ELSEVIER

Information Processing Letters 81 (2002) 179–185

Information Processing Letters

www.elsevier.com/locate/ipl

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:

```
id = λx.x
dec = id λy.y-1
```

and

```
inc = id λz.z+1.
```

The goal of CFA is to safely estimate which functions flow into each expression. Suppose we analyze the

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

* Corresponding author.
E-mail addresses: cookcu@kaist.ac.kr (O. Lee), kwang@cs.kaist.ac.kr (K. Yi), ypaek@ee.kaist.ac.kr (Y. Paek).

0020-0190/02/$ – see front matter
PII: S0020-0190(01)00223-X

two fragments together. Because of the two calls to `id`, `id`'s formal parameter `x` is bound to both λ`y.y-1` and λ`z.z+1`. This information is propagated back to the call sites that we conclude `inc` has λ`y.y-1` (a false flow) as well as λ`z.z+1`. On the other hand, analyzing the first fragment in isolation concludes that `id` has λ`x.x` and `dec` has λ`y.y-1`. Analyzing the second fragment with this information concludes that `inc` has only λ`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 "$n \to m$" 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 "$n \to \lambda x.e^l$" in the final result indicates that n may evaluate into (or, is bound to) function $\lambda x.e^l$ in the input program. The correctness of 0CFA is known [1,3].

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

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

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

Fig. 1. The language and its 0CFA.

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 ("`x` = 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 $M' \sqsubset M$, 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 dependence order and export some of its results that subsequent modules may need. For a given module $M = (\textit{decl}, \textit{sig})$, let the analysis phase be $\mathcal{A}(M, \delta)$ with $\mathcal{A} : \textit{Module} \times \textit{Results} \to \textit{Results}$. The second input δ is the exported results from the modules that M directly depends on. Let Δ be the analysis result. From Δ, we export only those parts of it that subsequent modules may need. Let this export phase be $\mathcal{E}(\Delta, \textit{sig})$ with $\mathcal{E} : \textit{Results} \times \textit{Signature} \to \textit{Results}$. For a program that consists of modules $M_1, \ldots, M_n$, each module M_i's analysis result Δ_i and its exported set δ_i (in Fig. 2) are $\Delta_i = \mathcal{A}(M_i, \bigcup_{M_j \sqsubset M_i} \delta_j)$ and $\delta_i = \mathcal{E}(\Delta_i, \textit{sig}_i)$, where $\textit{sig}_i$ is the signature of M_i. The final analysis result $\textit{Sol}(M_1, \ldots, M_n)$ 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.

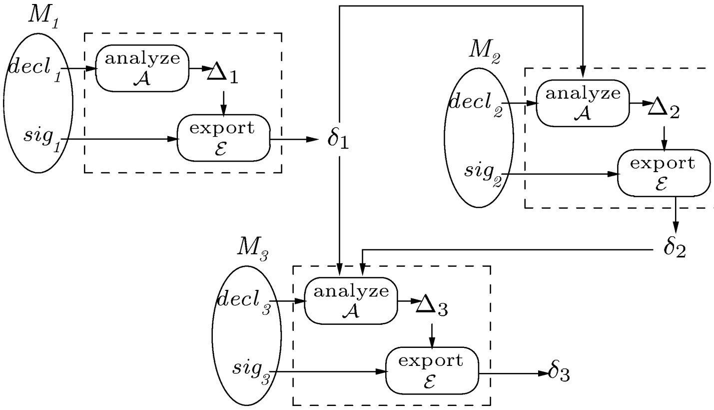

Fig. 2. Incremental model for modular analysis. Module M_2 uses names declared in M_1, and M_3 uses those of M_1 and M_2.

Signature	$sig ::= \{x_1, \ldots, x_n\}$	
Module	$M ::= (decl, sig)$	$decl ::= (x = e^l)^*$
Node	$n ::= x \mid l \mid \lambda x.e^l$	
Edge	$g ::= n \to n$	

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

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

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

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

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

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

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

4. 0CFA/m: A modularized 0CFA

We present a modular version of 0CFA in Fig. 3. Rules in the analysis phase $\mathcal{A}(M, \delta)$ 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 δ from the referenced modules. The premise "$\in M$ or δ" means "is a sub-expression in either module M or a node of δ". In the export phase $\mathcal{E}(\Delta, sig)$, 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 δ, 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 ($x \in Needed$), then

(a) its analysis results ($x \to \lambda y.e^l$) are all exported;
(b) we record ($FV(\lambda y.e^l) \subseteq \mathit{Needed}$) 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 (*ExportFn*) and re-use them in (*Var*), (*Lam*), and (*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:

$$M_1 = \left(\begin{array}{l} \mathtt{f} = (\lambda \mathtt{x}\,.\,\mathtt{x}^2)^1 \\ \mathtt{g} = (\mathtt{f}^4\,(\lambda \mathtt{y}\,.\,\mathtt{y}^6)^5)^3 \end{array},\ \{\mathtt{f}, \mathtt{g}\} \right)$$

and

$$M_2 = \left(\mathtt{h} = (\mathtt{f}^8\,(\lambda \mathtt{z}\,.\,\mathtt{z}^{10})^9)^7,\ \{\mathtt{h}\}\right).$$

If we analyze the whole program by 0CFA, the result includes a false-flow edge $\mathtt{h} \to \lambda\mathtt{y}.\mathtt{y}^6$. However, 0CFA/m does not conclude the false-flow edge. Analyzing the first module returns

$$4 \to \mathtt{f} \to 1 \to \lambda\mathtt{x}.\mathtt{x}^2,$$
$$\mathtt{g} \to 3 \to 2 \to \mathtt{x} \to 5 \to \lambda\mathtt{y}.\mathtt{y}^6,$$
$$6 \to \mathtt{y},$$

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

$$8 \to \mathtt{f} \to \lambda\mathtt{x}.\mathtt{x}^2,$$
$$\mathtt{g} \to \lambda\mathtt{y}.\mathtt{y}^6,$$
$$\mathtt{h} \to 7 \to 2 \to \mathtt{x} \to 9 \to \lambda\mathtt{z}.\mathtt{z}^{10}.$$

The false-flow edge $\mathtt{h} \to \lambda\mathtt{y}.\mathtt{y}^6$ is absent.

This situation does not mean that 0CFA/m is incorrect; 0CFA/m is still correct (with respect to the program 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 convenient 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 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 e_1 and e_2, 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 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 "$S \models_M^\sigma e^l$" means S is a correct solution which covers the situation that evaluating module M needs to evaluate e under environment σ. Environment σ maps free variables of e into the modules whose evaluation bind them. This environment determines the variable's module indices for the polyvariant effect. Note that, in comparison with judgment $(C, \rho) \models_m^{me} e^l$ in [3], we use S for (C, ρ), σ for *me*, and M for m.

$$
\begin{array}{lll}
 & \textit{Module} & M \\
 & \textit{ModEnv} & \sigma \in \textit{Var} \to \textit{Module} \\
 & \textit{Value} & v ::= (\lambda x.e^{l}, \sigma) \\
 & \textit{Solution} & S \in (\textit{Var} + \textit{Label}) \times \textit{Module} \to \textit{Value}
\end{array}
$$

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

Fig. 4. Module-variant 0CFA.

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 $S \models^{\emptyset}_{\varepsilon} \wp$ holds where $\emptyset$ is the empty module-context environment and ε is a dummy module index for the whole program.

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

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

Case (*app*). If an application is necessary ($S \models^{\sigma}_{M} (e_1^{l_1}\ e_2^{l_2})^{l}$) for evaluating module M, we propagate the same module context to its sub-expressions ($S \models^{\sigma}_{M} e_1^{l_1} \wedge S \models^{\sigma}_{M} e_2^{l_2}$). Moreover, for each function ($\forall (\lambda x.e^{l_0}, \sigma') \in S(l_1, M)$) that can be called,

(a) its formal parameter x and its body e^{l_0} have the same module context: $S \models^{\sigma'[x \mapsto M]}_{M} e^{l_0}$;
(b) actual parameter e^{l_2} flow to the formal parameter x: $S(l_2, M) \subseteq S(x, M)$;
(c) return value e^{l_0} flow to the call expression $(e_1^{l_1}\ e_2^{l_2})^{l}$: $S(l_0, M) \subseteq S(l, M)$.

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 "$x = e_1^{l_1}$" is a declaration in a module, we have to use this module context for the variable x and its definition $e_1^{l_1}$.

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 Δ_M be the solved edges in analyzing module M by 0CFA/m.

- Variable x reaches M_n via M_0 iff $M_0 = M_n$ and $x \in \Delta_{M_n}$, or there exists a path $M_0 \sqsubset M_1 \cdots \sqsubset M_n$ such that for all $0 \leqslant i < n$, $x \in Needed_{M_i}$, where $Needed_{M_i}$ denotes the *Needed* set of the exporting phase in analyzing module M_i by 0CFA/m (see Fig. 3).
- Environment σ reaches M iff, for all x in $dom(\sigma)$, x reaches M via $\sigma(x)$.

Definition 2 ($|Sol_{\text{0CFA/m}}(M_1, \ldots, M_n)|$). Let

$Sol_{\text{0CFA/m}}(M_1, \ldots, M_n)$

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

$$\begin{aligned}
&\left|Sol_{\text{0CFA/m}}(M_1, \ldots, M_n)\right|(n, M) \\
&\quad = \left\{(\lambda x.e^l, \sigma) \mid n \to \lambda x.e^l \in \Delta_M,\ \sigma \text{ reaches } M,\ dom(\sigma) = FV(\lambda x.e^l)\right\},
\end{aligned}$$

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

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

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

Proof. Let $S = |Sol_{\text{0CFA/m}}(M_1, \ldots, M_n)|$. Judgment $S \models^{\sigma}_{M} e^l$ holds if it is included in the greatest fixed point of the function $F : Judgments \to Judgments$ 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 $(S \models^{\emptyset}_{\varepsilon} \wp) \in Q$ and $Q \subseteq F(Q)$, then by the co-induction principle [9], Q is included in the greatest fixed point of F and $S \models^{\emptyset}_{\varepsilon} \wp$ holds.

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

$$\begin{aligned}
\wp \stackrel{\triangle}{=} &(\texttt{let f = } (\lambda \texttt{x}.\texttt{x}^2)^1 \texttt{ in} \\
&(\texttt{let g = } (\texttt{f}^4 (\lambda \texttt{y}.\texttt{y}^6)^5)^3 \texttt{ in} \\
&(\texttt{let h = } (\texttt{f}^8 (\lambda \texttt{z}.\texttt{z}^{10})^9)^7 \texttt{ in } c^{11})^{12})^{13})^{14},
\end{aligned}$$

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 $Sol_{\text{0CFA/m}}(M_1, M_2)$ be the analysis result as shown in Example 2, and S be $|Sol_{\text{0CFA/m}}(M_1, M_2)|$. Then by Definition 2,

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

and $S(n, M) = \emptyset$ for other (n, M). Now we can see that $S \models^{\emptyset}_{\varepsilon} \wp$ 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

[1] O. Shivers, Control-flow analysis of higher-order languages, Ph.D. Thesis, Carnegie Mellon University, Technical Report CMU-CS-91-145, May 1991.

[2] N. Heintze, D. McAllester, Linear-time subtransitive control flow analysis, in: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1997, pp. 261–272.

[3] F. Nielson, H.R. Nielson, Infinitary control flow analysis: A collecting semantics for closure analysis, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1997, pp. 332–345.

[4] K. Yi, S. Ryu, A cost-effective estimation of uncaught exceptions in Standard ML programs, Theoretical Comput. Sci., to appear; extended version of [5].

[5] K. Yi, S. Ryu, Towards a cost-effective estimation of uncaught exceptions in SML programs, in: P. van Hentenryck (Ed.), Proceedings of the 4th International Symposium on Static Analysis, Lecture Notes in Comput. Sci., Vol. 1302, Springer, Berlin, 1997, pp. 98–113.

[6] K. Yi, S. Ryu, SML/NJ Exception Analyzer 0.98, URL http://compiler.kaist.ac.kr/pub/exna/exna-README.html.

[7] A.W. Appel, D.B. MacQueen, Separate compilation for Standard ML, in: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 13–23.

[8] C. Flanagan, M. Felleisen, Componential set-based analysis, ACM Trans. Prog. Lang. Syst. 21 (2) (1999) 370–416.

[9] R. Milner, M. Tofte, Co-induction in relational semantics, Theoret. Comput. Sci. 87 (1991) 209–220.

[10] O. Lee, K. Yi, A proof method for the correctness of modularized kCFAs, Technical Memorandum ROPAS-2000-9, Research on program analysis system, Korea Advanced Institute of Science and Technology, Nov. 2000, URL http://ropas.kaist.ac.kr/memo/LeYi2000b.ps.gz.

[11] A.K. Wright, S. Jagannathan, Polymorphic splitting: An effective polyvariant flow analysis, ACM Trans. Prog. Lang. Syst. 20 (1) (1998) 166–207.