Benha University
Faculty of science
Dept. Mathematics

On Semantics of Programming Languages

By

HAMADA ALI MOHAMED ALI NAYEL

SUBMITTED FOR THE DEGREE OF MASTER IN COMPUTER SCIENCE
FACULTY OF SCIENCE, BENHA UNIVERSITY
BENHA, EGYPT

SUPERVISORS

Prof. Maher Zayed
Benha University
Faculty of Science
Department of Mathematics

Dr. Mohamed El Zawawy
Cairo University
Faculty of Science
Department of Mathematics

2012

APPROVAL SHEET

Title of M. Thesis

On Semantics of Programming Languages

Name of candidate

Hamada Ali Mohamed Ali Nayel

Submitted to

Faculty of Science - Benha University

Supervision Committee

Name	Position	Signature
Prof. Dr. Maher Shedid Zayed	Professor of Mathematics Department of Mathematics Faculty of Science Benha University	
Dr. Mohamed El Zawawy	Lecturer of Computer Science Department of Mathematics Faculty of Science Cairo University	

The Head of Mathematics Dept.

Prof. Dr. Abd El-Kareem Soliman

Addresses

Supervisors

- **Prof. Maher Shedid Zayed**

  Mathematics Department

  Faculty of Science

  Benha University

  Benha, Egypt

  email: maherzayed@hotmail.com

- **Dr Mohamed Abd El-Moneim El-Zawawy**

  Department of Mathematics

  Faculty of Science

  Cairo University

  Giza, Egypt

  email: maelzawawy@gmail.com

Author

- **Hamada Ali Mohamed Ali Nayel**

  Computer Science Department

  Faculty of Computers and Informatics

  Benha University

  Benha, Egypt

  email: hamada.ali@fci.bu.edu.eg

Acknowledgments

All gratitude and thankfulness to ALLAH for guiding and aiding me to bring this work out to light. It is impossible to give sufficient thanks to the people who gave help and advice (both taken and ignored) during the writing of this thesis. However, I would like to single out both my supervisors for their valuable discussions, enriching collaborations and their vital comments on the contents of this thesis. Without their kind help and generous encouragement, this work would not have seen the light.I am deeply indebted to Dr. *Mohamed El-Zawawy* for his kindness and emotional support throughout the period it took to prepare this work in its final form. I am particularly grateful for his genuine concern and his prompt replies to all the questions I addressed to him, for giving so generously of his time in revising this work and, in the process, pointing out some relevant and interesting ideas. I also wish to express my deep gratitude to Professor *Maher Zayed* for his guidance and moral support during the making of this work and also for his insightful remarks, fruitful suggestions and constructive criticism. Without his broad outlook and continual help, many aspects of this work would have remained unclarified. I also wish to express my deep gratitude to the staff of the Department of Mathematics-Faculty of Science-Benha University-Egypt especially Professor *Farouk El-Batanony* and Professor *Abd El-Kareem Soliman* for their guidance and moral support during my undergraduate studying and the making of this work. It is of course impossible to mention the names of everyone - teachers, friends and colleagues - who have taken part in my mathematical formation. Finally, I would like to specifically thank my family: my late father, my mother, my brother, my wife, my son and my sisters for their moral support and encouragement.

Abstract

Multi-threaded programs have many applications which are widely used such as operating systems. Analyzing multi-threaded programs differs from sequential ones; the main feature is that many threads execute at the same time. The effect of all other running threads must be taken in account. This these focuses on the analysis of multi-threaded programs. The first aim of our work is to implement partial redundancy elimination for multi-threaded programs via type systems. Partial redundancy elimination is among the most powerful compiler optimization: it performs loop invariant code motion and common subexpression elimination. In chapter 3, we present a type system with optimization component which performs partial redundancy elimination for a multi-threaded programs. In chapter 4, we designed a type systems based data race detector. Data race occurs when two threads try to access a shared variable at the same time without a proper synchronization. A detector is a software that determines if the program contains a data-race problem or not. In this these we develop a detector that has the form of a type system. We present a type system which discovers the data-race problems. We also prove the soundness of our type system.

Contents

1. Introduction

Chapter 1

Introduction

1.1 Background

1.1.1 Operational Semantics

The main constituents in designing, analyzing, and using programming languages are:-

Syntax Symbols and formal description to create well-formed expressions, sentences and programs in the language. Syntax deals solely with the form and structure of symbols in a language without any consideration given to their meaning.

Semantics The meaning of syntactically valid sentence in a language. Semantics describes the behavior that a computer follows when executing a program in a language.

Pragmatics Aspects of the language such as utility and scope of applications. Pragmatics includes issues such as ease of implementation, efficiency and programming methodology.

This thesis concerns with semantics. Semantics fall broadly into three categories: denotational, axiomatic, and operational. In a *denotational semantics* [71, 67, 43, 24], the meaning of programs is defined abstractly with suitable mathematical structure. In an *axiomatic semantics* (also called "*program logic*"), meaning is defined indirectly via axioms and rules of some logic of program prosperities. *Hoare logic* [27] is an instance of program logic whose axioms have the form $\{P\}C\{Q\}$. This means, if the pre-condition P is satisfied before running the program C and if C terminates then the post-condition Q will be satisfied afterwards. In an *operational semantics* (also called *"natural semantics"*), the meaning of programs is defined in terms of their behaviour (i.e the steps of computation they can take during program execution).

All of these categories are related and complement each other[45, 42, 1]. The noticeable difference between categories is that denotational and axiomatic semantic are commonly used with semantic studies in mathematical logic and linguistics, while operational semantic is distinctive in computer science. In this thesis we will use operational semantics to represent the execution of programs.

Operational Semantics Operational semantics is easy to understand and related to practical concerns and the mathematical theory. Programs can be operationally described with simple notion of *transition system*. A transition system generally contains a set of configurations(always called "*state*") and a relation. A *state* contains the elements of interest, depends on the purpose of study. It may contains variables, pointers or locations. A relation defines the rules of each phrase of the program.

1.1.2 Phases of Compilation Process

Compilation process has four basic phases:-

1. **Analysis phases**

   Which conceptually contain three phases lexical analysis, syntax analysis, and semantic analysis.

2. **Intermediate code generation**

   After analysis phases, some compilers generate an explicit intermediate representation of the source program. We can think of this intermediate representation as a program for an abstract machine. It should be easy to produce, and easy to translate into the target program.

3. **Code optimization**

   The code optimization phase attempts to improve the intermediate code. There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called optimizing compilers.

4. **Code generation**

   The final phase of the compiler is the generation of target code consisting normally of relocatable machine code or assembly code.

1.1.3 Compiler Optimization

The optimization is a transformation that improves the performance of the target code. Improving includes minimizing code and minimizing number of calculations. An optimization must satisfy the following conditions to be true:-

1. Must not change the output.

2. Must not cause errors that were not present in the original program.

3. Must be worth the effort.

There are two types of optimizations. *Local code optimization*: code improvement within a basic block. *Global code optimization*: improvements take into account what happens across basic blocks. Most global optimizations are based on *data-flow analysis*, which are algorithms to gather information about program. The results of data-flow analysis all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analysis differ in the properties they compute.

1.1.4 Examples of Optimizations

Partial Dead-code Elimination

Avoiding the execution of unnecessary statements at runtime is called partial dead-code elimination(PDE). PDE improves the runtime efficiency of a program. PDE is combined of two separate transformations; 1)sinking of assignments; moving assignments as far as possible in the direction of the control flow, 2) eliminating all assignments being dead after the moving step.

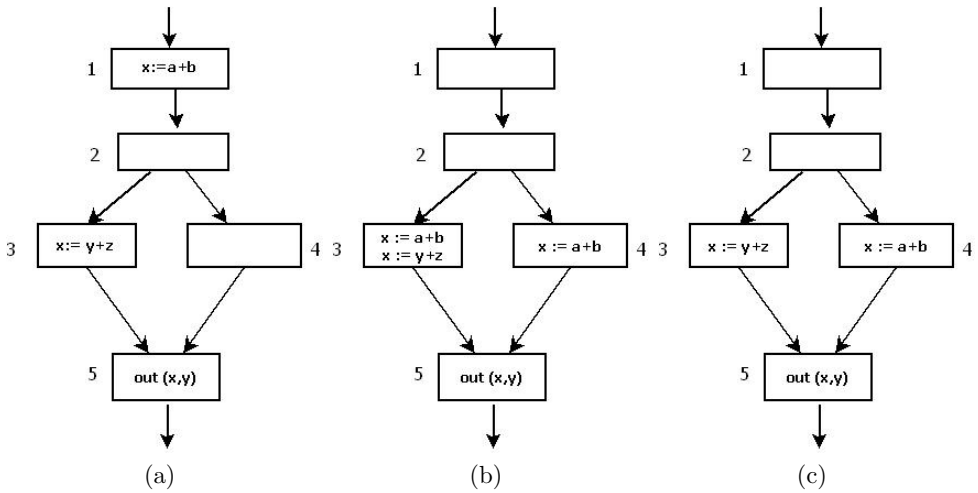


Figure 1.1: Example partial dead-code elimination

The following example, shown in figure 1.1 , illustrates that. The assignment

of node **1** of figure 1.1a is partially dead because it is dead on the left branch (as its left-hand side variable is not used afterwards in the program), but alive on the right one (as it used in node **5**). This assignment can be eliminated by moving it to the entries of node **3** and node **4** as shown in figure 1.1b. There, the occurrence of node **3** is (totaly) dead, and can be eliminated as shown in figure 1.1c.

Strength Reduction

Strength reduction[9, 69, 50, 52] is a compiler optimization which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

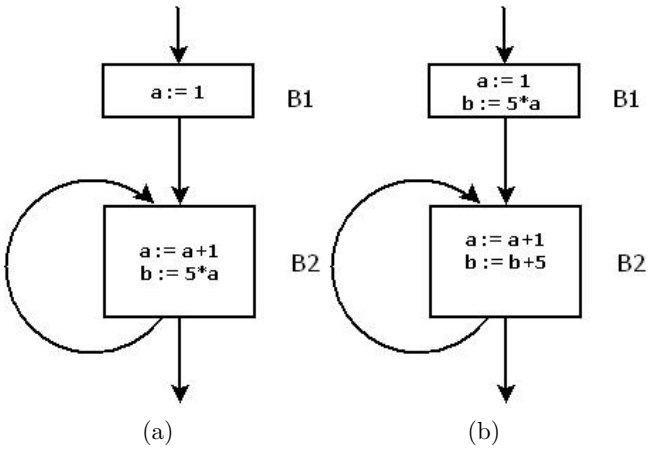


Figure 1.2: Example of strength reduction

Example Figure 1.2 shows an example that illustrates strength reduction, where the multiplication in figure 1.2a replaced by addition in figure 1.2b. As the relationship $b := 5 * a$ surly holds after such an assignment to b in figure 1.2a, and b is not changed elsewhere in the inner loop around `B2`, it follows that just after the statement $a := a + 1$ the relationship $b = 5a + 5$ must hold. We may therefore replace the assignment $b := 5 * a$ by $b = b + 5$. The only problem is that b does not have a value when we enter the block `B2` for first time. Since we must maintain the

relationship $b := 5 * a$ on entry to block `B2`, we place an initialization of b at the end of block where a itself is initialized.

1.1.5 Redundancy Elimination

Another category of optimization is redundancy elimination. Redundancy elimination is about eliminating one of two computations which are equivalent.

Causes of Redundancy

There are many redundant operations in a typical program. Sometimes the redundancy is available at the source level. For instance, a programmer may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary.

Types of redundancy elimination

1. ***Value Numbering***.

   Associate symbolic values to computations and identifies expressions that have the same value.

2. ***Common Subexpression Elimination***.

   Finds computations that are always performed at least twice on a given execution path and eliminates the second and later occurrences of them. This optimization requires data-flow analysis to locate redundant computations and almost always improves the performance of programs it is applied to.

3. ***Constant Propagation***.

   Identifies variables that have constant values and use the constants in place of variables.

4. ***Loop-invariant Code Motion***.

   Finds computations that produce the same result every time a loop is iterated and moves them out of the loop. This optimization almost improves the performance. For non-confusing also called *code motion*.

5. ***Partial Redundancy Elimination***.

   Inserts computations in path to convert partial redundancy to full redundancy. It is a complex optimization, it consists of *code motion* and *common subexpression elimination*

Partial redundancy elimination (PRE) is the most powerful compiler optimization which it encompasses common subexpression elimination and code motion. PRE was presented by Morel and Renvoice[46]. They solved the problem of PRE by solving more general problem, that is, the elimination of computations performed twice on a given execution path. Such computations called partially redundant. PRE performs insertions and deletions of computations in a flow graph in such away that after the transformations, each path, contains fewer occurrences of such computations than before. Most compilers today performs PRE. It is regarded as one of the most important optimizations and is an active area of research [8, 11, 30, 47, 48, 53, 76, 68, 5].

The algorithm presented by Morel and Renvoice[46] has some shortcomings. It does not eliminate all partial redundancies that exist in the program. It involves performing bidirectional data flow analysis which is more complex than unidirectional analysis [33]. To avoid these shortcoming Knoop et al. [33, 32] presented sequence of unidirectional analysis equivalent to that analysis and proposed optimal solution to the problem with no redundant code motion.

There are many methods used to represent PRE. J. Xue and J. Knoop represented PRE as a maximum flow problem [76]. It was amazing and innovative method to represent PRE. Ando Saabas and Tarmo Uustalu[61] used type systems as framework to solve PRE problem. They showed that type systems are a compact and useful framework to describe dataflow analysis and optimizations.

1.1.6 Type Systems

Type systems are formal methods ensure that a system (*program*) behaves correctly with respect to some specifications. Type systems are defined in many ways. The closest one that covers its informal usage by programming languages designers and implementers given by Benjamin [55] is:-

> *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute".*

In any type system, types provide a division or classification of some universe of possible values: a type of values that share some property.

Type checking In many languages, type checking is used to prevent some or all type errors. Some languages use type constraints in the definition of legal program. Implementations of these languages check types at compile time, before a program is started. In these languages, a program that violate a type constraints is not compiled and can not be run. In other languages, checks for type errors are made while the program is running.

Run-Time Type Checking In programming languages with run-time type checking, the compiler generates code so that, when an operation is performed, the code

checks to make sure that the operands have the correct type. An advantage of run-time type checking is that it catches type errors. A disadvantage is the run-time cost associated with making these checks.

Compile-Time Type Checking Many modern programming languages are designed so that it is possible to check expressions for potential type errors. In these languages, it is common to reject programs that do not pass the compile-time type checks. An advantage of compile-time type checking is that it catches errors earlier than run-time checking does. Compile-time checking can make it possible to produce more efficient code, because that compile-time checks may eliminate the need to check for certain errors at run-time.

Some languages may be called statically-typed. In these languages, each program phrase must have a type and the type can be determined efficiently from the syntactic form of the expression. A pragmatic reason to include types in programming languages is the role of types in many approaches to modular software design[10, 41].

Features of type systems Type system plays an important role in designing and using programming languages. Many features can be achieved by using type systems:

1. ***Detecting Errors***

   A good benefit of using static type checking is early detection of errors. Programming languages that are richly typed are a good environment for working. Not only trivial mistakes are detectable (such as multiplying an integer by a string), but also deeper conceptual errors (such as confusing units in scientific calculations).

2. ***Abstraction***

Type systems are considered to be the backbone of the modulo languages, which ties components together to produce large systems. Type systems allow the large system to be designed independently of particular implementations of its parts.

3. ***Documentation***

   Type information in procedure header and in module interfaces are a form of documentation, giving useful hints about behavior. This form of documentation is up to date, since it is checked during every run of the program.

4. ***Efficiency***

   Typing information be used by compilers to produce more efficient code. As an example, a first type system appeared in 1950s were to improve the efficiency of numerical calculations in Fortran.

5. ***Language Safety***

   There are many informal definitions of language safety. One of them: "*A safe language is one that protects its own high-level abstraction*". In these definition safety refers to the language's ability to guarantee the integrity of these abstractions. Another definition focuses on portability: "*A safe language is completely defined by its programmer's manual* ".

1.1.7 Multithreaded Programs

Multithreaded is a key consideration technique for modern software. Multithreaded programs are widely used by programmers. Programmers use multiple threads of control for many reasons. As an example, operating systems use multithreads as a basic technique. In operating systems, you can write your text file, playing an audio, updating software, and downloading file from internet at the same time. All of these

tasks called threads. All threads run at the same time. We do not know the order of execution. Another examples that use multithreading as core technique are building responsive servers that interact with multiple clients and producing sophisticated user interface[56]. Multithreading is extremely useful in practise. For example, a browser should be able to simultaneously download multiple images. A web server needs to be able to serve concurrent requests. Programming languages(like Java) use a thread to do garbage collection in the background. Graphical user interface(GUI) programs have a separate thread for gathering user interface events from the host operating environment.

Research in program analysis has traditionally focused on sequential programs [51]. Developing analysis for multithreaded programs is a challenging problem [44, 63]. The primary complication is characterizing the effect interactions between threads. Also, the order of execution of threads, which is unknown, add another complication. Pointer analysis known as a great research area in sequential programs [26]. Pointer analysis also takes a great place in analyzing multithreaded programs [23, 26, 15, 58, 62, 57].

1.2 Aims of thesis

1. The first aim is to apply PRE on a multi-threaded language. We use type systems as a framework. To achieve this aim, we firstly introduce *modified analysis*, which calculates the modified variables at each program point. Then, we define *concurrent modified function* $\mathcal{C}$. Definition of $\mathcal{C}$ depends on existing of threads. We define type system for *anticipability* and *conditional partial availability*, which differ from those in [61]. Type systems for optimization components were introduced. Then, we proved the soundness of type systems

with optimization components.

2. The second aim is to design a type systems based data race detector. We introduce a type system that detects data-race problem for multi-threaded programs of a simple language *m-while*. We also prove the soundness of proposed type system. To achieve this aim, we firstly introduce *read type system*, and *safe type system*. Soundness of proposed type systems are proved.

2. Related Work

Chapter 2

Related Work

This chapter reviews the work related to that presented in this thesis.

Organization

This chapter is organized as follow:-

1. Partial Redundancy Elimination.
2. Multi-threaded Programs.

2.1 Partial Redundancy Elimination

Partial redundancy elimination is the most powerful compiler optimization. It encompasses common subexpression elimination and code motion. In this section we will discuss the code motion and common subexpression elimination to build up our intuition about the problem. Then, we will discuss partial redundancy elimination.

2.1.1 Code Motion

Loops are a very important place for optimization, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may

be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

An important modification that decreases the amount of code in a loop is *code motion*. This transformation takes an expression that yield the same result independent of the number of times a loop is executed (a *loop-invariant computation*) and evaluates the expression before the loop.

Example 2.1: Evaluation of $limit - 2$ is a loop-invariant computation in the following while-statement in C-like language:

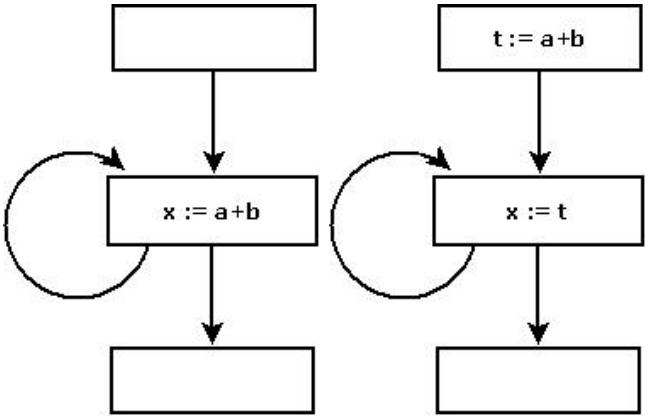
```
while (i≤ limit-2) \∗ statement does not change limit ∗\
```

Code motion will result in the equivalent code

```
t = limit-2
while (i≤ t) \∗ statement does not change limit ∗\
```

Now, the computation of $limit - 2$ is performed once, before we enter the loop. Previously, there would be $n + 1$ calculations of $limit - 2$ if we iterated the body of the loop n times.

Example 2.2: Fig. 2.1 shows an example of a loop-invariant expression (code motion). The expression $a + b$ is loop invariant assuming neither the variable a nor b is redefined within the loop. We can optimize the program by replacing all the re-executions in a loop by a single calculation outside the loop. We assign the computation to a temporary variable, say t, and then replace the expression in the loop by t. There is one more point we need to consider when performing "code motion" optimization such as this. We should not execute any instruction that would not have executed without the optimization.



(a) before optimization (b) after optimization

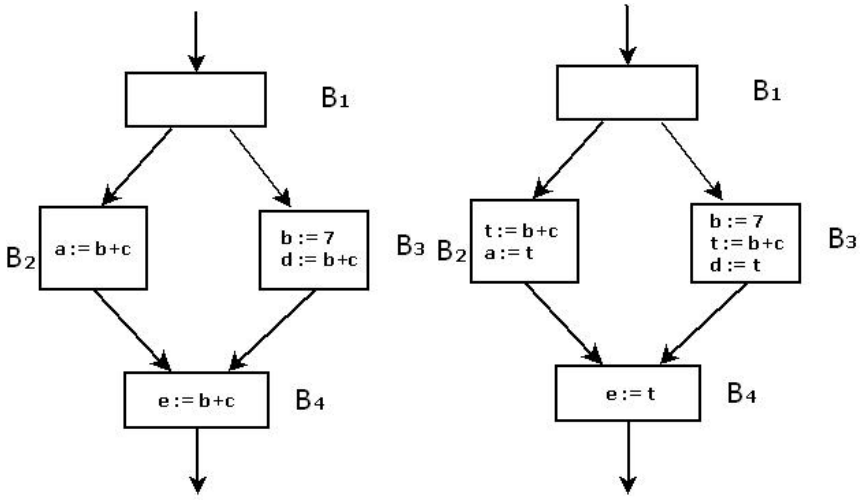
Figure 2.1: Example of code motion

2.1.2 Common Subexpression Elimination

An occurrence of an expression `E` is called *common subexpression* if `E` was previously computed and the values of the variables in `E` have not changed since the previous computation. We avoid recomputing `E` if we can use its previously computed value.

Example 2.3: In Fig. 2.2 the expression $b+c$ computed in block B_4 is redundant; it has already been evaluated by the time the flow of control reaches B_4 regardless of the path taken to get there. The value of the expression may be different on different paths. We can optimize the code by storing the result of the computations of $b+c$ in block B_2 and B_3 in the same temporary variable, say t, and then assigning the value of t to the variable e in block B_4, instead of reevaluating the expression. Had there been an assignment to either b or c after the last computation of $b+c$ but before block B_4 the expression in block B_4 would not be redundant.

Finally we say that an expression $b+c$ is fully redundant at point p, if it is available expression at that point. That is, the expression $b+c$ has been computed along all paths reaching p, and the variables b and c were not redefined after the



(a) before optimization (b) after optimization

Figure 2.2: Example of common subexpression elimination

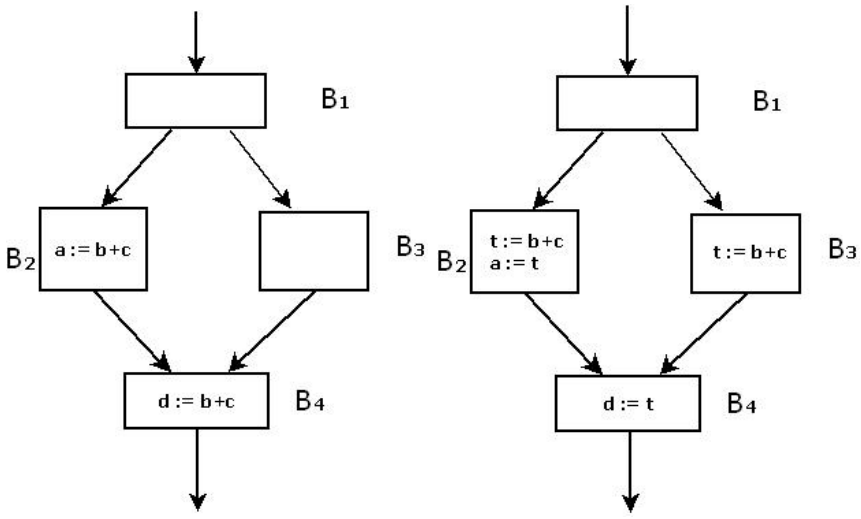
last expression was evaluated.

2.1.3 PRE

As an example of partially redundant expression is shown in Fig. 2.3. The expression $b + c$ in the block B_4 is redundant on the path $B_1 \rightarrow B_2 \rightarrow B_4$, but not on the path $B_1 \rightarrow B_2 \rightarrow B_4$. We can eliminate the redundancy on the former path by placing a computation of $b + c$ in block B_3. All the results of $b + c$ are written in a temporary variable t, and the calculation in block B_4 is replaced with t. Thus, like loop-invariant code motion, partial redundancy elimination requires the placement of new expression computation.

2.1.4 Morel and Renvoise's Algorithm for PRE

Now, we will present the first algorithm of PRE. It was proposed by Morel and Renvoise [46]. They presented an algorithm which performs PRE, by solving boolean systems of equations. This algorithm depends on some boolean properties associated



(a) before optimization (b) after optimization

Figure 2.3: Example of partial redundancy elimination

with expression. Some of these properties depends only on the commands of given block and are termed local. Other properties depend on interactions of different blocks and are termed global.

Local properties

1. *Transparency: TRANSP.* An expression is said to be transparent in a block i if its operands are not modified by the execution of the commands of block i.

2. *Local availability: COMP* An expression is said to be locally available in block i if there is at least one computation of the expression in the block i, and if the commands appearing in the block after the last computation of the expression do not modify its operands.

3. *Local anticipability: ANTLOC* An expression may be locally anticipated in a block i if there is at least one computation of the expression in the block i, and if the commands appearing in the block before the first computation of expression do not modify its operands.

Extending previous properties to a complete program makes these properties to be "*global*". Partial availability of an expression at a given point means that there is at least one path P leading from the entry point of the program to the point considered, and that computation of the expression inserted at this point would give the same result as the last computation of the expression made on the path P.

Systems of Boolean equations

- **Availability System**:-

$$\begin{pmatrix} AVIN_b = \prod_{p \in Pred(b)} AVOUT_p \\ AVOUT_b = COMP_b + AVIN * TRANSP_b \end{pmatrix}$$

- **Anticipability system**:-

$$\begin{pmatrix} ANTOUT_b = \prod_{s \in Succ(b)} ANTIN_s \\ ANTIN_b = ANTLOC_b + ANTOUT_b * TRANSP_b \end{pmatrix}$$

- **Partial Availability System**:-

$$\begin{pmatrix} PAVIN_b = \sum_{p \in Pred(b)} PAVOUT_p \\ PAVOUT_b = COMP_b + PAVIN * TRANSP_b \end{pmatrix}$$

where, $*$, $\prod$ assigned to Boolean conjunction, and $+$, $\sum$ are used for Boolean disjunction. The post IN and OUT were used to mean that the property at beginning of the block and after exiting the block respectively. $Pred(b)$ is the set of blocks before b, $Succ(b)$ the set of blocks after b.

Algorithm

The algorithm has the four steps:-

1. Resolution of the Boolean systems for availability, anticipability, and partial availability.

2. Determination of predecessors of the blocks containing the partial redundancies and where a ne computation may be introduced. This involves the computation of the Boolean properties PPIN and PPOUT (Placement possible on entry and placement possible on exit).

3. Determination of a subset of these blocks on exit of which a computation must be inserted. These blocks satisfy the Boolean properties INSERT.

4. Insertion of a new computation at the exit of the blocks satisfying the condition INSERT = TRUE and suppression of the partially redundant computations which are no redundant.

Now, we will define PPIN and PPOUT for every block b as follow:-

$$\left\{\begin{array}{l} PPIN_b = CONST_b * (ANTLOC_b + TRANSP_b * PPOUT_b) * \prod_{p \in Pred(b)} (PPOUT_p + AVOUT_p) \\ CONST_b = ANTIN_b * (PAVIN + TRANSP * \neg ANTLOC_b) \\ PPOUT_b = \prod_{s \in Succ} PPIN \end{array}\right\}$$

for each program block b $INSERT_b$ is computed by :-

$$INSERT_i = PPOUT_b * \neg AVOUT_b * (\neg PPIN_b + \neg TRANSP_b)$$

At the end of algorithm, new computations are inserted on exit of nodes satisfying $INSERT = TRUE$. Then the computation satisfying the following condition $ANTLOC * PPIN = TRUE$ are redundant and may be deleted.

2.1.5 PRE using type system

Idea of using type systems as framework for optimization were introduced at [61, 60]. A. Saabas and T. Uustau used type systems as framework to implement PRE. They used WHILE programs instead of using control-flow graph(CFG). Expressions contains at most one operator. This is an inessential restriction(deep expressions handled with little modifications on some infrastructures). The basic building blocks of WHILE are literals $l \in \mathbf{Lit}$, statements $s \in \mathbf{Stm}$, arithmetic expression $a \in \mathbf{AExp}$ and boolean expression $b \in \mathbf{BExp}$ are defined over a set of program variables $x \in \mathbf{Var}$ and numerals $n \in \mathbb{Z}$. The syntax of WHILE language is defined in figure 2.4. Note $\mathbf{AExp}^+ = \mathbf{AExp} \backslash \mathbf{Lit}$).

$$
\begin{aligned}
l &::= x \mid n \\
a &::= l \mid l_0 + l_1 \mid l_0 * l_1 \mid ... \\
b &::= l_0 = l_1 \mid l_0 \leq l_1 \mid ... \\
s &::= x := a \mid \mathbf{skip} \mid s_0; s_1 \mid \mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f \mid \mathbf{while}\ b\ \mathbf{do}\ s_t \mid
\end{aligned}
$$

Figure 2.4: Syntax of WHILE language

Natural Semantics The states $\sigma \in \mathbf{State}$ of the natural semantics are stores, $\mathbf{State} =_{\text{df}} \mathbf{Var} \longrightarrow \mathbb{Z}$. We write $[\![a]\!]\sigma$ for integer value of an arithmetic expression a and $[\![b]\!]\sigma$ for truth value of boolean expression b in a state σ. We write $\sigma \succ s \rightarrow \sigma'$ to mean that, if the state is σ before executing s then after execution the state will be σ'. The notation $\sigma[x \mapsto z]$ means that the state is σ updated at x with z. Figure 2.5 contains rules of natural semantics of WHILE language.

A. Saabas and T. Uustau presented two versions of PRE: *simple* PRE and *full* PRE. Simple PRE relies on backward anticipability analysis and forward conditional partial availability analysis. *Anticipability* analysis computes for each program point

$$\frac{}{\sigma \succ x := a \to \sigma[x \mapsto [\![a]\!]\sigma]}{:=_{ns}} \qquad \frac{}{\sigma \succ \mathbf{skip} \to \sigma}\mathbf{skip}_{ns} \qquad \frac{\sigma \succ s_0 \to \sigma'' \quad \sigma'' \succ s_1 \to \sigma'}{\sigma \succ s_0; s_1 \to \sigma'}\mathbf{comp}_{ns}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \to \sigma'}{\sigma \succ \mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f \to \sigma'}\mathbf{if}^{tt}_{ns} \qquad \frac{\sigma \not\models b \quad \sigma \succ s_f \to \sigma'}{\sigma \succ \mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f \to \sigma'}\mathbf{if}^{ff}_{ns}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \to \sigma'' \quad \sigma'' \succ \mathbf{while}\ b\ \mathbf{do}\ s_t\ \to \sigma'}{\sigma \succ \mathbf{while}\ b\ \mathbf{do}\ s_t\ \to \sigma'}\mathbf{while}^{tt}_{ns} \qquad \frac{\sigma \not\models b}{\sigma \succ \mathbf{while}\ b\ \mathbf{do}\ s_t\ \to \sigma}\mathbf{while}^{ff}_{ns}$$

Figure 2.5: Natural Semantics of WHILE language

which nontrivial arithmetic expressions will be evaluated on all paths before any of their operands are modified. *Partial availability* analysis computes, for each program point, which expressions have already been evaluated and later not modified on some path through this program point. *Conditional partial availability* analysis: an expression becomes partially available only hen it is anticipable. The inequations for the analyses are:

$$ANTOUT_i \subseteq \left\{ \begin{array}{ll} \phi & \texttt{if} \quad i = f \\ \bigcap\limits_{j \in succ(i)} ANTIN_j & \texttt{otherwise} \end{array} \right\}$$

$$ANTIN_i \subseteq (ANTOUT_i \backslash MOD_i) \cup EVAL_i$$

$$CPAVIN_i \supseteq \left\{ \begin{array}{ll} \phi & \texttt{if} \quad i = s \\ \bigcup\limits_{j \in pred(i)} CPAVOUT_j & \texttt{otherwise} \end{array} \right\}$$

$$CPAVOUT_i \supseteq ((CPAVIN_i \cup EVAL_i) \backslash MOD_i) \cap ANTOUT_i$$

$$CPAVIN_i \subseteq ANTIN_i$$

$$CPAVOUT_i \subseteq ANTOUT_i$$

These inequations relies on ANT, CPAV, EVAL,and MOD. ANT for anticipability of nontrivial arithmetic expressions. CPAV for conditional partial availability of

nontrivial arithmetic expressions. MOD denotes the set of expressions hose values might be modified. EVAL denotes the set of expressions which are evaluated. The post IN(OUT) and index b means the before entering(exit) the node b(s and f corresponding to start and finish of nodes of whole of CFG).

$$\frac{}{x := a : ant' \backslash mod(x) \cup eval(a) \to ant'} \qquad \frac{}{\mathbf{skip} : ant \to ant}$$

$$\frac{s_0 : ant \to ant'' s_1 : ant'' \to ant'}{s_0; s_1 : ant \to ant'} \qquad \frac{s_t : ant \to ant' \quad s_f : ant \to ant'}{\mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\quad s_f : ant \to ant'}$$

$$\frac{s_t : ant \to ant}{\mathbf{while}\ b\ \mathbf{do}\ s_t : ant \to ant} \qquad \frac{ant \leq ant_0 \quad s : ant_0 \to ant_0' \quad ant_0' \leq ant'}{s : ant \to ant'}$$

Figure 2.6: Type system for anticipability analysis

$$\frac{}{x := a : ant' \backslash mod(x) \cup eval(a), cpav \to ant', (cpav \cup aval(a) \backslash mod(x)) \cap ant'}$$

$$\frac{}{\mathbf{skip} : ant, cpav \to ant, cpav}$$

$$\frac{s_0 : ant, cpav \to ant'', cpav'' s_1 : ant'', cpav'' \to ant', cpav'}{s_0; s_1 : ant, cpav \to ant', cpav'}$$

$$\frac{s_t : ant, cpav \to ant', cpav' \quad s_f : ant, cpav \to ant', cpav'}{\mathbf{if}\ b\ \mathbf{then}\ \underline{s_t}\ \mathbf{else}\ s_f : ant, cpav \to ant', cpav'} \qquad \frac{s_t : ant, cpav \to ant, cpav}{\mathbf{while}\ b\ \mathbf{do}\ s_t : ant, cpav \to ant, cpav}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \to ant_0', cpav_0' \quad ant_0', cpav_0' \leq ant', cpav'}{s : ant, cpav \to ant', cpav'}$$

Figure 2.7: Type system for simple PRE

Type system for simple PRE

Type systems for anticipability and simple PRE are given in figures 2.6 and 2.7. We write $eval(a)$ to denotes set $\{a\}$ if if a is nontrivial expression and ϕ otherwise, also $mod(x) = \{a \in \mathbf{AExp}^+ | x \in FV(a)\}$. For combined type system, a type is a pair $(ant, cpav) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+)$ and subtyping $\leq$ is pointwise set inclusion i.e. $(ant, cpav) \leq (ant', cpav')$ iff $ant \subseteq ant'$ and $cpav \subseteq cpav'$. The type system of optimization component shown in figure 2.8.

$$\frac{a \notin cpav \quad a \notin ant'\backslash mod(x)}{\underline{x := a} : ant'\backslash mod(x) \cup eval(a), cpav \to ant', (cpav \cup aval(a)\backslash mod(x)) \cap ant' \hookrightarrow x := a}{:=}\mathbf{1pre}$$

$$\frac{a \notin cpav \quad a \in ant'\backslash mod(x)}{\underline{x := a} : ant'\backslash mod(x) \cup eval(a)\backslash mce(x := a), cpav \to ant', (cpav \cup aval(a)\backslash mod(x)) \cap ant' \hookrightarrow nv(a) = a; x := nv(a)}{:=}\mathbf{2pre}$$

$$\frac{a \in cpav}{\underline{x := a} : ant'\backslash mod(x) \cup eval(a), cpav \to ant', (cpav \cup aval(a)\backslash mod(x)) \cap ant' \hookrightarrow x := nv(a)}{:=}\mathbf{3pre}$$

$$\frac{}{\underline{\mathbf{skip}} : ant, cpav \to ant, cpav \hookrightarrow \mathbf{skip}}\mathbf{skip_{pre}}$$

$$\frac{\underline{s_0} : ant, cpav \to ant'', cpav'' \hookrightarrow s'_0 \quad \underline{s_1} : ant'', cpav'' \to ant', cpav' \hookrightarrow s'_1}{\underline{s_0; s_1} : ant, cpav \to ant', cpav' \hookrightarrow s'_0, s'_1}\mathbf{comp_{pre}}$$

$$\frac{\underline{s_t} : ant, cpav \to ant', cpav' \hookrightarrow s'_t \quad \underline{s_f} : ant, cpav \to ant', cpav' \hookrightarrow s'_f}{\mathbf{if}\ b\ \mathbf{then}\ \underline{s_t}\ \mathbf{else}\ \underline{s_f} : ant, cpav \to ant', cpav' \hookrightarrow \mathbf{if}\ b\ \mathbf{then}\ s'_t\ \mathbf{else}\ s'_f}\mathbf{if_{pre}}$$

$$\frac{\underline{s_t} : ant, cpav \to ant, cpav \hookrightarrow s'_t}{\mathbf{while}\ b\ \mathbf{do}\ \underline{s_t} : ant, cpav \to ant, cpav \hookrightarrow \mathbf{while}\ b\ \mathbf{do}\ s'_t}\mathbf{while_{pre}}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad \underline{s} : ant_0, cpav_0 \to ant'_0, cpav'_0 \hookrightarrow s' \quad ant'_0, cpav'_0 \leq ant', cpav'}{\underline{s} : ant, cpav \to ant', cpav' \hookrightarrow [nv(a) := a | a \in cpav_0 \backslash cpav]; s'; [nv(a) := a | a \in cpav' \backslash cpav'_0]}\mathbf{conseq_{pre}}$$

Figure 2.8: Type system for optimization component

Relational method used to prove the soundness of optimization components. Defining similarity relation $\sim$ on the states. Let $\sigma \sim_{cpav} \sigma'$ denote that two states σ and σ' are agree wrt. $cpav \subseteq AExp^+$ in the sense that $\forall x \in Var.\sigma(x) = \sigma'(x)$ and $\forall a \in cpav.[\![a]\!]\sigma = \sigma_*(nv(a))$. The following theorem of soundness can be stated

Theorem 1. *Soundness of Optimization Component*
If $\underline{s} : ant, cpav \to ant', cpav' \hookrightarrow s_*$ *and* $\sigma \sim_{cpav} \sigma_*$ *then*
- $\sigma \succ s \to \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{cpav'} \sigma'_*$ *and* $\sigma_* \succ s_* \to \sigma'_*$*, -* $\sigma_* \succ s_* \to \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{cpav'} \sigma'_*$ *and* $\sigma \succ s \to \sigma'$*.*

Extending a state and similarity relation were needed to prove the improvement property of simple PRE. As a main part of these work, they prove that type system of of optimization component preserves Hoare logic proofs. The following theorem assures that meaning.

Theorem 2. *Preservation of Hoare logic profs*
If $\underline{s} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s_*$ *then,*
- $\{P\}s\{Q\}$ *implies* $\{P|_{cpav}\}s_*\{Q|_{cpav}\}$.

Where, $\{P\}s\{Q\}$ means that the judgement that Q is a derivable postcondition for s and a precondition P, also $P|_{cpav}$ abbreviate $\bigwedge [nv(a) = a | a \in cpav] \wedge P$

Full PRE Simple PRE does not use all optimization opportunities, where it only takes into account anticipability. There is a condition to apply PRE fully and correctly that is, the path leading from the point where the expression becomes partially available to a point where the expression becomes partially anticipable contains no points at which the expression is neither anticipable nor available. This condition can be detected by to additional data flow analyses: safe partial availability and safe partial anticipability analyses. Their description rely on the notion of safety. A program point is said to be safe wrt. an expression if that expression is available or anticipable at that program point. All previous theorems introduced again but with concept of full PRE.

2.2 Analysis of Multi-threaded Programs.

The field of program analysis has focused primarily on sequential programs. But multithreading is becoming increasingly important, both as a program structuring mechanism and to support efficient parallel computations. There are several uses for analysis information extracted from multi-threaded programs. The first use is to enable optimizations, both generalization of traditional compiler optimizations to multi-threaded programs and optimizations that make sense only for multi-threaded programs. The second use is to detect anomalies in the parallel execution such as data races or deadlock.

2.2.1 Optimization Uses

A problem with directly applying traditional compiler optimizations to multi-threaded programs is that the optimizations may reorder accesses to shared data in ways that may be observed by threads running concurrently with the transformed thread[44]. One approach is to generalize standard program representations, analyses, and transformations to safely optimize multithreaded programs even in the presence of access to shared data[65, 34, 35]

A more conservative approach is to ensure that the optimizations preserve the semantics of the original program by first identifying regions of the program that do not interact with other threads, then applying optimizations only within these regions. The analysis problem is determining which statements may interact with other threads and which may not. Escape analysis is an obvious analysis to use for this purpose - it recognizes data that is captured within the current thread and therefore inaccessible to other threads [62]

2.2.2 Data Race Detection

In an unsafe language like C, there are a number of program actions that are almost always the result of programmer error, regardless of the context in which they occur. Examples include array bounds violations and accessing memory after it has been deallocated. If the program engages in these actions, it can produce behavior that is very difficult to understand. Several well-known language design and implementation techniques (garbage collection, array bounds checks) can completely eliminate these kinds of errors. The cost is additional execution overhead and a loss of programmer control over aspects of the program's execution. The result was that, for many years, the dominant programming language (C) provided no protection at all against this class of errors.

2.2.3 Pointer Analysis

Pointer analysis for sequential programs is relatively mature field [17]. This field has been extended to cover multithreaded programs [58, 15, 23]. The importance of pointer analysis comes from two reasons: the first is the importance of pointer information for many compiler optimizations and corrections [64], and the second is the growing interest in multithreading as a mainstream practise of programming.

memory safety analysis aims at statically proves that the program does not treat pointers illegally according to the language syntax. El-Zawawy [15] introduced type-systems as a framework to deal this problem. He present type systems for flow-sensitive, flow-insensitive, and memory safety analysis. Another algorithm for pointer analysis introduced by R. Rugina and M. C. Martin [58]. This algorithm is designed to handle programs with structured parallel programs. For each pointer and program point, the algorithm computes a conservative approximation of the memory locations to which that pointer may point.

2.3 Summary

We summarize the related work in the following section.
Multi-threading is a promising area of research. The most efficient challenging areas are compilation and program analysis[34, 44, 18]. The field of program analysis aims at collecting information about programs[51]. Analyzing may concentrate on whole program, or focuses on each program point. There are many aspects of analyzing multi-threaded programs: pointer analysis[15, 58], optimization uses [21, 39, 38, 35, 34], data race detection[59, 6] and deadlock detection [36, 3, 74].

A data race occurs very often in multi-threaded programs. It occurs when two threads try to access the same location without proper synchronization, and one of

them is write [25]. The data race always causes program bugs errors. The output of program depends on scheduling of accessing memory. Detecting data race is a promising area of research, it has been studied extensively [22, 6, 54, 29, 49, 73]. The first methodology to detect data races is the static race detection[22]. Detectors, in this strategy, determine whether a program will ever produce a data race when run on all possible inputs. The second methodology is dynamic race detection, where potential races are detected at runtime by executing the program on a given input.[66, 74]. In many detectors, data-race and deadlock bugs are bundled together. Some static detectors, like Warlock[70], depends on the annotations formed by the programmers to detect data-race and deadlock problems. Using theorem provers to detect many bugs including data race is the idea of extended static checker for Java[22, 40]. Some dynamic detectors are developed in the scientific parallel programming community[6, 13]. Others detectors detects data race in Java-like programs [7, 72]. Eraser[66], as a dynamic detector, monitors programs during execution and look for data-race bugs. In general, dynamic detectors have the advantage that they can check un-annotated programs.

Type systems are known to be a good framework for analyzing programs [37, 19]. Type systems have been extensively used in pointer analysis for both imperative and multi-threaded programs [17, 15, 16]. Type systems have been used to detect the memory safety of multi-threaded programs by El-Zawawy [15]. Type systems used in code optimization. Partial redundancy elimination was performed via type systems for imperative programs in [61], and for multi-threaded programs in[21]. Type systems also are used to prevent data-race and deadlocks in a specific language Java[54].

PRE was originated by Morel and Renvoice [46]. They applied PRE using static analysis and presented PRE as a general problem of global optimization using

boolean system of equations. [46] also presents an algorithm for global optimization, which does not need *control flow graph*. Efforts have been done to improve the formulation of PRE [12]. The work in [76, 75] formulates classic and commulative PRE as a maximum flow problem. PRE used as a framework and is extended to do more optimizations as strength reduction [31].

3. PRE for Multi-threaded Programs

Chapter 3

PRE for Multi-threaded Programs

3.1 Introduction

There are many methods for compiler optimization; a powerful one of them is partial redundancy elimination (PRE). PRE eliminates redundant computations on some but not necessarily all paths of programs. PRE is a complex optimization as it consists of loop invariant code motion and common subexpression elimination. PRE was established by Morel and Renvoise [46] where they introduce a more general problem (as a system of boolean equations). Xue and Cai formulated speculative PRE as a maximum flow problem [75]. Xue and Knoop proved that the classic PRE is a maximum flow problem [76]. Saabas and Uustalu use type-systems framework to approach this problem [61]. Some Optimizations have been added to PRE such as strength reduction[31] and global value numbering[4]. All methods mentioned above are established to operate on sequential programs.

In the present work, we achieve partial redundancy elimination for multi-threaded programs which are widely used. Operating system is an example of system software that depends on multi-threading. You can write your document in a word processor while running an audio file, downloading a file from the internet, and/or scanning for

viruses (each of these tasks is considered a thread of computations). Web browser as an example can explore your e-mail, while downloading a file in the background.

The key feature of multi-threaded programs is that many threads can be executed at the same time. Consequently, when executing a thread there is an effect that comes from executing other threads. In general, when analyzing multi-threaded programs, the effect of all threads at the same time must be taken in account. Hence, analyzing multi-threaded programs completely differs from sequential ones.

Deducing and stating properties of programs can be done using type systems as well as program analysis. Program analysis has algorithmic manner while type systems are more declarative and easy to understand with type derivations that provide human-friendly format of justifications. We present a type system for optimizing multi-threaded programs. Our type system depends on a new analysis, namely *modified analysis* and a function called *concurrent modified*, rather than on *anticipability* analysis and *conditional partial availability* analysis used for the while language.

Organization of this chapter is as follow. In section 2 we introduce an operational semantics for the language we study. Section 3 presents the concepts of modified analysis and concurrent modified function. Also the soundness of modified analysis, the anticipability analysis, and conditional partial availability analysis for multi-threaded language are discussed in this section. In section 4, we present the type system including the optimization component and prove its soundness. Section 5 and 6 outline related and future work, respectively.

3.2 Motivation

In this section we introduce the language we study (FWHILE), a motivating example, and a natural semantics of FWHILE

3.2.1 Fwhile Language

We assume that our reader is familiar with data flow analysis. We introduce a motivating example to show the importance and obstacles of applying PRE on multi-threaded programs. We use a simple language which we call FWHILE. The basic building blocks of FWHILE are literals $l \in \mathbf{Lit}$, statements $s \in \mathbf{Stm}$, arithmetic expressions $a \in \mathbf{AExp}$ and boolean expressions $b \in \mathbf{BExp}$. These blocks are defined over a set of program variables $x \in \mathbf{Var}$ and numerals $n \in \mathbb{Z}$ in the following way:-

$$l ::= x \mid n$$

$$a ::= l \mid l_0 + l_1 \mid l_0 * l_1 \mid ...$$

$$b ::= l_0 = l_1 \mid l_0 \leq l_1 \mid ...$$

$$s ::= x := a \mid \mathbf{skip} \mid s_0; s_1 \mid \mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f \mid \mathbf{while}\ b\ \mathbf{do}\ s_t \mid$$
$$\mathbf{fork}\{s_1, s_2, ..., s_n\}.$$

We use the following notation $\mathbf{AExp}^+$ for the non trivial arithmetic expressions i.e ($\mathbf{AExp}^+ = \mathbf{AExp} \backslash \mathbf{Lit}$) .

3.2.2 Motivating Example

The following is an example that motivates our research.

$v := a - c;$

$u := a + b;$

$\mathbf{fork}\{\{y := a + b; c = 2; z := a - c;\};$

$\{x := a + b; z := a - c; \}; \}$

In this example, expressions $a + b$ and $a - c$ are evaluated before reaching the **fork** statement, hence we can use their values in the **fork** statement. But, one of the threads modifies the value of c hence we cannot use expressions containing c; because we do not know when the thread contains this modification will be executed. After applying our analysis the optimized version of the program will be:-

$v := a - c;$

$t_1 := a + b;$

$u := t_1;$

fork$\{\{y := t_1; c = 2; z := a - c; \};$

$\{x := t_1; z := a - c; \}; \}$

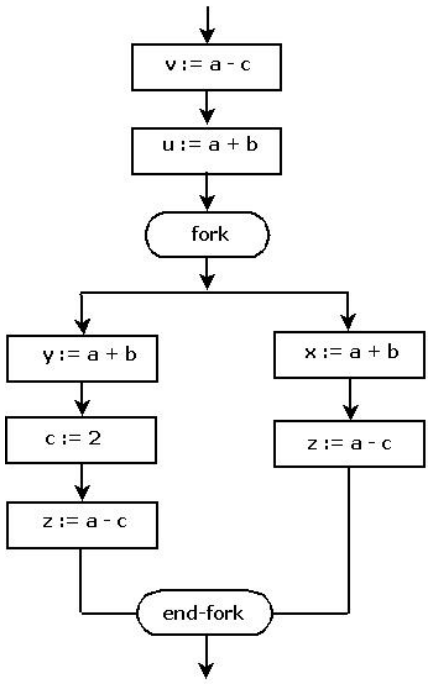


Figure 3.1: Before Optimization

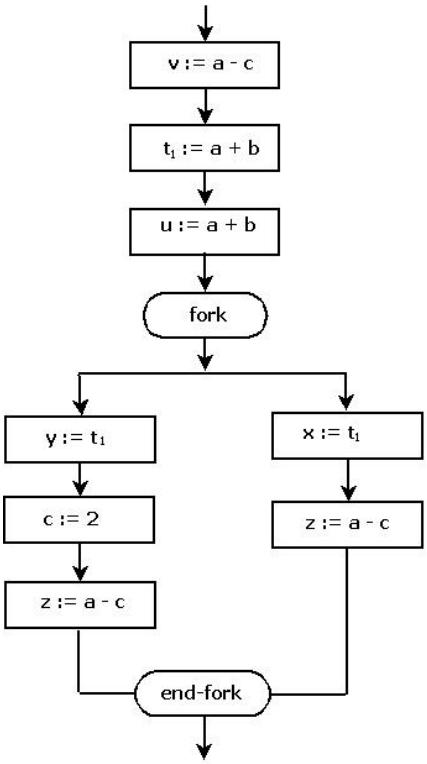


Figure 3.2: After Optimization

3.2.3 Natural Semantics

We use the semantics introduced by Mohamed El-Zawawy in [15]. We review the semantics in this section. We define a **State** as a function from a set of variables to integers: $\sigma \in \mathbf{State}$, $\sigma : \mathbf{Var} \longrightarrow \mathbb{Z}$. The state assigns a value for each variable. Expressions (arithmetic and boolean) are defined by semantic function $[\![-]\!] \in \mathbf{AExp} \cup \mathbf{BExp} \rightharpoonup \mathbb{Z} \cup \{tt, ff\}$ in denotational style. For $a \in \mathbf{AExp}$ and $b \in \mathbf{BExp}$ we write $[\![a]\!]\sigma$ and $[\![b]\!]\sigma$ to denote the evaluations of expressions a and b in a state σ, respectively. We write $\sigma \models b$ to denote that $[\![b]\!]\sigma = tt$(i.e evaluation of b in σ is true). Statements are written in the form of evaluation relation $\succ\!\!- \; \rightarrow \subseteq \mathbf{State} \times \mathbf{Stm} \times \mathbf{State}$. The notation $\sigma[x \mapsto [\![a]\!]\sigma]$ denotes that the state is σ rather than $\sigma(x) = [\![a]\!]\sigma$. Inference rules of the semantics are:

$$\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma]}:=_{ns} \qquad \frac{}{\sigma \succ \mathbf{skip} \rightarrow \sigma}\mathbf{skip}_{ns} \qquad \frac{\sigma \succ s_0 \rightarrow \sigma'' \quad \sigma'' \succ s_1 \rightarrow \sigma'}{\sigma \succ s_0; s_1 \rightarrow \sigma'}\mathbf{comp}_{ns}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'}{\sigma \succ \mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f \rightarrow \sigma'}\mathbf{if}^{tt}_{ns} \qquad \frac{\sigma \not\models b \quad \sigma \succ s_f \rightarrow \sigma'}{\sigma \succ \mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f \rightarrow \sigma'}\mathbf{if}^{ff}_{ns}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'' \quad \sigma'' \succ \mathbf{while}\ b\ \mathbf{do}\ s_t\ \rightarrow \sigma'}{\sigma \succ \mathbf{while}\ b\ \mathbf{do}\ s_t\ \rightarrow \sigma'}\mathbf{while}^{tt}_{ns} \qquad \frac{\sigma \not\models b}{\sigma \succ \mathbf{while}\ b\ \mathbf{do}\ s_t\ \rightarrow \sigma}\mathbf{while}^{ff}_{ns}$$

$$\frac{\sigma_i \succ s_{\theta(i)} \rightarrow \sigma_{i+1} \qquad \forall i \in \{1, 2, ...n\}}{\sigma_1 \succ \mathbf{fork}\{s_1, s_2, ..., s_n\}\ \rightarrow \sigma_{n+1}}\mathbf{FORK}_{ns}$$

where θ is a permutation on $\{1, 2, ..., n\}$

The rule $\mathbf{FORK_{ns}}$ depends on the order of executing the threads. We assume that order named θ which is a permutation on n (number of threads). In this rule we assume that the threads will execute one by one.

3.3 Program Analysis

In this section we will introduce the analysis of the multithreaded programs. We introduce type systems to help optimizing programs. Firstly, we introduce the modified analysis which tells which variables are modified. Secondly, we introduce a concurrent modified function. Also, we introduce traditional anticipability analysis and conditional partial availability analysis which are generalizations of the work of [61] (with additional rules for multi-threaded statements).

3.3.1 Modified Analysis

Modified analysis computes for each program point which variables have been modified. The type system is simple. It gathers the modified variables along the path to the point. Type $m \subseteq \mathbf{Var}$ is a set of variables. Modified analysis is a must forward analysis. The subtyping is the revered set inclusion (i.e $\leq =_{df} \supseteq$).

Definition 1. *For any program point, any state σ and a type modified $m \subseteq$* **Var**, *we write $\sigma \vDash m$ (σ entails m)* iff $m \subseteq dom(\sigma)$.

Type system is as follow :-

$$\frac{}{x := a : m \to m \cup \{x\}}\,\mathbf{m:=} \qquad \frac{}{\mathbf{skip} : m \to m}\,\mathbf{mskip} \qquad \frac{s_0 : m \to m'' \; s_1 : m'' \to m'}{s_0; s_1 : m \to m'}\,\mathbf{mcomp}$$

$$\frac{s_t : m \to m' \quad s_f : m \to m'}{\mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f : m \to m'}\,\mathbf{mif} \qquad \frac{s_t : m \to m}{\mathbf{while}\ b\ \mathbf{do}\ s_t : m \to m}\,\mathbf{mwh}$$

$$\frac{m \leq m_0 \quad s : m_0 \to m_0' \quad m_0' \leq m'}{s : m \to m'}\,\mathbf{mconseq}$$

$$\frac{s_{\theta(i)} : m \to m_i \quad \forall i \in \{1, 2, ...n\}}{\mathbf{fork}\{s_1, s_2, ..., s_n\} : m \to \bigcup\limits_{1 \leq i \leq n} m_i}\,\mathbf{mFORK}$$

The type system is clear and simple. The rule **m** := adds the assigned variable to pre type. Rules **mskip**, **mconseq**, **mif** and **mwh** are direct and similar to operational semantics. Rule **mconseq** for strengthen the pre-type. Rule **mFORK** is for threading. This rule computes the modified variables along **fork** statement by collecting all modified variables over all threads.

The following two lemmas state properties about modified analysis:-

Lemma 1. *Suppose $s : m \longrightarrow m'$, where m , $m' \subseteq$* **Var**. *Then $m \subseteq m'$.*

Proof It is clear that the statement which actually changes the set m is assignment statement where $m' = m \cup \{x\}$ *i.e* $m \subseteq m'$.

Lemma 2. *Suppose $s : m' \longrightarrow m''$ and $\sigma \succ\!- s \to \sigma'$, where m , m' , $m'' \subseteq$* **Var**. *Then $s : m' \cup m \longrightarrow m'' \cup m$.*

Proof We have $m \subseteq dom(\sigma) \Rightarrow m' \subseteq dom(\sigma')$. Then it is clear that:-
$m \cup m'' \subseteq dom(\sigma) \Rightarrow m' \cup m'' \subseteq dom(\sigma')$

3.3.2 Soundness of modified analysis

The following theorem proves the soundness of modified analysis

Theorem 3. *Suppose* $s : m \to m'$ *and* $\sigma \succ\!- s \to \sigma'$. *Then if* $\sigma \vDash m$ *then* $\sigma' \vDash m'$.

Proof The proof is by structure induction of type derivation.

- Type derivation is $\mathbf{m} :=$ and corresponding operational semantic is $:=_{ns}$.

  We have $\sigma \vDash m$ and $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$ which implies $dom(\sigma') = dom(\sigma) \cup \{x\}$.

  $$\because m \subseteq dom(\sigma) \Longrightarrow m \cup \{x\} \subseteq dom(\sigma) \cup \{x\} \Longrightarrow m' \subseteq dom(\sigma') \Longrightarrow \sigma' \vDash m'$$

- Type derivation is **mskip** the corresponding semantic is $\mathbf{skip_{ns}}$

  Let $\sigma \vDash m$ choosing $\sigma' = \sigma$ and $m' = m$, then $\sigma' \vDash m'$

- Type derivation is **mcomp** and the corresponding operational semantic is $\mathbf{comp_{ns}}$.

  From premises we get:-

  (i) $\sigma \succ\!- s_0 \to \sigma''$, $s_0 : m \to m''$ and $\sigma \vDash m \Longrightarrow \sigma'' \vDash m''$

  (ii) $\sigma'' \succ\!- s_1 \to \sigma'$, $s_1 : m'' \to m'$ and $\sigma'' \vDash m'' \Longrightarrow \sigma' \vDash m'$

  from (i) and (ii) then we have $\sigma \vDash m \Longrightarrow \sigma' \vDash m'$

- Type derivation is **mif** two cases arises:-

  (1) Case $\sigma \models b$ then the corresponding operational semantic is $\mathbf{if^{tt}_{ns}}$

  We have **if** b **then** s_t **else** $s_f : m \to m'$ and $\sigma \succ\!-$**if** b **then** s_t **else** $s_f \to \sigma'$.

  From premises the following holds:

  $s_t : m \to m'$ and $\sigma \succ\!- s_t \to \sigma'$ then $\sigma \vDash m \implies \sigma' \vDash m'$

  (2) Case $\sigma \not\models b$ then the corresponding operational semantic is $\mathbf{if^{ff}_{ns}}$

  We have **if** b **then** s_t **else** $s_f : m \to m'$ and $\sigma \succ\!-$**if** b **then** s_t **else** $s_f \to \sigma'$.

From premises the following holds:

$s_f : m \rightarrow m'$ and $\sigma \succ\!- s_f \rightarrow \sigma'$ then $\sigma \vDash m \implies \sigma' \vDash m'$

- Type derivation is **mwh**, we have **while** b **do** $s_t : m \rightarrow m$.

  (1) Case $\sigma \nvDash b$, the corresponding operational semantic is: $\mathbf{while}^{\mathbf{ff}}_{\mathbf{ns}}$.

  Same as **skip** statement proved above.

  (2) Case $\sigma \vDash b$ we have the following operational semantic $\mathbf{while}^{\mathbf{tt}}_{\mathbf{ns}}$ then **while** b **do** $s_t \equiv s_t$; **while** b **do** s_t applying composition rule stated above we find

$$\sigma \vDash m \implies \sigma' \vDash m'$$

- Type derivation is **mconseq** and $\sigma \succ\!- s \rightarrow \sigma'$

$$\begin{aligned} \sigma \vDash m &\implies \sigma \vDash m_0 \quad (m \leq m_0) \\ &\implies \sigma' \vDash m'_0 \quad (s : m_0 \rightarrow m'_0 \; and \; \sigma \succ\!- s \rightarrow \sigma') \\ &\implies \sigma' \vDash m' \quad (m'_0 \leq m') \\ i.e \quad \sigma \vDash m &\implies \sigma' \vDash m' \end{aligned}$$

- Type derivation is **mFORK** and the corresponding operational semantic is $\mathbf{FORK}_{\mathbf{ns}}$. We prove that :-

$$\sigma_1 \vDash m \Longrightarrow \sigma_{n+1} \vDash \bigcup_{1 \leq i \leq n} m_i$$

  From premises we have $s_{\theta(i)} : m \rightarrow m_i$, which by lemma 1 implies

  $m \subseteq m_i \quad \forall i \in \{1, 2 \ldots, n\}$.

  From lemma 2 and lemma 1 we can get :

  $s_{\theta(1)} : m \rightarrow m_1$

  $s_{\theta(2)} : m_1 \rightarrow m_1 \cup m_2$

$s_{\theta(3)} : m_1 \cup m_2 \rightarrow m_1 \cup m_2 \cup m_3$

$\vdots$

$s_{\theta(n)} : m_1 \cup m_2 \ldots \cup m_{n-1} \rightarrow m_1 \cup m_2 \ldots \cup m_n$

To simplify the notations we let

$\mathcal{M}_0 = m$

$\mathcal{M}_1 = m_1$

$\mathcal{M}_2 = m_1 \cup m_2$

$\vdots$

$\mathcal{M}_n = m_1 \cup m_2 \ldots \cup m_n = \bigcup_{1 \leq i \leq n} m_i$

The previous sequence can be written as:-

$$s_{\theta(i)} : \mathcal{M}_{i-1} \rightarrow \mathcal{M}_i, \quad also \quad we \quad have \quad \sigma_i \succ s_{\theta(i)} \rightarrow \sigma_{i+1}$$

Then we get:

$$\sigma_1 \models \mathcal{M}_0 \Longrightarrow \sigma_2 \models \mathcal{M}_1$$

$$\sigma_2 \models \mathcal{M}_1 \Longrightarrow \sigma_3 \models \mathcal{M}_2$$

$$\vdots$$

$$\sigma_n \models \mathcal{M}_{n-1} \Longrightarrow \sigma_{n+1} \models \mathcal{M}_n$$

From last sequence, we conclude

$$\sigma_1 \models m \Longrightarrow \sigma_{n+1} \models \mathcal{M}_n$$

3.3.3 Concurrent Modified Function $\mathcal{C}$

In this section we will present concurrent modified function $\mathcal{C}$. We start by defining *sub-statement* relation between statements.

Definition 2. *For any two statements s and t, we say that t is a sub-statement of s written as $(t \Subset s)$ iff :-*

1. *t=s or*

2. *$s = s_1; s_2$ and $t \Subset s_1$ or $t \Subset s_2$*

We mean by t=s, that t is identically(syntactically) equivalent to s.

Definition 3. *The concurrent modified function assigns a set of variables for each program point, i.e.*

$\mathcal{C} : \mathbf{stm} \longrightarrow \mathbf{Var}$:

1. *at* $\mathbf{fork}\{s_1, \ldots, s_n\}$ *statement*:-

$$\mathcal{C}(s_i) = \bigcup_{i \neq j} m_j, \quad where \quad s_i : m \rightarrow m_i$$

$$\mathcal{C}(t) = \mathcal{C}(s_i) \qquad for \qquad t \Subset s_i$$

2. *otherwise* $\mathcal{C}(s) = \varphi$

3.3.4 Anticipability Analysis

We now present anticipability analysis. For each program point it computes which non-trivial arithmetic expressions will be evaluated on all paths before any of their operands are modified. In the typing rule we use $eval(a)$ to denote $\{a\}$ if a is non-trivial expression and φ otherwise. For $a \in \mathbf{AExp}^+$ we define :-
$mod(x) =_{\mathbf{df}} \{a | x \in FV(a)\}$ *and* $mce(s) =_{\mathbf{df}} \{a | a \in mod(y). \forall y \in \mathcal{C}(s)\}$.
We use $\underline{s}$ to denote the full type derivation of $s : m \rightarrow m'$.
Inference rules of the type system are as follow:-

$$\frac{}{\underline{x := a} : (ant' \backslash mod(x) \cup eval(a)) \backslash mce(x := a) \rightarrow ant'} \qquad \frac{}{\underline{\textbf{skip}} : ant \rightarrow ant}$$

$$\frac{\underline{s_0} : ant \rightarrow ant'' \underline{s_1} : ant'' \rightarrow ant'}{\underline{s_0}; \underline{s_1} : ant \rightarrow ant'} \qquad \frac{\underline{s_t} : ant \rightarrow ant' \quad \underline{s_f} : ant \rightarrow ant'}{\textbf{if } b \textbf{ then } \underline{s_t} \textbf{ else } \underline{s_f} : ant \rightarrow ant'}$$

$$\frac{\underline{s_t} : ant \rightarrow ant}{\textbf{while } b \textbf{ do } \underline{s_t} : ant \rightarrow ant} \qquad \frac{ant \leq ant_0 \quad \underline{s} : ant_0 \rightarrow ant'_0 \quad ant'_0 \leq ant'}{\underline{s} : ant \rightarrow ant'}$$

$$\frac{\underline{s_i} : ant_i \longrightarrow Ant' \backslash mce(s_i) \qquad \forall i \in \{1, 2, ...n\}}{\textbf{fork}\{\underline{s_1}, \underline{s_2}, ...\underline{s_n}\} : \bigcup_{1 \leq i \leq n} ant_i \longrightarrow Ant'.}$$

The rules stated above follow the same line of corresponding rules introduced in [61] for the while language. The novelty of our work comes from fitting the fork statement into the type system of [61] and making necessary changes. We note that if no thread exists then for each statement s in the program $mce(s) = \varphi$. Besides the rule of **fork** statement which characterizes the multi-threaded concept. These rules prevent any modified expression *(i.e modified in concurrent threads)* from being used from the start of **fork** statement. As anticipability analysis is backward analysis, we remove modified expressions from Ant' of each statement. Also, in each assignment statement we remove modified expressions. All of these removals are guided by the set $mce(s)$.

3.3.5 Partial availability Analysis

It computes for each program point which non-trivial arithmetic expressions has already been evaluated and later not modified on some path through this program point and also anticipable.

Inference rules of the type system are:-

$$\frac{}{\underline{x := a} : (ant' \backslash mod(x) \cup eval(a)) \backslash mce(x := a), cpav \rightarrow ant', (cpav \cup aval(a) \backslash mod(x)) \cap ant'}$$

$$\frac{}{\underline{\textbf{skip}} : ant, cpav \rightarrow ant, cpav}$$

$$\frac{\underline{s_0} : ant, cpav \rightarrow ant'', cpav'' \; \underline{s_1} : ant'', cpav'' \rightarrow ant', cpav'}{s_0; s_1 : ant, cpav \rightarrow ant', cpav'}$$

$$\frac{\underline{s_t} : ant, cpav \rightarrow ant', cpav' \quad \underline{s_f} : ant, cpav \rightarrow ant', cpav'}{\textbf{if } b \textbf{ then } \underline{s_t} \textbf{ else } \underline{s_f} : ant, cpav \rightarrow ant', cpav'} \qquad \frac{\underline{s_t} : ant, cpav \rightarrow ant, cpav}{\textbf{while } b \textbf{ do } \underline{s_t} : ant, cpav \rightarrow ant, cpav}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad \underline{s} : ant_0, cpav_0 \rightarrow ant'_0, cpav'_0 \quad ant'_0, cpav'_0 \leq ant', cpav'}{\underline{s} : ant, cpav \rightarrow ant', cpav'}$$

$$\frac{\underline{s_i} : ant_i, CPAV \backslash mce(s_i) \longrightarrow Ant' \backslash mce(s_i), cpav'_i \qquad \forall i \in \{1, 2, ...n\}}{\textbf{fork}\{\underline{s_1}, \underline{s_2}, ...\underline{s_n}\} : \bigcup_{1 \leq i \leq n} ant_i, CPAV \longrightarrow Ant', \bigcup_{1 \leq i \leq n} cpav'_i}$$

The rules of program statements follow the same line of corresponding rules introduced in [61] for the while language. The novelty of our work comes from fitting the fork statement into the type system of [61] and making necessary changes. Here we excluded the expressions in concurrent modified of all threads of **fork** statement to avoid using these expressions after exiting **fork** statement.

3.4 Optimization Component

In this section we will introduce a type system with optimization components for multi-threaded programs. We mean by the notation $\underline{s} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s_*$ that, statement s with complete type system m, ant and $cpav$ is optimized to s_* Inference rules of the type system are:-

$$\frac{a \notin cpav \quad a \notin ant' \backslash mod(x)}{\underline{x := a} : (ant' \backslash mod(x) \cup eval(a)) \backslash mce(x := a), cpav \rightarrow ant', (cpav \cup aval(a) \backslash mod(x)) \cap ant' \hookrightarrow x := a}{:=}\textbf{1pre}$$

$$\frac{a \notin cpav \quad a \in ant' \backslash mod(x)}{\underline{x := a} : (ant' \backslash mod(x) \cup eval(a)) \backslash mce(x := a), cpav \rightarrow ant', (cpav \cup aval(a) \backslash mod(x)) \cap ant' \hookrightarrow nv(a) = a; x := nv(a)}{:=}\textbf{2pre}$$

$$\frac{a \in cpav}{\underline{x := a} : (ant' \backslash mod(x) \cup eval(a)) \backslash mce(x := a), cpav \rightarrow ant', (cpav \cup aval(a) \backslash mod(x)) \cap ant' \hookrightarrow x := nv(a)}{:=}\textbf{3pre}$$

$$\frac{}{\underline{\textbf{skip}} : ant, cpav \rightarrow ant, cpav \hookrightarrow \textbf{skip}}\textbf{skip}_{\textbf{pre}}$$

$$\frac{\underline{s_0} : ant, cpav \rightarrow ant'', cpav'' \hookrightarrow s'_0 \quad \underline{s_1} : ant'', cpav'' \rightarrow ant', cpav' \hookrightarrow s'_1}{\underline{s_0; s_1} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s'_0, s'_1}\textbf{comp}_{\textbf{pre}}$$

$$\frac{\underline{s_t} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s'_t \quad \underline{s_f} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s'_f}{\textbf{if } b \textbf{ then } \underline{s_t} \textbf{ else } \underline{s_f} : ant, cpav \rightarrow ant', cpav' \hookrightarrow \textbf{if } b \textbf{ then } s'_t \textbf{ else } s'_f}\ \textbf{if}_{\textbf{pre}}$$

$$\frac{\underline{s_t} : ant, cpav \rightarrow ant, cpav \hookrightarrow s'_t}{\textbf{while } b \textbf{ do } \underline{s_t} : ant, cpav \rightarrow ant, cpav \hookrightarrow \textbf{while } b \textbf{ do } s'_t}\ \textbf{while}_{\textbf{pre}}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad \underline{s} : ant_0, cpav_0 \rightarrow ant'_0, cpav'_0 \hookrightarrow s' \quad ant'_0, cpav'_0 \leq ant', cpav'}{\underline{s} : ant, cpav \rightarrow ant', cpav' \hookrightarrow [nv(a) := a | a \in cpav_0 \backslash cpav]; s'; [nv(a) := a | a \in cpav' \backslash cpav'_0]}\ \textbf{conseq}_{\textbf{pre}}$$

$$\frac{\underline{s_i} : ant_i, CPAV \backslash mce(s_i) \longrightarrow Ant' \backslash mce(s_i), cpav'_i \qquad \forall i \in \{1, 2, ...n\}}{\textbf{fork}\{\underline{s_1}, \underline{s_2}, ...\underline{s_n}\} : \bigcup_{1 \leq i \leq n} ant_i, CPAV \longrightarrow Ant', \bigcup_{1 \leq i \leq n} cpav'_i \hookrightarrow \textbf{fork}\{s'_1, s'_2, ...s'_n\}}\ \textbf{fork}_{\textbf{pre}}$$

The rules of program statements follow the same line of corresponding rules introduced in [61] for the while language, except the **fork** statement. The novelty of our work comes from fitting the fork statement into the type system of [61] and making necessary changes. Our new analysis also affect the component of optimizations as in the rule $\textbf{fork}_{\textbf{pre}}$, which achieves optimizations via optimizing each thread of its components.

3.4.1 Semantic soundness

We will present the soundness of the type system for PRE in the sense preservation of semantics, using relational method [2]. An original program and its optimized version simulate each other up to similarity relation $\sim$ on their states, indexed by conditional partial avialabilty types of program point[61].

Now we will define similarity relation $\sim$. Let $\sigma \sim_{cpav} \sigma_*$ denote that two states σ and σ_* of an original and optimized program agree on the auxiliary variables wrt. $cpav \subseteq \textbf{AExp}^+$ in the sense that $\forall x \in \textbf{Var}.\sigma(x) = \sigma_*(x)$ and $\forall a \in cpav.[\![a]\!]\sigma = \sigma_*(nv(a))$. Soundness can be obtained by the following theorem:-

Theorem 1. *Soundness of Optimization Component*
If $\underline{s} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s_*$ *and* $\sigma \sim_{cpav} \sigma_*$ *then*
- $\sigma \succ\!- s \rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{cpav'} \sigma'_*$ *and* $\sigma_* \succ\!- s_* \rightarrow \sigma'_*$*, -* $\sigma_* \succ\!- s_* \rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{cpav'} \sigma'_*$ *and* $\sigma \succ\!- s \rightarrow \sigma'$*.*

Proof The two parts of theorem are similar. We will proof the first part. The second is the same.

Given $\underline{s} : ant, cpav \rightarrow ant', cpav' \hookrightarrow s_*, \sigma \sim_{cpav} \sigma_*$ and $\sigma \succ s \rightarrow \sigma'$, we have to find σ'_* such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma_* \succ s_* \rightarrow \sigma'_*$.

The proof is by induction on the typing derivation. We will prove the following non-trivial cases.

- Case $:=_{pre}$: The type derivation is of the form

$$\overline{x := a : ant, cpav \rightarrow ant', cpav' \hookrightarrow s_*}$$

where $ant =_{df} ant' \backslash mod(x) \cup eval(a)$, $cpav' =_{df} (cpav \cup eval(a) \backslash mod(x)) \cap ant'$ We note that in particular this means that $cpav \cup eval(a) \supseteq cpav'$ and $cpav' \cap mod(x) = \phi$. The corresponding given semantic derivation must be of the form

$$\overline{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma]}$$

hence $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$

  - Subcase $:=_{1pre}$: We know that $a \notin cpav$. We also know that either $a \notin ant'$ or $a \in mod(x)$, so $cpav \supseteq cpav'$. Moreover, $s_* =_{df} x := a$.
    We have the semantic derivation

$$\overline{\sigma_* \succ x := a \rightarrow \sigma'_*}$$

    where $\sigma'_* =_{df} \sigma_*[x \mapsto [\![a]\!]\sigma_*]$. From $\sigma \sim_{cpav} \sigma_*$ it follows that $[\![a]\!]\sigma = [\![a]\!]\sigma_*$, so that , using $cpav \supseteq cpav'$ as well, we can conclude $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{cpav'} \sigma_*[x \mapsto [\![a]\!]\sigma] = \sigma_*[x \mapsto [\![a]\!]\sigma_*] = \sigma'_*$
  - Subcase $:=_{2pre}$: We have that $a \notin cpav$. We also have that a is nontrivial and $cpav \cup \{a\} \supseteq cpav'$. Also $s_* =_{df} nv(a) := a; x := nv(a)$. We have the

semantic derivation

$$\frac{\overline{\sigma_* \succ nv(a) := a \to \sigma''_*} \quad \overline{\sigma''_* \succ x := nv(a) \to \sigma'_*}}{\sigma_* \succ nv(a) := a; x := nv(a) \to \sigma'_*}$$

where $\sigma''_* =_{df} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_*]$ and $\sigma'_* =_{df} \sigma''_*[x \mapsto \sigma''_*(nv(a))] = \sigma''_*[x \mapsto [\![a]\!]\sigma_*]$. From $\sigma \sim_{cpav} \sigma_*$ it is immediate that $\sigma \sim_{cpav \cup \{a\}} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma] = \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* = \sigma'']$ and therefore by $cpav \cup \{a\} \supseteq cpav'$ we have $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{cpav'} \sigma''_*[x \mapsto [\![a]\!]\sigma] = \sigma''_*[x \mapsto [\![a]\!]\sigma_* = \sigma'_*]$.

– Subcase $:=_{3pre}$: We have that $a \in cpav$, soit follows that $cpav \supseteq cpav'$. We have $s_* =_{df} x := nv(a)$. We have the semantic derivation

$$\overline{\sigma_* \succ x := nv(a) \to \sigma'_*}$$

where $\sigma'_* =_{df} \sigma_*[x \mapsto \sigma_*(nv(a))]$. We know that $a \in cpav$, so from $\sigma \sim_{cpav} \sigma_*$ we learn $[\![a]\!]\sigma = \sigma_*(nv(a))$. Further, using also that $cpav \supseteq cpav'$, we realize that $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{cpav'} \sigma_*[x \mapsto [\![a]\!]\sigma] = \sigma_*[x \mapsto \sigma_*(nv(a))] = \sigma_*$.

- Case $conseq_{pre}$: The type derivation is of the form

$$\vdots$$

$$\frac{s : ant_0, cpav_0 \to ant'_0, cpav'_0 \hookrightarrow s_*}{s : ant, cpav \to ant', cpav' \hookrightarrow s'; s_*; s''}$$

where $(ant, cpav) \leq (ant_0, cpav_0), (ant'_0, cpav'_0) \leq (ant', cpav')$,
$s' =_{df} [nv(a) = a | a \in cpav_0 \backslash cpav]$ and $s'' =_{df} [nv(a) = a | a \in cpav' \backslash cpav'_0]$.
First we find σ_0 such that $\sigma_* \succ s' \to \sigma_0$ and $\sigma \sim_{cpav_0} \sigma_0$.
We have the semantic derivation $\overline{\sigma_* \succ s' \to \sigma_0}$
where $\sigma_0 =_{df} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* | a \in cpav_0 \backslash cpav]$. From $\sigma \sim_{cpav} \sigma_*$ using

$cpav \subseteq cpav_0$ we get that:-

$$\sigma \sim_{cpav_0} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma | a \in cpav_0 \backslash cpav]$$
$$\sigma \sim_{cpav_0} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* | a \in cpav_0 \backslash cpav]$$
$$\sigma \sim_{cpav_0} \sigma_0$$

since every expression in the difference of $cpav_0$ and $cpav$ is explicitly made equal to its corresponding auxiliary variable and no variables from **Var** are modified. From the induction hypothesis we obtain that there is a state σ_1 such that $\sigma_0 \succ\!\!- s_* \rightarrow \sigma_1$ and $\sigma' \sim_{cpav'_0} \sigma_1$. It is now enough to show that there is a state σ_* such that $\sigma_1 \succ\!\!- s'' \rightarrow \sigma'_*$ and $\sigma' \sim_{cpav'} \sigma'_*$. Similarly for the case of s', we have the derivation

$$\frac{}{\sigma_1 \succ\!\!- s'' \rightarrow \sigma'_*}$$

where $\sigma'_* =_{df} \sigma_1[nv(a) \mapsto [\![a]\!]\sigma_1 | a \in cpav' \backslash cpav'_0]$. Again it is easy to realize that $\sigma' \sim_{cpav'_0} \sigma_1$ with $cpav'_0 \subseteq cpav'$ gives us:-

$$\sigma' \sim_{cpav'} \sigma_1[nv(a) \mapsto [\![a]\!]\sigma' | a \in cpav' \backslash cpav'_0]$$
$$\sigma' \sim_{cpav'} \sigma_1[nv(a) \mapsto [\![a]\!]\sigma_1 | a \in cpav' \backslash cpav'_0]$$
$$\sigma' \sim_{cpav'} \sigma'_*$$

- Case $\mathbf{fork_{pre}}$: Soundness of the **fork** statement is as follow:

  We have $s_i : ant_i, CPAV \backslash mce(s_i) \longrightarrow Ant' \backslash mce(s_i), cpav'_i \hookrightarrow s'_i$.

  If $\mathbf{fork}\{\underline{s_1}, \underline{s_2}, ...\underline{s_n}\} : a, c \longrightarrow a', c' \hookrightarrow \mathbf{fork}\{s'_1, s'_2, ...s'_n\}$ and $\sigma_1 \sim_c \sigma_1^*$

  where $a = \bigcup_{1 \leq i \leq n} ant_i$, $c = CPAV$, $a' = Ant'$, $c' = \bigcup_{1 \leq i \leq n} cpav_i$ then we have to find β such that:

if $\sigma_1 \succ\!\!- \mathbf{fork}\{s_1, s_2, ..., s_n\} \rightarrow \sigma_{n+1}$ then $\sigma_{n+1} \sim_{c'} \beta$ and $\sigma_i^* \succ\!\!- \mathbf{fork}\{s'_1, s'_2, ...s'_n\} \rightarrow \beta$. The following is given:

if $s_i : ant_i, CPAV \backslash mce(s_i) \longrightarrow Ant' \backslash mce(s_i), cpav'_i \hookrightarrow s'_i$ and $\sigma_j \sim_{c_i} \sigma_j^*$

(where $j = \theta(i)$ and $c_i = CPAV \backslash mse(s_i)$) then:

$\sigma_i \succ\!\!- s_j \rightarrow \sigma_{i+1}$ implies the existence of γ_{i+1}^* such that:

$\gamma_{i+1}^* \sim_{cpav_j} \sigma_{i+1}$ and $\sigma_i^* \succ\!\!- s_j \rightarrow \gamma_{i+1}^*$

Choosing $\gamma_{i+1}^* = \sigma_{i+1}$, our choice is amenable where, it leads to the following:-

$$\sigma_i^* \succ\!\!- s'_j \rightarrow \sigma_{i+1}^* \quad \Longrightarrow \quad \sigma_1^* \succ\!\!- \mathbf{fork}\{s'_1, s'_2, ...s'_n\} \rightarrow \sigma_{n+1}^* \quad \ldots i$$

$$\sigma_{i+1} \sim_{cpav_j} \sigma_{i+1}^* \quad \Longrightarrow \quad \sigma_{n+1}^* \sim_{c'} \sigma_{n+1} \qquad \ldots ii$$

from i , ii and choosing $\beta = \sigma_{n+1}^*$ the proof is done.

3.5 Conclusion and Future Work

In this chapter the main contribution is the application of PRE to a multithreaded programming language. Up to our knowledge, this is the first deal with this problem. We use type systems as a tool to solve the problem. We designed a simple type system for optimizing multi-threaded programs. We approach the problem in a simple way; we use usual PRE with simple modifications. We look for variables that have been modified in other threads and exclude the expressions that contain any of the modified variables. For future work, we study more complicated optimization and consider using other tools. Many modifications can be applied.

4. Conclusion and Future Work

Chapter 4

Type Systems Based Data Race Detector.

4.1 Introduction

Developing and debugging software that depend on multi-threading is a trickly mission because of ingrained concurrency and indeterminism. There are many bugs occur according to these properties. Detecting and preventing these bugs are important areas of research. Bugs have several forms. The most extensively studied one is data-race: two concurrent threads accessing the same shared variable without proper synchronization. Data-race detector is a tool that determines whether a program is a data-race free or not. Two approaches are followed when developing detectors: static approach, and dynamic approach. Static detectors determine whether a program produce a data-race regardless of inputs of the program. Apart from static detectors, dynamic detectors determine whether a program produce a data-race of a given inputs at execution of the program.

The advantages of static detectors are consideration of different execution path (more elaborate), and the soundness of detector, i.e proving the bug-freeness of

programs. Examples of static detectors [29, 49, 73, 28]. On the other hand, dynamic detectors like [74, 66, 77] track program execution and repot a data-race problem if the program follow a certain concurrency order. These tools produce relevant result, according to order of execution or program inputs, and can not cover all execution paths; so are not sound.

Type systems can infer and gather information about programs as well as achieving program analysis. The merits of using type systems are attesting and rationalization of properties of programs directed by their phrase structures. Type systems are actually sufficient frameworks for describing data flow analysis . A general method for producing such description was presented [37].

Type systems are used as a framework for analyzing multi-threaded programs as well as imperative programs. In[15], type systems were used as a framework for pointer analysis for multi-threaded programs. In[21], type systems were used as a framework for eliminating redundancies in multi-threaded programs.

In this chapter we present a static detector. We introduce a type system that detects data-race problem for multi-threaded programs of a simple language *m-while*. We also prove the soundness of proposed type system. The rest of this chapter is organized as follows. Section 2 presents the language, a motivation example, and an operational semantics for the language. Read type system and the proof of its soundness are introduced in section 3. In section 4, we introduce safety type system and proof for its soundness. Realted and future works are outlined in section 5 and 6 respectively.

4.2 Motivation

In this section, we will present a simple example that demonstrates our motivations for this research. Firstly we will define a simple language, called *m-while*, that supports the multi-threading concepts. In this language, statements $s \in$ **Stm**, arithmetic expressions $a \in$ **AExp** and boolean expressions $b \in$ **BExp** are defined over a set of program variables $x \in$ **Var** in the following way:-

$$l ::= x \mid n$$

$$a ::= l \mid l_0 + l_1 \mid l_0 * l_1 \mid ...$$

$$b ::= l_0 = l_1 \mid l_0 \leq l_1 \mid ...$$

$$s ::= x := a \mid \textbf{skip} \mid s_0; s_1 \mid \textbf{if } b \textbf{ then } s_t \textbf{ else } s_f \mid \textbf{while } b \textbf{ do } s_t \mid$$
$$\textbf{fork } \{s_1\}, \{s_2\}, ..., \{s_n\} \textbf{ endfork}.$$

The following segment of a program motivates our work:-

$a := 4$

$b := 6;$

$x := a + b;$

fork

$\{\ y := a + b;\ z := b + 5;\ \},$

$\{\ a := a + 5;\ w := a + b;\ \},$

$\{\ x := x + 4;\ \},$

endfork;

This code shows that the variable a accessed by two threads; in the first thread with read operation ($y := a + b;$) and in the other thread with write operation ($a := a + 1;$). In this case data-race problem occurs.

4.2.1 Operational Semantics

The semantics is given in terms of states. The state is a pair, $\sigma = (\mathcal{R}, \mathcal{M})$, where $\mathcal{R}$ is a set of variables accessed by read operation, and $\mathcal{M}$ is a store. A store is a mapping from variables to integers $\mathcal{M} \in Store =_{df} \mathbf{Var} \rightharpoonup \mathbb{Z}$. The boolean and arithmetical expressions are interpreted as truth values and integers according to stores by the semantic function $[\![-]\!] \in AExp \cup BExp \rightarrow Stores \rightarrow Z$. For arithmetic expression $a \in AExp$, $[\![a]\!]\sigma$ denotes the arithmetic evaluation of a in the state σ. For boolean expression $b \in BExp$, $[\![b]\!]\sigma$ denotes the truth value of b in the state σ. We write $\sigma \models b$ to mean that $[\![b]\!]\sigma = tt$. We note that $\mathbf{FV}(a)$ is the set of free variables of expression a The operational semantics are defined by the following rules:-

$$\frac{}{x := a : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R} \cup \mathbf{FV}(a), \mathcal{M}[x \mapsto [\![a]\!]\mathcal{M}])}\,{}^{:=_{os}}$$

$$\frac{}{\mathbf{skip} : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}, \mathcal{M})}\,{}^{skip_{os}}$$

$$\frac{s_0 : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}'', \mathcal{M}'') \quad s_1 : (\mathcal{R}'', \mathcal{M}'') \rightarrow (\mathcal{R}', \mathcal{M}')}{s_0; s_1 : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}', \mathcal{M}')}\,{}^{seq_{os}}$$

$$\frac{\sigma \models b \quad s_t : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}', \mathcal{M}')}{\mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}', \mathcal{M}')}\,{}^{iftrue_{os}}$$

$$\frac{\sigma \not\models b \quad s_f : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}', \mathcal{M}')}{\mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}', \mathcal{M}')}\,{}^{iffalse_{os}}$$

$$\frac{\sigma \models b \quad s_t : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}'', \mathcal{M}'') \quad \mathbf{while}\ b\ \mathbf{do}\ s_t : (\mathcal{R}'', \mathcal{M}'') \rightarrow (\mathcal{R}', \mathcal{M}')}{\mathbf{while}\ b\ \mathbf{do}\ s_t : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}', \mathcal{M}')}\,{}^{whilet_{os}}$$

$$\frac{\sigma \not\models b}{\mathbf{while}\ b\ \mathbf{do}\ s_t : (\mathcal{R}, \mathcal{M}) \rightarrow (\mathcal{R}, \mathcal{M})}\,{}^{whilef_{os}}$$

$$\frac{s_{\theta(i)} : (\mathcal{R}, \mathcal{M}_i) \rightarrow (\mathcal{R}_i, \mathcal{M}_{i+1}) \qquad \forall i \in \{1, 2, ...n\}}{\mathbf{fork}\ \{s_1\}, \{s_2\}, ..., \{s_n\}\ \mathbf{endfork} : (\mathcal{R}, \mathcal{M}_1) \rightarrow (\bigcup_i \mathcal{R}_i, \mathcal{M}_{n+1})}\,{}^{fork_{os}}$$

where θ is a permutation on $\{1, 2, ..., n\}$

The rules show that the assignment statement actually changes the state; the free variables of the expression that has been evaluated are added to read variables, and

the store assigns new value for the variable being assigned a new value. The rules for imperative statements *skip, sequences, if,* and *while* changing the pre-state as usual for their classical meaning. The last rule (*fork* statement), that characterizes the multi-threading, describes that the fork statement changes a state using the states of each thread. One can see that our description of states helps in proving the soundness of our proposed type systems.

4.3 Read Type System

In this section, we introduce *read type system.* At each program point, the read type system determines the variables that have been accessed with a read operation. This type system acts as a flag to discover the overlapping of read type system and the concurrent modified set. A program point has type $r \subseteq Var$, if all variable in r are accessed by read operations (from beginning of the program to this point). The sub-typing is the set inclusion, i.e. $r \leq r'$ iff $r \subseteq r'$.
The rules of read type system will be as follow:-

$$\frac{}{x := a : r \to r \cup \mathbf{FV}(a)}{}^{:=_r}$$

$$\frac{}{\mathbf{skip} : r \to r}{}^{skip_r} \qquad \frac{s_0 : r \to r'' \; s_1 : r'' \to r'}{s_0; s_1 : r \to r'}{}^{seq_r}$$

$$\frac{s_t : r \to r' \quad s_f : r \to r'}{\mathbf{if}\ b\ \mathbf{then}\ s_t\ \mathbf{else}\ s_f : r \to r'}{}^{if_r} \qquad \frac{s_t : r \to r'}{\mathbf{while}\ b\ \mathbf{do}\ s_t : r \to r'}{}^{while_r}$$

$$\frac{r \leq r_1 \quad s : r_1 \to r_2 \quad r_2 \leq r'}{s : r \to r'}{}^{conseq_r}$$

$$\frac{s_{\theta(i)} : r \to r_i \quad \forall i \in \{1, 2, ...n\}}{\mathbf{fork}\ \{s_1\}, \{s_2\}, ..., \{s_n\}\ \mathbf{endfork} : r \to \bigcup r_i}{}^{fork_r}$$

Read Type System for *m-while* language.

The first rule $:=_r$ adds the free variables of computed expression to the pre-type. The rules $skip_r, seq_r, if_r$ and $while_r$ affect pre-type as expected considering

their classical meaning. The rule ($conseq_r$) is important for weakening the pre-type and strengthen the post-type. For the rule ($fork_r$) which characterizes the multi-threading, the post-type of fork statement is the union of all post-types of different threads.

4.3.1 Soundness of Read Type System.

In this section, we prove the soundness of read type system. Firstly, the following definition is introduced:-

Definition 4. *For a state $\sigma = (\mathcal{R}, \mathcal{M})$, we say that σ entails r, where $r \subseteq$ **Var** is the read type set, if $r \subseteq \mathcal{R}$. This is written as follow:- $\sigma \vdash r \Longleftrightarrow r \subseteq \mathcal{R}$*

The soundness of the concerned type system is introduced in the following theorem.

Theorem 1. *If $s : \sigma \to \sigma'$ and $s : r \to r'$, then $\sigma \vdash r \Longrightarrow \sigma' \vdash r'$*

Proof The proof is by structural induction on rules, we demonstrate some cases:-

- Case $:=_r$.

  We use the operational rule $:=_{os}$. Let $\sigma \vdash r \Longrightarrow r \subseteq \mathcal{R}$. Hence we have

  $r' = r \cup \mathbf{FV}(a) \quad \subseteq \mathcal{R} \cup \mathbf{FV}(a) \quad \subseteq \mathcal{R}' \quad \Longrightarrow \sigma' \vdash r'$

- Case $fork_r$.

  We use the operational rule $fork_{os}$. From premises the following are satisfied:

  $s_{\theta(i)} : r \longrightarrow r_i$, and $s_{\theta(i)} : \sigma_i = (\mathcal{R}, \mathcal{M}_i) \to \sigma_{i+1} = (\mathcal{R}_i, \mathcal{M}_{i+1})$.

  i.e. $\sigma_i \vdash r \Longrightarrow \sigma_{i+1} \vdash r_i$ or equivalently $r \subseteq \mathcal{R} \Longrightarrow r_i \subseteq \mathcal{R}_i$

  It is enough to prove that:-

$$\sigma_1 \vdash r \Longrightarrow \sigma_{n+1} \vdash \bigcup_i r_i.$$

$$\begin{aligned}\text{But, } \sigma_1 \vdash r &\Longrightarrow r \subseteq \mathcal{R} \\ &\Longrightarrow r_i \subseteq \mathcal{R}_i \\ &\Longrightarrow \bigcup_i r_i \subseteq \bigcup_i \mathcal{R}_i \\ &\Longrightarrow \sigma_{n+1} \vdash \bigcup_i r_i.\end{aligned}$$

4.4 Safety Type System

In this section, we introduce *safety type system*. Each program point has a type $d \in \{true, false\}$, where $true$ means that the program is safe at this point , and $false$ means that the program is unsafe at this point.In the following, $\mathcal{C}(s)$ denotes the concurrent modified set of statement s introduced in[21], and $p_1 \wedge p_2$ means the logical conjunction of boolean variables p_1 and p_2.

The following definition is needed:-

Definition 5. *The truth value of a set A is defined as follow:-*

$$Tr(A) = \begin{cases} true & A = \phi \\ false & A \neq \phi \end{cases}$$

For any two sets A and B the following properities are satisfied

$$Tr(A) \wedge Tr(B) = Tr(A \cup B) \quad and \quad Tr(A) \vee Tr(B) = Tr(A \cap B)$$

The rules of type system is defined as follow:-

$$\frac{x := a : r \longrightarrow r'}{x := a : d \rightsquigarrow d \wedge Tr(r' \cap \mathcal{C}(x := a))}{}^{:=}$$

$$\frac{}{\mathbf{skip} : d \rightsquigarrow d}{}^{skip} \qquad \frac{s_0 : d \rightsquigarrow d'' \; s_1 : d'' \rightsquigarrow d'}{s_0; s_1 : d \rightsquigarrow d'}{}^{seq}$$

$$\frac{s_t : d \rightsquigarrow d' \quad s_f : d \rightsquigarrow d'}{\textbf{if } b \textbf{ then } s_t \textbf{ else } s_f : d \rightsquigarrow d'}\,if \qquad \frac{s_t : d \rightsquigarrow d'}{\textbf{while } b \textbf{ do } s_t : d \rightsquigarrow d'}\,while$$

$$\frac{s_{\theta(i)} : d_i \rightsquigarrow d_{i+1} \quad \forall i \in \{1, 2, ...n\}}{\textbf{fork } \{s_1\}, \{s_2\}, ..., \{s_n\} \textbf{ endfork } : d \rightsquigarrow d_{n+1}}\,fork$$

Safety Type System for *m-while* language.

The first rule := checks the overlapping of concurrent modified set and the read set. All other rules are straight forward. In general we conclude that, the program is safe if each program point has a type *true* otherwise the program is unsafe.

4.4.1 Soundness of Safety Type System

Firstly we define the entailment of a type d in state $\sigma = (\mathcal{R}, \mathcal{M})$ with respect to set A as follow:-

$$\sigma \models_A d \Leftrightarrow d = Tr(\mathcal{R} \cap A)$$

The following theorem states and proves the soundness of the safety type system.

Theorem 1. *Let* $s : \sigma \rightarrow \sigma'$, *and* $s : d \rightsquigarrow d'$ *then*

$$\sigma \models_{\mathcal{C}} (s)d \Rightarrow \sigma' \models \mathcal{C}(s)d'$$

Proof

The proof is by structural induction on rules, we present some cases:-

- Case :=

  Suppose $x := a : \sigma \longrightarrow \sigma'$, and $x := a : r \longrightarrow r'$, where $\sigma = (\mathcal{R}, \mathcal{M})$, $\sigma' = (\mathcal{R}', \mathcal{M}')$, $r' = r \cup \mathbf{FV}(a)$, and $\mathcal{R}' = \mathcal{R} \cup \mathbf{FV}(a)$.

  From premises, $r \subseteq \mathcal{R} \Longrightarrow r' \subseteq \mathcal{R}'$.

  Let $\sigma \models_{\mathcal{C}(x:=a)} d$. Then $d = Tr(\mathcal{R} \cap \mathcal{C}(x := a))$.

Now $d' = d \wedge Tr(r' \cap \mathcal{C}(x := a))$

$= Tr(\mathcal{R} \cap \mathcal{C}(x := a)) \wedge Tr(r' \cap \mathcal{C}(x := a))$

$= Tr((\mathcal{R} \cap \mathcal{C}(x := a)) \cup (r' \cap \mathcal{C}(x := a)))$

$= Tr((\mathcal{R} \cup r') \cap \mathcal{C}(x := a))$

$= Tr((\mathcal{R} \cup r \cup \mathbf{FV}(a)) \cap \mathcal{C}(x := a))$

$= Tr((\mathcal{R} \cup \mathbf{FV}(a)) \cap \mathcal{C}(x := a))$

$= Tr((\mathcal{R}' \cap \mathcal{C}(x := a))$

$\Longrightarrow \sigma' \models_{\mathcal{C}(x:=a)} d'$

i.e. $\sigma \models_{\mathcal{C}(x:=a)} d \Longrightarrow \sigma' \models_{\mathcal{C}(x:=a)} d'$ which completes the proof.

- Case *seq*

  We use the operational semantic rule seq_{os}. From premises we have:-

  $\sigma \models_{\mathcal{C}(s_0)} d \Longrightarrow \sigma'' \models_{\mathcal{C}(s_0)} d''$, and $\sigma'' \models_{\mathcal{C}(s_1)} d'' \Longrightarrow \sigma' \models_{\mathcal{C}(s_1)} d'$,

  From the definition of concurrent modified function given by [21], we can conclude that for a sequence of statements $s_0; s_1$ the following satisfied:-

  $$\mathcal{C}(s_0) = \mathcal{C}(s_1) = \mathcal{C}(s_0; s_1)$$

  i.e. $\sigma \models_{\mathcal{C}(s_0;s_1)} d \Longrightarrow \sigma'' \models_{\mathcal{C}(s_0;s_1)} d''$, and $\sigma'' \models_{\mathcal{C}(s_0;s_1)} d'' \Longrightarrow \sigma' \models_{\mathcal{C}(s_0;s_1)} d'$, then we can conclude that $\sigma \models_{\mathcal{C}(s_0;s_1)} d \Longrightarrow \sigma' \models_{\mathcal{C}(s_0;s_1)} d'$.

- Case $fork$.

  To prove this rule we can consider the fork statement as the following sequence of statements **fork** $\{s_1\}, \{s_2\}, ..., \{s_n\}$ **endfork** $= s_{\theta(1)}; s_{\theta(2)}; ...; s_{\theta(n)}$. Now applying the sequence rule produce the proof.

4.4.2 Implementation

We implement a detector based on our type system. This program checks any program of *m-while* language for safety. For a program of *m-while* language, the detector computes the modified sets of each program point and computes read sets for each program point.Then the intersection of these two sets is calculated.

4.5 Conclusion and Future Work

Mathematical domains and maps between domains can be used to mathematically represent programs and data structures. This representation is called denotation semantics of programs [14, 20]. One of our directions for future research is to translate concepts of race detection to the side of denotation semantics. Doing so provides a good tool to mathematically study in deep race detection. Then obtained results can be translated back to the side of programs and data structures.

In this chapter we present a static data race detector. We use type systems as a frame work to implement this detector. Firstly, we present read type system which computes the variables accessed by read operations. Secondly, we present safe type system. This type system is based on read type system and decides if a program contains data race problems or not. The soundness of these type systems are discussed in this chapter as well. For future work we plan to use type systems as a tool to solve more complicated problems (like deadlock, pointer dangling). We expect type systems to be an amenable and trustable framework to deal with static analyses. We also plan to improve our work in many directions including extending our language to support the object-oriented concepts.

Bibliography

[1] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980.

[2] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 14–25, New York, NY, USA, 2004. ACM.

[3] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz. Symbolic data flow analysis for detecting deadlocks in ada tasking programs. In *Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '00, pages 225–237, London, UK, 2000. Springer-Verlag.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 159–170, New York, NY, USA, 1994. ACM.

[5] Dmitri Bronnikov. A practical adoption of partial redundancy elimination. *SIGPLAN Not.*, 39:49–53, August 2004.

[6] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks.

In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[7] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[8] Frederick Chow. *A Portable, Machine-independent Global Optimizer – Design and Measurements*. PhD thesis, Stanford University, 1984.

[9] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23:603–625, September 2001.

[10] Benjamin Cummings. Object oriented design with applications. page SEARCH IT, 1991.

[11] D. M. Dhamdhere. Practical adaption of the global optimization algorithm of morel and renvoise. *ACM Trans. Program. Lang. Syst.*, 13:291–294, April 1991.

[12] Dhananjay M. Dhamdhere. E-path–pre: Partial redundancy elimination made easy. *SIGPLAN Not.*, 37:53–65, August 2002.

[13] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, pages 85–96, New York, NY, USA, 1991. ACM.

[14] A. Mohamed El-Zawawy. *Semantic spaces in Priestly form*. PhD thesis, Brimingham University, 2006.

[15] Mohamed El-Zawawy. Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In Beniamino Murgante, Osvaldo Gervasi, Andrs Iglesias, David Taniar, and Bernady Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011*, volume 6786 of *Lecture Notes in Computer Science*, pages 355–369. Springer Berlin / Heidelberg, 2011.

[16] Mohamed A. El-Zawawy. Probabilistic pointer analysis for multithreaded programs. *ScienceAsia*, 47, December 2011.

[17] Mohamed A. El-Zawawy. Program optimization based pointer analysis and live stack-heap analysis. *International Journal of Computer Science Issues*, 8, March 2011.

[18] Mohamed A. El-Zawawy. Dead code elimination based pointer analysis for multithreaded programs. *Journal of the Egyptian Mathematical Society*, 2012.

[19] Mohamed A. El-Zawawy and N. Daoud. New error-recovery techniques for faulty-calls of functions. *Computer and Information Science*, To be appeared.

[20] Mohamed A. El-Zawawy and Achim Jung. Priestley duality for strong proximity lattices. *Electronic Notes in Theoretical Computer Science*, 158(0):199 – 217, 2006. Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII).

[21] Mohamed A. El-Zawawy and Hamada A. Nayel. Partial redundancy elimination for multi-threaded programs. *IJCSNS International Journal of Computer Science and Network Security*, 11:127–133, October 2011.

[22] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language*

design and implementation, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[23] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Race-free and memory-safe multithreading: design and implementation in cyclone. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10, pages 15–26, New York, NY, USA, 2010. ACM.

[24] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.

[25] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 1–13, New York, NY, USA, 2004. ACM.

[26] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[28] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 13–22, New York, NY, USA, 2009. ACM.

[29] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*, 21:627–676, May 1999.

[31] Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength reduction via ssapre. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 1998.

[32] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 224–234, New York, NY, USA, 1992. ACM.

[33] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, 16:1117–1155, July 1994.

[34] Jens Knoop and Bernhard Steffen. Code motion for explicitly parallel programs. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 13–24, New York, NY, USA, 1999. ACM.

[35] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 18:268–299, May 1996.

[36] Eric Koskinen and Maurice Herlihy. Dreadlocks: efficient deadlock detection. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 297–303, New York, NY, USA, 2008. ACM.

[37] Peeter Laud, Tarmo Uustalu, and Varmo Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theor. Comput. Sci.*, 364:292–310, November 2006.

[38] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. A constant propagation algorithm for explicitly parallel programs. *Int. J. Parallel Program.*, 26:563–589, October 1998.

[39] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 1–12, New York, NY, USA, 1999. ACM.

[40] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.

[41] B. Liskov and J. Guttly. *Abstraction and specification in software development.* MIT press, Cambridge, MA, USA, 1986.

[42] Jacques Loeckx, Kurt Sieber, and Ryan D. Stansifer. *The foundations of program verification.* John Wiley & Sons, Inc., New York, NY, USA, 1984.

[43] Ernest G Manes. *Algebraic approaches to program semantics.* Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[44] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proceedings of 1990 International Conference on Parallel Processing*, pages 105–113, 1990.

[45] Robert Milne and C. Strachey. *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA, 99th edition, 1977.

[46] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22:96–103, February 1979.

[47] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[48] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981.

[49] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[50] Phung Hua Nguyen and Jingling Xue. Strength reduction for loop-invariant types. In *Proceedings of the 27th Australasian conference on Computer science - Volume 26*, ACSC '04, pages 213–222, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[51] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[52] Karl J. Ottenstein. A simplified framework for reduction in strength. *IEEE Trans. Softw. Eng.*, 15:86–92, January 1989.

[53] V. K. Paleri, Y. N. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. Comput. Program.*, 48:1–20, July 2003.

[54] Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the java virtual machine language: 1. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '07, pages 10–, New York, NY, USA, 2007. ACM.

[55] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[56] John H. Reppy. *Higher-order Concurrency*. PhD thesis, Cornell University, 1992. Available as Computer Science Technical Report 92-1285.

[57] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 77–90, New York, NY, USA, 1999. ACM.

[58] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25:70–116, January 2003.

[59] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27:185–235, March 2005.

[60] Ando Saabas and Tarmo Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1-2):131 – 154, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).

[61] Ando Saabas and Tarmo Uustalu. Proof optimization for partial redundancy elimination. *Journal of Logic and Algebraic Programming*, 78(7):619 – 642, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[62] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[63] Vivek Sarkar. Code optimization of parallel programs: evolutionary vs. revolutionary approaches. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 1–1, New York, NY, USA, 2008. ACM.

[64] Vivek Sarkar. Challenges in code optimization of parallel programs. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 1–1, Berlin, Heidelberg, 2009. Springer-Verlag.

[65] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 633–655, London, UK, 1994. Springer-Verlag.

[66] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[67] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[68] Bernhard Scholz, Nigel Horspool, and Jens Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 221–230, New York, NY, USA, 2004. ACM.

[69] Jeffrey Sheldon, Walter Lee, Ben Greenwald, and Saman Amarasinghe. Strength reduction of integer division and modulo operations. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 254–273, Berlin, Heidelberg, 2003. Springer-Verlag.

[70] Nicholas Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, January 1993.

[71] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[72] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 70–82, New York, NY, USA, 2001. ACM.

[73] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[74] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In

Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI'08, pages 281–294, Berkeley, CA, USA, 2008. USENIX Association.

[75] Jingling Xue and Qiong Cai. A life time optimal algorithm for speculative pre. *ACM Trans. Archit. Code Optim.*, 3:115–155, June 2006.

[76] Jingling Xue and Jens Knoop. A fresh look at pre as a maximum flow problem. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin / Heidelberg, 2006.

[77] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.

Appendix A. Source Code

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <string>
#include <stdio.h>
using namespace std;
const int max=100;
string programLines[100];
int lengthOfProgram;
int forkPosition=0;
void removeWhite(string str,string &back){
        for(int counter=0;counter<str.length();counter++){
                if((str[counter]!=32)){
                        back+=str[counter];
                };
        };
};
class set{
private:
        char var[max];
        int  length;
public:
        set(){
                length=0;
        };
        bool exist(char ch){
                for(int i=0;i<length;i++)
                {
                        if(ch==var[i])
                                return true;
                }
                return false;
        };
        void add(set set1){
                for(int i=0;i<set1.length;i++)
                        add(set1.var[i]);
        };
        void add(char ch){
                if((ch>'0'&&ch<'9')||exist(ch))
                        return;
                var[length]=ch;
```

```
                length++;
        };
        void remove(char ch){
                for(int j=0;j<length;j++){
                        if(var[j]==ch){
                                for(int k=j;k<length;k++)
                                        var[k]=var[k+1];
                        };
                };
                length--;
        };
        void showSet(){
                for(int i=0;i<length;i++)
                        cout<<var[i]<<"     ";
                cout<<endl;
        };
        bool intersect(set s1){
                for(int i=0;i<s1.length;i++)
                {
                        if(exist(s1.var[i]))
                                return true;
                };
                return false;
        };
};
class statement{
private:
        string st;
        int type;
        set concurrentModified;
        statement* next;
        statement* s_t;
        statement* s_f;
public:
        set read;
        set modified;
        statement(){
                type=0;
                next=NULL;
        };
        statement(string str){
                st=str;
                next=NULL;
                s_t=NULL;
```

```
        s_f=NULL;
};
void write(){
        cout<<st;
        calcModified();
        modified.showSet();
        read.showSet();
};
void calcModified(){
        if(st[0]=='w'&&st[1]=='h'&&st[2]=='i'
                &&st[3]=='l'&&st[4]=='e')
        {
                int i=5;
                do{
                        i++;
                }while(!(st[i]=='d'&&st[i+1]=='o'));
                string temp="";
                for(int j=i+2;j<st.length();j++)
                        temp+=st[j];
                s_t= new statement(temp);
                s_t->calcModified();
                modified=s_t->modified;
                read=s_t->read;
        }
        else if(st[0]=='i'&&st[1]=='f')
        {
                int i=2;
                do{
                        i++;
                }while(!(st[i]=='t'&&st[i+1]=='h'
                                &&st[i+2]=='e'&&st[i+3]=='n'));
                string temp_f,temp_t="";
                for(int j=i+4;!(st[j]=='e'&&st[j+1]=='l'
                                &&st[j+2]=='s'&&st[j+3]=='e');j++)
                        temp_t+=st[j];

                s_t= new statement(temp_t);
                s_t->calcModified();
                modified=s_t->modified;
                read=s_t->read;
                for(int p=j+4;p<st.length();p++)
                        temp_f+=st[p];
                s_f= new statement(temp_f);
                s_f->calcModified();
```

```
                        modified.add(s_f->modified);
                        read.add(s_f->read);
                }
                else
                {
                        for(int i=0;i<st.length();i++){
                                if(st[i]==58&&st[i+1]==61){
                                        modified.add(st[i-1]);
                                        read.add(st[i+2]);
                                        read.add(st[i+4]);
                                };
                        };
                };
        };
};
class fork{
private:
        statement stms[max];
        set concurrent[max];
        int length;
public:
        fork(){
                length=0;
        };
        void addStm(statement st){
                stms[length]=st;
                stms[length].calcModified();
                length++;
        };
        void computeConcurrent(){
                for(int i=0;i<length;i++)
                        for(int j=0;j<length;j++){
                                if(i!=j)
                                concurrent[i].add(stms[j].modified);
                        };
        };
        bool safe(){
                computeConcurrent();
                for(int i=0;i<length;i++)
                {
                        if(concurrent[i].intersect(stms[i].read))
                                return false;
                };
                return true;
```

```
        };
        void show(){
                for(int i=0;i<length;i++)
                        stms[i].write();
        };
};
void readProgram(char* location){
    char line[max];
    ifstream inFile;
    inFile.open(location);
    if (!inFile) {
        cout << "Unable to open file. Check file name or location.\n";
        exit(1); // terminate with error
    }
        int i=0;
        ifstream infile1(location);
        while(!infile1.eof())
        {
                infile1.getline(line,max,'\n');
                removeWhite(line,programLines[i]);
                i++;
        };
        lengthOfProgram=i;
        infile1.close();
    for(int count=0;count<lengthOfProgram;count++){
                if(programLines[count]=="fork"){
                        forkPosition=count;
                        break;
                };
        };

};
int nextPosition(int p){
            for(int count=p;count<lengthOfProgram;count++){
                        if(programLines[count]=="fork"){
                                return count;
                        };
                };
                return -5;
};

#include "header1.h"
int main()
{
```

```
        statement temp;
        readProgram("sampleProgram2.txt");
        int position,kk=0;
        position=nextPosition(kk);
        while(position>0){
                fork default1;
        for(kk=position+1;programLines[kk]!="endfork";kk++)
                {
                    statement temp(programLines[kk]);
                default1.addStm(temp);
                };
                position=nextPosition(kk);
                if(default1.safe())
                        continue;
                else
                {
                        cout<<"The program is not safe."<<endl<<endl;
                        exit(1);
                };
        };
        cout<<"The program is safe."<<endl<<endl;
        return 0;
}
```