

Type Systems Based Data Race Detector

Mohamed A. El-Zawawy^{1,2}, & Hamada A. Nayel³

¹ College of Computer and Information Sciences, Al-Imam M. I.-S. I. University, Riyadh, Kingdom of Saudi Arabia

² Department of Mathematics, Faculty of Science, Cairo University, Giza, Egypt

³ Department of Computer Science, Faculty of Computers & Informatics, Benha University, Egypt

Correspondence: Mohamed A. El-Zawawy, College of Computer and Information Sciences, Al-Imam M. I.-S. I. University, Riyadh, Kingdom of Saudi Arabia; Department of Mathematics, Faculty of Science, Cairo University, Giza, Egypt. Tel: 20-114-746-3448. E-mail: maelzawawy@cu.edu.eg

Received: March 20, 2012 Accepted: April 5, 2012 Online Published: June 1, 2012

doi:10.5539/cis.v5n4p53

URL: <http://dx.doi.org/10.5539/cis.v5n4p53>

Abstract

Multi-threading is a methodology that has been extremely used. Modern software depends essentially on multi-threading. Operating systems, famous examples, are based on multi-threading; a user can write his document, play an audio file, and downloading a file from internet at the same time. Each of these tasks called a thread. A common problem occurs when implementing multi-threaded programs is a data-race. Data race occurs when two threads try to access a shared variable at the same time without a proper synchronization. A detector is software that determines if the program contains a data-race problem or not. In this paper, we develop a detector that has the form of a type system. We present a type system which discovers the data-race problems. We also prove the soundness of our type system.

Keywords: multi-threaded programs, type systems, data-race, semantics of programming languages

1. Introduction

Developing and debugging software that depends on multi-threading is a tricky mission because of ingrained concurrency and indeterminism. There are many bugs occur according to these properties. Detecting and preventing these bugs are important areas of research. Bugs have several forms. The most extensively studied one is data-race: two concurrent threads accessing the same shared variable without proper synchronization. Data-race detector is a tool that determines whether a program is a data-race free or not. Two approaches are followed when developing detectors: static approach, and dynamic approach. Static detectors determine whether a program produce a data-race regardless of inputs of the program. Apart from static detectors, dynamic detectors determine whether a program produce a data-race of a given inputs at execution of the program.

The advantages of static detectors are the consideration of different execution paths (more elaborate), and the soundness of detector, i.e. proving the bug-freeness of programs. Examples of static detectors (Kahlon, Sinha, Kruus, & Zhang, 2009; Kahlon, Yang, Sankaranarayanan, & Gupta, 2007; Naik, Aiken, & Whaley, 2006; Voung, Jhala, & Lerner, 2007). On the other hand, dynamic detectors like (Savage, Burrows, Nelson, Sobalvarro, & Anderson, 1997; Wang, Kelly, Kudlur, Lafortune, & Mahlke, 2008; Yu, Rodeheffer, & Chen, 2005) track program execution and report a data-race problem if the program follow a certain concurrency order. These tools produce relevant result, according to order of execution or program inputs, and can not cover all execution paths; so are not sound.

Type systems can infer and gather information about programs as well as achieving program analysis. The merits of using type systems are attesting and rationalization of properties of programs directed by their phrase structures. Type systems are actually sufficient frameworks for describing data flow analysis. A general method for producing such description was presented (Laud, Uustalu, & Vene, 2006). Type systems are used as a framework for analyzing multi-threaded programs as well as imperative programs. In (El-Zawawy, 2011), type systems were used as a framework for pointer analysis for multi-threaded programs. In (El-Zawawy & Nayel 2011) type systems were used as a framework for eliminating redundancies in multi-threaded programs.

In this paper we present a static detector. We introduce a type system that detects data-race problem for

multi-threaded programs of a simple language *m-while*. We also prove the soundness the proposed type system.

The rest of this paper is organized as follows. We outline the related work in Section 2. Section 3 presents the language, a motivation example, and an operational semantics for the language. Read type system and the proof of its soundness are introduced in Section 4. In Section 5, we introduce safety type system and a proof for its soundness. Future works are outlined in Section 6.

2. Related Work

Multi-threading is a promising area of research. The most efficient challenging areas are compilation and program analysis (El-Zawawy, 2012a; Knoop & Steffen, 1999; Midkiff & Padua, 1990). The field of program analysis aims at collecting information about programs (Nielson, Nielson, & Hankin, 1999). Analyzing may concentrate on whole program, or focuses on each program point. There are many aspects of analyzing multi-threaded programs: pointer analysis (El-Zawawy, 2012a; El-Zawawy, 2011; El-Zawawy, 2011a; El-Zawawy, 2011b; Rugina & Rinard, 2003), optimization uses (El-Zawawy & Nayel, 2011; Knoop & Steffen, 1999; Knoop, Steffen, & Vollmer, 1996; Lee, Midkiff, & Padua, 1998; Lee, Padua, & Midkiff, 1999), data race detection (Cheng, Feng, Leiserson, Randall, & Stark, 1998; Rugina & Rinard, 2005), and deadlock detection (Blieberger, Burgstaller, & Scholz, 2000; Koskinen & Herlihy, 2008; Wang, Kelly, Kudlur, Lafortune, & Mahlke, 2008).

A data race occurs very often in multi-threaded programs. It occurs when two threads try to access the same location without proper synchronization, and one of them is write (Henzinger, Jhala, & Majumdar, 2004). The data race always causes program bugs. The output of program depends on scheduling of accessing memory. Detecting data race is a promising area of research, it has been studied extensively (Cheng, Feng, Leiserson, Randall, & Stark, 1998; Flanagan et al., 2002; Kahlon, Yang, Sankaranarayanan, & Gupta, 2007; Permandla, Roberson, & Boyapati, 2007; Voung, Jhala, & Lerner, 2007). The first methodology to detect data races is the static race detection (Flanagan et al., 2002). Detectors, in this strategy, determine whether a program will ever produce a data race when run on all possible inputs. The second methodology is dynamic race detection, where potential races are detected at runtime by executing the program on a given input (Savage, Burrows, Nelson, Sobalvarro, & Anderson, 1997; Wang, Kelly, Kudlur, Lafortune, & Mahlke, 2008). In many detectors, data race and deadlock bugs are bundled together. Some static detectors, like Warlock (Sterling, January 1993), depends on the annotations formed by programmers to detect data races and deadlock problems. Using Theorem provers to detect many bugs including data race is the idea of extended static checker for Java (Flanagan et al., 2002; Leino, Saxe, & Stata, 1999). Some dynamic detectors are developed in the scientific parallel programming community (Cheng, Feng, Leiserson, Randall, & Stark, 1998; Dinning & Schonberg, 1991). Others detectors detects data race in Java-like programs (Choi et al., 2002; Praun & Gross, 2001). Eraser (Savage, Burrows, Nelson, Sobalvarro, & Anderson, 1997), as a dynamic detector, monitors programs during execution and looks for data race bugs. In general, dynamic detectors have the advantage that they can check un-annotated programs.

Type systems are known to be a good framework for analyzing programs (Laud, Uustalu, & Vene, 2006; El-Zawawy & Daoud, 2012). Type systems have been extensively used in pointer analysis for both imperative and multi-threaded programs (El-Zawawy, 2011; El-Zawawy, 2011b). Type systems have been used to detect the memory safety of multi-threaded programs (El-Zawawy, 2011; El-Zawawy, 2011b; El-Zawawy, 2011a). Type systems are used in code optimization. Partial redundancy elimination was performed via type systems for imperative programs (Saabas & Uustalu, 2009), and for multi-threaded programs (El-Zawawy & Nayel, 2011). Type systems also are used to prevent data race and dead lock in a specific Java language (Permandla, Roberson, & Boyapati, 2007).

Mathematical domains and maps between domains can be used to mathematically represent programs and data structures. This representation is called denotation semantics of programs (El-Zawawy & Jung, 2006; El-Zawawy, 2007). One of our directions for future research is to translate concepts of race detection to the side of denotation semantics. Doing so provides a good tool to mathematically study in deep race detection. Then obtained results can be translated back to the side of programs and data structures.

3. Motivation

In this section, we will present a simple example that demonstrates our motivations for this research. Firstly we define a simple language, called *m-while*, that supports the multi-threading concepts. In this language, statements $s \in \mathbf{Stm}$, arithmetic expression $a \in \mathbf{AExp}$, and $b \in \mathbf{BExp}$ are defined over a set of program variables $x \in \mathbf{Var}$ in the following way:

$$l ::= x \mid n$$

$$\begin{aligned}
a & ::= l \mid l_0 + l_1 \mid l_0 * l_1 \mid \dots \\
b & ::= l_0 = l_1 \mid l_0 < l_1 \mid \dots \\
s & ::= x := a \mid \mathbf{skip} \mid s_0; s_1 \mid \mathbf{if } b \mathbf{ then } s_t \mathbf{ else } s_f \mid \mathbf{while } b \mathbf{ do } s_t \mid \mathbf{fork } \{s_1\}, \{s_2\}, \dots, \{s_n\} \mathbf{endfork}
\end{aligned}$$

The following segment of a program motivates our work:

```

a := 4;
b := 6;
x := a + b;
fork
{ y := a + b ; z := b + 5 ; },
{ a := a + 5 ; w := a + b ; },
{ x := x + 4 ; },
endfork;

```

This code shows that the variable a accessed by two threads; in the first thread with read operation ($y := a + b$;) and the other thread with write operation ($a := a + 5$;) . In this case the data-race problem occurs.

3.1 Operational Semantics

The semantics is given in terms of states. A state is a pair, $\sigma = (\mathbf{R}, \mathbf{M})$, where \mathbf{R} is a set of variables accessed by read operations, and \mathbf{M} is a store. A store is a mapping from variables to integers $\mathbf{M} \in \mathbf{Store} =_{\text{df}} \mathbf{Var} \rightarrow \mathbf{Z}$. The Boolean and arithmetical expressions are interpreted as truth values and integers according to stores by the semantic function $[-] \in \mathbf{AExp} \cup \mathbf{BExp} \rightarrow \mathbf{Stores} \rightarrow \mathbf{Z}$. For arithmetic expression $a \in \mathbf{AExp}$, $[a] \sigma$ denotes the arithmetic evaluation of a in the state σ . For Boolean expression $b \in \mathbf{BExp}$, $[b] \sigma$ denotes the truth value of b in the state σ . We write $\sigma \models b$ to mean that $[b] \sigma = tt$. We note that $\mathbf{FV}(a)$ is the set of free variables of expression a . The operational semantics of m -while are given in Figure 1. The rules show that the assignment statement actually changes the state; the free variables of the expression that has been evaluated are added to read variables, and the store assigns new value for the variable being assigned a new value. The rules for imperative statements *skip*, *sequence*, *if*, and *while* changing the pre-state as usual for their classical meaning. The last rule (*fork* statement), that characterizes the multi-threading, describes that the fork statement changes a state using the states of each thread. One can see that our specific description of states helps in proving the soundness of our proposed type systems.

$$\begin{array}{c}
\frac{}{x := a : (R, M) \rightarrow (R \cup \mathbf{FV}(a), M[x \mapsto [a]M])} ::=_{os} \\
\frac{}{\mathbf{skip} : (R, M) \rightarrow (R, M)} \mathbf{skip}_{os} \\
\frac{s_0 : (R, M) \rightarrow (R'', M'') \quad s_1 : (R'', M'') \rightarrow (R', M')}{s_0; s_1 : (R, M) \rightarrow (R', M')} \mathbf{seq}_{os} \\
\frac{\sigma \models b \quad s_t : (R, M) \rightarrow (R', M')}{\mathbf{if } b \mathbf{ then } s_t \mathbf{ else } s_f : (R, M) \rightarrow (R', M')} \mathbf{iftrue}_{os} \\
\frac{\sigma \not\models b \quad s_f : (R, M) \rightarrow (R', M')}{\mathbf{if } b \mathbf{ then } s_t \mathbf{ else } s_f : (R, M) \rightarrow (R', M')} \mathbf{iffalse}_{os} \\
\frac{\sigma \models b \quad s_t : (R, M) \rightarrow (R'', M'') \quad \mathbf{while } b \mathbf{ do } s_t : (R'', M'') \rightarrow (R', M')}{\mathbf{while } b \mathbf{ do } s_t : (R, M) \rightarrow (R', M')} \mathbf{whilet}_{os} \\
\frac{\sigma \not\models b}{\mathbf{while } b \mathbf{ do } s_t : (R, M) \rightarrow (R, M)} \mathbf{whilef}_{os} \\
\frac{s_{\theta(i)} : (R, M_i) \rightarrow (R_i, M_{i+1}) \quad \forall i \in \{1, 2, \dots, n\}}{\mathbf{fork } \{s_1\}, \{s_2\}, \{s_3\} \mathbf{endfork} : (R, M_1) \rightarrow (\bigcup R_i, M_{n+1})} \mathbf{fork}_{os} \\
\text{where } \theta \text{ is a permutation on } \{1, 2, \dots, n\}
\end{array}$$

Figure 1. The operational semantics of m -while language

4. Read Type System

In this section, we introduce *read type system*. At each program point, the read type system determines the variables that have been accessed with a read operation. This type system acts as a flag to discover the overlapping of read type system and the concurrent modified set. A program point has type $r \subseteq \mathbf{Var}$, if all variables in r are accessed by read operations (from beginning of the program to this point). The sub-typing is the set inclusion, i.e. $r \leq r_1$ if $r \subseteq r_1$. The rules of read type system are given in Figure 2. The first rule ($:=_r$) adds the free variables of the computed expression to the pre-type. The rules $skip_r$, $sequence_r$, if_r , and $while_r$ affect pre-type as expected considering their classical meaning. The rule ($conseq_r$) is important for weakening the pre-type and strengthen the post-type. For the rule ($fork_r$) which characterizes the multi-threading, the post-type of *fork* statement is the union of all post-types of different threads.

$$\begin{array}{c}
 \frac{}{x := a : r \rightarrow r \cup FV(a)} :=_r \\
 \frac{}{skip : r \rightarrow r} skip_r \quad \frac{s_0 : r \rightarrow r'' \quad s_1 : r'' \rightarrow r'}{s_0 ; s_1 : r \rightarrow r'} seq_r \\
 \frac{s_i : r \rightarrow r' \quad s_j : r \rightarrow r'}{if \ b \ then \ s_i \ else \ s_j : r \rightarrow r'} if_r \quad \frac{s_i : r \rightarrow r'}{while \ b \ do \ s_i : r \rightarrow r'} while_r \\
 \frac{r \leq r_1 \quad s : r_1 \rightarrow r_2 \quad r_2 \leq r'}{s : r \rightarrow r'} conseq_r \\
 \frac{s_{\theta(i)} : r \rightarrow r_i \quad \forall i \in \{1, 2, \dots, n\}}{fork \ \{s_1\}, \{s_2\}, \dots, \{s_n\} \ endfork : r \rightarrow \bigcup_i r_i} fork_r \\
 \text{where } \theta \text{ is a permutation on } \{1, 2, \dots, n\}
 \end{array}$$

Figure 2. The rules of read type system of *m-while* language

4.1 Soundness of Read Type System

In this section, we prove the soundness of read type system. Firstly, the following definition is introduced:-

Definition 1.

For a state $\sigma = (\mathbf{R}, \mathbf{M})$, we say that σ entails r , where $r \subseteq \mathbf{Var}$ is a read type set, if $r \subseteq \mathbf{R}$. This is written as follows $\sigma \vdash r \iff r \subseteq \mathbf{R}$.

The soundness of the concerned type systems is introduced in the following theorem.

Theorem 1.

If $s : \sigma \rightarrow \sigma'$ and $s : r \rightarrow r'$, then $\sigma \vdash r \implies \sigma' \vdash r'$.

Proof:

The proof is by structural induction on rules, we demonstrate some cases:-

- Case $:=_r$

We use the operational rule $:=_{os}$. Let $\sigma \vdash r \implies r \subseteq \mathbf{R}$. Hence we have

$$r' = r \cup FV(a) \subseteq \mathbf{R} \cup FV(a) \subseteq \mathbf{R}' \implies \sigma' \vdash r'$$

- Case $fork_r$

We use the operational rule $fork_{os}$. From premises the following are satisfied:-

$s_{\theta(i)} : r \rightarrow r_i$, and $s_{\theta(i)} : \sigma_i = (\mathbf{R}, \mathbf{M}_i) \rightarrow \sigma_{i+1} = (\mathbf{R}_i, \mathbf{M}_{i+1})$. i.e.

$\sigma_i \vdash r \implies \sigma_{i+1} \vdash r_i$ or equivalently $r \subseteq \mathbf{R} \implies r_i \subseteq \mathbf{R}_i$. It is enough to prove that:-

$$\sigma \vdash r \implies \sigma_{n+1} \vdash \bigcup_i r_i$$

But, $\sigma \vdash r \implies r \subseteq \mathbf{R}$

$$\implies r_i \subseteq \mathbf{R}_i$$

$$\implies \bigcup_i r_i \subseteq \bigcup_i \mathbf{R}_i$$

$$\implies \sigma_{n+1} \vdash \bigcup r_i.$$

5. Safety Type System

In this section, we introduce the *safety type system*. Each program point has a type $d \in \{true, false\}$, where *true* means that the program is safe at this point, and *false* means that the program is unsafe at this point. In the following $C(s)$ denotes the concurrent modified set of the statement s introduced in (El-Zawawy & Nayel, 2011), and $p_1 \wedge p_2$ denotes the logical conjunction of Boolean variables p_1 and p_2 . The following definition is needed:-

Definition 2.

The truth value of a set A is defined as follow:-

$$\mathbf{Tr}(A) = true \quad \text{if} \quad A = \varnothing \quad \text{and} \quad false \quad \text{if} \quad A \neq \varnothing$$

For any two sets \mathbf{A} and \mathbf{B} the following properties are satisfied

$$\mathbf{Tr}(\mathbf{A}) \wedge \mathbf{Tr}(\mathbf{B}) = \mathbf{Tr}(\mathbf{A} \cup \mathbf{B})$$

and

$$\mathbf{Tr}(\mathbf{A}) \vee \mathbf{Tr}(\mathbf{B}) = \mathbf{Tr}(\mathbf{A} \cap \mathbf{B})$$

The rules of safety type system are defined in Figure 3. The first rule $:=$ checks the overlapping of concurrent modified set and the read set. All other rules are straightforward. In general we conclude that, the program is safe if each program point has a type true otherwise the program is unsafe.

$$\begin{array}{c} \frac{x := a : r \rightarrow r'}{x := a : d \multimap d \wedge \mathbf{Tr}(r' \cap C(x := a))} := \quad \frac{}{\text{skip} : d \multimap d} \text{skip} \\ \frac{s_0 : d \multimap d'' \quad s_1 : d'' \multimap d'}{s_0 ; s_1 : d \multimap d'} \text{seq} \\ \frac{s_i : d \multimap d' \quad s_f : d \multimap d'}{\text{if } b \text{ then } s_i \text{ else } s_f : d \multimap d'} \text{if} \quad \frac{s_i : d \multimap d'}{\text{while } b \text{ do } s_i : d \multimap d'} \text{while} \\ \frac{s_{\theta(i)} : d_i \multimap d_{i+1} \quad \forall i \in \{1, 2, \dots, n\}}{\text{fork } \{s_1\}, \{s_2\}, \dots, \{s_n\} \text{ endfork} : d_1 \multimap d_{n+1}} \text{fork} \end{array}$$

Figure 3. The rules of safety type system of m -while language

5.1 Soundness of Safety Type System

Firstly we define the entailment of a type d in state $\sigma = (\mathbf{R}, \mathbf{M})$ with respect to set \mathbf{A} as follow:-

$$\sigma \models_{\mathbf{A}} d \iff d = \mathbf{Tr}(\mathbf{R} \cap \mathbf{A})$$

The following theorem states and proves the soundness of the safety type system.

Theorem 2

Let $s : \sigma \rightarrow \sigma'$, and $s : d \multimap d'$. Then $\sigma \models_{C(s)} d \implies \sigma' \models_{C(s)} d'$

Proof:

The proof is by structural induction on rules, we present some cases:-

- Case $:=$

Suppose $x := a : \sigma \rightarrow \sigma'$, and $x := a : r \rightarrow r'$, where $\sigma = (\mathbf{R}, \mathbf{M})$, $\sigma' = (\mathbf{R}', \mathbf{M}')$, $r' = r \cup \mathbf{FV}(a)$, and $\mathbf{R}' = \mathbf{R} \cup \mathbf{FV}(a)$.

From premises, $r \subseteq \mathbf{R} \implies r' \subseteq \mathbf{R}'$.

Let $\sigma \models_{C(x:=a)} d$. Then $d = \mathbf{Tr}(\mathbf{R} \cap C(x := a))$

Now $d' = d \wedge \mathbf{Tr}(r' \cap C(x := a))$

$$\begin{aligned} &= \mathbf{Tr}(\mathbf{R} \cap C(x := a)) \wedge \mathbf{Tr}(r' \cap C(x := a)) \\ &= \mathbf{Tr}((\mathbf{R} \cap C(x := a)) \cup (r' \cap C(x := a))) \\ &= \mathbf{Tr}((\mathbf{R} \cup r') \cap C(x := a)) \end{aligned}$$

$$\begin{aligned}
&= \text{Tr}((\mathbf{R} \cup r \cup \mathbf{FV}(a)) \cap \mathbf{C}(x := a)) \\
&= \text{Tr}((\mathbf{R} \cup \mathbf{FV}(a)) \cap \mathbf{C}(x := a)) \\
&= \text{Tr}(\mathbf{R}' \cap \mathbf{C}(x := a)) \\
&\implies \sigma' \models_{C(x:=a)} d'
\end{aligned}$$

i.e., $\sigma \models_{C(x:=a)} d \implies \sigma' \models_{C(x:=a)} d'$ which completes the proof.

- *Case seq*

We use the operational semantic rule seq_{os} . From premises we have:-

$$\sigma \models_{C(s_0)} d \implies \sigma'' \models_{C(s_0)} d'', \text{ and } \sigma'' \models_{C(s_1)} d'' \implies \sigma' \models_{C(s_1)} d'$$

From the definition of concurrent modified function given by (El-Zawawy & Nayel, 2011), we can conclude that for a sequence of statements $s_0; s_1$ the following is satisfied:-

$$\mathbf{C}(s_0) = \mathbf{C}(s_1) = \mathbf{C}(s_0; s_1)$$

$$\text{i.e., } \sigma \models_{C(s_0; s_1)} d \implies \sigma'' \models_{C(s_0; s_1)} d'',$$

and

$$\sigma'' \models_{C(s_0; s_1)} d'' \implies \sigma' \models_{C(s_0; s_1)} d'$$

then we can conclude that $\sigma \models_{C(s_0; s_1)} d \implies \sigma' \models_{C(s_0; s_1)} d'$.

- *Case fork*

To prove this rule we can consider the fork statement as the following sequence of statements **fork** $\{s_1\}, \{s_1\}, \dots, \{s_n\}$ **endfork** = $s_{\theta(1)}; s_{\theta(2)}; \dots; s_{\theta(n)}$. Now applying the sequence rule produces the proof.

5.2 Implementation

We implemented a detector based on our type system. This program checks any program of *m-while* language for safety. For a program of *m-while* language, the detector computes the modified sets of each program point and computes read sets for each program point. Then the intersection of these two sets is calculated.

6. Conclusion and Future Work

In this paper we present a static data race detector. We use type systems as a framework to implement this detector. Firstly, we present read type system which computes the variables accessed by read operations. Secondly, we present safe type system. This type system is based on read type system and decides if a program contains data race problems or not. The soundness of these type systems are discussed in this paper as well. For future work we plan to use type systems as a tool to solve more complicated problems (like deadlock, pointer dangling.). We expect type systems to be an amenable and trustable framework to deal with static analyses. We also plan to improve our work in many directions including extending our language to support the object-oriented concepts.

References

- Blieberger, J., Burgstaller, B., & Scholz, B. (2000). Symbolic data flow analysis for detecting deadlocks in ada tasking programs. *Proceedings of the 5th ada-europe international conference on reliable software technologies*, 225-237. UK: Springer Verlag.
- Cheng, G. I, Feng, M., Leiserson, C. E., Randall, K. H., & Stark, A. F. (1998). *Detecting data races in cilk programs that use locks*. Proceedings of the tenth annual acm symposium on parallel algorithms and architectures, 298-309. New York, NY, USA; ACM.
- Choi, J. D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., & Sridharan, M. (2002). *Efficient and precise datarace detection for multithreaded object-oriented programs*. Proceedings of the acm sigplan 2002 conference on programming language design and implementation, 285-269. New York, NY, USA: ACM.
- Dinning, A., & Schonberg, E. (1991). *Detecting access anomalies in programs with critical sections*. Proceedings of the 1991 acm/onr workshop on parallel and distributed debugging, 85-96. New York, NY, USA: ACM.
- El-Zawawy, M. A., & Daoud, N. (2012). New error-recovery techniques for faulty-calls of functions. *Computer and Information Science*, 5(3).
- El-Zawawy, M. A. (2007). *Semantic spaces in Priestley form*. PhD thesis, University of Birmingham, UK.

- El-Zawawy, M. A. (2011). *Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs*. In Beniamino Murgante, Osvaldo Gervasi, Andr es Iglesias, David Taniar, and Bernady O. Aduhan, editors, *ICCSA (5)*, volume 6786 of *Lecture Notes in Computer Science*, pp. 355-369. Springer.
- El-Zawawy, M. A. (2011a). Probabilistic pointer analysis for multithreaded programs. *ScienceAsia*, 37(4).
- El-Zawawy, M. A. (2011b). Program optimization based pointer analysis and live stack-heap analysis. *International Journal of Computer Science Issues*, 8(2).
- El-Zawawy, M. A. (2012a). Dead code elimination based pointer analysis for multithreaded programs. *Journal of the Egyptian Mathematical Society*.
- El-Zawawy, M. A., & Jung, A. (2006). Priestley duality for strong proximity lattices. *Electr. Notes Theor. Comput. Sci.*, 158, 199-217. <http://dx.doi.org/10.1016/j.entcs.2006.04.011>
- El-Zawawy, M. A., & Nayel, H. (2011). Partial redundancy elimination for multi-threaded programs. *IJCSNS International Journal of Computer Science and Network Security*, 11(10).
- Flanagnn, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., & Stata, R. (2002). *Extended static checking for java*. Proceedings of the acm sigplan 2002 conference on programming language design and implementation, 234-245, New York, NY, USA: ACM.
- Henzinger, T. A., Jhala, R., & Majumdar, R. (2004). *Race checking by context inference*. Proceedings of the acm sigplan 2004 conference on programming language design and implementation, 1-13, New York, NY, USA: ACM
- Kahlon, V., Sinha, N., Kruus, E., & Zhang, Y. (2009). *Static data race detection for concurrent programs with asynchronous calls*. Proceedings of the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering, 13-22, New York, NY, USA: ACM.
- Kahlon, V., Yang, Y., Sankaranarayanan, S., & Gupta, A. (2007). *Fast and accurate static datarace detection for concurrent programs*. Proceedings of the 19th international conference on computer aided verification, 226-239. Berlin, Heidelberg: Springer-Verlag.
- Knoop, J., & Steffen, B. (1999). *Code motion for explicitly parallel programs*. Proceeding of the seventh acm sigplan symposium on principles and practice of parallel programming, 13-24. New York, NY, USA: ACM.
- Knoop, J., Steffen, B., & Vollmer, J. (1996). *Efficient and optimal bitvector analyses for parallel programs*. *ACM Trans. Program. Lang. Syst.*, 18, 268-299. <http://dx.doi.org/10.1145/229542.229545>
- Koskinen, E., & Herlihy, M. (2008). *Dreadlocks: Efficient deadlock detection*. Proceeding of the twentieth annual symposium on parallelism in algorithms and architectures, 297-303, New York, NY, USA: ACM.
- Laud, P., Uustalu, T., & Vene, V. (2006). Type systems equivalent to data-flow analyses for imperative languages. *Theor. Comput. Sci.*, 292-310. <http://dx.doi.org/10.1016/j.tcs.2006.08.013>
- Lee, J., Midkiff, S. P., & Padua, D. A. (1998). A constant propagation algorithm for explicitly parallel programs. *Int. J. Parallel Programming*, 26, 563-589. <http://dx.doi.org/10.1023/A:1018772514882>
- Lee, J., Midkiff, S. P., & Padua, D. A. (1999). *Basic compiler algorithms for parallel programs*. Proceedings of the seventh acm sigplan symposium on principles and practice of parallel programming, 1-12, New York, NY, USA: ACM.
- Leino, K. R. M., Saxe, J. B., & Stata, R. (1999). *Checking java programs via guarded commands*. Proceedings of the workshop on object-oriented technology, 110-111, London, UK: Springer-Verlag.
- Midkiff, S., & Padua, D. (1990). *Issues in the optimization of parallel programs*. Proceedings of 1990 international conference on parallel processing, 105-113.
- Naik, M., Aiken, A., & Whaley, J. (2006). *Effective static race detection for java*. Proceedings of the 2006 acm sigplan conference on programming language design and implementation, 308-319, New York, NY, USA: ACM.
- Nielson, F., Neilson, H. R., & Hankin, C. (1999). *Principles of program analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.
- Permandla, P., Roberson, M., & Boyapati, C. (2007). *A type system for preventing data races and deadlocks in the java virtual machine language*. Proceedings of the 2007 acm sigplan/sigbed conference on languages, compilers, and tools for embedded systems, 10-20, New York, NY, USA: ACM.

- Rugina, R., & Rinard, M. C. (2003). Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25, 70-116. <http://dx.doi.org/10.1145/596980.596982>
- Rugina, R., & Rinard, M. C. (2005). Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27, 185-235. <http://dx.doi.org/10.1145/1057387.1057388>
- Saabas, A., & Uustalu, T. (2009). Proof optimization for partial redundancies elimination. *Journal of Logic and Algebraic Programming*, 78(7), 619-642. <http://dx.doi.org/10.1016/j.jlap.2009.05.002>
- Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., & Anderson, T. (1997). Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15, 391-411. <http://dx.doi.org/10.1145/265924.265927>
- Sterling, N. (1993). *Warlock: A static data race analysis tool*. Usenix winter technical conference, 97-106.
- Von, P. C., & Gross, T. R. (2001). *Object race detection*. Proceeding of the 16th acm sigplan conference on object-oriented programming, systems, languages, and applications, 70-82, New York, NY, USA: ACM.
- Young, J. W., Jhala, R., & Lerner, S. (2007). *Relay: Static race detection on millions of lines of code*. Proceedings of the 6th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering, 205-214, New York, NY, USA: ACM.
- Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., & Mahlke, S. (2008). *Gadara: Dynamic deadlock avoidance for multithreaded programs*. Proceedings of 8th usenix conference on operating system design and implementation, 281-294, Berkeley, CA, USA:USENIX Association.
- Yu, Y., Rodeheffer, T., & Chen, W. (2005). *Racetrack: Efficient detection of data race conditions via adaptive tracking*. Proceedings of the twentieth acm symposium on operating systems principles, 221-234. New York, NY, USA: ACM.