A Formal Method for Program Slicing*

Yingzhou Zhang[1,2], Baowen Xu[1,2], Jose Emilio Labra Gayo[3]
([1]*Dep. of Computer Science and Engineering, Southeast Univ.*, Nanjing 210096, China)
([2]*Jiangsu Institute of Software Quality*, Nanjing 210096, China)
([3]*Dep.of C*omputer Science, Univ. of Oviedo, C/Calvo Sotelo s/n C.P. 33007, Oviedo, Spain)
mathzyz@163.com, *bwxu@seu.edu.cn*, *labra@lsi.uniovi.es*

Abstract

Program slicing is a well-known program analysis technique that extracts the elements of a program related to a particular computation. Based on modular monadic semantics of a programming language, this paper presents a new formal method for slicing, called modular monadic slicing*, by abstracting the computation of slicing as a* slice monad transformer*. With the use of slice transformer, the feature of program slicing can be combined in a modular way into semantic descriptions of the program analyzed. According to these, this paper gives both monadic dynamic and static slicing algorithms. They compute program slices directly on abstract syntax, without the needs to explicitly construct intermediate structures such as dependence graphs, or to record an execution history in dynamic slicing algorithm.*

1. Introduction

Program slicing is a well-known program analysis technique that extracts the elements of a program related to a particular computation. A program slice consists of those statements of a program that may directly or indirectly affect the variables computed at a given program point, referred to as a slicing criterion. Program slicing has applications in program comprehension, testing and debugging, re-engineering, and software maintenance [1-4].

In reverse engineering, program slicing provides a toolset for abstracting out of the source codes the design decisions and rationale from the initial development and understanding the algorithms chosen. In software maintenance, program slicing can help maintainers to determine whether a change at some place in a program will affect the behavior of other parts of the program. Program slicing can be used in software quality assurance to locate all code that contributes to the value of variables that might be part of a safety critical component.

The original program slicing method was expressed as a sequence of data flow analysis problems [5]. An alternative approach relied on program dependence graphs (PDG) [6]. Most of the existing slicing methods were evolved from these two approaches. As the behavior of a program is determined by the semantics of the language, it is reasonable to expect an approach for program slicing based on formal semantics of a program.

The program slicing methods focused on the semantics of programs are mainly based on the standard denotational semantics, i.e. *denotational slicing* [7-9]. Denotational semantics, however, lack modularity and reusability [10-14]. A practicable solution was to use *monads* [15] to structure denotational semantics, with the help of *monad transformers* [10, 16, 17] which can transform a given monad into a new one with new operations. S.Liang et al. used monads and monad transformers to specify the semantics of programming language and called it *modular monadic semantics* [18]. Based on this, this paper proposes a first approach for program slicing based on modular monadic semantics, called *modular monadic slicing*. It can compute slices directly on abstract syntax, without explicit construction of intermediate structures such as dependence graphs in the corresponding slicers.

The rest of the paper is organized as follows: In Section 2, we briefly introduce the fundamental concepts of modular monadic semantics through a simple example language. The monadic program slicing algorithms and their complexity are discussed in Section 3. In Section 4, we illustrate our monadic slicing algorithm by analyzing in detail a sample of the example

* This work was supported in part by the National Natural Science Foundation of China (60373066, 90412003), Young Scientist's Foundation of NSFC (60303024), National Research Foundation for the Doctoral Program of Higher Education of China (20020286004).

language, with the results from our slicer prototype tool under development. In Section 5, our novel algorithms are discussed in associated with related works. We conclude this paper with directions for future work in Section 6.

2. Preliminaries

Monads were discovered in category theory in the 1950s and introduced to the semantics community by Moggi in [15]. Monads are an abstract technique for encapsulating details of impure features such as states, nondeterminism and I/O used by the computations. Formally, a monad is a triple (m, $\mathbf{return}_m$, $\mathbf{bind}_m$), where m is a type constructor (a map from each type a to a corresponding type $m\ a$); $\mathbf{return}_m$ and $\mathbf{bind}_m$ are two primitive operators:

$$\mathbf{return}_m :: \ a \rightarrow m\ a$$
$$\mathbf{bind}_m :: \ m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

$\mathbf{return}_m\ a$ is a trivial computation that just return a as the result, whereas m '$\mathbf{bind}_m$' k computes m and passes the result to the rest of the computation k.

Intuitively, a monad is a transformation on types equipped with a composition method for transformed values. To add a new feature to a monadic semantics, we only need to add a semantic description of the new feature, and change the underlying monad, but not the semantic descriptions of the existing features. Traditional denotational semantics maps, say, a term, an environment and a continuation to an answer. In contrast, monadic semantics maps terms to computations, where the details of the environment, store, etc. are "hidden". The monadic style in which the descriptions are written is far easier to read than a typical denotational semantic description.

Moggi realized that for realistic semantics features had to be combined, and so he presented *monad constructors* that could add new notions of computation to a monad. D.Espinosa called them *monad transformers* in his system Semantic Lego [17]. A monad transformer consists of a type constructor t and an associated function $lift_t$, where t maps any given monad (m, $return_m$, $bind_m$) to a new monad ($t\ m$, $return_{t\,m}$, $bind_{t\,m}$); $lift_t$ is a function of type:

$$lift_t : \ m\ a \rightarrow t\ m\ a$$

Monad transformers provide the power needed to represent the abstract notion of programming language features, but still allow us to access low-level semantic details. Multiple monad transformers can be composed to form the underlying monad used for the semantic specification of the high-level language.

S :: = ide := l.e | S_1; S_2 | skip
| **read** ide | **write** l.e
| **if** l.e **then** S_1 **else** S_2 **endif**
| **while** l.e **do** S **endwhile**

Figure 2. Abstract syntax of W

The key of modular monadic semantics is the division of the monad m into a series of monad transformers, each representing a computation. What's more, *monad transformers can be designed once and for all* [12], because they are entirely independent of the language being described. Inspired by this, we try to abstract the computation of program slicing as a monad transformer. This will be discussed in next section.

Environment monad transformer:

type EnvT $\rho\ m\ a = \rho \rightarrow m\ a$
$\mathbf{return}_{\mathrm{EnvT}\ \rho\ m}\ x = \lambda\rho.\ \mathbf{return}_m\ x$
x '$\mathbf{bind}_{\mathrm{EnvT}\ \rho\ m}$' $f = \lambda\rho.\ \{a \leftarrow x\ \rho\ ;\ f\ a\ \rho\}_m$
$lift_{\mathrm{EnvT}\ \rho}\ x = \lambda\rho.\ x$ '$\mathbf{bind}_m$' $\mathbf{return}_m$
$rdEnv = \lambda\rho.\ \mathbf{return}_m\ \rho$
$inEnv\ \rho\ x = \lambda\_.\ x\ \rho$

State monad transformer:

type StateT $s\ m\ a = s \rightarrow m\ (s, a)$
$\mathbf{return}_{\mathrm{StateT}\ s\ m}\ x = \lambda s.\ \mathbf{return}_m\ (s, x)$
x '$\mathbf{bind}_{\mathrm{StateT}\ s\ m}$' $f = \lambda s.\ \{(s', a) \leftarrow x\ s\ ;\ f\ a\ s'\}_m$
$lift_{\mathrm{StateT}\ s}\ x = \lambda s.\ \{a \leftarrow x\ ;\ \mathbf{return}_m\ (s, a)\}_m$
$getState = \lambda s.\ \mathbf{return}_m\ (s, s)$
$setState\ s = \lambda\_.\ \mathbf{return}_m\ ((), s)$

Error monad transformer:

type Err a = Ok a | Err String
type ErrT $m\ a = m$ (Err a)
$\mathbf{return}_{\mathrm{ErrT}\ m}\ x = \mathbf{return}_m$ (Ok x)
x '$\mathbf{bind}_{\mathrm{ErrT}\ m}$' f = { $a \leftarrow x$; **case** a **of**
Ok y : $f\ y$
Err s: $\mathbf{return}_m$ (Err s) }$_m$
$lift_{\mathrm{ErrT}}\ x = \{a \leftarrow x$; $\mathbf{return}_m$ (Ok a) }$_m$
err s = $\mathbf{return}_m$ (Err s)

Input-Output monad transformer:

type IOT $m\ a$ = String $\rightarrow m\ (a$, String)
$\mathbf{return}_{\mathrm{IOT}\ m}\ a = \lambda s.\ \mathbf{return}_m\ (a, s)$
x '$\mathbf{bind}_{\mathrm{IOT}\ m}$' $f = \lambda s.\ \{(a, s') \leftarrow x\ s\ ;\ f\ a\ s'\}_m$
$lift_{\mathrm{IOT}\ s}\ x = \lambda s.\ \{a \leftarrow x$; $\mathbf{return}_m\ (a, s)\}_m$
$getvalue = \lambda s.\ \mathbf{return}_m\ (s, s)$
$putvalue\ s = \lambda\_.\ \mathbf{return}_m\ (s, ())$

Figure 1. Some common monad transformers

Figure 1 describes some common monad transformers which are similar to the ones given in [12, 15, 16, 18, 19].

For the purpose of this paper, we will focus our attention on a simple imperative language **W**. Its abstract syntax is provided in Figure 2, where *S* ranges over statements **Stmt**, ide ranges over a set of identifiers **Ide**, and l.e range over a set of labeled expressions **Exp**. The expressions, whose syntax is left unspecified for the sake of generality, are uniquely labeled. We assume that the labeled expressions have no side-effects.

In modular monadic semantics, the monad definition is simply a composition of the corresponding monad transformers, applied to a base monad. In this paper, we use the input/output monad IO as the base monad. We then select some monad transformers such as EnvT, StateT, ErrT showed in Figure 1, and apply them to the base monad IO, forming the resulting monad ComptM:

$$\text{ComptM} \equiv (\text{EnvT} \cdot \text{StateT} \cdot \text{ErrT})\ \text{IO}$$

Now the formal semantic description of language **W** can be given in Figure 3. In Figure 3, the identifier ***Fix*** denotes a fixpoint operator; *xtdEnv* and *lkpEnv* are the update and lookup operator of environments Env, respectively; *updSto* is the update function of stores Loc; *rdEnv* and *inEnv*, *putValue* and *getValue* are the basic operators of EnvT and IOT showed in Figure 1, respectively.

To conveniently discuss program slices later, following G.A.Venkatesh's idea in [9], we define *Syn*(s, *L*) for language **W** in Figure 4, where s is a W-program analyzed, v the variable of interest in a slicing criterion, ε null result, and *Refs*(l.e) the set of variables occurring in expression l.e. It guides us how to construct a syntactically valid subprogram of s from the set of labels of labeled expressions *L*. Furthermore, it allows us to concentrate on the labeled expressions in a program analyzed, since they are predominant parts in a program slice, and other parts can be captured through *Syn*(s, *L*).

Domains:

loc: Loc (Stores); *v*: Value (Values)

Semantics Functions:

$\boldsymbol{S}$ :: Stmt → ComptM ()

$\boldsymbol{E}$ :: Exp → ComptM Value

$\boldsymbol{S}[\![$ ide := l.e $]\!] = \{v \leftarrow \boldsymbol{E}[\![$ l.e $]\!];\ loc \leftarrow lkpEnv(\text{ide}, rdEnv);\ updSto(loc, return\ v)\}$

$\boldsymbol{S}[\![S_1; S_2]\!] = \{\boldsymbol{S}[\![S_1]\!];\ \boldsymbol{S}[\![S_2]\!]\}$

$\boldsymbol{S}[\![$ skip $]\!] = return\ ()$

$\boldsymbol{S}[\![$ **if** l.e **then** S_1 **else** S_2 **endif** $]\!] = \{v \leftarrow \boldsymbol{E}[\![$ l.e $]\!];$
case *v* **of** **tt** → $\boldsymbol{S}[\![S_1]\!];$
ff → $\boldsymbol{S}[\![S_2]\!]$ }

$\boldsymbol{S}[\![$ **while** l.e **do** *S* **endwhile** $]\!] = \boldsymbol{Fix}\ (\lambda f.\{v \leftarrow \boldsymbol{E}[\![$ l.e $]\!];$
case *v* **of** **tt** → $f \cdot \boldsymbol{S}[\![S]\!];$
ff → *return* () })

$\boldsymbol{S}[\![$ **read** ide $]\!] = \{loc \leftarrow lkpEnv(\text{ide}, rdEnv);\ v \leftarrow getValue;\ updSto(loc, return\ v)\}$

$\boldsymbol{S}[\![$ **write** l.e $]\!] = \{\ v \leftarrow \boldsymbol{E}[\![$ l.e $]\!];\ putValue\ (return\ v)\ \}$

Figure 3. Modular monadic semantics of W

Syn(s, *L*) =
case s of
"ide := l.e " : if l ∈ *L* then "ide := l.e" else ε
"S_1; S_2 " : *Syn*(S_1, *L*); *Syn*(S_2, *L*)
"skip " : ε
"**read** ide " : if ide ∈ {v} ∪ $\bigcup_{l \in L} Refs(\text{l.e})$ then "**read** ide" else ε
"**write** l.e " : if v ∈ *Refs*(l.e) then "**write** l.e" else ε
"**if** l.e **then** S_1 **else** S_2 **endif** " : if (*Syn*(S_1, *L*) = *Syn*(S_2, *L*) = ε)∧(l ∉ *L*) then ε
else "**if** l.e **then** *Syn*(S_1, *L*) **else** *Syn*(S_2, *L*) **endif** "
"**while** l.e **do** *S* **endwhile** " : if (*Syn*(*S*, *L*) = ε)∧(l ∉ *L*) then ε
else "**while** l.e **do** *Syn*(*S*, *L*) **endwhile** "

Figure 4. The definition of *Syn*(s, *L*)

3. Modular Monadic Program Slicing

3.1 Slice Monad Transformer

In this section, we try to abstract the computation of program slicing as an independent entity, *slice monad transformer*. This work is significant because a monad transformer can be designed once and for all. We give the definition of slice monad transformer in Figure 5, where L denotes a set of labels of expressions that were required to compute the current expression.

A slice monad transformer SliceT L m, takes an initial set of labels, and returns a computation of a pair of the resulting value and the new set of labels. The lifting function $lift_{SliceT\ L}$ 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 update of program slices.

In form, the slice monad transformer is similar to the state monad transformer. So, the correctness proof of its definition can be obtained easily by following the proofs for transformer StateT in [12] or [20].

Slice monad transformer:

type SliceT L m a = $L \rightarrow m\ (a, L)$

$\mathbf{return}_{\text{SliceT } L\ m}\ x = \lambda L.\ \mathbf{return}_m\ (x, L)$

m '$\mathbf{bind}_{\text{SliceT } L\ m}$' $f = \lambda L.\ \{(a, L') \leftarrow m\ L;\ f\ a\ L'\}_m$

$lift_{\text{SliceT } L}\ m = \lambda L.\ \{a \leftarrow m;\ \mathbf{return}_m\ (a, L)\}_m$

updateSlice $f = \lambda L.\ \mathbf{return}_m\ (f\ L, L)$

Figure 5. Slice monad transformer

3.2 Monadic Program Slicing Algorithms

For simplicity, we only consider end slicing with respect to a slicing criterion <p, v>, where p is the end point of a program, and v a variable. This can be easily generalized to a set of points and a set of variables at each point by taking the union of the individual slices [2].

Based on modular monadic semantics of a program language, the monadic algorithms for dynamic slicing and static slicing are provided in Figure 6 (where INPUT denotes the actual input during an execution), and Figure 7 respectively. These two algorithms are very similar except for Step 3. The main idea of monadic slicing algorithm can be briefly stated as follows: for obtaining the program slice w.r.t. a slicing criterion, we firstly 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 execute this program with an input (for dynamic slicing) or analyze each statement in sequence (for static slicing). Finally we will obtain the program slices of all single variables in the program, including the program slice of the variable of interest.

In Step 1, we initialize the set of labels, L, and all original slices in the table Slices with null set. The data structure of program slices Slices is defined as follows:

type Var = String
type Labels = [Int]
type Slices = [(Var, Labels)]
getSli :: ComptM Slices
setSli :: Slices → ComptM Slices
lkpSli :: Var → Slices → ComptM Labels
xtdSli ::
(Var, ComptM Labels)→Slices→ComptM ()
mrgSli :: Slices → Slices → ComptM Slices

where [] denotes a table data structure. Five operators *getSli*, *setSli*, *lkpSli*, *xtdSli* and *mrgSli*, represent to return and setup the current table of program slices, lookup, update and merge a slice corresponding to a variable in a given table of slices, respectively.

In Step 2, we want to combine the feature of program slicing into semantic descriptions through monad transformer SliceT given in Figure 5. As for language **W**, we compose SliceT with other transformers

Input: Slicing criterion <INPUT, p, v>
Output: Dynamic slice
1. Initialize the set L and the table Slices.
2. Add the feature of program slicing into semantic descriptions in a modular way, through slice transformer SliceT.
3. Execute the program analyzed with INPUT in accordance with the semantic description in Step 2, obtaining the final Slices.
4. Returning the final dynamic slicing result according to Slices and *Syn*(s, L), where L = *lkpSli* (v, *getSli*).

Figure 6. Dynamic slicing algorithm

Input: Slicing criterion <p, v>
Output: Static slice
1. Initialize the set L and the table Slices.
2. Add the feature of program slicing into semantic descriptions in a modular way, through slice transformer SliceT.
3. Compute static slices of each statement in sequence basing on the semantic description in Step 2, obtaining the final Slices.
4. Returning the final static slicing result according to Slices and *Syn*(s, L), where L = *lkpSli* (v, *getSli*).

Figure 7. Static slicing algorithm

(e.g. EnvT, StateT, ErrT) to form the resulting monad ComptM as follows:

$$\text{ComptM} \equiv (\text{SliceT} \cdot \text{StateT} \cdot \text{EnvT} \cdot \text{ErrT})\ \text{IO}$$

where the order of monad transformers can be changed at random. Meanwhile, we need to reify the intermediate set L' in $\mathbf{bind}_{\text{SliceT } L\ m}$ as following:

$$L' = \{\text{l}\} \cup L \cup \bigcup_{r \in Refs(\text{l.e})} lkpSli\ (r, getSli) \qquad (*)$$

This relation reflects when and how to change the set L. In addition, for recording the result of program slicing, we ought to add the operator *xtdSli* into semantic descriptions of assignment statements as the following bold terms:

$[\![\text{ide} := \text{l.e}]\!]$
$= \lambda L.\{\ v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];$
$L' \leftarrow \{\text{l}\} \cup L \cup \bigcup_{r \in Refs(\text{l.e})} lkpSli\ (r, getSli)\ ;$
$loc \leftarrow lkpEnv(\text{ide}, rdEnv);$
$updSto(loc, return\ v);$
$\boldsymbol{xtdSli(\text{ide}, L', getSli)}\ \}$

Similar to the way in *forward dynamic slicing* [21], in our modular monadic approach, the program slices for associated variables of each statement are computed immediately after this statement is executed/analyzed. After the last statement is executed/analyzed, the individual program slices for all variables of the program executed have been obtained.

Concretely, in monadic dynamic slicing algorithm, we compute dynamic slices while executing the program analyzed with INPUT (cf. Step 3 in Figure 6), according to the semantic description including the feature of program slicing. After the last statement is executed, we obtain a table Slices that includes individual dynamic slices for all variables. In contrast, in static slicing algorithm, we need to capture the statements that possibly affect the variable in the slicing criterion, besides those that actually affect this variable as dynamic slicing do. Therefore, we ought to modify semantic descriptions of conditional statement and loop statement, and to add in the operator *mrgSli* as following.

$[\![\textbf{if}\ \text{l.e}\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}]\!]$
$= \lambda L.\{\ v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];$
$L' \leftarrow \{\text{l}\} \cup L \cup \bigcup_{r \in Refs(\text{l.e})} lkpSli\ (r, getSli)\ ;$
$\text{T} \leftarrow getSli\ ; \text{T1} \leftarrow \{[\![S_1]\!] L'; getSli\};$
$setSli(\text{T}); \text{T2} \leftarrow \{[\![S_2]\!] L'; getSli\};$
$mrgSli(\text{T1}, \text{T2})\ \}$

$[\![\textbf{while}\ \text{l.e}\ \textbf{do}\ S\ \textbf{endwhile}]\!]$
$= \boldsymbol{Fix}\ (\lambda f.\ \lambda L.\{v \leftarrow \boldsymbol{E}[\![\text{l.e}]\!];$
$L' \leftarrow \{\text{l}\} \cup L \cup \bigcup_{r \in Refs(\text{l.e})} lkpSli\ (r, getSli)\ ;$
$\text{T} \leftarrow getSli;$
$\text{T}' \leftarrow \{f\, L' \cdot [\![S]\!] L'; getSli\};$
$mrgSli(\text{T}, \text{T}')\quad \}\ )$

The correctness proofs of our program slicing algorithm can refer to the way in [9, 22]. Informally, the term L and $\bigcup_{r \in Refs(\text{l.e})} lkpSli\ (r, getSli)$ in the definition of L' (see Equation (*)) can capture control dependences between statements and data dependences between variables, respectively.

3.3 The Complexity of the Algorithms

In this section, we will analyze the complexity of the monadic slicing algorithms presented in Section 3.2. The measures of system size used below are those associated with the data structure of program slice Slices (which is a Hash table).

In a monadic compiler/interpreter, our slice monad transformer could be modularly and safely combined into the semantic buildings, so the complexity analysis is restricted to L' and $Syn(\text{s}, L)$ of a concrete programming language. The intermediate label set L' can be determined in time $O(v)$, where v refers to the number of single variables in the program analyzed/executed. To determine the $Syn(\text{s}, L)$ shown in Figure 4 may cost $O(m)$, where m is the number of labeled expressions in the program. Therefore the time cost of the predominant part of program slicing in the monadic slicing algorithm is bounded by $O(v \times n)$, where n is the number of all labeled expressions appeared (perhaps repeatly) in the sequence of analyzing/executing the program. In addition, an extra time cost $O(v \times m)$ needs to get the executable slices of all single variables. Now we can see that the total time cost is $O(v \times n + v \times m)$. Since we finally obtain the slices of all variables after the last statement is analyzed/executed, the program slice of each variable, on the average, costs $O(n + m)$, which is linear.

To analyze the space complexity of the algorithms, we pay our attention to the constructions $Refs(\text{l.e})$, Slices, L' and L. We need space $O(v \times v)$ and $O(v \times m)$ to save $Refs(\text{l.e})$ and Slices, respectively. According to the definition of slice monad transformer SliceT in Figure 1, we need to introduce more intermediate labels when SliceT is applied to loop statements (eg. while statements), and can allocate a same space to the intermediate labels L' of other statements. So it takes the space $O(k \times m)$ to save intermediate labels, where k refers to the maximal times of analyzing/executing the

loop statements in the program. The label set L will cost the space $O(m)$. Therefore, the total space cost is $O(v \times v + v \times m + k \times m)$, which is irrelated to n.

(1) Source program

```
1     read n ;
2     read a ;
3     i := 1 ;
4     s := 1 ;
5     if (a > 0) then
6         s := 0
      else skip endif ;
7     while (i <= n) do
8       if (a > 0) then
9           s := s + 2
10      else s := s * 2 endif
11      i := (i + 1) ;
      endwhile ;
12    write (s + 4)
```

(2) Dynamic slice w.r.t s

```
1     read n ;
2     read a ;
3     i := 1 ;
4     s := 1 ;

7     while (i <= n) do
8       if (a > 0) then

10      else s := s * 2 endif
11      i := (i + 1) ;
      endwhile ;
12    write (s + 4)
```

Figure 8. A sample W-program and its slice w.r.t <(a=0, n=2), 20, s>

4. A Case Study

This section illustrates the dynamic slicing algorithm proposed in Figure 6 through the example W-program given in Figure 8 (1). We will use our modular monadic algorithm to compute the dynamic slice with respect to <(a=0, n=2), 12, s>.

We firstly label each expression in the example program with a unique label. Here we employ the line number of position where the expression occurs in source program. The resulting labeled expressions l.e and their corresponding set *Refs*(l.e) are showed in Figure 9.

Then, let INPUT is a=0 and n=2, and execute the program with INPUT according to the modular monadic semantics that includes the feature of program slicing. The following set is corresponding execution list, which comprises labels of instructions (i.e. statements or their snippets) that are the same order as they have been executed:

{1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 }

l. e		*Refs*(l.e)
3.	i := 1	{ i }
4.	s := 1	{ s }
5.	a > 0	{ a }
6.	s := 0	{ s }
7.	i <= n	{ i , n }
8.	a > 0	{ a }
9.	s :=s+2	{ s }
10.	s :=s*2	{ s }
11.	i :=i+1	{ i }
12.	s + 4	{ s }

Figure 9. l.e and *Refs*(l.e)

Lastly, as stated in monadic dynamic slice algorithm, the dynamic slices for associated variables of each instruction are computed immediately after this instruction is executed. In detail, while instructions in the above sequence list are being executed, the initial set L and dynamic slice table Slices are modified, according to Step 2 and 3 in our algorithm. In other words, after a labeled expression is executed, the set L is transformed into intermediate set L' through Equation (*), and this new set will be passed down the rest of the execution of the corresponding whole statement to this expression. Moreover, if this statement is an assignment one, the dynamic slice Slices need to update through the term *xtdSli* (ide, L', *getSli*).

For instance, at the beginning of executing for first time the 10th instruction in above list, the corresponding table Slices is showed in Figure 10 (1). At one time, the initial set L, passing through executions of the 7th and 8h instruction, is turned into an intermediate set L'' as follows:

$$L'' = \{8\} \cup L' \cup lkpSli\ (a, getSli) = \{3, 7, 8\}$$

where

$$L' = \{7\} \cup L \cup (lkpSli(i, getSli) \cup lkpSli(n, getSli)) = \{3, 7\}$$

After the first 10th instruction is executed, the intermediate set is changed to L''':

$$L''' = \{10\} \cup L'' \cup lkpSli\ (s, getSli) = \{10\} \cup \{3, 7, 8\} \cup \{4\} = \{3, 4, 7, 8, 10\}$$

Furthermore, the 10th is an assignment one, so the related data in Slices need to update through *xtdSli*. Here L''' replaces the dynamic slice of variable s before execution, and now the table Slices is given in Figure 10 (2). Going on in this way until finishing the execution of the last instruction (i.e. 12th) in above sequence list, we will obtain the final Slices as shown in Figure 10 (3). According to it and *Syn*(s, *L*), we could get the final result of dynamic slice, shown in Figure 8 (2), with respect to slice criterion <(a=0,n=2), 12, s>.

On the basis of Labra's language prototyping system LPS [23], which facilitates the modular development of interpreters from modular monadic semantics, we are now developing a monadic slicer [24]. As for this example, we can obtain the result (shown in Figure 11) from the current monadic slicer, where we didn't consider the function *Syn*(s, *L*) and directly include the Read and Write statements in the results.

(1) before execute first 10th

Var	Labels
n	ϕ
a	ϕ
i	{3}
s	{4}

(2) after execute first 10th

Var	Labels
n	ϕ
a	ϕ
i	{3}
s	{3, 4, 7, 8, 10}

(3) after execute the last instruction

Var	Labels
n	ϕ
a	ϕ
i	{3, 7, 11}
s	{3, 4, 7, 8, 10, 11}

Figure 10. Slices during execute instructions with INPUT

```
READ    1   ds  = [("n",[1])]
READ    2   ds  = [("n",[1]),("a",[2])]
ASSIGN  3   ds' = [("n",[1]),("a",[2]),("i",[3])]
ASSIGN  4   ds' = [("n",[1]),("a",[2]),("i",[3]),("s",[4])]
IF      5   ds  = [("n",[1]),("a",[2]),("i",[3]),("s",[4])]
WHILE   7   ds  = [("n",[1]),("a",[2]),("i",[3]),("s",[4])]
IF      8   ds  = [("n",[1]),("a",[2]),("i",[3]),("s",[4])]
ASSIGN  10  ds' = [("n",[1]),("a",[2]),("i",[3]),("s",[1,2,3,4,7,8,10])]
ASSIGN  11  ds' = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10])]
WHILE   7   ds  = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10])]
IF      8   ds  = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10])]
ASSIGN  10  ds' = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10,11])]
ASSIGN  11  ds' = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10,11])]
WHILE   7   ds  = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10,11])]
WRITE   12  ds  = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10,11,12])]
Value = Left 8
DSlices = [("n",[1]),("a",[2]),("i",[1,3,7,11]),("s",[1,2,3,4,7,8,10,11,12])]
```

Figure 11. The dynamic slicing result from our monadic slicer

5. Related Work

Most of the existing slicing algorithms rely on relation graphs such as system dependence graphs (SDG) or program dependence graphs (PDG). A few program slicing methods focused on the semantics of programs.

G.Canfora et al.'s *conditioned slicing* [25] adds a condition in a slicing criterion. Statements that do not satisfy the condition are deleted from the slice. M.Harman et al.'s *amorphous slicing* [26] allows for any simplifying transformations which preserve this semantic projection. These two methods are not directly based on formal semantics of a program. P.A.Hauser's *denotational slicing* [7, 8] employs the functional semantics of a program language in the denotational (and static) program slicer. Venkatesh also took account of denotational slicing with formal slicing algorithms including dynamic and static [9]. This approach is indeed based on the standard denotational

semantics of a program language. Inspired by this, we have proposed a new formal method for program slicing, called *modular monadic slicing*, which is based on modular monadic semantics.

Compared with the existing slicing methods, the modular monadic slicing has excellent flexibility and reusability properties, because it have abstracted the computation of program slicing as an language-independence object, *slice monad transformer*. The modular monadic slicing can compute slices directly on abstract syntax, without explicit construction of intermediate structures such as data flow graphs or dependence graphs in slicers. Despite this, it is still feasible, because there is a clear operational interpretation of modular monadic semantics, and some modular compilers/interpreters using monad transformers have already been constructed in [11, 12, 18, 20, 23, 27].

In respect of efficiency, our monadic algorithms are not less precise than PDG-based ones as shown from the example in Section 4. This is because the term L and $\bigcup_{r \in Refs(\mathrm{l.e})} lkpSli\ (r, getSli)$ in the definition of L' can accurately capture control dependences and data dependences respectively, which are the base of PDG-based algorithms. According to the complexity analysis in Sections 3.3, the space complexity is irrelated to the length of the program analyzed/executed; the time complexity of each variable is averagely linear, i.e. $O(n + m)$.

6. Summaries

Program slicing is an important decomposition technique. It can be roughly classified as static slicing and dynamic slicing, according to whether they only use statically available information or compute those statements that influence the value of a variable occurrence for a specific program input. It has been widely used in many software activities, such as software analyzing, understanding, debugging, testing, and maintenance.

In this paper, we have proposed a novel approach for program slicing; called it *modular monadic program slicing* as it is based on modular monadic semantics. It has excellent flexibility and reusability properties comparing with the existing program slicing algorithms. Furthermore, it is feasible, because modular monadic semantics is executable and some modular compilers/interpreters have already been existed. We now developed a program-slice prototype based on a modular monadic interpreter [23, 24].

In our future work, we will consider modular monadic slicing in the present of aliasing, pointer and array. At the same time, we will complete our prototype of monadic slicer and present the comparisons with other slicing methods in experiment.

Acknowledgements

The authors thank several anonymous referees for their constructive reviews and comments.

Reference

[1] F. Tip, “A Survey of Program Slicing Techniques”, *Journal of Programming Languages*, 1995, vol. 3, no. 3, pp. 121-189.

[2] D.Binkley, and K.B. Gallagher, “Program Slicing”, *Advances in Computers*, 1996, vol. 43, pp. 1-50.

[3] M. Kamkar, “An Overview and Comparative Classification of Program Slicing Techniques”, *Journal of Systems and Software*, 1995, vol. 31, no. 3, pp. 197-214.

[4] M. Harman, and R.M. Hierons, “An Overview of Program Slicing”, *Software Focus*, 2001, vol. 2, no. 3, pp. 85-92.

[5] M. Weiser, “Program Slicing”, *IEEE Transaction on Software Engineering*, 1984, vol. 16, no. 5, pp. 498-509.

[6] K.J. Ottenstein, and L.M. Ottenstein, “The program dependence graph in a software development environment”, *ACM SIGPLAN Notices*, 1984, vol.19, no. 5, pp. 177-184.

[7] P.A. Hausler, “Denotational Program Slicing”, *Proceeding of 22th Annual Hawaii International Conference on System Sciences*, 1989, vol. 2, pp. 486-495.

[8] L. Ouarbya, S. Danicic, M. Daoudi, M. Harman, and C. Fox, “A Denotational Interprocedural Program Slicer”, *Proceeding of 9th IEEE Working Conference on Reverse Engineering*, IEEE Press, Virginia, 2002, pp. 181-189.

[9] G. A. Venkatesh, “The Semantic Approach to Program Slicing”, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, 1991, pp. 26-28.

[10] E.Moggi, “Notions of Computation and Monads”, *Information and Computation*, 1991, vol. 93, pp. 55-92.

[11] S. Liang, and P. Hudak, “Modular Denotational Semantics for Compiler Construction”, *Proceeding of 6th European Symposium on Programming Languages and Systems, ESOP’96. LNCS 1058*, Springer-Verlag, Berlin, 1996, pp. 219-234.

[12] K. Wansbrough, “A Modular Monadic Action Semantics”, Master thesis, University of Auckland, Auckland, 1997.

[13] P.D. Mosses, “Semantics, Modularity, and Rewriting Logic”, *Proceeding of 2nd International Workshop on Rewriting Logic and its Applications, ENTCS 15*, Elsevier Press, Netherlands, 1998.

[14] J. Power, “Modularity in Denotational Semantics”, *Proceeding of 13th Annual Conference on Mathematical*

Foundations of Programming Semantics, Elsevier Press, New York, 2000.

[15] E. Moggi, "An Abstract View of Programming Languages", *LFCS Report, ECS-LFCS-90-113*, University of Edinburgh, 1989. http://www.lfcs.informatics.ed.ac.uk/reports/90/ECS-LFCS-90-113/.

[16] P. Wadler, "Comprehending monads", *ACM Conference on Lisp and Functional Programming*, ACM Press, France, 1990, pp. 61-78.

[17] D. Espinosa, "Semantic Lego", PhD dissertation, Columbia University, Columbia, 1995.

[18] S. Liang, P. Hudak, and M. Jones, "Monad Transformers and Modular Interpreters", *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, ACM Press, New York, 1995, pp. 333-343.

[19] J.E. Labra Gayo, M.C. Luengo Diez, J.M. Cueva Lovelle, and A. Cernuda del Rio, "Reusable Monadic Semantics of Object Oriented Programming Languages", *Proceeding of 6th Brazilian Symposium on Programming Languages, SBLP'02*, PUC-Rio University, Brazil, 2002.

[20] S. Liang, "Modular Monadic Semantics and Compilation", PhD dissertation, University of Yale, Yale, 1998.

[21] G.Tibor, B. Árpád, and F. István, "An Efficient Relevant Slicing Method for Debugging", *Software Engineering Notes, Software Engineering-ESEC/FSE'99 Springer ACM SIGSFT*, 1999, vol. 24, no. 6, pp. 303-321.

[22] G. A. Venkatesh, "Semantics of program slicing", *Bellcore TM-ARH-018561*, 1990.

[23] J.E. Labra Gayo, M.C. Luengo Diez, J.M. Cueva Lovelle, and A. Cernuda del Rio, "A Language Prototyping System Using Modular Monadic Semantics", *Workshop on Language Definitions, Tools and Applications, LDTA'01*, Netherlands, 2001.

[24] Website. https://sourceforge.net/projects/lps.

[25] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned Program Slicing", *Information and Software Technology*, 1998, vol. 40, no. 11/12, pp. 595-607.

[26] M. Harman, and S. Danicic, "Amorphous Program Slicing", *IEEE International Workshop on Program Comprehension, IWPC'97*, IEEE CS Press, Los Alamitos, 1997, pp. 70-79.

[27] W. Kahl, "A Modular Interpreter Built with Monad Transformers", *Lectures on Functional Programming*, CAS 781, 2003. http://www.cas.mcmaster.ca/~kahl/FP/2003/