Preventing Single Variable Atomicity Violations Using The Ava Programming Language

Name: Kadeem Austin
Student ID#: 413000397
Course Title: Computer Science Major Research Project
Course Code: Comp 3920
Supervisor : Dr. Thomas Edward

Abstract

In an effort to offer continual CPU performance gains and to overcome physical limitations such as heat dissipation, designers have changed their focus from packing more transistors onto single core CPUs to developing multicore architectures. This shift has brought concurrency into the light and now developers must learn to leverage it with all the profound challenges it presents. Despite our efforts to adapt our current tools to a concurrent environment we have come up woefully short. Even in cases where more appropriate tools are used we still fail due to inexperience. When developers are not properly equipped to handle this challenge notoriously hard to identify and reproduce bugs are introduced into code, the major type of which are atomicity violations.
This paper presents the programming language Ava, designed to abstract the complexity of concurrent programming by using simple concurrency primitives and inferring synchronization between operations on shared data. These atomic sections of code are then enclosed in transactions thereby preventing single variable atomicity violations. Code written in Ava is compiled to C++ using the ava compiler which is then compiled using GCC to platform specific object code. When evaluated it was found that Ava compiler was able to successfully infer the correct atomic sections, however due to the large size of the atomic sections generated this can lead to reduced performance when compared to other hand crafted solutions. In the future, work on Ava will be focused on improving performance and extending its model to other classes of concurrency bugs.

Acknowledgements

I would like to sincerely thank Dr.Edward for his guidance and support throughout this process. His insightfulness and willingness to help, while encouraging me to believe in my ability to find my own answers and his patience while searching for those answers made this project possible. Additionally, I would like to thank my colleagues who offered insightful hints and much needed feedback along the way.

Table Of Contents

Abstract **1**

Acknowledgements **2**

Table Of Contents **3**

Introduction **5**

Concurrency Issues 5
Concurrent Programs & State 6
Atomicity Violations 7
Atomicity Violation Prevention 9

Design **11**

The Ava Programming Language 12
Primitives 12
Identifiers 13
Reserved Words 13
Operators 14
Variable Declaration 14
Control Flow 15
Conditional Execution 15
Loops 15
Function Declarations 15
Compound Data Structures 15
Built in Data Structures & Literals 16
Concurrency and Synchronization 16
The Ava Compiler 17
The Parser 17
Detecting Concurrent Functions 19
Detecting Shared Variables 20
Transaction Injection 21

Implementation **23**

Scanner 23
The Parser 24
Analysis 25
Code Generation & Transaction Injection 26

Evaluation 27

Analysis Of Results 28

Related Work 29

Conclusions & Future Work 31

References 32

Introduction

Over the past half century, microprocessor designers and manufacturers have made great strides in improving processor technology. These improvements have been the result of increases in clock speed and CPU cache size, in addition to optimization techniques such as pipelining, branch prediction and out-of-order execution.

The main focus of these improvements has been to increase processing speed and throughput and to that effect have been a remarkable success. However, the rate of improvement has been decreasing rapidly due to factors such as increased power consumption and greater heat generation, to name a few. This plateau in processor improvement has led many to look towards alternative, non traditional ways to continue to increase computer system performance. As a result microprocessor designers have turned to hyper-threaded and multicore architectures to continue to offer these performance improvements (Sutter 2005).

As these multicore processors become more commonplace, software developers will need to learn how to effectively take advantage of them in order to develop ever more powerful applications to meet the increasing demands of modern computing. This turn towards concurrency marks an important shift in the direction of software development requiring developers to become very familiar with concurrent programming techniques and practices.

But what is concurrency? Concurrency, in computer systems, refers to the execution of multiple sequences of instructions by one or more processors in overlapping time periods (Breshears 2009, 7). These sequences of instructions, also called threads, are generated by the system processes and are multiplexed into a one or more instructions streams for execution, one per processor.

Currently, In order to exploit concurrency in modern applications, we have tried to adapt our current programming models and paradigms to these new multi core architectures. These current models however are woefully inadequate for effective multicore programming (Venkatasubramanian, Shakuntala, Gul 1992).

Concurrency Issues

One of the major problems that arise is the use of direct references to mutable data. References, in their current incarnation directly bind an identifier, to an object's identity which is often the memory addresses where the data is currently stored. The problem with this model is that it allows all references to a specific data item to see any changes to the data's state in the instant it occurs and as a result references cannot account for temporality. All entities which use a reference will see changes to the state of the referenced object as they occur. In some cases this is desired behaviour, however in most cases, especially in concurrent programs, this can lead to unpredictable errors.

To illustrate this point consider the following example. Tomorrow is the birthday of a close friend. In an attempt to be a good friend you buy them a jacket that they've been wanting in their size. That night as you go to bed, you fall into a coma. You miraculously awake a year later on the day of their birthday. You then meet up with your friend, however, you find that they have put on some weight and your gift jacket is now too small.

Below is the above example, illustrated in code.

```
gift_jacket(Friend friend) {
        size = friend.shirt_size
        jacket = buy_jacket(size)
        sleep()                 // fall into a year long coma during sleep, friend's size increases
        meet_up(friend)
        give_jacket(friend)     // throws an error, size too small
        celebrate()
}
```

In this example, the variable `friend` is a reference to an object of type `Friend`. Since the `gift_jacket` function has a non temporal reference to `friend`, at all times it maintains a reference to the most recent state of the object. If some change occurs to `friend` halfway through the function, it can invalidate some assumptions made about `friend` earlier in the function's lifetime, such as the jacket size, leading to an error in give_jacket.

Concurrent Programs & State

Let’s assume that a process exists isolated from other processes and therefore cannot interact with them and any of it’s threads be interrupted at any point by another one of it’s threads. This processes’ ‘world’ then, is composed of all the threads that are part of the process.

If this process is single threaded and the main thread’s execution is halted for some reason, then the programs ‘world’ is also halted since there are no other threads. A software developer in this single threaded world then, can make assumptions about both the state of a program at the time it is halted and can assert that the state is unchanged when that thread is resumed. In this environment the binding of an object’s identity to its state is a reasonable approach.

If the process is multithreaded however, if any single thread is blocked, then the process’ ‘world’ will not halt because there are other threads executing simultaneously. Because threads share the same memory space, and may manipulate each other’s memory, a thread or more specifically a developer cannot make assumptions that a thread's state at the time of interruption will be the same at the time it is resumed, and any assumptions made by the thread before it is interrupted may be invalidated by the time it resumes. Another problem that arises is the potential for a thread to be interrupted in the middle of a multi step operation on some entity. Another thread that then attempts to use this entity may end up with an inconsistent view of the data.

In the example given, when you fell into the coma your world stopped, you being a single thread. For everyone else (the other threads) it didn’t. As a result you made a false assumption based on the state you saw before your world stopped. The same thing occurs in a multithreaded program.

Atomicity Violations

Atomicity is derived from the greek word atomos meaning indivisible. In the context of concurrency, atomicity is a property of a sequence of actions performed by a single thread where they are considered to be one indivisible unit of work. No thread should be able to interrupt or divide that instruction or sequence of instructions into separate parts by interleaving one of its own instructions into the sequence. If some thread X is able to interleave one of its instructions (read or write) into to the instruction stream of an

operation desired to be atomic occurring on thread Y, it may observe or change the intermediate state of the thread and this may lead to an atomicity violation. The resulting atomicity violation has the tendency to lead to unexpected system behavior.

Whether or not unexpected behavior occurs can be attributed to whether the instruction interleaving is serializable. Serializability is a property of the concurrent execution of multiple operations such that their data manipulation effect is equivalent to if these operations were performed one after the other (Lu, et al. 2006, 37-48). When various operation streams are interleaved in such a way that the resulting stream is not serializable, it is referred to as an unserializable interleaving.

Recent work by Lu(2008) observed that the basic types of unserializable interleavings can be reduced to and categorized by a sequence of three memory accesses. If we consider these accesses as being performed by two threads, T1 and T2 such that T1 performs two accesses interleaved by a single access by T2. Then the first access is referred to as the Current Access (c-access), the current access being considered. The Preceding Access (p-access), the access before the current one. And finally, the Remote Access (r-access), an instruction executed by a remote thread which did not execute the c-access and the p-access. When these three accesses are interleaved into an unserializable sequence, atomicity violations occur.

	Thread 1 (T1)	Thread 2 (T2)
S1:	if(s->head != null)	
		S2: s->head = null;
S3:	return s->head->value;	

Figure 1 - An example of an unserializable interleaving which may lead to a null pointer dereference (Getting the value of the top of a stack) S1 is the p-access, S2 is the r-access and S3 is the c-access.

If we consider that each of these accesses may be either a read or write then we can represent the various interleavings that may occur in the table displayed below. A read(r) access occurs when the value of a variable is used in evaluating in an expression. A write(w) on the other hand occurs when a variable is assigned to, or, for composite types, when one of its fields is assigned to.

S1	S2	S3	Buggy Interleaving
r	r	r	no
r	r	w	no
r	w	r	yes
r	w	w	yes
w	r	r	no
w	r	w	yes
w	w	r	yes
w	w	w	yes

Figure 2 - Shows all possible access interleavings that may occur and whether they represent a buggy interleaving

If these accesses are all performed on a single memory location then the resulting atomicity violation can be called a single variable atomicity violation. If however, these accesses are performed on multiple variable then it is called a multivariable atomicity violation.

Of the 8 possible access interleavings, only 5 are unserializable interleaving and may lead to concurrency bugs. Figure 3 shows an example of a bug caused by each type of unserializable interleaving.

Interleaving Sequence	Thread 1	Thread 2
R -> W -> R	*buffer* =alloc_buffer(length(**input**)) store(*buffer*, **input**)	**input** = *longer_input*
R -> W -> W	if(**head** == null) { **head* = *node_2*	**x** = *node_1*
W -> R -> W	**name** = "Reindhart" **name_length** = 9	if(**name_length** < 5) do …

W -> W -> R	**x** = 15 *y* = *z* / **x**	**x** = 0
W -> W-> W	***head** = new_node() *(head).x = 10	**head** = null

Figure 3 - Shows an example for each type of unserializable interleaving

Concurrency Models & Atomicity Violation Prevention

Atomicity violations are often unintentionally introduced into programs because developers often do not fully estimate the extent of an atomic sections, especially in the case of multistep operations including multiple variables. This negligence allows threads to observe and change the state of other threads and lead to unpredictable behaviour.

One approach for preventing these errors is to prevent the interruption of a thread by completely disabling interrupts. However for responsiveness and general system safety this is not a viable solution. A similar solution then, is to ensure that the scheduler does not schedule threads with potential conflicts to run when a given thread is performing the sequence of actions that correspond to its atomic section. Also because of the absence of locks there is no contention. To make this solution viable it would necessitate a language runtime scheduler or VM scheduler and short atomic sections to allow for throughput for all threads and so that a thread with an extensive atomic section does not lead to the convoy effect.

Another solution to these issues is the use of transactional memories. Whether at the software or hardware level, transactional memories work by transforming atomic sections into transactions. Two threads mutating the same value race to complete their transactions, the first one to finish successfully updates the variable, causing the incomplete transaction to retry. Though it prevents contention it can lead to unnecessary work being done. This method is good when conflicts are rare and transactions are allowed to complete successfully.

An alternative to transactional memories and the dominant model for concurrent programs is locks. Locks are based on the concept of critical sections, they only allow one thread to enter into a critical section at a time. Locks can be generalized as

semaphores, which allow one or more threads to access a set of resources. The readers writers problem has led to the development of read-write locks which are more efficient than normal locks due to allowing multiple readers to access a resource without preventing each other from reading. However it blocks on writers. Due to being the prominent model for concurrency, locks have been the focus of extensive research and has led to multiple lock implementations and high performance lock techniques.

Locks based programming however, is notoriously hard to program in and has the tendency to lead to unintelligible code and deadlocks. Higher level and more abstract concurrency primitives such as actors and communicating sequential processes are now used to get around the difficulty of explicit locking. The ideas between the actor model and communicating sequential processes are very similar and generally boil down to independent entities (actors, processes) sending messages between each other and that is how they share data. Each entity is responsible for executing some behavior based on the objects they receive. This is less prone to errors when compared to lock based concurrency but because of the added abstraction are often slower than locks.

These methods above focus on synchronizing the sharing of data. However, functional programming suggests an alternative, preventing shared state altogether. Functional programming is useful because no data can be shared, therefore each thread gets a copy of the data it intends to update and makes it a local copy. As a result this copy cannot be changed when the thread is interrupted and the programmer can reason sequentially about the program.
The naive way to do this would be to use a copy on write system where each thread shares the data structure if they are reading from it. But upon writing, the thread makes a copy of the data and then performs its changes to the data. When the thread is finished it signals an update event and when all other reading threads are finished it is allowed to update the shared copy.

Copy on write systems, as one would assume are very inefficient due to the large amount of memory they use. A more efficient alternative would be to use persistent data structures. Persistent data structures are so called because updates do not destroy the previous copy of the data, but maintain it while creating a new version which shares data with the previous version.

Persistent data structures come in 3 variants based on the types of operations they allow.

Partially persistent data structures allow reads from any previous versions but only write to the most recent version. Full persistence allows reads and writes to any version of the data structure while confluent persistent allow reads and writes on any version and also allows 2 versions to be merged to create a new version.

Each of the above methods prove effective in certain cases but also have their drawbacks. In order to fully harness the power of concurrency a developer would be required to have knowledge of all these techniques and correctly know when and how to apply them. This mental burden in addition to preserving business rules in a rapid paced environment has the tendency to lead to many concurrency bugs being introduced.

The goal therefore is to design a concurrent language which abstracts the complexity of concurrent programming, in turn allowing developers to be more productive and be guaranteed to have thread safe code. The design of this language, called the Ava programming language will be presented in the following section.

Design

The Ava Programming Language

This section will offer a brief overview of the Ava programming language focusing on its key features and design philosophy. Ava is a statically typed descendent of the C programming language. It is developed on top of C++ and uses a subset of the language's primitives. The main design philosophy behind Ava is simplicity, with a special focus on concurrency. As such, Ava following the fork-join paradigm contains one mechanism for creating threads, the 'spawn' statement and two synchronization primitives 'wait' and 'wait_all' which are used for blocking until one thread finishes and blocking until all threads spawned by the current thread are finished, respectively. By keeping the language simple it allows developers to reason more easily about both serial and concurrent programming, reducing time to market and increasing both programmer and business productivity.

Primitives

Ava uses a small set of datatypes. Each numeric type has an alias which may be more familiar to users coming from other languages.

- Integer Types - Aliases are enclosed in braces
    - int8 (byte)
    - int16 (short)
    - Int32 (int)
    - Int64 (long)
- Floating Point Types -
    - float32 (float)
    - float64 (double)
- Character Sequences
    - char (single character sequence)
    - string (multi character sequence)
- Boolean Type
    - bool

Identifiers

An identifier is any sequence of characters conforming to the pattern [_a-zA-Z][_a-zA-Z0-9]*.
Identifiers can be used to refer to a variable, function or struct. An identifier is valid if it is not an Ava reserved word (outlined below) or a C++ reserved word.

Reserved Words

These are words with special meaning to the AVA compiler and cannot be used as identifiers.

and	or	not	for
break	goto	continue	if
else	return	struct	true
false	spawn	wait	wait_all
int	char	string	unsigned
float	null	var	function

Since AVA is a superset of the C++ programming language, the use of C++'s reserved words is also, not permitted. Therefore the additional keywords below are renamed by the compiler by prepending an underscore to their use in the AVA language.

auto	long	switch	enum

register	case	extern	union
short	unsigned	const	signed
void	sizeof	volatile	default
static	while	do	double

Operators

AVA preserves all arithmetic, comparison, assignment and bitwise operators from the C++ programming language. All logical operators are replaced by keywords which perform the same function. ‘&&’ is replaced by ‘and’, ‘||’ is replaced by ‘or’, ‘!’ is replaced by ‘not’. Since these logical keywords are replaced by C++ style logical operators during compilation, the operator precedence for AVA is the same as that of C++.

Variable Declaration

Given the various types and the rules for identifiers given above, the syntax for variable declarations follow below:

Var <identifier> : <type> ;

Variables can also be declared and initialized with the following syntax:

var <identifier> : <type> = <expression>;

Flow Control

Conditional Execution

Branching and conditional execution is performed by if-else statements, identical to those in C++.

Loops

There is only one looping construct in AVA and that is the for loop. AVA for loops follow a similar style to C++ style for loops. In order to perform a while loop a for loop of the following form is used:

for <condition> <block>

range based for loops follow the form

for var <identifier> : <type> = from .. to <block> [from and to are integers to > from]

Function Declarations

for functions that accept parameters:

<ident>(<ident> : <type> [, <ident> : <type>]*) -> <type> <block>

e.g : sayHello(name: string) -> str { return sprintf("Hello there %s!\n", name); }

for functions which do not accept parameters:

<ident> -> <type> <block>

e.g : function sayHello -> str { return sprintf("Hello there!\n"); }

All functions must indicate a return type, in cases where no data is returned this is indicated with a return type of void.

Compound Data Structures

Ava eschews classes in favor of structs and interfaces in a fashion similar to the Go programming language. Structs are declared using the struct keyword :

```
struct Person {
        name: string
        age: int
}
```

Fields are accessed using the dot syntax: <identifier>.<field>

Struct methods are declared using a special form of the function declaration syntax where the struct/interface name is the first parameter in the parameter list preceded by the special symbol @.

For example : sayName(@Person p, Person target) -> string { return sprintf("Hello %s, My name is %s", target.name, p.name); } declares the method 'sayName' on the Person object which takes a single Person as a parameter.

Interfaces on the other hand are declared using the interface keyword as follows:

```
Interface Reader {
        read(src: Source) -> (int, string)
}
```

Arrays

In Ava an array literal is constructed using []. In order to create an array of any type the syntax is:

[<type>] <identifier> = [<type_literal> [, <literal>]*];

So to create an array of integers from 1 to 10 we say

```
[int] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
```

Concurrency and Synchronization

Ava's concurrency model is a fork join model. The spawn statement is used for creating a new user level thread that executes a given function and will return an integer used to identify the new thread that was created. For synchronization of threads the keyword wait is used. This takes a thread id, in this case x, and waits until the terminates. In addition to the wait keyword which only works on single threads, Ava also uses the wait_all keyword which is equivalent to calling wait for all functions spawned within the current thread up to the point at which wait_all occurs. wait_all also returns the return values of all functions executing concurrently in the order in which they were spawned.

The Ava Compiler

This section describes the various major components and subcomponents of the Ava compiler and their functionality.

The Scanner

A simple scanner that splits character streams by tabs, spaces and newlines.

The Parser

Ava uses a bottom-up LALR parser to construct a parse tree of the program to be used by the static Analyzer.

The Static Analyzer

The main purpose of the static analyzer is to identify sections of code that should be executed atomically and insert annotations to indicate the start and end point of these atomic sections.
The algorithm used to detect atomic sections is a simplified version of the algorithm proposed Kempf, Veldema and Philippsen (International Conference on Compiler Construction, 204-223, 2013).This algorithm is presented in the appendix . Like their algorithm the algorithm used here will use data flow dependency analysis and control dependency analysis to detect shared variable dependencies and then use an annotation to identify the start of the critical section.

The task of identifying critical sections can be broken down as such:

1) Detecting which functions may be run concurrently to each other
2) Detecting variables which may be shared between those functions
3) Detecting explicit dependencies between shared variables within concurrent functions

Detecting Concurrent Functions

The first step in effectively preventing atomicity violations is detecting which functions may be run concurrently to others. Detecting which sections of code may be run concurrently is called the may-happen-in-parallel(MHP) problem. It is a decision problem which considers a pair of actions(functions) in a program and determines whether

these two actions may be run in parallel in some execution of the program. These functions, will be called concurrent functions. This phase is relatively simple and from a high level perspective, concurrent functions can be identified using 3 rules.

A function is a concurrent function if:

1) It is the subject of a spawn statement
2) A function is spawned within its body
3) It is called within the body of a concurrent function

Once a function is identified as a concurrent function the analyzer will add the function to the global concurrent function table called ccfns. The algorithm for identifying concurrent functions called MHP is outlined below.

```
function add_spawned(Function* f,
List[Function*] ccfns):
 for all statement in f.body do
  if type(statement) is 'SpawnStatement'
then
   fn ← statement.target // the function
being spawned
   if !ccfns.contains(fn) then
    ccfns.add(fn);
    identify_concurrent(fn, ccfns);
   endif
  endif
 end
 return ccfns;
end function
```

```
function add_nested_calls(function f,
List[Function*] ccfns):
 if ccfns.contains(f) then
  for all statement in f.body() do
   if type(statement) is 'FunctionCall' then
    fn = statement.callee
    if !ccfns.contains(fn) then
     ccfns.add(fn)
     add_called(fn, ccfns)
    end if
   end if
  end
 endif
 return ccfns;
end function
```

```
function MHP():
 global ccfns;
 global main;
 add_spawned(main, ccfns);
 add_called(main, ccfns)
end function
```

MHP works by first calling add_spawned a recursive procedure which is called on the main function and all spawned functions adding other spawned functions ccfns. Once these functions have been identified MHP then calls add_called which looks for functions which have been called in spawned functions and also adds them to ccfns.

Detecting Shared Variables

This phase occurs during symbol table generation. During this phase a component called the Shared Variable Detector (SVD) analyses each concurrent function for all variables used and tries to separate local variables from non local variables..

The algorithm for this procedure follows directly below:

```
function SVD(SymbolTable* symtab, List[Function*] fds):
// where fds is a list of all function declarations
 for all fd in fds do
  curr_table = symtab.addGlobalTable(fd.id)
  for all statement in fd.body do
   if type(statement) is 'Expression Statement' do
    for all var in statement.variables_used do
     if !islocal(var, curr_table, fd.id) then shared.add(var.id)
      end if
    end
   if type(statement) is 'VariableDeclaration' do
    curr_table.add(statement)
   end if
  end
 end
end function
```

Identifying Critical Sections

This phase is the most crucial part of the entire static analysis process. The key to preventing atomicity violations is in properly determining the extent of the critical section. Once shared variables have been determined, any statement that includes the use of that shared variable needs to be included in the critical section. It consists of linearly scanning the body of a function and inserting a 'start' annotation before the first

use of a variable in the function's *svset* and an 'end' annotation after the last use of a variable in the *svset*.
This however is not enough to fully determine the extent of the atomic section. Next a data flow dependency analysis is performed to identify all static relationships involving the variables in the function's *svset*. That is to say, if a sequence of instructions like: *b* = *a*; *c* = *b*; *d* = *c* occurs, and instruction '*a*' contains a shared variable then the statements *b,c,d* are to be included in the functions critical section by moving the 'end' annotation after the '*d*' statement.

Implicit dependencies are those which cannot be inferred from the code. An example of this is implementing a queue using two stacks. Once inserted into the queue the element should remain in the queue. If however an intermediate thread reads the queue while one element is being popped from one stack and placed onto the other then the element appears to not be in any of the 2 stacks and hence not in the queue. In these cases, unfortunately, we must rely on the programmer to give an annotation so that to compiler would know to enclose instructions operating on these variables in the atomic section.

Transaction Injection

After the compiler pass which indicates the start point of transactions comes the transaction injection pass. On this pass, the compiler injects transaction start points by replacing start point annotations with the code for creating a transaction and replace the end annotation with the code for closing a transaction.

Implementation

The process of developing the Ava language was separated into 4 distinct phases based on the components of the compiler. These four phases were:

- The Scanner
- The Parser
- The Static Analyzer
- Transaction injection and Code Generation

Scanner

The process of scanning/tokenizing a language involves recognizing tokens from an input stream by the rules of the language's microsyntax. These rules can either be hand coded into a custom scanner or can be defined in some format for use with a scanner generator tool. Originally, the Ava compiler used the former method however due to time constraints the latter method was chosen. The tool chosen for scanner generation was GNU Flex. Flex takes as input a '.lex' file which contains the rules of the microsyntax defined as a collection of regular expressions. This file was used to define the rules for lexical elements such as keywords (outlined in the design section), numeric and string literals in addition to unary and binary operators.

This '.lex' file is compiled using the flex command line interface to a C++ class called 'FlexLexer' defined in a 'lex.yy.cc' file which can then be included into other files. The FlexLexer class defines a method called 'yylex' which when called returns the token id of the next token in the input stream and returns 0 if the input stream is exhausted. Reading all the tokens from the input stream then was a simple matter of calling 'yylex' in the body of a while loop. A simplification of the code used is presented below.

```
int main( int /* argc */, char** /* argv */ ) {
   FlexLexer* lexer = new yyFlexLexer;
   Int token_id = -1;
   while(token_id = lexer->yylex() != 0) {
         // do something with token id
   }
   return 0;
}
```

This class was then passed on to the next phase of the compiler, the parser.

The Parser

Parsing is the process of taking the token stream generated by the scanner and checking whether the sequence conforms to certain syntactic rules, which can be considered the macrosyntax of the language. The parser then builds a parse tree of all the identified syntactical elements of the language. An example of a parse tree for an arithmetic expression is presented below.

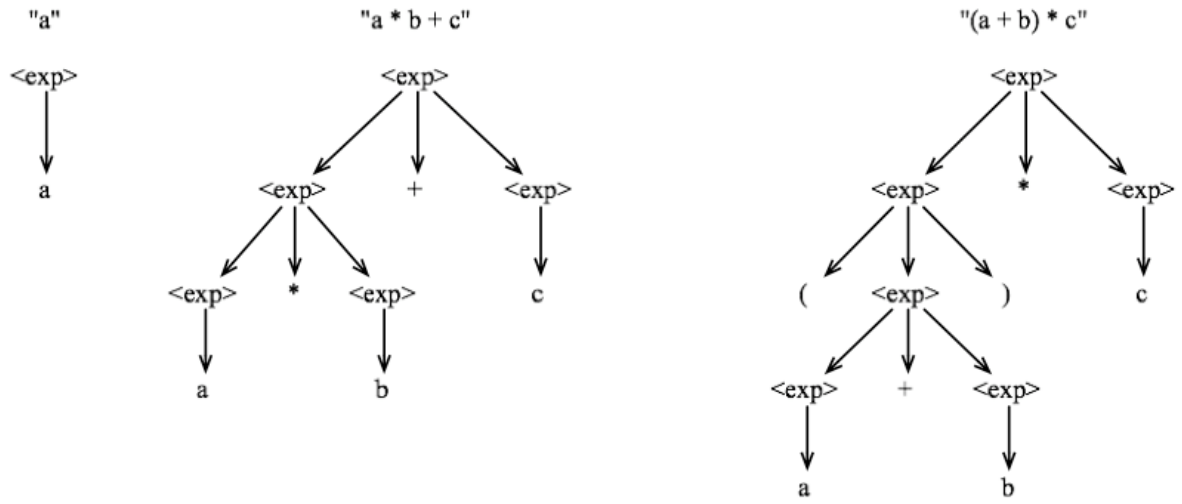
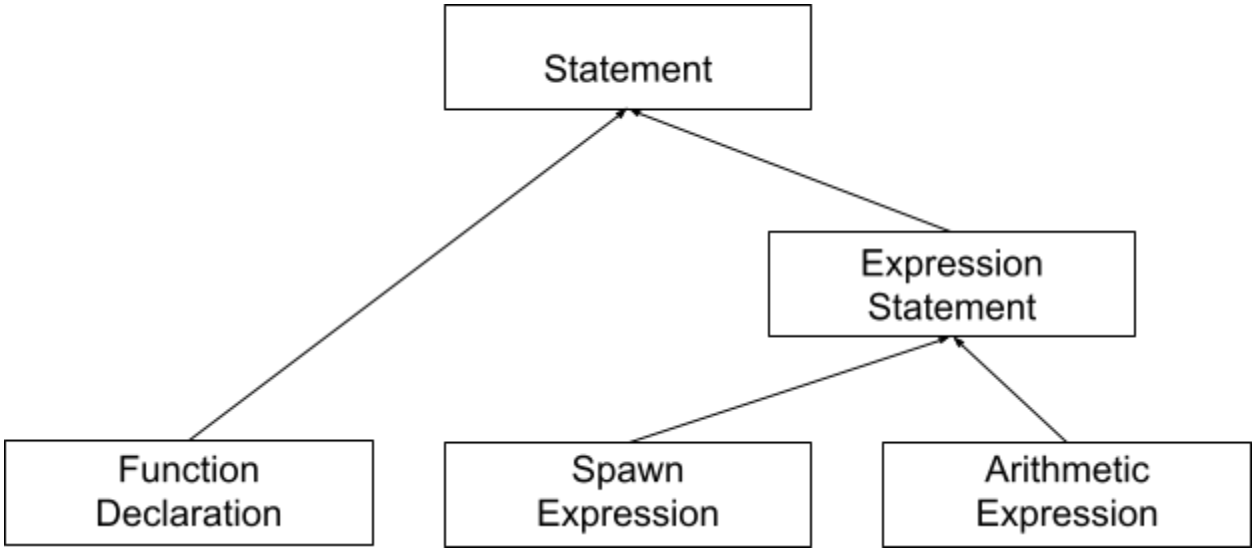


Figure X. A collection of parse trees generated from arithmetic expressions (taken from https://upload.wikimedia.org/wikipedia/commons/0/00/Grammar_example1_IPL.png)

The Ava parser was created using the parser generator tool GNU Bison. Although bison can be used with a custom built scanner it was mainly built to work with Flex and consequently are often used together, as is the case here.

Like Flex the rules (productions) of the syntax are defined in a file which is used to generate the parser. The language used to define the rules of the syntax is based on Backus Naur Form and as such is quite simple and convenient. A C++ function can be associated with each production to perform any desired action when a stream of tokens matches this production. These functions were used to create nodes for the parse tree, for example, when the production for a function declaration occurs the callback created a FunctionDeclaration object and attached it to the parse tree. Again, once compiled with a bison command line interface, a C++ class is created called ‘Parser’ and defined in the ‘parser.h’ file. This parser class defines a method called parse which is with the scanner class as a parameter.

This method returns a parse tree which can then be traversed. There are 2 classes of nodes in the parse tree: the ‘Statement’ and the ‘ExpressionStatement’ which inherits from the statement class. A simplified version of the inheritance diagram of the various nodes is shown below.



Each node defines a method called codeGen which is used to produce the corresponding C++ code to achieve its purpose. The parse tree / abstract syntax tree produced by the parser is crucial in the next phase, analysis.

Analysis

For the static analysis part of the compiler a copy of the parser was created with the actions for each production replaced with various analysis procedures. The creation of a duplicate parser instead of including analysis logic in the original parser was done for 2 reasons (i) to keep the phases distinct and true to the logical order presented in the design (ii) The nature of LALR parsers means that the actions assigned simpler productions are executed before the actions of more complex productions which in most cases was contrary to the execution order desired by the static analysis. An example of the latter is where for analysis reasons we would like to determine whether a function contains a spawn statement. In this case it would be better to perform a top down analysis where the function statement is recognized, the function id then created and passed on to a recursive procedure applied to the statements in the body to determine the function contains a spawn statement and registering the function as such with its id.The analysis parser creates a simplified parse tree only containing nodes relevant to the analysis procedures that follow. This parse tree is then used by procedures like MHP and SVD. After the static analysis and the atomic sections are determined annotation nodes are inserted into the parse tree generated by the original parser before the statement node where the atomic section begins. An ending annotation is also inserted into the tree after the statement node

where the atomic section ends. Each of these nodes implement the codeGen method and generate the appropriate c++ code.

Code Generation & Transaction Injection

After static analysis the parse tree is traversed in preorder generating the code for each node in the tree. The language chosen as the output was the C++ language mainly to maintain consistency in the compiler implementation language and output language, however it is worthwhile to note that this was not absolutely necessary and some language with a better and more flexible Software Transactional Memory (STM) implementation could have been selected in retrospect.

Since C++ has no stable STM implementation the GCC v4.7 experimental STM implementation had to be used. The problem with this implementation is that it does not allow for any side effects occurring in the body of the transaction. This severely limits the usefulness of the transaction block since certain statement such as variable declarations, which require memory allocation must be moved outside the scope of the transaction block.

In order to compensate for this, a procedure had to be developed which applies a transformation to the parse tree so as to move these types of statement outside of the transaction. In cases where this would lead to problems with variable shadowing such as where a variable in an inner scope with the same name as another variable in an outer scope are now moved up to the same scope the innermost variable is assigned a unique name using their id which is then propagated through that function replacing the old name.

Once these conflicts are resolved the transaction is inserted and the code is generated. This C++ code is compiled using GCC into object code for the particular platform.

Evaluation & Analysis Of Results

Given the goals of the project the evaluation of the Ava compiler is focused on 2 main aspects, (i) The correctness of the atomic sections generated and (ii) the execution time of the programs generated.

Methodology

In order to evaluate the correctness the the atomic sections generated by the compiler a set of programs were selected from the Stanford Transactional Applications for Multi Processing (STAMP) parallel processing benchmark suite. These programs were selected due the presence of atomic sections which have been analyzed and protected via synchronization inserted by experts. On this basis, two assumptions were made (i) the atomic sections and related synchronization mechanisms of the programs selected are correct and (ii) if the Ava compiler generates similar atomic sections then the Ava compiler produces correct code.

In preparation for analysis the programs selected were stripped of their synchronization primitives and rewritten in Ava to be compiled and have their atomic sections evaluated. Due to the large size of the programs in the STAMP suite only a few of the smaller programs were chosen. While it may be useful to infer from the potential performance of these programs it is not a thorough proof of the correctness of the techniques used. In order to conduct this more thorough evaluation a mathematical proof may be necessary.

The sample programs selected from STAMP were bayes and k-means a machine learning algorithm for learning the structure of bayesian networks and a data mining algorithm from observed data and grouping objects into n-dimensional space into k clusters respectively.

The following table shows the results of compiling and executing these programs using the ava compiler. The benchmarks were performed on a laptop with an Intel Core i5-6200u 2.3 GHz Dual-core CPU with 8Gb DDR4 memory.

Algorithm	Correct Atomic Section	Execution Speed Ava Compiled	Execution Speed Native
Bayes	True	9.68 sec	9.43 sec
K-means	True	16.35 sec	11.69 sec

For both programs used the Ava compiler generated the correct atomic sections defined as enclosing the first statement in the transaction of the original programs and the last statement enclosed in the last transaction. In the case of k clusters the compiler actually overestimated the size of the atomic section which indicates either an error in the logic of the atomic section detection or in its implementation. The atomic sections generated in each program were the same on multiple compilations. Since Ava currently does attempt to make the large atomic section that it generates, more granular it is to be expected that the speed of the ava compiled program may not be as quick as the STAMP version. In some informal tests it was found that in k-means algorithm, the performance with a single atomic section was actually better, which may be due to the overhead in the creation of multiple atomic sections.

The granularity of the generated atomic sections is likely to become an issue in the future and while in some cases it performs better due to the cost of creating multiple transactions, in larger programs it may lead to significant performance degradation and a method of reducing the atomic sections will need to be developed

Related Work

A recent study by Lu, et al. (2008, 329-399) found that concurrency bugs can be classified into a few categories based on their root causes. Excluding deadlock related bugs, they found that two of the most common bug patterns, accounting for approximately 97% of all concurrency bugs are atomicity violations and order violations. The most prevalent of these 2 however, are atomicity violations which account account for over 70% of non deadlock concurrency bugs. Atomicity violations violations come in two flavors, single variable atomicity violations and multivariable atomicity violations. Of the body of research on atomicity violations much of it is focused on single variable violations with only a few papers catering to multivariable violations.

Due to the difficulty in reasoning about concurrent programs the automatic detection of atomicity violations in these programs has been an area of research interest. This research has led to the development of several tools and techniques which are used to analyze code both statically and dynamically and detect these bugs in various programming languages. Static analysis approaches such as those proposed by Engler, Dawson and Ashcraft(2005, 237-252) often employ programmer included annotations to give hints to the runtime environment which enforces these rules. Systems such as these can be called explicit since they require the programmer to specify metadata to enable the tool to function as expected.

On the other hand, implicit systems such as those proposed by Chew, Lee & Lie(2010, 307-320) statically analyze code and may or may not generate internal annotations. These systems are implicit because they do not require the developers to specify the annotations themselves.

In addition to static analysis there has been research into dynamic analysis of concurrent programs. Dynamic analysis, monitors the execution of concurrent code and tries to enforce atomicity and prevent violations during runtime. Static analysis methods can often be more thorough when analyzing code because they are able to examine all existing code paths. Because of this visibility however, they often produce many false positives.

Dynamic methods on the other hand, though more efficient are not always as thorough as static analysis, as it cannot analyse paths that do not occur during the monitored execution. Hybrid approaches have been taken to combine the benefits of both methods. For example, Kivati

(Chew, Lee & Lie 2010), uses static analysis to detect potential locations which may require atomicity enforcements and marks these areas. They then useshardware watchpoints to identify potential violation and dynamically reorders this interleaving. This system is very efficient as it only imposes a 19% average run time cost while successfully preventing atomicity violations.

Though the methods of detecting and preventing atomicity violations explored above, are effective, they are all designed to be reactive, i.e they seek to detect and prevent atomicity violations after they have been introduced into programs. Methods that focus on proactively preventing atomicity violations, that is, providing mechanisms for detection and prevention while writing source code are comparatively few.

An example of one of these works is by Flanagan, Cormac & Qadeer (2003, 338-349) who proposed a type system for specifying and verifying the atomicity of methods in Java. When combined with their type checker was able to detect potential atomicity violations during compilation in some of Java's Built-in classes.

The Atomos programming language, derived from the Java programming language, though not explicitly designed to prevent atomicity violations is a prime example of providing language level support for concurrent programming. It does so through the use of language keywords and implicit transactions. When concurrent programs written in Atomos were compared to concurrent programs written in Java, it showed improved performance for parallel programs and easier development. Which is what this research aims to do.

Conclusions & Future Work

The Ava programming language has proven successful in generating atomic sections which avoid single variable atomicity violations, however, only in cases where data dependencies are explicit. In cases where the data dependencies are implicit, a non automatic method is needed. This work, I believe, can be extended to prevent multivariable atomicity violations and I believe should be an area of work in the future. There are several performance optimizations which can be made to the model and its implementation so that the performance of the auto generated code matches that of hand crafted code, developed by experts. I believe however that this represents a significant step toward reducing the complexity in concurrent programs and making it easier to develop them in the future.

References

Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software." Dr. Dobb's journal 30, no. 3 (2005): 202-210.

Breshears, Clay. The art of concurrency: A thread monkey's guide to writing parallel applications. " O'Reilly Media, Inc.", 2009.

Venkatasubramanian, Nalini, Shakuntala Miriyala, and Gul Agha. "Scalable concurrent computing." *Sadhana* 17, no. 1 (1992): 193-220.

Lu, Shan, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. "AVIO: detecting atomicity violations via access interleaving invariants." In *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 37-48. ACM, 2006.

Lu, Shan, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." In ACM Sigplan Notices, vol. 43, no. 3, pp. 329-339. ACM, 2008.

Lu, Shan. *Understanding, detecting and exposing concurrency bugs*. ProQuest, 2008.

Park, Soyeon, Shan Lu, and Yuanyuan Zhou. "CTrigger: exposing atomicity violation bugs from their hiding places." ACM SIGARCH Computer Architecture News. Vol. 37. No. 1. ACM, 2009.

Flanagan, Cormac, and Shaz Qadeer. "A type and effect system for atomicity." In *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 338-349. ACM, 2003.

Engler, Dawson, and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks." In ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 237-252. ACM, 2003.

Vallejo, Enrique, Marco Galluzzi, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenstrom, James E. Smith, and Mateo Valero. "Implementing kilo-instruction multiprocessors."

In *Pervasive Services, 2005. ICPS'05. Proceedings. International Conference on*, pp. 325-336. IEEE, 2005.

Chew, Lee, and David Lie. "Kivati: fast detection and prevention of atomicity violations." In *Proceedings of the 5th European conference on Computer systems*, pp. 307-320. ACM, 2010.

Chen, Qichang, Liqiang Wang, Zijiang Yang, and Scott D. Stoller. "HAVE: detecting atomicity violations via integrated dynamic and static analysis." In