Research Article

Frequent Statement and Dereference Elimination for Imperative and Object-Oriented Distributed Programs

Mohamed A. El-Zawawy\textsuperscript{1,2}

\textsuperscript{1} College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh 11432, Saudi Arabia
\textsuperscript{2} Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

Correspondence should be addressed to Mohamed A. El-Zawawy; maelzawawy@gmail.com

Received 18 August 2013; Accepted 21 September 2013; Published 27 April 2014

Academic Editors: P. Bala and S. Choi

Copyright © 2014 Mohamed A. El-Zawawy. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper introduces new approaches for the analysis of frequent statement and dereference elimination for imperative and object-oriented distributed programs running on parallel machines equipped with hierarchical memories. The paper uses languages whose address spaces are globally partitioned. Distributed programs allow defining data layout and threads writing to and reading from other thread memories. Three types systems (for imperative distributed programs) are the tools of the proposed techniques. The first type system defines for every program point a set of calculated (\textit{ready}) statements and memory accesses. The second type system uses an enriched version of types of the first type system and determines which of the \textit{ready} statements and memory accesses are used later in the program. The third type system uses the information gather so far to eliminate unnecessary statement computations and memory accesses (the analysis of frequent statement and dereference elimination). Extensions to these type systems are also presented to cover object-oriented distributed programs. Two advantages of our work over related work are the following. The hierarchical style of concurrent parallel computers is similar to the memory model used in this paper. In our approach, each analysis result is assigned a type derivation (serves as a correctness proof).

1. Introduction

Distributed programming is about building a software that has concurrent processes cooperating in achieving some task. For a problem specification, the type, number, and the way of interaction of processes needed to solve the problem are decided beforehand. Then a supercomputer can be computationally simulated by a group of workstations to carry different processes. A group of supercomputers can in turn be combined to provide a computing power greater than that provided by any single machine. This enormous computing power provided by distributed systems is why the distributed programming style [1–3] is quite important and attractive. Among examples of distributed programming languages (DPLs), based on machines having multicore processors and using partitioned-global model, are Unified Parallel C (UPC), Chapel, Titanium which is based on Java, and X10.

Among advantages of object-oriented programming (OOP) is combining other styles such as imperative, functional, and relational programming. Concepts of class, procedure, and inheritance are basics for OOP. These concepts result in dynamic behavior in various implementations of object-oriented programming languages.

Recomputing a nontrivial statement and reaccessing a memory location are waste of time and power if the value of the statement and the content of the location have not been changed. The purpose of frequent statement and dereference elimination analysis is to save such wasted power and time. This is an interesting analysis because it involves connecting statement and dereference calculations to program points where the calculated values may be reused. The analysis also requires changing program points at the ends of these connections. Such changes to program points have to be done carefully so that they do not destroy the compositionality. Our approach to treat this analysis is a type system [4, 5] built...
on a combination of two analyses; one of them builds on the results of the other one.

For different programming languages, in previous work [4, 5], we have proved that the type systems style is certainly an adaptable approach for achieving many static analyses. This paper proves that this style is flexibly useful to the involved and important problem of frequent statement and dereference elimination of imperative and object-oriented distributed programs.

This paper introduces new techniques for frequent statement and dereference elimination for imperative and object-oriented distributed programs running on hierarchical memories. Simply structured type systems are the main tools of this paper’s techniques presented using the languages while eX of Figure 2 and OODP of Figure 3. These languages are equipped with basic commands for distributed execution of programs and for pointer manipulations. The single program multiple data (SPMD) model is the execution archetypal used in this paper. On different data of different machines this archetypal runs the same program. The analysis of frequent statement and dereference elimination for distributed programs is achieved in three steps each of which is done using a type system. The first of these steps achieves ready statement and memory access analysis. The second step deals with semiexpectation analysis and builds on the type system of the first step. The third type system takes care of the analysis of frequent statement and dereference elimination and is built on the type system of the second step. The paper also illustrates how these type systems can be generalized to cover object-oriented distributed languages.

This paper is an extended and revised version of [6], which treats imperative distributed programs. The work of [6] was generalized in Section 5 of the current paper to cover object-oriented distributed programs. The soundness theorems of the current paper are stated using memory model and operational semantics in the appendix of [6].

Motivation. The left-hand-side of Figure 1 presents a motivating example of our work. We note that lines 4 and 6 dereference a * b which has already been dereferenced in line 2 with no changes to values of a and b in the path from 2 to 6. This is a waste of computational power and time (accessing a secondary storage). One objective of the research in this paper is to avoid such waste by transforming the program into that in the right-hand-side of the algorithm. This is not all; we need to do that in a way that provides a correctness proof for each transformation. We adopt a style (type systems) that provides these proofs (type derivations).

Contributions. Contributions of this paper are new techniques, in the form of type systems, for achieving the following analyses for imperative and object-oriented distributed programs.

(1) The analysis of ready statement and memory access.
(2) The analysis of semiexpectation.
(3) The analysis of frequent statement and dereference elimination.

Organization. The rest of the paper is organized as follows. Section 2 presents the type system achieving the analysis of ready statement and memory access for imperative distributed programs. The analysis of semiexpectation as an enrichment of the type system presented in Section 2 is outlined in Section 3. The main type system carrying the analysis of frequent statement and dereference elimination is contained in Section 4. Type systems of Sections 2, 3, and 4 are generalized in Section 5 to cover object-oriented distributed programs. Related and future works are discussed in Section 6.

2. Ready Statement and Memory

Access Analysis of while eX

If the value of a statement and the content of a memory location have not been changed, then the compiler should not recompute the statement or reaccess the location. The purpose of frequent statement and dereference elimination is to save the wasted power and time involved in these repeated computations. This is not a trivial task; compared to other program analyses, it is a bit complex. This task is done in stages. The first stage is to analyze the given program to recognize ready statements and memory locations.

The analysis of ready statements and memory locations calculates for every program point the set of statements and memory locations that are ready at that point in the sense of Definition 1. This section presents a type system (ready type system) to achieve this analysis for imperative distributed programs.

Definition 1. (1) At a program point pt, a statement S is ready if each computational path to pt

(a) contains an evaluation of S at some point (say pt ') and
(b) does not modify S (changing value of any of S’s variables) between pt' and pt.
(2) At a program point pt, a memory location l is ready if each computational path to pt

(a) reads l at some point (say pt') and

(b) does not modify content of l between pt' and pt.

The ready analysis is a forward analysis that takes as an input a set of statements and memory locations (the ready set of the first program point). It is sensible to let this set be the empty set. The set of types of our ready type system has the form: points-to-types × \( \mathcal{P}(\text{Stmt}^+ \cup \text{gAddr}) \), where

(1) \( \text{Stmt}^+ \) is the set of nontrivial statements (Figure 2),

(2) \( \text{gAddr} \) is the set of global addresses. This set is defined precisely in the appendix of [6], and

(3) points-to-types is a set of points-to-types (typically have the form of maps from the union of variables and global addresses to the power set of global addresses [4, 7]).

The subtyping relation has the form \( \preceq_p \times \succeq \), where \( \preceq_p \) is the order relation on the points-to-types and \( \succeq \) is the order relation on \( \mathcal{P}(\text{Stmt}^+ \cup \text{gAddr}) \). A state on an execution path is of type \( rs \in \mathcal{P}(\text{Stmt}^+ \cup \text{gAddr}) \) if all elements of \( rs \) are ready at this state according to Definition 1. Judgments of the ready type system have the form \( S : (p,rs) \rightarrow_{m} (A',p',rs') \). The symbols \( p \) and \( p' \) denote the points-to-types of the before and after states of executing \( S \). The set \( A' \) denotes the set of addresses that \( S \) may evaluate. We assume that all such pointer information is given along with the statement \( S \). Techniques like [4, 7] are available to compute the pointer information. For a given statement along with pointer information and a ready pretype \( rs \), we present a type system to calculate a post ready-type \( rs' \) such that \( S : (p,rs) \rightarrow_{m} (A',p',rs') \). The type derivation of this typing process is a proof for the correctness of the ready information. The meaning of the judgment is that if elements of \( rs \) are ready before executing \( S \), then elements of \( rs' \) are ready after executing \( S \).

The inference rules of the ready type system are presented in Algorithm 1. Comments on the inference rules are in order. We note that numbers, variables, and the allocating statement \( \text{new} \) do not affect the ready pretype. In line with semantic rules \( (t_{op}) \) and \( (b_{op}) \) [6], nontrivial arithmetic and Boolean statements and their nontrivial substatements are made ready. The direct assignment rule \( (:=) \) expresses that after executing the assignment the substatements of r.h.s. become ready and that all statements involving \( x \) become unready as the value of \( x \) may become different. The rule \( (\ast) \) reflects the fact that the statement \( \ast S \) becomes ready after executing the dereference. Moreover if \( S \) evaluates a single address according to the underlying pointer analysis, then this address becomes ready as well. However if \( S \) evaluates a large set of addresses (more than one), then we are not sure which of these addresses is the concerned one and hence cannot conclude any readiness information about addresses. The rule \( (\neg\ast) \) adds the substatements of \( S_1 \) and \( S_2 \) to the ready pretype. Since the content of address referenced by \( S_1 \) is possibly changed after executing the statement, all statements involving dereferencing this address are removed from the set of ready items. Remaining rules are self-explanatory. The Boolean statements true and false have inference rules similar to that of \( n \).

All in all, the information provided by type derivations obtained using this and the following type system is classified into two sorts. The first sort is about knowing the program point at which a particular statement becomes ready. The second sort of information is about the program point at which a precomputed value of a ready statement can be replaced with the statement.

Now we recall the assumption that our distributed system consists of \( |M| \) machines. For a given statement \( S \) and a given machine \( m \), the type system of Algorithm 1 calculates for each
TheScientificWorldJournal

The aim of frequent statement elimination is to introduce new variables to accommodate values of frequent statements and each machine to get new information is valid on any of the |M| machines.

Consider

\[
\forall m \in M : S : (\sup \{ p, p_j | j \neq i \}, \sup \{ rs, rs_j | j \neq i \}) \rightarrow_m (A_m, p_m, rs_m) (\text{main-rs}).
\]

The rule (main-rs) supposes a suitable notion for the join of pointer types. The soundness of the ready type system is stated as follows.

**Theorem 2.** Suppose that \((S, \delta) \rightarrow (V, \delta'), S : (p, rs) \rightarrow (A', p', rs')\), and the items of \(rs\) are ready at the point corresponding to \(\delta\) on the execution path. Then the items of \(rs'\) are ready at the point corresponding to \(\delta'\) on the execution path.

### 3. Semiexpectation Analysis of while

The aim of frequent statement elimination is to introduce new variables to accommodate values of frequent statements and
reusing these values rather than recomputing the statements. Analogously, the aim of frequent dereferences elimination is to introduce new variables to accommodate values of frequent dereferences and reusing these values rather than reaccessing the memory. The information gathered so far by the ready type system introduced in the previous section is not enough to achieve frequent statements and dereferences elimination. We need to enrich the ready information, assigned to each program point, with new information called semiexpectable information.

Definition 3. (1) At a program point \( p \), a statement \( S \) is semiexpectable if there is a computational path from \( p \) that

(a) contains an evaluation of \( S \) at some point (say \( p' \)), where \( S \) is ready at \( p' \), and

(b) does not evaluate \( S \) between \( p' \) and \( p \).

(2) At a program point \( p \), a memory location \( l \) is semiexpectable if each computational path to \( p \)

(a) reads \( l \) at some point (say \( p' \)) where \( l \) is ready at \( p' \), and

(b) does not read \( l \) between \( p' \) and \( p \).

The semiexpectation analysis is a backward analysis that takes as an input a set of statements and memory locations (the semiexpectable set of the last program point). It is sensible to let this set be the empty set. The following example gives an intuition for the previous definition:

\[
\text{if}(\cdots), \text{then } a := y + t \text{ else } b := *r; \quad c := \frac{(y + t)}{*r}. \quad (2)
\]

Neither the statement \( y + t \) nor the statement \( *r \) is ready after the if statement because they are not computed in all branches. Hence it is not true to replace these statements with variables towards optimizing the last statement of the example. The job of the type system presented in this section is to provide us with this sort of information. More precisely, as the statements \( y + t \) and \( *r \) are not ready after the if statement, the second statement of the example does not make them semiexpectable.

The semiexpectation analysis assigns for each program point the set of items that are semiexpectable. The analysis is based on the readiness analysis and is backward. The set of types of the semiexpectation type system has the form: \( \text{points-to-types} \times (\mathcal{P}(\text{Stmt}) \cup \mathcal{G}(\text{Addr})) \times (\mathcal{P}(\text{Stmt}) \cup \mathcal{G}(\text{Addr})) \). The subtyping relation has the form \( \leq_p \times \times \times \). A state on an execution path is of type \( se \in (\mathcal{P}(\text{Stmt}) \cup \mathcal{G}(\text{Addr})) \) if all elements of \( se \) are semiexpectable according to Definition 3. Judgments of the semiexpectation type system have the form \( S : (p, rs, se) \rightarrow (\mathcal{P}(\text{Stmt}) \cup \mathcal{G}(\text{Addr})) \). For a given statement along with pointer information, readiness information, and a semiexpectation type \( se' \), we present a type system to calculate a pre-semiexpectable-type \( se \) such that \( S : (p, rs, se) \rightarrow (\mathcal{P}(\text{Stmt}) \cup \mathcal{G}(\text{Addr})) \). The type derivation of this typing process is proof for the correctness of the semiexpectable information. The meaning of the judgment is that if elements of \( se' \) are semiexpectable after executing \( S \), then elements of \( se \) must have been semiexpectable before executing \( S \).

The inference rules of the semiexpectation type system are shown in Algorithm 2. Some comments on the inference rules are in order. In the rule \( (\text{seq-op}) \), given the posttype \( se' \), we calculate the pretype \( se'' \) for the statement \( S_2 \). Then the resulting pretype is used as a posttype for the statement \( S_1 \) to calculate the pretype \( se \). In line with Definition 3, the arithmetic statement \( S_1 \) \( i_{op} \) \( S_2 \) is added to \( se \) only if it belongs to \( rs \). Similar explanations illustrate the remaining rules mimic the rules of the ready type system.

Now we recall the assumption that our distributed system consists of \( |M| \) machines. For a given statement \( S \) and a given machine \( m \), the type system given above calculates for each program point of \( S \) the set of semiexpectable items. Now the following rule can be used to combine the information calculated for each machine to get new semiexpectable information for each program point. The new semiexpectable information is valid on any of the \( |M| \) machines.

Consider

\[
\forall m \in M \cdot S : \sup \left\{ p, p_j | j \neq i \right\}, \sup \left\{ rs, rs_j | j \neq i \right\} \rightarrow_m (A_m, p_m, rs_m)
\]

\[
S : (p, rs) \rightarrow_{\cup A_v, \sup \left\{ p_1, \ldots, p_n \right\}, \sup \left\{ rs_1, \ldots, rs_n \right\})
\]

\[
(3)
\]

4. Frequent Statement and Dereference Elimination of while_d

This section presents a type system that is an enrichment of the type system presented in the previous section. The type system of this section achieves the frequent statement and dereference elimination. The type system uses a function \( sn : S^+ \rightarrow \text{Stmt-names} \) that assigns each nontrivial statement a name. These names are meant to carry values of frequent statements and dereferences. The judgments of our type system have the form \( S : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rightarrow (ns, S') \). The type information \( (p, rs, se) \) and \( (A', p', rs', se') \)
were calculated by the previous type system. \( S' \) is the optimization of \( S \) and \( ns \) is a sequence of assignments that links optimized statements with the names of their unoptimized versions.

Algorithms 3 and 4 present inference rules for the frequent statements and dereferences elimination. We note the following on the inference rules. A big deal of optimization is achieved by the three rules for \( *S \). These rules are \( \ast_{1} \), \( \ast_{2} \), and \( \ast_{3} \). The rule \( \ast_{1} \) treats the case where \( S \) is semiexpectable and is not ready before calculating the statement. In this case, a statement name of \( S \) is used. The rule \( \ast_{2} \) considers the case where \( S \) is neither semiexpectable at the program point after execution nor ready before calculating the statement. In this case, the statement \( S \) does not get changed. Similarly, the three rules \( \ast_{3} \) treats different cases for arithmetic statements. The Boolean statements are treated with rules quite similar to that of arithmetic statements. The rule \( \ast_{3} \) reuses frequent substatements of the guard. This

\[
\begin{align*}
\text{Algorithm 2: Inference rules of the semiexpectation type system.}
\end{align*}
\]
Algorithm 3: Inference rules for the frequent statements and dereferences elimination (1).

is done via adding $\nu$ in the positions clarified in the rule. Remaining rules of system are self-explanatory.

For expressing the soundness, we introduce the following definition.

Definition 5. Suppose that $\delta$ is a state defined on the set of locations, $\text{Loc}$ (16, Definition 4). Suppose also that $\delta_\ast$ is a state defined on $\text{Loc} \cup \text{Stmt-names}$. The expression $\delta \equiv_\ast \delta_\ast$ denotes the fact that $\delta$ and $\delta_\ast$ are equivalent with respect to the \textit{semiexpectation} type $\text{se}$. More precisely $\delta \equiv_\ast \delta_\ast$, if and only if

(1) for all $j \in \text{Loc}$. $\delta(j) = \delta_\ast(j)$, and
(2) for all $S \in \text{se}$. $(S, \delta) \sim_\ast (\nu, \delta') \Rightarrow \delta_\ast(\text{sn}(S)) = \nu$.

The soundness of frequent statements and dereferences elimination means that the original and optimized programs are equivalent in the following sense:

(i) the states of the two programs coincide on the $\text{Loc}$, and
(ii) if a statement is both \textit{ready} and \textit{semiexpectable}, then its semantics in the original-program state equals...
The value of its corresponding name in optimized-program state.

This gives an intuition to the previous definition. The following soundness theorem is proved by a structure induction.

**Theorem 6.** Suppose that $S : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')$ and $δ ≡_δ δ'$. Then

(i) $(S, δ) →_m (v, δ') ⇒ ∃ δ' : δ' ≡_δ δ' and (S', δ') →_m (v, δ')$;

(ii) $(S', δ') →_m (v, δ') ⇒ ∃ δ' : δ' ≡_δ δ' and (S, δ) →_m (v, δ')$.

**5. Frequent Statement and Dereference Elimination of OODP Programs**

This section generalizes the type systems of previous sections to cover object-oriented distributed programs. Hence, a new model for object-oriented distributed programs and necessary changes to proposed type systems for the analysis of frequent statement and dereference elimination are presented in this section. Object-oriented concepts such as subtyping and inheritance are included in the model language (dubbed OODP) whose syntax is shown in Figure 3.

In line with OOP concepts, local variables are contained in functions and live while their functions are live. While parameters of function are represented using local variables, a class's internal state is contained in its instance variables. A class is a container for a set of function definitions. Each function $f$ has parameter $p_f$, a main statement $S_f$, and a statement $S_f$ representing value returned by the function. Hence an OODP program is a set of classes followed by a "main" function. Figure 4 presents semantic spaces and naming conventions used in the rest of the paper.

As shown in the previous sections, the analysis of frequent statement and dereference elimination for imperative distributed programs is achieved in three steps. In the following, we show necessary changes to the three type systems presented so far to cover object-oriented distributed programs.

For each program point, ready statements and memory locations (Definition 1) are computed by the analysis of

\[
S : (p, rs, se) → m(A'', p'', rs'', se'') ⇒ (ns, S')
\]

if $S$ then $S_f$ else $S_j : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')$

\[
(λx : S)[S₁/x] : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
S : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

while $S$ do $S_j : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')$

\[
fd \text{ name} : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
\text{name} : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
λx : S : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
\text{name} \cdot S : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
\text{name} \cdot S : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
\text{new}_1 : p → m(A', p') \text{ new}_2 : p → m(A', p')
\]

\[
\text{letrec} = \text{S} \in S' : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
\text{convert} (S, n) : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
\text{transmit} S_₁ \text{ from } S_₂ : (p, rs, se) → m(A', p', rs', se') ⇒ (ns, S')
\]

\[
(p₁', rs₁', se') ≤ (p₂, as₂, pa₂) ⇒ (ns, S')
\]

\[
S : (p₁', rs₁', se') → m(p₂, as₂, pa₂) ⇒ (ns, S')
\]

\[
\text{Defs} : \emptyset \Rightarrow \text{fd} \ S : (p, rs, se) → m(A, p', rs', se') ⇒ (ns, S')
\]

\[
\text{Defs} : S : (p, rs, se) → m(A, p', rs', se') ⇒ \text{Defs} : ns; S'
\]
ready statements and memory locations. Adding rules of Algorithm 5 to that of Algorithm 1 results in a type system that calculates this analysis for object-oriented distributed programs of Figure 3. Using semantics notions of Figure 4, Definitions 1, 3, and 5 are applicable and convenient for the analyses in this section for the language OODP.

Comments on the inference rules are in order. The rules of Algorithm 5 suppose the existence of a class analysis that calculates the set of classes that a statement may reference. For all functions f, the body and return statements of f are added to rs to produce rs' . Then for any class C that S₁ may reference, statements involving C ∨ v are removed from rs'. In the rule (:= o₂ f), Cₛ includes classes that o₂ may reference. For all functions named f in classes of Cₛ, the body and return statements are enumerated in the set {S₁, . . . , Sₘ}. Ready substatements of these statements are added to rs to produce rsₘ₊₁. Then all statements involving o₁ are removed from rsₘ₊₁.

Using semantics notations of Figure 4, soundness of the type system of Algorithm 5 is stated as follows.

**Theorem 7.** Suppose that (S, s, h) → (V, s', h'), S : (p, rs) → (A, p', rs') and the items of rs are ready at the point corresponding to (s, h) on the execution path. Then the items of rs' are ready at the point corresponding to (s', h') on the execution path.

The goals of main analysis of this section for OODP are as follows.

Introducing new variables to maintain values of frequent statements and dereferences and then reusing these values instead of recomputing the statements and reaccessing the memory.

To achieve this goal the ready information needs to be enriched with information of semiexpectable.

Adding rules of Algorithm 6 to that of Algorithm 2 results in a type system that calculates the analysis of semiexpectable for object-oriented distributed programs of Figure 3. Some comments on the inference rules of Algorithm 6 are in order. In the rule (:= o₂ f), starting with the posttype se', the pretype se'' is calculated for the statement S₂. Then se' is used as a posttype for S₁ to get the main pretype se. Similarly to (:= o₂ f), the rule (:= o₂ f) enumerates body and return statements of convenient functions. Then sequentially se is calculated starting from se'. The remaining rules mimic the rules of the ready type system.

Using semantics notations of Figure 4, soundness of the type system of Algorithm 6 is stated as follows.

**Theorem 8.** Suppose that (S, s, h) → (V, s', h'), S : (p, rs, se) → (A', p', rs', se'), and the items of se are semiexpectable at the point corresponding to (s', h') on the execution path. Then the items of se are semiexpectable at the point corresponding to (s, h) on the execution path.
The following definition generalizes Definition 5 and is necessary to express soundness.

**Definition 9.** Suppose that \((s, h)\) is a state defined on the set of locations \(Loc\) \((\{6, \text{Definition 4}\})\). Suppose also that \((s_*, h_*)\) is a state defined on \(Loc \cup Stmt\)-names. The expression \((s, h) \equiv_{\omega}(s_*, h_*)\) denotes the fact that \((s, h)\) and \((s_*, h_*)\) are equivalent with respect to the *semi-expectation* type system. More precisely \((s, h) \equiv_{\omega}(s_*, h_*)\) if and only if

1. For all \(l \in Loc\), \(s(l) = s_*(l)\), and
2. For all \(s \in se\), \((s, s) \rightarrow_m (v, s', h') \Rightarrow s_*(sn(S)) = v\).

Using semantics notations of Figure 4, soundness of the type system of Algorithm 7 is stated as follows.

**Theorem 10.** Suppose that \(S : \{p, rs, se\} \rightarrow_m (A_1', p', rs', s_e') \Rightarrow (rs, S') \equiv_{\omega}(s*, h_*)\). Then

\[
(i) (s, s, h) \rightarrow_m (v, s', h') \Rightarrow \exists (s'_*, h'_*) \Rightarrow (s_*, h_*) \equiv_{\omega}(s'_*, h'_*)
\]

and

\[
(ii) (s'_*, h'_*) \rightarrow_m (v', s, h) \Rightarrow \exists (s'_*, h'_*) \Rightarrow (s_*, h_*) \equiv_{\omega}(s'_*, h'_*)
\]
6. Related Work

The techniques of common subexpression elimination (CSE) [8, 9] are closed to our work. In [10], a type system for CSE of the while language is introduced. The work presented in our paper can be realized as a generalization of that presented in [10]. The generality of our work is evident in our language models which are much richer with distributed, pointer, and object-oriented commands. Consequently, the operational semantics that we measure the soundness of our system against are much more involved than that used in [10]. Using new opportunities appearing while scheduling control-intensive designs, the work in [11] introduces a technique that dynamically eliminates CSE. To optimize polynomial expressions (important for applications like domains, computer graphics, and signal processing), the paper [12] generalizes algebraic techniques originally designed for multilevel logic synthesis. The generalization in [12] uses factoring to eliminate common subexpressions of polynomial expressions.

There are many analyses for optimizing object-oriented programs. In [13] evolutionary multiobjective optimization methods are used to present a Class-Based Elliptic Genetic Algorithm (CBEGA) for testing OOP. A new method to optimize OOP for field access in concurrent object-oriented programs is presented in [14]. This work utilizes the correctness concept that concurrency control must be used by programmers. A new model concurrency abstraction is presented in [15]. This model has the advantage of separating the specification of the synchronization code from the method bodies.

The association of a correctness proof with each result of the static analysis is important and needed by applications like proof-carrying code and certified code. The work presented in this paper has the advantage over most related work of constructing these proofs. Adding to the value of using type systems, the proofs constructed in our proposed approach have the form of type derivations. The work in [4, 16, 17] presents many examples of other static analyses that are in the form of type systems.

In [18], a technique for flow-insensitive pointer analysis of programs that run on parallel and hierarchical machines and that share memory is introduced. Via a two-level hierarchy, [19, 20] present constraint-based approaches to evaluate locality information and sharing attributes of references. Our language model is a generalization of models presented in [18, 19].
Much research activities [18, 21] was devoted to analyze distributed programming. This is motivated by the importance of distributed programming as a main stream of programming today. The examining and capturing of causal and concurrent relationships are among important issues to many distributed systems applications. In [22], an analysis that examines the source code of each process constructs an inclusive graph, POG, of the possible behaviors of systems. Data racing bugs [23] can be a side effect of the parallel access of cores of a multicore process to a physically distributed memory. In [23] a technique, called DRARS, is proposed for avoidance and replay of this data race. Parallel programs on DSM or multicore systems can be debugged using DRARS. The classical problems of satisfiability decidability and algorithmic decidability are approached in [24] on the distributed programs model of message sending. In this work, distributed programs are represented by communicating via buffers.

7. Conclusion

This paper introduces new techniques for the analysis of frequent statement and dereference elimination for imperative and object-oriented distributed programs running on parallel machines equipped with hierarchical memories. Type systems are the tools of the techniques presented in this paper. The first sort of proposed type systems defines for program points of a distributed program sets of calculated (ready) statements and memory accesses. The second sort determines which of the ready statements and memory accesses are used later in the program. The final sort eliminates unnecessary statement computations and memory accesses.

Disclosure

This is an extended and revised version of [6].

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

References


