A proof method for the correctness of modularized 0CFA✩

Oukseh Lee, Kwangkeun Yi*, Yunheung Paek

Department of Electrical Engineering and Computer Science,
KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, South Korea

Received 1 November 2000; received in revised form 26 April 2001
Communicated by R. Backhouse

Keywords: Control-flow analysis; Functional languages; Separate compilation; Static analysis; Formal semantics

1. This work

Modular program analysis, which analyzes separated program sources such as modules, is a practical alternative to whole-program analysis. It does not need the entire program text as its input, and if some parts of the program are modified, it re-analyzes only the dependent parts of a modified module.

This article is about our findings when we tried to derive a modular version from a whole-program control-flow analysis (CFA) [1–3], to be used inside a modularized version of our exception analysis [4–6]:

• Deriving a modular version from a whole-program monovariant (or context-insensitive) CFA makes the resulting analysis polyvariant (or context-sensitive) at the module level.

• Hence the correctness of its modularized version cannot be proven in general with respect to the original CFA.

• A convenient stepping stone to prove the correctness of a modularized version (instead of proving it against the program semantics) is a whole-program CFA that is polyvariant at the module level.

Because CFA is a basis of almost all analyses for higher-order programs, our result can be seen as a general hint of using the module-variant whole-program analysis in order to ease the correctness proof for a modularized version. We think this is worthwhile to report because usually in practice we first design a whole-program analysis, prove its correctness against the program semantics, and then only after its cost-accuracy balance is assured we start designing its modularized version. Our work can also be seen as a formal investigation, for CFA, of the folklore that modularization improves the analysis accuracy.

Example 1. As an example that modularization improves the accuracy, consider a CFA of the following two higher-order code fragments:

\[ \text{id} = \lambda x.x \]

\[ \text{dec} = \text{id} \lambda y.y-1 \]

and

\[ \text{inc} = \text{id} \lambda z.z+1. \]

The goal of CFA is to safely estimate which functions flow into each expression. Suppose we analyze the
two fragments together. Because of the two calls to \texttt{id}, \texttt{id}'s formal parameter \( x \) is bound to both \( \lambda y.y-1 \) and \( \lambda z.z+1 \). This information is propagated back to the call sites that we conclude \texttt{inc} has \( \lambda y.y-1 \) (a false flow) as well as \( \lambda z.z+1 \). On the other hand, analyzing the first fragment in isolation concludes that \( \texttt{inc} \) has only \( \lambda x.x \). Analyzing the second fragment with this information concludes that \texttt{inc} has only \( \lambda z.z+1 \).

Section 2 shows the language and its monovariant CFA (0CFA), and Section 3 describes an incremental model for our modular analysis. Section 4 presents 0CFA's modular version (0CFA/m). Section 5 shows that 0CFA/m is not a conservative extension of the original 0CFA. Section 6 presents a module-variant whole-program 0CFA and Section 7 proves that 0CFA/m is its conservative extension.

2. 0CFA

The whole-program 0CFA [1], whose modular version we are designing, is shown in Fig. 1. We present 0CFA in the style similar to [2]. Nodes are syntactic objects: the variables or sub-expressions of the input program. All variables and labels are 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 Fig. 1, we collect such edges until no more additions are possible. Edge “\( n \rightarrow \lambda x.e' \)” in the final result indicates that \( n \) may evaluate into (or, is bound to) function \( \lambda x.e' \) in the input program. The correctness of 0CFA is known [1,3].

3. Incremental model for modular analysis

We assume that a modular analysis works inside an incremental compilation environment [7]. A module consists of variable declarations (“\( \times = e' \)” and a signature that lists a subset of the declared variables visible from other modules. Module \( M \) directly depends on another module \( M' \), written \( M' \sqsubseteq M \), iff \( M \) uses variables of \( M' \).

We assume an acyclic dependency between modules and we analyze modules in sequence by their topological order [7]. In cases that modules have a cyclic dependency, (1) we can consider mutually-dependent modules as one unit of a modular analysis, or (2) we repeat analyzing mutually-dependent modules until their analyses reach a fixpoint. In this paper, we do not consider cyclic module dependencies.

Fig. 2 illustrates our incremental model of modular analysis. We analyze each module in its dependency order and export some of its results that subsequent modules may need. For a given module \( M = (\texttt{decl}, \texttt{sig}) \), let the analysis phase be \( A(M, \delta) \) with \( A : \texttt{Module} \times \texttt{Results} \rightarrow \texttt{Results} \). The second input \( \delta \) is the exported results from the modules that \( M \) directly depends on. Let \( \Delta \) be the analysis result. From \( \Delta \), we export only those parts of it that subsequent modules may need. Let this export phase be \( E(\Delta, \texttt{sig}) \) with \( E : \texttt{Results} \times \texttt{Signature} \rightarrow \texttt{Results} \). 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 Fig. 2) are \( \Delta_i = A(M_i, \bigcup_{j \sqsubseteq M_i} \delta_j) \) and \( \delta_i = E(\Delta_i, \texttt{sig}_i) \), where \( \texttt{sig}_i \) is the signature of \( M_i \). The final analysis result \( \texttt{Sol}(M_1, \ldots, M_n) \) for the whole-program is \( \Delta_1 \cup \cdots \cup \Delta_n \).

It is clear that this model has an inherent effect of polyvariant analysis; a module’s analysis result is separately copied in analyzing subsequent modules. Our point here is to show how to ease the correctness proof of a modularized version when we move a whole-program analysis into this modular analysis model.

\begin{table}[h]
\centering
\begin{tabular}{llll}
\hline
\textbf{Label} & \textbf{Var} & \textbf{Constant} & \textbf{c} \\
\hline
\textbf{Expr} & \( e ::= x \mid \lambda x.e' \mid e' \mid c \) & & \\
\textbf{Decl} & \( d ::= x = e' \) & & \\
\textbf{Program} & \( \varnothing ::= d^\varnothing \) & & \\
\textbf{Node} & \( n ::= x \mid l \mid \lambda x.e' \) & & \\
\textbf{Edge} & \( g ::= n \rightarrow n \) & & \\
\hline
\( x = e' \in \varnothing \) & \( l \rightarrow l \) & \( \lambda x.e'\) & \( \rightarrow \lambda x.e' \) \\
\hline
\( (e_1', e_2') \in \varnothing \) & \( l_1 \rightarrow \lambda x.e' \) & \( n \rightarrow m \) & \( m \rightarrow \lambda x.e' \) \\
\hline
\end{tabular}
\caption{The language and its 0CFA.}
\end{table}
Fig. 2. Incremental model for modular analysis. Module $M_2$ uses names declared in $M_1$, and $M_3$ uses those of $M_1$ and $M_2$.

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

**Signature**  
$\text{sig} ::= \{x_1, \ldots, x_n\}$

**Module**  
$M ::= (\text{decl}, \text{sig})$  
$\text{decl} ::= (x = e')$  

**Node**  
$n ::= x \mid l \mid \lambda x. e'$

**Edge**  
$g ::= n \rightarrow n$

**Analysis phase.** $A(M, \delta) = \text{edge-set } \Delta$, closed by the five rules:

(Dec)  
\[
x = e' \in M \\
x \rightarrow l \in \Delta
\]

(App)  
\[
l_1 \rightarrow \lambda x. e' \in \Delta \\
(e'_1 l_1)^l \in M \text{ or } \delta
\]

(Var)  
\[
l \rightarrow x \in \Delta
\]

(Tr)  
\[
n \rightarrow m \in \Delta \\
m \rightarrow \lambda x. e' \in \Delta
\]

(Lam)  
\[
l \rightarrow \lambda x. e' \in \Delta
\]

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

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

(ExportFn)  
\[
\frac{x \in \text{Needed}}{FV(\lambda y. e') \subseteq \text{Needed} \\
x \rightarrow \lambda y. e' \in \delta}
\]

4. **0CFA/m: A modularized 0CFA**

We present a modular version of 0CFA in Fig. 3. Rules in the analysis phase $A(M, \delta)$ are the same as the rules in 0CFA except that instead of examining the whole-program text, they only examine the current module $M$ and the exported edges $\delta$ from the referenced modules. The premise “$\in M$ or $\delta$” means “is a sub-expression in either module $M$ or a node of $\delta$”. In the export phase $E(\Delta, \text{sig})$, we conservatively export all the edges that may be needed by subsequent modules. We calculate Needed, the set of variables needed by other modules, and exported edges $\delta$, as follows:

Case (Sig): The starting point is the signature. For a variable $x$ in the signature, $x$’s bindings are needed to analyze subsequent modules.

Case (ExportFn): If variable $x$ is needed to analyze subsequent modules ($x \in \text{Needed}$), then
(a) its analysis results \((x \to \lambda y. e')\) are all exported;
(b) we record \((\text{FV}(\lambda y. e') \subseteq \text{Needed})\) that the free
variables of the function are needed to analyze
subsequent modules.
Note that, even if a variable is not in the signature, its
analysis results can be exported.
Algorithm for 0CFA/m is the same as for 0CFA:
we add edges by applying the rules until no more
additions are possible. Note that we export code-
segments in \((\text{ExportFn})\) and re-use them in \((\text{Var}),\)
\((\text{Lam}),\) and \((\text{App}).\) For an efficient implementation of
0CFA/m, we can replace code-segments by equivalent
edges using simplification algorithms [8].

5. 0CFA/m is not a conservative extension of 0CFA

The 0CFA/m analysis is more accurate than 0CFA.

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

\[
M_1 = \left( f = (\lambda x. x^2)^1, \quad g = (\lambda y. y^6)^3, \quad \{f, g\} \right)
\]
and

\[
M_2 = \{h = (\lambda z. z^{10})^9, \{h\}\}.
\]

If we analyze the whole program by 0CFA, the re-
result includes a false-flow edge \(h \to \lambda y. y^6\). However,
0CFA/m does not conclude the false-flow edge. Analy-
zing the first module returns

\[
4 \to f \to 1 \to \lambda x. x^2,
\]

\[
g \to 3 \to 2 \to x \to 5 \to \lambda y. y^6,
\]

\[
6 \to y,
\]

among which 0CFA/m exports only two edges: \(f \to
\lambda x. x^2\) and \(g \to \lambda y. y^6\). Note that \(x \to \lambda y. y^6\) is
not included. With the exported edges from the first
module, analyzing the second module returns

\[
8 \to f \to \lambda x. x^2,
\]

\[
g \to \lambda y. y^6,
\]

\[
h \to 7 \to 2 \to x \to 9 \to \lambda z. z^{10}.
\]

The false-flow edge \(h \to \lambda y. y^6\) is absent.

This situation does not mean that 0CFA/m is incor-
rect; 0CFA/m is still correct (with respect to the pro-
gram semantics), but because modularization makes the
resulting analysis polyvariant, 0CFA/m fails to be
a conservative extension of the original 0CFA.

In order to prove the correctness of 0CFA/m, we
want to find a correct analysis \(A\) such that it is easy to
prove that 0CFA/m is a conservative extension of \(A\).

We show that such analysis \(A\) is a whole-program
analysis that is polyvariant at the module-level. We
call it module-variant 0CFA. This analysis is a con-
vienent stepping stone to proving the correctness of
0CFA/m because:

• the proof is between two static analyses (0CFA/m
and module-variant 0CFA) that have a smaller gap
than between a static analysis (0CFA/m) and the
program semantics, and

• the correctness of module-variant 0CFA is free
since it is an instance of the infinitary CFA of
Nielson and Nielson [3].

6. Module-variant 0CFA

Module-variant 0CFA distinguishes the same ex-
pression label (or variable) by the originating mod-
ules whose evaluations need its values. For example,
if \(\lambda x. x\) is called from modules \(M_1\) and \(M_2\) with actual
argument \(e_1\) and \(e_2\), then we distinguish the formal pa-
ter \(x\) 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 also has
two instances, indexed by \(M_1\) and \(M_2\).

The definition of the module-variant 0CFA is shown
in Fig. 4. In order to achieve its correctness for free,
we define it as an instance of the infinitary CFA [3]. In
order to fit with the program syntax in the infinitary
CFA, we assume that a program (declarations in
modules) is a single nested let-expression whose
innermost let-body is a dummy constant.

A judgment “\(S \models_{\sigma}^{\text{Var}} e\)” means \(S\) is a correct solu-
tion which covers the situation that evaluating mod-
ule \(M\) needs to evaluate \(e\) under environment \(\sigma\). En-
vironment \(\sigma\) maps free variables of \(e\) into the mod-
ules whose evaluation bind them. This environment
determines the variable’s module indices for the poly-
variant effect. Note that, in comparison with judgment
\((C, \rho) \models_{\text{me}}^{m} e\) in [3], we use \(S\) for \((C, \rho), \sigma\) for \(\text{me}\),
and \(M\) for \(m\).
For the input program $\wp$ that consists of modules $M_1, \ldots, M_n$, its module-variant 0CFA is defined [3] as the least $S$ such that $S \vDash S(x, \sigma(x)) \subseteq S(l, M)$ holds where $\emptyset$ is the empty module-context environment and $\emptyset$ is a dummy module index for the whole program.

Case (var). If a variable is necessary ($S \vDash S(x, \sigma(x)) \subseteq S(l, M)$) 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 ($S \vDash S(l, M)$) for module $M$, then the analysis result $S(l, M)$ at the label must include it.

Case (app). If an application is necessary ($S \vDash S(l, M)$) for evaluating module $M$, we propagate the same module context to its sub-expressions ($S \vDash S(l, M)$) of the variable. Moreover, for each function $(\forall (\lambda x. e^0, \sigma') \in S(l_1, M))$ that can be called, (a) its formal parameter $x$ and its body $e^0$ have the same module context: $S \vDash S[l_1 \mapsto M] e^0$; (b) actual parameter $e^2$ flow to the formal parameter $x$: $S(l_2, M) \subseteq S(x, M)$; (c) return value $e^0$ flow to the call expression $(e^1_1 e^2_2)l$: $S(l_0, M) \subseteq S(l, M)$.

Note that the module-variant effect occurs because the function’s argument and body have the call expression’s module index.

Case (let). Similar to the application case, except that because the let-binding “$x = e^1_1$” is a declaration in a module, we have to use this module context for the variable $x$ and its definition $e^1_1$.

Because the module-variant 0CFA is an instance of the infinitary control flow analysis [10], it is correct by Theorem 4.1 of Nielson and Nielson [3].

7. 0CFA/m is a conservative extension of module-variant 0CFA

We show that there exists a solution $S$ of the module-variant 0CFA that is covered by the result of 0CFA/m. Definition 2 defines such a solution $S$, and Theorem 1 asserts that the $S$ is a solution of the module-variant 0CFA.
Definition 1. Let $\Delta_M$ 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 $x \in \Delta_{M_0}$, or there exists a path $M_0 \sqsubset M_1 \cdots \sqsubset M_n$ such that for all $0 \leq i < n$, $x \in \text{Needed}_{M_i}$, where $\text{Needed}_{M_i}$ denotes the Needed set of the exporting phase in analyzing module $M_i$ by 0CFA/m (see Fig. 3).
- Environment $\sigma$ reaches $M$ iff, for all $x$ in $\text{dom}(\sigma)$, $x$ reaches $M$ via $\sigma(x)$.

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

$$\text{Sol}_{0\text{CFA}/m}(M_1, \ldots, M_n)(n, M) = \{ (\lambda x. e', \sigma) \mid n \rightarrow \lambda x. e' \in \Delta_M, \sigma \text{ reaches } M, \text{ dom}(\sigma) = FV(\lambda x. e') \},$$

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

Fact. By definition, $\langle \text{Sol}_{0\text{CFA}/m}() \rangle$ is “covered by” $\text{Sol}_{0\text{CFA}/m}()$: $(\lambda x. e') \in \langle \text{Sol}_{0\text{CFA}/m}() \rangle(n, M)$ implies $(n \rightarrow \lambda x. e') \in \text{Sol}_{0\text{CFA}/m}()$.

Theorem 1 (Correctness of 0CFA/m). Let program $\wp$, as a let-expression, consist of modules $M_1, \ldots, M_n$.

$\text{Sol}_{0\text{CFA}/m}(M_1, \ldots, M_n) \models \wp$ holds, where $\wp$ is the empty module-context environment and $\varepsilon$ is a dummy module index for the whole program.

Proof. Let $S = \langle \text{Sol}_{0\text{CFA}/m}(M_1, \ldots, M_n) \rangle$. Judgment $S \models^\sigma e'$ holds if it is included in the greatest fixed point of the function $F: \text{Judgments} \rightarrow \text{Judgments}$ derived from Fig. 4 [3]. $F(Q)$ gives us a set of left-hand side judgments asserted by the rules of Fig. 4 assuming that judgments in $Q$ hold. If we find a set $Q$ of judgments such that $(S \models^\sigma \wp) \in Q$ and $Q \subseteq F(Q)$, then by the co-induction principle [9], $Q$ is included in the greatest fixed point of $F$ and $S \models^\sigma \wp$ holds.

Therefore, the module-variant 0CFA’s solution, which is defined as the least $X$ such that $X \models^\sigma \wp$, is included in the modularized solution $\text{Sol}_{0\text{CFA}/m}(M_1, \ldots, M_n)$. The detailed proof is in [10]. □

Note that the module-variant 0CFA is not a modular analysis. It is a whole-program analysis, found as facilitating the correctness proof of the modular 0CFA (0CFA/m).

Example 3. Let us consider an example of Theorem 1. Consider the program in Example 2. In order to fit with the program syntax in the infinitary CFA, the program can be considered as:

$$\wp = (\text{let } f = (\lambda x. x^2)^1 \text{ in } \text{let } g = (x^4 (\lambda y. y^2)^3 \text{ in } \text{let } h = (x^8 (\lambda z. z^{10})^7 \text{ in } e^{11})^{12})^{13})^{14},$$

where $c$ is a dummy constant. Note that the module of $f$ and $g$ is $M_1$ and the module of $h$ is $M_2$. Let $\text{Sol}_{0\text{CFA}/m}(M_1, M_2)$ be the analysis result as shown in Example 2, and $S$ be $\langle \text{Sol}_{0\text{CFA}/m}(M_1, M_2) \rangle$. Then by Definition 2,

$$S(1, M_1) = S(4, M_1) = S(8, M_1) = S(\varepsilon, M_1) = S(\varepsilon, M_1) = \{(\lambda x. x^2, \emptyset)\},$$

$$S(2, M_1) = S(3, M_1) = S(5, M_1) = S(\varepsilon, M_1) = S(\varepsilon, M_1) = \{(\lambda y. y^0, \emptyset)\},$$

$$S(2, M_2) = S(7, M_2) = S(9, M_2) = S(\varepsilon, M_2) = S(\varepsilon, M_2) = \{(\lambda z. z^{10}, \emptyset)\},$$

and $S(n, M) = \emptyset$ for other $(n, M)$. Now we can see that $S \models^\sigma \wp$ holds. This can be proved by induction (co-induction is not necessary because $\wp$ has no recursive function).

8. Discussion

One question is: what if we modularize a more sophisticated CFA than 0CFA? The situation is similar to 0CFA. In case of context-sensitive CFAs, modularization can still improve their accuracies. For example, modularized versions of kCFA [1] or the polymorphic-splitting CFA [11] can be more accurate than their original whole-program versions [10]. The correctness of their modularized versions can be proven similarly,
by using module-variant whole-program versions. Detailed proof is available in [10].

Another question is: how far-reaching is the principle of module-variant analysis? If CFAs are already polyvariant at the module level (e.g., one in [11, p. 178]), then their modularizations cannot improve their accuracies, hence no need for module-variant versions to facilitate the correctness proof. For any analysis in general, we conjecture the same is true.

References