Detection of Probabilistic Dangling References in Multi-core Programs Using Proof-Supported Tools

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
maelzawawy@cu.edu.eg

Abstract. This paper presents a new technique for detection of probabilistic dangling references in multi-core programs. The technique has the form of a simply structured type system and provides a suitable framework for proof-carrying code applications like mobile code applications that have limited resources. The type derivation of each individual analysis serves as a proof for the correctness of the analysis. The type system is designed to analyze parallel programs with structured concurrent constructs: fork-join constructs, conditionally spawned cores, and parallel loops.

For a given program S, a probabilistic threshold p_{ms}, and a probabilistic reference analysis for S, if S is well-typed in our proposed type system then all computational paths with probabilities greater than or equal to p_{ms} will contain no dangling pointers at run time. The soundness of the presented type system is proved in this paper with respect to a probabilistic operational semantics to our model language.

1 Introduction

Multi-core (multithreading) [36] is one of the 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.

One of the vital and attractive attributes of multi-core programs is memory safety (mainly including dangling-references detection) [9]. 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. 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 [9] is a critical compiler analysis used to decide whether a given piece of code 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 [30] 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 [34] can aggressively take advantage of the prevalent ”maybe” case, especially in the presence of a probabilistic memory-safety (dangling-references) analysis.

Reference analysis [11,9,10,13,14,12] 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 [34,10] 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 presents a new approach for detecting dangling references in multi-core programs. The proposed technique is 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 [10]), the proposed technique decides 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 technique is flow-sensitive.

The algorithmic style [5,1], 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 [11,9,20,10,13,14,12,21]. 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 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 technique presented in this paper for memory safety of multi-core programs has the form of a type system. The key to the proposed approach 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.

$$\begin{aligned} & n \in \mathbb{Z},\ x \in \mathit{Var},\ \mathit{and}\ \oplus \in \{+,-,\times\} \\ e \in \mathit{Aexprs} &::= x \mid n \mid e_1 \oplus e_2 \\ b \in \mathit{Bexprs} &::= \mathit{true} \mid \mathit{false} \mid \neg b \mid e_1 = e_2 \mid e_1 \leq e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\ S \in \mathit{Stmts} &::= x := e \mid x := \&y \mid *x := e \mid x := *y \mid \mathit{skip} \mid S_1; S_2 \mid \mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f \mid \\ & \quad \mathit{while}\ b\ \mathit{do}\ S_t \mid \mathit{par}\{\{S_1\},\ldots,\{S_n\}\} \mid \mathit{par\text{-}if}\{(b_1,S_1),\ldots,(b_n,S_n)\} \mid \mathit{par\text{-}for}\{S\}. \end{aligned}$$

Fig. 1. Our programming language model

Figure 1 presents the 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 a probabilistic memory-safety analysis. 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 (dereferencing of *a* in line 5). However the first two paths are memory safe. The motivation of our work is to design a technique that for a program like this one and a probabilistic threshold (for example 0.4) decides whether the program paths with probabilities greater than or equal to 0.4 are memory safe. The desired technique is also required to associate its decision with a correctness proof.

```
1.  if (a > b)
2.     then a := &b
3.     else a := 2;
4.  par{
5.        {b := *a}
6.        {b := &c}
7.        };
```

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

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 multi-core programs.
2. A formal proof for the soundness of the proposed type system with respect to a probabilistic operational semantics.

Organization. The rest of the paper is organized as follows. Section 2 presents a type system for probabilistic memory safety of multi-core programs. This section also presents a formal correctness proof for the proposed type system with respect to a probabilistic operational semantics presented in an appendix to the paper. Future work and a survey of related work to memory safety (including the used of type systems in program analysis, and the analysis of multi-core programs) are discussed in Section 3.

2 Memory Safety

This section presents a new technique for memory safety (including dangling-references detection) of multi-core programs. 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 (Definition 4) 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.

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 [34,10] 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 [10] and reviewed in Definition 1. To build the memory safety analysis on robust ground, surely the underlying reference analysis has to be sound with respect to a robust semantics; in our case the operational semantics of this paper appendix. Suppose that for a statement S, the reference analysis associates a pre-reference type pts and a post-reference type pts', i.e. $S : pts \longrightarrow pts'$. The soundness has the intuition that if the execution of S from a state (γ, p) of type pts ends at a state (γ, p'), then this final state has to be of type pts'.

Definition 1. *1. $Addrs = \{x' \mid x \in Var\}$ and $Addrs_p = Addrs \times [0, 1]$.*

2. $Pre\text{-}PTS = \{pts \mid pts : Var \longrightarrow 2^{Addrs_p}$ s.t. $(y', p_1), (y', p_2) \in pts(x) \Longrightarrow p_1 = p_2\}$.

3. For $pts \in Pre\text{-}PTS$ and $x \in Var$, $\sum_{pts} x = \sum_{(z',p)\in pts(x)} p$.

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

4. *For* $pts \in$ *Pre-PTS and* $x \in Var$, $A_{pts}(x) = \{z' \mid \exists p > 0.\ (z', p) \in pts(x)\}$.
5. $PTS = \{pts \in \textit{Pre-PTS} \mid \forall x \in Var.\ \sum_{pts} x \leq 1\}$.
6. $pts \leq pts' \overset{\text{def}}{\iff} (\forall x, y \in Var.\ (y', p) \in pts(x) \Longrightarrow \exists p'.\ p \geq p' \,\&\, (y', p') \in pts'(x))$.
7. $\gamma \models pts \overset{\text{def}}{\iff} (\forall x \in Var.\ \gamma(x) \in Addrs \Longrightarrow \exists p > 0.\ (\gamma(x), p) \in pts(x))$.

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 and p_s is 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*
- $p_s \in [0, 1]$.

– $(pts, v, p_s) \leq (pts', v', p'_s) \overset{\text{def}}{\iff} pts \leq pts', v \supseteq v'$, *and* $p_s \geq p'_s \geq p_{ms}$.

– *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* $p_{ms} \leq p_s \leq p$.

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 $S : pts \rightarrow pts'$. 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 (Definition 5; has no execution paths with probability less than or equal to p_{ms}).

2.2 Inference Rules

The inference rules of our type system for probabilistic memory safety are as follows:

$$\frac{y \in v}{y : (x, pts, v) \rightarrow v \cup \{x\}}\ (y_1^m) \qquad \frac{\sum_{pts} y < p_{ms}}{y : (x, pts, v) \rightarrow v \setminus \{x\}}\ (y_2^m) \qquad \frac{}{n : (x, pts, v) \rightarrow v \setminus \{x\}}\ (n^m)$$

$$\frac{\forall y \in FV(e_1 \oplus e_2).\ \sum_{pts} y = 0}{e_1 \oplus e_2 : (x, pts, v) \rightarrow v \setminus \{x\}}\ (\oplus^m) \qquad \frac{x := e : pts \rightarrow pts' \quad e : (x, pts, v) \rightarrow v'}{x := e : (pts, v, p_s) \rightarrow (pts', v', p_s)}\ (:=^m)$$

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

$$\frac{x \in v \quad pts(x) = \{(z'_1, p_1), \ldots, (z'_n, p_n)\} \quad \forall z'_i \in A_{pts}(x).\ z_i := e : (pts, v, p_s) \rightarrow (pts_i, v', p_s)}{*x := e : (pts, v, p_s) \rightarrow (\Upsilon(pts, pts_1, \ldots, pts_n), v', p_s)}\ (* :=^m)$$

$$\frac{y \in v \quad pts(y) = \{(z'_1, p_1), \ldots, (z'_n, p_n)\} \quad \forall i.\ x := z_i : (pts, v, p_s) \rightarrow (pts_i, v', p_s)}{x := *y : (pts, v, p_s) \rightarrow (\Upsilon(pts, pts_1, \ldots, pts_n), v', p_s)}\ (:= *^m)$$

$$\frac{x := \&y : pts \rightarrow pts'}{x := \&y : (pts, v, p_s) \rightarrow (pts', v \cup \{x\}, p_s)}\ (:= \&^m) \qquad \frac{}{skip : (pts, v, p_s) \rightarrow (pts, v, p_s)}$$

$$\frac{S_1 : (pts, v, p_s) \rightarrow (pts'', v'', p''_s) \qquad S_2 : (pts'', v'', p''_s) \rightarrow (pts', v', p'_s)}{S_1; S_2 : (pts, v, p_s) \rightarrow (pts', v', p'_s)}\ (seq^m)$$

$$\frac{\forall y \in FV(b)(\sum_{pts} y = 0) \quad \begin{array}{ll} S_t : (pts, v, p_s) \rightarrow (pts_t, v_t, p_t) & p_{ms} \leq p_t \times p_{if} \\ S_f : (pts, v, p_s) \rightarrow (pts_f, v_f, p_f) & p_{ms} > p_s \times (1 - p_{if}) \end{array}}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (pts, v, p_s) \rightarrow (\Upsilon(pts_t, pts_f), v_t, p_t \times p_{if})}\ (if_1^m)$$

$$\frac{\forall y \in FV(b)(\sum_{pts} y = 0) \quad \begin{array}{ll} S_t : (pts, v, p_s) \rightarrow (pts_t, v_t, p_t) & p_{ms} > p_s \times p_{if} \\ S_f : (pts, v, p_s) \rightarrow (pts_f, v_f, p_f) & p_{ms} \leq p_f \times (1 - p_{if}) \end{array}}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (pts, v, p_s) \rightarrow (\Upsilon(pts_t, pts_f), v_f, p_f \times (1 - p_{if}))}\ (if_2^m)$$

$$\frac{\forall y \in FV(b)(\sum_{pts} y = 0) \quad \begin{array}{ll} S_t : (pts, v, p_s) \rightarrow (pts_t, v_t, p_t) & p_{ms} \leq p_t \times p_{if} \\ S_f : (pts, v, p_s) \rightarrow (pts_f, v_f, p_f) & p_{ms} \leq p_f \times (1 - p_{if}) \end{array}}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (pts, v, p_s) \rightarrow (\Upsilon(pts_t, pts_f), v_t \cap v_f, \min\{p_t \times p_{if}, p_f \times (1 - p_{if})\})}\ (if_3^m)$$

$$\frac{S_i : (\Psi(pts, \ldots, pts_j, \ldots \mid j \neq i), v \cap \cap_{j \neq i} v_j, \min\{p_s, p_j \mid j \neq i\}) \rightarrow (pts_i, v_i, p_i) \quad p_{ms} \leq \frac{\min_i p_i}{n!}}{par\{\{S_1\}, \ldots, \{S_n\}\} : (pts, v, p_s) \rightarrow (\Upsilon(pts_1, \ldots, pts_n), \cap_i v_i, \frac{\min_i p_i}{n!})}\ (par^m)$$

$$\frac{par\{\{\textit{if } b_1 \textit{ then } S_1 \textit{ else skip}\}, \ldots, \{\textit{if } b_n \textit{ then } S_n \textit{ else skip}\}\} : (pts, v, p_s) \rightarrow (pts', v', p'_s)}{par\text{-}if\{(b_1, S_1), \ldots, (b_n, S_n)\} : (pts, v, p_s) \rightarrow (pts', v', p'_s)}\ (par\text{-}if^m)$$

$$\frac{S : (\Psi(pts, pts'), v \cap v', \min\{p_s, p'_s\}) \rightarrow (pts', v', p'_s)}{par\text{-}for\{S\} : (pts, v, p_s) \rightarrow (pts', v', p'_s)}\ (par\text{-}for^m)$$

$$\frac{\begin{array}{c} \forall i \in [1, n], \forall y \in FV(b)(\sum_{pts_i} y = 0) \\ (pts_1, v_1, p_{s_1}) \overset{S_t}{\rightarrow} (pts_2, v_2, p_{s_2}) \overset{S_t}{\rightarrow} \ldots \overset{S_t}{\rightarrow} (pts_{n+1}, v_{n+1}, p_{s_{n+1}}) \end{array}}{\textit{while } b \textit{ do } S_t : (pts_1, v_1, p_{s_1}) \rightarrow (\Upsilon(pts_{n+1}), v_{n+1}, p_{s_{n+1}})}\ (whl^m)$$

$$\frac{\begin{array}{l} (pts'_1, v'_1, p'_{s_1}) \leq (pts_1, v_1, p_{s_1}) \\ S : (pts_1, v_1, p_{s_1}) \rightarrow (pts_2, v_2, p_{s_2}) \end{array} \quad (pts_2, v_2, p_{s_2}) \leq (pts'_2, v'_2, p'_{s_1})}{S : (pts'_1, v'_1, p'_{s_1}) \rightarrow (pts'_2, v'_2, p'_{s_2})}\ (csq^m)$$

Judgments produced by the type system above has two forms. The judgment of an arithmetic expression has the form $e : (x, pts, v) \rightarrow v'$. 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 at a post-state, then this post-state is of the post-type. This is proved in Theorem 1.

Comments on the inference rules above 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 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 integrations 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 1. As it is common with a probabilistic reference analysis [10], 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. $(pts, v, p_s) \leq (pts', v', p_s') \Longrightarrow (\forall (\gamma, p). \, (\gamma, p) \models_{p_{ms}} (pts, v, p_s) \Longrightarrow (\gamma, p) \models_{p_{ms}} (pts', v', p_s'))$.

Proof. It is straightforward to prove the second item. 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 1 y contains no address at the program point before the assignment statement. Hence $\gamma'(x)$ is not assured to contain

[3] The interested reader can check [10] for the details of calculating $\Upsilon(pts, pts_1, \ldots, pts_n)$.

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.

2.3 Soundness

For a statement S that has types in our probabilistic memory-safety 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* (Definition 5) state is a state that does not start any executions paths with probability less than p_{ms}. Theorem 1 also proves soundness of memory-safety 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
  - $p_t \times p_{if} \geq p_{ms} > p_s \times (1 - p_{if})$, 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) (Definition 5), $[\![b]\!]\gamma =$ true. Now by induction hypothesis S_t does not abort at (γ, p) because $S_t : (pts, v, p_s) \to (pts_t, v_t, p_t), (\gamma, p) \models_{p_{ms}} (pts, v, p_s)$, and S_t is positive at (γ, p) by 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 $S_t : (pts, v, p_s) \to$

(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 pretty much 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

$$p_{ms} \leq p_t \times p_{if} \leq p \times p_{if} \quad \text{and} \quad p_{ms} \leq p_f \times (1 - p_{if}) \leq p \times (1 - p_{if})$$

  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 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_t and $v_f \supseteq (v_t \cap v_f)$.
- The case of (par^m): (1) Suppose that $\theta : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ is a permutation. $(\gamma, p) \models_{p_{ms}} (pts, v, p_s)$ implies $(\gamma, p) \models_{p_{ms}} (\Psi(pts, \ldots, pts_j, \ldots \mid j \neq \theta(1)), v \cap \cap_{j \neq \theta(1)} v_j, \min\{p_s, p_j \mid j \neq \theta(1)\})$. Recall that $\Psi(pts, \ldots, pts_j, \ldots \mid j \neq \theta(1))$ is a lower bound for pts. By 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 $(\gamma_2, p_2) \models_{ms} (\Psi(pts, \ldots, pts_j, \ldots \mid j \neq \theta(2)), v \cap \cap_{j \neq \theta(2)} v_j, \min\{p_s, p_j \mid j \neq \theta(2)\})$. Therefore, clearly (1) is proved via a simple induction on n.

  (2) In this case the existence of a permutation $\theta : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ and $n + 1$ states $(\gamma, p) = (\gamma_1, p_1), \ldots, (\gamma_{n+1}, p_{n+1}) = (\gamma', p'')$ 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 $(\gamma_1, p_1) \models_{p_{ms}} (\Psi(pts, \ldots, pts_j, \ldots \mid$

$j \neq \theta(1)), v \cap \cap_{j\neq\theta(1)} v_j, \min\{p_s, p_j \mid j \neq \theta(1)\})$. Hence $(\gamma_2, p_2) \models_{p_{ms}} (pts_{\theta(1)}, v_{\theta(1)}, p_{\theta(1)})$ by the induction hypothesis. This implies $(\gamma_2, p_2) \models_{p_{ms}} (\Psi(pts, \ldots, pts_j, \ldots \mid j \neq \theta(2)), v \cap \cap_{j\neq\theta(2)} v_j, \min\{p_s, p_j \mid j \neq \theta(2)\})$. 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 $(\gamma', p'') \models_{p_{ms}} (\Upsilon(pts_1, \ldots, pts_n), \cap_i v_i, \min_i p_i)$. Hence because $p_{ms} \leq \frac{\min_i p_i}{n!}$, we get $(\gamma', p') \models_{p_{ms}} (pts', v', p') = (\Upsilon(pts_1, \ldots, pts_n), \cap_i v_i, \frac{\min_i p_i}{n!})$ 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 $par\{\{S\}_1, \ldots, \{S\}_n\} : (\gamma, p) \rightsquigarrow (\gamma', p')$. 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 $par\{\{S\}_1, \ldots, \{S\}_n\} : (pts, v, p_s) \rightarrow (pts', v', p'_s)$. Consequently by the soundness of (par^m), $(\gamma', p') \models_{p_{ms}} (pts', v', p'_s)$.

3 Related and Future Work

Related work includes security vulnerabilities, memory management, debugging and testing, garbage collection, failure masking, analysis of multi-core 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 [2] of the heap and buffering allocation requests [4]. However this classical approach is believed not to be very effective on 32-bit operating systems [33]. More recent work [29] 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 [6,15,16] prevent heap corruption via separating metadata from the heap, other managers [31] just recognize heap corruption.

Via simulation and multiple rewrites on run time, techniques for debugging and testing [27] discover errors of memory 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 [23].

The drawback of garbage collection [29], 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 [29] 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 [32] 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 [9]. 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 [26], concurrent static single assignment forms [35], reaching definitions [7], constant propagation [5], code motion [25], file safety [17], faulty function-calls [18]. 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 [22], race detection [24], reference analysis [9], and deadlock analysis [1].

The use of type systems in program analysis [11,9,20,10,13,14,12,21] 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 [28]. Type systems for program optimizations based live stack-heap and pointer analyses are presented in [11]. Constant folding, common subexpression elimination, and dead code elimination for *while* language as type systems are presented in [3].

In the area of denotational semantics, data structures and programs are mathematically represented by mathematical domains (sets) and maps between domains. For future work, we are interested in translating concepts of probabilistic memory-safety analysis to the side of denotational semantics [19,8]. This translation will facilitate theoretical studies about probabilistic memory-safety analysis. Obtained theoretical results can be then translated back to the side of data structures and programs.

References

1. Ahmad, F., Huang, H., Wang, X.-L.: Petri net modeling and deadlock analysis of parallel manufacturing processes with shared-resources. J. Syst. Softw. 83, 675–688 (2010)
2. Antonatos, S., Anagnostakis, K.G.: TAO: Protecting against hitlist worms using transparent address obfuscation. In: Leitold, H., Markatos, E.P. (eds.) CMS 2006. LNCS, vol. 4237, pp. 12–21. Springer, Heidelberg (2006)
3. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) POPL, pp. 14–25. ACM (2004)
4. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14th Conference on USENIX Security Symposium, Berkeley, CA, USA, vol. 14, p. 17. USENIX Association (2005)
5. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural constant propagation. SIGPLAN Not 39, 155–166 (2004)

6. Chang, Y.-H., Kuo, T.-W.: A management strategy for the reliability and performance improvement of mlc-based flash-memory storage systems. IEEE Trans. Computers 60(3), 305–320 (2011)
7. Collard, J.-F., Griebl, M.: A precise fixpoint reaching definition analysis for arrays. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, Springer, Heidelberg (2000)
8. El-Zawawy, M.A.: Semantic spaces in Priestley form. PhD thesis, University of Birmingham, UK (January 2007)
9. El-Zawawy, M.A.: Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part V. LNCS, vol. 6786, pp. 355–369. Springer, Heidelberg (2011)
10. El-Zawawy, M.A.: Probabilistic pointer analysis for multithreaded programs. ScienceAsia 37(4), 344–354 (2011)
11. El-Zawawy, M.A.: Program optimization based pointer analysis and live stack-heap analysis. International Journal of Computer Science Issues 8(2), 98–107 (2011)
12. El-Zawawy, M.A.: Abstraction analysis and certified flow and context sensitive points-to relation for distributed programs. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part IV. LNCS, vol. 7336, pp. 83–99. Springer, Heidelberg (2012)
13. El-Zawawy, M.A.: Dead code elimination based pointer analysis for multithreaded programs. Journal of the Egyptian Mathematical Society 20(1), 28–37 (2012)
14. El-Zawawy, M.A.: Heap slicing using type systems. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part III. LNCS, vol. 7335, pp. 592–606. Springer, Heidelberg (2012)
15. El-Zawawy, M.A.: Recognition of logically related regions based heap abstraction. Journal of the Egyptian Mathematical Society 20(2) (2012)
16. El-Zawawy, M.A.: Frequent statement and de-reference elimination for distributed programs. In: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Quang, N.H., Taniar, D., Apduhan, B.O., Gervasi, O. (eds.) ICCSA 2013. LNCS, vol. 7975, pp. 82–97. Springer, Heidelberg (2013)
17. El-Zawawy, M.A., Daoud, N.M.: M. Daoud. Dynamic verification for file safety of multithreaded programs. IJCSNS International Journal of Computer Science and Network Security 12(5), 14–20 (2012)
18. El-Zawawy, M.A., Daoud, N.M.: New error-recovery techniques for faulty-calls of functions. Computer and Information Science 5(3), 67–75 (2012)
19. El-Zawawy, M.A., Jung, A.: Priestley duality for strong proximity lattices. Electr. Notes Theor. Comput. Sci. 158, 199–217 (2006)
20. El-Zawawy, M.A., Partial, H.A.N.: redundancy elimination for multi-threaded programs. IJCSNS International Journal of Computer Science and Network Security 11(10), 127–133 (2011)
21. El-Zawawy, M.A., Nayel, H.A.: Type systems based data race detector. IJCSNS International Journal of Computer Science and Network Security 5(4), 53–60 (2012)
22. Hall, M.W., Amarasinghe, S.P., Murphy, B.R., Liao, S.-W., Lam, M.S.: Interprocedural parallelization analysis in suif. ACM Trans. Program. Lang. Syst. 27, 662–731 (2005)
23. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. In: Mukherjee, S., McKinley, K.S. (eds.) ASPLOS, pp. 156–164. ACM (2004)
24. Kim, Y.-C., Jun, Y.-K.: Restructuring parallel programs for on-the-fly race detection. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662, pp. 446–451. Springer, Heidelberg (1999)
25. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. SIGPLAN Not 39, 460–472 (2004)
26. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst. 18, 268–299 (1996)

27. Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. J. Syst. Softw. 83, 2416–2430 (2010)
28. Riis Nielson, H., Nielson, F.: Flow logic: A multi-paradigmatic approach to static analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
29. Novark, G., Berger, E.D.: Dieharder: securing the heap. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM Conference on Computer and Communications Security, pp. 573–584. ACM (2010)
30. Novark, G., Berger, E.D., Zorn, B.G.: Exterminator: Automatically correcting memory errors with high probability. Commun. ACM 51, 87–95 (2008)
31. Robertson, W.K., Krügel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: LISA, pp. 51–60. USENIX (2003)
32. Sardiña, S., Padgham, L.: A bdi agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems 23(1), 18–70 (2011)
33. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, pp. 298–307. ACM, New York (2004)
34. Da Silva, J., Steffan, J.G.: A probabilistic pointer analysis for speculative optimizations. In: Shen, J.P., Martonosi, M. (eds.) ASPLOS, pp. 416–425. ACM (2006)
35. Srinivasan, H., Hook, J., Wolfe, M.: Static single assignment for explicitly parallel programs. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993, pp. 260–272. ACM, New York (1993)
36. Ungerer, T., Robič, B., Šilc, J.: A survey of processors with explicit multithreading. ACM Comput. Surv. 35, 29–63 (2003)

Appendix: Probabilistic Operational Semantics

This appendix reviews and augments a probabilistic operational semantics that we presented in [10] 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*$\{\{S_1\},\dots,\{S_n\}\}$ (fork-join), *par-if*$\{(b_1,S_1),\dots,(b_n,S_n)\}$ (conditionally spawned threads), and *par-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-if*$\{(b_1,S_1),\dots,(b_n,S_n)\}$ 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-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. The following is the formal definition for computational states:

Definition 3. *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 pointers.

$$\llbracket n \rrbracket\gamma = n \quad \llbracket \&x \rrbracket\gamma = x' \quad \llbracket x \rrbracket\gamma = \gamma(x) \quad \llbracket \mathit{true} \rrbracket\gamma = \mathit{true} \quad \llbracket \mathit{false} \rrbracket\gamma = \mathit{false}$$

$$\llbracket *x \rrbracket\gamma = \begin{cases} \gamma(y) & \text{if } \gamma(x) = y', \\ ! & \text{otherwise.} \end{cases} \qquad \llbracket e_1 \oplus e_2 \rrbracket\gamma = \begin{cases} \llbracket e_1 \rrbracket\gamma \oplus \llbracket e_2 \rrbracket\gamma & \text{if } \llbracket e_1 \rrbracket\gamma, \llbracket e_2 \rrbracket\gamma \in \mathbb{Z}, \\ ! & \text{otherwise.} \end{cases}$$

$$\llbracket \neg A \rrbracket\gamma = \begin{cases} \neg(\llbracket A \rrbracket\gamma) & \text{if } \llbracket A \rrbracket\gamma \in \{\mathit{true}, \mathit{false}\}, \\ ! & \text{otherwise.} \end{cases} \qquad \llbracket e_1 = e_2 \rrbracket\gamma = \begin{cases} ! & \text{if } \llbracket e_1 \rrbracket\gamma = ! \text{ or } \llbracket e_2 \rrbracket\gamma = !, \\ \mathit{true} & \text{if } \llbracket e_1 \rrbracket\gamma = \llbracket e_2 \rrbracket\gamma \neq !, \\ \mathit{false} & \text{otherwise.} \end{cases}$$

$$\llbracket e_1 \leq e_2 \rrbracket\gamma = \begin{cases} ! & \text{if } \llbracket e_1 \rrbracket\gamma \notin \mathbb{Z} \text{ or } \llbracket e_2 \rrbracket\gamma \notin \mathbb{Z}, \\ \llbracket e_1 \rrbracket\gamma \leq \llbracket e_2 \rrbracket\gamma & \text{otherwise.} \end{cases}$$

$$\text{For } \diamond \in \{\wedge, \vee\},\ \llbracket b_1 \diamond b_2 \rrbracket\gamma = \begin{cases} ! & \text{if } \llbracket b_1 \rrbracket\gamma = ! \text{ or } \llbracket b_2 \rrbracket\gamma = !, \\ \llbracket b_1 \rrbracket\gamma \diamond \llbracket b_2 \rrbracket\gamma & \text{otherwise.} \end{cases}$$

The following are the inference rules of our probabilistic operational semantics (transition relation).

$$\frac{\llbracket e \rrbracket\gamma = !}{x := e : (\gamma,p) \rightsquigarrow \mathit{abort}} \quad \frac{\llbracket e \rrbracket\gamma \neq !}{x := e : (\gamma,p) \rightsquigarrow (\gamma[x \mapsto \llbracket e \rrbracket\gamma], p)} \quad \frac{\gamma(x) = z' \quad z := e : (\gamma,p) \rightsquigarrow \mathit{state}}{*x := e : (\gamma,p) \rightsquigarrow \mathit{state}}$$

$$\frac{\gamma(x) \notin \mathit{Addrs}}{*x := e : (\gamma,p) \rightsquigarrow \mathit{abort}} \quad \frac{}{x := \&y : (\gamma,p) \rightsquigarrow (\gamma[x \mapsto y'], p)} \quad \frac{\gamma(y) = z' \quad x := z : (\gamma,p) \rightsquigarrow (\gamma',p)}{x := *y : (\gamma,p) \rightsquigarrow (\gamma',p)}$$

$$\frac{\gamma(y) \notin \mathit{Addrs}}{x := *y : (\gamma,p) \rightsquigarrow \mathit{abort}} \quad \frac{}{\mathit{skip} : (\gamma,p) \rightsquigarrow (\gamma,p)} \quad \frac{S_1 : (\gamma,p) \rightsquigarrow \mathit{abort}}{S_1; S_2 : (\gamma,p) \rightsquigarrow \mathit{abort}}$$

$$\frac{S_1 : (\gamma,p) \rightsquigarrow (\gamma'',p'') \quad S_2 : (\gamma'',p'') \rightsquigarrow \mathit{state}}{S_1; S_2 : (\gamma,p) \rightsquigarrow \mathit{state}} \quad \frac{\llbracket b \rrbracket\gamma = !}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : (\gamma,p) \rightsquigarrow \mathit{abort}}$$

$$\frac{\llbracket b \rrbracket\gamma = \mathit{true} \quad S_t : (\gamma,p) \rightsquigarrow \mathit{abort}}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : (\gamma,p) \rightsquigarrow \mathit{abort}} \quad \frac{\llbracket b \rrbracket\gamma = \mathit{true} \quad S_t : (\gamma,p) \rightsquigarrow (\gamma',p')}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : (\gamma,p) \rightsquigarrow (\gamma', p_{if} \times p')}$$

$$\frac{\llbracket b \rrbracket\gamma = \mathit{false} \quad S_f : (\gamma,p) \rightsquigarrow \mathit{abort}}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : (\gamma,p) \rightsquigarrow \mathit{abort}} \quad \frac{\llbracket b \rrbracket\gamma = \mathit{false} \quad S_f : (\gamma,p) \rightsquigarrow (\gamma',p')}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : (\gamma,p) \rightsquigarrow (\gamma', (1 - p_{if}) \times p')}$$

$$\frac{\llbracket b \rrbracket\gamma = !}{\mathit{while}\ b\ \mathit{do}\ S_t : (\gamma,p) \rightsquigarrow \mathit{abort}} \quad \frac{\llbracket b \rrbracket\gamma = \mathit{false}}{\mathit{while}\ b\ \mathit{do}\ S_t : (\gamma,p) \rightsquigarrow (\gamma,p)} \quad \frac{\llbracket b \rrbracket\gamma = \mathit{true} \quad S : (\gamma,p) \rightsquigarrow \mathit{abort}}{\mathit{while}\ b\ \mathit{do}\ S_t : (\gamma,p) \rightsquigarrow \mathit{abort}}$$

$$\frac{\llbracket b \rrbracket\gamma = \mathit{true} \quad S : (\gamma,p) \rightsquigarrow (\gamma'',p'') \quad \mathit{while}\ b\ \mathit{do}\ S_t : (\gamma'',p'') \rightsquigarrow \mathit{state}}{\mathit{while}\ b\ \mathit{do}\ S_t : (\gamma,p) \rightsquigarrow \mathit{state}}$$

- **Fork-join:**

$$\frac{}{\mathit{par}\{\{S_1\},\ldots,\{S_n\}\} : (\gamma,p) \rightsquigarrow (\gamma', \frac{p'}{n!})}\,\dagger \qquad \frac{}{\mathit{par}\{\{S_1\},\ldots,\{S_n\}\} : (\gamma,p) \rightsquigarrow \mathit{abort}}\,\ddagger$$

† there exist a permutation $\theta : \{1,\ldots,n\} \to \{1,\ldots,n\}$ and $n+1$ states $(\gamma,p) = (\gamma_1,p_1),\ldots,(\gamma_{n+1},p_{n+1}) = (\gamma',p')$ such that for every $1 \leq i \leq n$, $S_{\theta(i)} : (\gamma_i,p_i) \rightsquigarrow (\gamma_{i+1},p_{i+1})$.

‡ there exist m such that $1 \leq m \leq n$, a one-to-one map $\beta : \{1,\ldots,m\} \to \{1,\ldots,n\}$, and $m+1$ states $(\gamma,p) = (\gamma_1,p_1),\ldots,(\gamma_{m+1},p_{m+1}) = \mathit{abort}$ such that for every $1 \leq i \leq m$, $S_{\beta(i)} : (\gamma_i,p_i) \rightsquigarrow (\gamma_{i+1},p_{i+1})$.

- **Conditionally spawned threads:**

$$\frac{\mathit{par}\{\{\mathit{if}\ b_1\ \mathit{then}\ S_1\ \mathit{else\ skip}\},\ldots,\{\mathit{if}\ b_n\ \mathit{then}\ S_n\ \mathit{else\ skip}\}\} : (\gamma,p) \rightsquigarrow (\gamma',p')}{\mathit{par\text{-}if}\{(b_1,S_1),\ldots,(b_n,S_n)\} : (\gamma,p) \rightsquigarrow (\gamma',p')}$$

$$\frac{\mathit{par}\{\{\mathit{if}\ b_1\ \mathit{then}\ S_1\ \mathit{else\ skip}\},\ldots,\{\mathit{if}\ b_n\ \mathit{then}\ S_n\ \mathit{else\ skip}\}\} : (\gamma,p) \rightsquigarrow \mathit{abort}}{\mathit{par\text{-}if}\{(b_1,S_1),\ldots,(b_n,S_n)\} : (\gamma,p) \rightsquigarrow \mathit{abort}}$$

- **Parallel loops:**

$$\frac{\exists n.\ \mathit{par}\{\{S\}_1,\ldots,\{S\}_n\} : (\gamma,p) \rightsquigarrow (\gamma',p')}{\mathit{par\text{-}for}\{S\} : (\gamma,p) \rightsquigarrow (\gamma',p')} \qquad \frac{\exists n.\ \mathit{par}\{\{S\}_1,\ldots,\{S\}_n\} : (\gamma,p) \rightsquigarrow \mathit{abort}}{\mathit{par\text{-}for}\{S\} : (\gamma,p) \rightsquigarrow \mathit{abort}}$$

The following definitions introduce terminologies that are used above to discuss and prove soundness of our proposed type system for probabilistic memory safety.

Definition 4. *For a statement S, a judgement of the form $S : (\gamma,p) \to (\gamma',p')$ is described as a computation (or an execution) path. The quantity p' is the probability of this execution path.*

Definition 5. *Suppose that $S : (pts,v,p_s) \to (pts',v',p'_s)$. 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 $S : (pts,v,p_s) \to (pts',v',p'_s)$, 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 the following lemma which is used in the soundness proof above:

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