Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/090560

Probabilistic Dangling References of Imperative and Object-Oriented Multi-Core Programs

Mohamed A. El-Zawawy[1,2,*]

[1] College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia.
[2] Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt.

Received: 4 Feb. 2015, Revised: 13 May 2015, Accepted: 17 May 2015
Published online: 1 Sep. 2015

Abstract: Speculative optimizers of modern compilers are designed on techniques for probabilistic static analyses of programs. For imperative and object-oriented multi-core programs, this paper focuses on the problem of revealing probabilistic dangling references. This important problem is treated in this paper via type systems. Besides being simply structured, the type systems provide suitable frameworks for proof-carrying code applications. One class of such applications is that of mobile codes having limited resources. Using the proposed technique, each analysis case is supported by a correctness proof in the form of a type derivation. Most important concurrent constructs such as fork-join constructs, conditionally spawned cores, and parallel loops are treated in this paper. This is done for both of imperative and object-oriented parallel programs. The paper proves the soundness of presented techniques using a probabilistic operational semantics for the language models.

Keywords: Speculative Optimizers, Probabilistic Dangling References, Imperative Multi-Core Programs, Object-Oriented Multi-Core Programs, Type Systems.

1 Introduction

Multi-core (multithreading) [1] and object-oriented [2] are among main programming styles today. The use of multiple cores (threads) has many advantages; simplifying the process of structuring huge software systems, hiding the delay caused by commands waiting for resources, and boosting the performance of applications executed on multiprocessors. However the interactions between different cores complicate the compilation and analysis of multi-core programs. The object-oriented programming (OOP) paradigm has the advantage of combining other approaches like functional, relational, and imperative programming. OOP is based on concepts of method, class, and inheritance. The flexibility of these concepts facilities varying degrees of dynamic behaviour in different object-oriented languages implementations.

One of the vital and attractive attributes of multi-core programs is memory safety (mainly including dangling-references detection) [3]. A dangling reference is a reference (pointer) that does not refer to a valid object of the pertinent type. The importance that memory safety enjoys is justified by several facts including the fact that the absence of memory safety can cause the execution of programs to abort. This absence can be maliciously used to cause security breaches like in many recent cases [4]. However low-level parallel programming languages, used to write most existing parallel-software applications, scarify safety for the sake of improving performance. Violating memory safety takes several forms including memory leaks, buffer overflows, and dangling pointers. Among causes for memory safety violations are explicit allocation and deallocation, pointer arithmetic, casting, and the interactions between multiple cores (threads).

Memory safety [3] is a critical compiler analysis used to decide whether a given piece of code

* Corresponding author e-mail: maelzawawy@cu.edu.eg

© 2015 NSP
Natural Sciences Publishing Cor.

contains memory violations (basically including dangling references). A conventional memory-safety analysis deduces whether a given program (i) is definitely safe (all program execution paths are safe), (ii) is definitely not safe (all program paths contain memory violations), or (iii) maybe safe (some paths are not definitely safe). A probabilistic memory-safety [5] analysis is a program analysis that decides for a given program S and a probability ϵ whether all execution paths with probabilities greater than or equal to ϵ are definitely memory-safe. On the one hand most traditional compiler optimizations count on precise memory-safety checks, and to ensure correctness cannot optimize in the "maybe" case which is the prevalent case. But on the other hand new speculative optimizations [6] can aggressively take advantage of the prevalent "maybe" case, especially in the presence of a probabilistic memory-safety (dangling-references) analysis.

Reference analysis [7] of a program calculates for each program point a reference (points-to) relationship that captures information about the memory addresses that may be referenced by (pointed-at) by program references (pointers). A probabilistic reference analysis [6,7] statically anticipates the likelihood of every reference relationship at each program point. An absolute memory-safety analysis follows an absolute reference analysis i.e. builds on the result of an absolute reference analysis. It is also the case that probabilistic memory-safety analysis follows or builds on the result of a probabilistic reference analysis.

This paper is an extended and revised version of [8] which presents an approach for detecting dangling references in imperative multi-core programs. The current paper extends the approach of [8] to cover object-oriented multi-core programs. Section 3 includes the main extensions of the current paper over [8]. These extensions include:

1. Generalizing type concepts presented in [8] to cover object-oriented multi-core programs rather than imperative multi-core programs.
2. Extending inference rules of type system presented in [8].
3. Enriching semantics domains used in [8].
4. Generalizing soundness concepts established in [8].

The proposed techniques of this paper are probabilistic in its nature. For a given program S, probability threshold ϵ, and the result *pts* of a probabilistic reference analysis for S (like that in [7]), the proposed techniques decide wether execution paths of S with probabilities greater than or equal to ϵ are memory safe (dangling-references free) with respect to *pts*. The proposed techniques are flow-sensitive.

The algorithmic style [9,10], which relies on data-flow analysis, is typically used to present static analysis and optimization techniques of multi-core programs. Another framework for program analyses and optimizations is provided by type systems [7]. While the type-systems style works directly on the phrase structure of programs, the algorithmic style works on control-flow graphs (intermediate forms) of programs. One advantage of the type-systems approach over the algorithmic one is that the former provides communicable justifications (type derivations) for analysis results. Certified code is an example of an area where such machine-checkable justifications are required. Another advantage of type-systems style is the relative simplicity of its inference rules. The techniques presented in this paper for memory safety of multi-core programs have the form of type systems. The key to the proposed approaches is to compute a post-type starting with the trivial type as a pre-type. Then a program that has this post-type is guaranteed to be memory safe over all computational paths whose probabilities greater than or equal to a given probabilistic threshold.

Figure 1 presents a programming language that we study. The set *Var* is a finite set of program variables. The language is the simple *while* language enriched with basic commands for parallel computations; fork join, conditionally spawned cores, and parallel loops.

Motivation

Figure 2 presents a motivating example for the analysis of probabilistic memory-safety. For this program we suppose that the condition of *if* statement in line 1 is true with probability 0.8. This program has four possible execution paths;

1. The *then* statement (line 2) followed by the first core (thread) (line 5) followed by the second core (line 6).
2. The *then* statement followed by the second core followed by the first core.
3. The *else* statement (line 3) followed by the first core followed by the second core.
4. The *else* statement followed by the second core followed by the first core.

The probability of each of the first two paths is 0.4 and that of each of the last two paths is 0.1. The last two paths are not memory safe as they contain dangling pointers (de-referencing of a in line 5). However the first two paths are memory safe. The motivation of our work is to design techniques that for a program like this one and a probabilistic

© 2015 NSP
Natural Sciences Publishing Cor.

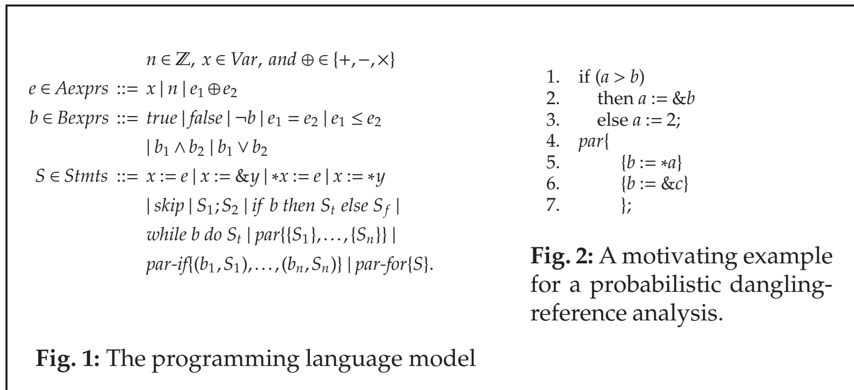

Fig. 1: The programming language model

Fig. 2: A motivating example for a probabilistic dangling-reference analysis.

threshold (for example 0.4) decides whether the program paths with probabilities greater than or equal to 0.4 are memory safe. The desired techniques are also required to associate their decision with a correctness proof.

Contributions

Contributions of this paper are the following:

1. An original type system carrying a probabilistic analysis for memory safety (mainly dangling-references detection) of imperative multi-core programs.
2. An original type system carrying a probabilistic analysis for memory safety (mainly dangling-references detection) of object-oriented multi-core programs.
3. A formal proof for the soundness of the proposed type systems with respect to a probabilistic operational semantics.

Organization

The rest of the paper is organized as follows. Sections 2 and 3 present type systems for probabilistic memory safety of imperative multi-core programs and object-oriented multi-core programs, respectively. Section 2 also presents a formal correctness proof for its type system with respect to the probabilistic operational semantics presented in the appendix of this paper. A survey of related work to memory safety (including the used of type systems in program analysis, and the analysis of multi-core programs) is presented in Section 4.

2 Probabilistic Memory Safety of Imperative Multi-Core Programs

This section presents a new technique for memory safety (including dangling-references detection) of imperative multi-core programs where a dangling reference is a reference (pointer) that does not refer to a valid object of the pertinent type. The proposed technique is both forward and static (to be used during compilation time). The technique is also probabilistic in the sense that for a given program S and a probabilistic threshold (denoted by p_{ms} in this paper) the technique decides whether all computation paths of S with probability greater or equal to p_{ms} are memory safe. This sort of information is required and intensively used in speculative optimizations that are parts of most modern compilers. Examples of such speculative approaches are speculative parallelization ([11]), speculative loop execution([12]), speculative redundancy elimination, speculative dead store elimination, speculative code scheduling, and speculative copy propagation ([13,14], and [15]).

Memory safety of programs is a forward program analysis that is typically built on the result of a reference analysis. For probabilistic memory safety, the underlying reference analysis has to be probabilistic [6,7] as well. Hence we assume that our input program S is associated with the probabilistic threshold p_{ms} and the result of a probabilistic reference analysis[1] for S. To be more precise, we assume that the underlying probabilistic reference analysis associates each program point with a reference type *pts* drawn from a set of probabilistic reference types *PTS*. A natural formalization of the set *PTS* together with a subtyping relation on its types is introduced in [7] and reviewed in Definition 1. To build the memory safety analysis on robust ground, surely the underlying reference

[1] The reference analysis results (reference information) are typically assigned to program points of S.

© 2015 NSP
Natural Sciences Publishing Cor.

analysis has to be sound with respect to a robust semantics; in our case the operational semantics in the appendix of this paper. Suppose that for a statement S, the reference analysis associates a pre-reference type pts and a post-reference type pts', i.e. $S : pts \to pts'$. The soundness has the intuition that if the execution of S from a state (γ,p) (Definition 6) of type pts ends at a state (γ,p'), then this final state has to be of type pts'.

The intuitions of terminologies used in Definition 1 are in order. A *reference type* (denoted by *pts*) is a map from program variables to sets of pairs of addresses and probabilities. The symbol $Addrs_p$ is an enrichment of the set of symbolic addresses $Addrs$ (variable addresses) with probabilities. The idea behind the condition on the elements of *Pre-PTS* is to exclude functions mapping a program variable to an address with two different probabilities. The probability that a variable x contains an address, with respect to pts, is denoted by $\sum_{pts} x$. The set of addresses having a non-zero probability to be contained in x, with respect to pts, is denoted by $A_{pts}(x)$. The notion $\gamma \models pts$ denotes that a computational state γ (Definition 6) is of type pts.

Definition 1. 1.*$Addrs = \{x' \mid x \in Var\}$ and $Addrs_p = Addrs \times [0,1]$.*
2.*$Pre\text{-}PTS = \{pts \mid pts : Var \to 2^{Addrs_p}$ s.t. $(y',p_1),(y',p_2) \in pts(x) \Longrightarrow p_1 = p_2\}$.*
3.*For $pts \in Pre\text{-}PTS$ and $x \in Var$, $\sum_{pts} x = \sum_{(z',p)\in pts(x)} p$.*
4.*For $pts \in Pre\text{-}PTS$ and $x \in Var$, $A_{pts}(x) = \{z' \mid \exists p > 0.\ (z',p) \in pts(x)\}$.*
5.*$PTS = \{pts \in Pre\text{-}PTS \mid \forall x \in Var.\ \sum_{pts} x \le 1\}$.*
6.*$pts \le pts' \overset{\text{def}}{\Longleftrightarrow} (\forall x,y \in Var.\ (y',p) \in pts(x) \Longrightarrow \exists p'.\ p \ge p' \,\&\, (y',p') \in pts'(x))$.*
7.*$\gamma \models pts \overset{\text{def}}{\Longleftrightarrow} (\forall x \in Var.\ \gamma(x) \in Addrs \Longrightarrow \exists p > 0.\ (\gamma(x),p) \in pts(x))$.*

*Note 1.*The generality of the language model and the importance of the based probabilistic reference analysis explained in this section recommend that a direct application of the proposed technique is to be included in the code optimization phases of modern compilers. As it is clear above our model model/analysis assumes a general reference analysis. Now given a certain reference analysis (such as [7]), the information of the analysis can easily support our model. This so as most existing models already produce our assumed form for reference analysis. The remaining models produce binary relations (for reference information) that can easily be transformed to our assumed form.

2.1 Types

Our proposed approach for memory safety has the form of a type system. The types of this type system are enrichments of that of the underlying reference types (PTS). Therefore each memory-safety type is a triple (pts,v,p_s) where v is a set of variables that are guaranteed to contain addresses at the program point assigned this type. The symbol p_s denotes a lower bound for probabilities of reaching the program point assigned this type. The following definition gives a precise formalization for the set of memory-safety types (called safety types). The formal interpretation of assigning a safety type to a state is also introduced in the following definition.

Definition 2. –*A safety type is a triple (pts,v,p_s) such that*
–*$pts \in PTS$,*
–*$v \subseteq Var$ such that for every $x \in v$, there exists a pair $(z',p) \in pts(x)$ with $p > p_{ms}$, and*
–*$p_s \in [0,1]$.*

–*$(pts,v,p_s) \le (pts',v',p'_s) \overset{\text{def}}{\Longleftrightarrow} pts \le pts', v \supseteq v'$, and $p_s \ge p'_s \ge p_{ms}$.*
–*A state (γ,p) has type (pts,v,p_s) with respect to the probability p_{ms}, denoted by $(\gamma,p) \models_{p_{ms}} (pts,v,p_s)$, if $\gamma \models pts, \forall x \in v(\gamma(x) \in Addrs)$, and $p_{ms} \le p_s \le p$.*

The key to our technique for probabilistic memory safety of multi-core programs is the following. Suppose that we have a statement S and a reference analysis for S in the form $S : pts \to pts'$. Then for a safety pre-type (pts,v,p_s)[2], a post-type derivation is attempted for S in the memory-safety type system. If such post-type exists then Theorem 1 below guarantees the following. It is memory-safe to execute S starting from a state (γ,p) that is of type (pts,v,p_s) and that is *positive* (has no execution paths with probability less than or equal to p_{ms}).

2.2 Inference Rules

Figure 3 presents the inference rules of the type system for probabilistic memory safety of imperative multi-core programs. Judgments produced by the type system have two forms. The judgment of an arithmetic expression has the form $e : (x,pts,v) \to v'$. The existence of such judgment for an expression e guarantees that calculating e in a state (γ,p) of type (pts,v,p_s) w.r.t. p_{ms}, i.e. $(\gamma,p) \models_{p_{ms}} (pts,v,p_s)$, does not fail. The judgment also guarantees that if the execution of the statement $x := e$ at the state (γ,p) ends at a state (γ',p'), then elements of v' are guaranteed to contain addresses w.r.t. γ'. This is formalized in Lemma 1. The judgment of a statement S has the from $S : (pts,v,p) \to (pts',v',p')$ and assures that if the execution of S from a pre-state of the pre-type ends

[2] Typically $(pts,v,p_s) = (pts,\emptyset,1)$.

© 2015 NSP
Natural Sciences Publishing Cor.

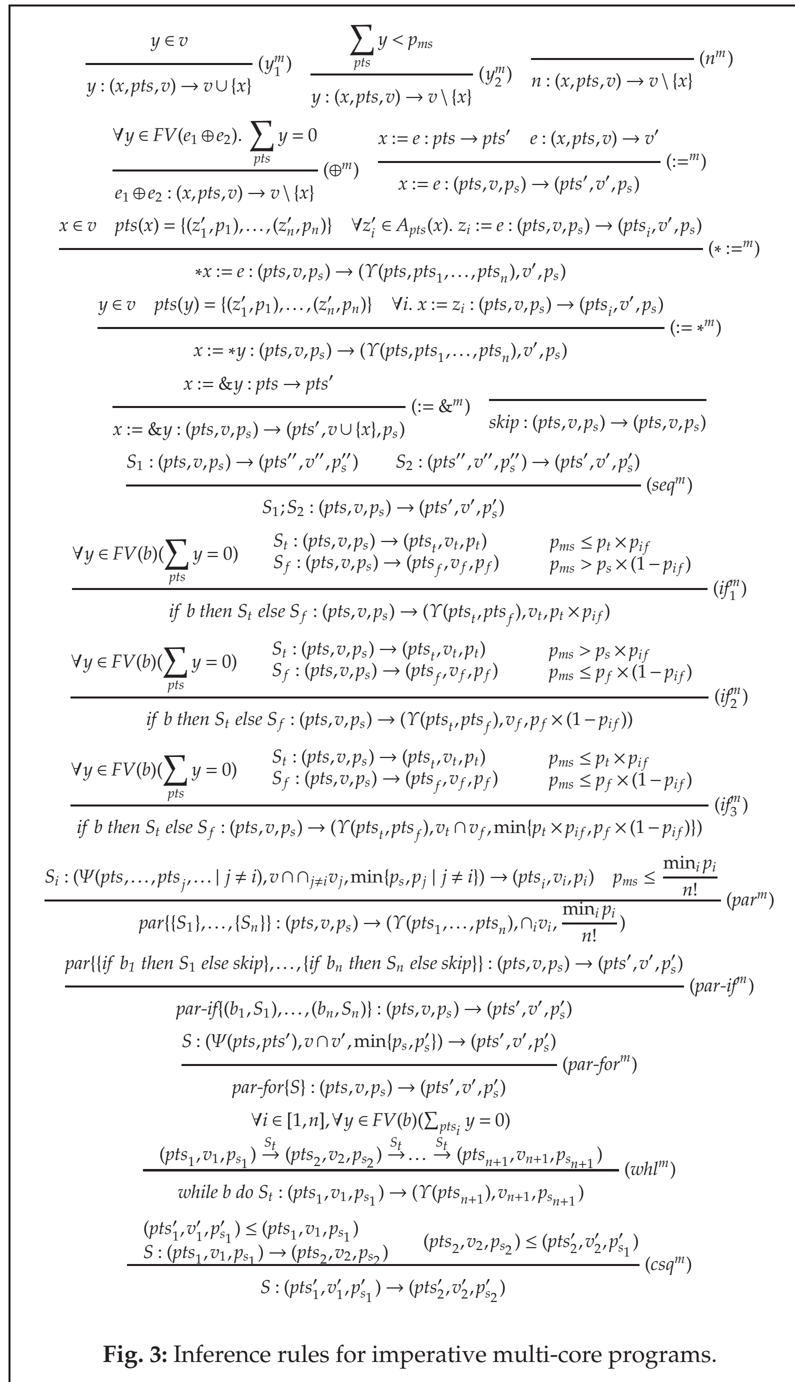

Fig. 3: Inference rules for imperative multi-core programs.

at a post-state, then this post-state is of the post-type. This is proved in Theorem 1.

Comments on the inference rules of Figure 3 are in order. The condition $\sum_{pts} y \geq p_{ms}$ of the rule (y_1^m) assures that when reaching the program point being assigned a type along any of the computation paths whose probabilities greater than or equal to p_{ms}, y will contain an address. The condition $\forall y \in FV(e_1 \oplus e_2)(\sum_{pts} y = 0)$ of the rule $(\oplus^m)$ assures that all free variables of the expression contain integers and hence guarantees the success of calculating the expression $e_1 \oplus e_2$ at any state of the type (pts, v, p_s). In the rules $(* :=^m)$ and $(:= *^m)$, the expression $\Upsilon(pts, pts_1, \ldots, pts_n)$ denotes the reference post-type calculated by the underlying reference

© 2015 NSP
Natural Sciences Publishing Cor.

analysis[3]. $\Upsilon(pts, pts_1, \ldots, pts_n)$ is naturally a function in $\{pts, pts_1, \ldots, pts_n\}$ and its precise shape does not contribute to the calculations of the inference rules $(*:=^m)$ and $(:=*^m)$. The rule (if_1^m) treats the case when the probability of the *then* path is greater than or equal to the threshold p_{ms} and that of the *else* path is strictly less than p_{ms}. In this case, it is sensible to consider the analysis results of S_t and to neglect that of S_f. The rule (par^m) has this shape in order to treat any possible interactions between the statement threads. For the rule (whl^m), n is an upper bound for the trip-count of the loop. Therefore the post-type of the rule is an upper bound for post-types corresponding to number of iterations bounded by n. The statistical and probabilistic information concerning correctness probabilities of *if* statements and trip counts of loops can be obtained using edge-profiling techniques. Heuristics can be used in absence of edge-profiling methods.

*Remark.*Concerting the analysis of while loops, our rule relies on the use of edge profiling for bounds on the number of iterations a loop can take. However, there are techniques that can statically or dynamically evaluate these bounds [16,17]. The application of any of these methods to get the bound required by our method is straightforward. This is so due to the generality of our langauge model.

*Remark.*As it is common with a probabilistic reference analysis [7], we assume that our underlying reference analysis satisfies the following condition. Suppose that pts is the reference type assigned to a program point t of a statement S and $\sum_{pts} y = p$. Then for all computational paths of S with probabilities less than p, the variable y contains no address at the point t.

Lemma 1. *1.Suppose* $(\gamma, p) \models_{p_{ms}} (pts, v, p_s)$ *and* $e : (x, pts, v) \to v'$. *Then* $[\![e]\!]\gamma \neq !$, *and*

$$x := e : (\gamma, p) \to (\gamma', p') \Longrightarrow \forall y \in Var. \ (y \in v' \Longrightarrow \gamma'(y) \in Addrs).$$

2.$(pts, v, p_s) \leq (pts', v', p'_s) \Longrightarrow (\forall (\gamma, p). \quad (\gamma, p) \models_{p_{ms}} (pts, v, p_s) \Longrightarrow (\gamma, p) \models_{p_{ms}} (pts', v', p'_s))$.

*Proof.*The first item is proved by induction on the structure of type derivations:

–The case of the rule (y_1^m): in this case $\gamma' = \gamma[x \mapsto \gamma(y)], p' = p$, and $v' = v \cup \{x\}$. Since $y \in v, y$ is guaranteed to contain an address at the program point before the assignment statement. Therefore $\gamma'(x)$ has an address at the program point after the assignment statement. This justifies adding x to v.

–The case of the rule (y_2^m): in this case $\gamma' = \gamma[x \mapsto \gamma(y)], p' = p$, and $v' = v \setminus \{x\}$. Since $\sum_{pts} y < p_{ms}$ and $p_{ms} \leq p_s \leq p$, by Remark 2.2 y contains no address at the program point before the assignment statement. Hence $\gamma'(x)$ is not assured to contain an address at the program point after the assignment statement. This legitimizes removing x from v.

–The case of the rule $(\oplus^m)$: in this case $p' = p, \gamma' = \gamma[x \mapsto [\![e_1 \oplus e_2]\!]\gamma]$, and $v' = v \setminus \{x\}$. The condition $\forall y \in FV(e_1 \oplus e_2)$ $(\sum_{pts} y = 0)$ assures that $\forall y \in FV(e_1 \oplus e_2).(\gamma(y) \in \mathbb{Z})$. Therefore $[\![e_1 \oplus e_2]\!]\gamma \in \mathbb{Z}$. Hence $\gamma'(x) \in \mathbb{Z}$ which legitimizes removing x from v.

It is straightforward to prove the second item.

2.3 Soundness

For a statement S that has types in the imperative type system, $S : (pts, v, p_s) \to (pts', v', p'_s)$, Theorem 1 assures the following fact about S. It is memory safe to execute S from a positive state (γ, p) of type (pts, v, p_s) w.r.t. p_{ms}, i.e. $(\gamma, p) \models_{p_{ms}} (pts, v, p_s)$. The memory safety means that the program does not abort due to faulty de-referencing (dangling pointers). A *positive* state is a state that does not start any executions paths with probability less than p_{ms}. Theorem 1 also proves soundness of the imperative type system.

Theorem 1.*(Soundness and Probabilistic Memory Safety) Suppose* $S : (pts, v, p_s) \to (pts', v', p'_s)$. *Then*

1.If $(\gamma, p) \models_{p_{ms}} (pts, v, p_s)$ *and S is positive at* (γ, p) *then S does not abort at* (γ, p) *i.e.* $S : (\gamma, p) \not\rightsquigarrow abort$.

2.If $S : (\gamma, p) \rightsquigarrow (\gamma', p')$ *then* $(\gamma, p) \models_{p_{ms}} (pts, v, p_s) \Longrightarrow (\gamma', p') \models_{p_{ms}} (pts', v', p'_s)$.

*Proof.*The proof is by structure induction on the type derivation. Main cases are shown as follows:

–The case of $(:=^m)$: this case follows from Lemma 1 and the soundness of reference analysis.

–The case of $(*:=^m)$: because $x \in v$, there exists $z \in Var$ such that $\gamma(x) = z'$. And because $(\gamma, p) \models_{p_{ms}} (pts, v, p_s)$, we have $z' \in A_{pts}(x)$. $z := e$ does not abort at (γ, p) by induction hypothesis and hence neither does $*x := e$. We also have $z := e : (\gamma, p) \rightsquigarrow (\gamma', p')$. By assumption, it is true that $z := e : (pts, v, p_s) \to (pts', v', p')$. Hence by soundness of $(:=^m), (\gamma', p') \models_{p_{ms}} (pts', v', p')$.

–The case of (if_1^m): the condition $\forall y \in FV(b)(\sum_{pts} y = 0)$ guarantees that all free variables of the condition b have integers (not addresses) under the state γ. This is so because $(\gamma, p) \models_{p_{ms}} (pts, v, p_s)$. Therefore the semantics of b with respect to γ is a Boolean value. We have the following inequalities

[3] The interested reader can check [7] for the details of calculating $\Upsilon(pts, pts_1, \ldots, pts_n)$.

© 2015 NSP
Natural Sciences Publishing Cor.

–$p_t \times p_{if} \geq p_{ms} > p_s \times (1-p_{if})$, and
–$p \geq p_s \geq p_t$.
These inequalities imply that $p \times p_{if} \geq p_t \times p_{if} \geq p_{ms} > p_s \times (1-p_{if})$ which implies $p \times p_{if} \geq p_{ms} > p_s \times (1-p_{if})$. Because S is positive at (γ,p), $[\![b]\!]\gamma = \text{true}$. Now by induction hypothesis S_t does not abort at (γ,p) because $S_t : (pts,v,p_s) \rightarrow (pts_t,v_t,p_t)$, $(\gamma,p) \models_{p_{ms}} (pts,v,p_s)$, and S_t is positive at (γ,p) by [8, Lemma 2]. Therefore the *if* statement does not abort at (γ,p) which completes the proof of (1) for this case.
(2) In this case, we have $(\gamma',p') = (\gamma',p_{if} \times p'')$ where $S_t : (\gamma,p) \rightsquigarrow (\gamma',p'')$. We also have $(pts',v',p') = (\Upsilon(pts_t,pts_f),v_t,p_t \times p_{if})$ where $S_t : (pts,v,p_s) \rightarrow (pts_t,v_t,p_t)$. By induction hypothesis on S_t we have $(\gamma',p'') \models_{p_{ms}} (pts_t,v_t,p_t)$. Therefore $p'' \geq p_t$ which implies $p'' \times p_{if} \geq p_t \times p_{if} \geq p_{ms}$ because $p_t \times p_{if} \geq p_{ms}$. Hence $(\gamma',p') \models_{p_{ms}} (pts_t,v_t,p_t \times p_{if})$ which implies $(\gamma',p') \models_{p_{ms}} (\Upsilon(pts_t,pts_f),v_t,p_t \times p_{if})$ because $(pts_t,v_t,p_t \times p_{if}) \leq (\Upsilon(pts_t,pts_f),v_t,p_t \times p_{if})$. The last inequality holds because $\Upsilon(pts_t,pts_f)$ is an upper bound for pts_t.
–The case of (if_2^m) is similar to the case of (if_1^m).
–The case of (if_3^m): the condition $\forall y \in FV(b)(\sum_{pts} y = 0)$ guarantees that $[\![b]\!]\gamma$ is a Boolean value. We have the following inequalities
–$p_{ms} \leq p_t \times p_{if}$,
–$p_{ms} \leq p_f \times (1-p_{if})$,
–$p_t \leq p_s \leq p$, and
–$p_f \leq p_s \leq p$.
These inequalities imply that

$$p_{ms} \leq p_t \times p_{if} \leq p \times p_{if}$$

and

$$p_{ms} \leq p_f \times (1-p_{if}) \leq p \times (1-p_{if})$$

which implies
–$p_{ms} \leq \min\{p_t \times p_{if}, p_f \times (1-p_{if})\} \leq p \times p_{if}$ and
–$p_{ms} \leq \min\{p_t \times p_{if}, p_f \times (1-p_{if})\} \leq p \times (1-p_{if})$.
Now we consider the case $[\![b]\!]\gamma = \text{false}$. In this case
$S_f : (pts,v,p_s) \rightarrow (pts_f,v_f,p_f)$, $(\gamma,p) \models_{p_{ms}} (pts,v,p_s)$, and S_f is positive at (γ,p) by [8, Lemma 2]. Hence by induction hypothesis S_f does not abort at (γ,p). Consequently the *if* statement does not abort at (γ,p) which completes the proof of (1) for this case.
(2) In this case, we have $(\gamma',p') = (\gamma',p_{if} \times p'')$ where $S_f : (\gamma,p) \rightsquigarrow (\gamma',p'')$. We also have $(pts',v',p') = (\Upsilon(pts_t,pts_f),v_t \cap v_f,\min\{p_t \times p_{if},p_f \times (1-p_{if})\})$ where $S_f : (pts,v,p_s) \rightarrow (pts_f,v_f,p_f)$. By induction hypothesis on S_f we have $(\gamma',p'') \models_{p_{ms}} (pts_f,v_f,p_f)$. Therefore $p'' \geq p_f$ which implies $p'' \times (1-p_{if}) \geq p_f \times (1-p_{if}) \geq \min\{p_t \times p_{if},p_f \times (1-p_{if})\} \geq p_{ms}$ because $\min\{p_t \times p_{if},p_f \times (1-p_{if})\} \geq p_{ms}$. Hence $(\gamma',p') \models_{p_{ms}} (pts_f,v_f,\min\{p_t \times p_{if},p_f \times (1-p_{if})\})$ which implies $(\gamma',p') \models_{p_{ms}} (\Upsilon(pts_t,pts_f),v_t \cap v_f,\min\{p_t \times p_{if},p_f \times (1-p_{if})\})$ because $(pts_f,v_f,\min\{p_t \times p_{if},p_f \times (1-p_{if})\}) \leq (\Upsilon(pts_t,pts_f),v_t \cap v_f,\min\{p_t \times p_{if},p_f \times (1-p_{if})\})$. The last inequality holds because $\Upsilon(pts_t,pts_f)$ is an upper bound for pts_f and $v_f \supseteq (v_t \cap v_f)$.
–The case of (par^m): (1) Suppose that $\theta : \{1,\dots,n\} \rightarrow \{1,\dots,n\}$ is a permutation. $(\gamma,p) \models_{p_{ms}} (pts,v,p_s)$ implies $(\gamma,p) \models_{p_{ms}} (\Psi(pts,\dots,pts_j,\dots \mid j \neq \theta(1)),v \cap \cap_{j\neq\theta(1)} v_j,\min\{p_s,p_j \mid j \neq \theta(1)\})$. Recall that $\Psi(pts,\dots,pts_j,\dots \mid j \neq \theta(1))$ is a lower bound for pts. By Lemma [8, Lemma 2], $S_{\theta(1)}$ is positive at (γ,p). Therefore $S_{\theta(1)}$ does not abort at γ by induction hypothesis. Hence either the execution of $S_{\theta(1)}$ terminates at a state (γ_2,p_2) such that $(\gamma_2,p_2) \models_{ms} (pts_{\theta(1)},v_{\theta(1)},p_{\theta(1)})$ or enters an infinite loop at (γ,p). Therefore $(\gamma_2,p_2) \models_{ms} (\Psi(pts,\dots,pts_j,\dots \mid j \neq \theta(2)),v \cap \cap_{j\neq\theta(2)} v_j,\min\{p_s,p_j \mid j \neq \theta(2)\})$. Therefore, clearly (1) is proved via a simple induction on n.
(2) In this case the existence of a permutation $\theta : \{1,\dots,n\} \rightarrow \{1,\dots,n\}$ and $n+1$ states $(\gamma,p) = (\gamma_1,p_1),\dots,(\gamma_{n+1},p_{n+1}) = (\gamma',p'')$ such that for every $1 \leq i \leq n$, $S_{\theta(i)} : (\gamma_i,p_i) \rightsquigarrow (\gamma_{i+1},p_{i+1})$ is guaranteed. In this case $p' = \frac{p''}{n!}$. The fact that $(\gamma_1,p_1) \models_{p_{ms}} (pts,v,p_s)$ implies the fact that $(\gamma_1,p_1) \models_{p_{ms}} (\Psi(pts,\dots,pts_j,\dots \mid j \neq \theta(1)),v \cap \cap_{j\neq\theta(1)} v_j,\min\{p_s,p_j \mid j \neq \theta(1)\})$. Hence $(\gamma_2,p_2) \models_{p_{ms}} (pts_{\theta(1)},v_{\theta(1)},p_{\theta(1)})$ by the induction hypothesis. This implies $(\gamma_2,p_2) \models_{p_{ms}} (\Psi(pts,\dots,pts_j,\dots \mid j \neq \theta(2)),v \cap \cap_{j\neq\theta(2)} v_j,\min\{p_s,p_j \mid j \neq \theta(2)\})$. Hence Again $(\gamma_3,p_3) \models_{p_{ms}} (pts_{\theta(2)},v_{\theta(2)},p_{\theta(2)})$ by the induction hypothesis. Therefore a simple induction on n shows that $(\gamma',p') = (\gamma_{n+1},p_{n+1}) \models_{p_{ms}} (pts_{\theta(n)},v_{\theta(n)},p_{\theta(n)})$ implying
$(\gamma',p'') \models_{p_{ms}} (\Upsilon(pts_1,\dots,pts_n),\cap_i v_i,\min_i p_i)$. Hence because $p_{ms} \leq \frac{\min_i p_i}{n!}$, we get $(\gamma',p') \models_{p_{ms}} (pts',v',p') = (\Upsilon(pts_1,\dots,pts_n),\cap_i v_i,\frac{\min_i p_i}{n!})$ as required.
–The case of $(par-for^m)$: (1) The proof of this item is in line with item (1) of the (par^m) case.
(2) In this case there exists n such that $par\{\{S\}_1,\dots,\{S\}_n\} : (\gamma,p) \rightsquigarrow (\gamma',p')$. We get $S : (\Psi(pts,pts'),v \cap v',\min\{p_s,p'_s\}) \rightarrow (pts',v',p'_s)$ by induction hypothesis. Then by (par^m) we infer that $par\{\{S\}_1,\dots,\{S\}_n\} : (pts,v,p_s) \rightarrow (pts',v',p'_s)$. Consequently by the soundness of (par^m), $(\gamma',p') \models_{p_{ms}} (pts',v',p'_s)$.

© 2015 NSP
Natural Sciences Publishing Cor.

3 Probabilistic Memory Safety for Multi-Core Object-Oriented Programs

This section generalizes the type system of Section 2 to cover multi-core object-oriented programs. More specifically, this sections introduces a new model for multi-core object-oriented programs together with a new type system for the analysis of their probabilistic memory safety. The model language (dubbed *MC-OOP*) is equipped with basic object-oriented concepts as inheritance and sub-typing. Figure 4 presents the syntax of *MC-OOP*.

Functions host local variables which are alive as long as their functions are active. Function parameters are local variables. Instance variables of a class store the class's internal state. A set of classes and a "main" function are the components of programs of *MC-OOP*. A class has function definitions. A function f consists of a parameter p_f, a statement S_f, and an expression e_f which denotes the returned value by the function. Semantic spaces and naming conventions used in this section are presented in Definition 3.

Definition 3.*(Semantic spaces and naming conventions)*

- *C denotes the set of class names with $C, D \in \mathbf{C}$.*
- *$LVar$ denotes the set of local variables with $o \in LVar$.*
- *$IVar_C$ denotes instance variables of C with $v \in IVar_C$.*
- *$FunNames$ denotes the set of function names with $f \in FunNames$.*
- *Fun_C denotes functions of C and $p \in [0,1]$.*
- *$\mathcal{D} = \mathbb{Z} \cup \mathcal{A} \cup \{\perp\}$ denotes model values with $d \in D$.*
- *$\mathcal{A}$ denotes the set of memory addresses with $\alpha \in \mathcal{A}$.*
- *$\mathcal{S} = \{s \mid s : LVar \to \mathcal{D}\}$ is the set of stacks with $s \in \mathcal{S}$.*
- *$\mathcal{O} = IVar_C \to \mathcal{D}$ is the set of objects contents with $I_{(C,n)} \in \mathcal{O}$.*
- *$\mathcal{H} = \mathcal{A} \to_p \{(C, n, I_{(C,n)})\}$ is the set of heaps with $h \in \mathcal{H}$.*
- *$States = \mathcal{S} \times \mathcal{H}$ denotes memory states with $(s,h) \in States$.*
- *$h_i(\alpha), i \in \{1,2,3\}$ denotes the ith component of $h(\alpha)$).*
- *this is the current active object.*
- *$Fun_C \to LVar \times Stmt \times AExpr$ denotes functions components with $F_C \in Fun_C$ where $f \mapsto (p_f, S_f, e_f)$.*
- *$C \ll D$ denotes that C is a subclass of a class D.*
- *$\leq$ is the reflexive transitive closure of $\ll$.*
- *$Loc = LVar \cup (\cup_{C \in \mathbf{C}, v \in IVar_C}\{(C,v)\})$ is the set of all locations with $l \in Loc$.*

As clarified earlier, a probabilistic memory safety is typically built on a probabilistic reference analysis. We assume an underlying probabilistic reference analysis that calculates for every program (of the langauge *MC-OOP*) point a reference type rt taken from a set of probabilistic reference types RT. Definition 4 presents a formalization of the set RT and a sub-typing relation on its types. The soundness of the proposed memory safety analysis relies on that of the underlying reference analysis with respect to a relevant semantics. For a given statement S of *MC-OOP*, the reference analysis calculates for a given pre-reference type rt a post-reference type rt', such that $S : rt \to rt'$ (in the underlying reference analysis). The soundness means that if executing S at a state (s,h,p) of type rt ends at (s',h',p'), then the last state is of type rt'.

A *class* analysis is also necessary to achieve the analysis of probabilistic memory safety of the language *MC-OOP*. We assume the existence of such analysis which calculates for every program point a class type ct taken from a set of class types CT. Definition 4 presents a formalization of the set CT and a sub-typing relation on its types. The soundness of the *class* analysis is analog to that of reference analysis. Hence for a given statement and two pre-types rt and ct, the reference and class analyses find two post types rt' and ct' such that $S : rt \to rt'$ and $S : ct \to ct'$ (denoted by $S : (rt, ct) \to (rt', ct')$). For an arthemtic expression e, the judgment $e : (rt, ct) \to Cs$ denotes that Cs is the set of classes that e may points to in state of types rt and ct.

Definition 4. 1.$\mathcal{A}_p = \mathcal{A} \times [0,1]$.

2.$Ppst = \{rt \mid rt : Loc \to 2^{\mathcal{A}_p} \text{ s.t. } (\alpha, p_1), (\alpha, p_2) \in rt(l) \Longrightarrow p_1 = p_2\}$.

3.$\sum_{rt} l = \sum_{(\alpha,p) \in rt(l)} p$.

4.$A_{rt}(l) = \{\alpha \mid \exists p > 0.\ (\alpha, p) \in rt(l)\}$.

5.$RT = \{rt \in Ppst \mid \forall l \in Loc.\ \sum_{rt} l \leq 1\}$.

6.$rt \leq rt' \overset{\text{def}}{\Longleftrightarrow} (\forall l, l' \in Loc.\ (\alpha, p) \in rt(l) \Longrightarrow \exists p'.\ p \geq p' \,\&\, (\alpha, p') \in rt'(l))$.

7.$(s,h) \models rt \overset{\text{def}}{\Longleftrightarrow} (\forall l \in Loc.\ \ (s(l) \in \mathcal{A} \vee h(l) \in \mathcal{A}) \Longrightarrow \exists p > 0.\ ((s(l), p) \in rt(l) \vee (h(l), p) \in rt(l))$.

8.$CT = Loc \to 2^{\mathbf{C}}$.

9.$(s,h) \models ct \overset{\text{def}}{\Longleftrightarrow} \forall l \in Loc.\ (s(l) \vee h(l)$ *points at a class* $C \Longrightarrow C \in ct(l))$.

10.$ct \leq ct' \overset{\text{def}}{\Longleftrightarrow} \forall l \in Loc.\ ct(l) \subseteq ct'(l)$.

3.1 Types

The new technique proposed in this section is in the form of a type system. Enrichments of reference and class types are the types of proposed type system. Hence, a type for the proposed type system has the form (rt, ct, V, p_s). The symbols V and p_s have the same meanings as in the previous section. Definition 5 gives the formal description of the set of memory-safety types. This definition also formalizes the concept of assigning a safety type to a memory state.

Definition 5. –*A safety type is a quadruple (rt, ct, V, p_s) such that*

- *$rt \in RT$, $ct \in CT$,*
- *$V \subseteq Loc$ such that for every $l \in V$, there exists a pair $(\alpha, p) \in rt(l)$ with $p > p_{ms}$, and*

© 2015 NSP

$$
\begin{aligned}
e \in \text{AExpr} &::= n \mid o \mid e.v \mid e_1 i_{op}\ e_2 \mid \text{this} \mid (C)e \\
b \in \text{BExpr} &::= \text{true} \mid \text{false} \mid e_1 c_{op}\ e_2 \mid b_1 b_{op}\ b_2 \\
S \in \text{Stat} &::= o := e \mid e_1.v := e_2 \mid o_1 := o_2.f(e) \mid \\
&\quad\ o_1 := super.f(e) \mid o := \text{ new } C \mid S;S \mid \\
&\quad\ \text{if } b \text{ then } S_t \text{ else } S_f \mid \text{ while } b \text{ do } S_t \mid \\
&\quad\ par\{\{S_1\},\ldots,\{S_n\}\} \mid par\text{-}for\{S\} \mid \\
&\quad\ par\text{-}if\{(b_1,S_1),\ldots,(b_n,S_n)\} \\
\text{funs} \in \text{Fun} &::= \text{f}(p)\{S;\ \text{return}(e);\} \\
\text{inhrt} \in \text{Inherits} &::= \epsilon \mid \text{ inherits } C \\
\text{cls} \in \text{Class} &::= \text{ class } C \text{ inhrt } \{\text{fun}^*\} \\
\text{prog} \in \text{Prog} &::= \text{cls}^* \text{ main() } \{S\}
\end{aligned}
$$

Fig. 4: The programming language MC-OOP.

– $p_s \in [0,1]$.
– $(rt,ct,V,p_s) \le (rt',ct',V',p'_s) \overset{\text{def}}{\iff} rt \le rt', ct \le ct', V \supseteq V'$, *and* $p_s \ge p'_s \ge p_{ms}$.
– *A state* (s,h,p) *has type* (rt,ct,V,p_s) *with respect to the probability* p_{ms}, *denoted by* $(s,h,p) \models_{p_{ms}} (rt,ct,V,p_s)$, *if* $(s,h) \models rt$, $(s,h) \models ct, \forall l \in V(s(l) \in \mathcal{A} \vee h(l) \in \mathcal{A})$, *and* $p_{ms} \le p_s \le p$.

The idea behind the type system of probabilistic memory safety of object oriented multi-core programs presented in this section is the following. We assume that we are given a program whose main statement is S and a reference and class analyses for S in the form $S : (rt,ct) \to (rt',ct')$. If a post-type for a safety pre-type (rt,ct,V,p_s) exists (via a post-type derivation) in the proposed type system then the following is guaranteed. Running the program in a positive state (s,h,p) of type (rt,ct,V,p_s) is memory-safe.

3.2 Inference Rules

Figures 5 and 6 present the inference rules of the type system for probabilistic memory safety of object oriented multi-core programs. Two forms of judgments are produced by the type system. The first of them is for arithmetic expressions and has the form $e : (l,rt,ct,V) \to V'$. This judgment guarantees:

– The success of calculating e in a state (s,h,p) of type (rt,ct,V,p_s) w.r.t. p_{ms}, i.e. $(s,h,p) \models_{p_{ms}} (rt,ct,V,p_s)$.
– If the state (s',h',p') resulted from executing $o := e$ in a state of type (rt,ct,V,p_s) then elements of V' are addresses in (s',h').

This is presented in Lemma 2. The second form of judgment is of the form $S : (rt,ct,V,p) \to (rt',ct',V',p')$. The semantics of the judgment is that if the execution of S in a pre-state of type (rt,ct,V,p) ends at a post-state, then the post-state is of type (rt',ct',V',p'). Theorem 2 formalizes this semantics.

Comments on the inference rules in the previous section apply on rules of Figures 5 and 6. However more comments are in order. In the rule $(:=^{mt}_{e.v})$, V' is an approximation of all sets of locations resulting by using different classes that e may evaluate to. The precondition of the rule $(:=^{mt}_{o.f})$ presumes that Cs is the set of all classes that o_2 may evaluate to. For each $C \in Cs$, the precondition assumes a common reference and class analyses for the statement of the function f of C $(S_f : (rt,ct) \to (rt',ct'))$. Finally the precondition assumes a set V' that covers all the expressions returned by all functions f of classes in Cs. In the rule $(:=^{mt}_{super})$, Cs' is the set of all potential active classes and Cs denotes the super classes (of elements of Cs') that include a definition for f. The rules assumes a procedure *super*, whose inputs are a function name and a class name. This procedure finds the first ancestor of the class containing the function.

Lemma 2. *1.Suppose* $(s,h,p) \models_{p_{ms}} (rt,ct,V,p_s)$ *and* $e : (l,rt,ct,V) \to V'$. *Then* $[\![e]\!](s,h) \neq \perp$, *and*

$$o := e : (s,h,p) \to (s',h',p') \Longrightarrow \forall l \in V' \ (s(l) \in dom(h) \vee h(l) \in dom(h)).$$

2. $(rt,ct,V,p_s) \le (rt',ct',V',p'_s) \Longrightarrow (\forall (s,h,p).\ (s,h,p) \models_{p_{ms}} (rt,ct,V,p_s) \Longrightarrow (s,h,p) \models_{p_{ms}} (rt',ct',V',p'_s))$.

For a typed statement S in the probabilistic memory-safety type system (i.e. $S : (pts,v,p_s) \to (pts',v',p'_s)$), Theorem 2 guarantees the memory safety of executing S from a positive state (s,h,p) such that $(\gamma,p) \models_{p_{ms}} (pts,v,p_s)$.

Theorem 2. *Suppose* $S : (rt,ct,V,p_s) \to (rt',ct',V',p'_s)$. *Then*

© 2015 NSP
Natural Sciences Publishing Cor.

$$\frac{}{n : (l,rt,ct,V) \to V \setminus \{l\}}\ (n^{mt}) \qquad \frac{o \in V}{o : (l,rt,ct,V) \to V \cup \{l\}}\ (o_1^{mt}) \qquad \frac{\sum_{rt} o < p_{ms}}{o : (l,rt,ct,V) \to V \setminus \{l\}}\ (o_2^{mt})$$

$$\frac{e : (rt,ct) \to Cs \quad \forall C \in Cs.\ (C,v) \in V}{e.v : (l,rt,ct,V) \to V \cup \{l\}}\ (e.o_1^{mt}) \qquad \frac{e : (rt,ct) \to Cs \quad \forall C \in Cs.\ \sum_{rt} C.v < p_{ms}}{e.v : (l,rt,ct,V) \to V \setminus \{l\}}\ (e.o_2^{mt})$$

$$\frac{\forall l \in FV(e_1 \oplus e_2).\ \sum_{rt} l = 0}{e_1 \oplus e_2 : (l,rt,ct,V) \to V \setminus \{l\}}\ (\oplus^{mt}) \qquad \frac{e : (l,rt,ct,V) \to V'}{C(e) : (l,rt,ct,V) \to V'}\ (C(e)^{mt})$$

$$\frac{o := e_2 : (rt,ct) \to (rt',ct') \quad e : (o,rt,ct,V) \to V'}{o := e : (rt,ct,V,p_s) \to (rt',ct',V',p_s)}\ (:=_o^{mt})$$

$$\frac{\begin{array}{l} e_1.v := e_2 : (rt,ct) \to (rt',ct') \\ e_1 : (rt,ct) \to Cs \end{array} \quad \forall C \in Cs.\ e_2 : (C.v,rt,V) \to V'}{e_1.v := e_2 : (rt,ct,V,p_s) \to (rt',ct',V',p_s)}\ (:=_{e.v}^{mt})$$

$$\frac{\begin{array}{l} o_2 : (rt,ct) \to Cs \\ \forall C \in Cs.\ F_C(f) = (p_f,S_f,e_f) \wedge S_f : (rt,ct) \to (rt',ct') \wedge e_f : (o_1,rt,V) \to V' \end{array}}{o_1 := o_2.f(e) : (rt,ct,V,p_s) \to (rt',ct',V',p_s)}\ (:=_{o.f}^{mt})$$

$$\frac{\begin{array}{l} this : (rt,ct) \to Cs' \qquad Cs = \{C \mid \exists D \in Cs \wedge super(D,f) = C\} \\ \forall C \in Cs.\ F_C(f) = (p_f,S_f,e_f) \wedge S_f : (rt,ct) \to (rt',ct') \wedge e_f : (o_1,rt,V) \to V' \end{array}}{o_1 := super.f(e) : (rt,ct,V,p_s) \to (rt',ct',V',p_s)}\ (:=_{super}^{mt})$$

$$\frac{o := \text{ new } C : (rt,ct) \to (rt',ct')}{o := \text{ new } C : (rt,ct,V,p_s) \to (rt',ct',V \cup \{o\},p_s)}\ (:=_{new}^{mt})$$

$$\frac{\begin{array}{l} S_1 : (rt.ct,V,p_s) \to (rt'',ct'',V'',p_s'') \\ S_2 : (rt'',ct'',V'',p_s'') \to (rt',ct',V',p_s') \end{array}}{S_1;S_2 : (rt,ct,V,p_s) \to (rt',ct',V',p_s')}\ (seq^{mt})$$

$$\frac{\forall l \in FV(b)(\sum_{rt} l = 0) \quad \begin{array}{l} S_t : (rt,ct,V,p_s) \to (rt_t,ct_t,V_t,p_t) \\ S_f : (rt,ct,V,p_s) \to (rt_f,ct_f,V_f,p_f) \end{array} \quad \begin{array}{l} p_{ms} \le p_t \times p_{if} \\ p_{ms} > p_s \times (1-p_{if}) \end{array}}{if\ b\ then\ S_t\ else\ S_f : (rt,ct,v,p_s) \to (\Upsilon(rt_t,rt_f),\Upsilon(ct_t,ct_f),V_t,p_t \times p_{if})}\ (if_1^{mt})$$

$$\frac{\forall l \in FV(b)(\sum_{rt} l = 0) \quad \begin{array}{l} S_t : (rt,ct,V,p_s) \to (rt_t,ct_t,V_t,p_t) \\ S_f : (rt,ct,V,p_s) \to (rt_f,ct_f,V_f,p_f) \end{array} \quad \begin{array}{l} p_{ms} > p_s \times p_{if} \\ p_{ms} \le p_f \times (1-p_{if}) \end{array}}{if\ b\ then\ S_t\ else\ S_f : (rt,ct,V,p_s) \to (\Upsilon(rt_t,rt_f),\Upsilon(ct_t,ct_f),V_f,p_f \times (1-p_{if}))}\ (if_2^{mt})$$

$$\frac{\forall l \in FV(b)(\sum_{rt} l = 0) \quad \begin{array}{l} S_t : (rt,ct,V,p_s) \to (rt_t,ct_t,V_t,p_t) \\ S_f : (rt,ct,V,p_s) \to (rt_f,ct_f,V_f,p_f) \end{array} \quad \begin{array}{l} p_{ms} \le p_t \times p_{if} \\ p_{ms} \le p_f \times (1-p_{if}) \end{array}}{\begin{array}{l} if\ b\ then\ S_t\ else\ S_f : (rt,ct,V,p_s) \to \\ (\Upsilon(rt_t,rt_f),\Upsilon(ct_t,ct_f),V_t \cap V_f,\min\{p_t \times p_{if},p_f \times (1-p_{if})\}) \end{array}}\ (if_3^{mt})$$

Fig. 5: Inference rules for object-oriented multi-core programs 1.

1. *If* $(s,h,p) \models_{p_{ms}} (rt,ct,V,p_s)$ *and* S *is positive at* (s,h,p) *then* S *does not abort at* (s,h,p).
2. *If* $S : (s,h,p) \rightsquigarrow (s',h',p')$ *then* $(s,h,p) \models_{p_{ms}} (rt,ct,V,p_s) \Longrightarrow (s',h',p') \models_{p_{ms}} (rt',ct',V',p_s')$.

The proof of Lemma 2 and Theorem 2 are inline with that for Lemma 1 and Theorem 1, respectively. The proof of Theorem 2 in particular is by induction on the program structures using the rules of the proposed type system. The most involved case is that of (par^{mt}). However is it pretty similar to the the proof in Theorem 1 for the case of the rule (par^m). The obtained result is quite important for modern compilers as it is a main object nowadays of such compilers to guarantee memory safety rather than just do the basic compilation process.

4 Related Work

Related work includes security vulnerabilities, memory management, debugging and testing, garbage collection, failure masking, analysis of multi-core programs, analysis of object-oriented programs, and type systems in program analysis.

A classical trend to reduce vulnerabilities of heaps to security attacks is to use a randomization approach for both choosing the base address [18] of the heap and buffering allocation requests [19].

© 2015 NSP
Natural Sciences Publishing Cor.

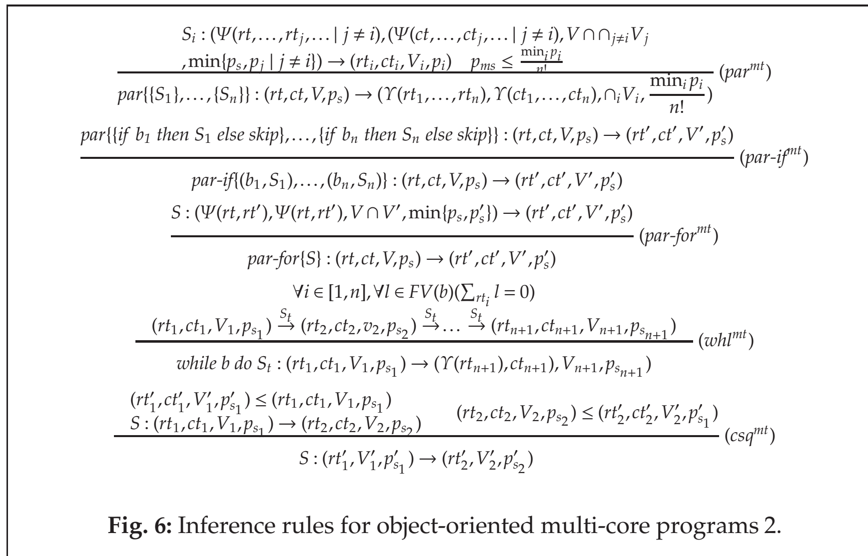

Fig. 6: Inference rules for object-oriented multi-core programs 2.

However this classical approach is believed not to be very effective on 32-bit operating systems [20]. More recent work [21] hides object layouts from attackers in any duplicate.

To maintain fast allocation and low fragmentation, dynamic techniques for memory management scarify strength. Repeated memory frees and heap corruption due to buffer overflows affect most *malloc* implementations. While some memory managers [22] prevent heap corruption via separating metadata from the heap, other managers [23] just recognize heap corruption.

Via simulation and multiple rewrites on run time, techniques for debugging and testing [24] discover memory errors in programs. Drawbacks of these techniques include increasing space costs and restrictive runtime overheads. These burdens can only be tolerated during testing. Other techniques significantly reduce runtime overhead and discover memory leaks via using sampling [25].

The drawback of garbage collection [21], a technique helping avoiding errors caused by dangling pointers, is that to perform reasonably it requires an ample amount of space. In particular, the technique of [21] prevents overwrites via separating metadata from heap. This technique, which is probabilistic rather than absolute like most other related techniques, also neglects multiply and faulty frees.

Failure masking [26] is a terminology describing stopping programs from aborting. Pool allocation, a technique of failure masking, classifies objects into pools according to their types and hence guarantees that objects overwrite only dangling pointers of the same type. The drawback of this technique is the unpredictability of behavior of the produced program. Other techniques, failure-oblivious systems, neglect faulty writes and create values for reading uninitialized memory.

None of techniques mentioned above that treat dangling pointers deal with multi-core programs nor provide proofs for correctness of each individual test. Sound type systems for reference analysis and memory safety of Multi-core programs are presented in [3]. However all techniques mentioned so far are absolute; not probabilistic like the technique presented here. Hence our work has the advantage, over all the related work, of being usable in speculative-optimizations sections of modern compilers.

The analysis of multi-core programs is receiving a growing research interest. The possible interactions between various cores significantly complicate analysis of multi-core programs. Work in this area is typically classified into two main categories: techniques designed specifically for optimization or error-detection of multi-core programs and techniques originally designed for analysis of sequential programs and successfully extended to cover multi-core programs.

The work in the first category above includes dataflow frameworks for bitvector problems [27], concurrent static single assignment forms [29], reaching definitions [28], constant propagation [9], code motion [30]. None of these techniques studies memory-safety of multi-core programs leaving alone probabilistic memory safety of these programs. The work in the other category above includes synchronization analysis [31], race detection [32], reference analysis [3], and deadlock analysis [10].

Reference and class escape analysis that is combined class-modular and that is analyzing class

© 2015 NSP
Natural Sciences Publishing Cor.

declarations is presented in [33]. In [34], a group of answer set programming rules to represent the generation of reference information through paths of object-oriented programs is introduced.

The use of type systems in program analysis is becoming a mainstream approach for applications that require a proof for each individual program analysis like certified code. General methods for transforming monotone data-flow analyses (forward and backward) into type systems are presented in [35]. Constant folding, common subexpression elimination, and dead code elimination for *while* language as type systems are presented in [36].

Acknowledgement

The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

Disclosure

This paper is an extended and revised version of [8].

Appendix: Probabilistic Operational Semantics

This appendix reviews and augments a probabilistic operational semantics that we presented in [7] for the programming language (Figure 1) that we study. This language is the simple *while* language extended with new basic statements for parallel programming: $par\{\{S_1\},\ldots,\{S_n\}\}$ (fork-join), $par\text{-}if\{(b_1,S_1),\ldots,(b_n,S_n)\}$ (conditionally spawned threads), and $par\text{-}for\{S\}$ (parallel loops). At the begin of the *par* command, the basic parallel command, a main core initiates the run of various concurrent inner cores. The subsequent statement (to the main core) can only be executed when the run of all inner cores are finished. The semantics of conditionally spawned command is akin to that of fork-join. The run of $par\text{-}if\{(b_1,S_1),\ldots,(b_n,S_n)\}$ includes initiating the conditionally concurrent runs of the n cores; only if b_i is *true*, S_i is executed. The following command (to the *par-if* statement) can only be executed when the runs of all conditional cores are finished. The semantics of parallel loop construct $par\text{-}for\{S\}$ includes running concurrently a statically unknown number of cores that all are S.

Semantically a computational state is a pair (γ,p): γ is a mapping from variables to values (integers plus symbolic addresses) and $p\in[0,1]$. The intuition is that p is the probability of reaching γ. Definition 6 presents a formal definition for computational states.

Definition 6. *1.* $Addrs = \{x' \mid x \in Var\}$ *and* $Val = \mathbb{Z}\cup Addrs$.
2. $\gamma \in \Gamma = Var \longrightarrow Val$.
3. A state is either an abort or a pair (γ,p) *such that* $p\in[0,1]$.

We adopt the usual semantics for arithmetic and Boolean expressions, except that we do not allow arithmetic and Boolean operations on references (pointers). Figure 7 presents inference rules of our probabilistic operational semantics (transition relation).

Definitions 7 and 8 introduce terminologies that are used above to discuss and prove soundness of our proposed type system for probabilistic memory safety.

Definition 7. *For a statement* S*, a judgement of the form* $S:(\gamma,p)\rightarrow(\gamma',p')$ *is described as a computation (or an execution) path. The quantity* p' *is the probability of this execution path.*

Definition 8. *Suppose that* $S:(pts,v,p_s)\rightarrow(pts',v',p'_s)$*. Then* S *is positive at a state* (γ,p) *of type* (pts,v,p_s) *if along any execution path of* S *that starts at* (γ,p) *whenever an if statement, whose condition is true with probability* p_{if}*, is encountered at a state* (γ'',p'') *whose type is* (pts'',v'',p''_s) *in the proof tree of* $S:(pts,v,p_s)\rightarrow(pts',v',p'_s)$*, i.e.*

$$(\gamma,p) \rightsquigarrow \ldots (\gamma'',p'') \overset{if\ b\ then\ldots}{\rightsquigarrow} \ldots$$

the following are true:

–*if* $p_{if}\times p''\geq p_{ms}>(1-p_{if})\times p''_s$*, then* $[\![b]\!]\gamma''=true$*.*
–*if* $p_{if}\times p''_s<p_{ms}\leq(1-p_{if})\times p''$*, then* $[\![b]\!]\gamma''=false$*.*

A simple structure induction proves lemma 3 which is used in the soundness proof above.

Lemma 3. *Suppose that* $S:(pts,v,p_s)\rightarrow(pts',v',p'_s),(\gamma,p)\models_{ms}(pts,v,p_s)$*, and* S *is positive at* (γ,p)*. Suppose also that along an execution path of* S *that starts at* (γ,p)*, a sub-statement* S' *of* S *is encountered at a state* (γ'',p'')*, i.e.*

$$(\gamma,p) \rightsquigarrow \ldots (\gamma'',p'') \overset{S'}{\rightsquigarrow} \ldots$$

If $S':(pts_1,v_1,p_{1s})\rightarrow(pts'_2,v'_2,p'_{2s})$ *in the proof tree of* $S:(pts,v,p_s)\rightarrow(pts',v',p'_s)$ *and* $(\gamma'',p'')\models_{ms}(pts_1,v_1,p_{1s})$ *then* S' *is positive at* (γ'',p'')*.*

© 2015 NSP
Natural Sciences Publishing Cor.

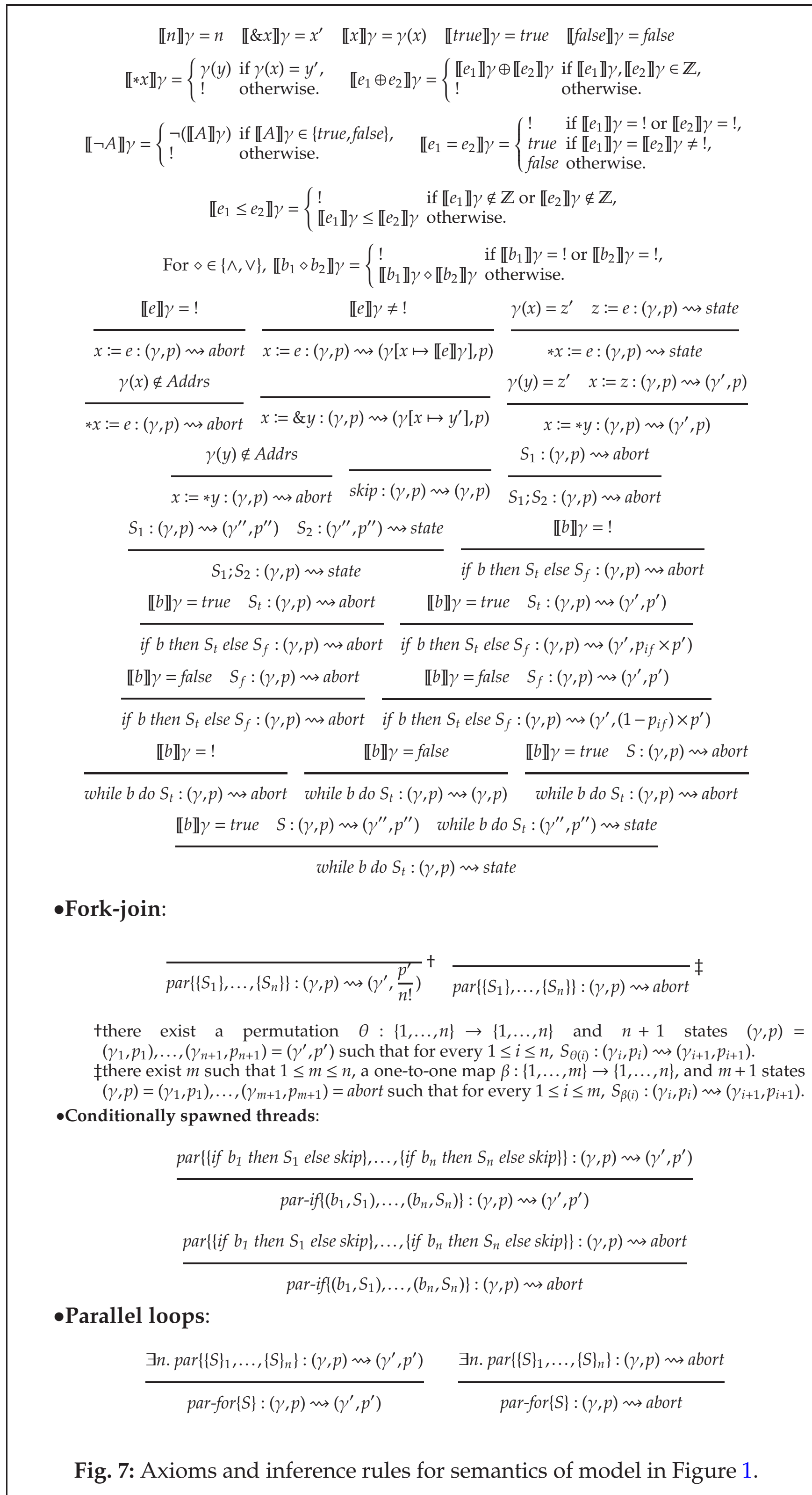

Fig. 7: Axioms and inference rules for semantics of model in Figure 1.

© 2015 NSP
Natural Sciences Publishing Cor.

References

[1] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, **35**, 29-63 (2003).

[2] P. J. Deitel. C++ How to Program. P.J. Deitel, H.M. Deitel. Pearson Education, 7th edition, 2010.

[3] M. A. El-Zawawy. Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In Beniamino Murgante, Osvaldo Gervasi, Andryes Iglesias, David Taniar, and Bernady O. Apduhan, editors, ICCSA (5), Lecture Notes in Computer Science, *Springer*, **6786**, 355-369 (2011).

[4] J. Seibert, H. Okkhravi, E. Sderstrm. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. ACM Conference on Computer and Communications Security 54-65 (2014).

[5] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun.ACM* **51**, 87-95 (2008).

[6] J. Da Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In John Paul Shen and Margaret Martonosi, editors, ASPLOS, *ACM*, 416-425 (2006).

[7] M. A. El-Zawawy. Probabilistic pointer analysis for multithreaded programs. *ScienceAsia* **37(4)**, 344-354 (2011).

[8] M. 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), Lecture Notes in Computer Science, *Springer* **7975**, 516-530 (2013).

[9] D. Callahan, K.h D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. SIGPLAN Not. **39**, 155-166 (2004).

[10] F. Ahmad, H. Huang, and X. LongWang. Petri net modeling and deadlock analysis of parallel manufacturing processes with shared-resources. *J. Syst. Softw.* **83**, 675-688 (2010).

[11] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Lujcyan. Optimizing software runtime systems for speculative parallelization. TACO **4**,39-49 (2013).

[12] L. Gao, L. Li, J. Xue, and P. Yew. Seed: A statically greedy and dynamically adaptive approach for speculative loop execution. *IEEE Trans. Computers* **62(5)**,1004-1016 (2013).

[13] Xi. Dai, A. Zhai, W. Hsu, and P. Yew. Ageneral compiler framework for speculative optimizations using data speculative code motion. In CGO, *IEEE Computer Society*, 280-290 (2005).

[14] J. Lin, T. Chen, W. Hsu, and P. Yew. Speculative register promotion using advanced load address table (alat). In CGO, *IEEE Computer Society*, 125-134 (2003).

[15] K. Cleereman, M. Cheatham, and K. Thirunarayan. Runtime support of speculative optimization for offline escape analysis. In Hamid R. Arabnia and Hassan Reza, editors, Software Engineering Research and Practice, CSREA Press, 484-489 (2007).

[16] R. Blanc, T. A. Henzinger, T. Hottelier, L. Kovcs. ABC: Algebraic Bound Computation for Loops. LPAR (Dakar), 103-118 (2010).

[17] B. Rieder, P. P. Puschner, I. Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. WISES, 1-7 (2008).

[18] S. Antonatos and K. G. Anagnostakis. Tao: Protecting against hitlist worms using transparent address obfuscation. In Herbert Leitold and Evangelos P. Markatos, editors, Communications and Multimedia Security, volume of Lecture Notes in Computer Science, *Springer* **4237**, 12-21 (2006).

[19] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In Proceedings of the 14th conference on USENIX Security Symposium - Berkeley, CA, USA, 2005. USENIX Association **14**, 17-17 (2005).

[20] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM conference on Computer and communications security, CCS04, New York, NY, USA, ACM, 298-307 (2004).

[21] G. Novark and E. D. Berger. Dieharder: securing the heap. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, ACM Conference on Computer and Communications Security, ACM, 573-584 (2010).

[22] Y. Chang and T. Kuo. A management strategy for the reliability and performance improvement of mlc-based flash-memory storage systems. *IEEE Trans. Computers*, **60(3)**, 305-320 (2011).

[23] W. K. Robertson, C.r K. ugel, D. Mutz, and F. Valeur. Runtime detection of heap-based overflows. In LISA, USENIX, 51-60 (2003).

[24] W.m B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *J. Syst. Softw.* **83**, 2416-2430 (2010).

[25] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In Shubu Mukherjee and Kathryn S. McKinley, editors, ASPLOS, ACM, 156-164 (2004).

[26] S. Sardi .na and L. Padgham. A bdi agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi- Agent Systems, **23(1)**, 18-70 (2011).

[27] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, **18**, 268-299 (1996).

[28] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL93, New York, NY, USA, ACM, 260-272 (1993).

[29] J. Collard and M. Griebl. A precise fixpoint reaching definition analysis for arrays. In Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, LCPC99, London, UK, Springer-Verlag, 286-302 (2000). .

© 2015 NSP
Natural Sciences Publishing Cor.

[30] J. Knoop, O. Ruthing, and Bernhard Steffen. Lazy code motion. SIGPLAN Not. **39**,460-472 (2004).

[31] M. W. Hall, S. P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.* **27**, 662-731 (2005).

[32] Y. Kim and Y. Jun. Restructuring parallel programs for on-the-fly race detection. In Proceedings of the 5th International Conference on Parallel Computing Technologies, PaCT99, London, UK, 446-451 (1999). Springer-Verlag.

[33] A. Herz and K. Apinis. Class-modular, class-escape and points-to analysis for object-oriented languages. In Alwyn Goodloe and Suzette Person, editors, NASA Formal Methods, Lecture Notes in Computer Science, *Springer* **7226**, 106-119 (2012).

[34] B. Yang, M. Zhang, and Y. Zhang. Applying answer set programming to points-to analysis of object-oriented language. In De-Shuang Huang, Yong Gan, Vitoantonio Bevilacqua, and Juan Carlos Figueroa Garcya, editors, ICIC (1), Lecture Notes in Computer Science, *Springer* **6838**, 676-685 (2011).

[35] H. R. Nielson and F. Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In Torben. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, The Essence of Computation, Lecture Notes in Computer Science, *Springer* **2566**, 223-244 (2002).

[36] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, POPL, ACM, 14-25 (2004).

Mohamed A. El-Zawawy received: PhD in Computer Science from the University of Birmingham in 2007, M.Sc. in Computational Sciences in 2002 from Cairo University, and a BSc. in Computer Science in 1999 from Cairo University. Dr El-Zawawy is an associate professor of Computer Science at Faculty of Science, Cairo University Since 2014. Currently, Dr. El-Zawawy is on a sabbatical from Cairo University to College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia. During the year 2009, Dr. El-Zawawy held the position of an extra-ordinary senior research at the Institute of Cybernetics, Tallinn University of Technology, Estonia. Dr. El-Zawawy worked as a teaching assistant at Cairo University from 1999 to 2003 and latter at Birmingham University from 2003 to 2007. Dr. El-Zawawy worked as an assistant professor at Cairo University from 2007 to 2014. Dr. El-Zawawy is interested in static analysis, shape analysis, type systems, and semantics of programming languages.

© 2015 NSP
Natural Sciences Publishing Cor.