

## Distributed Data and Programs Slicing

Mohamed A. El-Zawawy<sup>1,2</sup>

<sup>1</sup>College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU),  
Riyadh, Kingdom of Saudi Arabia

<sup>2</sup>Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt  
[maelzawawy@cu.edu.eg](mailto:maelzawawy@cu.edu.eg)

**Abstract:** This paper presents a new technique for data slicing of distributed programs running on a hierarchy of machines. Data slicing can be realized as a program transformation that partitions heaps of machines in a hierarchy into independent regions. Inside each region of each machine, pointers preserve the original pointer structures in the original heap hierarchy. Each heap component of the base type (e.g., the integer type) goes only to a region of one of the heaps. The proposed technique has the shape of a system of inference rules. In addition, this paper presents a simply structure type system to decide type soundness of distributed programs. Using this type system, a mathematical proof that the proposed slicing technique preserves typing properties is outlined in this paper as well. [El-Zawawy MA. **Distributed Data and Programs Slicing**. *Life Sci J* 2013;10(4):1361-1369]. (ISSN: 1097-8135). <http://www.lifesciencesite.com>. 180

**Keywords:** Distributed data; distributed Programs; data slicing; programs slicing; type systems; program transformation.

### 1. Introduction

Breaking down a large distributed program into smaller pieces or minimizing its size is essential for many software analysis techniques such as parallelization [19], debugging [22], program comprehension [14], testing [16], downsizing, and restructuring. Introduced by Mark Weiser [20], data and program slicing [3] are applicable techniques for narrowing the focus of a program to a certain region of the memory. Interesting enough, data slicing was motivated by the desire of guiding students through debugging programs. A program slice can be defined as an executable subset of the program that simulates the original program on a certain data slice (region of the memory). Data slicing is useful when compilers need to modify data structures in the program being compiled without breaching pre-compiled assumptions about data layout.

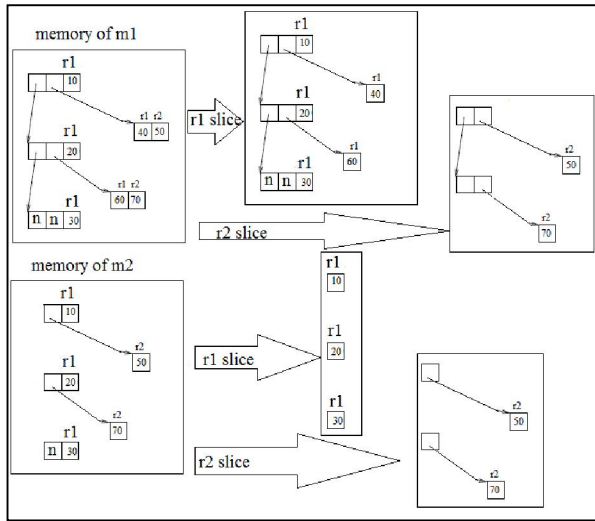
Distributed systems [15] are typically constructed on hierarchical memory structure (Local stores and caches of processors, such as cell game processor, are organized in a hierarchal fashion). This memory system equips each process with explicitly controlled local caches or stores. Distributed computations and their hierarchy models of memories have been the focus of much research activities. This is partially justified by the existence of low-cost processors facilitating building such distributed systems. Rather than distributed systems of a number of processors create immunity against different sorts of failures, they also have incremental growth capabilities and high throughput. One example of memory hierarchy is to partition memory into computational grids consisting of clusters

partitioned into nodes including program threads. Practically, memories of most distributed programming languages have a two level abstraction. The model of distributed systems used in this paper is the *single program multi data (SPMD)* model [15].

Type systems [13] are theoretical means for proving type soundness which amounts to the absence of method-not-found and field-not-found errors. For distributed programming languages, like *DLang* of this paper, a type system guaranties that every use of a location considers its predefined type. Proving this property is, however, not easy due to the potential intervention of executing the program on different machines of hierarchy.

This paper presents a new technique for slicing distributed programs running on hierarchal memories. The proposed technique has the form of inference rules which are simply structured. The new technique is illustrated using a simple, however rich, model of distributed programming language (*DLang* - Figure 2). The paper also presents a type system that checks type soundness of programs of *DLang* and programs resulting from the proposed slicing technique. A prove that prosperity of type soundness of a program is preserved by the slicing technique is also presented in this paper.

Rather than the traditional algorithmic way, using a system of inference rules [4–6] to achieve static analyses and transformations of distributed programs has recently proved to be a good choice. This is so as derivations in the system of rules work as simply-structured correctness proofs for results of the system. Such proofs are required by many applications like proof-carrying code [9].



**Fig. 1.** A motivating example for distributed data and program slicing.

**Motivation**

Suppose a distributed system including two machines  $m_1$  and  $m_2$  each of them has two memory regions  $r_1$  and  $r_2$ . On this system, suppose also a distributed program defining the following types:

$type\ t_1 = struct$   
 $\{y_1: ptr_1^1, y_2: ptr_2^1, y_3: ptr_2^2, y_4: int(r_1, \{m_1, m_2\})\};$   
 $type\ t_2 = struct$   
 $\{y_1: int(r_1, \{m_1\}), y_2: int(r_2, \{m_1, m_2\})\};$

The left-hand-side of Figure 1 illustrates how these data types will be allocated on memories of machines  $m_1$  and  $m_2$ . The right-hand-side of the figure explains distributed data slicing effects on the two machines into the two regions. Annotations are used in integer fields of data structures to determine regions of machines that will host these fields. The slicing includes inserting shapes of original data structures in each region of each machine. However such a shape of a region typically contains only relevant data and necessary pointers.

The aim of this paper is to provide formal techniques to transform memories of  $m_1$  and  $m_2$  into that on the right-hand-side Figure 1. Moreover the proposed technique is required to slice the program that includes the type definitions into slices each works on the data of a certain region.

**Contributions**

Contributions of this paper include:

1. A new type system for checking type soundness of distributed programs.
2. A novel and sound technique for slicing distributed data and programs running on hierarchal systems of memories.

3. A mathematical proof that the proposed slicing technique preserves typing properties.

**Organization**

The organization in the rest of the paper is as following. The syntax of the target language and a system of inference rules for type checking of the language constructs are presented in Section 2. Section 3 introduces the new technique for slicing of distributed programs running on hierarchal machines. The type system of Section 2 is used in Section 3 to prove that the proposed slicing technique preserves type-soundness of sliced programs. Related work is reviewed in Section 4.

$x, y \in IVar, n \in \mathbb{Z}, i_{op} \in \mathbb{I}_{op}, \text{ and } b_{op} \in \mathbb{B}_{op},$   
 $\text{ and } m \in M \subseteq \mathcal{M}$   
 $\tau \in Types ::= int(r_i, M) \mid ptr^m \tau \mid t$   
 $\quad \mid struct\{y_1: \tau_1, \dots, y_n: \tau_n\}.$   
 $d \in Defs ::= type\ t = \tau \mid t_1 t_2 \mid \epsilon.$   
 $l \in lexpr ::= x \mid l.y \mid *e.$   
 $e \in Expr ::= l \mid e_1\ i_{op}\ e_2 \mid \&l \mid new\ \tau \mid$   
 $modify - w(e, m) \mid compute\ e\ at\ m \mid cast <$   
 $ptr^m \tau \rightarrow int(r_i, M) > e \mid cast < int(r_j, M_j) \rightarrow$   
 $int(r_i, M_i) > e.$   
 $S \in Stmts ::= skip \mid l := e \mid compute\ S\ at\ m$   
 $\quad \mid S_1; S_2 \mid if\ e\ then\ S_t\ else\ S_f$   
 $\quad \mid while\ e\ do\ S_t.$   
 $p \in Progs ::= dS$

**Fig. 2.** The programming language model,  $\mathcal{DLang}$ .

**2. Target Language  $\mathcal{DLang}$ : Syntax and Type Checking**

This section presents the syntax of the target language and a system of inference rules for type checking of the language constructs. The syntax of the target programming language,  $\mathcal{DLang}$ , is presented in Figure 2. We assume an arbitrary hierarchy of machines. In term of parallel programs, a single execution thread resembles a machine. A countably infinite collection of variables (machine-private) is used in the language and denoted by  $IVar$  with typical elements  $x, y$ . Sets of finite arithmetic and Boolean operations are denoted by  $\mathbb{I}_{op}$  and  $\mathbb{B}_{op}$ , respectively. The set of machine identifiers is denoted by  $\mathcal{M}$ . The types of the language are integer, pointer, and named types ( $t$ ). The empty structure,  $struct\{\}$ , is denoted by  $void$  as a shorthand. It is noted that base types (integer types) are quantified (Similar type quantification can be found in [3]) with pairs of a region and a set of machines.

This is so to determine the locations of the data of these types. The language programs are executed on a distributed system of  $\delta$  machines with *identifiers*  $m_1$  through  $m_\delta$ . Each machine has  $\alpha$  regions named  $r_1$  through  $r_\alpha$ . The set of all the regions is denoted by  $\mathfrak{R}$ .  $\mathcal{DLang}$  can be realized as a generalization of the language in [10].

The hierarchy level hosting the smallest common ancestor of two machines is the *distance* between the two machines. The number of levels in the machines hierarchy is dubbed the *depth* of the hierarchy. The *width* of a pointer on a machine is the distance between the machine hosting the pointer and the machine hosting the location pointed-at by the pointer. We assume a function *width-f* that assigns each pointer its width. The symbol  $h$  denotes the height of the machine hierarchy. Therefore the set of widths of pointers is  $\{1, \dots, h\}$  and the pointer type is parameterized by the machine  $id$  of the location it points-at.

Expressions ( $e$ ) and l-expressions ( $l$ ) are inspired by that of  $\mathcal{C}$ . Binary operations (arithmetic and boolean) are only applicable to integers in the same region of possibly different machines. Expressions include:

- *new*  $\tau$ : allocates a memory location of type  $\tau$  and returns the allocated address.
- *modify*  $- w(e, m)$ : changes the pointer width. More specifically, it changes the machine the expression points-at. Hence the type of  $e$  becomes the reference  $ptr^m \tau$ , rather than  $ptr^{m'} \tau$ .

- *compute e at m*: computes the expression  $e$  on the machine  $m$  and distributes the value to other machines.
- *cast*  $\langle ptr^m \tau \rightarrow int(r_i, M) \rangle e$ : casts from pointers to integers in different regions of different machines.
- *cast*  $\langle int(r_j, M_j) \rightarrow int(r_i, M_i) \rangle e$ : casts between integers in different regions of different machines.

The statement *compute S at m* executes the statement  $S$  on the machine  $m$ . A  $\mathcal{DLang}$  program is a sequence of type definitions followed by a statement.

**Remark 1.** Primitive values such as Boolean values and integers are not incorporated in the language  $\mathcal{DLang}$ . It is straightforward to include them as an extension. Although the language  $\mathcal{DLang}$  is  $SPMD$ , the proposed data-slicing technique is easily extendable to other parallelism models. However these extensions are not considered in this paper.

A system of inference rules for type checking of components of  $\mathcal{DLang}$  is presented in Figures 3 and 4. A *context*,  $\Gamma$ , is a map from variables and type names to types. A program is well-typed (WT) if its body,  $S$ , is well-typed with initial environment  $\Gamma_f$ . In the allocation rule (*new* $_t$ ), the expression generates a pointer type  $ptr^m \tau$  for all  $m$ . As the allocation takes place on the same machine that is executing the statement, the width of the created pointer is 1.

$$\begin{array}{c}
 \frac{\Gamma \models_l l : \tau}{\Gamma \models_e l : \tau} (l^t) \quad \frac{\Gamma \models_e e_1 : int(r_i, M) \quad \Gamma \models_e e_2 : int(r_i, M)}{\Gamma \models_e e_1 \text{ } i_{op} \text{ } e_2 : int(r_i, M)} ((e_1 \text{ } i_{op} \text{ } e_2)^t) \\
 \\
 \frac{m \in \mathcal{M}}{\Gamma \models_e \text{new } \tau : ptr^m \tau} (new^t) \quad \frac{\Gamma \models_l l : \tau}{\Gamma \models_e l : ptr^m \tau} (\&l^t) \quad \frac{\Gamma \models_e e : \tau' \quad \tau' \sqsubseteq \tau}{\Gamma \models_e e : \tau} (\sqsubseteq^t) \\
 \\
 \frac{\text{width} - f(e) = m' \quad \Gamma \models_e e : ptr^{m'} \tau}{\Gamma \models_e \text{modify} - w(e, m) : ptr^m \tau} (\text{modify} - w^t) \quad \frac{\Gamma \models_e e : \tau}{\Gamma \models_e \text{compute } e \text{ at } m : \tau} (\text{comp}^t) \\
 \\
 \frac{\Gamma \models_e e : int(r_j, M_j)}{\Gamma \models_e \text{cast} \langle int(r_j, M_j) \hookrightarrow int(r_i, M_i) \rangle e : int(r_i, M_i)} (\text{cast}_1^t) \\
 \\
 \frac{\Gamma \models_e e : ptr^m \tau(r_i, M) \text{ has a type reachable form } \tau}{\Gamma \models_e \text{cast} \langle ptr^m \tau \hookrightarrow int(r_i, M) \rangle e : int(r_i, M)} (\text{cast}_2^t)
 \end{array}$$

**Fig. 3.** Typing rules for expression.

The *modify* expression enables modifying the machine *id* that a pointer references. This, of course, results in changing the top-level expression width. This way is used to decrease the width of the expression more often than to increase it. This is so because of the subtyping rule. After a dynamic analysis, the *modify* statement can be used to express that the expression references data on a machine closer than initially thought. The inclusion relationship in the rule ( $\subseteq_t$ ) applies only on *structure* types. The remaining rules are self-explanatory.

Hence if, for example, the data of a liked list in the original heap hierarchy are annotated with different regions of different machines, then every region on each machine will include a similar linked list with only the list data for this region. Hence whereas base fields will be divided among regions of different machines in the hierarchy according to their annotations, the same pointer maybe replicated into many regions as necessary. Therefore in data slicing base fields are separated and pointers are replicated.

$$\begin{array}{c}
\frac{}{\Gamma \vDash_t \epsilon : WT}(\epsilon^t) \quad \frac{\Gamma(t) = \tau}{\Gamma \vDash_t \text{type } t = \tau : WT}(t^t) \quad \frac{\Gamma \vDash_t d_1 : WT \quad \Gamma \vDash_t d_2 : WT}{\Gamma \vDash_t d_1 d_2 : WT}(t^t) \\
\frac{x \in \text{dom}(\Gamma)}{\Gamma \vDash_1 x : \Gamma(x)}(x_1^t) \quad \frac{x \in \text{dom}(\Gamma)}{\Gamma \vDash_1 x.(r_i, m) : \Gamma(x)}(x_2^t) \quad \frac{\Gamma \vDash_1 y : \tau_1 \quad \{y : \tau_2\} \subseteq \tau_1}{\Gamma \vDash_1 l.y : \tau_2}(l.y^t) \\
\frac{\Gamma \vDash_e e : ptr^m \tau}{\Gamma \vDash_l *e : \tau}(*e^t) \quad \frac{}{\Gamma \vDash_s \text{skip} : WT}(\text{skip}^t) \quad \frac{\Gamma \vDash_1 l : \tau \quad \Gamma \vDash_e e : \tau}{\Gamma \vDash_s l := e : WT}(:=^t) \\
\frac{\Gamma \vDash_s S : WT}{\Gamma \vDash_s \text{compute } S \text{ at } n : WT}(\text{compute}^t) \quad \frac{\Gamma \vDash_s S_1 : WT \quad \Gamma \vDash_s S_2 : WT}{\Gamma \vDash_s S_1 S_2 : WT}(\text{Seq}^t) \\
\frac{\Gamma \vDash_e e : \text{int}(r_i, M) \quad \Gamma \vDash_s S_t : WT \quad \Gamma \vDash_s S_f : WT}{\Gamma \vDash_s \text{if } e \text{ then } S_t \text{ else } S_f : WT}(\text{if}^t) \\
\frac{\Gamma \vDash_e e : \text{int}(r_i, M) \quad \Gamma \vDash_s S_t : WT}{\Gamma \vDash_s \text{while } e \text{ do } S_t : WT}(\text{wle}^t)
\end{array}$$

**Fig. 4.** Typing rules for type definitions, left expressions, statements, and programs.

### 3. Data Slicing of *DLang*

This section presents a new technique for data slicing [3] of distributed programs [10] running on hierarchal machines. The type system of the previous section is used later in this section to prove that the proposed technique preserves type-soundness of sliced programs. Data slicing of distributed programs aims at dividing heaps of hierarchy (distributed) machines into separate regions. The base types (only integers in our case) of the input program of data slicing have to be region-machine-annotated. The result of the slicing is a new program whose data structures are split into separate regions of the heaps of hierarchy machines. Of course the new and original programs have to be semantically equivalent. Data must be contained in regions of machine hierarchy according to data annotations; adding new pointers to do so is allowed. However complete independence of regions is assumed; cross-region boundaries pointers are not allowed.

As a result of data slicing, every machine region reflects the original structure of heaps in machine hierarchy (for an example see Figure 1).

**Example:** Consider applying the slicing technique on the statement compute:

*compute \*x.y<sub>1</sub> at m<sub>1</sub>,*

where the type of *x* is *ptr*<sup>1</sup>*t*<sub>2</sub> and *t*<sub>2</sub> is the type defined in the motivating example illustrated by Figure 1. The result of the slicing will be the statement:

*compute \*x.(r<sub>1</sub>, m<sub>1</sub>).y<sub>1</sub> at m<sub>1</sub>.*

This amounts to returning the value of *y*<sub>1</sub> of structure *x* hosted by region *r*<sub>1</sub> of machine *m*<sub>1</sub>.

A simple structure induction, on structure of type's  $\tau$ , proves Lemma 1 reasoning about type transformations of Figure 5.

**Lemma 1.** *Suppose that  $\tau \rightsquigarrow_{(m,i)}^\tau \tau'$ . Then ( $\tau' \neq \text{void}$ ) implies  $\tau$  and  $\tau'$  are of the same type.*

Using Lemma1, it is not hard to prove Corollary 1, describing transformations of Figure 6.

**Corollary 1.** *Suppose that  $d \rightsquigarrow_{(m,i)}^\tau d'$ . Then  $d$  and  $d'$  define equivalent types.*

Figures 5, 6, and 7 present the proposed slicing technique. Inference rules for slicing types over regions and machines of a distributed system are shown in Figure 5. The main notation in this figure is

$\tau \rightsquigarrow_{(m,i)}^{\tau} \tau'$  meaning that the type  $\tau'$  is the slice of the original type  $\tau$  on the region  $r_i$  of the machine  $m$ . The rules  $(int_1^s)$  and  $(int_2^s)$  express that the slice of region  $r_i$  of machine  $m$  includes only integers annotated with the pair  $(m, i)$ . The rules  $(ptr_1^s)$ ,  $(ptr_2^s)$ , and  $(str^s)$  for pointers, and structures recursively invoke the inference rules of the figure. For a machine  $m$ , the rule  $(str^m)$  calculates the slices of a type  $\tau$  on the regions of the machine.

the rule only involves moving a single integer between regions. On the other side, as clarified by the rule  $(cast_2^s)$ , the transformation of an expression of pointer casting is affected by the casting being region-specific. Rules for slicing over machines of left expression, statement, and program are presented in figure 8. According to the rule  $(l^s)$  slicing a reference to a variable amounts to selecting the element of the variable belonging to the addressed region. A key rule in the proposed technique is that of

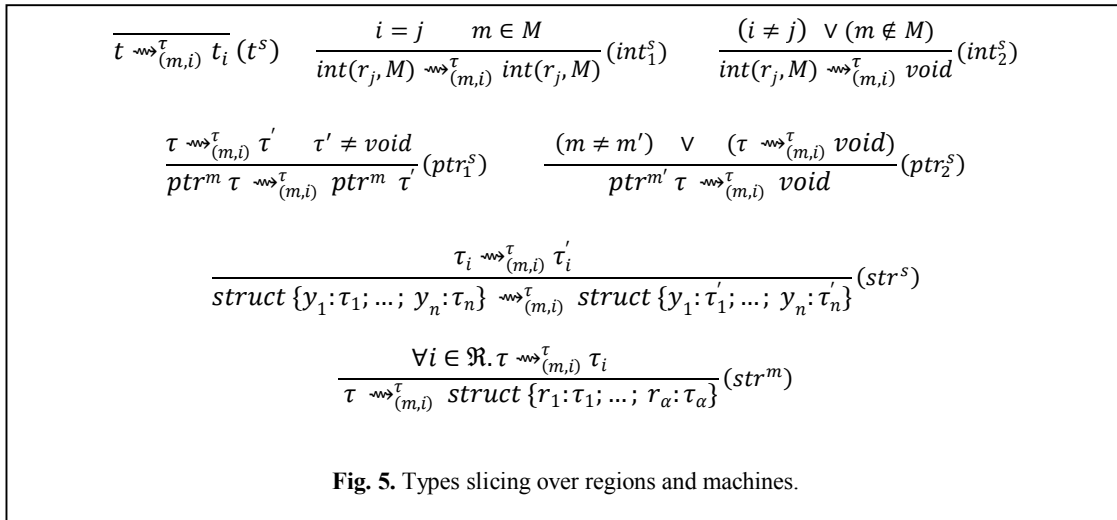


Fig. 5. Types slicing over regions and machines.

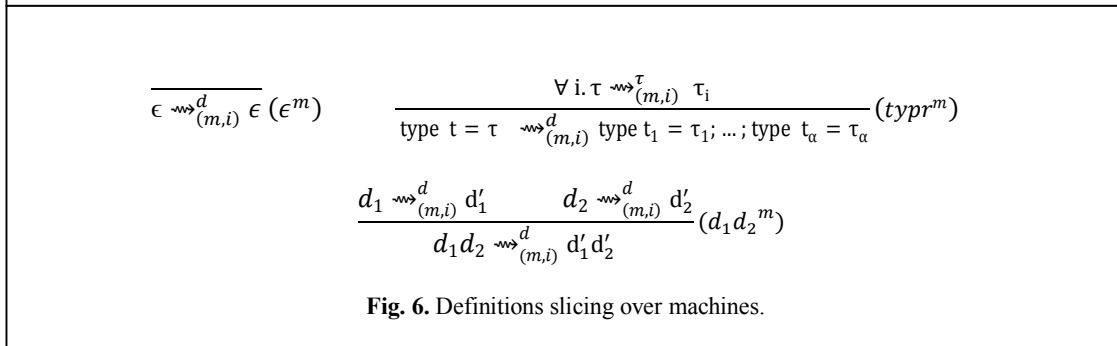


Fig. 6. Definitions slicing over machines.

Figure 6 shows rules for slicing definitions over regions of machines. The rule  $(typr^m)$  is the basic one for definition slicing. In this rule, the right-hand-side of a type definition is sliced over different regions of a machine  $m$  transforming the original definition statement into  $\alpha$  statements on  $m$ .

Inference rules for expression slicing over regions are included in Figure 7. The rule  $(cast_1^s)$  recursively uses expression rules to compute an integer in region  $r_j$ . This integer is then casted into this region. Clearly the assumption that inter-region pointers are not allowed is preserved by this rule as

assignment  $(:=^s)$ . The main idea behind this rule is to achieve, if the assigned type is not void, the corresponding assignment in every region of every machine.

**Remark 2.** The fact that assignments are done separately in different regions of different machines is the reason that most of the inference rules on the proposed technique rules are region-oriented. This is done assuming the existence of the expression sliced-type in the addressed region.

The following results prove that the proposed data slicing technique preserves type-checking properties of distributed programs.



$$\begin{array}{c}
\frac{l \rightsquigarrow_{(m,i)}^l l'}{l \rightsquigarrow_{(m,i)}^e l'} (l^s) \quad \frac{e_1 \rightsquigarrow_{(m,i)}^e e'_1 \quad e_2 \rightsquigarrow_{(m,i)}^e e'_2}{e_1 \text{ i}_{op} e_2 \rightsquigarrow_{(m,i)}^e e'_1 \text{ i}_{op} e'_2} ((e_1 \text{ i}_{op} e_2)^s) \\
\\
\frac{\tau \rightsquigarrow_{(m,i)}^\tau \tau'}{\text{new } \tau \rightsquigarrow_{(m,i)}^\tau \text{new } \tau'} (\text{new}^s) \quad \frac{l \rightsquigarrow_{(m,i)}^l l'}{\& l \rightsquigarrow_{(m,i)}^e \& l'} (\&l^s) \\
\\
\frac{e \rightsquigarrow_{(m,i)}^e e'}{\text{modify} - w(e, d) \rightsquigarrow_{(m,i)}^e \text{modify} - w(e', d)} (\text{modify} - w^s) \\
\\
\frac{e \rightsquigarrow_{(m,i)}^e e'}{\text{compute } e \text{ at } n \rightsquigarrow_{(m,i)}^e \text{compute } e' \text{ at } n} (\text{comp}^s) \\
\\
\frac{e \rightsquigarrow_{(m,i)}^e e'}{\text{cast} < \text{int}(r_j, M_j) \rightarrow \text{int}(r_i, M_i) > e \rightsquigarrow_{(m,i)}^e \text{cast} < \text{int}(r_j, M_j) \rightarrow \text{int}(r_i, M_i) > e'} (\text{cast}_1^s) \\
\\
\frac{\tau \rightsquigarrow_{(m,i)}^\tau \tau' \quad e \rightsquigarrow_{(m,i)}^e e'}{\text{cast} < \text{ptr}^m \tau \rightarrow \text{int}(r_i, M) > e \rightsquigarrow_{(m,i)}^e \text{cast} < \text{ptr}^m \tau' \rightarrow \text{int}(r_i, M) > e'} (\text{cast}_2^s)
\end{array}$$

**Fig. 7.** Expression slicing over regions.

**Lemma 2.** Suppose that  $l \rightsquigarrow_{(m,i)}^l l'$  and  $\Gamma \vDash_l l: \tau$ . Then  $\Gamma \vDash_l l': \tau$ .

**Proof.** The proof is by structure induction on the structure of left expressions,  $l$ , as follows.

- The case  $l = x$ : in this case  $l' = x.(r_i, m)$ . By rules  $(x_1^t)$  and  $(x_1^s)$ ,  $\Gamma \vDash_l l: \Gamma(l)$  and  $\Gamma \vDash_l l': \Gamma(l)$ . Hence the required is satisfied.
- The case  $l = l.y$ : in this case  $l' = l'.y$ . By the rule  $(l^s)$ , it is true that  $l \rightsquigarrow_{(m,i)}^l l'$ . Now since  $\Gamma \vDash_l l.y: \tau$ , then by rule  $(l.y^t)$  there exists  $\tau_1$  such that  $\Gamma \vDash_l l: \tau_1$  and  $\{y: \tau_1\} \subseteq \tau$ . By induction hypothesis,  $\Gamma \vDash_l l': \tau_1$ . Therefore by the rule  $(l.y^t)$ ,  $\Gamma \vDash_l l'.y: \tau$  as required.
- The case  $l = *e$ : in this case  $l' = *e'$ . By the rule  $(*e^s)$ , it is true that  $e \rightsquigarrow_{(m,i)}^e e'$ . Now since  $\Gamma \vDash_l *e: \tau$ , then by rule  $(*e^t)$  there exists  $\tau_1$  such that  $\Gamma \vDash_e e: \tau_1$  and  $\tau_1 = \text{ptr}^m \tau$ . By Lemma 3,  $\Gamma \vDash_e e': \text{ptr}^m \tau$ . Therefore by the rule  $(*e^t)$ ,  $\Gamma \vDash_l *e': \tau$  as required.

**Lemma 3.** Suppose that  $e \rightsquigarrow_{(m,i)}^e e'$  and  $\Gamma \vDash_e e: \tau$ . Then  $\Gamma \vDash_e e': \tau$ .

**Proof.** The proof is by structure induction on the structure of expressions,  $e$ . Some cases are shown below.

- The case  $e = l$ : in this case, by the rule  $(l^s)$ ,  $l \rightsquigarrow_{(m,i)}^l l'$  and  $e' = l'$ . By Lemma 2,  $\Gamma \vDash_l l: \tau$  implies  $\Gamma \vDash_l l': \tau$ . Hence by the rule  $(l^t)$ ,  $\Gamma \vDash_e e': \tau$  as required.
- The case  $e = e_1 \text{ i}_{op} e_2$ : in this case, by the rule  $((e_1 \text{ i}_{op} e_2)^t)$ ,  $\tau = \text{int}(r_i, M)$ . Moreover  $\Gamma \vDash_e e_1: \text{int}(r_i, M)$  and  $\Gamma \vDash_e e_2: \text{int}(r_i, M)$ . Also by the rule  $((e_1 \text{ i}_{op} e_2)^s)$ , we have  $e_1 \rightsquigarrow_{(m,i)}^e e'_1$  and  $e_2 \rightsquigarrow_{(m,i)}^e e'_2$ . Hence by induction hypothesis  $\Gamma \vDash_e e'_1: \text{int}(r_i, M)$  and  $\Gamma \vDash_e e'_2: \text{int}(r_i, M)$ . Therefore by the rule  $((e_1 \text{ i}_{op} e_2)^t)$ ,  $\Gamma \vDash_e e'_1 \text{ i}_{op} e'_2: \text{int}(r_i, M) = \tau$ , as required.
- The case  $e = \text{modify} - w(e, d)$ : in this case, by the rule  $(\text{modify} - w^s)$ ,  $e' = \text{modify} - w(e', d)$  and  $e \rightsquigarrow_{(m,i)}^e e'$ . By the rule  $(\text{modify} - w^t)$ , it is true that  $\Gamma \vDash_e e: \text{ptr}^m \tau$ . Hence by induction hypothesis  $\Gamma \vDash_e e': \text{ptr}^m \tau$ . Therefore  $\Gamma \vDash_e \text{modify} - w(e', d): \tau$ , by the rule  $(\text{modify} - w^t)$  as required for this case.
- The case  $e = \text{cast} < \text{int}(r_j, M_j) \rightarrow \text{int}(r_i, M_i) > e$ : in this case, by the rule  $(\text{cast}_1^s)$ ,  $e' = \text{cast} < \text{int}(r_j, M_j) \rightarrow \text{int}(r_i, M_i) > e'$  and  $e \rightsquigarrow_{(m,i)}^e e'$ . By the rule  $(\text{cast}_1^t)$ , it is true that  $\Gamma \vDash_e e: \text{int}(r_j, M_j)$ . Hence by induction

hypothesis  $\Gamma \models_e e' : \text{int}(r_j, M_j)$ .  
Therefore  $\Gamma \models_e \text{cast} < \text{int}(r_j, M_j) \rightarrow \text{int}(r_i, M_i) > e' : \text{int}(r_j, M_j)$ , by the rule ( $\text{cast}_1^t$ ). This completes the proof for this case.

Corollary 2 results directly from Lemma 3.

**Corollary 2.** Suppose that  $e \rightsquigarrow_{(m,i)}^e e'$ . Then  $\text{width} - f(e) = \text{width} - f(e')$ .

$S = \text{compute } S' \text{ at } n$  and  $S \rightsquigarrow_{(m,i)}^S S'$ . Also since  $\Gamma \models_s \text{compute } S \text{ at } n : \text{WT}$ , it is true that  $\Gamma \models_s S : \text{WT}$ , by the rule ( $\text{compute}^t$ ). Now by induction hypothesis, we conclude  $\Gamma \models_s S' : \text{WT}$ . Hence by ( $\text{compute}^t$ ),  $\Gamma \models_s \text{compute } S' \text{ at } n : \text{WT}$ . This completes the proof for this case.

- The case  $S = S_1; S_2$ : in this case, by the rule ( $\text{seq}^s$ ),  $S' = S'_1; S'_2$ ,  $S_1 \rightsquigarrow_{(m,i)}^S S'_1$ , and  $S_2 \rightsquigarrow_{(m,i)}^S S'_2$ . Also since  $\Gamma \models_s S_1; S_2 : \text{WT}$ , it

$$\begin{array}{c}
 \frac{}{x \rightsquigarrow_{(m,i)}^l x. (r_i, m) (x^s)} \quad \frac{l \rightsquigarrow_{(m,i)}^l l'}{l. y \rightsquigarrow_{(m,i)}^l l'. y} (l^s) \quad \frac{e \rightsquigarrow_{(m,i)}^e e'}{* e \rightsquigarrow_{(m,i)}^l * e'} (* e^s) \\
 \\
 \frac{}{\text{skip} \rightsquigarrow_{(m,i)}^S \text{skip} (x^s)} \quad \frac{l \rightsquigarrow_{(m,i)}^l l' \quad e \rightsquigarrow_{(m,i)}^e e'}{l := e \rightsquigarrow_{(m,i)}^S l' := e'} (:=^s) \\
 \\
 \frac{S \rightsquigarrow_{(m,i)}^S S'}{\text{compute } S \text{ at } n \rightsquigarrow_{(m,i)}^S \text{compute } S' \text{ at } n} (\text{compute}^s) \\
 \\
 \frac{S_1 \rightsquigarrow_{(m,i)}^S S'_1 \quad S_2 \rightsquigarrow_{(m,i)}^S S'_2}{S_1; S_2 \rightsquigarrow_{(m,i)}^S S'_1; S'_2} (\text{seq}^s) \\
 \\
 \frac{e \rightsquigarrow_{(m,i)}^e e' \quad S_t \rightsquigarrow_{(m,i)}^S S'_t \quad S_f \rightsquigarrow_{(m,i)}^S S'_f}{\text{if } e \text{ then } S_t \text{ else } S_f \rightsquigarrow_{(m,i)}^S \text{if } e' \text{ then } S'_t \text{ else } S'_f} (\text{if}^s) \\
 \\
 \frac{e \rightsquigarrow_{(m,i)}^e e' \quad S_t \rightsquigarrow_{(m,i)}^S S'_t}{\text{while } e \text{ do } S_t \rightsquigarrow_{(m,i)}^S \text{while } e' \text{ do } S'_t} (\text{while}^s) \\
 \\
 \frac{d \rightsquigarrow_{(m,i)}^e d' \quad S \rightsquigarrow_{(m,i)}^S S'}{dS \rightsquigarrow_{(m,i)}^p d'S'} (\text{prog}^m)
 \end{array}$$

**Fig. 8.** Left expression, statement, and program slicing over machines.

**Theorem 1.** Suppose that  $S \rightsquigarrow_{(m,i)}^S S'$  and  $\Gamma \models_s S : \text{WT}$ . Then  $\Gamma \models_s S' : \text{WT}$ .

**Proof.** The proof is by structure induction on structure of statements,  $S$ . some cases are shown below.

- The case  $S = l := e$ : in this case, by the rule ( $:=^s$ ),  $S' = l' := e'$ . Moreover  $l \rightsquigarrow_{(m,i)}^l l'$  and  $e \rightsquigarrow_{(m,i)}^e e'$ . Also since  $\Gamma \models_s l := e : \text{WT}$ , it is true that  $\Gamma \models_e e : \tau$  and  $\Gamma \models_1 l : \tau$  for some  $\tau$ , by the rule ( $:=^t$ ). Now by Lemmas 2 and 3, we conclude  $\Gamma \models_e e' : \tau$  and  $\Gamma \models_1 l' : \tau$ . Hence by ( $:=^t$ ),  $\Gamma \models_s l' := e' : \text{WT}$ , as required.
- The case  $S = \text{compute } S \text{ at } n$ : in this case, by the rule ( $\text{compute}^s$ ),

is true that  $\Gamma \models_s S_1 : \text{WT}$  and  $\Gamma \models_s S_2 : \text{WT}$ , by the rule ( $\text{seq}^t$ ). Now by induction hypothesis, we conclude  $\Gamma \models_s S'_1 : \text{WT}$  and  $\Gamma \models_s S'_2 : \text{WT}$ . Hence by ( $\text{seq}^t$ ),  $\Gamma \models_s S'_1; S'_2 : \text{WT}$  which completes the proof for this case.

Using Theorem 1 and Corollary 1, it is straightforward to conclude Corollary 3.

**Corollary 3.** (Soundness of program slicing) Suppose that  $dS \rightsquigarrow_{(m,i)}^p d'S'$  and  $\Gamma \models_p dS : \text{WT}$ . Then  $\Gamma \models_p d'S' : \text{WT}$ .

**Remark 3.** The type system of Section 2 can be realized as static semantics of the language  $\mathcal{DLang}$ . The proofs of Lemma 2 and 3 appear to rely on each other. This is absolutely true as expressions of a program are finite. The source of this sort of

recursion is the syntactic structures of expressions and left expressions (Figure 2).

#### 4. Related Work

Program slicing [23, 3] is a technique that enables a method to focus on certain part of a program. At a specific program point and with reference to a group of certain variables, a slice is an executable collection of program statements that maintain the original program behavior. Program slicing has many applications like parallelization [19], debugging [22], program comprehension [14], testing [16], downsizing, and restructuring. Statements deletions are the bases of the original concept [21] of program slice. However, there are many variants of this notion such as quasi static slicing [1, 17], dynamic slicing [22], conditioned slicing [18], and simultaneous dynamic slicing [7]. Other concepts [8] of slicing are based on generic notions of transformation such as simple statement deletion.

Typically, a slice is built on slicing criterion concept which is a pair  $\langle p, V \rangle$  of a program point and  $V$  is a collection of variables. Hence at a program point  $p$  and with reference to  $V$ , a slice that is based on  $\langle p, V \rangle$  is an executable collection of program statements that maintain the original program behavior. The maintainability here means that values of variables of  $V$  are the same for the slice and the original program at the program point  $p$ . The concept of static slicing referees to maintaining the behavior of the original program on any input. However other forms of slicing maintain the behavior for a subset of program inputs.

Quasi static slicing [1, 17] is a hybrid technique for slicing that associates static and dynamic slicing [7]. Such hybrid techniques are required when analyzing programs that have fixed input variables and varying input values. Therefore on a set of potential program inputs, a quasi slice keeps the program behavior w.r.t. slicing variables. Potential value combinations assumed by unconstrained input variables specify the set of potential program inputs. Interestingly, the quasi static slice amounts to a static slice when all variables are unconstrained.

An alternative slicing approach is dynamic slicing [12, 11]. In this technique a dynamic analysis is used to find statements affected by a certain set of variables, on a specific anomalous execution path. This approach results in a considerable reduction in the size of the slice, and hence facilities bugs allocation. Moreover, dynamic slicing treats pointer variables and arrays in a practical way (in terms of run-time). Rather than treating every use (definition) of an array element as a use (definition) of the full array [2], dynamic slicing separately treats every

array element. Equivalently, all along the execution of a program, dynamic slicing recognizes objects referenced by pointer variables. Interestingly, the quasi static slice amounts to a dynamic slice if all variable inputs are fixed.

A general version of slicing based on statement deletion is conditioned slicing [18]. This approach uses a slicing pattern for a collection of program executions to represent the original program behavior using only a collection of program statements. On the input, the first order logic is hence used to describe initial states of the program that specify these program executions.

Simultaneous dynamic program slicing [7, 23] constructs slices with reference to a collection of program executions. This approach is an extension of dynamic slicing in the form of a simultaneous application of dynamic slicing to a group of test cases, instead of only one case. However, on a group of test cases, a simultaneous program slice does not amount to applying dynamic slicing on the concerned test cases. Moreover, this multi-application of dynamic slicing is unsound in the sense that the simultaneous validity is not maintained on all the inputs. Simultaneous dynamic slicing is typically achieved iteratively, beginning with a group of statements. Hence simultaneous dynamic slices are built incrementally, via computations in each iteration.

#### Acknowledgements:

Foundation item: Al Imam University (IMSIU) Project (No.: 330917). Author is grateful to Al Imam University (IMSIU), KSA for financial support to carry out this work.

#### References

- [1] Hakam W. Alomari, Michael L. Collard, and Jonathan I. Maletic. A very efficient and scalable forward static slicing approach. In *WCRE*, pages 425–434. IEEE Computer Society, 2012.
- [2] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The santé tool: Value analysis, program slicing and test generation for c program debugging. In Martin Gogolla and Burkhart Wolff, editors, *TAP*, volume 6706 of *Lecture Notes in Computer Science*, pages 78–83. Springer, 2011.
- [3] Jeremy Condit and George C. Necula. Data slicing: Separating the heap into independent regions. In Rastislav Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2005.



- [4] Mohamed A. El-Zawawy. Probabilistic pointer analysis for multithreaded programs. *Science Asia*, 37(4):344–354, December 2011.
- [5] Mohamed A. El-Zawawy. Detection of probabilistic dangling references in multi-core programs using proof-supported tools. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, Carmelo Mario Torre, Hong-Quang Nguyen, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi, editors, *ICCSA (5)*, volume 7975 of *Lecture Notes in Computer Science*, pages 516–530. Springer, 2013.
- [6] Mohamed A. El-Zawawy. Frequent statement and de-reference elimination for distributed programs. In Beniamino Murgante, Sanjay Misra, Maurizio Carlini, Carmelo Mario Torre, Hong-Quang Nguyen, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi, editors, *ICCSA (3)*, volume 7973 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2013.
- [7] Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Autom. Softw. Eng.*, 2(1):33–53, 1995.
- [8] Paritosh Jain and Nitish Garg. A novel approach for slicing of object oriented programs. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–4, 2013.
- [9] Romain Jobredeaux, Heber Herencia-Zapana, Natasha A. Neogi, and Eric Feron. Developing proof carrying code to formally assure termination in fault tolerant distributed controls systems. In *CDC*, pages 1816–1821. IEEE, 2012.
- [10] Amir Kamil and Katherine A. Yelick. Hierarchical pointer analysis for distributed programs. In Hanne Riis Nielson and Gilberto Fil'e, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2007.
- [11] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.
- [12] Vijay Nagarajan, Dennis Jeffrey, Rajiv Gupta, and Neelam Gupta. A system for debugging via online tracing and dynamic slicing. *Softw., Pract. Exper.*, 42(8):995–1014, 2012.
- [13] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. 1 edition (February 1, 2002).
- [14] Juergen Rilling and Sudhir P. Mudur. 3d visualization techniques to support slicing based program comprehension. *Computers & Graphics*, 29(3):311–329, 2005.
- [15] A. Udaya Shankar. *Distributed Programming - Theory and Practice*. Springer, 2013.
- [16] Vivekananda M. Vedula, Jacob A. Abraham, Jayanta Bhadra, and Raghuram S. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *J. Electronic Testing*, 19(2):149–160, 2003.
- [17] G. A. Venkatesh. The semantic approach to program slicing. In David S. Wise, editor, *PLDI*, pages 107–119. ACM, 1991.
- [18] Gustavo Villavicencio. Formal program reversing by conditioned slicing. In *CSMR*, pages 368–378. IEEE Computer Society, 2003.
- [19] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin fook Ngai, and Jesse Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In Michael Gschwind, Alexandru Nicolau, Valentina Salapura, and Jos'e E. Moreira, editors, *ICS*, pages 158–168. ACM, 2009.
- [20] Mark Weiser. Program slicing. In Seymour Jeffrey and Leon G. Stucki, editors, *ICSE*, pages 439–449. IEEE Computer Society, 1981.
- [21] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
- [22] Franz Wotawa. On the use of constraints in dynamic slicing for program debugging. In *ICST Workshops*, pages 624–633. IEEE Computer Society, 2011.
- [23] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.