모듈을 지원하는 함수 환류 분석*
(Modularized Control Flow Analyses)

어우세 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 $k$CFA; we present safe modularized $k$CFAs, 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:

\[
\begin{align*}
  f &= \lambda x.x \\
  g &= f \lambda y.y \\
  h &= f \lambda z.z
\end{align*}
\]

*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 \( \lambda y \cdot y \) and \( \lambda z \cdot z \). This information is propagated back to the call sites that we conclude \( h \) has \( \lambda y \cdot y \) (a false-flow) as well as \( \lambda z \cdot z \).

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 \( \lambda x \cdot x \) and \( g \) has \( \lambda y \cdot y \). Analyzing the second fragment with this information concludes that \( h \) has only \( \lambda z \cdot z \).

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 \( k \) CFA.
- Safe two results are true for any \( k \) in \( k \) CFA.

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 3 we present its modular version 0CFA/m in Section 4, and show in Section 5 that it is not sound with respect to the original 0CFA. In Section 6 we define a module-variant 0CFA and prove in Section 7 that the modular version is correct for this module-variant 0CFA. In Section 8, we show that the same holds for \( k \) CFA in general: we show that modularized version \( k \) CFA/m is not sound with respect to \( k \) CFA yet is safe with respect to the whole-program module-variant \( k \) CFA. 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 \( M' \sqsubseteq M \), 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
modules may need. This exported results will be used by modules that depend on the current one. For a given module $M = (\text{decl}, \text{sig})$, let the analysis phase be $A(M, \delta)$ with $A : \text{Module} \times \text{Results} \rightarrow \text{Results}$. The second input $\delta$ is the exported results from the modules that $M$ depends on. Let the result of this analysis be $\Delta$. From $\Delta$, we export only some parts of it that subsequent modules may need. Let this export phase be $E(\Delta, \text{sig})$ with $E : \text{Results} \times \text{Signature} \rightarrow \text{Results}$. For a program that consists of modules $M_1, \ldots, M_n$, each module $M_i$'s analysis result $\Delta_i$ and its exported set $\delta_i$ (in Figure 1) are defined as $\Delta_i = A\left(M_i, \bigcup_{M_j \subseteq M_i} \delta_j\right)$ and $\delta_i = E(\Delta_i, \text{sig}_i)$, where $\text{sig}_i$ is the signature of $M_i$. The final analysis result $Sol(M_1, \ldots, M_n)$ 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
nodes are finite for a given program. Edge “n → λx.e” in the final result indicates that n may evaluate into (or, is bound to) function λx.e 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 A(M, δ) 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 “∈ M or δ” means “is a sub-expression in either module M or a node of δ.” In the export phase E(Δ, 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:

\[
\frac{x \in \text{sig}}{x \in \text{Needed}(\text{Sig})}.
\]

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

\[
\frac{x \in \text{Needed} \quad x \rightarrow \lambda y.e \in \Delta}{\text{FV}(\lambda y.e) \subseteq \text{Needed}} \quad \frac{\text{Needed}}{x \rightarrow \lambda y.e} \quad \text{(ExportFn)}.
\]
Analysis phase. \( \mathcal{A}(M, \delta) = \text{edge-set } \Delta \text{ closed from module } M \text{ and imported edges } \delta \text{ by the five rules:} \)

\[
\begin{align*}
\frac{x = e' \in M}{x \to l} \quad & (Dec) \\
\frac{l_1 \to \lambda x.e_b}{l \to l_0 \quad x \to l_2} \quad & (App) \\
\frac{x^l \in M \text{ or } \delta}{l \to x} \quad & (Var) \\
\frac{n \to m \quad m \to \lambda x.e'}{n \to \lambda x.e'} \quad & (Tr) \\
\frac{\lambda x.e_b}{l \to \lambda x.e_b} \quad & (Lam)
\end{align*}
\]

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

\[
\begin{align*}
\frac{x \in \text{sig}}{x \in \text{Needed}} \quad & (Sig) \\
\frac{x \in \text{Needed} \quad x \to \lambda y.e' \in \Delta \quad FV(\lambda y.e') \subseteq \text{Needed}}{x \to \lambda y.e'} \quad & (ExportFn)
\end{align*}
\]

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 1 and its modular analysis:

\[
M_1 = \begin{pmatrix}
\text{f} &= (\lambda x.x^2)^1 \\
\text{g} &= \{\text{f}^4 (\lambda y.y^6)^5\}^3, \{\text{f}, \text{g}\}
\end{pmatrix}
\]

\[
M_2 = \begin{pmatrix}
\text{h} &= (\lambda x.z^{10})^7, \{\text{h}\}
\end{pmatrix}
\]

If we analyze the whole program by 0CFA, the result includes a false-flow edge \( h \to \lambda y.y \)
Because
\[ h \rightarrow (f \; \lambda z . z) \rightarrow x \quad \rightarrow \lambda z . z \]
\[ \downarrow \lambda y . y \]

(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
\[ f \rightarrow 1 \rightarrow \lambda x . x^2, \]
\[ g \rightarrow 3 \rightarrow 2 \rightarrow x \rightarrow 5 \rightarrow \lambda y . y^6, \]
\[ 6 \rightarrow y, \]
among which 0CFA/m exports only two edges:
\[ f \rightarrow \lambda x . x^2 \quad \text{and} \quad g \rightarrow \lambda y . y^6. \]

Note that \( x \rightarrow \lambda y . y \) is not included. With the exported edges from the first module, analyzing the second module returns
\[ h \rightarrow 7 \rightarrow 2 \rightarrow x \rightarrow 9 \rightarrow \lambda z . z^{10}. \]
The false-flow edge \( h \rightarrow \lambda y . y \) is absent. \( \square \)

Because 0CFA/m is more accurate than the whole 0CFA, the correctness relation between CFAs is
\[
\text{Semantics} \subseteq 0\text{CFA} \quad \text{(correctness of 0CFA)}
\]
\[ 0\text{CFA/m} \subseteq 0\text{CFA} \quad \text{(0CFA/m is not correct w.r.t. 0CFA)} \]
where \( A \subseteq B \) means that the result of \( B \) includes that of \( A \). In order to prove the correctness of 0CFA/m
\[
\text{Semantics} \subseteq 0\text{CFA/m}
\]
we want to find an analysis \( * \) which is not only safe with respect to the semantics (Semantics \( \subseteq * \subseteq 0\text{CFA/m} \)) but for which it is easy to prove that \( * \subseteq 0\text{CFA/m} \).

It seems that such an analysis \( * \) must be polyvariant at module-level. In Example 2, there are two applications whose function part is \( f \): \( (f \; \lambda y . y) \) in \( M_1 \) and \( (f \; \lambda z . z) \) in \( M_2 \). In 0CFA, both application nodes link to \( x \): \( (f \; \lambda y . y) \rightarrow x \) and \( (f \; \lambda z . z) \rightarrow x \). The former \( x \) is bound to \( \lambda y . y \) and the latter \( x \) to \( \lambda z . z \). 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 \( * \). 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 Nielsen and Nielsen [NN97].
### 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 $\varphi$ consists of modules $M_1, \ldots, M_n$” 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 “let $x = e_1$ in let $y = e_2$ in let $z = x$ in $c$.” We assume that all the declara-
tions in a merged let-expression/ have their associated module names.

Judgment “\(S \models_M e\)” means solution \(S\) respects the situation that evaluating expressions of module \(M\) under environment \(\sigma\) needs to evaluate \(e\). Environment \(\sigma\) 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 \(\varphi\) that consists of modules \(M_1, \ldots, M_n\), its module-variant 0CFA is defined \([NN97]\) as the least \(S\) such that \(S \models_\varnothing \varphi\) where \(\varnothing\) is the empty module-context environment and \(\varepsilon\) is a dummy module index for the whole program.

Let’s consider the rules, case by case.

Case (var). If a variable is necessary \((S \models_M x^l)\) 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 \((S \models_M (\lambda x.e^b)^l)\) for module \(M\) then the analysis result \(S(l, M)\) at the label must include it.

Case (app). If an application is necessary \((S \models_M (e_{11}^l e_{22}^l)^l)\) 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: \(S \models_M e_{11}^l \land S \models_M e_{22}^l\). For each function \(\forall (\lambda x.e^b, \sigma') \in S(l_1, M)\) that can be called, (1) its formal parameter \(x\) and its body \(e^b\) have the same module context: \(S \models_M^{\sigma: \sigma': M} e^b\), (2) values flow from argument expression \(e_{12}^l\) to the formal parameter \(x\): \(S(l_2, M) \subseteq S(x, M)\), and (3) values flow from body expression \(e^b\) to the call expression \((e_{11}^l e_{22}^l)^l\): \(S(l_0, M) \subseteq S(l, M)\). 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 “\(x=e^l\)” is a declaration in a module, we have to use this module context for the variable \(x\) and its definition \(e^l\).

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:

\[
\text{\text{Mem}} \triangleq \text{Module} \\
\text{\text{MC}} \triangleq \text{Module}
\]

Hence the projection is defined as \(\pi(M, \sigma) \triangleq M\). 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 \((M)\):

\[
\text{\text{\text{T}_{\text{app}}}}^{\text{fn}} (\sigma, M, \sigma', M_0, e^l; M_0, M_x) \triangleq M_0 = M_x = M.
\]

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 \( (M') \):

\[
\mathcal{I}_{le} (\sigma, M, (\text{let } x = e_1 \text{ in } e_2); M_1, M_2, M_x) \\
\triangleq (M = M_2) \land (M_x = M_1 = M') \\
\text{where } x = e_1 \in M'.
\]

For other cases, instantiators do not change the context:

\[
\mathcal{I}_{fn} (\sigma, M, (\lambda x. e_2); M_0) \triangleq M_0 = M \\
\mathcal{I}_{app} (\sigma, M, (e_1 e_2); M_1, M_2) \triangleq M_1 = M_2 = M.
\]

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 2 defines a solution \( S \) that is covered by the result of 0CFA/m, and Theorem 1 asserts that \( S \) is a solution of the module-variant 0CFA. We write \( \text{Needed}_M \) to denote the \( \text{Needed} \) set of the exporting phase in analyzing module \( M \) by 0CFA/m (See Figure 3).

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

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

**Definition 2** \( (\lvert \text{SOL}_{CFA/m}(M_1, \ldots, M_n) \rvert) \) Let \( \text{SOL}_{CFA/m}(M_1, \ldots, M_n) \) be the result edges from analyzing modules \( M_1, \ldots, M_n \) by 0CFA/m. Its corresponding form \( \lvert \text{SOL}_{CFA/m}(M_1, \ldots, M_n) \rvert \) in the solution space for the module-variant 0CFA is defined as:

\[
\lvert \text{SOL}_{CFA/m}(M_1, \ldots, M_n) \rvert(n, M) = \left\{ (\lambda x.e^i, \sigma) \mid n \rightarrow \lambda x.e^i \in \Delta_M, \sigma \text{ reaches } M, \text{dom}(\sigma) = FV(\lambda x.e^i) \right\}
\]

where \( \Delta_M \) is the 0CFA/m’s solution for module \( M \).

**Fact.** By definition, \( \lvert \text{SOL}_{CFA/m}(\cdot) \rvert \) is “covered by” \( \text{SOL}_{CFA/m}(\cdot) \): \( (\lambda x.e^i, \cdot) \in \lvert \text{SOL}_{CFA/m}(\cdot) \rvert \) \( (n, M) \) implies \( n \rightarrow \lambda x.e^i \in \text{SOL}_{CFA/m}(\cdot) \).

**Theorem 1** (Correctness of 0CFA/m) Let program \( \varphi \), as a let-expression, consist of modules \( M_1, \ldots, M_n \). \( \lvert \text{SOL}_{CFA/m}(M_1, \ldots, M_n) \rvert \models^0 \varphi \) holds, where \( \varphi \) is the empty module-context environment and \( \varepsilon \) is a dummy module index for the whole program.
Proof. Let \( S \neq j \). Judgment \( S \models_M e \) holds if it is included in the greatest fixed point of the function 
\[
F : \text{Judgments} \rightarrow \text{Judgments}
\]
derived from Figure 4[NN97]. \( F(Q) \) gives us a set of left-hand side judgments asserted by the rules of Figure 4 assuming that judgments in \( Q \) hold. If we find a set \( Q \) of judgments such that \( (S \models^0_M \emptyset) \in Q \) and \( Q \subseteq F(Q) \), then by the co-induction principle [MT91], \( Q \) is included in the greatest fixed point of \( F \) and \( S \models^0_M \emptyset \) holds.

Therefore, the module-variant 0CFA’s solution, which is defined as the least \( X \) such that \( X \models^0_M \emptyset \), is included in the modularized solution \( Sol_{0CFA/m}(M_1, \ldots, M_n) \). \( \Box \)

The correctness relation between CFAs becomes:
\[
\text{Semantics} \subseteq \text{module-variant 0CFA} \subseteq 0CFA/m \subseteq 0CFA.
\]

8 Modularizing \( k \)CFA

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

Example 3 Consider the following two modules.
\[
M_1 = \left( \begin{array}{c}
\text{f} = (\lambda x. (\lambda z. z) \ x) \\
\text{g} = \text{f} \ \lambda y. y \\
\end{array} \right), \{\text{f,g}\}
\]
\[
M_2 = \left( \begin{array}{c}
\text{h} = \text{f} \ \lambda w. w \\
\end{array} \right)
\]

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 \( \lambda z. z \) inside \( \text{f} \), which occurred for \( \text{g} \) and \( \text{h} \). That is, the two instances of \( z \) are not distinguished. Hence, it concludes that \( \text{h} \) can evaluate to both \( \lambda w. w \) and \( \lambda y. y \). However, analyzing modules separately, we can export from the first module \( \text{f} \rightarrow (\lambda x. (\lambda z. z) \ x) \) and \( \text{g} \rightarrow \lambda y. y \), and conclude that \( \text{h} \) can evaluate to only \( \lambda w. w \). For any \( k \) in \( k \)CFA, we can show similar counter-example where its modular version is more accurate than \( k \)CFA. \( \Box \)

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

In the following sections we will present \( k \)CFA, its modular version \( k \)CFA/m, a module-variant \( k \)CFA, and our proof that \( k \)CFA/m is a sound extension of the module-variant \( k \)CFA.
8.1 $k$CFA

Figure 5 is $k$CFA, 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 $k$CFA, the sequence’s length is at most $k$; we keep only the most recent $k$ call-sites. We write $\epsilon$ for the empty call-site sequence. The context information of the current active-call sequence is propagated by the relation “$C, \sigma \vdash e$,” which indicates that expression $e$ can be executed when the active-call sequence is $C$ and the call environment is $\sigma$. 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:

$$
\frac{C, \sigma \vdash (e_1 \bowtie e_2)^l \quad a_C \rightarrow (\lambda x_1.e_1', \sigma')}{l \oplus C, \sigma'[x \mapsto l \oplus C] \vdash e'}(App_k)
$$

For an application expression $l$ at context $(C, \sigma)$, suppose its function part $a_C$ is $(\lambda x_1.e_1', \sigma')$. 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 $l$: $l \oplus C, \sigma'[x \mapsto l \oplus C] \vdash e'$.

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

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

8.2 $k$CFA/m: A Modularized Version of $k$CFA

Our modularized version $k$CFA/m is achieved similarly to 0CFA/m. Rules in the analysis phase $A(M, \delta)$ are identical to those in the whole-program $k$CFA except that rule (Dec$_k$) is replaced by rule (Dec$_k^*$) that examines only the current module text. In the export phase $E(\Delta, sig)$, 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:

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

Note that every declared variable in modules has no call-site ($\epsilon$) when it is bound to values. If the binding of variable $x_C$ is needed to analyze subsequent modules ($x_C \in Needed$), then (1) its analysis result $(x_C \rightarrow (\lambda y.e'_1, \sigma))$ is exported and (2) the bindings of the free variables in the
Label $l, a, b \in Lab$

Context $C, \epsilon \in Lab \leq k$

ConEnv $\sigma \in \text{Var} \rightarrow \text{Context}$

Node $n : = x_C \mid l_C \mid (\lambda x. e', \sigma)$

Edge $g : = n \rightarrow n$

\[
\frac{x = e' \in \emptyset}{x_e \rightarrow l_e \in \epsilon, \sigma \vdash e'} (Dec_{k, \emptyset})
\]

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

\[
\frac{C, \sigma \vdash e'}{l_C \rightarrow x_{\sigma(x)} \ (Var_k)}
\]

\[
\frac{C, \sigma \vdash (\lambda x. e_k)' \quad l_C \rightarrow (\lambda x. e_k_0, \sigma_{FV(\lambda x. e_k)})}{(Lam_k)}
\]

\[
\frac{C, \sigma \vdash e_1' \quad C, \sigma \vdash e_2'}{C, \sigma \vdash (e_1', e_2')'} (Appd_k)
\]

\[
\frac{C, \sigma \vdash (e_1', e_2')'}{a_C \rightarrow (\lambda x. e', \sigma')} (App_k)
\]

\[
\frac{l \oplus C, \sigma'[x \mapsto l \oplus C] \vdash e'}{l_C \rightarrow \nu l_{\oplus C} \quad x_{l \oplus C} \rightarrow b_C \ (Tr_k)}
\]

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

\[
\frac{n_{C_1} \rightarrow m_{C_3} \quad m_{C_3} \rightarrow (\lambda x. e', \sigma)}{n_{C_1} \rightarrow (\lambda x. e', \sigma)}
\]

Figure 5: $kCFA$.

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

\[
\frac{x_C \in \text{Needed}}{x_C \rightarrow (\lambda y. e', \sigma) \in \Delta \ (ExportFn_k)}
\]

\[
\left\{ z_{\sigma(z)} \mid z \in \text{dom}(\sigma) \right\} \subseteq \text{Needed}
\]

8.3 $kCFA/m$ Is Not Safe For $kCFA$

The result of $kCFA/m$ does not always include that of $kCFA$. 
**Analysis phase.** \( A(M, \delta) = \) edge-set \( \Delta \) closed from module \( M \) and \( \delta \) by the identical rules in \( k \text{CFA} \), except that rule \((\text{Dec}_k^\epsilon)\) is replaced by rule \((\text{Dec}_k)\):

\[
\frac{x = e^l \in M}{x_{\epsilon} \rightarrow I_{x_{\epsilon}} \quad \epsilon, \sigma \vdash e} \quad (\text{Dec}_k)
\]

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

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

\[
\frac{x \in \text{sig}}{x_{\epsilon} \in \text{Needed}} \quad (\text{Sig}_k)
\]

\[
\frac{x_C \in \text{Needed}}{x_C \rightarrow (\lambda y.e^l, \sigma)} \quad (\text{ExportFn}_k)
\]

\[
\left\{ z_{\sigma(z)} \mid z \in \text{dom}(\sigma) \right\} \subseteq \text{Needed}
\]

Figure 6: \( k \text{CFA}/m \).

**Example 4** Let us analyze the two modules in Example 3 by \( 1 \text{CFA}/m \):

\[
M_1 = \begin{pmatrix}
    f = (\lambda x.((\lambda z. z^2)^3 x^4)^2)^1, & \{f,g\} \\
    g = (f^7 (\lambda y.y^9)^8)^6 & \\
\end{pmatrix}
\]

\[
M_2 = \begin{pmatrix}
    h = (f^{11} (\lambda w.w^{13})^{12})^{10}, & \{h\} \\
\end{pmatrix}
\]

Analyzing the first module returns

\[
7_{e} \rightarrow f_{e} \rightarrow 1_{e} \rightarrow \lambda x.((\lambda z. z^5)^3 x^4)^2,
\]

\[
g_{e} \rightarrow 6_{e} \rightarrow 2_{e} \rightarrow 5_{e} \rightarrow z_{e} \rightarrow 4_{e} \rightarrow x_{e} \rightarrow 8_{e} \rightarrow \lambda y.y^9,
\]

\[
3_{e} \rightarrow (\lambda z. z^5)_{e}.
\]

among which \( 1 \text{CFA}/m \) exports only two edges

\[
f_{e} \rightarrow \lambda x.((\lambda z. z^5)^3 x^4)^2 \text{ and } g_{e} \rightarrow \lambda y.y^9.
\]

Note that \( z_{e} \rightarrow \lambda y.y^9 \) is not included. Analyzing the second module returns

\[
h_{e} \rightarrow 10_{e} \rightarrow 2_{10} \rightarrow 5_{2} \rightarrow z_{2} \rightarrow 4_{10} \rightarrow x_{10} \rightarrow 12_{e} \rightarrow \lambda w.w^{13},
\]

\[
3_{10} \rightarrow \lambda z. z^5.
\]

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

### 8.4 Module-Variant \( k \text{CFA} \)

\( k \text{CFA}/m \) is a correct analysis, which can be proven, not with respect to the original \( k \text{CFA} \), but with respect to a module-variant \( k \text{CFA} \).
Our module-variant $k$CFA is a "conjunctive" combination of the original $k$CFA and the module-variant 0CFA (in Figure 7). A context of the module-variant $k$CFA is a pair containing an active-call sequence of length up to $k$ (as in $k$CFA) and a module index (as in module-variant 0CFA). The way to manipulate the call sequence follows that of $k$CFA, 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 $k$CFA) and $M$ (as in the module-variant 0CFA). This module-variant $k$CFA 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 $k$CFA's correctness follows from Theorem 4.1 in [NN97].

For the input program $\varphi$ that consists of modules $M_1, \ldots, M_n$, its module-variant $k$CFA is defined [NN97] as the least $S$ such that $S \models^0_{\epsilon, \emptyset, \varphi}$, where $\emptyset$ is the empty environment, $\epsilon$ is the empty context sequence, and $\varphi$ is a dummy module index for the whole program.
8.5 $k\text{CFA/m}$ Is Safe For Module-Variant $k\text{CFA}$

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

**Definition 3** ($x$ reaches $M$ via $M'$) Let $\Delta_M$ be the solved edges in analyzing module $M$ by $k\text{CFA/m}$.

- Variable $x$ reaches $M_{n}$ via $M_{0}$ if and only if $M_{0} = M_{n}$ and $x \in \Delta_{M_n}$, or there exists a path $M_0 \subseteq M_1 \subseteq \cdots \subseteq M_n$ such that for all $0 \leq i < n$, $x \in \text{Needed}_{M_i}$.

- Environment $\sigma$ reaches $M$ if and only if, for all $x \mapsto (C, M') \in \sigma$, $x$ reaches $M$ via $M'$.

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

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

$$|\text{Sol}_{k\text{CFA/m}}(M_1, \ldots, M_n)|(n, (C, M)) = \left\{ (\lambda x.e', \sigma) \mid n_C \rightarrow (\lambda x.e', |\sigma|) \in \Delta_M, \right.$$

$$\left. \sigma \text{ reaches } M, \text{ dom}(\sigma) = FV(\lambda x.e') \right\}.$$

**Fact.** By definition, $|\text{Sol}_{k\text{CFA/m}}(\cdot)|$ is “covered by” $\text{Sol}_{k\text{CFA/m}}(\cdot)$: $(\lambda x.e', \sigma) \in |\text{Sol}_{k\text{CFA/m}}(\cdot)|(n, (C, M))$ implies that $n_C \rightarrow (\lambda x.e', |\sigma|) \in \text{Sol}_{k\text{CFA/m}}(\cdot)$.

**Theorem 2** (Correctness of $k\text{-CFA/m}$) Let program $\emptyset$ consist of modules $M_1, \ldots, M_n$. $|\text{Sol}_{k\text{CFA/m}}(M_1, \ldots, M_n)| \models_{e, \varepsilon} \emptyset$ holds, where $\emptyset$ is the empty environment, $\varepsilon$ is the empty context sequence, and $e$ is a dummy module index for the whole program.

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

Therefore, the module-variant $k\text{CFA}$’s solution, which is defined as the least $X$ such that $X \models_{e, \varepsilon} \emptyset$, is included in the modularized solution $\text{Sol}_{k\text{CFA/m}}(M_1, \ldots, M_n)$. □

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 $k$CFA (for any $k$ in the framework of incremental analysis) makes the resulting analysis ($k$CFA/m) more accurate than the original $k$CFA. Then, as an answer to the consequent problem of proving the correctness of $k$CFA/m, we showed that the proof, which is impossible with respect to the original whole-program $k$CFA, is possible with respect to a module-variant $k$CFA, a whole-program $k$CFA that was polynvariant at module-level.

Because $k$CFAs are the bases of almost all analyses for higher-order programs, our results (in addition to the safe modular $k$CFA/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 componental 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 $k$CFA/m to reduce the size of the exported edges.

Reference


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 $B$:

\[ \overline{Mem} \triangleq \overline{Mem}_A \times \overline{Mem}_B \]
\[ \overline{MEnv} \triangleq \text{Var} \to (\overline{Mem}_A \times \overline{Mem}_B) \]
\[ \overline{MC} \triangleq \overline{MC}_A \times \overline{MC}_B. \]

Then the projection function $\pi$ 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

\[ \sigma^A \triangleq \{ x \mapsto m^A \mid x \mapsto m \in \sigma \} \]
\[ \sigma^B \triangleq \{ x \mapsto m^B \mid x \mapsto m \in \sigma \}. \]

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

\[ I(m, \sigma, e^i; m') \triangleq I_A(m^A, \sigma^A, e^i; m^A) \]
\[ \land I_B(m^B, \sigma^B, e^i; m^B). \]

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

\[ \overline{Mem} \triangleq \overline{Mem}_A \times \overline{Mem}_B \]
\[ \overline{MEnv} \triangleq \text{Var} \to (\overline{Mem}_A \times \overline{Mem}_B) \]
\[ \overline{MC} \triangleq \overline{MC}_A \times \overline{MC}_B. \]