Journal of Southeast University (English Edition) Vol. 24, No. 2, pp. 178 – 182 June 2008 ISSN 1003—7985

Monadically slicing programs with procedures

Zhang Yingzhou Zhang Weifeng

(College of Computer, Nanjing University of Posts and Telecommunications, Nanjing 210003, China)

Abstract: A two-phase monadic approach is presented for monadically slicing programs with procedures. In the monadic slice algorithm for interprocedural programs, phase 1 initializes the slice table of formal parameters in a procedure with the given labels, and then captures the callees' influence on callers when analyzing call statements. Phase 2 captures the callees' dependence on callers by replacing all given labels appearing in the corresponding sets of formal parameters. By the introduction of given labels, this slice method can obtain similar summary information in system-dependence-graph(SDG)-based algorithms for addressing the calling-context problem. With the use of the slice monad transformer, this monadic slicing approach achieves a high level of modularity and flexibility. It shows that the monadic interprocedural algorithm has less complexity and it is not less precise than SDG algorithms.

Key words: program slicing; monadic semantics; interprocedural slicing; calling-context problem

Program slicing[1] is an effective technique for narrowing the focus of attention to the relevant parts of a program. Program slicing has been widely used in many software activities[2-3]. Program slicing can be classified into static slicing and dynamic slicing. This paper focuses on the static slicing methods.

The original program slicing method was expressed as a sequence of data flow analysis problems[1]. An alternative approach relied on program dependence graphs (PDG)[4]. Most of the existing slicing methods are evolved from the two approaches. The current slicing methods, however, are singular, and are mainly based on reachability problems of PDG or SDG (system dependence graph)[5]. In addition, the existing slicing methods are incremental and sequential, not compositional. However modern programming languages support modularized programming, and programs consist of a set of modules. So the program analysis should reflect this design technology, and the analysis methods (including program slicing) should be flexible and reusable for improving efficiency. To solve these problems, we present in Refs. [6 – 7] a novel formal method of program slicing, called modular monadic program slicing, which is based on modular monadic semantics. This paper extends it to slice interprocedural programs.

Weiser gave a data-flow method for interprocedural slicing where he directly used the existing intraprocedural slicing algorithm[1]. This method is simple and easy to understand and implement. However, Weiser's method does not account for the calling context of a called procedure; thus, the resulting slices may be imprecise[5]. Subsequently, Horwitz et al.[5] developed the SDG representation and a two-phase graph-reachability algorithm over the SDG to compute interprocedural slices. Their algorithm involves two steps: First, it constructs the corresponding SDG with summary edges, which represent transitive dependences due to procedure calls; secondly, it computes slices through two-phase traverses on the SDG. This slicing algorithm can address the calling-context problem, but the construction of the SDG may be complex.

1 Preliminaries

Because monads can avoid any reference to irrelevant semantic components when defining the semantics of each construct, they are a more disciplined notation than λ-notation whose unrestricted use makes denotational semantics lack modularity and reusability. In general, a monad is a triple $(m, \text{return}_m, \text{bind}_m)$, where m is a type constructor, and return_m and bind_m are its basic operators. A monad transformer consists of a type constructor t and its lifting function "lift", where t maps the monad, $(m, \text{return}_m, \text{bind}_m)$, to the new one, $(t\ m,\ \text{return}_{t\ m}, \text{bind}_{t\ m})$. The concept of lifting allows us to consider the interactions between various features. For more details of monads and monad transformers, please see Refs. [8 – 10].

There exist many monad transformers, such as state monad transformer StateT, environment monad transformer EnvT, error monad transformer ErrT[8,11], and slice monad transformer SliceT[6] which is shown as follows:

$$\begin{aligned}
&\text{type SliceT } L\ m\ a = L \to m(a, L)\\
&\text{return}_{\text{SliceT } L\ m}\ x = \lambda L.\ \text{return}_m(x, L);\\
&m\ \text{“bind}_{\text{SliceT } L\ m}\text{”}\ f = \lambda L.\ \{(a, L') \leftarrow m\ L; f\ a\ L'\}_m\\
&\text{lift}_{\text{SliceT } L} m = \lambda L.\ \{a \leftarrow m; \text{return}_m(a, L)\}_m;\\
&\text{updateSlice } f = \lambda L.\ \text{return}_m(f\ L, L)
\end{aligned}$$

Modular monadic semantics specifies the semantics of a language by mapping terms to computations. The flexibility of modular monadic program slicing allows us first to consider a simple language W, whose abstract syntax is as follows:

$$\begin{aligned}
S ::= &\ \text{ide} := \text{l. e} \mid S_1; S_2 \mid \text{skip} \mid \text{read ide} \mid \text{write l. e}\\
&\mid \text{if l. e then } S_1 \text{ else } S_2 \text{ endif}\\
&\mid \text{while l. e do } S \text{ endwhile}
\end{aligned}$$

We assume that the labeled expressions have no side-effects. The expressions, whose syntax is left unspecified for the sake of generality, are uniquely labeled. For constructing a syntactically valid slice result, we can define $\text{Syn}(s, L)$ for language W as is done in Refs. [6 – 7]. It allows us to con-

Received 2007-11-12.

Biography: Zhang Yingzhou (1978—), male, doctor, associate professor, zhangyz@njupt.edu.cn.

Foundation item: The National Outstanding Young Scientist Foundation by NSFC (No. 60703086, 60503020).

Citation: Zhang Yingzhou, Zhang Weifeng. Monadically slicing programs with procedures [J]. Journal of Southeast University (English Edition), 2008, 24(2): 178 – 182.

centrate on the labeled expressions in an analyzed program, since they are predominant parts in a program slice, and other parts can be captured through Syn(s, L).

2 Intraprocedural Monadic Slicing

In Refs. [6 − 7], we abstracted the computation of program slicing as a slice monad transformer SliceT L m(see section 1), where L denotes a set of labels of expressions that are required to compute the current expression. A slice monad transformer takes an initial set of labels, and returns a computation of a pair of the resulting value and a new set of labels. The lifting function lift says that a computation in the monad m behaves identically in the monad SliceT L m and makes no changes to the set of labels. The operation updateSlice supports the update of program slices.

For simplicity, we only consider end static slicing for a single variable, i. e. the slicing criterion is $\langle p, v\rangle$, where p is the end program point, and v the variable of interest. One can easily generalize this to a set of points and a set of variables at each point by taking the union of the individual slices. Based on modular monadic semantics of a program language, we gave the monadic static-slicing algorithm for single-procedure programs in Refs. [6 −7]. Its main idea can be briefly stated as follows: For obtaining the program slice w. r. t. a slicing criterion, we first apply the program-slice transformer SliceT to semantic description of the program analyzed. It makes the resulting semantic description include the program-slice semantic feature. According to this semantic description, we then analyze each statement in sequence. Finally we obtain the program slices of all single variables in the program through the final slice table Slices and Syn(s, L), including the program slice of the variable of interest.

3 Interprocedural Monadic Slicing

The modular monadic approach mentioned previously is flexible enough that we can easily introduce a new program feature for analysis. In this section, we will illustrate this power by considering an extension of the language W with call-by-value-result procedures.

The dependence relations among procedures can be divided into two categories. One is callees' influence on callers, since a callee (i. e. a program called) may change the data dependences in its callers. The other is callees' dependence on callers, since the execution of a callee depends on its callers. Both the relations should be considered during analyzing a call statement.

To analyze the first class of dependence relations, we focus on the dependencies among parameters. During the executive process of a sequence program, the influence of a callee on its callers can only be passed on by the corresponding formal parameters and the non-local variables. This allows us to first analyze independently each procedure, obtaining the dependencies among formal parameters, and then compute interprocedural slicing through the relations between the formal and the actual parameters[12].

In this paper, we consider the in-out formal parameter, corresponding to call by value-result. The non-local variables are treated as in out parameters. The return value of a function is treated as an out parameter, whose name is the same as the function name. So a function can be treated as equal to a procedure.

To obtain callees' dependence on callers, we need to gather all calling information of a procedure (such as A), i. e., the slice information of all call statements for calling A. In such calling information, there may be some given labels which need to be determined.

Therefore, we analyze the interprocedural slicing in two phases corresponding to the above dependence relations. In phase 1, we capture the callees' influence on callers when analyzing call statements. Meanwhile, we record for phase 2 some calling information of call statements. In phase 2, we determine(backfill/replace) all given labels in the calling information gathered in phase 1. Now we can obtain the final slice results of all procedures.

Phase 1 should pay much attention to call statements, and other statements can be analyzed in the same way as in the intraprocedural slicing method. According to the analysis above, the procedure call statements are as easy to handle as other statements(e. g. special loop statements). When analyzing statements with procedure calls, the monadic slices of the actual-parameter variables can be converted to those of the corresponding formal-parameter variables. Such slices of formal parameters can be obtained by the intraprocedural monadic slicing, and the results can be reused. Therefore, after the statement s with a procedure call is analyzed, the monadic slices of each variable x in lkpSli(x, Tcall) need to be updated as follows:

$$(\mathrm{lkpSli}(a, \mathrm{Tproc}) - \bigcup_{y \in \mathrm{Ref}(s,x)} \{l_b\}) \cup (\bigcup_{y \in \mathrm{Ref}(s,x)} \mathrm{lkpSli}(y, \mathrm{Tcall}) \cup \{l\} \cup L)$$

where Tcall is the table slice at the call site; a and b are the corresponding formal parameters of x and y, respectively; l_b is the given label; l is the label of the call statement; and L is the label which may influence the execution of the call statement.

In this way, the interprocedural monadic slicing is converted to a set of independent single intraprocedural monadic slicings. In phase 1, the procedures should be analyzed in a given order. This can be done by a call-relation set or call graph. For more details, see Ref. [12] where the analysis order of recursive procedures is also discussed.

For serving phase 2, we gather the calling information at call statements. Concretely, at each call statement, we record the information of all actual parameters as follows:

$$\{l\} \cup L \cup \mathrm{lkpSli}(\mathrm{ide}, \mathrm{getSli})$$

where ide is an actual parameter. Such information is saved in the table Lcall, which consists of several sub-tables, each representing the corresponding sets of all formal parameters of a procedure.

Phase 2 captures the callees' dependence on callers, and obtains the final slice table Tp of a procedure by determining all given labels in the table Tproc of this procedure.

To capture the callees' dependence on callers, we compute the corresponding sets(with no given labels) of all formal parameters through the table Lcall with given labels. We analyze in the reverse order of that in phase 1. For example, if procedure A calls procedure B, we compute the corresponding sets of all formal parameters in A before doing

those in B. For convenience, we write Lcall(A, a) to represent the corresponding set of the formal parameter a in A. If there is a given label l_a in Lcall(B, x), where x and a are respectively the formal parameters in B and A, we replace l_a with all the elements of Lcall(A, a). Since we begin to analyze B after doing A, there will be no given labels in Lcall (A, a). As a result, we can obtain the final Lcall(B, x) with no given labels.

Through Lcall with no given labels, we can easily obtain the final slice table Tp of a procedure. In the concrete, we can first copy the table Tproc of a procedure(e. g. C) as Tp, then replace each given label, say l_b, in Tp by all elements of Lcall(C, b), where b is one of the formal parameters in C. The resulting table Tp(with no given labels) is the final monadic slice table of C.

As for the extension of the language W with procedures, its syntax is as follows:

$$D::= \text{procedure ide}(\text{arg}_1, \ldots, \text{arg}_n) \text{ is } S$$

$$S::= \text{call l. ide}(\text{ide}_1, \ldots, \text{ide}_n) \mid \ldots \text{ as before } \ldots$$

where D belongs to the domain of declarations. We need to add some slice operators into the semantic descriptions of procedures, shown as follows:

```
[call l. ide(ide_1, ..., ide_n)]
  = λL. {loc_1 ←lkpEnv(ide_1, rdEnv); ...;
         loc_n ←lkpEnv(ide_n, rdEnv);
         L_C1 ← {l} ∪ L ∪ lkpSli(ide_1, getSli); ...;
         L_Cn ← {l} ∪ L ∪ lkpSli(ide_n, getSli);
         proc ←lkpEnv(l. ide, rdEnv);
         Tcall ← getSli;
         proc(loc_1, ..., loc_k+n);
         setSli(Tproc_ide);
    updSli(arg_1, ∪_{arg_i ∈ Dep(ide, arg_1)} (lkpSli(arg_1, getSli) − {l_i}) ∪
L_Ci, getSli);
    ...; updSli(arg_n, ∪_{arg_i ∈ Dep(ide, arg_n)} (lkpSli(arg_n, getSli) −
{l_i}) ∪ L_Ci, getSli);
    Tpcall ←getSli;
    setSli(Tcall);
    updSli(ide_1, lkpSli(arg_1, Tpcall), getSli);
    ...;
    updSli(ide_n, lkpSli(arg_n, Tpcall), getSli);
    Lcall(ide) ← Lcall(ide) ∪
        [(arg_1, L_C1), ..., (arg_n, L_Cn)]
    }
[procedure ide(arg_1, ..., arg_n: in out) is S]
  = {let proc = λloc_1 ... loc_n. {ρ ←rdEnv;
    ρ' ←{xtdEnv(arg_1, lkpSto(loc_1), ρ);
    ...; xtdEnv(arg_n, lkpSto(loc_n), ρ)};
    if Tproc = Null then
        Tini ← [(arg_1, {l_1}), ..., (arg_n, {l_n})];
        setSli(Tini);
        inEnv ρ' [S];
        updSto(loc_1, lkpSto(lkpEnv(arg_1, rdEnv)));
        ...; updSto(loc_n,
            lkpSto(lkpEnv(arg_n, rdEnv)));
        Tproc_ide ← getSli;
    For all i, j: 1 < = i < = n ∧ 1 < = j < = n
        if l_j ∈ lkpSli(arg_i, getSli) then
      Dep(ide, arg_i) ←Dep(ide, arg_i) ∪ {arg_j};
    };
    xtdEnv(ide, return proc, ρ')
  }
```

For obtaining the final results of monadic interprocedural slicing, we also need to add the rules below to the definition of Syn (s, L) given in Refs. [6 −7]:

"procedure ide$(\text{arg}_1, \ldots, \text{arg}_n)$ is S":
  if Syn$(S, L) = \varepsilon$ then ε
  else "procedure ide$(\text{arg}_1, \ldots, \text{arg}_n)$ is S"
"call l. ide$(\text{ide}_1, \ldots, \text{ide}_n)$":
  if $l \in L$ then "call l. ide$(\text{ide}_1, \ldots, \text{ide}_n)$"
  else ε

4 A Case Study and Complexity Analysis

To illustrate our two-phase interprocedural slicing algorithm further, we consider a concrete program[5] as follows:

```
Procedure Main()
 1 sum: =0;
 2 i: =1;
 3 while i<11 do
 4     call A(sum, i);
   endwhile
Procedure A(x, y)
 5 call Add(x, y);
 6 call Inc(y);
   skip
Procedure Add(a, b)
 7 a: =a+b;
   skip
Procedure Inc(z)
 8 call Add(z, 1);
   skip
```

In phase 1, based on the call graph of the example program, we can obtain the analysis order of the example program as

$$\text{Add} \rightarrow \text{Inc} \rightarrow A \rightarrow \text{Main}$$

Before analyzing procedures in this order, we need to initialize Tproc with the given labels; that is to say, Tproc_Add, Tproc_Inc and Tproc_A are initialized as $[(a, \{l_a\}), (b, \{l_b\})]$, $[(z, \{l_z\})]$ and $[(x, \{l_x\}), (y, \{l_y\})]$, respectively. At the same time, we make the other tables(such as Tp, Dep, Lcall) null.

According to the analysis order, the procedure Add is first analyzed. Since there is no call statement in Add, it is easy to get its final slice table Tproc_Add(given in Tab. 1) through the intraprocedural monadic algorithm. From Tproc_Add, the dependencies among the formal parameters in Add can be obtained(see Tab. 2):

$$\text{Dep}(a) = \{a, b\}, \quad \text{Dep}(b) = \{b\}$$

The 8h statement in Inc is a call statement. According to the monadic algorithm in section 3, before entering Add, the

temporary label sets representing the dependence need to be computed:

$$L_{\mathrm{Ca}} = \{8\} \cup L \cup \mathrm{lkpSli}(z, \mathrm{getSli}) = \{8, l_z\}$$

$$L_{\mathrm{Cb}} = \{8\} \cup L = \{8\}$$

Tab. 1 The slice table Tproc

Proc	Var	Labels
Add	a	$\{7, l_a, l_b\}$
	b	$\{l_b\}$
Inc	z	$\{7, 8, l_z\}$
A	x	$\{5, 7, l_x, l_y\}$
	y	$\{6, 7, 8, l_y\}$
Main	sum	$\{1, 2, 3, 4, 5, 7\}$
	i	$\{2, 3, 4, 6, 7, 8\}$

Tab. 2 The dependence table Dep

Proc	Var	Dep
A	x	$\{x, y\}$
	y	$\{y\}$
Add	a	$\{a, b\}$
	b	$\{b\}$
Inc	z	$\{z\}$

This is also the calling information that should be saved in the table Lcall (shown in Tab. 3). Because Add has been analyzed, we can obtain the table Tpcall directly from Tproc_Add and Dep. With the help of Tpcall, we update Tcall according to the relationship between the formal and the actual parameters. Since the 8h statement is also the last statement in Inc, Tproc_Inc (see Tab. 1) can be obtained by copying the Tcall. Moreover, the dependencies among the formal parameters can also be obtained (shown in Tab. 2): $\mathrm{Dep}(z) = \{z\}$.

Tab. 3 The table Lcall

Proc	Var	Labels
A	x	$\{1, 2, 3, 4\}$
	y	$\{2, 3, 4\}$
Add	a	$\{5, 8, l_x, l_z\}$
	b	$\{5, 8, l_y\}$
Inc	z	$\{6, l_y\}$

Similarly, we analyze A and Main in turn, and obtain the tables Tproc_A, Tproc_Main, Lcall and Dep.

In phase 2, the main work is to replace all given labels appearing in the corresponding sets of formal parameters (in Lcall). The analysis order for replacing is the reverse one in phase 1:

$$\mathrm{Main} \rightarrow A \rightarrow \mathrm{Inc} \rightarrow \mathrm{Add}$$

Since Main has no formal parameters, we need not replace it. Because the corresponding set of A (in Lcall) has no given label, we also need not replace it. There is a given label l_y in the Lcall(Inc, z) (shown in Tab. 3), so we need to replace l_y with all the elements of Lcall(A, y). Now Lcall(Inc, z) is updated to $\{2, 3, 4, 6\}$, which has no given label. Similarly, we can obtain the Lcall(Add, a) and Lcall(Add, b) with no given labels, and the final results are shown as follows:

$$\mathrm{Lcall}(A, x) = \{1, 2, 3, 4\}$$

$$\mathrm{Lcall}(A, y) = \{2, 3, 4\}$$

$$\mathrm{Lcall}(\mathrm{Inc}, z) = \{2, 3, 4, 6\}$$

$$\mathrm{Lcall}(\mathrm{Add}, a) = \{1, 2, 3, 4, 5, 6, 8\}$$

$$\mathrm{Lcall}(\mathrm{Add}, \mathrm{b}) = \{2, 3, 4, 5, 8\}$$

Through the new Lcall, we can obtain the final slice tables Tp (given in Tab. 4) from Tproc by replacing their corresponding given labels.

Tab. 4 The final slice table Tp

Proc	Var	Labels
Main	sum	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
	i	$\{2, 3, 4, 6, 7, 8\}$
A	x	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
	y	$\{2, 3, 4, 6, 7, 8\}$
Inc	z	$\{2, 3, 4, 6, 7, 8\}$
Add	a	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
	b	$\{2, 3, 4, 5, 8\}$

On the basis of the complexity of the intraprocedural slicing algorithm in Ref. [7], we discuss in this section the complexity of our two-phase monadic slicing algorithm in section 3. Besides the cost of the intraprocedural slicing algorithm, the interprocedural slicing algorithm needs additional time cost for call statements and backfilling (replacing) given labels, and additional space for saving the Hash tables Tp, Dep and Lcall.

We first suppose s, p and m are the numbers of call statements, procedures and labeled expressions in the entire program, respectively; para is the largest number of formal parameters in any procedure, and v is the largest number of variables in any procedure. Then, we will analyze the time cost of the additional parts mentioned above. Each call statement will cost time $O(\mathrm{para} \times \mathrm{para})$. The set of reference parameters in procedures may cost time $O(p \times \mathrm{para} \times \mathrm{para})$. Thus, the total cost in phase 1 of the algorithm is $O(\mathrm{s} \times \mathrm{para} \times \mathrm{para} + \mathrm{p} \times \mathrm{para} \times \mathrm{para})$. In phase 2, obtaining the new corresponding sets may cost $O(\mathrm{para} \times p \times \mathrm{para} \times p)$, and it may cost $O(p \times v \times \mathrm{para})$ to obtain the final Tp; thus, the total cost is time $O(\mathrm{para} \times p \times \mathrm{para} \times p + p \times \mathrm{para} \times v)$. So the additional time cost is $O(s \times v \times \mathrm{para} + \mathrm{para} \times p \times \mathrm{para} \times p + p \times \mathrm{para} \times v)$.

As for the space complexity, we also only focus on the additional parts—Tproc, Tp, Dep and Lcall. In the worst case, the table Tproc, Tp, Dep and Lcall may need space $O(v \times p \times m)$, $O(v \times p \times m)$, $O(p \times \mathrm{para} \times \mathrm{para})$ and $O(\mathrm{para} \times p \times m)$, respectively. Therefore, the total space cost of the additional parts is $O(v \times p \times m + p \times \mathrm{para} \times \mathrm{para} + p \times \mathrm{para} \times m)$.

According to the complexity analysis, our algorithm has less complexity than the SDG-based algorithms (since SDG need not be constructed in our algorithm).

5 Conclusion

In this paper, we propose an approach for interprocedural slicing, i. e. , backfilling-given-labels based two-phase monadic slicing, which is based on modular monadic semantics of programming languages. With the introduction of given labels, our method can obtain similar summary information for addressing the calling-context problem.

Compared with the SDG-based interprocedural slicing methods, our method only needs to save some tables such as Tp, Dep and Lcall. Instead of maintaining a SDG which may cost much space, our method saves some space. In addition, our monadic algorithm is not less precise than SDG-based ones as shown from the examples in section 4. This is because the term L and $\bigcup_{r \in \mathrm{Refs}(l.\,e)} \mathrm{lkpSli}\ (r, \mathrm{getSli})$ in the definition of L' can accurately capture control dependences and data dependences respectively, which are the bases of SDG-based algorithms.

References

[1] Weiser M. Program slicing [J]. *IEEE Transaction on Software Engineering*, 1984, **16**(5): 498 - 509.

[2] Tip F. A survey of program slicing techniques [J]. *Journal of Programming Languages*, 1995, **3**(3): 121 - 189.

[3] Gallagher K B, Lyle J R. Using program slicing in software maintenance [J]. *IEEE Transactions on Software Engineering*, 1991, **17**(8): 751 - 761.

[4] Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment [J]. *ACM SIGPLAN Notices*, 1984, **19**(5): 177 - 184.

[5] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs [J]. *ACM Transactions on Programming Languages and Systems*, 1990, **12**(1): 26 - 60.

[6] Zhang Y Z, Xu B W, Shi L, et al. Modular monadic program slicing [C]//*The 28th Annual International Computer Software and Applications Conference*. Hong Kong, China, 2004: 66 - 71.

[7] Zhang Y Z, Xu B W. A novel formal approach to program slicing [J]. *Science in China*, *Ser E*, *Info Sci*, 2008, **38**(2): 161 - 320.

[8] Wansbrough K. A modular monadic action semantics [D]. Auckland: University of Auckland, 1997.

[9] Moggi E. An abstract view of programming languages [R]. Edinburgh: University of Edinburgh, 1989.

[10] Liang S. Modular monadic semantics and compilation [D]. New Haven: University of Yale, 1998.

[11] Liang S, Hudak P, Jones M. Monad transformers and modular interpreters [C]//*22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1995: 333 - 343.

[12] Xu B W, Chen Z Q. Dependence analysis for recursive java programs [J]. *ACM SIGPLAN Notices*, 2001, **36**(12): 70 - 76.

含过程程序的单子切片

张迎周　　张卫丰

(南京邮电大学计算机学院,南京 210003)

摘要:为解决含过程程序的单子切片问题,提出基于回填待定标号的两阶段单子切片算法. 算法第 1 阶段用给定标号初始化子过程的形参切片表,并通过分析调用语句捕获被调者对调用者的影响. 算法第 2 阶段主要是通过回填切片表中相应的待定标号来捕获施调者对被调者的影响. 待定标号的引入使得所提切片算法可捕获类似基于系统依赖图(SDG)切片算法中的概要信息,且也可避免上下文调用问题. 借助于切片单子转换器,所提单子切片算法将具有较高的模块性和适应性,且其复杂度不劣于基于 SDG 的切片算法.

关键词:程序切片;单子语义;过程间切片;上下文调用问题

中图分类号:TP311