2

Program Slicing Based on Monadic Semantics

Yingzhou Zhang[1,2,3]
[1]*College of Computer, Nanjing University of Posts and Telecomm., Nanjing*
[2]*Jiangsu High Technology Research Key Lab. for Wireless Sensor Networks,*
[3]*Key Lab of Broadband Wireless Communication and Sensor Network Technology,*
Ministry of Education Jiangsu Province, Nanjing,
China

1. Introduction

A program slice consists of those statements of a program that may directly or indirectly affect the variables computed at a given program point (Weiser, 1984). The program point p and the variable set V, denoted by <p, V>, is called a slicing criterion. Program slicing has applications in software testing and debugging, measurement, re-engineering, program comprehension and so on (Kamkar, 1995; Tip, 1995; Harman, 2001; Binkley, 1996; Gallagher, 1991).

Program slicing algorithms can be roughly classified as static slicing and dynamic slicing methods, 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. Most of the existing slicing algorithms rely on relation graphs such as system dependence graphs (SDG) or program dependence graphs (PDG). These slicing methods are incremental, sequential, not combinatorial or not parallelizable easily for multi-core systems. However modern programming languages support modularized programming and programs might consist of a set of modules. So the program analysis should reflect this design technology, and their methods (including program slicing) should be flexible, combinable, and parallelizable for improving the efficiency.

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. On the basis of this view, this paper proposes an 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.

The program slicing methods focused on the semantics of programs can be found in ref. (Hausler, 1989; Ouarbya, 2002; Venkatesh, 1991). These methods are based on the standard denotational semantics of a program language. As mentioned in ref. (Moggi, 1991; Liang & Hudak, 1996; Wansbrough, 1997; Mosses, 1998; Zhang & Xu, 2004), traditional denotational semantics lack modularity and reusability. A better solution was to use *monads* (Moggi, 1991) to structure denotational semantics, with the help of *monad transformers* (Moggi, 1991; Wadler, 1992; Espinosa, 1995) 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; called it *modular monadic semantics*. In this paper, we will employ it in our program slicing algorithm.

In our previous work (Zhang et al, 2004, 2005, 2006; Zhang & Xu, 2005; Wu et al. 2006), we abstracted the computation of program slicing as a simple slice monad transformer, which took only the label set *L* as its parameter, without reflecting explicitly the change of the slice table Slices. In ref. (Zhang, 2007), we presented theoretical foundation for our previous work. Based on these theories of the monadic slicing and from the view of the practical implementation, this paper will redesign the static slice monad transformer. The extensibility and reusability of our monadic method will be showed by easily introducing a new program feature (such as pointers) to slicing analysis.

The rest of the paper is organized as follows: In Section 2, we briefly introduce and illustrate the concepts of modular monadic semantics through a simple example language. The computation of program slicing is abstracted as *slice monad transformer* in Section 3. In Section 4, we discuss and illustrate our static slicing algorithm in detail. In Section 5, we show how our slicing algorithm can be readily adapted to an extension for the example language with pointers. In Section 6 and 7, we address the implementation, the time and space complexity analysis. We conclude this paper with directions for future work in Section 8.

Along the paper, the presentation follows the monadic semantics style. We use Haskell[1] notation with some freedom in the use of mathematical symbols and declarations. For brevity and convenience, we will omit the type constructors in some definitions.

2. Modular monadic semantics

In this section, we briefly review the theory of monads (Wadler & Thiemann, 2003; Moggi, 1989), monad transformers and modular monadic semantics (Liang, 1998). Readers familiar with these topics may skip the section, except for the last three paragraphs (about the syntax and monadic semantics of an example language).

2.1 Monads and monad transformers

Monads, originally coming from philosophy, were discovered in category theory in the 1950s and introduced to the semantics community by Moggi in 1990s (Moggi, 1989). After this work, Wadler popularized Moggi's ideas in the functional programming community (esp. in Haskell) (Wadler, 1995). In the monad-based view of computation, a monad is a way to structure computations in terms of values and sequences of computations using those values (Newbern, 2002). The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required. From this view, a monad can be thought as a strategy for combining computations into more complex computations.

In Haskell, monads are implemented as a type constructor class with two member operations/functions.

[1] Haskell is an advanced purely functional programming language. Please visit its official website (http://www.haskell.org or http://haskell.org) for more information.

$$\begin{array}{l}\textbf{class}\ \text{Monad}\ m\ \textbf{where}\\ \quad return :: a \to m\ a\\ \quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

Here, *return* is the Haskell name for the unit and >>= (pronounced "bind") is the extension operation of the monad. The above definition of the monad class means: a parameterized type *m* (which may think of as a function from types to types) is a monad if it supports the two operations *return* and >>= with the types given. Using the combinator analogy, a monad *m* is a combinator that can apply to different values. *m a* is a combinator applying to a value of type *a*. The *return* operation puts a value into a monadic combinator. The >>= operation takes the value from a monadic combinator and passes it to a function to produce a monadic combinator containing a new value, possibly of a different type. The >>= operation is known as "bind" because it binds the value in a monadic combinator to the first argument of an operation.

To be a proper monadic combinators, the *return* and >>= operations must work together according to some simple laws. Monads laws state in essence that >>= operation (sequential composition) is associative, and *return* is its unit/identity. Failure to satisfy these laws will result in monads that do not behave properly and may cause subtle problems when using the do-notation[2].

A monad (call it *m*) therefore defines a type of computation. The nature of the computation is captured by the choice of the type *m*. The *return* operation constructs a trivial computation that just renders its argument as its result. The >>= operation combines two computations together to make more complex computations of that type.

To make the use of monads more convenient, we adopt the following syntactic sugar (which is similar to S. Liang's notation and the do-notation in Haskell as well):

$$\begin{array}{ll}\{e\} & \equiv e\\ \{m;\ e\}_m & \equiv m \gg\!= \backslash\ \_ \to \{e\}\\ \{x \leftarrow m;\ e\}_m & \equiv m \gg\!= \backslash\ x \to \{e\}\\ \{\textbf{let}\ exp;\ e\} & \equiv \textbf{let}\ exp\ \textbf{in}\ \{e\}\end{array}$$

In practice, the computations can't be performed in isolation. In this case, we need a monad that combines the features of the two monads into a single computation. It is impossible in general to combine two monads to form a new monad. Moreover, it is inefficient and poor practice to write a new monad instance with the required characteristics each time a new combination is desired. Instead, there is the technique, called monad transformers (Liang, 1998), which can transform a given monad into a new one that has both the new operations and maintains those of the former monad. The concept of monad transformers was rediscovered by D.Espinosa in Moggi's original work. He developed a system, Semantic Lego, which implemented Moggi's original *monad constructors* to give a modular semantics for languages.

[2] Do notation is an expressive shorthand for building up monadic computations. In short, the do notation allows us to write monadic computations using a pseudo-imperative style with named variables. The result of a monadic computation can be "assigned" to a variable using a left arrow ← operator.

In Haskell, a monad transformer can be defined as any type constructor *t* such that if *m* is a monad, so is "*t m*", by using the two parameter constructor class MonadTrans:

```
class (Monad m, Monad t m) ⇒ MonadTrans t m where
    lift :: m a → t m a
```

The member function *lift* lifts a monadic computation in the inner monad *m* into the combined monad "*t m*". Furthermore, we expect a monad transformer to add features, without changing the nature of an existing computation. This can be obtained by the properties of *lift* function above (also called *monad transformer laws*). The monad transformer laws guarantee the basic lifting property that any program, which does not use the added features, should behave in the same way after a monad transformer is applied. Intuitively, these laws say that lifting a null computation brings about a null computation, and that lifting a sequence of computations is equivalent to first lifting them individually, and then combining them in the lifted monad.

For example, Figure 1 gives the environment monad transformer, EnvT, which can be used to add environment reading functionality to other monads. In Figure 1, the functions *rdEnv* and *inEnv*, return the current environment and perform a computation in a given environment, respecitively.

```
newtype EnvT r m a = EnvT {runEnvT :: r → m a}
instance (Monad m) ⇒ Monad (EnvT r m) where
   return a = EnvT (\_→ return a)
   m ≫= k  = EnvT (\r→ {a ← runEnvT m r; runEnvT (k a) r} )
instance MonadTrans (EnvT r) where
       lift m = EnvT (\_→ m)
class (Monad m) ⇒ EnvMonad r m where
       inEnv :: r → m a → m a
       rdEnv :: m r
instance (Monad m) ⇒ EnvMonad r (EnvT r m) where
       inEnv r m = EnvT (\_→ runEnvT m r)
       rdEnv = EnvT (\r→ return r)
```

Fig. 1. Environment monad transformer EnvT(Liang,1998).

2.2 Modular monadic semantics

Modular monadic semantics specifies the semantics of a programming language by mapping terms to computations, where the details of the environment, store, etc. are hidden within a monad. This is difference from traditional denotational semantics, which maps a term (an environment or a continuation) to an answer. The modular monadic semantics is composed of two parts: *modular semantic building blocks* and *monad transformers*. Semantic building blocks define the monadic semantics of individual source language features. They are independent of each other.

Monad transformers define the kernel-level operations in a modular way. Multiple monad transformers can be composed to form the underlying monad used by all the semantic building blocks. The crucial property of modular monadic semantics is the division of the monad *m* into a series of monad transformers, each representing a computation. As mentioned in the previous section, monad transformers provide the power to represent the abstract notion of programming language features, but still allow us to access low-level semantic details. The concept of lifting allows us to consider the interactions between various features. In some sense, *monad transformers can be designed once and for all* (Liang, 1998), since they are entirely independent of the language being described. From this view, we can draw the computation of program slicing as an entity that is independent of the language being analyzed. This will be discussed in the next section. Before doing it, we illustrate the modular monadic semantic description of a very simple imperative programming language **W**.

The **W** language considered in this paper is very similar to the language described in ref. (Slonneger & Kurtz, 1995).The abstract syntax and semantic building blocks of the **W** language are provided in Figure 2 and 3 respectively. In Figure 3, the identifier ***Fix*** denotes a fixpoint operator; *xtdEnv* and *lkpEnv* are the updating and lookup operators of environments Env, respectively; *updSto* and *alloc* are the updating and allocation functions of stores Loc, respectively; *rdEnv* and *inEnv* are the basic operators of the enviornment monad EnvMonad (given in Figure 1); *putValue* and *getValue* are the writting and reading functions of I/O actions, respectively. Following Venkatesh's assumption for expressions in ref. (Venkatesh, 1990), we also assume that the labeled expressions have no side-effects. The expressions, whose syntax is left unspecified for the sake of generality, consist of operations over identifiers and are uniquely labeled. The label is for the entire expression.

Domains:

arg: Arg (Arguments); b: Blk (Blocks); c: Cmd (Commands);

d: Dec (Declarations); e: Exp (Expressions); ide: Ide (Identifiers);

l: Label (Labels); p: Prg (Programs); t: Type (Types)

Abstract Syntax:

p :: = **program** ide **is** b

b :: = d **begin** c **end**

d :: = **const** ide = l.e | **var** ide : t | d_1; d_2

c :: = ide := l.e | c_1; c_2 | skip | **read** ide | **write** l.e

| **while** l.e **do** c **endwhile** | **if** l.e **then** c_1 **else** c_2 **endif**

Fig. 2. Abstract syntax of the **W** language.

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, say StateT and EnvT, and apply them to the base monad IO, forming the combined monad ComptM:

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

The environment monad transformer adds an environment to the given monad. The *return* function ignores the environment, while $\gg\!=$ passes the inherited environment to both sub-computations. Equipped with the monad transformers, the resulting monad ComptM can support all of the semantic building blocks in Figure 3, which gives the formal semantic description we expected.

Domains:

r: Env (Environments); loc: Loc (Stores); s: State (States); v: Value (Values)

Semantics Functions:

$$\boldsymbol{M} :: \text{Prg} \to \text{ComptM}\ ()$$

$$\boldsymbol{M}\llbracket\, \textbf{program}\ \text{ide}\ \textbf{is}\ \text{b} \,\rrbracket = \boldsymbol{B}\llbracket\, \text{b} \,\rrbracket$$

$$\boldsymbol{B} :: \text{Blk} \to \text{ComptM}\ ()$$

$$\boldsymbol{B}\llbracket\, \text{d}\ \textbf{begin}\ \text{c}\ \textbf{end} \,\rrbracket = \{r' \leftarrow \boldsymbol{D}\llbracket\, \text{d} \,\rrbracket;\ inEnv\ r'\ \boldsymbol{C}\llbracket\, \text{c} \,\rrbracket\}$$

$$\boldsymbol{D} :: \text{Dec} \to \text{ComptM Env}\ ;\quad \boldsymbol{E} :: \text{Exp} \to \text{ComptM Value}$$

$$\boldsymbol{D}\llbracket\, \textbf{const}\ \text{ide} = \text{l.e} \,\rrbracket = \{v \leftarrow \boldsymbol{E}\llbracket\, \text{l.e} \,\rrbracket;\ r \leftarrow rdEnv;\ xtdEnv\ (\text{ide}, v, r)\ \}$$

$$\boldsymbol{D}\llbracket\, \textbf{var}\ \text{ide} : \text{t} \,\rrbracket = \{loc \leftarrow alloc;\ r \leftarrow rdEnv;\ xtdEnv\ (\text{ide}, loc, r)\}$$

$$\boldsymbol{D}\llbracket\, \text{d}_1;\ \text{d}_2 \,\rrbracket = \{r \leftarrow rdEnv;\ r' \leftarrow inEnv\ r\ \boldsymbol{D}\llbracket\, \text{d}_1 \,\rrbracket;\ inEnv\ r'\ \boldsymbol{D}\llbracket\, \text{d}_2 \,\rrbracket)\}$$

$$\boldsymbol{C} :: \text{Cmd} \to \text{ComptM}\ ()$$

$$\begin{aligned}
&\boldsymbol{C}\llbracket\, \text{ide} := \text{l.e} \,\rrbracket = \{v \leftarrow \boldsymbol{E}\llbracket\, \text{l.e} \,\rrbracket;\ r \leftarrow rdEnv;\ loc \leftarrow lkpEnv(\text{ide}, r);\ updSto(loc, v)\}\\
&\boldsymbol{C}\llbracket\, \text{c}_1;\ \text{c}_2 \,\rrbracket = \{\boldsymbol{C}\llbracket\, \text{c}_1 \,\rrbracket;\ \boldsymbol{C}\llbracket\, \text{c}_2 \,\rrbracket\}\\
&\boldsymbol{C}\llbracket\, \text{skip} \,\rrbracket = return\ ()\\
&\boldsymbol{C}\llbracket\, \textbf{if}\ \text{l.e}\ \textbf{then}\ \text{c}_1\ \textbf{else}\ \text{c}_2\ \textbf{endif} \,\rrbracket = \{v \leftarrow \boldsymbol{E}\llbracket\, \text{l.e} \,\rrbracket;\ \boldsymbol{case}\ v\ \boldsymbol{of}\ \textbf{TRUE} \to \boldsymbol{C}\llbracket\, \text{c}_1 \,\rrbracket\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{FALSE} \to \boldsymbol{C}\llbracket\, \text{c}_2\}\\
&\boldsymbol{C}\llbracket\, \textbf{while}\ \text{l.e}\ \textbf{do}\ \text{c}\ \textbf{endwhile} \,\rrbracket = \boldsymbol{Fix}\ (\backslash f \to \{v \leftarrow \boldsymbol{E}\llbracket\, \text{l.e} \,\rrbracket;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boldsymbol{case}\ v\ \boldsymbol{of}\ \textbf{TRUE} \to f \cdot \boldsymbol{C}\llbracket\, \text{c} \,\rrbracket\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{FALSE} \to return\ ()\ \}\ )\\
&\boldsymbol{C}\llbracket\, \textbf{read}\ \text{ide} \,\rrbracket = \{loc \leftarrow lkpEnv(\text{ide}, rdEnv);\ v \leftarrow getValue;\ updSto(loc, return\ v)\}\\
&\boldsymbol{C}\llbracket\, \textbf{write}\ \text{l.e} \,\rrbracket = \{\ v \leftarrow \boldsymbol{E}\llbracket\, \text{l.e} \,\rrbracket;\ putValue\ v\ \}
\end{aligned}$$

Fig. 3. Semantic building blocks of the **W** language.

3. Static slice monad transformer

As mentioned above, each monad transformer represents a single notion of computation. Since static program slicing can be viewed as a computation, we can abstract it as a language-independent notion of a computation by using a *static slice-monad transformer*

SliceT. Its definition is given in Figure 4, where *l* denotes a set of labels of expressions that were required to compute the current statement; *st* denotes a slice table Slices whose data structure is defined as follows:

```
                 type Var = String
                 type Labels = [Int]
                 type Slices = [(Var, Labels)]
lkpSli :: Var → Slices → ComptM Labels
updSli :: (Var, ComptM Labels) →Slices →ComptM ()
mrgSli :: Slices → Slices → ComptM Slices
```

Here, [] denotes a table data structure. The three operators, *lkpSli*, *updSli* and *mrgSli*, represent to lookup the slice of a variable in a given Slices table, to update a table Slices through a variable with its slice, and to merge two given slice tables into a new one, respectively.

In the similar way as ref. (Zhang, 2005, 2007), the following theorems are straightforward. These theorems guarantee the correctness of the definition of the slice monad transformer in Figure 4.

Theorem 1. *SliceT l st m* is a monad.
Theorem 2. *SliceT l st* is a monad transformer.

A static slice-monad transformer *SliceT l st*, taking an initial set of labels and a slice table, returns a computation of a pair of the resulting value and the new slice table. The operator *return* returns the given value with the unchanged slice table. The operator >>= takes a monad *m* and a function *k*. It passes the initial set of labels *l* and slice table *st* to the monad *m*; this yields a value *a* paired with an intermediate slice table *st*′; function *k* is applied to the value *a*, yielding a monad (*k a*); this yields in *l* and *st*′ the final result paired with the final slice table.

The lifting function *lift* says that a computation in the monad *m* behaves identically in the monad *SliceT l st m* and makes no changes to the slice table. The operation *rdLabel*/*inLabel* and *getSli*/*setSli* support reading/setting of the parameter *l* and *st* in the static slice-monad SliceMonad, respectively.

With the use of the transformer *SliceT*, other monads can be easily transformed into the static slice-monad SliceMonad. For instance, we can lift respectively the basic operators of monads StateMonad and EnvMonad through *SliceT* as shown in Figure 4.

4. A monadic static slicing algorithm

The static slice for a variable in a program is the collection of all possible computations of values of that variable. In this section, we only consider end slicing for a single variable, i.e. the slicing criterion is <p, v>, where v the variable of interest, and p the end program point. 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 (Binkley, 1996).

The main idea of monadic static slicing algorithms can be briefly stated as follows: for obtaining a static slice, we firstly apply the slice transformer SliceT to semantic building blocks of the program analyzed. It makes the resulting semantic description include

```
newtype SliceT l st m a = SliceT {runSliceT :: (l, st) → m (a, st)}
instance (Monad m) ⇒ Monad (SliceT l st m) where
   return a = SliceT (\(_, st)→ return (a, st) )
       m ≫= k  = SliceT (\(l, st)→ {(a, st′) ← runSliceT m (l, st);
                                          runSliceT (k a) (l, st′)} )
instance MonadTrans (SliceT l st) where
        lift m = SliceT (\(_, st)→ {a ←  m; return (a, st)} )
class (Monad m) ⇒ SliceMonad l st m where
      rdLabel :: m l
      inLabel :: l → m a → m a
      getSli :: m st
      setSli :: st → m st
instance (Monad m) ⇒ SliceMonad l st (SliceT l st m) where
      rdLabel    = SliceT return
      inLabel l m  = SliceT (\(_, st)→ runSliceT m (l, st) )
      getSli     = SliceT (\(_, st)→ return (st, st) )
      setSli  st  = SliceT (\(l, _)→ return (l, st) )
instance (EnvMonad r m, MonadTrans (SliceT l st) m)
   ⇒ EnvMonad r (SliceT l st m) where
       inEnv r (SliceT l st m) = SliceT (\(l,st)→ inEnv r (runSliceT m (l,st)) )
       rdEnv = lift rdEnv
instance (StateMonad s m, MonadTrans (SliceT l st) m)
   ⇒ StateMonad s (SliceT l st m) where
       update = lift . update
```

Fig. 4. Static slice-monad transformer SliceT.

program slice semantic feature. According to the semantic description, we then compute static slices of each statement in sequence. Finally we will obtain the static slices of all single variables in the program. In fact, with the process of analyzing a program, the Slices table, which includes the current individual program slices for all variables of this program, is modified steadily according to the monadic slicing algorithm.

Concretely, with respect to the example program **W** mentioned in Section 2, Figure 5 gives the main part of our monadic static algorithm. Figure 5 gives the rules of when and how to modify the current slice table. It adds the computation of static slicing into program analysis modularly, with the help of the monad transformers SliceT given in Section 3. SliceT can be composed with other transformers such as EnvT and StateT as follows, and apply them to monad IO, forming the underlying monad ComptM:

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

In Figure 5 (or the monadic slicing algorithm in a sense), for each computation of a labeled expression l.e, there is an intermediate set L' as follows:

$$L' = \{\text{l}\} \cup L \cup \bigcup_{x \in Refs(\text{l.e})} lkpSli(x,T)$$

where T represents the current slice table before analyzing the expression l.e; *Refs*(l.e) denotes the set of variables appeared in l.e. The relation above means that after an expression l.e included in the current expression is computed, the initial set L should be transformered by adding the label of expression l.e, i.e., label l, and all sets of labels of expressions influenced by the variables of the expression l.e.

In addition, the *updSli* operation of the Slices type is applied to record the result of the static slicing in program analysis. In case of the language **W**, only when describe the semantics of assignment statement and initial assignment statement within a variable declaration, the corresponding operator *updSli* should be added in as shown in Figure 5.

A static slice includes the statements that possibly affect the variable in the slicing criterion. Therefore, for capturing these possible statements, in Figure 5 we ought to add the operator *mrgSli* into semantic descriptions of conditional statement and loop statement.

After the last statement of a program is analyzed, we could obtain, from the result Slices table, the static slice of each single variable (say *var*) of the program, which is the set of labels of all expressions influenced on the *var* variable:

$$L = lkpSli\ (var, getSli)$$

For getting the final result of the static slices, i.e., a syntactically valid subprogram, we -- following Venkatesh (Venkatesh, 1990) -- define *Syn*(s, L) for language **W** in Figure 6, where s is a W-program analyzed. It guides us how to construct a syntactically valid subprogram of s from the set L. It just gives a strategy for returning the final result/subprogram from a given set L, so it could be changed to cater to different people's need. For example, if one does not consider variable declaration as part of slices, then one might change the corresponding term in Figure 6 as follows:

"**var** ide : t " : "skip"

The correctness proofs of our monadic static slicing algorithms can refer to their termination theorem and their consistency with PDG-based slicing algorithms, given in ref. (Zhang, 2007). In fact, the term L and $\bigcup_{x \in Refs(\text{l.e})} lkpSli(x,T)$ in the above definition of L' can accurately capture control dependences and data dependences related, respectively.

For more about the algorithm, we now illustrate to use the rules in Figure 5 to compute the static slice w.r.t. <8, sum> of an example **W** program in Figure 7. Its each expression is uniquely labeled through the label (marked in source program) of the place where the expression presences. So the fourth expression is "i := 1".

According to the rule/semantics of assignment statements in Figure 5, after the third expression (i.e. "sum := 0") is analyzed, its intermediate set L (whose initial value is ϕ) is changed to L':

$$L' = \{3\} \cup L \cup lkpSli\ (\text{sum}, T) = \{3\} \cup \phi \cup \phi = \{3\}$$

Where T is the current slice table, including the static slices of the "i" and "sum" variables, written briefly as L(i) and L(sum), respectively. Since this expression is an assignment one, the related data in Slices need to update through *updSli*, i.e. L(sum) = L' = {3}. Similarly, after the 4th expression is analyzed, we have

$$L' = \{4\} \cup L \cup lkpSli\ (\text{i}, T) = \{4\};\ \ \text{L(i)} = \{4\}$$

And keep going for the 5th-7th expressions, we obtain

$$\text{5th: } L' = \{5\} \cup \phi \cup \mathrm{L(i)} = \{4, 5\}$$

$$\text{6th: } L'' = \{6\} \cup L' \cup (\mathrm{L(sum)} \cup \mathrm{L(i)}) = \{3, 4, 5, 6\};\ \mathrm{L(sum)} = \{3, 4, 5, 6\}$$

$$\text{7th: } L'' = \{7\} \cup L' \cup \mathrm{L(i)} = \{4, 5, 7\};\ \mathrm{L(i)} = \{4, 5, 7\}$$

$\boldsymbol{M} :: \mathrm{Prg} \rightarrow \mathrm{ComptM}\ ()$

$\boldsymbol{M}[\![\ \textbf{program}\ \mathrm{ide}\ \textbf{is}\ \mathrm{b}\ ]\!] = \mathit{return}\ ()$

$\boldsymbol{B} :: \mathrm{Blk} \rightarrow \mathrm{ComptM}\ ()$

$\boldsymbol{B}[\![\ \mathrm{d}\ \textbf{begin}\ \mathrm{c}\ \textbf{end}\ ]\!] = \mathit{return}\ ()$

$\boldsymbol{D} :: \mathrm{Dec} \rightarrow \mathrm{ComptM\ Env} \quad \boldsymbol{E} :: \mathrm{Exp} \rightarrow \mathrm{ComptM\ Value}$

$\boldsymbol{D}[\![\ \textbf{const}\ \mathrm{ide} = \mathrm{l.e}\ ]\!] = \{\ L \leftarrow \mathit{rdLabels};\ T \leftarrow \mathit{getSli};$

$\qquad L' \leftarrow \{\mathrm{l}\} \cup L \cup \bigcup_{x \in \mathit{Refs}(\mathrm{l.e})} \mathit{lkpSli}(x, T)\ ;\ \mathit{updSli}(\mathrm{ide}, L', T\,)\ \}$

$\boldsymbol{D}[\![\ \textbf{var}\ \mathrm{ide} : \mathrm{t}\ ]\!] = \mathit{return}\ ()$

$\boldsymbol{D}[\![\ \mathrm{d}_1;\ \mathrm{d}_2\ ]\!] = \mathit{return}\ ()$

$\boldsymbol{C} :: \mathrm{Cmd} \rightarrow \mathrm{ComptM}\ ()$

$\boldsymbol{C}[\![\ \mathrm{ide} := \mathrm{l.e}\ ]\!] = \{\ L \leftarrow \mathit{rdLabels};\ T \leftarrow \mathit{getSli};$

$\qquad L' \leftarrow \{\mathrm{l}\} \cup L \cup \bigcup_{x \in \mathit{Refs}(\mathrm{l.e})} \mathit{lkpSli}(x, T)\ ;\ \mathit{updSli}(\mathrm{ide}, L', T\,)\ \}$

$\boldsymbol{C}[\![\ \mathrm{c}_1;\ \mathrm{c}_2\ ]\!] = \{\boldsymbol{C}[\![\ \mathrm{c}_1\ ]\!]\ ;\ \boldsymbol{C}[\![\ \mathrm{c}_2\ ]\!]\ \}$

$\boldsymbol{C}[\![\ \mathrm{skip}\ ]\!] = \mathit{return}\ ()$

$\boldsymbol{C}[\![\ \textbf{if}\ \mathrm{l.e}\ \textbf{then}\ \mathrm{c}_1\ \textbf{else}\ \mathrm{c}_2\ \textbf{endif}\ ]\!] = \{\ L \leftarrow \mathit{rdLabels};\ T \leftarrow \mathit{getSli};$

$\qquad L' \leftarrow \{\mathrm{l}\} \cup L \cup \bigcup_{x \in \mathit{Refs}(\mathrm{l.e})} \mathit{lkpSli}(x, T)\ ;$

$\qquad \mathit{inLabels}\ L'\ \boldsymbol{C}[\![\ c_1\ ]\!]\ ;\ T1 \leftarrow \mathit{getSli};\ \mathit{setSli}(T);$

$\qquad \mathit{inLabels}\ L'\ \boldsymbol{C}[\![\ c_2\ ]\!]\ ;\ T2 \leftarrow \mathit{getSli};\ \mathit{mrgSli}(T1, T2)\ \}$

$\boldsymbol{C}[\![\ \textbf{while}\ \mathrm{l.e}\ \textbf{do}\ \mathrm{c}\ \textbf{endwhile}\ ]\!] = \boldsymbol{\mathit{Fix}}\ (\lambda f.\ \{L \leftarrow \mathit{rdLabels};\ T \leftarrow \mathit{getSli};$

$\qquad L' \leftarrow \{\mathrm{l}\} \cup L \cup \bigcup_{x \in \mathit{Refs}(\mathrm{l.e})} \mathit{lkpSli}(x, T)\ ;$

$\qquad f \cdot \{\mathit{inLabels}\ L'\ \boldsymbol{C}[\![\ c\ ]\!]\ ;\ T' \leftarrow \mathit{getSli};\ \mathit{mrgSli}(T, T')\}\})$

$\boldsymbol{C}[\![\ \textbf{read}\ \mathrm{l.ide}\ ]\!] = \{L \leftarrow \mathit{rdLabels};\ T \leftarrow \mathit{getSli};\ L' \leftarrow \{\mathrm{l}\} \cup L;\ \mathit{updSli}(\mathrm{ide}, L', T)\}$

$\boldsymbol{C}[\![\ \textbf{write}\ \mathrm{l.e}\ ]\!] = \mathit{return}\ ()$

Fig. 5. Monadic semantics of static slicing for **W** programs.

```
     program Example is
1    var sum
2    var i
     begin
3     sum := 0 ;
4     i := 1;
5     while i < 11 do
6        sum := sum + i;
7        i := i + 1
           endwhile;
8    write sum
           end
```

Fig. 7. An example program.

Because of the loop of While statement, the 5th-7th expressions are analyzed again:

$$\text{5th: } L'' = \{5\} \cup L' \cup \mathrm{L(i)} = \{4, 5, 7\}$$

$$\text{6th: } L''' = \{6\} \cup L'' \cup (\mathrm{L(sum)} \cup \mathrm{L(i)}) = \{3, 4, 5, 6, 7\};$$

$$\mathrm{L(sum)} = \{3, 4, 5, 6, 7\}$$

$$\text{7th: } L''' = \{7\} \cup L'' \cup \mathrm{L(i)} = \{4, 5, 7\};\ \mathrm{L(i)} = \{4, 5, 7\}$$

Now, if the whole loop body is analyzed over again in this way, the slice L(sum) and L(i) will not be changed again, reaching their fixpoint. So after finishing the analysis of the last statement (8h), we obtain the final table Slices as follows:

$$\mathrm{L(i)} = \{4, 5, 7\};\ \ \mathrm{L(sum)} = \{3, 4, 5, 6, 7\}$$

According the rules of *Syn*(s, *L*) in Figure 6, we can obtain the final result of static slice w.r.t. <8, sum> of the example program.

5. Extending W language with pointers

The modular monadic approach mentioned previously is flexible enough that we can easily introduce a new program feature to analysis. In this section, we will illustrate this power by considering an extension of the language **W** with pointers. We shall show how to adapt the implementation to this extension, with a small change in our existing monadic slice algorithm.

The introduction of a pointer will lead to aliasing problems (Horwitz, 1989; Hind, 1999) (i.e. multiple variables access the same memory location), so we need pointer analysis to obtain the corresponding data dependency information. In order to represent the unbounded data

Syn(s, *L*) = case s of

"**program** ide **is** b " :	***if*** *Syn*(b, *L*) = "skip" ***then*** "skip" ***else*** "**program** ide **is** b"
"d **begin** c **end** " :	***if*** *Syn*(d, *L*)=*Syn*(c, *L*)="skip" ***then*** "skip" ***else*** "d **begin** c **end**"
"**const** ide = l.e " :	***if*** $l \in L$ ***then*** "**const** ide = l.e" ***else*** "skip"
"**var** ide : t " :	***if*** ide $\in \{var\} \cup \bigcup_{l \in L} Refs(l.e)$ ***then*** "**var** ide : t " ***else*** "skip"
"d_1; d_2 " :	*Syn*(d_1, *L*); *Syn*(d_2, *L*)
"ide := l.e " :	***if*** $l \in L$ ***then*** "ide := l.e" ***else*** "skip"
"c_1; c_2 " :	*Syn*(c_1, *L*); *Syn*(c_2, *L*)
"skip " :	"skip"
"**read** ide " :	***if*** ide $\in \{var\} \cup \bigcup_{l \in L} Refs(l.e)$ ***then*** "**read** ide" ***else*** "skip"
"**write** l.e " :	"skip"
"**if** l.e **then** c_1 **else** c_2 **endif** ":	***if*** (*Syn*(c_1, *L*) = *Syn*(c_2, *L*) = "skip") $\wedge$ ($l \notin L$) ***then*** "skip"
	else "**if** l.e **then** *Syn*(c_1, *L*) **else** *Syn*(c_2, *L*) **endif** "
"**while** l.e **do** c **endwhile** " :	***if*** (*Syn*(c, *L*) = "skip") $\wedge$ ($l \notin L$) ***then*** "skip"
	else "**while** l.e **do** *Syn*(c, *L*) **endwhile**"

Fig. 6. The definition of *Syn*(s, *L*), where *L* = *lkpSli*(*var*, *getDSli*).

structures in a finite way for the presence of pointers, we consider that all point in the same procedure applied to form an array of heap space, and will deal with this array as a whole. The extended algorithm combines the point-to analysis by data-flow iteration with forward monad slicing. With the illumination of this idea, the key issue to be addressed is the solution to assignments. The other statements (such as conditional statements, loop statements, etc.) could be resolved by adding point-to computation to the existing slicing methods. Before going on, we introduce a data structure for point-to analysis.

Similar to the Slices datatype in Section 3, we design an abstract datatype PT for point-to analysis:

type Var = String
type PtSet = [Var]
type PT = [(Var, PtSet)]
    getPT :: ComptM PT
    setPT :: PT $\rightarrow$ ComptM PT
    lkpPT :: Var $\rightarrow$ PT $\rightarrow$ ComptM PtSet

$$updPT :: \text{Var} \rightarrow \text{PtSet} \rightarrow \text{PT} \rightarrow \text{ComptM} \; ()$$
$$mrgPT :: \text{PT} \rightarrow \text{PT} \rightarrow \text{ComptM PT}$$

The point-to table, PT, is a table of pairs of a single variable and its associated point-to set (a set of variables). It has five operators *getPT*, *setPT*, *lkpPT*, *updPT* and *mrgPT*, which return and set the current table of point-to sets, lookup a point-to set corresponding to a variable in a given table of point-to sets, update some point-to sets corresponding to a list of variables in a given table of point-to sets, and merge two table of point-to sets into one table, respectively.

With the pointers, we sometimes need to update the slices or the point-to sets of some variables at the same time, so we extend the operator *xtdSli* for Slices datatype, and the operator *xtdPT* for PT datatype as follows:

$$xtdSli :: [\text{Var}] \rightarrow \text{Labels} \rightarrow \text{Slices} \rightarrow \text{ComptM} \; ()$$

$$xtdPT :: [\text{Var}] \rightarrow \text{PtSet} \rightarrow \text{PT} \rightarrow \text{ComptM} \; ()$$

Now we can study in depth the assignment statements with pointers. For simplicity, we only consider single dereference of a pointer (e.g. *x), since multi-dereference (e.g. **x) can be divided into multiple single dereferences. We decompose an assignment into a left-value expression (such as x or *x), and a right-value expression (also notated as l.e, but may contain *y or &y). So we need to expand the definition of *Refs*(l.e) in Section 4. The variables appeared in a right-value expression can be divided into three categories: reference variables, dereference variables and address variables. So we have

$$Refs(\text{l.e}) = \{\text{x} \mid \text{x is a reference variable}\} \cup \{\text{y} \mid \text{y is a dereference variable}\}$$

$$\cup \{\text{z} \mid \text{z} \in \boldsymbol{lkpPT}(\text{y}, \boldsymbol{getPT}), \text{where y is a dereference variable}\}$$

The detail algorithm of *Refs*(l.e) is shown in Figure 8, where the *PtInfo*(l.e) function can obtain the point-to information generated by l.e.

The algorithm in Figure 8 addresses the issues of the reference and point-to information of a right-value expression, and hence facilitates the expansion of the existing slicing algorithm in Figure 5. The final expansion for the static slices of a **W** program with pointers is shown in Figure 9, which is generated by adding the bold and blue terms in Figure 5.

By introducing point-to analysis to our previous monadic slicing, we presented (in Figure 9) an approach of monadic slicing for a program with pointers. This approach obtained the point-to information through the data-flow iteration. Being different from the traditional methods where the point-to information and slicing are analyzed in two different phases, they are computed in the same phase in our method, by combining the forward monad slicing with data-flow iteration. Instead of recording point-to information for every statement, we only need to record the information for current analysis statements. So our method saves space without losing the precision. In addition, our approach also reserves the excellent properties of compositionality and language-flexibility from the original monadic slicing method.

```
Input:  an expression (e.g. e)
Output:  the reference set, Refs, and the point-to set, PtInfo, of the expression
Algorithm:
  case e of
    e is the expression of the form &y  →
        $Refs(e) = \phi$;  $PtInfo(e) = \{y\}$
    e is the expression of the form *y  →
        Let ys = lkpPT(y, getPT) where ys is the current point-to set of y;
        $Refs(e) = \{y\} \cup ys$;  $PtInfo(e) = \bigcup_{v \in ys} \textbf{lkpPT}(v, \textbf{getPT})$
    e is a pure variable such as y  →
        Let ys = lkpPT(y, getPT);
        $Refs(e) = \{y\}$;  $PtInfo(e) = ys$
    e is the compound expression with two sub-expressions $e_1$ and $e_2$  →
        $Refs(e) = Refs(e_1) \cup Refs(e_2)$;  $PtInfo(e) = PtInfo(e_1) \cup PtInfo(e_2)$
    e is a constant of a value type  →
        $Refs(e) = \phi$;  $PtInfo(e) = \phi$
  end case;
```

Fig. 8. The algorithm for the reference set and the point-to set of an expression.

6. Implementation and complexity analysis

In this section, we will implement the monadic slicing algorithms, and analyze its complexity as well.

Because of the use of monad transformers, modular denotational semantics achieves a high level of modularity and extensibility. Despite this, it is still executable: there is a clear operational interpretation of the semantics (Wadler, 1995). In ref. (Liang, 1995, 1998; Wadler, 1995), some modular compilers/interpreters using monad transformers were constructed. On the basis of these works, our monadic approach for static program slicing is feasible. Based on Labra's language prototyping system LPS (Labra et al, 2001), we developed a simple monadic slice prototype MSlicer (for more, see ref. (Zhang, 2007) or its website: https://sourceforge.net/projects/ lps). Its implementation language is Haskell, which is a purely functional language with lazy evaluation. The beauty of laziness allows Haskell to deal with infinite data, because Haskell will only load the data as it is needed, and because the garbage collector will throw out the data after it has been processed. Using higher-order

$\boldsymbol{M} :: \text{Prg} \to \text{ComptM}\ ()$

$\boldsymbol{M}[\![\ \textbf{program}\ \text{ide}\ \textbf{is}\ \text{b}\ ]\!] = \textit{return}\ ()$

$\boldsymbol{B} :: \text{Blk} \to \text{ComptM}\ ()$

$\boldsymbol{B}[\![\ \text{d}\ \textbf{begin}\ \text{c}\ \textbf{end}\ ]\!] = \textit{return}\ ()$

$\boldsymbol{D} :: \text{Dec} \to \text{ComptM Env} \quad \boldsymbol{E} :: \text{Exp} \to \text{ComptM Value}$

$\boldsymbol{D}[\![\ \textbf{const}\ \text{ide} = \text{l.e}\ ]\!] =$

$\{\ L \leftarrow \textit{rdLabels};\ T \leftarrow \textit{getSli};\ \boldsymbol{P \leftarrow \textit{getPT}};\ \textbf{ps} = \boldsymbol{\textit{PtInfo}}\textbf{(l.e)};\ \textbf{xs} = \boldsymbol{\textit{lkpPT}}\ \textbf{(ide,}\ \boldsymbol{P}\textbf{)};$

if (ide is of the form *x) then

$\boldsymbol{L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in \textit{Refs}(\text{l.e}) \cup \{x\}} \textit{lkpSli}\ (r, T)}\ ;\ \boldsymbol{\textit{xtdSli}}\ \textbf{(xs,}\ \boldsymbol{L', T}\textbf{)};\ \boldsymbol{\textit{xtdPT}}\ \textbf{(xs, ps,}\ \boldsymbol{P}\textbf{)};$

else

$L' \leftarrow \{l\} \cup L \cup \bigcup_{x \in \textit{Refs}(\text{l.e})} \textit{lkpSli}(x, T)\ ;\ \textit{updSli}(\text{ide}, L', T);\ \boldsymbol{\textit{updPT}}\ \textbf{(ide, ps,}$
$\boldsymbol{P}\textbf{)}\}$

$\boldsymbol{D}[\![\ \textbf{var}\ \text{ide} : \text{t}\ ]\!] = \textit{return}\ ()$

$\boldsymbol{D}[\![\ \text{d}_1;\ \text{d}_2\ ]\!] = \textit{return}\ ()$

$\boldsymbol{C} :: \text{Cmd} \to \text{ComptM}\ ()$

$\boldsymbol{C}[\![\ \text{ide} := \text{l.e}\ ]\!] =$

$\{\ L \leftarrow \textit{rdLabels};\ T \leftarrow \textit{getSli};\ \boldsymbol{P \leftarrow \textit{getPT}};\ \textbf{ps} = \boldsymbol{\textit{PtInfo}}\textbf{(l.e)};\ \textbf{xs} = \boldsymbol{\textit{lkpPT}}\ \textbf{(ide,}\ \boldsymbol{P}\textbf{)};$

if (ide is of the form *x) then

$\boldsymbol{L' \leftarrow \{l\} \cup L \cup \bigcup_{r \in \textit{Refs}(\text{l.e}) \cup \{x\}} \textit{lkpSli}\ (r, T)}\ ;\ \boldsymbol{\textit{xtdSli}}\ \textbf{(xs,}\ \boldsymbol{L', T}\textbf{)};\ \boldsymbol{\textit{xtdPT}}\ \textbf{(xs, ps,}\ \boldsymbol{P}\textbf{)};$

else

$L' \leftarrow \{l\} \cup L \cup \bigcup_{x \in \textit{Refs}(\text{l.e})} \textit{lkpSli}(x, T)\ ;\ \textit{updSli}(\text{ide}, L', T);\ \boldsymbol{\textit{updPT}}\ \textbf{(ide, ps,}\ \boldsymbol{P}\textbf{)}\}$

$\boldsymbol{C}[\![\ \text{c}_1;\ \text{c}_2\ ]\!] = \{\boldsymbol{C}[\![\ \text{c}_1\ ]\!];\ \boldsymbol{C}[\![\ \text{c}_2\ ]\!]\ \}$

$\boldsymbol{C}[\![\ \text{skip}\ ]\!] = \textit{return}\ ()$

$\boldsymbol{C}[\![\ \textbf{if}\ \text{l.e}\ \textbf{then}\ \text{c}_1\ \textbf{else}\ \text{c}_2\ \textbf{endif}\ ]\!] =$

$\{\ L \leftarrow \textit{rdLabels};\ T \leftarrow \textit{getSli};\ \boldsymbol{P \leftarrow \textit{getPT}};\ L' \leftarrow \{l\} \cup L \cup \bigcup_{x \in \textit{Refs}(\text{l.e})} \textit{lkpSli}(x, T)\ ;$

$\textit{inLabels}\ L'\ \boldsymbol{C}[\![\ c_1\ ]\!];\ T1 \leftarrow \textit{getSli};\ \boldsymbol{P1 \leftarrow \textit{getPT}};\ \textit{setSli}(T);\ \boldsymbol{\textit{setPT}(P)};$
$\textit{inLabels}\ L'\ \boldsymbol{C}[\![\ c_2\ ]\!];\ T2 \leftarrow \textit{getSli};\ \boldsymbol{P2 \leftarrow \textit{getPT}};\ \textit{mrgSli}(T1, T2);\ \boldsymbol{\textit{mrgPT}(P1, P2)}\}$

$\boldsymbol{C}[\![\ \textbf{while}\ \text{l.e}\ \textbf{do}\ \text{c}\ \textbf{endwhile}\ ]\!] =$

$\boldsymbol{\textit{Fix}}(\lambda f.\ \{L \leftarrow \textit{rdLabels};\ T \leftarrow \textit{getSli};\ L' \leftarrow \{l\} \cup L \cup \bigcup_{x \in \textit{Refs}(\text{l.e})} \textit{lkpSli}(x, T)\ ;\ \boldsymbol{P \leftarrow}$
$\boldsymbol{\textit{getPT}};$

$f \cdot \{\textit{inLabels}\ L'\ \boldsymbol{C}[\![\ c\ ]\!];\ T' \leftarrow \textit{getSli};\ \boldsymbol{P' \leftarrow \textit{getPT}};\ \textit{mrgSli}(T, T');\ \boldsymbol{\textit{mrgPT}}\textbf{(P, P')}\ \}\})$

$\boldsymbol{C}[\![\ \textbf{read}\ \text{l.ide}\ ]\!] = \{\ L \leftarrow \textit{rdLabels};\ T \leftarrow \textit{getSli};\ L' \leftarrow \{l\} \cup L;\ \boldsymbol{P \leftarrow \textit{getPT}};$
if (ide is of the form *x) then $\boldsymbol{\textit{xtdSli}(\textit{lkpPT}}\textbf{(x,}\ \boldsymbol{P}\textbf{),}\ \boldsymbol{L', T}\textbf{)}$
else $\textit{updSli}(\text{ide}, L', T)\ \}$

$\boldsymbol{C}[\![\ \textbf{write}\ \text{l.e}\ ]\!] = \textit{return}\ ()$

Fig. 9. Monadic slicing semantics for the **W** language with pointers.

functions from the libraries, Haskell modules can be written to concisely describe each language feature (Peterson et al, 1997; Thompson, 1996). Features such as arbitrary precision integer arithmetic, list comprehensions, infinite lists, all come in handy for the effective monadic slicing of a large program.

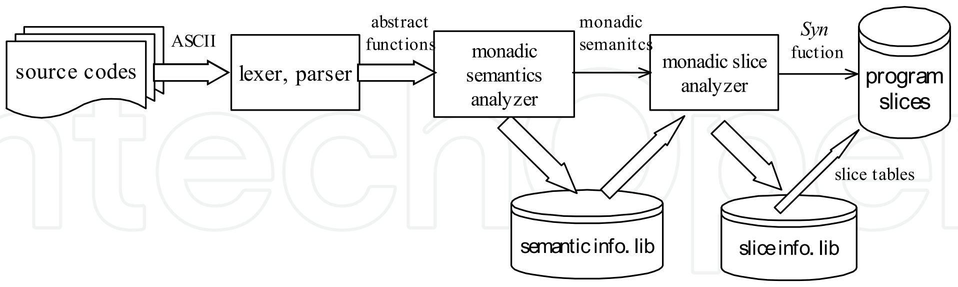
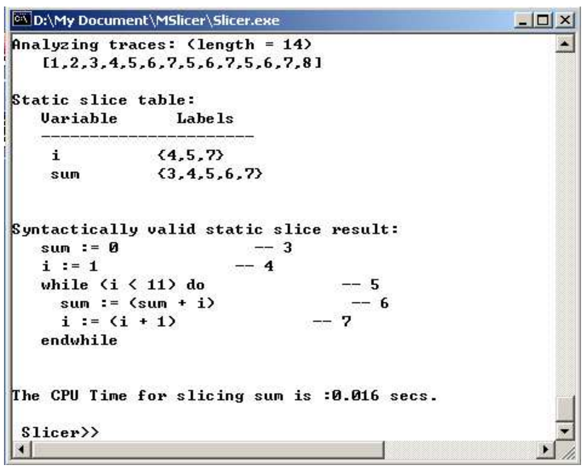


Fig. 10. The framework of the prototype MSlicer.



```
Analyzing traces: (length = 14)
   [1,2,3,4,5,6,7,5,6,7,5,6,7,8]

Static slice table:
   Variable      Labels
   ----------------------
    i           (4,5,7)
    sum         (3,4,5,6,7)


Syntactically valid static slice result:
   sum := 0                -- 3
   i := 1               -- 4
   while (i < 11) do                 -- 5
     sum := (sum + i)                   -- 6
     i := (i + 1)              -- 7
   endwhile


The CPU Time for slicing sum is :0.016 secs.

 Slicer>>
```

Fig. 11. The slice results of the sample program in Figure 7 from our MSlicer.

m	**statements**
1	var v1 = 1;
	⋮
5	var v5 = 5;
6	while (1 < 2) do
	⋱
5+*l*	while (1 < 2) do
6+*l*	v1 := v2;
	⋮
10+*l*	v5 := v1
	endwhile;

Fig. 12. A loop sample with *l* while statements.

Figure 10 gives the framework of the monadic slicer MSlicer. Figure 11 gives the static slice results for the example program in Figure 7 from our current monadic slicer. The final results also include the output value, analysis trace, static slice table and CPU time.

In practice, in order to obtain good performance, we choice the Haskell type IntSet instead of the original set type of Labels (i.e. [Int]) in Section 3. The implementation of IntSet is based on big-Endian Patricia trees (Okasaki & Gill, 1998; Morrison, 1968). This data structure performs especially well on binary operations like union and intersection. Many operations have a worst-case complexity of $\mathbf{O(min(}\boldsymbol{n},\boldsymbol{W}\mathbf{))}$, where n is the number of elements; W is the number of bits in an Int (32 or 64). This means that the operation can become linear in the number of elements with a maximum of W.

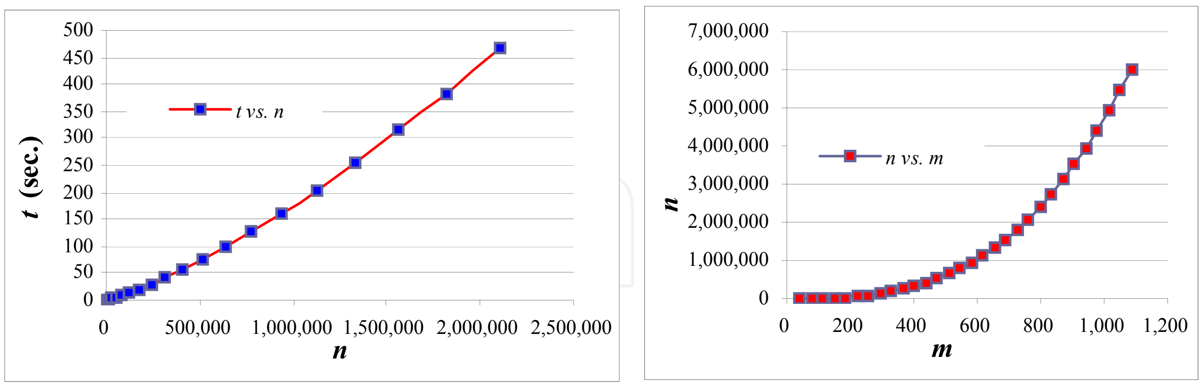
The measures of system size used below are those associated with the data structure of program slice Slices (which is a Hash table).

In a modular 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(s, L) of a concrete programming language. In the case of our example language **W**, the intermediate label set L' can be determined in worst-case time $O(v \times m)$, where v refers to the number of single variables in the program analyzed; and m is the number of labeled expressions in the program. To determine the Syn(s, L) shown in Figure 6 may cost $O(\min(m,W))$. Therefore the time cost of the predominant part of program slicing in the monadic slicing algorithm is bounded by $O(v \times m \times n)$, where n is the number of all labeled expressions appeared (perhaps repeatly) in the sequence of analyzing the program. In addition, an extra time cost $O(v \times \min(m,W))$ needs to get the executable slices of all single variables. Now we can see that the worst-case time of the whole static slice is $O(v \times m \times n)$. Since we finally obtain the static slices of all variables after the last statement is analyzed, the program slice of each variable, on the average, costs $O(m \times n)$. In fact, $n = O(m^2)$ at worst, for more see the following loop statement shown in Figure 12. So all of its static slices will cost the worst-case time $O(m^3)$.

To analyze the space complexity of the algorithms, we pay our attention to the constructions $Refs$(l.e), Slices, L' and L. We need space $O(v \times v)$ and $O(v \times m)$ to save $Refs$(l.e) and Slices, respectively. According to the definition of slice monad transformer SliceT in Figure 4, we need more intermediate labels when SliceT is applied to loop statements (e.g. while statements). So it takes the space $O(k \times m)$ to save intermediate labels, where k refers to the maximal times of analyzing the loop statements in the program (until the slice stabilizes) . The label set L will cost the space $O(m)$. Therefore, the total space cost is $O(v \times v + v \times m + k \times m)$.

By analyzing the complexity of algorithms in Figure 8 and 9, we find that the cost of point-to analysis is less than the cost of slicing. So our expansion algorithm to pointers has no additional complexity.

We have tested the complexity analysis of our monadic static algorithms by using the program with l while loop statements shown in Figure 12, which is similar to the while program in ref. (Binkley & Gallagher, 1996). From the results given in Figure 13, we can see that $n = O(m^2)$ at worst. This shows that the prototype monadic slicer MSlicer without optimization (such as BDD or SEQUITUR (Zhang et al, 2003) for the slice and trace tables) can compute slices for large analysis histories in reasonable time and space. (The experiments run on a Pentium Willamette (0.18) Windows PC system with 1GB RAM).



(a) t vs. n (b) n vs. m

Fig. 13. The relations between CPU time, *t*, and the number of all labeled expressions analyzed in practice, *n*, or the number of labeled expressions in the program, *m*.

7. Related work and comparisons

The original program slicing method was expressed as a sequence of data flow analysis problems (Weiser, 1984). An alternative approach was relied on program dependence graphs (PDG) (Ottenstein & Ottenstein, 1984). Most of the existing slicing methods were evolved from the two approaches. A few program slicing methods focused on the semantics of programs.

G.Canfora et al.'s *conditioned slicing* (Canfora et al, 1998) 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* (Harman & Danicic, 1997) allows for any simplifying transformations which preserve this semantic projection. These two methods are not really based on formal semantics of a program. P.A.Hauser et.al 's *denotational slicing* (Hausler, 1989; Ouarbya et al, 2002) employs the functional semantics of a program language in the denotational (and static) program slicer. G.A.Venkatesh (Venkatesh, 1991) also took account of denotational slicing with formal slicing algorithms including dynamic and static. This approach is indeed based on the standard denotational semantics of a program language. The language Venkatesh considered is a very simply one without pointers. We have extended it in this paper to a more realistic programming language containing pointers, but take an entirely different approach called *modular monadic slicing*.

Compared with the existing static slicing algorithms, the monadic static-slice algorithm has excellent flexibility, combinability and parallelizability properties, because it has abstracted the computation of static slicing as an independent entity, *static slice-monad transformer*. Our algorithm has allowed that static slices could be computed directly on abstract syntax, with no needs to explicitly construct intermediate structures such as dependence graphs.

In respect of accuracy, in Section 4 or in ref. (Zhang, 2007) we have stated that the slice results of monadic static slicing algorithm are not less precise than PDG-based ones. This is because the term L and $\bigcup_{r \in Refs(1.e)} lkpSli\ (r, getSli\ )$ 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 6) for monadic slicing algorithms, their time complexity of each variable is averagely $O(m^3)$ time. While the intra-procedural slicing algorithms based on dataflow equations can compute a slice in $O(v \times n' \times e)$ time, or averagely in $O(n' \times e)$ time for each variable, where n' is the number of vertices in the control flow graph (CFG) and e the number of edges in CFG (Weiser, 1984). Although the PDG-based algorithms extract slices in linear time (i.e. O(V + E), where V and E are the number of vertices and edges in the slice, respectively) after the PDG has been computed, a PDG can be constructed in $O(n' \times e + n' \times d)$ time, where d is the number of definitions in the program (Tip, 1995). Here V, n', e and d are the same complexity level of m, so the whole time of PDG-based algorithms (including the PDG-construct time) is also $O(m^3)$ nearly.

8. Conclusions and future work

In this paper, we have proposed a new approach for program slicing. We have called it *modular monadic program slicing* as it is based on modular monadic semantics. We have abstracted the computation of program slicing as a language-independence object, *slice monad transformer*. Therefore, the modular monadic slicing has excellent flexibility and reusability properties comparing with the existing program slicing algorithms. The modular monadic slicing algorithm has allowed that program slices could be computed directly on abstract syntax, with no needs to explicitly construct intermediate structures such as data flow graphs or dependence graphs.

As the behavior of a program is determined by the semantics of the language, it is reasonable to present the modular monadic program slicing. Furthermore, it is feasible, because modular monadic semantics is executable and some modular compilers/interpreters have already been existed.

For our future work, we will analyze slicing for programs with special features such as concurrent, object-oriented, exceptions and side-effects, by combining slice monad transformer with existing ones such as concurrent (Papaspyrou, 2001), object-oriented (Labra, 2002), non-determination, exceptions and side-effects (Moggi, 1991; Wadler, 1992, 2003). At the same time, we will improve our prototype of monadic slicers and give more comparisons with other slicing methods in experiments.

9. Acknowledgments

We thank Dr. J.Labra for his cooperating with us in the implementation of the slicer. This work was supported in part by the National Natural Science Foundation of China (60703086, 60903026, 60873049, 60973046), the Priority Academic Program Development of Jiangsu Higher Education Institutions(PAPD), Qing Lan Project of Jiangsu Province, NSF of Jiangsu Higher Education Institutions of China (10KJB520015),"Climbing"Program of Nanjing University of Posts and Telecommunications of China (NY210009), Open Foundation of Guangxi Key Laboratory of Trusted Software(Guilin University of Electronic Technology).

10. References

Weiser M (1984). Program Slicing. *IEEE Transaction on Software Engineering*, vol. 16, no. 5, pp. 498-509.

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

Tip F (1995). A Survey of Program Slicing Techniques. *Journal of Programming Languages*, vol.3, no.3, pp.121-189.

Harman M & Hierons R (2001). An Overview of Program Slicing. *Software Focus*, vol. 2, no. 3, pp. 85-92.

Binkley D & Gallagher K (1996). Program Slicing. *Advances in Computers*, vol. 43, pp. 1-50.

Gallagher K & Lyle J (1991). Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751-761.

Hausler P (1989). Denotational Program Slicing. *Proceeding of 22th Annual Hawaii International Conference on System Sciences*, vol. 2, pp. 486-495.

Ouarbya L; Danicic S & Daoudi M, et al (2002). A Denotational Interprocedural Program Slicer. *Proceeding of 9th IEEE Working Conference on Reverse Engineering*, IEEE Press, Virginia, pp. 181-189.

Venkatesh G (1991). The Semantic Approach to Program Slicing. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, pp. 26-28.

Moggi E (1991). Notions of Computation and Monads. *Information and Computation*, vol. 93, pp. 55-92.

Liang S & Hudak P (1996). Modular Denotational Semantics for Compiler Construction. *Proceeding of 6th European Synposium on Programming Languages and Systems, ESOP'96. LNCS 1058*, Springer-Verlag, Berlin, pp. 219-234.

Wansbrough K (1997). A Modular Monadic Action Semantics. Master thesis, University of Auckland, Auckland.

Mosses P (1998). Semantics, Modularity, and Rewriting Logic. *Proceeding of 2nd International Workshop on Rewriting Logic and its Applications, ENTCS 15*, Elesvier Press, Netherlands.

Zhang Y Z & Xu B W (2004). A Survey of Semantic Description Frameworks for Programming Languages. *ACM SIGPLAN Notices*, vol. 39, no. 3, pp.14-30.

Wadler P (1992). Comprehending monads. *Mathematical Structures in Computer Science*, vol. 2, pp. 461-493.

Espinosa D (1995), "Semantic Lego", PhD dissertation, Columbia University, Columbia.

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

Zhang Y Z; Xu B W & Shi L, et al (2004). Modular Monadic Program Slicing, *The 28th Annual International Computer Software and Applications Conference, COMPSAC 2004*, Hong Kong, China, pp. 66-71.

Zhang Y Z; Xu B W & Labra G (2005). A Formal Method for Program Slicing, *Australian Software Engineering Conference, ASWEC'05*, Australian, pp. 140-148.

Zhang Y Z & Xu B W (2005). A Slice Monad Transformer and Its Applications in Program Slicing, *The 10th IEEE International Conference on Engineering of Complex Computer Systems,* ICECCS'05, Shanghai, China, pp. 147-154.

Wu Z Q; Zhang Y Z & Xu B W (2006). Modular Monadic Slicing in the Presence of Pointers, *In: Alexandrov V N, Albada G D, Sloot P M, et al, eds. 6th International Conference on Computational Science. LNCS 3994*. Reading: Springer-verlag, pp.748-756.

Zhang Y Z; Labra G & del R (2006). A Monadic Program Slicer, *ACM SIGPLAN Notices*, vol. 41, no. 5, pp. 30-38.

Zhang Y Z (2007). A Novel Formal Approach to Program Slicing, *Science in China F: Information Sciences*, vol. 50, no. 5, pp. 657-670.

Wadler P & Thiemann P (2003). The Marriage of Effects and Monads, *ACM Transactions on Computational Logic*, vol. 4, no. 1, pp. 1-32.

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

Liang S (1998). Modular Monadic Semantics and Compilation, PhD dissertation, University of Yale, Yale.

Wadler P (1995). Monads for Functional Programming, *Lecture Notes on Advanced Functional Programming Techniques*, *LNCS 925*, Springer-Verlag, Berlin, pp. 24-52.

Newbern J (2002). All About Monads, http://www.haskell.org/all about_monads/html /index.html

Slonneger K & Kurtz B (1995). *Formal Syntax And Semantics of Programming Language: A Lab Based Approach*. Addison & Wesley.

Venkatesh G (1990). Semantics of Program Slicing, *Bellcore TM-ARH-018561*.

Horwitz S; Pfeiffer P & Reps T (1989). Dependence Analysis For Pointer Variables, In: *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 28-40.

Hind M; Burke M & Carini P, et al (1999). Interprocedural Pointer Alias Analysis, ACM Transactions on Programming Languages and Systems, 21(4): 848~894.

Kahl W (2003). A Modular Interpreter Built with Monad Transformers, *Course Lectures on Functional Programming*, CAS 781.

Labra G; Luengo D & Cueva L (2001). A Language Prototyping System Using Modular Monadic Semantics, *Workshop on Language Definitions, Tools and Applications, LDTA'01*, Netherlands.

Peterson J; Hammond K & Augustsson L, et al (1997). Report on the Programming Language Haskell 1.4, A Non-strict Purely Functional Language, *Yale University Technical Report, YALEU/DCS /RR-1106*.

Thompson S (1996). *Haskell: The Craft of Functional Programming*, Addison-Wesley, Harlow, England.

Okasaki C & Gill A (1998). Fast Mergeable Integer Maps, *Workshop on ML*, September, pp.77-86.

Morrison D (1968). PATRICIA-- Practical Algorithm To Retrieve Information Coded In Alphanumeric, *Journal of the ACM*, vol. 15, no. 4, pp. 514-534.

Zhang X; Gupta R; Zhang Y (2003). Precise Dynamic Slicing Algorithms, *Proceedings of the 25th International Conference on Software Engineering*, IEEE CS Press, Washington DC, pp. 319-329.

Ottenstein K & Ottenstein M (1984). The program dependence graph in a software development environment, *ACM SIGPLAN Notices*, vol.19, no. 5, pp. 177-184.

Canfora G; Cimitile & De L (1998). Conditioned Program Slicing, *Information and Software Technology*, vol. 40, no. 11/12, pp. 595-607.

Harman M & Danicic S (1997). Amorphous Program Slicing, *IEEE International Workshop on Program Comprehesion*, *IWPC'97*, IEEE CS Press, Los Alamitos, pp. 70-79.

Papaspyrou N (2001). A Resumption Monad Transformer and its Applications in the Semantics of Concurrency, *Technical Report CSD-SW-TR-2-01*, National Technical University of Athens, Software Engineering Laboratory.

Labra G; Luengo D & Cueva L, et al (2002). Reusable Monadic Semantics of Object Oriented Programming Languages, *Proceeding of 6th Brazilian Symposium on Programming Languages*, *SBLP'02*, PUC-Rio University, Brazil.