Hindawi Publishing Corporation
The Scientific World Journal
Volume 2014, Article ID 839121, 13 pages
http://dx.doi.org/10.1155/2014/839121

Research Article

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

Mohamed A. El-Zawawy[1,2]

[1] *College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh 11432, Saudi Arabia*
[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

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 type 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 (*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 *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

```
(0)                                                    l := c * d; k := a * b;
(1)  x := a * b + c * d;                               x := k + l;
(2)  x := convert(*(a * b), 2);                        x := convert(*k, 2);
(3)  y := transmit c * d from (3);                       y := transmit l from (3);
(4)  if (*(a * b) = *(c * d))                          if (*k = *l)
(5)      then y := transmit *(c * d) from (2);             then y := transmit *(c * d) from (2);
(6)      else x := convert (*(a * b), 2);                  else x := convert(*(a * b), 2);
```

Figure 1: Motivating example.

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 $while_d$ 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 $while_d$

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.

$$\begin{aligned}
\mathit{name} &::= \text{“}\mathit{string\ of\ characters}\text{”}.\\
S \in \mathit{Stmts} &::= n \mid \mathit{true} \mid \mathit{false} \mid x \mid S_1\, i_{\mathrm{op}}\, S_2 \mid S_1\, b_{\mathrm{op}}\, S_2 \mid *S \mid \mathit{skip} \mid \mathit{name} \mid x := S \mid S_1 \leftarrow S_2 \mid\\
&\quad S_1; S_2 \mid \mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f \mid \mathit{while}\ S\ \mathit{do}\ S_t \mid \lambda x \cdot S \mid S_1 S_2 \mid \mathit{letrec}\ x = S\ \mathit{in}\ S' \mid\\
&\quad \mathit{new}_l \mid \mathit{convert}\ (S, n) \mid \mathit{transmit}\ S_1\ \mathit{from}\ S_2.\\
\mathit{Defs} &::= (\mathit{name} = S); \mathit{Defs} \mid \varepsilon.\\
\mathit{Program} &::= \mathit{Defs}\colon S.
\end{aligned}$$

where
$x \in \mathit{lVar}$, an infinite set of variables, $n \in \mathbb{Z}$ (integers), $i_{\mathrm{op}} \in \mathbb{I}_{\mathrm{op}}$ (integer-valued binary operations), and $b_{\mathrm{op}} \in \mathbb{B}_{\mathrm{op}}$ (Boolean-valued binary operations).

Figure 2: The programming language while_d.

$$\begin{aligned}
S \in \mathit{Stmts} &::= n \mid o \mid o \cdot v \mid S_1\, i_{\mathrm{op}}\, S_2 \mid \mathrm{this} \mid (C)S \mid \mathrm{true} \mid \mathrm{false} \mid S_1\, c_{\mathrm{op}}\, S_2 \mid S_1\, b_{\mathrm{op}}\, S_2 \mid \mathit{skip} \mid\\
&\quad o := S \mid S_1 \cdot v := S_2 \mid o_1 := o_2 \cdot f(S) \mid o_1 := \mathit{super} \cdot f(S) \mid o := \mathrm{new}\ C \mid S_1; S_2 \mid\\
&\quad \mathrm{if}\ b\ \ \mathrm{then}\ \ S_t\ \mathrm{else}\ \ S_f \mid \mathrm{while}\ b\ \mathrm{do}\ S_t \mid \mathit{convert}\ (S, n) \mid \mathit{transmit}\ \ S_1\ \mathit{from}\ S_2.\\
\mathrm{fun} \in \mathrm{Funs} &::= f\ (p)\{S;\ \mathrm{return}\ (S_r);\}\\
\mathrm{inhrt} \in \mathrm{Inherits} &::= \varepsilon \mid \mathrm{inherits}\ C\\
\mathrm{cls} \in \mathrm{Classes} &::= \mathrm{class}\ C\ \mathrm{inhrt}\ \{\mathit{fun}^*\}\\
\mathrm{prog} \in \mathrm{Progs} &::= \mathrm{cls}^*\ \mathrm{main}()\ \{S\}
\end{aligned}$$

Figure 3: The programming language *OODP*.

(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: $\mathit{points\text{-}to\text{-}types} \times \mathscr{P}(\mathit{Stmt}^+ \cup \mathit{gAddrs})$, where

(1) Stmt^+ is the set of nontrivial statements (Figure 2),

(2) gAddrs 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 $\le_p \times \supseteq$, where $\le_p$ is the order relation on the points-to-types and $\supseteq$ is the order relation on $\mathscr{P}(\mathit{Stmt}^+ \cup \mathit{gAddrs})$. A state on an execution path is of type $rs \in \mathscr{P}(\mathit{Stmt}^+ \cup \mathit{gAddrs})$ if all elements of rs are *ready* at this state according to Definition 1. Judgments of the *ready* type system have the form $S : (p, rs) \to_m (A', p', rs')$. 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 $S : (p, rs) \to_m (A', p', rs')$. 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 (*new*) do not affect the *ready* pretype. In line with semantic rules (i^r_{op}) and (b^r_{op}) [6], nontrivial arithmetic and Boolean statements and their nontrivial substatements are made *ready*. The direct assignment rule ($:=^r$) 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 ($*^r$) reflects the fact that the statement $*S$ 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 ($\leftarrow^r$) 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

$$\frac{n : p \rightarrow_m (A', p')}{n :\ (p, rs) \rightarrow_m (A', p', rs)} \qquad \frac{\begin{array}{c} S_1 :\ (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_2 :\ (p'', rs'') \rightarrow_m (A', p', rs') \end{array}}{S_1\, i_{\text{op}}\, S_2 : (p, rs) \rightarrow_m \left(\emptyset, p', rs' \cup \left\{S_1\, i_{\text{op}}\, S_2\right\}\right)}\ (i^r_{\text{op}}) \qquad \frac{x : p \rightarrow_m (A', p')}{x : (p, rs) \rightarrow_m (A', p', rs)}$$

$$\frac{\begin{array}{c} S_1\ :\ (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_2\ :\ (p'', rs'') \rightarrow_m (A', p', rs') \end{array}}{S_1\, b_{\text{op}}\, S_2 :\ (p, rs) \rightarrow_m \left(\emptyset, p', rs' \cup \left\{S_1\, b_{\text{op}}\, S_2\right\}\right)}\ (b^r_{\text{op}}) \qquad \frac{*S : p \rightarrow_m (A', p') \quad S : (p, rs) \rightarrow_m (A'', p'', rs'')}{*S : (p, rs) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}), & A'' = \{g\}; \\ (A', p', rs'' \cup \{*S\}), & |A''| \neq 1 . \end{cases}}\ (*^r)$$

$$\frac{}{skip : (p, rs) \rightarrow_m (\emptyset, p, rs)} \qquad \frac{x := S : p \rightarrow_m (A', p') \quad S : (p, rs) \rightarrow_m (A'', p'', rs'')}{x := S : (p, rs) \rightarrow_m (A', p', rs'' \setminus \{S \in Stmt \mid x \in free\,(S)\})}\ (:=^r)$$

$$\frac{\begin{array}{c} S_1 \leftarrow S_2 : p \rightarrow_m (A', p') \\ S_1 : (p, rs) \rightarrow_m (A_1, p_1, as_1) \end{array} \quad S_2 : (p_1, as_1) \rightarrow_m (A_2, p_2, as_2)}{S_1 \leftarrow S_2 : (p, rs) \rightarrow_m (A', p', as_2 \setminus (A' \cup \{S \mid S : p \rightarrow_m (A_s, —)\ \&\ A_s \cap A' \neq \emptyset\}))}\ (\leftarrow^r)$$

$$\frac{\begin{array}{c} S_1 : (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_2 : (p'', rs'') \rightarrow_m (A', p', rs') \end{array}}{S_1; S_2 : (p, rs) \rightarrow_m (A', p', rs')}\ (seq^r) \qquad \frac{\begin{array}{c} S : (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_t : (p, rs'') \rightarrow_m (A', p', rs') \\ S_f : (p, rs'') \rightarrow_m (A', p', rs') \end{array}}{if\ S\ then\ S_t\ else\ S_f : (p, rs) \rightarrow_m (A', p', rs')}\ (if^r)$$

$$\frac{S_1\,[S_2/x] : (p, rs) \rightarrow_m (A', p', rs')}{(\lambda x \cdot S_1)\, S_2 : (p, rs) \rightarrow_m (A', p', rs')}\ (appl^r) \qquad \frac{\begin{array}{c} S : (p, rs)\ \rightarrow\ (A'', p'', rs') \\ S_t : (p'', rs') \rightarrow_m (A', p', rs) \end{array}}{while\ S\ do\ S_t : (p, rs) \rightarrow_m (A', p', rs')}\ (whl^r)$$

$$\frac{S : (p, rs) \rightarrow_m (A', p', rs')}{\lambda x \cdot S : (p, rs) \rightarrow_m (A', p', rs')}\ (abs^r) \qquad \frac{fd\,(name) : (p, rs) \rightarrow_m (A', p', rs')}{name : (p, rs) \rightarrow_m (A', p', rs')}\ (name^r)$$

$$\frac{(\lambda x \cdot S')\, S : (p, rs) \rightarrow_m (A', p', rs')}{letrec\ x = S\ in\ S' : (p, rs) \rightarrow_m (A', p', rs')}\ (letrec^r) \qquad \frac{new_l : p \rightarrow_m (A', p')}{new_l : (p, rs) \rightarrow_m (A', p', rs)}\ (new^r)$$

$$\frac{\begin{array}{c} convert\,(S, n) : p \rightarrow_m (A', p') \\ S : (p, rs) \rightarrow_m (A, p'', rs') \end{array}}{convert\,(S, n) : (p, rs) \rightarrow_m (A', p', rs')}\ (convert^r) \qquad \frac{\begin{array}{c} transmit\ S_1\ from\ S_2 : p\ \rightarrow\ (A', p') \\ S_2 : (p, rs) \rightarrow_m (A_2, p_2, as_2) \\ S_1\ : (p_2, as_2) \rightarrow_m (A_1, p_1, rs') \end{array}}{transmit\ S_1\ from\ S_2 : (p, rs) \rightarrow_m (A', p', rs')}\ (trans^r)$$

$$\frac{\begin{array}{c} (p'_1, rs'_1) \leq (p_1, as_1) \\ S : (p_1, as_1) \rightarrow_m (p_2, as_2) \\ (p_2, as_2) \leq (p'_2, rs'_2) \end{array}}{S : (p'_1, rs'_1) \rightarrow_m (p'_2, rs'_2)}\ (csq^r) \qquad \frac{Defs : \emptyset \curvearrowright fd \quad S : (p, rs) \rightarrow_m (A', p', rs')}{Defs : S : (p, rs) \rightarrow_m (A', p', rs')}\ (prg^r)$$

Algorithm 1: Inference rules of the *ready* type system.

program point of S, the set of *ready* items. The following rule can be used to combine the information calculated for each machine to get new *ready* information for each program point. The new *ready* information is valid on any of the $|M|$ machines.

Consider

$$\frac{\forall m \in M \cdot S : \left(\sup\left\{p, p_j \mid j \neq i\right\}, \sup\left\{rs, rs_j \mid j \neq i\right\}\right) \longrightarrow_m (A_m, p_m, rs_m)}{S : (p, rs) \longrightarrow_M (\cup_i A_i, \sup\{p_1, \ldots, p_n\}, \sup\{rs_1, \ldots, rs_n\})}\ (main\text{-}rs) . \tag{1}$$

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

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* δ *on the execution path. Then the items of* rs' *are ready at the point corresponding to* δ' *on the execution path.*

3. *Semiexpectation* Analysis of $while_d$

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:

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

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: $\textit{points-to-types} \times \mathcal{P}(Stmt^{+} \cup gAddrs) \times \mathcal{P}(Stmt^{+} \cup gAddrs)$. The subtyping relation has the form $\leq_p \times \supseteq \times \supseteq$. A state on an execution path is of type $se \in \mathcal{P}(Stmt^{+} \cup gAddrs)$ if all elements of se are *semiexpectable* according to Definition 3. Judgments of the *semiexpectation* type system have the form $S : (p, rs, se) \rightarrow_m (A', p', rs', se')$. 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 $S : (p, rs, se) \rightarrow_m (A', p', rs', se')$. 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 (i^{e}_{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 rule ($*^{e}$). 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

$$\frac{\forall m \in M \cdot S : \left(\sup\left\{p, p_j \mid j \neq i\right\}, \sup\left\{rs, rs_j \mid j \neq i\right\}\right) \longrightarrow_m \left(A_m, p_m, rs_m\right)}{S : (p, rs) \longrightarrow_M \left(\cup_i A_i, \sup\left\{p_1, \ldots, p_n\right\}, \sup\left\{rs_1, \ldots, rs_n\right\}\right)} \times (\textit{main-rs})\,. \tag{3}$$

The difference in the way that this rule treats the *semiexpectable* information and the way *ready* information is treated is explained by the fact that the *ready* analysis is forward while the *semiexpectation* analysis is backward.

It is not hard to prove the soundness of the above type system.

Theorem 4. *Suppose that* $(S, \delta) \rightarrow (V, \delta'), S : (p, rs, se) \rightarrow (A', p', rs', se')$ *and the items of* se' *are semiexpectable at the point corresponding to* δ' *on the execution path. Then the items of* se *are semiexpectable at the point corresponding to* δ *on the execution path.*

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 $sn : S^{+} \rightarrow \textit{Stmt-names}$ 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')

$$\frac{n : p \rightarrow_m (A', p')}{n : (p, rs, se) \rightarrow_m (A', p', rs, se)} \qquad \frac{x : p \rightarrow_m (A', p')}{x : (p, rs, se) \rightarrow_m (A', p', rs, se)}$$

$$\frac{S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \quad S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se')}{S_1\, i_{\mathrm{op}}\, S_2 : \left(p, rs, se \cup \left(rs \cap \left\{S_1\, i_{\mathrm{op}}\, S_2\right\}\right)\right) \rightarrow_m \left(\emptyset, p', rs' \cup \left\{S_1\, i_{\mathrm{op}}\, S_2\right\}, se'\right)} (i^e_{\mathrm{op}})$$

$$\frac{S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \quad S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se')}{S_1\, b_{\mathrm{op}}\, S_2 : (p, rs, se) \rightarrow_m (\emptyset, p', rs' \cup \{S_1\, b_{\mathrm{op}}\, S_2\}, se')} (b^e_{\mathrm{op}})$$

$$\frac{*S : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'')}{*S : (p, rs, se \cup (re \cap \{*S, g\})) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}, se''), & A' = \{g\}; \\ (A', p', rs'' \cup \{*S\}, se''), & |A'| \neq 1 . \end{cases}} (*^e)$$

$$\frac{}{skip : (p, rs, se) \rightarrow_m (\emptyset, p, rs, se)} \qquad \frac{x := S : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'')}{x := S : (p, rs, se) \rightarrow_m (A', p', rs'' \setminus \{S \in Stmt \mid x \in free(S)\}, se'')} (:=^e)$$

$$\frac{\begin{array}{c} S_1 \leftarrow S_2 : p \rightarrow_m (A', p') \\ S_1 : (p, rs, se) \rightarrow_m (A_1, p_1, as_1, se_1) \end{array} \quad S_2 : (p_1, as_1, se_1) \rightarrow_m (A_2, p_2, as_2, se_2)}{S_1 \leftarrow S_2 : (p, rs, se) \rightarrow_m (A', p', as_2 \setminus (A' \cup \{S \mid S : p \rightarrow_m (A_s, \_) \,\&\, A_s \cap A' \neq \emptyset\}), se_2)} (\leftarrow^e)$$

$$\frac{\begin{array}{c} S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \\ S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se') \end{array}}{S_1 ; S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se')} (seq^e) \qquad \frac{\begin{array}{c} S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \\ S_t : (p, rs'', se'') \rightarrow_m (A', p', rs', se') \\ S_f : (p, rs'', se'') \rightarrow_m (A', p', rs', se') \end{array}}{if\ S\ then\ S_t\ else\ S_f : (p, rs, se) \rightarrow_m (A', p', rs', se')} (if^e)$$

$$\frac{S_1[S_2/x] : (p, rs, se) \rightarrow_m (A', p', rs', se')}{(\lambda x \cdot S_1) S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se')} (appl^e)$$

$$\frac{S : (p, rs, se) \rightarrow (A'', p'', rs', se') \quad S_t : (p'', rs', se') \rightarrow_m (A', p', rs, se)}{while\ S\ do\ S_t : (p, rs, se) \rightarrow_m (A', p', rs', se')} (whl^e)$$

$$\frac{fd(name) : (p, rs, se) \rightarrow_m (A', p', rs', se')}{name : (p, rs, se) \rightarrow_m (A', p', rs', se')} (name^e) \qquad \frac{S : (p, rs, se) \rightarrow_m (A', p', rs', se')}{\lambda x \cdot S : (p, rs, se) \rightarrow_m (A', p', rs', se')} (abs^e)$$

$$\frac{(\lambda x \cdot S') S : (p, rs, se) \rightarrow_m (A', p', rs', se')}{letrec\ x = S\ in\ S' : (p, rs, se) \rightarrow_m (A', p', rs', se')} (letrec^e) \qquad \frac{new_l : p \rightarrow_m (A', p')}{new_l : (p, rs, se) \rightarrow_m (A', p', rs, se)} (new^e)$$

$$\frac{convert\ (S, n) : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A, p'', rs', se')}{convert\ (S, n) : (p, rs, se) \rightarrow_m (A', p', rs', se')} (convert^e)$$

$$\frac{\begin{array}{c} transmit\ S_1\ from\ S_2 : p \rightarrow (A', p') \\ S_2 : (p, rs, se) \rightarrow_m (A_2, p_2, as_2, se_2) \end{array} \quad S_1 : (p_2, as_2, se_2) \rightarrow_m (A_1, p_1, rs', se')}{transmit\ S_1\ from\ S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se')} (trans^e)$$

$$\frac{\begin{array}{c} (p_1', rs_1', se_1') \leq (p_1, as_1, se_1) \\ S : (p_1, as_1, se_1) \rightarrow_m (p_2, as_2, se_2) \\ (p_2, as_2, se_2) \leq (p_2', rs_2', se_2') \end{array}}{S : (p_1', rs_1', se_1) \rightarrow_m (p_2', rs_2', se_2')} (csq^e) \qquad \frac{Defs : \emptyset \curvearrowright fd \quad S : (p, rs, se) \rightarrow_m (A, p', rs', se')}{Defs : S : (p, rs, se) \rightarrow_m (A, p', rs', se')} (prg^e)$$

Algorithm 2: Inference rules of the *semiexpectation* type system.

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 $(*^f_1)$, $(*^f_2)$, and $(*^f_3)$. The rule $(*^f_1)$ takes care of the case where $*S$ is *ready* and is replaceable by its name under the function sn. The rule $(*^f_2)$ 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 $(*^f_3)$ 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 $(i^f_{op(1)})$, $(i^f_{op(2)})$, and $(i^f_{op(3)})$ treat different cases for arithmetic statements. The Boolean statements are treated with rules quite similar to that of arithmetic statements. The rule (whl^f) reuses frequent substatements of the guard. This

$$\frac{n : p \rightarrow_m (A', p')}{n : (p, rs, se) \rightarrow_m (A', p', rs, se) \Rrightarrow (skip, n)} \qquad \frac{x : p \rightarrow_m (A', p')}{x : (p, rs, se) \rightarrow_m (A', p', rs, se) \Rrightarrow (skip, x)}$$

$$\frac{*S \in rs \quad *S : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns, S')}{*S : (p, rs, se \cup (re \cap \{*S, g\})) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}, se''), & A' = \{g\}; \\ (A', p', rs'' \cup \{*S\}, se''), & |A'| \neq 1 . \end{cases} \Rrightarrow (ns, sn(*S))} \left(*^f_1\right)$$

$$\frac{*S \notin rs \quad *S \in se' \quad *S : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns, S')}{\begin{array}{c} *S : (p, rs, se \cup (re \cap \{*S, g\})) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}, se''), & A' = \{g\}; \\ (A', p', rs'' \cup \{*S\}, se''), & |A'| \neq 1 . \end{cases} \\ \Rrightarrow (ns; sn(*S) := *S', sn(*S)) \end{array}} \left(*^f_2\right)$$

$$\frac{*S \notin rs \quad *S \notin se' \quad *S : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns, S')}{*S : (p, rs, se \cup (re \cap \{*S, g\})) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}, se''), & A' = \{g\} ; \\ (A', p', rs'' \cup \{*S\}, se''), & |A'| \neq 1 . \end{cases} \Rrightarrow (ns, *S')} \left(*^f_3\right)$$

$$\frac{S_1\, i_{\mathrm{op}}\, S_2 \in rs \quad \begin{array}{c} S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns_1, S'_1) \\ S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se') \Rrightarrow (ns_2, S'_2) \end{array}}{\begin{array}{c} S_1\, i_{\mathrm{op}}\, S_2 : \left(p, rs, se \cup \left(rs \cap \left\{S_1\, i_{op}\, S_2\right\}\right)\right) \rightarrow_m \left(\emptyset, p', rs' \cup \left\{S_1\, i_{\mathrm{op}}\, S_2\right\}, se'\right) \\ \Rrightarrow (ns_1; ns_2, sn(S_1\, i_{op}\, S_2)) \end{array}} (i^f_{\mathrm{op}(1)})$$

$$\frac{\begin{array}{c} S_1\, i_{\mathrm{op}}\, S_2 \notin rs \quad S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns_1, S'_1) \\ S_1\, i_{\mathrm{op}}\, S_2 \in se' \quad S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se') \Rrightarrow (ns_2, S'_2) \end{array}}{\begin{array}{c} S_1\, i_{\mathrm{op}}\, S_2 : \left(p, rs, se \cup \left(rs \cap \left\{S_1\, i_{op}\, S_2\right\}\right)\right) \rightarrow_m \left(\emptyset, p', rs' \cup \left\{S_1\, i_{\mathrm{op}}\, S_2\right\}, se'\right) \\ \Rrightarrow \left(ns_1; ns_2; sn\left(S_1\, i_{\mathrm{op}}\, S_2\right) := \left(S'_1\, i_{\mathrm{op}}\, S'_2\right), sn\left(S_1\, i_{\mathrm{op}}\, S_2\right)\right) \end{array}} \left(i^f_{\mathrm{op}(2)}\right)$$

$$\frac{\begin{array}{cc} S_1\, i_{\mathrm{op}}\, S_2 \notin rs & S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns_1, S'_1) \\ S_1\, i_{\mathrm{op}}\, S_2 \notin se' & S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se') \Rrightarrow (ns_2, S'_2) \end{array}}{\begin{array}{c} S_1\, i_{\mathrm{op}}\, S_2 : \left(p, rs, se \cup \left(rs \cap \left\{S_1\, i_{\mathrm{op}}\, S_2\right\}\right)\right) \rightarrow_m \left(\emptyset, p', rs' \cup \left\{S_1\, i_{\mathrm{op}}\, S_2\right\}, se'\right) \\ \Rrightarrow \left(ns_1; ns_2, S'_1\, i_{\mathrm{op}}\, S'_2\right) \end{array}} \left(i^f_{op(3)}\right)$$

$$\frac{}{skip : (p, rs, se) \rightarrow_m (\emptyset, p, rs, se) \Rrightarrow (skip, skip)}$$

$$\frac{x := S : p \rightarrow_m (A', p') \quad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (sn, S')}{x := S : (p, rs, se) \rightarrow_m (A', p', rs'' \setminus \{S \in Stmt \mid x \in free(S)\}, se'') \Rrightarrow (skip, ns; x := S')} \left(:=^f\right)$$

$$\frac{\begin{array}{c} S_1 : (p, rs, se) \rightarrow_m (A_1, p_1, as_1, se_1) \Rrightarrow (ns_1, S'_1) \\ S_2 : (p_1, as_1, se_1) \rightarrow_m (A_2, p_2, as_2, se_2) \Rrightarrow (ns_2, S'_2) \end{array} \quad S_1 \leftarrow S_2 : p \rightarrow_m (A', p')}{\begin{array}{c} S_1 \leftarrow S_2 : (p, rs, se) \rightarrow_m (A', p', as_2 \setminus (A' \cup \{S \mid S : p \rightarrow_m (A_s, -)\ \&\ A_s \cap A' \neq \emptyset\}), se_2) \\ \Rrightarrow (skip, ns_1; ns_2; S'_1 \leftarrow S'_2) \end{array}} (\leftarrow^f)$$

$$\frac{\begin{array}{c} S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \Rrightarrow (ns_1, S'_1) \\ S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se') \Rrightarrow (ns_2, S'_2) \end{array}}{S_1; S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rrightarrow (ns_1; ns_2, S'_1; S'_2)} (seq^f)$$

ALGORITHM 3: Inference rules for the *frequent statements and dereferences elimination* (1).

is done via adding *ns* 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 δ is a state defined on the set of locations, *Loc* ([6, Definition 4]). Suppose also that δ_* is a state defined on *Loc* $\cup$ *Stmt-names*. The expression $\delta \equiv_{se} \delta_*$ denotes the fact that δ and δ_* are equivalent with respect to the *semiexpectation* type *se*. More precisely $\delta \equiv_{se} \delta_*$ if and only if

(1) for all $j \in Loc$. $\delta(j) = \delta_*(j)$, and

(2) for all $S \in se$. $(S, \delta) \rightsquigarrow_m (v, \delta') \Rightarrow \delta_*(sn(S)) = v$.

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 *Loc*, and

(ii) if a statement is both *ready* and *semiexpectable*, then its semantics in the original-program state equals

$$\frac{S:(p,rs,se)\rightarrow_m(A'',p'',rs'',se'')\Rrightarrow(ns,S') \quad \begin{array}{c} S_t:(p,rs'',se'')\rightarrow_m(A',p',rs',se')\Rrightarrow(ns_t,S'_t) \\ S_f:(p,rs'',se'')\rightarrow_m(A',p',rs',se')\Rrightarrow(ns_f,S'_f)\end{array}}{\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(\mathit{skip},ns;\mathit{if}\ S'\ \mathit{then}\ ns_t;S'_t\ \mathit{else}\ ns_f;S'_f)}(\mathit{if}^f)$$

$$\frac{(\lambda x\cdot S_1)[S_2/x]:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S')}{(\lambda x\cdot S_1)S_2:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S')}(\mathit{appl}^f)$$

$$\frac{S:(p,rs,se)\ \rightarrow\ (A'',p'',rs',se')\Rrightarrow(ns,S') \quad S_t:(p'',rs',se')\rightarrow_m(A',p',rs,se)\Rrightarrow(ns_t,S'_t)}{\mathit{while}\ S\ \mathit{do}\ S_t:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(\mathit{skip},ns;\mathit{while}\ S'\ \mathit{do}\ (ns_t;S'_t;ns))}(\mathit{whl}^f)$$

$$\frac{\mathit{fd}(\mathit{name}):(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S')}{\mathit{name}:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S')}(\mathit{name}^f)$$

$$\frac{S:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S')}{\lambda x\cdot S:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(\mathit{skip},ns;\lambda x\cdot S')}(\mathit{abs}^f)$$

$$\frac{\mathit{new}_l:p\rightarrow_m(A',p')}{\mathit{new}_l:(p,rs,se)\rightarrow_m(A',p',rs,se)\Rrightarrow(\mathit{skip},\mathit{new}_l)}(\mathit{new}^f)$$

$$\frac{(\lambda x\cdot S')S:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S'')}{\mathit{letrec}\ x=S\ \mathit{in}\ S':(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S'')}(\mathit{letrec}^f)$$

$$\frac{\mathit{convert}\ (S,n):p\rightarrow_m(A',p') \quad S:(p,rs,se)\rightarrow_m(A,p'',rs',se')\Rrightarrow(ns,S')}{\mathit{convert}\ (S,n):(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(\mathit{skip},ns;\mathit{convert}\ (S',n))}(\mathit{convert}^f)$$

$$\frac{\begin{array}{l}\mathit{transmit}\ S_1\ \mathit{from}\ S_2:p\ \rightarrow\ (A',p') \\ S_2:(p,rs,se)\rightarrow_m(A_2,p_2,as_2,se_2)\Rrightarrow(ns_2,S'_2) \\ S_1:(p_2,as_2,se_2)\rightarrow_m(A_1,p_1,rs',se')\Rrightarrow(ns_1,S'_1)\end{array}}{\mathit{transmit}\ S_1\ \mathit{from}\ S_2:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns_1;ns_2,\mathit{transmit}\ S'_1\ \mathit{from}\ S'_2)}(\mathit{trans}^f)$$

$$\frac{\begin{array}{l}(p'_1,rs'_1,se'_1)\le(p_1,as_1,pa_1) \\ (p_2,as_2,pa_2)\le(p'_2,rs'_2,se'_2)\end{array} \quad S:(p_1,as_1,pa_1)\rightarrow_m(p_2,as_2,pa_2)\Rrightarrow(ns,S')}{S:(p'_1,rs'_1,pa_1)\rightarrow_m(p'_2,rs'_2,se'_2)\Rrightarrow(ns,S')}(\mathit{csq}^f)$$

$$\frac{\mathit{Defs}:\emptyset\curvearrowright\mathit{fd} \quad S:(p,rs,se)\rightarrow_m(A,p',rs',se')\Rrightarrow(ns,S')}{\mathit{Defs}:S:(p,rs,se)\rightarrow_m(A,p',rs',se')\Rrightarrow\mathit{Defs}:ns;S'}(\mathit{prg}^f)$$

Algorithm 4: Inference rules for the *frequent statements and dereferences elimination* (2).

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* $S:(p,rs,se)\rightarrow_m(A',p',rs',se')\Rrightarrow(ns,S')$ *and* $\delta\equiv_{se}\delta_*$. *Then*

(i) $(S,\delta)\rightsquigarrow_m(v,\delta')\ \Rightarrow\ \exists\delta'_*.\ \delta'\equiv_{se'}\delta'_*$ *and* $(S',\delta_*)\rightsquigarrow_m(v,\delta'_*)$;

(ii) $(S',\delta_*)\rightsquigarrow_m(v,\delta'_*)\ \Rightarrow\ \exists\delta'.\ \delta'\equiv_{se'}\delta'_*$ *and* $(S,\delta)\rightsquigarrow_m(v,\delta')$.

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

$\mathscr{D} = \mathbb{Z} \cup gAddrs \cup \{true, false, \bot\}$	(model values)	$d \in D$
$\mathscr{M} = \{1, 2, \ldots, \delta\}$	(the set of machine identifiers)	$m \in \mathscr{M}$
$\mathscr{C}$	(the set of class names)	$C, D \in \mathscr{C}$
$lVar$	(the set of local variables on each machine)	$o \in lVar$
$gVar = lVar \times \mathscr{M}$	(the set of global variables on all machines)	$(o, m) \in gVar$
$iVar_C$	(instance variables of C)	$v \in iVar_C$
$lAddrs$	(the set of local addresses on each machine)	$\alpha \in lAddrs$
$gAddrs = lAddrs \times \mathscr{M}$	(the set of global addresses on all machine)	$(\alpha, m) \in gAddrs$
$\mathscr{S} = \{s \mid s : gVar \rightarrow_p \mathscr{D}\}$	(the set of stacks)	$s \in \mathscr{S}$
$\mathscr{O} = \{I_C \mid I_C : iVar_C \rightarrow_p \mathscr{D}\}$	(the set of objects contents)	$I_C \in \mathscr{O}$
$\mathscr{H} = gAddrs \rightarrow_p \mathscr{O}$	(the set of heaps)	$h \in \mathscr{H}$
$States = \mathscr{S} \times \mathscr{H}$	(a memory state)	$(s, h) \in States$
$this$	(the current active object)	
$FunNames$	(the set of function names)	$f \in FunNames$
Fun_C	(functions of C)	
$Fun\text{-}def_C = Fun_C \rightarrow LVar \times Stmt \times AExpr$	(functions definitions)	$F_C \in Fun\text{-}def_C$
$f \mapsto (p_f, S_f, e_f)$		
$C \ll D$	(C is a subclass of a class D)	
$\leq$	(the ref. tran. closure of $\ll$)	
$Loc = gAddrs \cup gVar \cup (\cup_{C \in \mathscr{C}, v \in iVar_C} \{(C, v)\})$	(the set of all locations)	$l \in Loc$

Figure 4: Semantic spaces and naming conventions.

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. The judgments of the proposed analysis have the form $S : p \rightarrow Cs$. The intuition of such judgments is that the pointer information are used to calculate the set Cs. In the rule $(:=^r_{S\cdot v})$, *ready* substatements of S_1 and S_2 are added to rs to produce rs'. Then for any class C that S_1 may reference, statements involving $C \cdot v$ are removed from rs'. In the rule $(:=^r_{o\cdot f})$, Cs includes classes that o_2 may reference. For all functions named f in classes of Cs, the body and return statements are enumerated in the set $\{S_1, \ldots, S_m\}$. *Ready* substatements of these statements are added to rs to produce rs_{m+1}. Then all statements involving o_1 are removed from rs_{m+1}.

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

Theorem 7. *Suppose that* $(S, s, h) \rightarrow (V, s', h'), S : (p, rs) \rightarrow (A', p', rs')$ *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 *semiexpectation* for object-oriented distributed programs of Figure 3. Some comments on the inference rules of Algorithm 6 are in order. In the rule $(:=^e_{S\cdot v})$, starting with the posttype se', the pretype se'' is calculated for the statement S_2. Then se' is used as a posttype for S_1 to get the main pretype se. Similarly to $(:=^r_{o\cdot f})$, the rule $(:=^e_{o\cdot f})$ 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* $(S, s, h) \rightarrow (V, s', h'), S : (p, rs, se) \rightarrow (A', p', rs', se')$, *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.*

Adding rules of Algorithm 7 to that of Algorithm 3 results in the main type system achieving the analysis of *frequent statement and dereference elimination* for object-oriented distributed programs of Figure 3. We note the following on the inference rules. Optimization is based on rules for $(C)S$; $((C)S_1^f)$, $((C)S_2^f)$, and $((C)S_3^f)$. The case that $(C)S$ is *ready* and is replaceable by its name under the function sn is treated by $((C)S_1^f)$. The case $(C)S$ is *semiexpectable* but not *ready* before calculating the statement is treated by $((C)S_2^f)$. The rule $((C)S_3^f)$ takes care of the case, where $(C)S$ is neither *ready* before the calculation nor *semiexpectable* after execution.

$$\frac{o\cdot v : p \to_m (A',p')}{o\cdot v : (p,rs) \to_m (A',p',rs)} \qquad \frac{\text{this} \ : p \to_m (A',p')}{\text{this} \ : (p,rs) \to_m (A',p',rs)} \qquad \frac{S : (p,rs) \to_m (A',p',rs')}{(C)S : (p,rs) \to_m (A',p',rs' \cup \{(C)S\})}$$

$$\frac{\begin{array}{cc} S_1\cdot v := S_2 : p \to_m (A',p') & S_1 : (p,rs) \to_m (A_1,p_1,rs_1) \\ S_1 : p \ \to \ Cs & S_2 : (p_1,rs_1) \to_m (A_2,p',rs') \end{array}}{S_1\cdot v := S_2 : (p,rs) \to_m (A',p',rs' \setminus \{S \in Stmts \mid C \in Cs \land C\cdot v \in free(S)\})} \left(:=^r_{S\cdot v}\right)$$

$$\frac{\begin{array}{c} o_1 := o_2\cdot f(S) : p \to_m (A',p') \quad \{S_1,\ldots,S_m\} = \{S_f, S_r \mid \exists C \in Cs \land F_C(f) = (p,S_f,S_r)\} \\ S : (p,rs) \to_m (A_1,p_1,rs_1) \quad o_2 : p \ \to \ Cs \quad S_i : (p_i,rs_i) \to_m (A_{i+1},p_{i+1},rs_{i+1}) \end{array}}{o_1 := o_2\cdot f(S) : (p,rs) \to_m (A',p',rs_{m+1} \setminus \{S \in Stmts \mid o_1 \in free(S)\})} \left(:=^r_{o\cdot f}\right)$$

$$\frac{\begin{array}{c} o_1 := super\cdot f(S) : p \to_m (A',p') \quad S : (p,rs) \to_m (A_1,p_1,rs_1) \\ this : p \ \to \ Cs' \quad S_i : (p_i,rs_i) \to_m (A_{i+1},p_{i+1},rs_{i+1}) \\ \{S_1,\ldots,S_m\} = \{S_f, S_r \mid \exists D \in Cs' \land super(D,f) = C \land F_C(f) = (p,S_f,S_r)\} \end{array}}{o_1 := super\cdot f(S) : (p,rs) \to_m (A',p',rs_{m+1} \setminus \{S \in Stmts \mid o_1 \in free(S)\})} \left(:=^r_{super}\right)$$

$$\frac{o \ := \ \text{new}\ C : p \to_m (A',p')}{o \ := \ \text{new}\ C : (p,rs) \to_m (A',p',rs \setminus \{S \in Stmts \mid o \in free(S)\})} \left(:=^r_{new}\right)$$

Algorithm 5: An extension for the *ready* type system to cover *OODP* programs.

$$\frac{o\cdot v : p \to_m (A',p')}{o\cdot v : (p,rs,se) \to_m (A',p',rs,se)} \qquad \frac{\text{this} : p \to_m (A',p')}{\text{this} : (p,rs,se) \to_m (A',p',rs,se)}$$

$$\frac{S : (p,rs) \to_m (A',p',rs')}{(C)\,S : (p,rs,se \cup (rs \cap \{(C)S\})) \to_m (A',p',rs' \cup \{(C)S\},se')}$$

$$\frac{\begin{array}{cc} S_1\cdot v := S_2 : p \to_m (A',p') & S_1 : (p,rs,se) \to_m (A_1,p_1,rs_1,se_1) \\ S_1 : p \ \to \ Cs & S_2 : (p_1,rs_1,se_1) \to_m (A_2,p',rs',se') \end{array}}{S_1\cdot v := S_2 : (p,rs,se) \to_m (A',p',rs' \setminus \{S \in Stmts \mid C \in Cs \land C\cdot v \in free(S)\},se')} \left(:=^e_{S\cdot v}\right)$$

$$\frac{\begin{array}{c} o_1 := o_2\cdot f\,(S) : p \to_m (A',p') \quad \{S_1,\ldots,S_m\} = \left\{S_f, S_r \mid \exists C \in Cs \land F_C(f) = \left(p,S_f,S_r\right)\right\} \\ S : (p,rs,se) \to_m (A_1,p_1,rs_1,se_1) \quad o_2 : p \ \to \ Cs \quad S_i : (p_i,rs_i,se_i) \to_m (A_{i+1},p_{i+1},rs_{i+1},se_{i+1}) \end{array}}{o_1 := \ o_2\cdot f(S) : (p,rs,se) \to_m (A',p',rs_{m+1} \setminus \{S \in Stmts \mid o_1 \in free(S)\},se_{m+1})} \left(:=^e_{o\cdot f}\right)$$

$$\frac{\begin{array}{c} o_1 := super\cdot f\,(S) : p \to_m (A',p') \quad S : (p,rs,se) \to_m (A_1,p_1,rs_1,se_1) \\ this : p \ \to \ Cs' \quad S_i : (p_i,rs_i,se_i) \to_m (A_{i+1},p_{i+1},rs_{i+1},se_{i+1}) \\ \{S_1,\ldots,S_m\} = \{S_f, S_r \mid \exists D \in Cs' \land super(D,f) = C \land F_C(f) = (p,S_f,S_r)\} \end{array}}{o_1 := super\cdot f(S) : (p,rs,se) \to_m (A',p',rs_{m+1} \setminus \{S \in Stmts \mid o_1 \in free(S)\},se_{m+1})} \left(:=^e_{super}\right)$$

$$\frac{o \ := \ \text{new}\,C : p \to_m (A',p')}{o \ := \ \text{new}\,C : (p,rs,se) \to_m (A',p',rs \setminus \{S \in Stmts \mid o \in free(S)\},se)} \left(:=^e_{new}\right)$$

Algorithm 6: An extension for the *semi-expectation* type system.

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, Definition 4]). Suppose also that (s_*,h_*) is a state defined on $Loc \cup Stmt\text{-}names$. The expression $(s,h) \equiv_{se} (s_*,h_*)$ denotes the fact that (s,h) and (s_*,h_*) are equivalent with respect to the *semiexpectation* type *se*. More precisely $(s,h) \equiv_{se} (s_*,h_*)$ if and only if

(1) for all $l \in Loc.\ s(l) = s_*(l)$, and

(2) for all $S \in se.\ (S,s,h) \to_m (v,s',h') \Rightarrow s_*(sn(S)) = v$.

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

Theorem 10. *Suppose that* $S \ : \ (p,rs,se) \to_m (A',p',rs',se') \Rrightarrow (ns,S')$ *and* $(s,h) \equiv_{se} (s_*,h_*)$. *Then*

(i) $(S,s,h) \to_m (v,s',h') \Rightarrow \exists (s'_*,h'_*).\ (s',h') \equiv_{se'} (s'_*,h'_*)$ *and* $(S',s_*,h_*) \to_m (v,s'_*,h'_*)$;

(ii) $(S',s_*,h_*) \to_m (v,s'_*,h'_*) \Rightarrow \exists (s',h').\ (s',h') \equiv_{se'} (s'_*,h'_*)$ *and* $(S,s,h) \to_m (v,s',h')$.

$$\frac{o\cdot v : p \rightarrow_m (A', p')}{o\cdot v : (p, rs, se) \rightarrow_m (A', p', rs, se) \Rrightarrow (\text{skip}\ , o\cdot v)} \qquad \frac{\text{this}\ : p \rightarrow_m (A', p')}{\text{this}\ : (p, rs, se) \rightarrow_m (A', p', rs, se) \Rrightarrow (\text{skip}\ ,\ \text{this})}$$

$$\frac{(C)\,S \in rs \quad S : (p, rs) \rightarrow_m (A', p', rs') \Rrightarrow (ns, S')}{(C)\,S : (p, rs, se \cup (rs \cap \{(C)\,S\})) \rightarrow_m (A', p', rs' \cup \{(C)\,S\}, se') \Rrightarrow (ns, ns\,((C)\,S))} \left((C)\,S_1^f\right)$$

$$\frac{(C)\,S \notin se' \setminus rs \quad S : (p, rs) \rightarrow_m (A', p', rs') \Rrightarrow (ns, S')}{\begin{array}{c}(C)\,S : (p, rs, se \cup (rs \cap \{(C)\,S\})) \rightarrow_m (A', p', rs' \cup \{(C)\,S\}, se') \\ \Rrightarrow (ns; ns\,((C)\,S) = (C)\,S', ns\,((C)\,S))\end{array}} \left((C)\,S_2^f\right)$$

$$\frac{(C)\,S \notin rs \cup se' \quad S : (p, rs) \rightarrow_m (A', p', rs') \Rrightarrow (ns, S')}{(C)\,S : (p, rs, se \cup (rs \cap \{(C)\,S\})) \rightarrow_m (A', p', rs' \cup \{(C)\,S\}, se') \Rrightarrow (ns, (C)\,S')} \left((C)\,S_3^f\right)$$

$$\frac{\begin{array}{cc}S_1 \cdot v := S_2 : p \rightarrow_m (A', p') & S_1 : (p, rs, se) \rightarrow_m (A_1, p_1, rs_1, se_1) \Rrightarrow (ns_1, S_1') \\ S_1 : p \rightarrow Cs & S_2 : (p_1, rs_1, se_1) \rightarrow_m (A_2, p', rs', se') \Rrightarrow (ns_2, S_2')\end{array}}{\begin{array}{l}S_1 \cdot v := S_2 : (p, rs, se) \rightarrow_m (A', p', rs' \setminus \{S \in Stmts \mid C \in Cs \wedge C\cdot v \in free\,(S)\}, se') \\ \quad \Rrightarrow (ns_1; ns_2, S_1' \cdot v := S_2')\end{array}} \left(:=_{S\cdot v}^f\right)$$

$$\frac{\begin{array}{c}o_1 := o_2 \cdot f(S) : p \rightarrow_m (A', p') \quad o_2 : p \rightarrow Cs \\ S : (p, rs, se) \rightarrow_m (A_1, p_1, rs_1, se_1) \Rrightarrow (ns, S') \\ \{S_1, \ldots, S_m\} = \{S_f, S_r \mid \exists C \in Cs \wedge F_C(f) = (p, S_f, S_r)\} \\ S_i : (p_i, rs_i, se_i) \rightarrow_m (A_{i+1}, p_{i+1}, rs_{i+1}, se_{i+1}) \Rrightarrow (ns_i, S_i')\end{array}}{\begin{array}{l}o_1 := o_2 \cdot f(S) : (p, rs, se) \rightarrow_m (A', p', rs_{m+1} \setminus \{S \in Stmts \mid o_1 \in free(S)\}, se_{m+1}) \\ \quad \Rrightarrow (ns; ns_1; \ldots; ns_m, o_1 := o_2 \cdot f(S'))\end{array}} \left(:=_{o\cdot f}^f\right)$$

$$\frac{\begin{array}{cc}o_1 := super \cdot f(S) : p \rightarrow_m (A', p') & S : (p, rs, se) \rightarrow_m (A_1, p_1, rs_1, se_1) \Rrightarrow (ns, S') \\ this : p \rightarrow Cs' & S_i : (p_i, rs_i, se_i) \rightarrow_m (A_{i+1}, p_{i+1}, rs_{i+1}, se_{i+1}) \Rrightarrow (ns_i, S_i') \\ \multicolumn{2}{c}{\{S_1, \ldots, S_m\} = \{S_f, S_r \mid \exists D \in Cs' \wedge super(D, f) = C \wedge F_C(f) = (p, S_f, S_r)\}}\end{array}}{\begin{array}{l}o_1 := super \cdot f(S) : (p, rs, se) \rightarrow_m (A', p', rs_{m+1} \setminus \{S \in Stmts \mid o_1 \in free(S)\}, se_{m+1}) \\ \quad \Rrightarrow (ns; ns_1; \ldots; ns_m, o_1 := super \cdot f(S'))\end{array}} \left(:=_{super}^f\right)$$

$$\frac{o := \text{new}\,C : p \rightarrow_m (A', p')}{o\ :=\ \text{new}\,C : (p, rs, se) \rightarrow_m (A', p', rs \setminus \{S \in Stmts \mid o \in free\,(S)\}, se) \Rrightarrow (skip, o := \text{new}\,C)} \left(:=_{new}^f\right)$$

Algorithm 7: An extension for the *frequent statements and de-references elimination* type system.

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 Elitist 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 acclivities [18, 21] was devoted to analyze distributed programs. 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

[1] S. S. Barpanda and D. P. Mohapatra, "Dynamic slicing of distributed object-oriented programs," *IET Software*, vol. 5, no. 5, pp. 425–433, 2011.

[2] C. Seragiotto Jr. and T. Fahringer, "Performance analysis for distributed and parallel Java programs with Aksum," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05)*, IEEE Computer Society, pp. 1024–1031, May 2005.

[3] H.-L. Truong and T. Fahringer, "Soft computing approach to performance analysis of parallel and distributed programs," in *Proceedings of the 11th International Euro-Par Conference (Euro-Par '05)*, J. C. Cunha and D. Pedro, Eds., vol. 3648 of *Lecture Notes in Computer Science*,, pp. 50–60, September 2005.

[4] M. A. El-Zawawy, "Probabilistic pointer analysis for multithreaded programs," *ScienceAsia*, vol. 37, no. 4, pp. 344–354, 2011.

[5] M. A. El-Zawawy, "Detection of probabilistic dangling references in multi-core programs using proof-supported tools," in *Proceedings of the International conference on Computational Science and Its Applications (ICCSA '13)*, B. Murgante, S. Misra, M. Carlini et al., Eds., vol. 7975 of *Lecture Notes in Computer Science*, pp. 516–530, Springer, 2013.

[6] M. A. El-Zawawy, "Frequent statement and de-reference elimination for distributed programs," in *Proceedings of the International conference on Computational Science and Its Applications (ICCSA '13)*, B. Murgante, S. Misra, M. Carlini et al., Eds., vol. 7973 of *Lecture Notes in Computer Science*, pp. 82–97, Springer, 2013.

[7] M. A. El-Zawawy, "Abstraction analysis and certified flow and context sensitive points-to relation for distributed programs," in *Proceedings of the International conference on Computational Science and Its Applications (ICCSA '12)*, B. Murgante, O. Gervasi, S. Misra et al., Eds., vol. 7336 of *Lecture Notes in Computer Science*, pp. 83–99, 2012.

[8] H. Ho, V. Szwarc, and T. Kwasniewski, "Low complexity reconfigurable DSP circuit implementations based on common sub-expression elimination," *Journal of Signal Processing Systems*, vol. 61, no. 3, pp. 353–365, 2010.

[9] S. Gopalakrishnan and P. Kalla, "Algebraic techniques to enhance common sub-expression elimination for polynomial system synthesis," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 1452–1457, April 2009.

[10] A. Saabas and T. Uustalu, "Program and proof optimizations with type systems," *Journal of Logic and Algebraic Programming*, vol. 77, no. 1-2, pp. 131–154, 2008.

[11] A. Nicolau, S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, and R. Gupta, "Dynamic common sub-expression elimination during scheduling in high-level synthesis," in *Proceedings of the 15th International Symposium on System Synthesis*, IEEE Computer Society, pp. 261–266, October 2002.

[12] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing polynomial expressions by algebraic factorization and common subexpression elimination," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2012–2021, 2006.

[13] P. Maragathavalli and S. Kanmani, "Evolutionary multi-objective optimization for data-flow testing of object-oriented programs," in *Proceedings of the International Conference on Advances in Computing and Information Technology (ACITY '12)*, N. Meghanathan, D. Nagamalai, and N. Chaki, Eds., vol. 177 of *Advances in Intelligent Systems and Computing*, pp. 263–271, Springer, 2012.

[14] K. Heffner, D. Tarditi, and M. D. Smith, "Extending object-oriented optimizations for concurrent programs," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, IEEE Computer Society, pp. 119–129, September 2007.

[15] S. Kumar and D. P. Agrawal, "A concurrency abstraction model for avoiding inheritance anomaly in object-oriented programs," in *Compiler Optimizations for Scalable Parallel Systems Languages*, S. Pande and D. P. Agrawal, Eds., vol. 1808 of *Lecture Notes in Computer Science*, pp. 109–140, Springer, 2001.

[16] M. A. El-Zawawy, "Dead code elimination based pointer analysis for multithreaded programs," *Journal of the Egyptian Mathematical Society*, vol. 20, no. 1, pp. 28–37, 2012.

[17] M. A. El-Zawawy, "Heap slicing using type systems," in *Proceedings of the International conference on Computational Science

and Its Applications (ICCSA '12), B. Murgante, O. Gervasi, S. Misra et al., Eds., vol. 7335 of *Lecture Notes in Computer Science*, pp. 592–606, Springer, 2012.

[18] A. Kamil and K. Yelick, "Hierarchical pointer analysis for distributed programs," *Proceedings of the 14th International Static Analysis Symposium (SAS '07)*, vol. 4634, pp. 281–297, 2007.

[19] B. Liblit and A. Aiken, "Type systems for distributed data structures," in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles og Programming Languages (POPL '00)*, pp. 199–213, January 2000.

[20] B. Liblit, A. Aiken, and K. A. Yelick, "Type systems for distributed data sharing," in *Proceedings of the International Static Analysis Symposium (SAS '03)*, R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, pp. 273–294, Springer, 2003.

[21] P. Lindberg, J. Leingang, D. Lysaker, S. U. Khan, and J. Li, "Comparison and analysis of eight scheduling heuristics for the optimization of energy consumption and makespan in large-scale distributed systems," *Journal of Supercomputing*, vol. 59, no. 1, pp. 323–360, 2012.

[22] S. Simmons, D. Edwards, and P. Kearns, "Communication analysis of distributed programs," *Scientific Programming*, vol. 14, no. 2, pp. 151–170, 2006.

[23] Y.-C. Chiu, C.-K. Shieh, T.-C. Huang, T.-Y. Liang, and K.-C. Chu, "Data race avoidance and replay scheme for developing and debugging parallel programs on distributed shared memory systems," *Parallel Computing*, vol. 37, no. 1, pp. 11–25, 2011.

[24] V. V. Toporkov, "Dataflow analysis of distributed programs using generalized marked nets," in *Proceedings of the International Conference on Dependability of Computer Systems (DepCoS- RELCOMEX '07)*, pp. 73–80, June 2007.