\ genv\ lenv\ st = \text{case}\ \mathbb{E}_{\texttt{fj}\lambda}[\![E_1]\!]\ genv\ lenv\ st\ \text{of} \\
\quad (\texttt{loc}(n), st_1) \rightarrow \text{case}\ st_1(\texttt{loc}(n))\ \text{of} \\
\qquad \text{obj}(T, \rho) \rightarrow \text{case}\ genv(T, M)\ \text{of} \\
\qquad\quad MD \rightarrow \text{case}\ \mathbb{E}_{\texttt{fj}\lambda}[\![E_n]\!]\ genv\ lenv\ st_1\ \text{of} \\
\qquad\qquad (V_2, st_2) \rightarrow \mathbb{MD}_{\texttt{fj}\lambda}[\![MD]\!]\ \texttt{loc}(n)\ v_2\ genv\ st_2
\end{array}
$$

Figure 8. Denotational Semantics of FJ$_\lambda$

4.3 SSAFJ-EH to FJ$_\lambda$Translation using CPS

We describe the SSAFJ-EH to FJ$_\lambda$translation using CPS in Figures 9 and 10. Specifically, we use command-based continuation pass style.

There are mainly two types of continuations, the exception continuation `Exception => void` and the normal continuation `void => void`. Each function in CPS form expects the first argument as the exception continuation and the second one as the normal continuation, except for the top level method.

$$
\begin{array}{l}
\mathbb{CMD}_{\text{cps}}[\![\cdot]\!] :: \texttt{MethodDecl} \rightarrow \texttt{METHODDECL} \\
\mathbb{CMD}_{\text{cps}}[\![t'\ m\ (t\ x)\{\overline{vd};\overline{b}\}]\!] = \\
\quad \text{let } \overline{VD} = \mathbb{CVD}_{\text{cps}}[\![\overline{vd}]\!] \\
\quad\quad (\overline{VD'}, E) = \mathbb{CB}_{\text{cps}}[\![[x/input]\overline{b}]\!]\ [\,]\ [\,] \\
\quad\quad T = t; T' = t'; X = x; M = m \\
\quad\quad D = T \Rightarrow (Exception \Rightarrow void) \Rightarrow (T' \Rightarrow void) \Rightarrow void\ M_{cps} = \\
\quad\quad\quad (T\ in) \rightarrow (Exception \Rightarrow void\ raise) \rightarrow (T' \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad\quad \{input = in; E(raise)(() \rightarrow k(res))\}; \\
\quad \text{in } T'\ M\ (T\ X)\ \{\overline{VD} + +\overline{VD'} + +[D]; T\ input; T'\ res; Exception\ ex; \\
\quad\quad M_{cps}(X)(id_{raise})(r \rightarrow \{res = r; return;\}); return\ res;\}
\end{array}
$$

$$
\mathbb{CVD}_{\text{cps}}[\![\cdot]\!] :: [\texttt{VarDecl}] \rightarrow [\texttt{VARDECL}]
$$

$$
\begin{array}{l}
\mathbb{CB}_{\text{cps}}[\![\cdot]\!] :: [\texttt{Block}] \rightarrow [\texttt{Phi}] \rightarrow [\texttt{Phi}] \rightarrow ([\texttt{VARDECL}], \texttt{EXPRESSION}) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\text{if } (e)\ \{\overline{b'}\}\text{else}\{\overline{b''}\}\text{ join }\{\overline{\phi}\}\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \\
\quad \text{let } (\overline{D'}, E') = \mathbb{CB}_{\text{cps}}[\![\overline{b'}]\!]\ \overline{\phi}\ \overline{\phi_r} \\
\quad\quad (\overline{D''}, E'') = \mathbb{CB}_{\text{cps}}[\![\overline{b''}]\!]\ \overline{\phi}\ \overline{\phi_r} \\
\quad\quad E = \mathbb{CE}_{\text{cps}}[\![e]\!] \\
\quad\quad (\overline{D'''}, E''') = \mathbb{CK}_{\text{cps}}[\![\overline{\phi_k}]\!]\ l \\
\quad \text{in}(\overline{D'} + +\overline{D''} + +\overline{D'''}, seq(ifelse(() \rightarrow E, E', E''), E''')) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\text{if } (e)\ \{\overline{b'}\}\text{else}\{\overline{b''}\}\text{ join }\{\overline{\phi}\}; \overline{b}\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \\
\quad \text{let } (\overline{D'}, E') = \mathbb{CB}_{\text{cps}}[\![\overline{b'}]\!]\ \overline{\phi}\ \overline{\phi_r} \\
\quad\quad (\overline{D''}, E'') = \mathbb{CB}_{\text{cps}}[\![\overline{b''}]\!]\ \overline{\phi}\ \overline{\phi_r} \\
\quad\quad E = \mathbb{CE}_{\text{cps}}[\![e]\!] \\
\quad\quad (\overline{D'''}, E''') = \mathbb{CB}_{\text{cps}}[\![\overline{b}]\!]\ \overline{\phi_k}\ \overline{\phi_r} \\
\quad \text{in}(\overline{D'} + +\overline{D''} + +\overline{D'''}, seq(ifelse(() \rightarrow E, E', E''), E''')) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\text{join }\{\overline{\phi}\}\text{ while } (e)\ \{\overline{b'}\}\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \\
\quad \text{let } (\overline{D}, E) = \mathbb{CK}_{\text{cps}}[\![\overline{\phi}]\!]\ minLabel(\overline{\phi}) \\
\quad\quad E' = \mathbb{CE}_{\text{cps}}[\![e]\!] \\
\quad\quad (\overline{D''}, E'') = \mathbb{CB}_{\text{cps}}[\![\overline{b'}]\!]\ \overline{\phi}\ \overline{\phi_r} \\
\quad\quad (\overline{D'''}, E''') = \mathbb{CK}_{\text{cps}}[\![\overline{\phi_k}]\!]\ l \\
\quad \text{in } (\overline{D} + +\overline{D''} + +\overline{D'''}, seq(E, loop(() \rightarrow E', E'', E'''))) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\text{join }\{\overline{\phi}\}\text{ while } (e)\ \{\overline{b'}\}; \overline{b}\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \\
\quad \text{let } (\overline{D}, E) = \mathbb{CK}_{\text{cps}}[\![\overline{\phi}]\!]\ minLabel(\overline{\phi}) \\
\quad\quad E' = \mathbb{CE}_{\text{cps}}[\![e]\!] \\
\quad\quad (\overline{D''}, E'') = \mathbb{CB}_{\text{cps}}[\![\overline{b'}]\!]\ \overline{\phi}\ \overline{\phi_r} \\
\quad\quad (\overline{D'''}, E''') = \mathbb{CB}_{\text{cps}}[\![\overline{b}]\!]\ \overline{\phi_k}\ \overline{\phi_r} \\
\quad \text{in } (\overline{D} + +\overline{D''} + +\overline{D'''}, seq(E, loop(() \rightarrow E', E'', E'''))) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\text{throw } e\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \text{let } \overline{(X, E)} = \mathbb{CF}_{\text{cps}}[\![\overline{\phi_r}]\!]\ l \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow void\ m_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{\overline{X = E}; \text{return } raise(\mathbb{CE}_{\text{cps}}[\![e]\!]);\} \\
\quad \text{in } ([D], m_l) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\text{return } e\}]\!]\overline{\phi_k}\ \overline{\phi_r} = \text{let } E = \mathbb{CE}_{\text{cps}}[\![e]\!] \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow voidm_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{res = E; \text{return } k();\} \\
\quad \text{in } ([D], m_l) \\
\mathbb{CB}_{\text{cps}}[\![l : \text{try}\{\overline{b}\}\text{join}\{\overline{\phi'_r}\}\text{ catch } (t\ x)\{\overline{b'}\}\text{join }\{\phi'_k\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \\
\quad \text{let } (\overline{D}, E) = \mathbb{CB}_{\text{cps}}[\![\overline{b}]\!]\ \overline{\phi'_k}\ \overline{\phi'_r} \\
\quad\quad (\overline{D'}, E') = \mathbb{CB}_{\text{cps}}[\![\overline{b'}]\!]\ \overline{\phi'_k}\ \overline{\phi_r} \\
\quad\quad E'' = (Exception\ x) \rightarrow \{ex = x; \text{return } E';\} \\
\quad\quad (\overline{D'''}, E''') = \mathbb{CK}_{\text{cps}}[\![\overline{\phi_k}]\!]\ l \\
\quad \text{in } (\overline{D} + +\overline{D'} + +\overline{D'''}, seq(trycatch(E, E''), E''')) \\
\mathbb{CB}_{\text{cps}}[\![l : \text{try}\{\overline{b}\}\text{join}\{\overline{\phi'_r}\}\text{ catch } (t\ x)\{\overline{b'}\}\text{join }\{\phi'_k\}; \overline{b''}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \\
\quad \text{let } (\overline{D}, E) = \mathbb{CB}_{\text{cps}}[\![\overline{b}]\!]\ \overline{\phi'_k}\ \overline{\phi'_r} \\
\quad\quad (\overline{D'}, E') = \mathbb{CB}_{\text{cps}}[\![\overline{b'}]\!]\ \overline{\phi'_k}\ \overline{\phi_r} \\
\quad\quad E'' = (Exception\ x) \rightarrow \{ex = x; \text{return } E';\} \\
\quad\quad (\overline{D'''}, E''') = \mathbb{CB}_{\text{cps}}[\![\overline{b''}]\!]\ \overline{\phi_k}\ \overline{\phi_r} \\
\quad \text{in } (\overline{D} + +\overline{D'} + +\overline{D'''}, seq(trycatch(E, E''), E'''))
\end{array}
$$

Figure 9. SSAFJ-EH to FJ$_\lambda$Translation (Part 1)

$$
\begin{array}{l}
\mathbb{CB}_{\text{cps}}[\![l : \{\overline{a}\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \text{let } \overline{A} = \mathbb{CA}_{\text{cps}}[\![\overline{a}]\!] \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow void\ m_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{\overline{A}; \text{return } k();\} \\
\quad\quad (\overline{D'}, E') = \mathbb{CK}_{\text{cps}}[\![\overline{\phi_k}]\!]\ l \\
\quad \text{in}([D] + +\overline{D'}, seq(m_l, E')) \\
\mathbb{CB}_{\text{cps}}[\![l : \{\overline{a}\}; \overline{b}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \text{let } \overline{A} = \mathbb{CA}_{\text{cps}}[\![\overline{a}]\!] \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow void\ m_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{\overline{A}; \text{return } k();\} \\
\quad\quad (\overline{D'}, E') = \mathbb{CB}_{\text{cps}}[\![\overline{b}]\!]\ \overline{\phi_k}\ \overline{\phi_r} \\
\quad \text{in } ([D] + +\overline{D'}, seq(m_l, E')) \\
\mathbb{CB}_{\text{cps}}[\![l : \{x = e_1.m(e_2)\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \text{let } E_1 = \mathbb{CE}_{\text{cps}}[\![e_1]\!] \\
\quad\quad E_2 = \mathbb{CE}_{\text{cps}}[\![e_2]\!] \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow m_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{E_1.m_{cps}(E_2)(raise)((T\ v) \rightarrow \{x = v; \text{return } k();\})\} \\
\quad\quad (\overline{D'}, E') = \mathbb{CK}_{\text{cps}}[\![\overline{\phi_k}]\!]\ l \\
\quad \text{in } ([D] + +\overline{D'}, seq(m_l, E')) \\
\mathbb{CB}_{\text{cps}}[\![l : \{x = e_1.m(e_2); \overline{b}\}]\!]\ \overline{\phi_k}\ \overline{\phi_r} = \text{let } E_1 = \mathbb{CE}_{\text{cps}}[\![e_1]\!] \\
\quad\quad E_2 = \mathbb{CE}_{\text{cps}}[\![e_2]\!] \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow m_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{E_1.m_{cps}(E_2)(raise)((T\ v) \rightarrow \{x = v; \text{return } k();\})\} \\
\quad\quad (\overline{D'}, E') = \mathbb{CB}_{\text{cps}}[\![\overline{b}]\!]\ \overline{\phi_k}\ \overline{\phi_r} \\
\quad \text{in } ([D] + +\overline{D'}, seq(m_l, E'))
\end{array}
$$

$$
\begin{array}{l}
\mathbb{CK}_{\text{cps}}[\![\cdot]\!] :: [\texttt{Phi}] \rightarrow Label \rightarrow ([\texttt{VARDECL}], \texttt{EXPRESSION}) \\
\mathbb{CK}_{\text{cps}}[\![\overline{\phi}]\!]\ l = \text{let } \overline{(X, E)} = \mathbb{CF}_{\text{cps}}[\![\overline{\phi}]\!]\ l \\
\quad\quad D = (Exception \Rightarrow void) \Rightarrow (void \Rightarrow void) \Rightarrow void\ mk_l = \\
\quad\quad\quad (Exception \Rightarrow void\ raise) \rightarrow (void \Rightarrow void\ k) \rightarrow \\
\quad\quad\quad \{\overline{X = E}; \text{return } k();\} \\
\quad \text{in } ([D], mk_l)
\end{array}
$$

$$
\mathbb{CA}_{\text{cps}}[\![\cdot]\!] :: [\texttt{Assignment}] \rightarrow [\texttt{ASSIGNMENT}]
$$

$$
\mathbb{CE}_{\text{cps}}[\![\cdot]\!] :: \texttt{Expression} \rightarrow \texttt{EXPRESSION}
$$

$$
\begin{array}{l}
\mathbb{CF}_{\text{cps}}[\![\cdot]\!] :: [\texttt{Phi}] \rightarrow Label \rightarrow [(\texttt{VARIABLE}, \texttt{EXPRESSION})] \\
\mathbb{CF}_{\text{cps}}[\![[\,]]\!]\ \_ = [\,] \\
\mathbb{CF}_{\text{cps}}[\![x = phi(..., l_i : x_i, ...); \overline{\phi}]\!]\ l \mid l == l_i = (x, x_i) : \mathbb{CF}_{\text{cps}}[\![\overline{\phi}]\!]\ l
\end{array}
$$

Figure 10. SSAFJ-EH to FJ$_\lambda$Translation (Part 2)

The function $\mathbb{CMD}_{\text{cps}}[\![\cdot]\!]$ converts a method from SSAFJ-EH to FJ$_\lambda$using CPS. Given an input method in SSAFJ-EH has type `t => t'`, the conversion synthesizes the output (or translated) method of type `T => (Exception => void) => (T' => void) => void`, by letting $T = t$ and $T' = t'$. The first argument is the input, the second argument is an exception continuation, and the third argument is the normal continuation. The conversion consists of the following steps. Firstly we apply the helper function $\mathbb{CVD}_{\text{cps}}[\![\cdot]\!]$ to translate the local variable declarations. $\mathbb{CVD}_{\text{cps}}[\![\cdot]\!]$ is an identity function, we omit its details. As the second step, we apply the helper function $\mathbb{CB}_{\text{cps}}[\![\cdot]\!]$ which translates the list of blocks from the source method. The result of the translation is a pair consisting of a list of local lambda declarations and a main expression. The main expression E is then applied to the exception continuation *raise* and the normal continuation

$() \rightarrow k(res)$. At last we synthesize the public interfacing method M which wraps around the CPS counter-part M_{cps}. [2]

Most of the translation tasks are computed in the helper function $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$. The function expects a list of blocks, a list of ϕ assignments from the subsequent block in the normal continuation and a list of ϕ assignments from the subsequent block in the exception continuation. $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$ translates the blocks structurally.

- In case of a singleton list containing an if-else block, we apply a helper function $\mathbb{CE}_{\mathsf{cps}}[\![\cdot]\!]$ to translate the conditional expression. Then we apply $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$ recursively to the blocks from the then-branch and the else-branch by using the ϕ assignments, $\overline{\phi}$, from the if-else statement's join clause. To "connect" the translated if-else back to the subsequent block in the normal continuation, we apply another helper function $\mathbb{CK}_{\mathsf{cps}}[\![\cdot]\!]$ to construct a continuation that resolves $\overline{\phi_k}$ with respect to l. The main expression is constructed structurally from the derived expressions from the various sub-steps with seq and $ifelse$ combinators, whose definitions can be found in Figure 3.
- In case of a non singleton list of which the head is an if-else block, we perform a trick similar to the previous case, except that we do not construct a continuation with $\mathbb{CK}_{\mathsf{cps}}[\![\cdot]\!]$ to resolve $\overline{\phi}_k$. Instead we apply $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$ to $\overline{b}$ recursively.
- In case of a singleton list containing a while block, we first need to apply $\mathbb{CK}_{\mathsf{cps}}[\![\cdot]\!]$ to resolve the $\overline{\phi}$ with respect to the label of the block from which we enter the while loop. For convenience, we assume that there exists a partial order among labels, i.e. $L_i \prec L_j$ implies that L_i must be on the path leading from L_o to L_j, where L_0 is the method's entry label. We assume that there are only two labels in the $\overline{\phi}$ assignments in all while blocks, i.e. the first label is the entry label to the while block, and the second label is loop-back label, and $minLabel(\overline{\phi})$ returns the entry label. Such a restrictive form does not limit the expressiveness of the language. We assume that there exists a pre-processing step that convert any programs into this form.
  After resolving $\overline{\phi}$ with respect to the entry label to the while block, we apply $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$ to $\overline{b}$ recursively to translate the while body. Lastly we apply $\mathbb{CK}_{\mathsf{cps}}[\![\cdot]\!]$ to construct a continuation that resolves $\overline{\phi}_k$ with respect to l. We build the main expression using the seq and $loop$ combinators, whose definitions can be found in Figure 3.
- In case of a singleton list containing a try-catch block, we apply $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$ recursively to the block in the try clause $\overline{b}$ with $\overline{\phi'_k}$ as ϕ assignments from the normal continuation and $\overline{\phi'_r}$ from the exception continuation. The catch clause block $\overline{b'}$ is translated with $\overline{\phi'_k}$ as ϕ assignments from the normal continuation and $\overline{\phi'_r}$ from the exception continuation. In order to bind the exception into the variable x, we define a wrapper lambda expression which expects the exception as input and assigns it to ex. (Recall that ex is defined in the top level method). Lastly, we construct a connecting continuation by resolving $\overline{\phi_k}$ with the current label l.
- In case of a singleton list containing a throw block, we first resolve the $\overline{\phi_r}$ from the exception handler with respect to the current label l by calling $\mathbb{CF}_{\mathsf{cps}}[\![\cdot]\!]$. Taking the result from the ϕ resolution, we define a continuation function m_l in which we bind the results, and call the exception continuation $raise()$ with the translation of the e.
- In case of a singleton list containing a return block, we define a continuation function m_l in which we assign the translation of e to res. (Recall that res is defined in the top level method). Then we call the continuation k.
- In cases of a list with a method invocation as the head, we translate the sub-expressions e_1 and e_2 into E_1 and E_2. We define a continuation function m_l in which we invoke $E_1.m_{cps}(E_2)$ with $raise$ as the exception continuation and the normal continuation is a lambda expression that captures the result of the method invocation into an argument v. In the body of the lambda exression we assign v to x before invoking the continuation k. Note that we treat M_{cps} same as m_{cps} and the call of m_{cps} could a recursive call or another method sharing the same closure context in the same scope.

The rest of the $\mathbb{CB}_{\mathsf{cps}}[\![\cdot]\!]$ cases are trivial.

The helper function $\mathbb{CF}_{\mathsf{cps}}[\![\cdot]\!]$ takes a list of ϕ assignments, a label and returns a list pair variable-expression pairs. For each ϕ assignment, it picks the right x_i associated with the matching label l as the second component of the resulting pair.

The helper function $\mathbb{CK}_{\mathsf{cps}}[\![\cdot]\!]$ synthesizes a continuation function that connects the block with label l with the block that $\overline{\phi}$ is defined, by making use of $\mathbb{CF}_{\mathsf{cps}}[\![\cdot]\!]$.

Helper functions $\mathbb{CA}_{\mathsf{cps}}[\![\cdot]\!]$ and $\mathbb{CE}_{\mathsf{cps}}[\![\cdot]\!]$ are identity functions, whose definitions are omitted.

In Figure 11, we find the full CPS translation of the `get` method in Figure 5. The result should be identical to the one in Figure 2, except that we do not apply "flattening" to nested and curry function calls, we insert extra connection blocks thanks to the ϕ resolutions.

Definition 1 (Consistent Global Environments). *Let $genv \in$ GEnv and $genv' \in \mathsf{GENV}$. Then we say $genv \vdash genv'$ iff $\forall(C,m) \in dom(genv) : genv'(C,m) = \mathbb{CMD}_{\mathsf{cps}}[\![genv(C,m)]\!]$.*

Lemma 4.1 (SSAFJ-EH to FJ_λ Translation Consistency). *Let m be a SSAFJ-EH method of a class C, o be (a reference to)*

[2] The continuation $r \rightarrow \{res = r; return;\}$ could have been simplified to $r \rightarrow \{return;\}$. However we keep to former just for consistency.

```
int get(int x) {
  int i_1, i_2, i_5, i_6, t_6, r_1, r_2. r_7;
  int input, res; Exception ex;

  int => ExCont => (int => void) => void get_cps =
  x -> raise -> k -> {
    input = x;
    return seq(get1, seq(trycatch
      (seq(ifelse(n-> {input < i_1},
            get4,seq(getk3b, loop( n->{i_5<input},
                seq(get6,get6k), seq(get7,get7k))))
                , getk3a)
      , e -> {ex = e; return seq(get8, get8k);}),
          get9)
      )(raise)(n->k(res))
    }
  ExCont => NmCont => void get1 =
      (ExCont raise) -> (NmCont k) -> {
          i_1 = this.lpos; r_1 = -1; return k();
  }
  ExCont => NmCont => void getk3a = raise -> k
    -> {r_2 = r_7; return k();}
  ExCont => NmCont => void get4 = raise -> k
    -> {i_2 = i_1; raise(new Exception())}
  ExCont => NmCont => void getk3b = raise -> k
    -> {i_5 = i_1; return k();}
  ExCont => NmCont => void get6 = raise -> k
    -> { t_6 = this.f1 + this.f2; this.f1 = this.
        f2;
         this.f2 = t_6; i_6 = i_5 + 1; return k()
             ;}
  ExCont => NmCont => void getk6 = raise -> k
    -> {i_5 = i_6; return k();}
  ExCont => NmCont => void get7 = raise -> k
    -> { this.lpos = i_5; r_7 = this.f2; return k
        ();}
  ExCont => NmCont => void get7k = rise -> k
    -> { i_2 = i_5; return k() }
  ExCont => NmCont => void get8 = raise -> k
    -> { System.out.println("..."); return k();}
  ExCont => NmCont => void get8k = raise -> k
    -> { r_2 = r_1; return k();}
  ExCont => NmCont => void get9 = raise -> k
    -> { res = r_2; return k(); }
  get_c(x)(id_raise)(i -> res = i; return);
  return res;
}
```

Figure 11. SSA to CPS Translation of `fib`

an object of class C, v be a value such that $o.m(v)$ is well-typed and terminating. Let $M = \mathbb{CMD}_{\mathsf{cps}}[\![m]\!]$. Let $genv \in$ GEnv, $genv' \in$ GENV such that $genv \vdash genv'$. Then we have $\mathbb{MD}_{\mathsf{ssa}}[\![m]\!]\ o\ v\ genv\ \{\} = \mathbb{MD}_{\mathtt{fj}\lambda}[\![M]\!]\ o\ v\ genv'\ \{\}$.

5 Obfuscation Potency Analysis

We analyze the potency of the CPS-based control flow obfuscation.

Let's try to apply inter procedural control flow analysis to the obfuscated code in Figures 2 and 3. Recall that the goal of the control flow analysis is to approximate the set of possible lambda abstractions that a program variable may capture during the run-time. From that result, as an attacker, we can create a global control flow graph with all the lambdas and methods involved.

Let Λ denote the set of all possible lambda values in the obfuscated program in FJ$_\lambda$. We have the following lattice $(2^\Lambda, \subseteq)$, whose top element $\top$ is Λ and $\bot$ is the empty set. We define the abstract state of the analysis as a map lattice mapping variables to sets of lambda functions.

$$(\text{STATE}) \quad \sigma \quad \subseteq \quad (\mathtt{VARIABLE} \times 2^\Lambda)$$

5.1 Context Insensitive Control Flow Analysis

We define the flow function $[\![\cdot]\!](\cdot) :: \mathtt{STATEMENT} \longrightarrow \mathtt{STATE} \longrightarrow \mathtt{STATE}$. The flow function takes a statement and a state and returns an updated state. Given a statement S, we write $[\![S]\!]$ to denote $[\![S]\!](\sigma_S)$ by making σ_S an implicit argument where

$$\sigma_S = join(S)$$

Let S be a statement and $pred(S)$ denote the set of preceding statements of S, we define the join function $join(S)$ as

$$join(S) = \bigcup_{P \in pred(S)} [\![P]\!]$$

The definition of flow function is given as follows.

$$\begin{array}{rcl}
[\![\text{return } E]\!](\sigma_S) & = & \sigma_S \\
[\![\text{if } E\ \{S\} \text{ else } \{S'\}]\!](\sigma_S) & = & \sigma_S \\
[\![E_1.F = E_2]\!](\sigma_S) & = & \sigma_S \\
[\![X = c]\!](\sigma_S) & = & \sigma_S - X \cup [X \mapsto \emptyset] \\
[\![X = E.F]\!](\sigma_S) & = & \sigma_S - X \cup [X \mapsto \emptyset] \\
[\![X = E\ op\ E']\!](\sigma_S) & = & \sigma_S - X \cup [X \mapsto \emptyset] \\
[\![X = \text{new } T()]\!](\sigma_S) & = & \sigma_s - X \cup [X \mapsto \emptyset] \\
[\![X = \lambda]\!](\sigma_S) & = & \sigma_S - X \cup [X \mapsto \{\lambda\}] \\
[\![X = Y]\!](\sigma_S) & = & \sigma_S - X \cup [X \mapsto \sigma_S(Y)] \\
[\![X = G(E_1, ..., E_n)]\!](\sigma_S) & = & \sigma_S - X \cup [X \mapsto \mathtt{returned}] \\
\text{where} & & \\
\mathtt{returned} & = & \bigcup_{\lambda \in \sigma_S(G)} [\![\mathtt{returnStmt}(\lambda)]\!] \\
[\![X = E_1.M(E_2)]\!](\sigma_S) & = & \sigma_s - X \cup [X \mapsto \mathtt{returned}] \\
\text{where} & & \\
T & = & typeof(E_1) \\
\mathtt{returned} & = & \bigcup_{\lambda \in \mathsf{GENV}(T,M)} [\![\mathtt{returnStmt}(\lambda)]\!]
\end{array}$$

The first three cases handle return statement, if statement and field update. They do not contribute any changes to abstract state. In the cases of constant assignment, field assignment, binary operator and object instantiation we update the variable X with an empty set. In the case of lambda assignment, we set X to be a singleton set. In case of variable aliasing assignment, we set X's mapping to the same as the rhs. In case of lambda function invocation, we update the mapping of the variable X with a union of all returnable states from all the possible bindings of the variable G which is bound to some lambda expressions. In case of method call, it is similar to the lambda function except that we look up the lambda expression from the global environment.

We overload the flow function for a lambda function declaration, whose output abstract state, will serve as the predecessor of the first statement in the function body.

$$[\![\lambda]\!] = \bigcup_{S \in \texttt{caller}(\lambda)} \bot[a_1 \mapsto \texttt{eval}([\![S]\!], E_1^S), ..., a_n \mapsto \texttt{eval}([\![S]\!], E_n^S)]$$

where $\{a_1, ..., a_n\} = \texttt{formalArgs}(\lambda)$. Given a statement S that calls λ, E_i^S , denotes the actual argument at ith position.

The helper function $\texttt{caller}(\cdot) :: \Lambda \longrightarrow [\texttt{STATEMENT}]$ returns the set statements in which the function λ is invoked. Let $\bar{\sigma}$ denotes all the abstract states collected from all the statements of the target program.

$$\begin{array}{rcl} \texttt{caller}(\lambda) & = & \{S | S \in \texttt{STATEMENT} \wedge X \in dom(\bar{\sigma}) \wedge \\ & & \lambda \in \sigma_S(X) \wedge X(\overline{E}) \in rhs(S) \text{ for some } \overline{E}\} \end{array}$$

Helper function $\texttt{eval}(\cdot, \cdot) :: \texttt{STATE} \longrightarrow \texttt{EXPRESSION} \longrightarrow 2^{\Lambda}$, takes an abstract state and returns a set of lambda functions which the expression might evaluate to.

$$\begin{array}{rcl} \texttt{eval}(\sigma, c) & = & \emptyset \\ \texttt{eval}(\sigma, \lambda) & = & \{\lambda\} \\ \texttt{eval}(\sigma, X) & = & \sigma(X) \\ \texttt{eval}(\sigma, E\ op\ E') & = & \emptyset \\ \texttt{eval}(\sigma, E.F) & = & \emptyset \\ \texttt{eval}(\sigma, \texttt{new}\ T()) & = & \emptyset \end{array}$$

We apply the above analysis to our running example in Figures 2 and 3 until the abstract state reaches the fix point. We observe the following results.

var	func	var	func	var	func
$\texttt{raise}_7$	λ_{37}	$\texttt{k}_7$	λ_{37}	$\texttt{get}_2$	λ_{78}
get3	λ_{90}	get5	λ_{52}	get_1_2	λ_{68}
pseq	λ_{68}	pseq_raise	λ'_{68}	$\texttt{raise}_{18}$	λ_{43}
$\texttt{k}_{18}$	λ_{70}	$\texttt{hdl}_{23}$	λ_{23}	$\texttt{raise}_{26}$	λ_{79}
$\texttt{k}_{26}$	λ_{70}	$\texttt{raise}_{28}$	λ_{79}	$\texttt{k}_{28}$	λ_{55}
$\texttt{raise}_{31}$	λ_{79}	$\texttt{k}_{31}$	λ_{70}	$\texttt{raise}_{33}$	λ_{43}
$\texttt{k}_{33}$	λ_{70}	$\texttt{raise}_{35}$	λ_{43}	$\texttt{k}_{35}$	λ_{15}
$\texttt{cond}_{50}$	λ_8	visitor	λ_{28}	exit	λ_{31}
$\texttt{raise}_{52}$	λ_{79}	$\texttt{k}_{52}$	λ_{70}	visitor_raise	λ'_{28}
ploop	λ_{50}	ploop_raise	λ'_{52}	exit_raise	λ'_{31}
$\texttt{raise}_{68}$	λ_{43}	first	$\lambda_{18}, \lambda_{78}$	second	$\lambda_{35}, \lambda_{68}$
$\texttt{k}_{68}$	λ_{15}	first_raise	$\lambda'_{18}, \lambda'_{78}$	second_raise	$\lambda'_{35}, \lambda'_{68}$
$\texttt{raise}_{78}$	λ_{43}	$\texttt{k}_{78}$	λ_{70}	hdl_ex	λ_{33}
tr_hdl	λ'_{90}	hdl_ex_raise	λ'_{33}	$\texttt{cond}_{88}$	λ_9
th	λ_{26}	el	λ_{52}	$\texttt{raise}_{90}$	λ_{79}
$\texttt{k}_{90}$	λ_{70}	th_raise	λ'_{26}	el_raise	λ'_{52}
id_bind	λ_{37}	get_x	λ'_7	get_x_raise	λ''_7
tr	λ_{90}	$\texttt{hdl}_{77}$	λ_{23}	get_cps	λ_7
cond3	λ_9	cond5	λ_8	get1	λ_{18}
get4	λ_{26}	get6	λ_{28}	get7	λ_{31}
get8	λ_{33}	get9	λ_{35}	id_raise	λ_{43}
n_loop	λ_{55}	n_second	λ_{70}	loop	λ_{50}
seq	λ_{67}	trycatch	λ_{77}	ex_hdl	λ_{79}
ifelse	λ_{88}	n_k_res	λ_{15}		

For clarity and brevity, we adopt the following naming convention. We add line numbers to make common variables unique, e.g. $\texttt{raise}_7$ denotes the raise from line 7.

As we can observe from the above, most of variables are given a unique lambda term to which they can be bound, except for first, second, first_raise and second_raise. This is caused by the fact that the function seq is invoked in two different locations.

Through the analysis result, we can approximate a caller-callee relation between lambda abstractions. We reconstruct a global CFG of the obfuscated get by combining the call graphs and the local control flow graphs. The resulting CFG is presented in Figure 4.

As we discuss in the earlier section, the loss of precision is caused by the incompleteness of the context sensitive control flow analysis.

5.2 Context Sensitive Control Flow Analysis

A smarter attacker may attempt to uncover the CFG with better precision with context sensitive analysis.

In context sensitive analysis, we extend the abstract state with a context.

$$(\text{STATE}) \quad \sigma \quad \subseteq \quad (\texttt{Variable} \times 2^{\Lambda}) \cup \{\texttt{unreachable}\}$$

In this lattice, unreachable is the new $\bot$.

We redefine the flow function $[\![\cdot]\!](\cdot)(\cdot) :: \texttt{STATEMENT} \longrightarrow \texttt{CONTEXT} \longrightarrow \texttt{STATE} \longrightarrow \texttt{STATE}$. The flow function takes a statement, a context and a state and returns an updated state. Given a statement S and a context c, we write $[\![S]\!](c)$ to denote $[\![S]\!](c)(\sigma_S)$ by making σ_S an implicit argument where

$$\sigma_S = join(c, S)$$

Recall from on our running example, the imprecision of the context insensitive analysis is caused by the two calls of seq in lines 12 and 13 in Figure 2. If we define the context to be last call sites of the function, i.e. program locations, we would achieve a better precision,

var	context	func	var	context	func
first	12	λ_{78}	second	12	λ_{35}
first	13	λ_{18}	second	13	λ_{68}

This is also known as the context sensitive with call string. In the above case we use a call string with size of 1. However in the presence of multiple loops in the source code, the obfuscated code will contain multiple calls to the loop combinator, which contains a recursion. Choosing the size of the call string is a non-trivial task. A similar observation applies to other context sensitive analyses, such as functional approach, which consider the abstract state at the call site to be the context. The worst case complexity of these context sensitive analyses makes them less-practical to be applied in reverse engineering attacks without using heuristics [13].

5.3 Complexity of Sub-graph isomorphism

Regardless of the precision of the static analysis result, it is computationally expensive to match the original control flow graph with the approximated control flow graph in general.

Let the original CFG to be H and the approximated CFG to be G, we want to check whether H is sub graph isomorphic to G, which is NP-complete [6]. For instance Ullmann's algorithm [18] is known to be exponential. Some improvement with heuristic algorithms exist. There is no known algorithms solving this problem in polynomial time. This check only returns yes or no. Finding all possible isomorphic sub-graphs leads to sub graph matching problem, which is also NP-Complete.

Note that some linear algorithm exists for the special case in which one of the input graphs is fixed and the other is a planar graph. Unfortunately CFG generated from Java in general is not guaranteed to be a planar graph [17].

6 Related Works

The CPS-based control flow obfuscation is rooted from the connection between SSA forms in imperative programming languages and lambda terms in functional programming languages [2, 4, 9]. Our translation scheme is an extension of Lu's work[11] and is inspired by Kelsey's work [9]. In contrast with Lu's work, we are targeting FJ instead of C style language. As an improvement to Lu's work, our translation scheme supports exception handling, recursive call and call to methods within the same scope with continuations. In contrast to Kelsey's work, our translation is targeted at an imperative language extended with higher order function instead of Scheme. Giacobazzi et al proposed a method to construct general obfuscators using partial evaluation with distorted interpreters [7]. Their work provides a uniform reasoning of how attacks using abstract interpretation can be foiled by a particular obfuscation method (by constructing a specific distorted interpreter). Anonca and Corradi [1] formalized SSA form for FJ. They applied SSAFJ to improve the type analysis of object oriented languages such as Java.

7 Conclusion

We extend and develop CPS-based control flow obfuscation for FJ with exception handling. We formalize the strategy as a source to source translation scheme. We show that the control flow obfuscation technique is effective against attacks using static control flow analysis, in particular context insensitive analysis. We are in the process of implementing the reported technique. The progress and some examples can be found in our development repository [12].

References

[1] Davide Ancona and Andrea Corradi. 2016. A formal account of SSA in Java-like languages. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2016, Rome, Italy, July 17-22, 2016*, Vladimir Klebanov (Ed.). ACM, 2. https://doi.org/10.1145/2955811.2955813

[2] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Not.* 33, 4 (April 1998), 17–20. https://doi.org/10.1145/278283.278285

[3] Jan Cappaert and Bart Preneel. 2010. A General Model for Hiding Control Flow. In *Proceedings of the Tenth Annual ACM Workshop on Digital Rights Management (DRM '10)*. ACM, New York, NY, USA, 35–42. https://doi.org/10.1145/1866870.1866877

[4] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2003. A Functional Perspective on SSA Optimisation Algorithms. In *COCV '03: Compiler Optimization Meets Compiler Verification (Electronic Notes in Theoretical Computer Science)*, Jens Knoop and Wolf Zimmermann (Eds.), Vol. 82. Elsevier Science, 347–361. Issue 2. https://doi.org/10.1016/S1571-0661(05)82596-4

[5] Jien-Tsai Chan and Wuu Yang. 2004. Advanced obfuscation techniques for Java bytecode. *Journal of Systems and Software* 71, 1 (2004), 1 – 10. https://doi.org/10.1016/S0164-1212(02)00066-3

[6] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *IN STOC*. ACM, 151–158.

[7] Roberto Giacobazzi, Neil D. Jones, and Isabella Mastroeni. 2012. Obfuscation by Partial Evaluation of Distorted Interpreters. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM '12)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/2103746.2103761

[8] Guardsquare. 2020. ProGuard: Open Source Optimizer for Java and Kotlin. https://www.guardsquare.com/en/products/proguard

[9] Richard A. Kelsey. 1995. A Correspondence Between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/202529.202532

[10] Tímea László and Ákos Kiss. 2007. Obfuscating C++ Programs via Control Flow Flattening. In *10th Symposium on Programming Languages and Software Tools (SPLST 2007)*. 15–20.

[11] Kenny Zhuo Ming Lu. 2019. Control flow obfuscation via CPS transformation. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 54–60. https://doi.org/10.1145/3294032.3294083

[12] Kenny Zhuo Ming Lu. 2020. Control flow obfuscation for Java code. http://github.com/luzhuomi/obsidian/.

[13] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[14] Oracle. 2020. Oracle Java Technologies. https://www.oracle.com/java/technologies/

[15] K. Petersson. 1990. *Syntax and Semantics of Programming Languages.*

[16] Davide Pizzolotto and Mariano Ceccato. 2019. Obfuscating Java Programs by Translating Selected Portions of Bytecode to Native Libraries. *CoRR* abs/1901.04942 (2019). https://doi.org/10.1109/SCAM.2018.00012 arXiv:1901.04942

[17] Raphael. 2011. On Planarity of Control Flow Graphs. http://lmazy.verrech.net/2011/10/on-planarity-of-control-flow-graphs/

[18] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *JOURNAL OF THE ACM* 28, 1 (1976), 31–42.

[19] Balachandran Vivek, Sufatrio, Tan Darell, and Thing Vrizlynn. 2016. Control flow obfuscation for Android applications. *Computers & Security* 61 (05 2016). https://doi.org/10.1016/j.cose.2016.05.003

[20] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software Tamper Resistance: Obstructing Static Analysis of Programs.* Technical Report. Charlottesville, VA, USA.

A Appendix

A.1 Pre-processing step that fix while block that has multiple entry labels

The only possible case that violates the restrictive form is the use of try-catch with a while loop in the handler.

```
try {
  ...
  Li : throw new Exception();
  ...
  Lj : throw new Exception();
```

```
} join (...) catch (Exception e) {
  Ll:join (x = phi(Li:xi Lj:xj, Lm:xm)) while (e)
      {
    Lm: ...
  }
}
```

The above can be converted into the restrictive form by inserting an empty assignment block in front of the while block.

```
try {
  ...
  Li : throw new Exception();
  ...
  Lj : throw new Exception();
} join (...) catch (Exception e) {
  Lk: { };
  Ll: join (x = phi(Lk:xk, Lm:xm)) while (e) {
    Lm: ...
  }
}
```