
ROPAS

Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
2000-4
February 3, 2000

Modularization of 0-CFA Makes It Polyvariant

Oukseh Lee and Kwangkeun Yi
{cookcu; kwang}@ropas.kaist.ac.kr

Abstract

This article shows that (1) deriving a modular version (in the framework of incremental analysis) from a whole-program CFA makes the resulting analysis polyvariant at module-level, (2) if the original whole-program CFA was less accurate than or incomparable to module-level polyvariant analyses, then the correctness of its modular version may not be proven in general with respect to the original CFA, and (3) a convenient stepstone to prove the correctness of modular analyses is a whole-program CFA that is polyvariant at module-level. Our result 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.

1 Introduction

Modular analyses are necessary for analyzing large, 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.

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

In deriving a modular version from a whole analysis, however, it is not obvious to check whether the modular version is sound with respect to the original whole version. We tried to derive a modular version of the standard 0-CFA [Shi88] and observed that modularization of 0-CFA makes it *unsound* with respect to the original 0-CFA. That is, our derived modular analyses give us more accurate result than 0-CFA does.

Example 1 Consider the following two modules:

$$\texttt{f = }\lambda\texttt{x.x} \quad \texttt{g = f }\lambda\texttt{y.y} \qquad \text{and} \qquad \texttt{h = g (f }\lambda\texttt{z.z).}$$

0-CFA concludes that `h` can be evaluated to λ`z.z` and λ`y.y`, because 0-CFA estimates that `h` can be evaluated to `x` and that `x` can be evaluated to both functions from "`f` λ`y.y`" and "`f` λ`z.z`". On the other hand, separate analyses of the modules (from left to right) give more accurate result. From the result of the left module, it is natural to bring only information that `f` can be evaluated to λ`x.x` and `g` can be λ`y.y`. Then the analysis result of the second module determines that `h` can be evaluated to only λ`z.z`. □

We have found that we can prove the safety of the modular version with respect to a whole-program CFA that is polyvariant at module-level. We call this analysis "module-variant CFA."

[0]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

After we describe the model for our modular analysis in Section 2, we describe the language and its 0-CFA in Section 3. We then present, in Section 4, a naive modular version of 0-CFA, show that it is not sound with respect to the original 0-CFA, introduce a module-variant CFA, and prove the correctness of the naive modular version by using this module-variant CFA. In Section 5, we consider a more sophisticated modular analysis and proves its correctness also by using the module-variant CFA. We discuss the case when we modularize already polyvariant CFAs in Section 6, and conclude in Section 7.

2 Incremental Model for Modular Analysis

We assume that a modular analysis is to be integrated with an incremental compilation environment [AM94]. A module consists of declarations (implementation) and a signature (interface) which lists exported variables that are visible from the outside of the module. We assume that there exists partially ordered dependency between modules and we analyze modules in sequence by its topological order, as in the incremental compilation system. No circular dependency between modules is assumed.

2.1 Incremental Model

Figure 1 illustrates our incremental model of modular analysis. For a given module $M = (decl, sig)$, we first derive equations $\mathcal{D}(M)$:

$$\mathcal{D} : \mathit{Module} \rightarrow \mathit{Equations}.$$

Then we solve $\mathcal{S}(\mathcal{D}(M))$ the derived equations:

$$\mathcal{S} : \mathit{Equations} \rightarrow \mathit{Equations}.$$

Let the result be Δ. If the module uses some variables of other modules, we obtain Δ by solving the set of equations of the current module and those exported from the referred modules. Among this Δ, we export some equations $\mathcal{E}(\Delta, sig)$ that can be needed to analyze subsequent modules:

$$\mathcal{E} : \mathit{Equations} \times \mathit{Signature} \rightarrow \mathit{Equations}.$$

For modules M_1, M_2, $\cdots$, and M_n, the intermediate solution Δ_i and exported set δ_i for module M_i (in Figure 1) are thus defined as

$$\begin{aligned} \Delta_i &= \mathcal{S}\left(\mathcal{D}(M_i) \cup \left(\cup_{M_j \sqsubset M_i} \delta_j\right)\right) \text{ and} \\ \delta_i &= \mathcal{E}(\Delta_i, sig_i) \end{aligned}$$

where relation $M_j \sqsubset M_i$ denotes that module M_i uses variables of module M_j, and sig_i is the signature of M_i. The final analysis result $Sol(M_1, \cdots, M_n)$ for the whole-program is:

$$Sol(M_1, \cdots, M_n) = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n.$$

3 0-CFA

The whole-program 0-CFA [Shi88], whose modular versions we are designing, is shown in Figure 2. An input program is a sequence of declarations, and an expression is either a function, an application or a variable. We describe 0-CFA by the notation similar to that of Heintze

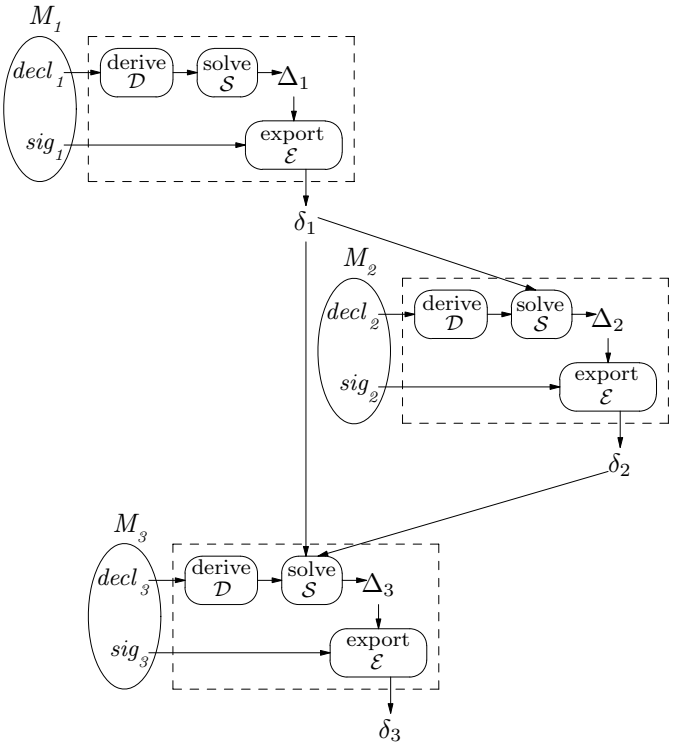

Figure 1: Incremental Model for Modular Analysis. The case is that M_2 uses M_1's functions and M_3 uses functions of M_1 and M_2.

and McAllester [HM97]. Edge "$n \to m$" means that the analysis result of n includes that of m. Applying the rules of Figure 2, we collect edges until no more edge is added. "$n \to \lambda(x, l)$" in the collected edges denotes that the expression of node n can be evaluated to the function value $\lambda x.e$ where e's label is l.

4 N: Naive Modular 0-CFA

In this section, we present a naive incremental version of 0-CFA. Rules in the deriving phase are those of 0-CFA that collect call-graph edges by examining only the program text. Rules in the solving phase are those of 0-CFA that estimate the information flow in function calls. In the exporting phase, we export every edge reachable from the names in the signature.

Interestingly enough, this obvious modular analysis **N** is not *sound* with respect to 0-CFA. This situation is not because **N** is an incorrect analysis *but* because even such a trivial modularization makes 0-CFA polyvariant. Therefore we need to find a new reference other than the original 0-CFA for proving the correctness of **N**.

We have found that a polyvariant version of 0-CFA at module-level, which is a straightforward extension to 0-CFA, can be conveniently used as a reference for the correctness of our modular 0-CFA. It is "convenient" because the proofs are between two static analyses which have a smaller semantic gap than that between our modular 0-CFA and the program's actual control flow.

$$
\begin{array}{lll}
\textit{Label} & l & \\
\textit{Variable} & v & ::= \quad x \mid f \\
\textit{Expr} & e & ::= \quad v \mid \lambda x.e^{l} \mid e^{l}\ e^{l} \\
\textit{Decl} & d & ::= \quad f = e^{l} \\
\textit{Program} & \wp & ::= \quad d^{*} \\
 & & \\
\textit{Node} & n & ::= \quad v \mid l \mid \lambda(x,l) \\
\textit{Edge} & c & ::= \quad n \to n
\end{array}
$$

$$
\frac{f = e^{l} \in \wp}{f \to l}\ (Dec_0) \qquad \frac{(e_1^{l_1}\ e_2{}^{l_2})^{l} \in \wp \quad l_1 \to \lambda(x,l_0)}{l \to l_0 \quad x \to l_2}\ (App_0)
$$

$$
\frac{v^{l} \in \wp}{l \to v}\ (Var_0) \qquad \frac{n \to m \quad m \to \lambda(x,l)}{n \to \lambda(x,l)}\ (Tr_0)
$$

$$
\frac{(\lambda x.e^{l_0})^{l} \in \wp}{l \to \lambda(x,l_0)}\ (Lam_0)
$$

Figure 2: The language and its 0-CFA.

Let us examine the exact definition of **N**, why it is unsound with respect to the whole 0-CFA, and how the module-variant whole-program CFA helps in its soundness proof.

4.1 Definition

Figure 3 has the definition of **N**.

The deriving and solving phases are directly from the rules of 0-CFA. We divide the (App_0) rule into two: one (*Appd*) that needs the source text and the other (*App*) that does not. Rule (*Appd*) is applied in the deriving phase and rule (*App*) is applied in the solving phase. In order to make this separation clear, we have to add edge "$l \to @(l_1, l_2)$," which encodes that there is an application labeled by l whose function part is labeled by l_1 and whose argument is labeled by l_2.

This separation is necessary because modular analysis is not supposed to see the entire source of the program. Edges from analyzing module A that will be exported have to be encoded such that the analyses of subsequent modules work without module A's text.

We devise the exporting phase conservatively in order to make sure that all the edges that can be needed by subsequent modules are exported. We simply export the edges that are reachable from the names in the signature. All declaration variables in the signature are in the *Reachable* set because they are directly referred to by subsequent modules (*Sig*). If subsequent modules may reach n, then they may reach n's sub-expressions: (*SubLam*) and (*SubApp*). If subsequent modules may reach n, and n can be evaluated to m ($n \to m$), then $n \to m$ is exported ($n \to m \in \delta$) to subsequent modules, and we estimate that m is reachable (*Out*). We repeatedly apply these rules until no more edge is exportable.

4.2 Proving Correctness by Using Module-Variant CFA

N is *unsound* with respect to 0-CFA. If a module has a dead code, the result of **N** does not

$$
\begin{array}{llll}
\textit{Signature} & \textit{sig} & ::= & \{f_1, \cdots, f_n\} \\
\textit{Declaration} & \textit{decl} & ::= & \wp \\
\textit{Module} & M & ::= & (\textit{decl}, \textit{sig}) \\
 & & & \\
\textit{Node} & n & ::= & v \mid l \mid \lambda(x,l) \mid @(l,l) \mid \top \\
\textit{Edge} & c & ::= & n \to n
\end{array}
$$

$\mathcal{D}$ derives edges from module M :

$$\frac{f = e^l \in M}{f \to l}\ (Dec) \qquad \frac{v^l \in M}{l \to v}\ (Var)$$

$$\frac{(e_1^{l_1}\ e_2^{l_2})^l \in M}{l \to @(l_1, l_2)}\ (Appd) \qquad \frac{(\lambda x.e^{l_0})^l \in M}{l \to \lambda(x, l_0)}\ (Lam)$$

$\mathcal{S}$ solves the edges :

$$\frac{l \to @(l_1, l_2) \quad l_1 \to \lambda(x, l_0)}{l \to l_0 \quad x \to l_2}\ (App) \qquad \frac{n \to m \quad m \to \lambda(x, l)}{n \to \lambda(x, l)}\ (Tr)$$

$\mathcal{E}$ exports edges δ from Δ with respect to the signature *sig* :

$$\frac{f \in \textit{sig}}{f \in \textit{Reachable}}\ (Sig) \qquad \frac{@(l_1, l_2) \in \textit{Reachable}}{l_1 \in \textit{Reachable} \quad l_2 \in \textit{Reachable}}\ (SubApp)$$

$$\frac{\lambda(x, l) \in \textit{Reachable}}{l \in \textit{Reachable}}\ (SubLam) \qquad \frac{n \in \textit{Reachable} \quad n \to m \in \Delta}{m \in \textit{Reachable} \quad n \to m \in \delta}\ (Out)$$

Figure 3: **N** – a naive incremental version of 0-CFA.

include that of 0-CFA. The following example is an evidence. When we present examples, as nodes, we use expressions instead of labels if it is not ambiguous.

Example 2 Consider the following two modules:

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

$$M_2 = \left(\texttt{ h = f } \lambda\texttt{w.w } , \{\texttt{h}\} \right).$$

If we analyze the whole program by 0-CFA, the result includes "$\texttt{w} \to \lambda\texttt{z.z}$" because $(\texttt{f } \lambda\texttt{y.y}) \to \texttt{x} \to \lambda\texttt{w.w}$. However, the exporting phase of **N** does not export $(\texttt{f } \lambda\texttt{y.y}) \to \texttt{x}$ because $(\texttt{f } \lambda\texttt{y.y})$ is not reachable from `f` which is in the signature. Thus the result of analyzing M_2 by **N** does not include $\texttt{w} \to \lambda\texttt{z.z}$. □

Because **N** is more accurate than the whole 0-CFA, we cannot prove its safety with respect to 0-CFA. We have to find a new reference to prove its correctness. The correctness relation between CFAs unexpectedly becomes

$$\begin{array}{rcll} \text{Semantics} & \subseteq & \text{0-CFA} & \text{(correctness of 0-CFA)} \\ \mathbf{N} & \subseteq & \text{0-CFA} & (\mathbf{N} \text{ is not correct with respect to 0-CFA}) \end{array}$$

$$\begin{array}{lrcl} \textit{Node} & n & ::= & x_M \mid l_M \mid f \mid \lambda(x, e^l) \\ \textit{Edge} & c & ::= & M \vdash e^l \mid n \to n \end{array}$$

$$\frac{f = e^l \in M}{M \vdash e^l \quad f \to l_M}\ (Dec_M) \qquad \frac{M \vdash (e_1^{l_1}\ e_2^{l_2})^l}{M \vdash e_1^{l_1} \quad M \vdash e_2^{l_2}}\ (Appd_M)$$

$$\frac{M \vdash x^l}{l_M \to x_M}\ (Bvar_M) \qquad \frac{M \vdash (e_1^{l_1}\ e_2^{l_2})^l \quad l_{1M} \to \lambda(x, e^{l_0})}{M \vdash e^{l_0} \quad x_M \to l_{2M} \quad l_M \to l_{0M}}\ (App_M)$$

$$\frac{M \vdash (\lambda x.e^{l_0})^l}{l_M \to \lambda(x, e^{l_0})}\ (Lam_M) \qquad \frac{n \to m \quad m \to \lambda(x, e^l)}{n \to \lambda(x, e^l)}\ (Tr_M)$$

$$\frac{M \vdash f^l}{l_M \to f}\ (Dvar_M)$$

Figure 4: Module-Variant CFA.

where $A \subseteq B$ means that the result of B includes that of A. The goal is to establish

$$\text{Semantics} \subseteq \mathbf{N}.$$

We want to find an analysis $\star$ which is obviously safe with respect to the semantics (Semantics$\subseteq \star \subseteq \mathbf{N}$) and which is also easy to prove that $\star \subseteq \mathbf{N}$.

Such an analysis $\star$ must be polyvariant in terms of modules. In Example 2, there are two applications whose function part is `f`: (`f` λ`y.y`) in M_1 and (`f` λ`w.w`) in M_2. By rule (App_0) of 0-CFA, both applications can be evaluated to `x`; that is, (`f` λ`y.y`)$\to$`x` and (`f` λ`w.w`)$\to$`x`. The former `x` is bound to only λ`y.y` while the latter is bound to both λ`y.y` and λ`w.w`. The `x`'s two occurrences in the two modules are distinguished during our modular analysis.

Thus, as such an analysis $\star$, we devise a polyvariant CFA at module-level. We call it *module-variant CFA*. The module-variant CFA is similar to the 1-CFA [Shi88]. Note that 1-CFA distinguishes the same variable by the last caller's label. Similarly, the module-variant CFA distinguishes the same variable (or label) by a module name whose analysis reaches it. For example, if $\lambda x.x$ is called by applications in modules M_1 and M_2 with arguments y and z, respectively, then we distinguish x by its callers' module: M_1 and M_2. That is, we bind y to x_{M_1} and z to x_{M_2}. Thus we can conclude that the application in M_1 can be evaluated to only y and the application in M_2 can be evaluated to only z.

This module-variant CFA is defined in Figure 4. Relation "$M \vdash e$" means that expression e is necessary in evaluating expressions in module M. Program variable x (or label l) is indexed x_M (or l_M) with module name M, which indicates that the values of x (or l) are accessed or bound when expressions in module M are evaluated. The starting point is declarations ($f = e^l \in M$). A declared expression is necessary in evaluating expressions of its own module ($M \vdash e^l$) and the values of the function name include those of the declared expression ($f \to l_M$):

$$\frac{f = e^l \in M}{M \vdash e^l \quad f \to l_M}\ (Dec_M).$$

If a function (respectively, a name) is necessary in evaluating expressions of module M, we just

let its label link to the function value (respectively, the name):

$$\frac{M \vdash (\lambda x.e^{l_0})^l}{l_M \to \lambda(x, e^{l_0})}\ (Lam_M),\quad \frac{M \vdash x^l}{l_M \to x_M}\ (Bvar_M),\ \text{and}\ \frac{M \vdash f^l}{l_M \to f}\ (Dvar_M).$$

Note that a bound variable keeps the module index, hence having the effect of being module-variant. If an application is needed in evaluating expressions of module M, then its sub-expressions are also necessary:

$$\frac{M \vdash (e_1^{l_1}\ e_2^{l_2})^l}{M \vdash e_1^{l_1} \quad M \vdash e_2^{l_2}}\ (Appd_M).$$

If an application is necessary in evaluating expressions of module M ($M \vdash (e_1^{l_1}\ e_2^{l_2})^l$) and, in addition, the function part of the application can be a value ($l_{1\,M} \to \lambda(x, e^{l_0})$), then (1) the body of the function value is needed to evaluate the expressions of module M ($M \vdash e^{l_0}$), (2) the values of the application includes those of the function body ($l_M \to l_{0\,M}$), and (3) the function can have, as its argument, the values of the argument part of the application ($x_M \to l_{2\,M}$):

$$\frac{M \vdash (e_1^{l_1}\ e_2^{l_2})^l \quad l_{1\,M} \to \lambda(x, e^{l_0})}{M \vdash e^{l_0} \quad x_M \to l_{2\,M} \quad l_M \to l_{0\,M}}\ (App_M).$$

Note that the function body keeps the caller's module index, having the effect of being module-variant. Finally, we propagate function values; that is, if m can be a function value ($m \to \lambda(x, e^l)$) and the values of n includes those of m ($n \to m$), then n can also be a function value ($n \to \lambda(x, e^l)$):

$$\frac{n \to m \quad m \to \lambda(x, e^l)}{n \to \lambda(x, e^l)}\ (Tr_M).$$

Now, by using the correctness of the module-variant CFA, we can prove the correctness of **N**. Thanks to [NN97], the correctness of the module-variant CFA is easy to prove:

Theorem 1 (Correctness of Module-Variant CFA) *The module-variant CFA is correct.*

Proof. The module-variant CFA is correct by Theorem 4.1 in [NN97]. It is an instance of the infinitary control flow analysis. □

The correctness of **N** is now proven by showing that **N**'s result includes that of the module-variant CFA.

Corollary 1 (Correctness of N) *Let $Sol_m(\wp)$ be the solution of a merged whole program $\wp$ of modules M_1, M_2, $\cdots$, and M_n, by the module-variant CFA. If $n \to \lambda(x, e^l) \in Sol_m(\wp)$ then $|n| \to |\lambda(x, e^l)| \in Sol_{\mathbf{N}}(M_1, M_2, \cdots, M_n)$ which is the final analysis result by **N**. Note that $|n|$ means the corresponding node of **N** to node n of the module-variant CFA.*

Proof. This is a corollary of Theorem 2. □

Theorem 2 *For a merged whole program $\wp$ of modules, the following are true:*

- *If expression e is necessary to analyze expressions of a subsequent module M in the module-variant CFA, then naive modular analyses of modules before M assert that expression e is reachable.*

- *If $e \to e'$ occurs in analyzing module M in the module-variant CFA, then*
  - *$|e| \to |e'|$ occurs in naive modular analysis of module M.*
  - *the naive modular analyses of modules before M assert that the two expressions are reachable.*

Proof. The full proof is in Appendix A. □

Thanks to the proofs of Theorem 1 and Corollary 1, we can avoid the burden of proof with respect to the standard semantics. We claim that the proofs of Theorem 1 and Corollary 1 are relatively simple because they are between two static analyses which have a smaller semantic gap than that between our modular 0-CFA and the program's actual control flow.

The correctness relation between CFAs becomes:

$$\text{Semantics} \subseteq \text{Module-variant CFA} \subseteq \mathbf{N} \subseteq \text{0-CFA}.$$

5 S: More Sophisticated Modular 0-CFA

Given that module-variant CFA is a convenient vehicle to prove the correctness of the naive modular CFA, the next question is: is it also good for more sophisticated modular versions? In this section, we present another modular version **S** that exports less edges than **N** after an analysis of each module, and show that the module-variant CFA is still a reference for its correctness proof.

S has the same deriving and solving phases as **N** but has a sharper exporting phase (shown in Figure 5). The exporting is the process of finding edges that subsequent modules may depend on:

Definition 1 Expression e_1 depends on expression e_2 if and only if evaluating e_1 evaluates e_2. Moreover, module m depends on expression e if and only if an expression of m depends on e.

(Note that **N**'s exporting phase using the notion of reachability is a naive upper approximation of this dependence relation.)

The starting point for our new exporting rules is the signature as usual. For a variable f in the signature, subsequent modules can directly refer to f; that is, subsequent modules may depend on f:

$$\frac{f \in sig}{f \in Depend}\ (Sig).$$

If such f can be evaluated to a function value, then subsequent modules may depend on the function value ($\lambda(x,l) \in Depend$) and the solution edge ($f \to \lambda(x,l) \in \delta$) is exported:

$$\frac{f \in Depend \quad f \to \lambda(x,l) \in \Delta}{\lambda(x,l) \in Depend \quad f \to \lambda(x,l) \in \delta}\ (FtoLam).$$

If subsequent modules may depend on a function value, then they may depend on the function's body:

$$\frac{\lambda(x,l) \in Depend}{l \in Depend}\ (SubLam).$$

If subsequent modules may depend on label l, then we export its edges excluding ($\Delta \cap \Pi$) those that are introduced during the solving phase of the current module. This exclusion is because such edges will be re-introduced anyway if they are needed to analyze the subsequent modules:

$$\frac{l \in Depend \quad l \to n \in \Delta \cap \Pi}{n \in Depend \quad l \to n \in \delta}\ (Excl)$$

$$\frac{f \in sig}{f \in Depend}\ (Sig) \qquad \frac{f \in Depend \quad f \to \lambda(x,l) \in \Delta}{\lambda(x,l) \in Depend \quad f \to \lambda(x,l) \in \delta}\ (FtoLam)$$

$$\frac{@(l_1,l_2) \in Depend}{l_1 \in Depend \quad l_2 \in Depend}\ (SubApp) \qquad \frac{l \in Depend \quad l \to n \in \Delta \cap \Pi}{n \in Depend \quad l \to n \in \delta}\ (Excl)$$

where $\Pi = \bigcup_{i=1}^{k} \mathcal{D}(M_i)$

$$\frac{\lambda(x,l) \in Depend}{l \in Depend}\ (SubLam)$$

Figure 5: The exporting phase of **S** for module M_k.

where Π is a set $\cup_{i=1}^{k}\mathcal{D}(M_i)$ of edges from the deriving phases of the previous modules including the current one M_k. If subsequent modules may depend on an application, then they may depend on its sub-expressions:

$$\frac{@(l_1,l_2) \in Depend}{l_1 \in Depend \quad l_2 \in Depend}\ (SubApp).$$

Compared with the exporting phase of **N**, **S** exports less edges. Rules (*Sig*), (*SubApp*) and (*SubLam*) are the same as in **N**, but rule (*Out*) of **N** becomes more selective in **S** by the two rules (*Excl*) and (*FtoLam*).

Example 3 Consider Example 1 and its edges:

$$\left(\begin{array}{l} \texttt{f = } (\lambda \texttt{x.x}^2)^1 \\ \texttt{g = } (\texttt{f}^4\ (\lambda \texttt{y.y}^6)^5)^3 \end{array} \ , \{\texttt{f},\texttt{g}\} \right) \quad \text{and} \quad \begin{array}{l} \texttt{f} \to 1 \to \lambda(\texttt{x},2) \\ \texttt{g} \to 3 \to 2 \to \texttt{x} \to 5 \to \lambda(\texttt{y},6) \\ 3 \to @(4,5) \\ 6 \to \texttt{y}. \end{array}$$

N exports all edges, while **S** exports $\texttt{f} \to \lambda(\texttt{x},2)$ and $\texttt{g} \to \lambda(\texttt{y},6)$ by (*FtoLam*), and $2 \to \texttt{x}$ and $6 \to \texttt{y}$ by (*Excl*). □

S is still correct with respect to the module-variant CFA (Figure 4).

Corollary 2 (Correctness of S) *For a merged whole program $\wp$ of modules M_1, M_2, $\cdots$, and M_n, if $n \to \lambda(x,e^l) \in Sol_m(\wp)$, then $|n| \to |\lambda(x,e^l)| \in Sol_{\mathbf{S}}(M_1, M_2, \cdots, M_n)$ which is the final analysis result by* **S**.

Proof. This is a corollary of Theorem 3 in Appendix B. Theorem 3 is a simple adaptation of Theorem 2 for the new exporting rules. □

6 Modularization of Polyvariant CFAs

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 module-variant. In other words, modular versions of k-CFA can be more accurate than the original k-CFA, hence, its correctness cannot be proven with respect to the original k-CFA.

Example 4 Consider the following two modules.

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

1-CFA concludes that `h` can be evaluated to both λ`w.w` and λ`y.y`, because they cannot distinguish `z`'s bindings by the applications of `g` and `h`. (1-CFA can only distinguish variables by their *last* call sites.) However, we can export naturally, from the first module, `f` $\rightarrow$ `(`λ`x.(`λ`z.z) x)` and `g` $\rightarrow$ λ`y.y`, and we can conclude that `h` can be evaluated to only λ`w.w`, which is same to the result of the module-variant CFA. For any k-CFA, we can show similar counter-examples. □

Therefore, in order to prove the correctness of modular version of k-CFA, we have to devise a module-variant k-CFA.

On the other hand, it is noteworthy that modular versions of the polymorphic-splitting CFA [WJ98] are always sound with respect to the original version. It is because the polymorphic-splitting CFA is already more accurate than the module-variant CFA. When an application uses a let-bound (declared) function, the polymorphic-splitting CFA separately copies the equations derived from the function, and analyzes the application with the copied equations. This process makes the polymorphic-splitting CFA already polyvariant at the declaration-level which is of a finer grain than module-level.

7 Conclusion

Designing modular analyses, which are practical alternatives to whole-program analysis for large-scale realistic programs, is usually *after* whole-program analyses' cost-accuracy balance is assured. It is, however, not much studied on how to derive a correct modular version from a given whole-program analysis. For example, to our knowledge, the abstract interpretation framework [CC77, CC92] does not yet cover the practices of designing modular static analyses; it assumes that the whole-program text is available a priori.

This article is about our findings that we realized when we tried to derive modular control-flow analyses for a higher-order language from a whole-program analysis:

- Deriving a modular version (in the framework of incremental analysis) from a whole-program CFA makes the resulting analysis polyvariant at module-level.
- If the original whole-program CFA was less accurate than or incomparable to module-level polyvariant analyses, then the correctness of its modular version may not be proven with respect to the original analysis, because the modular CFA, in that case, becomes more accurate than the original whole-program CFA.
- A convenient vehicle to prove the correctness of modular CFAs is a whole-program CFA that is polyvariant at module-level.

Our result 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.

Appendix

A Proof of Corollary 1

We use the following notations. Solution $Sol_m(\wp)$ denotes a set of edges introduced in analyzing a program $\wp$ by the module-variant CFA. Relation $A \leq B$ denotes a transitive closure of module-dependency relation $\sqsubset$. Relation $A < B$ denotes that $A \leq B$ and $A \neq B$. Set $Reachable_I$ denotes the *Reachable* set of the exporting phase in analyzing module I by **N**. Set $Depend_I$ denotes the *Depend* set of the exporting phase in analyzing module I by **S**. Node $|n|$ denotes the node of modular CFAs that corresponds to node n of the module-variant CFA. For example, $|x_M| = x$, $|f| = f$ and $|\lambda(x, e^l)| = \lambda(x, l)$.

Theorem 2 *For a merged whole program $\wp$ of modules, the followings are true:*

- *If $M \vdash e^l \in Sol_m(\wp)$ where $e^l \in A$, then $l \in Reachable_I$ for $A \leq I < M$.*
- *If $n \to m \in Sol_m(\wp)$ where $m \in A$, and n has M as its index or n is a function name in M, then*
  - *$|n| \to |m| \in \Delta_M$ and*
  - *$|m| \in Reachable_I$ for $A \leq I < M$.*

where Δ_M is the solved edges Δ (in Section 2) in analyzing module M by **N**.

Proof. We prove by induction on the size of the edge construction tree.

- **case** (Dec_M): Suppose that the edge is constructed by (Dec_M):

$$\frac{f = e^l \in M}{M \vdash e^l \quad f \to l_M}\ (Dec_M).$$

  Because $e^l \in M$, we have nothing to prove for $M \vdash e^l$. By (*Dec*), $f = e^l \in M$ implies $f \to l \in \Delta_M$.

- **case** ($Dvar_M$): Suppose that the edge is constructed by ($Dvar_M$):

$$\frac{M \vdash f^l}{l_M \to f}\ (Dvar_M)$$

  where $f \in A$ and $f^l \in B$. By induction hypothesis,

$$l \in Reachable_I \text{ for } B \leq I < M. \tag{1}$$

  By (*Var*), $f^l \in B$ implies $l \to f \in \Delta_B$. By (1) and (*Out*),

$$l \to f \in \Delta_I \text{ for } B \leq I \leq M$$

  and

$$f \in Reachable_I \text{ for } B \leq I < M.$$

  Because B directly refers to f, f is a function name in B (that is, $A = B$), or A is the next module of B ($f \in sig_A$ and $A \sqsubset B$). In the latter case, by (*Sig*), $f \in Reachable_A$. Therefore, whether $A = B$ or not,

$$f \in Reachable_I \text{ for } A \leq I < M.$$

- **case** ($Bvar_M$): Suppose that the edge is constructed by ($Bvar_M$):

$$\frac{M \vdash x^l}{l_M \to x_M}$$

where $x^l \in A$. By (Var), $x^l \in A$ implies $l \to x \in \Delta_A$. By induction hypothesis, $l \in \mathit{Reachable}_I$ for $A \leq I < M$. Thus by (Out),

$$l \to x \in \Delta_I \text{ for } A \leq I \leq M$$

and

$$x \in \mathit{Reachable}_I \text{ for } A \leq I < M.$$

- **case** (Lam_M): Similar to the case of ($Bvar_M$).

- **case** ($Appd_M$): Suppose that the edges are constructed by ($Appd_M$):

$$\frac{M \vdash (e_1^{l_1}\ e_2^{l_2})^l}{M \vdash e_1^{l_1} \quad M \vdash e_2^{l_2}}$$

where $(e_1^{l_1}\ e_2^{l_2})^l \in A$. By ($Appd$), $l \to @(l_1, l_2) \in \Delta_A$. By induction, $l \in \mathit{Reachable}_I$ for $A \leq I < M$. By (Out), $@(l_1, l_2) \in \mathit{Reachable}_I$ for $A \leq I < M$. Thus by ($SubApp$),

$$l_1, l_2 \in \mathit{Reachable}_I \text{ for } A \leq I < M.$$

- **case** (App_M): Suppose that the edges are constructed by (App_M):

$$\frac{M \vdash (e_1^{l_1}\ e_2^{l_2})^l \quad l_{1M} \to \lambda(x, e^{l_0})}{M \vdash e^{l_0} \quad x_M \to l_{2M} \quad l_M \to l_{0M}}$$

where $(e_1^{l_1}\ e_2^{l_2})^l \in A$ and $\lambda x.e^{l_0} \in B$. By induction hypothesis,

$$l \in \mathit{Reachable}_I \text{ for } A \leq I < M, \tag{2}$$
$$l_1 \to \lambda(x, l_0) \in \Delta_M, \text{ and} \tag{3}$$
$$\lambda(x, l_0) \in \mathit{Reachable}_I \text{ for } B \leq I < M. \tag{4}$$

By (4) and ($SubLam$),

$$l_0 \in \mathit{Reachable}_I \text{ for } B \leq I < M.$$

By ($Appd$), $(e_1^{l_1}\ e_2^{l_2})^l \in A$ implies $l \to @(l_1, l_2) \in \Delta_A$. Then by (2) and ($Out$),

$$l \to @(l_1, l_2) \in \Delta_I \text{ for } A \leq I \leq M \tag{5}$$

and $@(l_1, l_2) \in \mathit{Reachable}_I$ for $A \leq I < M$. Thus by ($SubApp$),

$$l_2 \in \mathit{Reachable}_I \text{ for } A \leq I < M.$$

By (3), (5) and (App),

$$l \to l_0 \in \Delta_M$$

and

$$x \to l_2 \in \Delta_M.$$

- **case** (Tr_M): Suppose that the edge is constructed by (Tr_M):

$$\frac{n_N \to m_M \quad m_M \to \lambda(x, e^l)}{n_N \to \lambda(x, e^l)}$$

where $m \in A$ and $\lambda(x, e^l) \in B$. For convenience, if m is a declaration variable, then we assume that m has m's own module A as its index (Note that a declaration variable has no index). By induction hypothesis,

$$n \to m \in \Delta_N, \tag{6}$$
$$m \in \mathit{Reachable}_I \text{ for } A \le I < N, \tag{7}$$
$$m \to \lambda(x, l) \in \Delta_M, \tag{8}$$
$$\lambda(x, l) \in \mathit{Reachable}_I \text{ for } B \le I < M \tag{9}$$

Because $A \le M \le N$, (7), (8) and (Out) imply

$$m \to \lambda(x, l) \in \Delta_I \text{ for } M \le I \le N \tag{10}$$

and $\lambda(x, l) \in \mathit{Reachable}_I$ for $M \le I < N$; that is, with (9),

$$\lambda(x, l) \in \mathit{Reachable}_I \text{ for } B \le I < N.$$

By (6), (10) and (Tr),

$$n \to \lambda(x, l) \in \Delta_N. \quad \square$$

B Proof of Corollary 2

Theorem 3 *For a merged whole program $\wp$ of modules, the followings are true:*

- *If $M \vdash e^l \in \mathit{Sol}_m(\wp)$ and $e^l \in A$, then $l \in \mathit{Depend}_I$ for $A \le I < M$.*
- *If $n \to m \in \mathit{Sol}_m(\wp)$, and*
  - *n is not a declaration variable or*
  - *n is a declaration variable and m is a function value,*

  where $m \in A$, and n has M as its index or n is a declaration variable in M, then
  - *$|n| \to |m| \in \Delta_M$,*
  - *$|m| \in \mathit{Depend}_I$ for $A \le I < M$.*

where Δ_M is the solved edges Δ (in Section 2) in analyzing module M by **S**.

Proof. We also prove by induction on the size of the edge construction tree. There is nothing to prove for the case of (Dec_M), and other cases, except for the case of (Tr_M), are the same as the proof of Theorem 2 if we replace (Out) by $(Excl)$, and *Reachable* by *Depend*.

- **case** (Tr_M): Suppose that the edges are constructed by (Tr_M):

$$\frac{n \to m \quad m \to \lambda(x, e^l)}{n \to \lambda(x, e^l).}$$

We prove for two cases: when m is a declaration variable and not.

– If m is a declaration variable g: then the edge is constructed by the following:

$$\frac{l_M \to g \quad g \to \lambda(x, e^{l_0})}{l_M \to \lambda(x, e^l)}$$

where $g \in A$ and $\lambda(x, e^{l_0}) \in B$. By induction hypothesis,

$$l \to g \in \Delta_M, \tag{11}$$
$$g \in \mathit{Depend}_I \text{ for } A \leq I < M, \tag{12}$$
$$g \to \lambda(x, l_0) \in \Delta_A, \text{ and} \tag{13}$$
$$\lambda(x, l_0) \in \mathit{Depend}_I \text{ for } B \leq I < A. \tag{14}$$

By (12), (13) and (*FtoLam*),

$$g \to \lambda(x, l_0) \in \Delta_I \text{ for } A \leq I \leq M \tag{15}$$

and $\lambda(x, l_0) \in \mathit{Depend}_I$ for $A \leq I < M$; that is, with (14),

$$\lambda(x, l_0) \in \mathit{Depend}_I \text{ for } B \leq I < M.$$

By (11), (15) and (*Tr*),

$$l \to \lambda(x, l_0) \in \Delta_M.$$

– If m is not a declaration variable: then the edge is constructed by the following:

$$\frac{n_M \to m_M \quad m_M \to \lambda(x, e^l)}{n_M \to \lambda(x, e^l)}$$

where $\lambda(x, e^l) \in A$. By induction hypothesis,

$$n \to m \in \Delta_M, \tag{16}$$
$$m \to \lambda(x, l) \in \Delta_M, \text{ and} \tag{17}$$
$$\lambda(x, l) \in \mathit{Depend}_I \text{ for } A \leq I < M. \tag{18}$$

By (16), (17) and (*Tr*),

$$n \to \lambda(x, l) \in \Delta_M. \quad \square$$

References

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

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992. Also as a tech report: Ecole Polytechnique, no. LIX/RR/92/10.

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

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 1997.

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

[WJ98] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, 1998.