모듈을 지원하는 함수 흐름 분석[*]

(Modularized Control Flow Analyses)

이욱세 이광근[†]
Oukseh Lee Kwangkeun Yi
KAIST

Abstract.

We show that for control-flow-analysis(CFA) of higher-order programs, modularizing CFAs makes them *unsound* with respect to the original whole-program CFAs; the derived modular analyses can give more accurate results than the original CFAs do. We then show the correctness of such modularized CFAs can be proven with respect to whole-program CFAs that are polyvariant at module-level. We show that the above two results are true for any k in kCFA; we present safe modularized kCFAs, proven by the above approach.

1 Introduction

Modular analyses, which analyze incomplete programs such as modules, are practical alternatives to whole-program analysis for large-scale realistic programs. A whole-program analysis needs the entire program text as its input, and it has to solve a large set of equations at once. If some parts of the program are modified, it has to re-analyze the entire program. A modular analysis, on the other hand, does not need the entire program and re-analyzes only the dependent parts of the modified module.

Usually, a program analysis is initially designed as a whole-program analysis, and then, only after its cost-effectiveness is assured by extensive testing, its modular version is designed.

In deriving a modular version from a whole analysis, however, it is not trivial to prove that the modular version is a sound extension of the original whole-program analysis. In particular, contrary to our expectation, modularized analysis can be more accurate than the whole-program analysis. Reminding us of the folklore in static analysis that improving the analysis accuracy can reduce the analysis cost, this phenomenon witnesses its reverse also holds: reducing the analysis overhead (by modularization) can improve the analysis accuracy.

Example 1 Consider the control-flow-analysis(CFA) of the following two higher-order code-fragments:

$$\begin{array}{l}\texttt{f = }\lambda\texttt{x.x}\\ \texttt{g = f }\lambda\texttt{y.y}\end{array} \quad \text{and} \quad \texttt{h = f }\lambda\texttt{z.z}$$

[*]본 연구는 과학기술부 창의적연구진흥사업 추진으로 얻어진 결과임.

[†]E-mail: {cookcu; kwang}@ropas.kaist.ac.kr

First, suppose we analyze the two fragments together. Because of the two calls to `f`, `f`'s formal parameter `x` is bound to both λ`y.y` and λ`z.z`. This information is propagated back to the call sites that we conclude `h` has λ`y.y` (a false-flow) as well as λ`z.z`.

On the other hand, separate analyses of the fragments (from left to right) give more accurate result. Analyzing the left fragment concludes that `f` has λ`x.x` and `g` has λ`y.y`. Analyzing the second fragment with this information concludes that `h` has only λ`z.z`. $\square$

This paper's main results are:

- For control-flow-analysis(CFA) of higher-order programs, modularizing CFAs makes them *unsound* with respect to the original whole-program CFAs; the derived modular analyses can give more accurate results than the original CFAs do.
- The correctness of such modularized CFA can be proven with respect to a whole-program CFA that is polyvariant at module-level. We call such analyses *module-variant* CFA.
- Above two results are true for any k in kCFA.
- Safe modularized kCFAs, proven by the above approach.

We consider CFA in this paper because it is the most basic analysis for higher-order programs. CFA is *the* prerequisite for almost all program analyses for higher-order programs. CFA safely estimates which functions will be called at each application. Estimating program executions along these control paths exposes program properties of interest. CFA's results are also directly used by compilers in de-functionalizing [Rey98a, Rey98b] application expressions into case selections on tags of closures.

After we describe the model for our modular analysis in Section 2, we first show the case for 0CFA. After the definition of 0CFA in Section 3we present its modular version 0CFA/m in Section 4, and show in Section 5that it is not sound with respect to the original 0CFA. In Section 6we define a module-variant 0CFA and prove in Section 7that the modular version is correct for this module-variant 0CFA. In Section 8, we show that the same holds for kCFA in general: we show that modularized version kCFA/m is not sound with respect to kCFA yet is safe with respect to the whole-program module-variant kCFA. We conclude in Section 9.

2 Incremental Model for Modular Analysis

We assume that a modular analysis works inside an incremental compilation environment [AM94]. A module consists of variable declarations ("`x = `e") and a signature that enlists a subset of the declared variables visible from the other modules. Module M depends on another module M', written $M' \sqsubset M$, iff module M uses variables of module M'. We assume that there exists a dependency between modules and we analyze modules in sequence by their topological order, as in the incremental compilation system. We assume there is no circular dependency between modules.

Figure 1 illustrates our incremental model of modular analysis. For each module in its dependence order, we analyze it and export some of the analysis results that subsequent

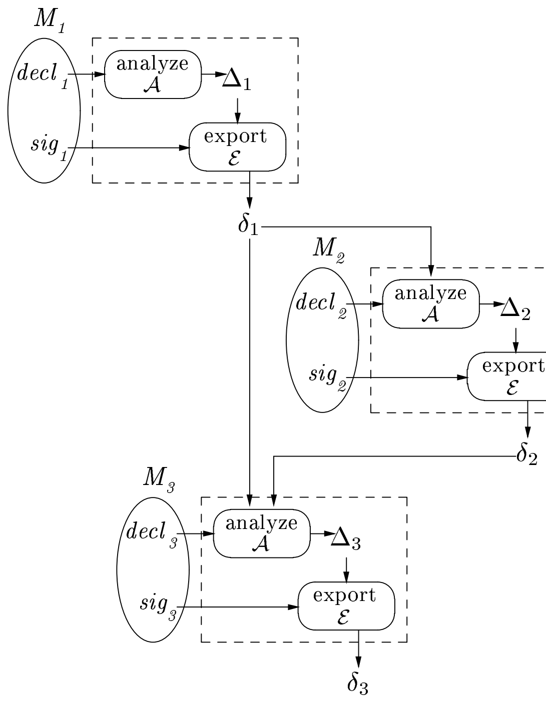

Figure 1: Incremental model for modular analysis. Module M_2 uses names declared in M_1, and M_3 uses those of M_1 and M_2.

modules may need. This exported results will be used by modules that depend on the current one. For a given module $M = (\mathit{decl}, \mathit{sig})$, let the analysis phase be $\mathcal{A}(M, \delta)$ with $\mathcal{A} : \mathit{Module} \times \mathit{Results} \rightarrow \mathit{Results}$. The second input δ is the exported results from the modules that M depends on. Let the result of this analysis be Δ. From Δ, we export only some parts of it that subsequent modules may need. Let this export phase be $\mathcal{E}(\Delta, \mathit{sig})$ with $\mathcal{E} : \mathit{Results} \times \mathit{Signature} \rightarrow \mathit{Results}$. For a program that consists of modules $M_1, \cdots, M_n$, each module M_i's analysis result Δ_i and its exported set δ_i (in Figure 1) are defined as $\Delta_i = \mathcal{A}\left(M_i, \cup_{M_j \sqsubset M_i} \delta_j\right)$ and $\delta_i = \mathcal{E}(\Delta_i, \mathit{sig}_i)$, where sig_i is the signature of M_i. The final analysis result $\mathit{Sol}(M_1, \cdots, M_n)$ for the whole-program is $\Delta_1 \cup \cdots \cup \Delta_n$.

3 0CFA

The whole-program 0CFA [Shi88], whose modular version we are designing, is shown in Figure 2. We present 0CFA in the style of Heintze and McAllester [HM97]. An input program is a sequence of declarations, and an expression is either a function, an application, a variable, or a non-function constant. The analysis computes edge "$n \rightarrow m$" between two nodes n and m. Nodes are syntactic objects: the variables or non-variable sub-expressions of the input program. Every variable is assumed distinct. Edge "$n \rightarrow m$" indicates that n may have the values of m (or, values of m may flow into n.). Applying the rules of Figure 2, we collect such edges until no more additions are possible. This process terminates because the number of

$$\begin{array}{llll} \textit{Label} \; l & \textit{Var} \; x & \textit{Constant} \; c & \\ \textit{Expr} & e & ::= & x \mid \lambda x.e^l \mid e^l \, e^l \mid c \\ \textit{Decl} & d & ::= & x = e^l \\ \textit{Program} & \wp & ::= & d^* \\ \\ \textit{Node} & n & ::= & x \mid l \mid \lambda x.e^l \\ \textit{Edge} & g & ::= & n \to n \end{array}$$

$$\frac{x = e^l \in \wp}{x \to l} \qquad \frac{(e_1^{l_1} \, e_2^{l_2})^l \in \wp \quad l_1 \to \lambda x.e^{l_0}}{l \to l_0 \quad x \to l_2}$$

$$\frac{x^l \in \wp}{l \to x} \qquad \frac{n \to m \quad m \to \lambda x.e^l}{n \to \lambda x.e^l}$$

$$\frac{(\lambda x.e^{l_0})^l \in \wp}{l \to \lambda x.e^{l_0}}$$

Figure 2: The language and its 0CFA.

nodes are finite for a given program. Edge "$n \to \lambda x.e^l$" in the final result indicates that n may evaluate into (or, is bound to) function $\lambda x.e^l$ in the input program. The correctness of 0CFA is known.

4 0CFA/m: A Modularized 0CFA

We present a modular version of 0CFA in Figure 3. Rules in the analysis phase $\mathcal{A}(M, \delta)$ are the same as the rules in the whole-program 0CFA except that instead of examining the whole-program text, they only examine the current module M and the exported edges δ from the referenced modules. The premise "$\in M$ or δ" means "is a sub-expression in either module M or a node of δ." In the export phase $\mathcal{E}(\Delta, \textit{sig})$, we conservatively export all the edges that can be needed by subsequent modules. The starting point is the signature. For a variable x in the signature, x's bindings are needed to analyze subsequent modules:

$$\frac{x \in \textit{sig}}{x \in \textit{Needed}} \; (\textit{Sig}).$$

If variable x is needed to analyze subsequent modules ($x \in \textit{Needed}$), then (1) its analysis result ($x \to \lambda y.e^l$) is exported and (2) we record ($FV(\lambda y.e^l) \subseteq \textit{Needed}$) that the free variables of the function are needed to analyze subsequent modules:

$$\frac{x \in \textit{Needed} \quad x \to \lambda y.e^l \in \Delta}{FV(\lambda y.e^l) \subseteq \textit{Needed} \quad x \to \lambda y.e^l} \; (\textit{ExportFn}).$$

$$\begin{array}{llll}
\textit{Signature} & \textit{sig} & ::= & \{x_1, \cdots, x_n\} \\
\textit{Declaration} & \textit{decl} & ::= & d^* \\
\textit{Module} & M & ::= & (\textit{decl}, \textit{sig}) \\
\\
\textit{Node} & n & ::= & x \mid l \mid \lambda x.e^l \\
\textit{Edge} & g & ::= & n \to n
\end{array}$$

Analysis phase. $\mathcal{A}(M, \delta)$ = edge-set Δ closed from module M and imported edges δ by the five rules:

$$\frac{x = e^l \in M}{x \to l}\ (\textit{Dec}) \qquad \frac{l_1 \to \lambda x.e^{l_0} \quad (e_1^{l_1}\ e_2^{l_2})^l \in M \text{ or } \delta}{l \to l_0 \quad x \to l_2}\ (\textit{App})$$

$$\frac{x^l \in M \text{ or } \delta}{l \to x}\ (\textit{Var}) \qquad \frac{n \to m \quad m \to \lambda x.e^l}{n \to \lambda x.e^l}\ (\textit{Tr})$$

$$\frac{(\lambda x.e^{l_0})^l \in M \text{ or } \delta}{l \to \lambda x.e^{l_0}}\ (\textit{Lam})$$

Export phase. $\mathcal{E}(\Delta, \textit{sig})$ = exported-edge-set δ closed by the two rules:

$$\frac{x \in \textit{sig}}{x \in \textit{Needed}}\ (\textit{Sig})$$

$$\frac{x \in \textit{Needed} \quad x \to \lambda y.e^l \in \Delta}{FV(\lambda y.e^l) \subseteq \textit{Needed} \quad x \to \lambda y.e^l}\ (\textit{ExportFn})$$

Figure 3: 0CFA/m: a modularized 0CFA.

5 0CFA/m Is Not Safe For 0CFA

Unexpectedly, this modular analysis 0CFA/m is more accurate than 0CFA. This situation does not mean that 0CFA/m is incorrect; 0CFA/m is still correct but because modularization makes the resulting analysis polyvariant, 0CFA/m fails to be a safe extension of the original 0CFA. The following example shows that the result of 0CFA/m does not always include that of 0CFA.

Example 2 Consider the program (consisting of two modules) in Example 1and its modular analysis:

$$M_1 = \left(\begin{array}{l} \texttt{f = } (\lambda\texttt{x.x}^2)^1 \\ \texttt{g = } (\texttt{f}^4\ (\lambda\texttt{y.y}^6)^5)^3 \end{array},\ \{\texttt{f},\texttt{g}\} \right)$$
$$M_2 = \left(\texttt{h = } (\texttt{f}^8\ (\lambda\texttt{z.z}^{10})^9)^7\ ,\ \{\texttt{h}\} \right)$$

If we analyze the whole program by 0CFA, the result includes a false-flow edge $\texttt{h} \to \lambda\texttt{y.y}$

because

$$\begin{tikzcd} \mathtt{h} \arrow[r] & (\mathtt{f}\ \lambda\mathtt{z.z}) \arrow[r] & \mathtt{x} \arrow[r] \arrow[dr] & \lambda\mathtt{z.z} \\ & & & \lambda\mathtt{y.y} \end{tikzcd}$$

(In presenting analysis results, we will not show transitively-closed edges.)

However, if we analyze the two modules by 0CFA/m this false-flow edge is avoided. Analyzing the first module returns

$$\begin{aligned} &\mathtt{f} \to 1 \to \lambda\mathtt{x.x}^2, \\ &\mathtt{g} \to 3 \to 2 \to \mathtt{x} \to 5 \to \lambda\mathtt{y.y}^6, \\ &6 \to \mathtt{y}, \end{aligned}$$

among which 0CFA/m exports only two edges:

$$\mathtt{f} \to \lambda\mathtt{x.x}^2 \quad \text{and} \quad \mathtt{g} \to \lambda\mathtt{y.y}^6.$$

Note that $\mathtt{x} \to \lambda\mathtt{y.y}$ is not included. With the exported edges from the first module, analyzing the second module returns

$$\mathtt{h} \to 7 \to 2 \to \mathtt{x} \to 9 \to \lambda\mathtt{z.z}^{10}.$$

The false-flow edge $\mathtt{h} \to \lambda\mathtt{y.y}$ is absent. □

Because 0CFA/m is more accurate than the whole 0CFA, the correctness relation between CFAs is

$$\begin{aligned} \text{Semantics} &\subseteq \text{0CFA} \quad \text{(correctness of 0CFA)} \\ \text{0CFA/m} &\subseteq \text{0CFA} \quad \text{(0CFA/m is not correct w.r.t. 0CFA)} \end{aligned}$$

where $A \subseteq B$ means that the result of B includes that of A. In order to prove the correctness of 0CFA/m

$$\text{Semantics} \subseteq \text{0CFA/m}$$

we want to find an analysis $\star$ which is not only safe with respect to the semantics (Semantics$\subseteq \star \subseteq$ 0CFA/m) but for which it is easy to prove that $\star \subseteq$ 0CFA/m.

It seems that such an analysis $\star$ must be polyvariant at module-level. In Example 2, there are two applications whose function part is `f`: (`f` λ`y.y`) in M_1 and (`f` λ`z.z`) in M_2. In 0CFA, both application nodes link to `x`: (`f` λ`y.y`)$\to$`x` and (`f` λ`z.z`)$\to$`x`. The former `x` is bound to λ`y.y` and the latter `x` to λ`z.z`. These two distinct bindings for `x` can be separated ("polyvariant") if we differentiate the variable `x` by the two modules M_1 and M_2.

This condition is indeed sufficient; we show that a polyvariant 0CFA at module-level is such an analysis $\star$. We call it *module-variant 0CFA*. This analysis is a convenient stepping-stone to proving correctness because (1) the proofs are between two static analyses (0CFA/m and module-variant 0CFA) that have a smaller gap than that between 0CFA/m and the actual control-flow semantics of programs and (2) the correctness of module-variant 0CFA is free since it is an instance of the infinitary CFA of Nielson and Nielson [NN97].

$$
\begin{array}{llll}
\mathit{Module} & M & & \\
\mathit{ModEnv} & \sigma & \in & \mathit{Var} \to \mathit{Module} \\
\mathit{Value} & & & \mathit{Lambda} \times \mathit{ModEnv} \\
\mathit{Solution} & S & \in & (\mathit{Var} + \mathit{Label}) \times \mathit{Module} \to \mathit{Value}
\end{array}
$$

$$
\begin{array}{lll}
(\mathit{var}) \quad S \models^{\sigma}_{M} x^{l} & \text{iff} & S(x, \sigma(x)) \subseteq S(l, M) \\
(\mathit{fn}) \quad S \models^{\sigma}_{M} (\lambda x.e^{l_0})^{l} & \text{iff} & (\lambda x.e^{l_0}, \sigma|_{FV(\lambda x.e^{l_0})}) \in S(l, M) \\
(\mathit{app}) \quad S \models^{\sigma}_{M} (e_1^{l_1}\ e_2^{l_2})^{l} & \text{iff} & S \models^{\sigma}_{M} e_1^{l_1} \quad \wedge \\
 & & S \models^{\sigma}_{M} e_2^{l_2} \quad \wedge \\
 & & \forall (\lambda x.e^{l_0}, \sigma') \in S(l_1, M) : \\
 & & \qquad S \models^{\sigma'[x \mapsto M]}_{M} e^{l_0} \quad \wedge \\
 & & \qquad S(l_2, M) \subseteq S(x, M) \quad \wedge \\
 & & \qquad S(l_0, M) \subseteq S(l, M) \\
(\mathit{con}) \quad S \models^{\sigma}_{M} c^{l} & \text{iff} & \text{true} \\
(\mathit{let}) \quad S \models^{\sigma}_{M} (\texttt{let}\ x = e^{l_1}\ \texttt{in}\ e^{l_2})^{l} & \text{iff} & S \models^{\sigma}_{M'} e^{l_1} \quad \wedge \\
 & & S \models^{\sigma[x \mapsto M']}_{M} e^{l_2} \quad \wedge \\
 & & S(l_1, M') \subseteq S(x, M') \quad \wedge \\
 & & S(l_2, M) \subseteq S(l, M) \\
 & & \text{where } x = e^{l_1} \text{ is in module } M'
\end{array}
$$

Figure 4: Module-variant 0CFA.

6 Module-Variant 0CFA

6.1 Definition

Module-variant 0CFA distinguishes the same expression label (or variable) by the originating modules whose evaluations need its values. For example, if $\lambda x.x$ is called from modules M_1 and M_2 with actual argument expressions e_1 and e_2, respectively, then we distinguish the formal parameter x by M_1 and M_2, binding e_1 to (x, M_1) and e_2 to (x, M_2). The function's body expression x also has two instances, indexed by M_1 and M_2, whose values are respectively those of e_1 and e_2.

The exact definition of the module-variant 0CFA is shown in Figure 4. In order to achieve its correctness freely, we define it as an instance of the infinitary CFA [NN97]. In order to fit with the program syntax in the infinitary CFA, we assume that a program is not just a collection of declarations (a collection of modules) but a single nested let-expression.

Throughout this paper, we say "a program $\wp$ consists of modules $M_1, \cdots, M_n$" to imply that the program is a nested let-expression that has exactly the declarations of the modules in the topological order of their dependencies, and whose let-body is a dummy constant. For example, a program that consists of two modules $M_1 = (x = e_1\ y = e_2, \{x\}), M_2 = (z = x, \{z\})$ is expressed "$\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{let}\ y = e_2\ \texttt{in}\ \texttt{let}\ z = x\ \texttt{in}\ c$." We assume that all the declarations in a merged let-expression have their associated module names.

Judgment "$S \models^{\sigma}_{M} e^{l}$" means solution S respects the situation that evaluating expressions of module M under environment σ needs to evaluate e. Environment σ maps free variables of e into the modules whose evaluation bind them. This environment determines the variable's module indices for the polyvariant effect.

For the input program $\wp$ that consists of modules $M_1, \cdots, M_n$, its module-variant 0CFA is defined [NN97] as the least S such that $S \models^{\emptyset}_{\varepsilon} \wp$ where $\emptyset$ is the empty module-context environment and ε is a dummy module index for the whole program.

Let's consider the rules, case by case.

Case (var). If a variable is necessary ($S \models^{\sigma}_{M} x^{l}$) for evaluating expressions of module M then the values $S(l, M)$ of its label must include those $S(x, \sigma(x))$ of the variable.

Case (fn). If an immediate function expression is needed ($S \models^{\sigma}_{M} (\lambda x.e^{l_0})^{l}$) for module M then the analysis result $S(l, M)$ at the label must include it.

Case (app). If an application is necessary ($S \models^{\sigma}_{M} (e_1^{l_1}\ e_2^{l_2})^{l}$) for evaluating expressions in module M, we propagate the same module context to its sub-expressions and to the body of the called function, and we determine value-flows across the call. The application's sub-expressions have the same module context: $S \models^{\sigma}_{M} e_1^{l_1} \wedge S \models^{\sigma}_{M} e_2^{l_2}$. For each function ($\forall(\lambda x.e^{l_0}, \sigma') \in S(l_1, M)$) that can be called, (1) its formal parameter x and its body e^{l_0} have the same module context: $S \models^{\sigma'[x \mapsto M]}_{M} e^{l_0}$, (2) values flow from argument expression e^{l_2} to the formal parameter x: $S(l_2, M) \subseteq S(x, M)$, and (3) values flow from body expression e^{l_0} to the call expression $(e_1^{l_1}\ e_2^{l_2})^{l}$: $S(l_0, M) \subseteq S(l, M)$. Note that a module-variant effect occurs because the function's argument and body have the call expression's module index.

Case (let). Everything is the same as in the application case, except that because the let-binding "x=e^{l_1}" is a declaration in a module, we have to use this module context for the variable x and its definition e^{l_1}.

6.2 Module-Variant 0CFA Is Correct

Because the module-variant 0CFA is an instance of the infinitary control flow analysis [NN97], it is correct by Theorem 4.1 of Nielson and Nielson [NN97].

Our module-variant 0CFA is instantiated as the following. For the context domains, whose elements are used to enable polyvariance, we choose as follows:

$$\begin{array}{rcl} \widehat{Mem} & \stackrel{\triangle}{=} & Module \\ \widehat{MC} & \stackrel{\triangle}{=} & Module \end{array}$$

Hence the projection is defined as $\pi(M, \sigma) \stackrel{\triangle}{=} M$. For the instantiators (of Table 3 [NN97, p.339]) we choose as follows. For application expression, we ensure that the context of a function body (M_0) and the context of its argument (M_x) are the same as the context of the application (M):

$$\mathcal{I}^{fn}_{app}(\sigma, M, \sigma', M'_0, e^{l}; M_0, M_x) \stackrel{\triangle}{=} M_0 = M_x = M.$$

For let-expression, we ensure that the context of a declaration (M_1) and the context of its declared variable (M_x) are the same as their associated module (M'):

$$\begin{aligned}&\mathcal{I}_{let}(\sigma, M, (\texttt{let } x\,\texttt{=}\,e^{l_1} \texttt{ in } e^{l_2})^l; M_1, M_2, M_x)\\ &\quad\stackrel{\triangle}{=} (M = M_2) \wedge (M_x = M_1 = M')\\ &\quad\quad \text{where } x\,\texttt{=}\,e^{l_1} \in M'.\end{aligned}$$

For other cases, instantiators does not change the context:

$$\begin{aligned}&\mathcal{I}_{fn}(\sigma, M, (\lambda x.e^{l_0})^l; M_0) \stackrel{\triangle}{=} M_0 = M\\ &\mathcal{I}_{app}(\sigma, M, (e_1^{l_1}\ e_2^{l_2})^l; M_1, M_2) \stackrel{\triangle}{=} M_1 = M_2 = M.\end{aligned}$$

7 0CFA/m Is Safe For Module-Variant 0CFA

0CFA/m is a safe extension of the module-variant 0CFA. We prove this by showing that there exists a solution S of the module-variant 0CFA that is covered by the result of 0CFA/m. Definition 2defines a solution S that is covered by the result of 0CFA/m, and Theorem 1asserts that the S is a solution of the module-variant 0CFA. We write Needed_M to denote the *Needed* set of the exporting phase in analyzing module M by 0CFA/m (See Figure 3).

Definition 1 (x reaches M via M') *Let Δ_M be the solved edges in analyzing module M by 0CFA/m.*

- *Variable x reaches M_n via M_0 iff $M_0 = M_n$ and $x \in \Delta_{M_n}$, or there exists a path $M_0 \sqsubset M_1 \cdots \sqsubset M_n$ such that for all $0 \le i < n$, $x \in \mathit{Needed}_{M_i}$.*
- *Expression e^l reaches M iff e^l is in module M or e^l occurs in the exported edges from the referenced modules of M.*
- *Environment σ reaches M iff, for all x in $dom(\sigma)$, x reaches M via $\sigma(x)$.*

Definition 2 ($|\mathit{Sol}_{\mathrm{0CFA/m}}(M_1, \cdots, M_n)|$) *Let $\mathit{Sol}_{\mathrm{0CFA/m}}(M_1, \cdots, M_n)$ be the result edges from analyzing modules $M_1, \cdots, M_n$ by 0CFA/m. Its corresponding form $|\mathit{Sol}_{\mathrm{0CFA/m}}(M_1, \cdots, M_n)|$ in the solution space for the module-variant 0CFA is defined as:*

$$|\mathit{Sol}_{\mathrm{0CFA/m}}(M_1, \cdots, M_n)|(n, M) = \left\{ (\lambda x.e^l, \sigma) \;\middle|\; \begin{array}{l} n \to \lambda x.e^l \in \Delta_M,\ \sigma \textit{ reaches } M, \\ dom(\sigma) = FV(\lambda x.e^l) \end{array} \right\}$$

where Δ_M is the 0CFA/m's solution for module M.

Fact. By definition, $|\mathit{Sol}_{\mathrm{0CFA/m}}(\cdot)|$ is "covered by" $\mathit{Sol}_{\mathrm{0CFA/m}}(\cdot)$: $(\lambda x.e^l, \_) \in |\mathit{Sol}_{\mathrm{0CFA/m}}(\cdot)|$ (n, M) implies $n \to \lambda x.e^l \in \mathit{Sol}_{\mathrm{0CFA/m}}(\cdot)$.

Theorem 1 (Correctness of 0CFA/m) *Let program $\wp$, as a let-expression, consist of modules $M_1, \cdots, M_n$. $|\mathit{Sol}_{\mathrm{0CFA/m}}(M_1, \cdots, M_n)| \models_{\varepsilon}^{\emptyset} \wp$ holds, where $\emptyset$ is the empty module-context environment and ε is a dummy module index for the whole program.*

Proof. Let $S = |Sol_{0\text{CFA/m}}(M_1, \cdots, M_n)|$. Judgment $S \models^{\sigma}_{M} e^l$ holds if it is included in the greatest fixed point of the function

$$F : \mathit{Judgments} \rightarrow \mathit{Judgments}$$

derived from Figure 4[NN97]. $F(Q)$ gives us a set of left-hand side judgments asserted by the rules of Figure 4assuming that judgments in Q hold. If we find a set Q of judgments such that $(S \models^{\emptyset}_{\varepsilon} \wp) \in Q$ and $Q \subseteq F(Q)$, then by the co-induction principle [MT91], Q is included in the greatest fixed point of F and $S \models^{\emptyset}_{\varepsilon} \wp$ holds.

Therefore, the module-variant 0CFA's solution, which is defined as the least X such that $X \models^{\emptyset}_{\varepsilon} \wp$, is included in the modularized solution $Sol_{0\text{CFA/m}}(M_1, \cdots, M_n)$. □

The correctness relation between CFAs becomes:

$$\text{Semantics} \subseteq \text{module-variant 0CFA} \subseteq \text{0CFA/m} \subseteq \text{0CFA}.$$

8 Modularizing kCFA

The next question is: what if we modularize an already polyvariant kCFA[Shi88]? The answer is that modularizing an already polyvariant kCFA can also make it more accurate. That is, kCFA/m can be more polyvariant than the original kCFA.

Example 3 Consider the following two modules.

$$M_1 = \left(\begin{array}{l} \texttt{f = (}\lambda\texttt{x.(}\lambda\texttt{z.z) x)} \\ \texttt{g = f }\lambda\texttt{y.y} \end{array} , \{\texttt{f,g}\} \right)$$
$$M_2 = \left(\texttt{h = f }\lambda\texttt{w.w} , \{\texttt{h}\} \right)$$

First consider the 1CFA for the whole program consisting of three declarations. Because 1CFA distinguishes the same variable by the call sites which bind the variable, it cannot distinguish the two dynamic calls to λ`z.z` inside `f`, which occurred for `g` and `h`. That is, the two instances of `z` are not distinguished. Hence, it concludes that `h` can evaluate to both λ`w.w` and λ`y.y`. However, analyzing modules separately, we can export from the first module `f` $\rightarrow$ `(`λ`x.(`λ`z.z) x)` and `g` $\rightarrow$ λ`y.y`, and conclude that `h` can evaluate to only λ`w.w`. For any k in kCFA, we can show similar counter-example where its modular version is more accurate than kCFA. □

Though the correctness of modularized kCFA *cannot* be proven in general with respect to the original kCFA, we can prove, as in 0CFA, that kCFA/m is correct with respect to a module-variant kCFA. The module-variant kCFA is achieved simply by coupling kCFA with the module-variant 0CFA.

In the following sections we will present kCFA, its modular version kCFA/m, a module-variant kCFA, and our proof that kCFA/m is a sound extension of the module-variant kCFA.

8.1 kCFA

Figure 5is kCFA, whose modular version will be presented in the next section. The rules are basically the same as in 0CFA, except that the nodes in the resulting control flow graph are indexed by the active-call sequence. An active-call sequence C is a sequence of call-sites that are currently active. For kCFA, the sequence's length is at most k; we keep only the most recent k call-sites. We write ϵ for the empty call-site sequence. The context information of the current active-call sequence is propagated by the relation "$C, \sigma \vdash e^l$," which indicates that expression e can be executed when the active-call sequence is C and the call environment is σ. A call environment maps variables to their active-call sequences at their bindings. Edge "$n \rightarrow m$" indicates that n may have the values of m. A single expression (or variable) in program has distinct instances in the analysis result, identified by different call-sequence indices.

For example, consider the (App_k) rule:

$$\frac{C, \sigma \vdash (e_1^a\ e_2^b)^l \quad a_C \rightarrow (\lambda x.e^{l'}, \sigma')}{\begin{array}{c} l \oplus C, \sigma'[x \mapsto l \oplus C] \vdash e^{l'} \\ x_{l \oplus C} \rightarrow b_C \quad l_C \rightarrow l'_{l \oplus C} \end{array}}\ (App_k)$$

For an application expression l at context (C, σ), suppose its function part a_C is $(\lambda x.e^{l'}, \sigma')$. The actual parameter b_C of the current context C is bound to the formal parameter: $x_{l \oplus C} \rightarrow b_C$. The incremented call-sequence context for x reflects the new call at l. The return value $l'_{l \oplus C}$ from the body becomes the value of the application: $l_C \rightarrow l'_{l \oplus C}$. The body expression's context reflects the new call at l: $l \oplus C, \sigma'[x \mapsto l \oplus C] \vdash e^{l'}$.

Applying the rules of Figure 5, we collect edges and relations until no more additions are possible. This process terminate because the number of nodes are finite for a given program.

kCFA is correct; it is straightforward to show by co-induction that a program's kCFA solution is a model for the program's uniform-kCFA [NN97].

8.2 kCFA/m: A Modularized Version of kCFA

Our modularized version kCFA/m is achieved similarly to 0CFA/m. Rules in the analysis phase $\mathcal{A}(M, \delta)$ are identical to those in the whole-program kCFA except that rule $(\mathrm{Dec}_{k\wp})$ is replaced by rule (Dec_k) that examines only the current module text. In the export phase $\mathcal{E}(\Delta, sig)$, we conservatively export all the edges that can be needed by subsequent modules. The starting point is the signature. For a variable x in the signature, x's bindings are needed to analyze subsequent modules because subsequent modules can directly refer to x, hence:

$$\frac{x \in sig}{x_\epsilon \in \mathit{Needed}}\ (Sig_k).$$

Note that every declared variable in modules has no call-site (ϵ) when it is bound to values. If the binding of variable x_C is needed to analyze subsequent modules ($x_C \in \mathit{Needed}$), then (1) its analysis result ($x_C \rightarrow (\lambda y.e^l, \sigma)$) is exported and (2) the bindings of the free variables in the

$$\begin{array}{llcl}
\mathit{Label} & l, a, b & \in & \mathit{Lab} \\
\mathit{Context} & C, \epsilon & \in & \mathit{Lab}^{\leq k} \\
\mathit{ConEnv} & \sigma & \in & \mathit{Var} \rightarrow \mathit{Context} \\
\mathit{Node} & n & ::= & x_C \mid l_C \mid (\lambda x.e^l, \sigma) \\
\mathit{Edge} & g & ::= & n \rightarrow n
\end{array}$$

$$\frac{x = e^l \in \wp}{x_\epsilon \rightarrow l_\epsilon \quad \epsilon, \sigma \vdash e^l} \ (\mathit{Dec}_k \wp)$$

$$\text{where } \sigma \supseteq \{ y \mapsto \epsilon \mid y \in FV(e) \}$$

$$\frac{C, \sigma \vdash x^l}{l_C \rightarrow x_{\sigma(x)}} \ (\mathit{Var}_k)$$

$$\frac{C, \sigma \vdash (\lambda x.e^{l_0})^l}{l_C \rightarrow \left(\lambda x.e^{l_0}, \sigma|_{FV(\lambda x.e^{l_0})} \right)} \ (\mathit{Lam}_k)$$

$$\frac{C, \sigma \vdash (e_1^{l_1}\ e_2^{l_2})^l}{C, \sigma \vdash e_1^{l_1} \quad C, \sigma \vdash e_2^{l_2}} \ (\mathit{Appd}_k)$$

$$\frac{C, \sigma \vdash (e_1^a\ e_2^b)^l \quad a_C \rightarrow (\lambda x.e^{l'}, \sigma')}{\begin{array}{c} l \oplus C, \sigma'[x \mapsto l \oplus C] \vdash e^{l'} \\ l_C \rightarrow l'_{l \oplus C} \quad x_{l \oplus C} \rightarrow b_C \end{array}} \ (\mathit{App}_k)$$

$$\text{where } l \oplus (l_n \cdots l_1) = \begin{cases} l l_n \cdots l_1 & \text{if } n < k; \\ l l_n \cdots l_2 & \text{if } n = k. \end{cases}$$

$$\frac{n_{C_1} \rightarrow m_{C_2} \quad m_{C_2} \rightarrow (\lambda x.e^l, \sigma)}{n_{C_1} \rightarrow (\lambda x.e^l, \sigma)} \ (\mathit{Tr}_k)$$

Figure 5: kCFA.

function values are needed to analyze subsequent modules ($\left\{ z_{\sigma(z)} \mid z \in \mathit{dom}(\sigma) \right\} \subseteq \mathit{Needed}$):

$$\frac{x_C \in \mathit{Needed} \quad x_C \rightarrow (\lambda y.e^l, \sigma) \in \Delta}{\begin{array}{c} x_C \rightarrow (\lambda y.e^l, \sigma) \\ \left\{ z_{\sigma(z)} \mid z \in \mathit{dom}(\sigma) \right\} \subseteq \mathit{Needed} \end{array}} \ (\mathit{ExportFn}_k)$$

8.3 kCFA/m Is Not Safe For kCFA

The result of kCFA/m does not always include that of kCFA.

Analysis phase. $\mathcal{A}(M,\delta)$ = edge-set Δ closed from module M and δ by the identical rules in kCFA, except that rule (Dec$_{k\wp}$) is replaced by rule (Dec$_k$):

$$\frac{x = e^l \in M}{x_\epsilon \to l_\epsilon \quad \epsilon, \sigma \vdash e^l}\ (Dec_k)$$
$$\text{where } \sigma \supseteq \{y \mapsto \epsilon \mid y \in FV(e)\}$$

Export phase. $\mathcal{E}(\Delta, sig)$ = exported-edge-set δ closed by the two rules:

$$\frac{x \in sig}{x_\epsilon \in Needed}\ (Sig_k)$$

$$\frac{x_C \in Needed \quad x_C \to (\lambda y.e^l, \sigma) \in \Delta}{\begin{array}{c} x_C \to (\lambda y.e^l, \sigma) \\ \left\{ z_{\sigma(z)} \mid z \in dom(\sigma) \right\} \subseteq Needed \end{array}}\ (ExportFn_k)$$

Figure 6: kCFA/m.

Example 4 Let us analyze the two modules in Example 3by 1CFA/m:

$$M_1 = \left(\begin{array}{l} \texttt{f = }(\lambda\texttt{x.}((\lambda\texttt{z.z}^5)^3\ \texttt{x}^4)^2)^1 \\ \texttt{g = }(\texttt{f}^7\ (\lambda\texttt{y.y}^9)^8)^6 \end{array},\ \{\texttt{f,g}\} \right)$$
$$M_2 = \left(\texttt{h = }(\texttt{f}^{11}\ (\lambda\texttt{w.w}^{13})^{12})^{10}\ ,\ \{\texttt{h}\} \right)$$

Analyzing the first module returns

$$\begin{array}{l} 7_\epsilon \to \texttt{f}_\epsilon \to 1_\epsilon \to \lambda\texttt{x.}((\lambda\texttt{z.z}^5)^3\ \texttt{x}^4)^2, \\ \texttt{g}_\epsilon \to 6_\epsilon \to 2_6 \to 5_2 \to \texttt{z}_2 \to 4_6 \to \texttt{x}_6 \to 8_\epsilon \to \lambda\texttt{y.y}^9, \\ 3_6 \to (\lambda\texttt{z.z}^5)_6, \end{array}$$

among which 1CFA/m exports only two edges

$$\texttt{f}_\epsilon \to \lambda\texttt{x.}((\lambda\texttt{z.z}^5)^3\ \texttt{x}^4)^2 \text{ and } \texttt{g}_\epsilon \to \lambda\texttt{y.y}^9.$$

Note that $z_2 \to \lambda\texttt{y.y}^9$ is not included. Analyzing the second module returns

$$\begin{array}{l} \texttt{h}_\epsilon \to 10_\epsilon \to 2_{10} \to 5_2 \to \texttt{z}_2 \to 4_{10} \to \texttt{x}_{10} \to 12_\epsilon \to \lambda\texttt{w.w}^{13}, \\ 3_{10} \to \lambda\texttt{z.z}^5. \end{array}$$

Thus 1CFA/m concludes that `h` can evaluate to only λ`w.w`. On the other hand, as discussed in Example 3, 1CFA for the whole-program concludes that `h` has λ`y.y` (a false-flow) as well as λ`w.w`. □

8.4 Module-Variant kCFA

kCFA/m is a correct analysis, which can be proven, not with respect to the original kCFA, but with respect to a module-variant kCFA.

$$
\begin{array}{rccl}
\mathit{ContMod} & (C,M) & \in & \mathit{Lab}^{\leq k} \times \mathit{Module} \\
\mathit{ContModEnv} & \sigma & \in & \mathit{Var} \to \mathit{ContMod} \\
\mathit{Value} & & & \mathit{Expr} \times \mathit{ContModEnv} \\
\mathit{Solution} & S & \in & (\mathit{Var} + \mathit{Label}) \times \mathit{ContMod} \\
 & & & \to \mathit{Value}
\end{array}
$$

$$
\begin{array}{lll}
(\mathit{var}) \quad S \models^{\sigma}_{C,M} x^l & \text{iff} & S(x,\sigma(x)) \subseteq S(l,(C,M)) \\
(\mathit{fn}) \quad S \models^{\sigma}_{C,M} (\lambda x.e^{l_0})^l & \text{iff} & (\lambda x.e^{l_0}, \sigma|_{FV(\lambda x.e^{l_0})}) \in S(l,(C,M)) \\
(\mathit{app}) \quad S \models^{\sigma}_{C,M} (e_1^{l_1}\ e_2^{l_2})^l & \text{iff} & S \models^{\sigma}_{C,M} e_1^{l_1} \quad \wedge \\
 & & S \models^{\sigma}_{C,M} e_2^{l_2} \quad \wedge \\
 & & \forall (\lambda x.e^{l_0}, \sigma') \in S(l_1,(C,M)) : \\
 & & \qquad S \models^{\sigma'[x \mapsto (l \oplus C, M)]}_{l \oplus C, M} e^{l_0} \quad \wedge \\
 & & \qquad S(l_2,(C,M)) \subseteq S(x,(l \oplus C, M)) \quad \wedge \\
 & & \qquad S(l_0,(l \oplus C, M)) \subseteq S(l,(C,M)) \\
(\mathit{con}) \quad S \models^{\sigma}_{C,M} c^l & \text{iff} & \text{true} \\
(\mathit{let}) \quad S \models^{\sigma}_{C,M} (\texttt{let}\ x = e_1^{l_1}\ \texttt{in}\ e_2^{l_2})^l & \text{iff} & S \models^{\sigma}_{C,M'} e_1^{l_1} \quad \wedge \\
 & & S \models^{\sigma[x \mapsto (C,M')]}_{C,M} e_2^{l_2} \quad \wedge \\
 & & S(l_1,(C,M')) \subseteq S(x,(C,M')) \quad \wedge \\
 & & S(l_2,(C,M)) \subseteq S(l,(C,M)) \\
 & & \text{where } x = e^{l_1} \text{ is in module } M'
\end{array}
$$

Figure 7: Module-variant kCFA = kCFA $\times$ the module-variant 0CFA.

Our module-variant kCFA is a "conjunctive" combination of the original kCFA and the module-variant 0CFA (in Figure 7). A context of the module-variant kCFA is a pair containing an active-call sequence of length up to k (as in kCFA) *and* a module index (as in module-variant 0CFA). The way to manipulate the call sequence follows that of kCFA, and the way to manipulate the module index follows that of the module-variant 0CFA. For example, in (*app*), if the context of a call-site labeled l is a pair containing a call sequence C and module M, then the context of the called function's body is a pair, $l \oplus C$ (as in kCFA) and M (as in the module-variant 0CFA). This module-variant kCFA is achieved straightforwardly; a general recipe for combining two CFAs is illustrated in Appendix A. Because such a combined CFA is also an instance of the infinitary CFA of Nielson and Nielson [NN97], the module-variant kCFA's correctness follows from Theorem 4.1 in [NN97].

For the input program $\wp$ that consists of modules $M_1, \cdots, M_n$, its module-variant kCFA is defined [NN97] as the least S such that $S \models^{\emptyset}_{\epsilon,\varepsilon} \wp$, where $\emptyset$ is the empty environment, ϵ is the empty context sequence, and ε is a dummy module index for the whole program.

8.5 kCFA/m Is Safe For Module-Variant kCFA

kCFA/m is a sound extension of the module-variant kCFA. The proof technique is exactly the same as in the case for 0CFA/m. We prove by showing that there exists a solution S of the module-variant kCFA that is covered by the result of kCFA/m. Definition 5defines a solution S that is covered by the result of kCFA/m, and Theorem 2asserts that the S is a solution of the module-variant kCFA.

Definition 3 (x_C reaches M via M') *Let Δ_M be the solved edges in analyzing module M by kCFA/m.*

- *Variable x_C reaches M_n via M_0 if and only if $M_0 = M_n$ and $x_C \in \Delta_{M_n}$, or there exists a path $M_0 \sqsubset M_1 \sqsubset \cdots \sqsubset M_n$ such that for all $0 \le i < n$, $x_C \in \mathit{Needed}_{M_i}$.*
- *Environment σ reaches M if and only if, for all $x \mapsto (C, M') \in \sigma$, x_C reaches M via M'*

Definition 4 ($|\sigma|$) *For a given environment σ of the module-variant kCFA, the corresponding environment $|\sigma|$ for kCFA/m is $\{x \mapsto C \mid x \mapsto (C, M) \in \sigma\}$.*

Definition 5 ($|\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)|$) *Let $\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)$ be the result edges from analyzing modules $M_1, \cdots, M_n$ by kCFA/m. Its corresponding form $|\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)|$ in the solution space for the module-variant kCFA is defined as:*

$$|\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)|(n, (C, M)) = \left\{ (\lambda x.e^l, \sigma) \;\middle|\; \begin{array}{l} n_C \to (\lambda x.e^l, |\sigma|) \in \Delta_M, \\ \sigma \textit{ reaches } M,\ \mathit{dom}(\sigma) = FV(\lambda x.e^l) \end{array} \right\}.$$

Fact. By definition, $|\mathit{Sol}_{k\mathrm{CFA/m}}(\cdot)|$ is "covered by" $\mathit{Sol}_{k\mathrm{CFA/m}}(\cdot)$: $(\lambda x.e^l, \sigma) \in |\mathit{Sol}_{k\mathrm{CFA/m}}(\cdot)|$ $(n, (C, M))$ implies that $n_C \to (\lambda x.e^l, |\sigma|) \in \mathit{Sol}_{k\mathrm{CFA/m}}(\cdot)$.

Theorem 2 (Correctness of k-CFA/m) *Let program $\wp$ consist of modules $M_1, \cdots, M_n$. $|\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)| \models^{\emptyset}_{\epsilon,\varepsilon} \wp$ holds, where $\emptyset$ is the empty environment, ϵ is the empty context sequence, and ε is a dummy module index for the whole program.*

Proof. Let $S = |\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)|$. $F(Q)$ gives us a set of left-hand-side judgments asserted by the rules of Figure 7assuming that judgments in Q hold. If we find a set Q of judgments such that $(S \models^{\emptyset}_{\epsilon,\varepsilon} \wp) \in Q$ and $Q \subseteq F(Q)$, then by the co-induction principle [MT91], Q is included in the greatest fixed point of F and $S \models^{\emptyset}_{\epsilon,\varepsilon} \wp$ holds.

Therefore, the module-variant kCFA's solution, which is defined as the least X such that $X \models^{\emptyset}_{\epsilon,\varepsilon} \wp$, is included in the modularized solution $\mathit{Sol}_{k\mathrm{CFA/m}}(M_1, \cdots, M_n)$. $\square$

9 Conclusion

Modular analyses, which are necessary in practice for analyzing large-scale programs, are usually designed *after* whole-program analyses' cost-accuracy balance is assured by extensive tests against realistic programs. A cost-effective, practical program analysis is achieved by first finding an effective whole-program analysis and then modularizing it. This practice is observed also in literature; only recently have modular versions of particular flow analyses been reported [CRL99, CmWH00, FF99].

However, deriving a correct modular version from a given whole-program analysis has received little attention. For example, the abstract interpretation framework [CC77, CC92] does not yet cover the practice of designing modular analyses; it assumes that the whole-program text is available a priori. Within the type system framework, modular analysis is considered free [TJ94, Ban97, PS92, PL99, TJ92]. This is because the type environment, which has the analysis solution for the free variables (i.e., other modules), are manifest in the typing rules. Also, the notion of principal types and type polymorphism coincide very well with the modular analysis model [Ban97]. But the type-based modular analyses are limited to typed languages.

In this article we reported that, contrary to our expectation, modularizing kCFA (for any k in the framework of incremental analysis) makes the resulting analysis (kCFA/m) more accurate than the original kCFA. Then, as an answer to the consequent problem of proving the correctness of kCFA/m, we showed that the proof, which is impossible with respect to the original whole-program kCFA, is possible with respect to a module-variant kCFA, a whole-program kCFA that was polyvariant at module-level.

Because kCFAs are the bases of almost all analyses for higher-order programs, our results (in addition to the safe modular kCFA/m per se) can be seen as a clarification of possible problems in designing a correct modular analysis from a whole-program analysis; and as a hint of using the *module-variant* whole-program analysis in proving the correctness of modular static analyses.

The componential set-based analysis [FF99] is orthogonal to our result; its constraint simplification algorithms for each component (or module) preserve the accuracy of the original whole-program analysis, hence we can just use their simplification algorithms in kCFA/m to reduce the size of the exported edges.

Reference

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, June 1994.

[Ban97] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *ACM SIGPLAN International Conference on Functional Programming*, pages 1–10, 1997.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[CmWH00] Ben-Chung Cheng and Wen mei W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, June 2000.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, January 1999.

[FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, June 1997.

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345, January 1997.

[PL99] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 276–290, January 1999.

[PS92] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1992.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, August 1972.

[Rey98a] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998. [Rey72]'s reprint.

[Rey98b] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, December 1998.

[Shi88] Olin Shivers. Control flow analysis in scheme. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[TJ92] Jean-Piere Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TJ94] Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *Lecture Notes in Computer Science*, volume 789, pages 224–243. Springer-Verlag, proceedings of the theoretical aspect in computer science edition, 1994.

Appendix

A Coupled CFA

For two CFAs A and B defined as an instance of the infinitary CFA, the coupled CFA of A and B is defined as follows. The context is the pair of the contexts of A and B, and the label distinguisher is also the pair of those of A and B:

$$\begin{array}{rcl} \widehat{Mem} & \triangleq & \widehat{Mem}_A \times \widehat{Mem}_B \\ \widehat{MEnv} & \triangleq & Var \to (\widehat{Mem}_A \times \widehat{Mem}_B) \\ \widehat{MC} & \triangleq & \widehat{MC}_A \times \widehat{MC}_B. \end{array}$$

Then the projection function π is defined by those of A and B:

$$\pi(m, \sigma)] \triangleq (\pi_A(m^A, \sigma^A), \pi_B(m^B, \sigma^B))$$

where $(m_1, m_2)^A \triangleq m_1$, $(m_1, m_2)^B \triangleq m_2$, and

$$\begin{array}{l} \sigma^A \triangleq \left\{x \mapsto m^A \mid x \mapsto m \in \sigma\right\} \\ \sigma^B \triangleq \left\{x \mapsto m^B \mid x \mapsto m \in \sigma\right\}. \end{array}$$

The instantiators have to ensure the conditions of the instantiators of A and B; that is, an instantiator $\mathcal{I}(m, \sigma, e^l; m')$ is defined as:

$$\begin{array}{rcl} \mathcal{I}(m, \sigma, e^l; m') & \triangleq & \mathcal{I}_A(m^A, \sigma^A, e^l; m'^A) \\ & \wedge & \mathcal{I}_B(m^B, \sigma^B, e^l; m'^B). \end{array}$$

The coupled CFA is an instance of the infinitary CFA, and is more accurate than both A and B.