Probabilistic pointer analysis for multithreaded programs

Mohamed A. El-Zawawy

College of Computer and Information Sciences, Al-Imam M.I.-S.I. University, Riyadh 11432, Kingdom of Saudi Arabia
Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

e-mail: maelzawawy@cu.edu.eg

Received 13 Apr 2011
Accepted 3 Nov 2011

ABSTRACT: The use of pointers and data-structures based on pointers results in circular memory references that are interpreted by a vital compiler analysis, namely pointer analysis. For a pair of memory references at a program point, a typical pointer analysis specifies if the points-to relation between them may exist, definitely does not exist, or definitely exists. The “may be” case, which describes the points-to relation for most of the pairs, cannot be dealt with by most compiler optimizations. This is so to guarantee the soundness of these optimizations. However, the “may be” case can be capitalized by the modern class of speculative optimizations if the probability that two memory references alias can be measured. Focusing on multithreading, a prevailing technique of programming, this paper presents a new flow-sensitive technique for probabilistic pointer analysis of multithreaded programs. The proposed technique has the form of a type system and calculates the probability of every points-to relation at each program point. The key to our approach is to calculate the points-to information via a post-type derivation. The use of type systems has the advantage of associating each analysis results with a justification (proof) for the correctness of the results. This justification has the form of a type derivation and is very much required in applications like certified code.

KEYWORDS: static analysis, speculative optimizations, probabilistic alias analysis, distributed programs, semantics of multithreaded programs, type systems

INTRODUCTION

Multithreading is enjoying a growing interest and becoming a prevailing technique of programming. The use of multiple threads has several advantages: (a) concealing the delay of commands like reading from a secondary storage, (b) improving the action of programs, like web servers, that run on multi-processors, (c) building complex systems for user interface, (d) simplifying the process of organizing huge systems of code. However, the static analysis of multithreaded programs is intricate due to the possible interaction between multiple threads.

Among effective tools of modern programming languages are pointers, which empower coding intricate data structures. Not only does the uncertainty of pointer values at compile time complicate analysis of programs, but also retard program compilation by compelling the program optimization and analysis to be conservative. The pointer analysis of programs is a challenging problem in which researchers have trade space and time costs for precision. However, binary decision diagrams have been used to ease the difficulty of this trade off.

At any program point and for every pair of memory references, a traditional pointer analysis figures out whether one of these references may point to, definitely points to, or definitely does point to the other reference. For most of pairs of the memory references the points-to relation is of type “may be”. This is specially the case for techniques that prefer speed over accuracy. Traditional optimization techniques are not robust enough to treat the cases “may be” and “definitely” differently. The idea behind speculative optimization is to subsidize the “maybe” case, specially if the probability of “maybe” cab be specifically quantified.

Pointer analysis is among the most important program analyses of multithreaded programs. Pointer analysis of multithreaded programs has many applications; (a) mechanical binding of file operations that are in abeyance, (b) optimizations for memory systems like prefetching and relocating remote data calculations, (c) equipping compilers with necessary information for optimizations like common subexpression elimination and induction variable elimination, and
(d) relaxing the process of developing complex tools for software engineering like program slicers and race detectors.

This paper presents a new technique for pointer analysis of multithreaded programs. The proposed technique is probabilistic; it anticipates precisely for every program point the probability of every points-to relation. Building on a type system, the proposed approach is control-flow-sensitive. The key to the presented analysis is to calculate probabilities for points-to relations through the compositional use of inference rules of a type system. The proposed technique associates with every analysis a proof (type derivation) for the correctness of the analysis.

Among techniques to approach static analysis of programs is the algorithmic style. However, the proposed technique of this paper has the form of a type system. The algorithmic style does not reflect how the analysis results are obtained because it works on control-flow graphs of programs; not on phrase structures as in the case of type systems. Therefore the type-system approach is perfect for applications that require to handle a justifications (proof) for correctness of analysis results together with each individual analysis. An example of such applications is certified code. What contributes to suitability of type-system tools to produce such proofs is the relative simplicity of its inference rules. This simplicity is a much appreciated property in applications that require justifications. In the type-system approach, the justifications take the form of type derivations.

Motivation

Fig. 1 presents a motivating example of our work. This example uses three pointer variables \(a, b,\) and \(c\) that point at two variables \(c\) and \(d\). We suppose that (i) the condition of the \texttt{if} statement at line 2 is true with probability 0.6, (ii) the condition of the \texttt{if} statement at line 9 is true with probability 0.5, and (iii) the loop at line 8 iterates at most 100 times. These statistical and probabilistic information can be obtained using edge profiling. In absence of edge profiling, heuristics can be used. The work presented in this paper aims at introducing a probabilistic pointer analysis that produces results like that in Table 1. The aim is also to associate each such pointer-analysis result with a justification for the correctness of the result. This justification takes the form of a type derivation in our proposed technique which is based on a type system.

Contributions

Contributions of this paper are the following:

1. A new pointer analysis technique, that is probabilistic and flow-sensitive, for multithreaded programs.


Organization

The remainder of the paper is organized in three sections as follows. The first of these sections presents a simple language equipped with parallel and pointer constructs. This section also presents a new probabilistic operational semantics for the constructs of the language that we study. The second of these sections introduces a type system to carry probabilistic pointer analysis of parallel programs. This involves introducing suitable notions for pointer types, a subtyping relation, and a detailed proof for the soundness of the proposed type system w.r.t. the semantics presented in the paper. Related work is reviewed in the last section of the paper.

![Fig. 1 A motivating example.](image-url)
\[ n \in \mathbb{Z}, \ x \in \text{Var}, \ \text{and} \ \oplus \in \{+,-,\times\} \]

\[ e \in \text{Aexprs} ::= x \mid n \mid e_1 \oplus e_2 \]
\[ b \in \text{Bexprs} ::= \text{true} \mid \text{false} \mid \neg b \mid e_1 = e_2 \mid e_1 \leq e_2 \mid b_1 \land b_2 \mid b_1 \lor b_2 \]
\[ S \in \text{Stmts} ::= x := e \mid x := \& y \mid \ast x := e \mid x := \ast y \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_i \text{ else } S_f \mid \text{while } b \text{ do } S_i \mid \text{par}\{(S_1), \ldots, (S_n)\} \mid \text{par-if}\{(b_1, S_1), \ldots, (b_n, S_n)\} \mid \text{par-for}\{S\}. \]

**Fig. 2** The programming language.

**PROBABILISTIC OPERATIONAL SEMANTICS**

This section presents the programming language we study and a probabilistic pointer analysis for its constructs. We build our language (Fig. 2) on the *while* language, originally presented by Hoare in 1969, by equipping it with commands dealing with pointers and parallel computations. The parallel concepts deal with in our language are fork-join, conditionally spawned threads, and parallel loops. These concepts are represented by the commands *par*, *par-if*, and *par-for*, respectively. States of our proposed operational semantics are defined as follows:

**Definition 1** 1. \( \text{Addr} = \{x' \mid x \in \text{Var}\} \) and \( \text{Val} = \mathbb{Z} \cup \text{Addr} \).
2. \( \gamma \in \Gamma = \text{Var} \rightarrow \text{Val} \).
3. \( \text{state} \in \text{States} = \{(\gamma, p) \mid \gamma \in \Gamma \land p \in [0,1]\} \cup \{\text{abort}\} \).

Typically, a state is a function from the set of variables to the set of values (integers). In our work, we enrich the set of values with a set of symbolic addresses and enrich each state with a probabilistic value that is meant to measure the probability with which this state is reached. The *abort* state is there to capture any case of de-reference that is unsafe; i.e., de-referencing a variable that contains no address. We assume that the set of program variables, \( \text{Var} \), is finite.

Except that arithmetic and Boolean operations are not allowed on pointers, the semantics of arithmetic and Boolean expressions are defined as usual (Fig. 3). The inference rules of Fig. 4 define the transition relation \( \rightsquigarrow \) of our operational semantics.

We notice that none of the assignment statements changes the probability component of a given pre-state to produce the corresponding post-state. The symbol \( p_f \) used in the inference rules of the *if* statement denotes a number in \([0,1]\) and measures the probability that the condition of the statement is true. This probabilistic information can be obtained using edge profiling\(^{15-18}\). In absence of edge profiling, heuristics can be used.

The *par* command is the main parallel concept. This concept is also known as cobegin-coend or fork-join. The execution of this command amounts to starting concurrently executing the threads of the command at the beginning of the construct and then to wait for the completion of these executions at the end of the construct. Then the subsequent command can be executed. The inference rule (*par-sem*) approximates the execution method of the *par* command. The probability \( p' \) in the rule (*par-sem*) is multiplied by \( 1/n! \) (not by \( 1/n \) as the reader may expect) because the permutation \( \theta \) finds one of the \( n! \) ways in which the threads can be sorted and then executed. As an example, the reader may consider applying the rule *par-sem* when \( n = 3 \) and the threads are \( S_1 : a := b + c, S_2 : b := a \times c, \) and \( S_3 : c := a - b \). The semantics of *par-if* and *par-for* commands are defined using that of the *par* command.

**PROBABILISTIC POINTER ANALYSIS**

The purpose of a typical pointer analysis is to assign to every program point a points-to function. The domain of this function is the set of all pairs of pointers and the codomain is the set \{definitely exists, definitely does not exist, may exist\}. The codomain describes the points-to relation between pairs of memory references. For most of the pointer pairs, the points-to relation is “may exist”. This is specially the case for techniques of pointer analysis that give priority for speed over efficiency. The common drawback for most existing program optimization techniques is that they cannot treat the “maybe” and “definitely does not exist” cases differently. Speculative optimizations are meant to overcome this disadvantage via working on the result of analyses that can measure the probability that a points-to relation exist between two pointers.

This section presents a new technique for probabilistic pointer analysis for multithreaded programs. The technique has the form of a type system and its goal is to accurately calculate the likelihood at each program point for every points-to relation. The advantages of the proposed technique include the
simplicity of the inference rules of the type system and that no dependence profile information (information describing dependencies between threads) is required. Dependence profile information, required by some multithreading techniques like Ref. 19, is expensive to get. The proposed technique is flow-sensitive.

The key to our technique is to calculate points-to probabilities via a post type derivation for a given program using the bottom points-to type as a pre type.

The following definition presents some notations that are used in the rest of the paper.

**Definition 2.** 1. \( Addrs = \{ x' | x \in Var \} \) and \( Addrs_p = Addrs \times [0, 1] \).
2. Pre-PTS = \{ \{pts \mid pts : Var \rightarrow 2^{Addrs_p} \text{ s.t. } \forall y \in Var. (y', p_1), (y', p_2) \in pts(x) \implies p_1 = p_2 \} \}.
3. For \( pts \in \text{Pre-PTS} \) and \( x \in Var, \sum_{pts \in \text{Pre-PTS}} x = \sum_{(z', p) \in pts(x)} p. \)
4. For every \( pts \in \text{Pre-PTS} \) and \( x \in Var, A_{pts}(x) = \{ z' | \exists p > 0. (z', p) \in pts(x) \}. \)
5. For \( A \in Addrs_p, pts \in \text{Pre-PTS}, \) and \( 0 \leq q \leq 1,\)
   \( (a) A \times q = \{ (y', p \times q) | (y', p) \in A \}. \)
   \( (b) pts \times q \) is the function defined by \( (pts \times q)(x) = pts(x) \times q. \)

We note that the set of symbolic addresses \( Addrs \) is enriched with probabilities to form the set \( Addrs_p \). In line with real situations, the condition on the elements of \( \text{Pre-PTS} \) excludes maps that assign the same address for a variable with two different probabilities. The notation \( \sum_{pts} x \) denotes the probability that the variables \( x \) has an address with respect to \( pts \). The notation \( A_{pts}(x) \) denotes the set of addresses that have a non-zero probability to get into \( x \). The multiplication operations of Definition 2.5 are necessary to join many points-to types (each with a different probability) into one type.

A formalization for the concepts of the set of points-to types \( PTS \), the subtyping relation \( \leq \), and the relation \( \models \subseteq \Gamma \times PTS \) are in the subsequent definition.

**Definition 3.** 1. \( PTS = \{ pts \in \text{Pre-PTS} \mid \forall x \in Var. \sum_{pts \in \text{Pre-PTS}} x \leq 1 \}. \)
2. \( pts \leq pts' \triangleq \forall x. A_{pts}(x) \subseteq A_{pts'}(x) \).
3. \( pts = pts' \triangleq \forall x. A_{pts}(x) = A_{pts'}(x) \).
4. \( (\gamma, p) \models pts \triangleq (\forall x. \gamma(x) \in Addrs \implies \exists q > 0. (\gamma(x), q) \in pts(x)) \).

A way to calculate an upper bound for a set of \( n \) points-to types is introduced in the following definition.

**Definition 4.** Suppose \( pts_1, \ldots, pts_n \) is a sequence of \( n \) points-to types and \( 0 \leq q_1, \ldots, q_n \leq 1 \) is a sequence of \( n \) numbers whose sum is less than or equal to 1. Then \( \nabla((pts_1, q_1), \ldots, (pts_n, q_n)) : Var \rightarrow 2^{Addrs_p} \) is the function defined by:
\[
\nabla((pts_1, q_1), \ldots, (pts_n, q_n))(x) = \{ (z', p) | (\exists i. z' \in A_{pts_i}(x)) \wedge (p = \sum_{(z', p_k) \in pts_k(x)} q_k \times p_k) \}.
\]

We note that the order of the points-to lattice is the point-wise inclusion. However, probabilities are implicitly taken into account in the definition of supremum which is based on Definition 4. Letting the probabilities of points-to relations be involved in the definition of the order relation complicates the formula of calculating the lattice supremum. Besides that this complication is not desirable, introducing probabilities apparently does not improve the type system results. The definition for \( (\gamma, p) \models pts \) makes sure that a variable that has an address under \( \gamma \) is allowed (positive probability) to contain the same address.
The following lemma proves that the upper bound of the previous definition is indeed a points-to type.

**Lemma 1** The map $\nabla((pts_1, q_1), \ldots, (pts_n, q_n))$ of previous definition is a points-to type.

**Proof:** Suppose that $\nabla((pts_1, q_1), \ldots, (pts_n, q_n))(x) = \{(z_1', t_1), (z_2', t_2), \ldots, (z_m', t_m)\}$. To show the required we need to show that (a) $0 \leq t_i \leq 1$ and (b) $0 \leq \sum_i t_i \leq 1$. Since (b) implies (a), it is enough to show (b). Suppose that $\forall 1 \leq i \leq n, pts_i(x) =$
\[
\begin{array}{ccc}
& pts_1 & pts_2 & \ldots & pts_n \\
z_1' & p_{11} & p_{12} & \ldots & p_{1n} \\
z_2' & p_{21} & p_{22} & \ldots & p_{2n} \\
& & & \vdots & \\
z_m' & p_{m1} & p_{m2} & \ldots & p_{mn} \\
\end{array}
\]

\[
\left( \begin{array}{c}
q_1 \\
q_2 \\
\vdots \\
q_n \\
\end{array} \right) = \left( \begin{array}{c}
t_1 \\
t_2 \\
\vdots \\
t_m \\
\end{array} \right)
\]

Fig. 5 A matrix multiplication needed in the proof of Lemma 1.

\{ (z_1', p_{1i}), (z_2', p_{2i}), \ldots, (z_m', p_{mi}) \}, \text{ where } \forall 1 \leq j \leq m, \ p_{ji} = 0 \text{ if } z_j \notin A_{pts}(x). \text{ Then according to Definition 4 the values } t_1, \ldots, t_m \text{ can be equivalently calculated by the matrix multiplication of Fig. 5.}

Then:

\[\Sigma_i t_i = (\Sigma_i q_i \times p_{1i}) + (\Sigma_i q_i \times p_{2i}) + \ldots + (\Sigma_i q_i \times p_{mi}) = (q_1 \times \Sigma_i p_{1i}) + (q_2 \times \Sigma_i p_{2i}) + \ldots + (q_n \times \Sigma_i p_{ni}).\]

We note that \(\forall j, 0 \leq i \leq p_{ij} \leq 1\) by definition of \(pts_j\) and \(\forall j, 0 \leq q_j \leq 1\). Therefore this last summation is less than 1. \(\Box\)

**Lemma 2** Suppose that \(A = \{pts_1, \ldots, pts_n\} \subseteq PTS\) and \(pts = \nabla((pts_1, 1/n), \ldots, (pts_n, 1/n)).\) Then with respect to definitions of \(\nabla\), the subtyping, and equality relations introduced in Definitions 3.2, 3.3, and 4, respectively, the set \(PTS\) is a complete lattice where \(\forall A = pts\).

**Proof:** Clearly \(pts\) is an upper bound for \(A\). Moreover for every \(x, A_{pts}(x) = \bigcup_i A_{pts_i}(x)\). Therefore \(pts\) is the least upper bound of \(A\). \(\Box\)

The inference rules of our proposed type system for probabilistic pointer analysis are shown in Fig. 6.

The judgement of an arithmetic expression has the form \(e : pts \rightarrow A\). The intuition (Lemma 3) of this judgement is that any address that \(e\) evaluates to in a state of type \(pts\) is included in the set \(A\) as the second component of a pair whose first component is a non-zero probability. The judgement for a statement \(S\) has the form \(S : pts \rightarrow pts'\) and guarantees that if the execution of \(S\) in a state of type \(pts\) terminates then the reached state is of type \(pts'\). This is proved in Theorem 1.

Concerning the inference rules, some comments are in order. In the rule \((:= \text{prob})\), since there are \(n\) possible ways to modify \(x\), the post-type is calculated from the pre-type by assigning \(x\) its value according to the upper bound of the \(n\) ways. The upper bound is considered to enable the analysis to cover all possible executions of the statement. In the rule \((\ast := \text{prob})\), there are \(n\) variables, \(\{z_1, \ldots, z_n\}\), that have a chance of getting modified. This produces \(n\) post-types in the pre conditions of the rule. Therefore the post-type is calculated from the pre-type by assigning each of the \(n\) variables its image under the upper bound of the \(n\) post-types. In the rule \((if^\text{prob})\), \(p\) is the probability that the condition of the \(if\) statement is true. The rule \((par\text{prob})\) has this form in order for the analysis result of any thread \(S_1\) of the \(par\) statement to consider the fact that any other thread may have been executed before the thread in hand. As it is the case in the operational semantics, the rules for conditionally spawned threads \((par-if^\text{prob})\) and parallel loops \((par-for^\text{prob})\) are built on the rule \((par\text{prob})\). In the following we give an example for the application of the rule \((par\text{prob})\). Let:

- \(S_1 : if \ b_1 then x := \& y \ else x := 5,\)
- \(S_2 : x := \& z;\)
- \(S_{par} : par\{\{S_1\}, \{S_2\}\},\)
- \(pts = \{t \mapsto 0 \mid t \in \text{Var}\},\)
- \(pts_1 = \{x \mapsto \{y', 0.4\}, t \mapsto 0 \mid x \neq t \in \text{Var}\},\)
- \(pts_2 = \{x \mapsto \{z', 1\}, t \mapsto 0 \mid x \neq t \in \text{Var}\}\)

We suppose that the condition \(b_1\) in \(S_1\) succeeds with probability 0.4. Then we have the following:

\[
\begin{align*}
\nabla((pts_1, 1/2), (pts_2, 1/2)) &= \{x \mapsto \{y', 0.25\}, t \mapsto 0 \mid x \neq t \in \text{Var}\}, \\
\nabla((pts_1, 1/2), (pts_2, 1/2)) &= \{x \mapsto \{z', 0.5\}, t \mapsto 0 \mid x \neq t \in \text{Var}\}.
\end{align*}
\]

Clearly, \(S_1 : \nabla((pts_1, 1/2), (pts_2, 1/2)) \rightarrow pts_1\) and \(S_2 : \nabla((pts_1, 1/2), (pts_2, 1/2)) \rightarrow pts_2\). These two judgements constitute the hypotheses for the rule \((par\text{prob})\). Therefore using the rule \((par\text{prob})\), we can conclude that \(S_{par} : pts \rightarrow \nabla((pts_1, 1/2), (pts_2, 1/2))\). The post type of \(S_{par}\) clearly covers all semantics states that can be reached by executing \(S_{par}\). Now we give an example for the application of the rule \((par-if\text{prob})\). Let:

- \(S_1 : x := \& y,\)
- \(S_2 : x := \& z,\)
The following theorem proves the soundness of the type system. The meant soundness implies that the evaluation of the expression with respect to the state is an address, then this evaluation is surely (positive probability) approximated by the soundness of the points-to type system. Theorem 1 formalizes the soundness of points-to types.

\[ \text{Lemma 3.1: } \forall n \in \mathbb{N}, \forall s \in S, \text{points-to } t_s \Rightarrow \text{points-to } t_{s'} \quad \text{where } t_{s'} = t_s \times n \]

\[ \text{Lemma 3.2: } \text{for a certain state that succeeds with respect to the relation } \models \text{ whose definition is based on probabilities.} \]

\[ \text{Theorem 1 (Soundness) } \text{Suppose that } S : \text{points-to } t_s, S : (\gamma, p) \Rightarrow (\gamma', p'), \text{and } (\gamma, p) \models \text{points}. \text{ Then } (\gamma', p') \models \text{points}. \]

\[ \text{Proof: } \text{A structure induction on type derivation can be used to complete the proof of this theorem. Some cases are presented below.} \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]

\[ \text{The case of } (\equiv) : \text{in this case } p' = \]
Therefore by the soundness of \(\gamma' \models (pts', p')\). Hence by Lemma 3.2, \(\gamma' \models (pts, p)\) implies \(\gamma' \models (pts', p')\).

The case of \(\langle \ast := \text{prob} \rangle\): in this case for some \(z \in \text{Var}, \gamma(y) = z'\) and \(x := z : (\gamma, p) \rightsquigarrow (\gamma', p)\).

For some \(i, z'_i = z'_i\) since \((\gamma, p) \models pts\). Hence by assumption \(x := z_i : pts \rightarrow pts_i\). Therefore by soundness of \(\langle \ast := \text{prob} \rangle\), \((\gamma', p) \models pts_i \leq pts' \models pts[x \mapsto \nabla((pts_1, p_1), \ldots, (pts_n, p_n))(x)]\).

The case of \(\langle \text{par} \rangle\): in this case there exists a permutation \(\theta : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}\) and \(\gamma, p, pts' = pts[x \mapsto A]\), and \(\gamma' = \gamma[x \mapsto [e] \gamma]\). Hence by Lemma 3.2, \(\gamma' \models (pts', p')\).

...
Probabilistic pointer analysis and speculative optimizations:

Although pointer analysis is a well-established program analysis and many techniques have been suggested, there is no single technique that is believed to be the best choice\textsuperscript{25}. The trade-off between accuracy and time costs hinders a universal pointer analysis and motivates application-directed techniques for pointer analysis\textsuperscript{26}. A probabilistic pointer analysis that is flow-sensitive and context-insensitive has been presented for Java programs\textsuperscript{27}. While our work is based on type systems, previous work is based on interprocedural control flow graphs whose edges are enriched with probabilities. While our work treats multithreaded programs, the work in Ref. 27 treats only sequential programs. Context-sensitive and control-flow-sensitive pointer analyses\textsuperscript{4,10,28,29} are known to be accurate but not scalable. On the other hand the context-insensitive control-flow-insensitive techniques\textsuperscript{6,11} are scalable but excessively conservative. A convenient mixture of accuracy and scalability is introduced by some technique\textsuperscript{7,30,31} to optimize the trade-off mentioned above. The probabilistic pointer analysis of a simple imperative language\textsuperscript{8,9} and the pointer analysis of multithreaded programs\textsuperscript{32} have been studied. However, none of these typical techniques for pointer analysis study the probabilistic pointer analysis of multithreaded programs.

Speculative optimizations\textsuperscript{33–36} are considered by many program analyses. A probabilistic technique for memory disambiguation was proposed\textsuperscript{34}. This technique measures the probability that two array references alias. Nevertheless this approach is not convenient to pointers. By lessening the safety of analysis, a pointer analysis that considers speculation was introduced\textsuperscript{35}. Another unsafe analysis, which achieves scalability using transfer functions, was proposed\textsuperscript{36}. The problem with these last two approaches is that they do not compute the probability information required by speculative optimizations.

Type systems in program analysis:

There are general algorithms\textsuperscript{4,13,14,37–39} for using type systems to present data flow analyses, which are monotone and forward or backward. While a way\textsuperscript{14,37} to reason about program pairs using relational Hoare logic exists, program optimizations\textsuperscript{14,38} as type systems also exist. Type systems were also used to cast safety policies for resource usage, information flow, and carrying-code abstraction\textsuperscript{40,41}. Proving the soundness of compiler optimizations for imperative languages, using type systems, gained much interest\textsuperscript{12–14} of many researchers. Other work studies translating proofs of functional correctness using wp-calculus\textsuperscript{42} and using a Hoare logic\textsuperscript{14}. There are other optimizations\textsuperscript{43} that boost program quality besides maintaining program semantics.

Edge and path profiling:

Edge (path) profiling research simply aims at profiling programs edges (paths). The profiling process can be done statically or dynamically. Profiling techniques can be classified into:

- Sample-based techniques\textsuperscript{16,17} which profile representative parts of active edges and paths,
- One-time profiling methods which profile only part of the execution of the program to cut down the overhead\textsuperscript{17,44},
- Instrumentation-based techniques\textsuperscript{45} which are more convenient for programs with comparably anticipated behaviour, and
- Hardware profiling which employs hardware to gather edge profiles using existing hardware for branch anticipation\textsuperscript{18}.

Using a parallel data-flow diagram\textsuperscript{46}, many of these techniques are applicable to the language studied in this paper. In particular, a hybrid sampling and instrumentation approach\textsuperscript{15} is a convenient choice giving its simplicity and powerful.

Acknowledgements: This work was started during the author’s sabbatical at Institute of Cybernetics, Estonia in the year 2009. The author is grateful to T. Uustalu for fruitful discussions. This work was partially supported by the EU FP6 IST project MOBIUS. The author is also indebted to the anonymous reviewers whose queries and comments improved the paper.

REFERENCES


paradigmatic approach to static analysis. In: Mogensen
TAE, Schmidt DA, Sudborough IH (eds) *The Essence
of Computation*, Springer, vol 2566 of Lecture Notes in
Computer Science, pp 223–44.

HR, Probst CW, Pugliese R (2010) From flow logic
to static type systems for coordination languages. *Sci
Comput Program* 75, 376–97.

40. Beringer L, Hofmann M, Momigliano A, Shkaravska
O (2004) Automatic certification of heap consump-
tion. In: Baader F, Voronkov A (eds) *LPAR*, Springer,
vol 3452 of Lecture Notes in Computer Science,
pp 347–62.

carrying code from certified abstract interpretation and

Program Lang Syst* 31, Article 18.

176, 37–59.


46. Grunwald D, Srinivasan H (1993) Data flow equa-

the ACM SIGPLAN 2004 Conference on Programming
Language Design and Implementation 2004*, Washing-
ton DC, USA, June 9–11, 2004, ACM.