Spring 2015

Inferring Types to Eliminate Ownership Checks in an Intentional JavaScript Compiler

Chris Hebert
University of New Hampshire, Durham

Inferring Types to Eliminate Ownership Checks in an Intentional JavaScript Compiler

BY

Chris Hebert

B.S., University of New Hampshire (2013)

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

May 2015

This thesis has been examined and approved in partial fulfillment for the degree of Master of Science in Computer Science.

Thesis Director, Philip J. Hatcher, Professor of Computer Science

Michel Charpentier, Associate Professor of Computer Science

Wheeler Ruml, Associate Professor of Computer Science

On April 21, 2015

Original approval signatures are on file with the University of New Hampshire Graduate School.

CONTENTS

LIST OF TABLES iv

LIST OF FIGURES v

ABSTRACT vi

1 Introduction **1**

2 Related Work **4**

3 Approach **6**

3.1 Initial Type Inference 7

3.2 Reaching Definitions Dataflow 9

3.3 Iterative Type Inference 13

3.4 Ownership Check Dataflow 15

4 Results and Analysis **17**

4.1 Ownership Check Removal 17

4.2 Type Inferencing 20

5 Conclusions **21**

5.1 Conclusion 21

5.2 Future Work 22

BIBLIOGRAPHY **24**

APPENDIX **25**

LIST OF TABLES

3.1 Reaching definitions iterative dataflow analysis. 12

4.1 Results of our check removal algorithm. The numbers in parentheses represent highest number of checks that could be removed. 17

LIST OF FIGURES

1-1 The Tscript modes.. 2

3-1 A simple example of reaching definitions. 10

3-2 Control flow graph for program in Figure 3-1. 11

3-3 The dataflow equations for reaching definitions. 11

3-4 Pseudo-code for iterative dataflow analysis. 12

3-5 The dataflow equations for ownership checking. 15

5-1 Program to estimate pi. 25

5-2 Program to estimate pi (part 2). 26

5-3 Program to count words in a file. 26

5-4 Program to count words in a file (part 2). 27

5-5 Program to count words in a file (part 3). 28

5-6 Implementation of the Jacobi algorithm for temperature distribution. 28

5-7 Implementation of the Jacobi algorithm for temperature distribution (part 2). . . 29

5-8 Implementation of the Jacobi algorithm for temperature distribution (part 3). . . 30

5-9 Implementation of the Jacobi algorithm for temperature distribution (part 4). . . 31

5-10 Find the twenty longest words in multiple files. 32

5-11 Find the twenty longest words in multiple files (part 2). 33

5-12 Find the twenty longest words in multiple files (part 3). 34

5-13 Find the longest word that appears in each of multiple files. 34

5-14 Find the longest word that appears in each of multiple files (part 2). 35

5-15 Find the longest word that appears in each of multiple files (part 3). 36

ABSTRACT

Inferring Types to Eliminate Ownership Checks in an Intentional JavaScript Compiler

by

Chris Hebert
University of New Hampshire, May, 2015

Concurrent programs are notoriously difficult to develop due to the non-deterministic nature of thread scheduling. It is desirable to have a programming language to make such development easier. Tscript comprises such a system. Tscript is an extension of JavaScript that provides multithreading support along with intent specification. These intents allow a programmer to specify how parts of the program interact in a multithreaded context. However, enforcing intents requires run-time memory checks which can be inefficient. This thesis implements an optimization in the Tscript compiler that seeks to improve this inefficiency through static analysis. Our approach utilizes both type inference and dataflow analysis to eliminate unnecessary run-time checks.

CHAPTER 1

Introduction

Writing correct concurrent programs is often difficult, especially during the testing and debugging phases. This difficulty arises due to non-determinism caused by scheduling of threads within the operating system. Such scheduling can cause race conditions that can be challenging to detect and fix. To solve this issue, we desire a framework to help us detect these bugs. The language Tscript provides such a framework.

Tscript is an intentional concurrent programming language designed as an extension to ECMAScript 5.1 by Charpentier and Hatcher [2014]. The intentional nature of the language means that a programmer must state their intent for how each object can be accessed in a multithreaded situation. Violations of intents are detected dynamically. The goal of intent specification is to make it easier to detect potential concurrency bugs before they occur. These bugs often arise through unintentional violations of intent (e.g. sharing a thread-private object with another thread). The key components of this process in Tscript are the run-time system and the compiler.

The run-time system for Tscript is a set of Java classes that implement the Tscript objects, functions, environments, and so on. The compiler transforms Tscript source code into Java code to be run by the JVM (Java Virtual Machine). The compiler first builds a data structure to store an internal representation of the code. This structure is called an abstract syntax tree, or AST. The AST encodes the semantic information contained in the Tscript source code. The compiler may perform type inferencing and other optimizations on the AST, as well as semantic error checking. After this step, the compiler generates Java code to be executed.

The framework for Tscript involves both memory accesses and modes. Every read or write access of memory by a thread must be checked to ensure that the access conforms to the stated

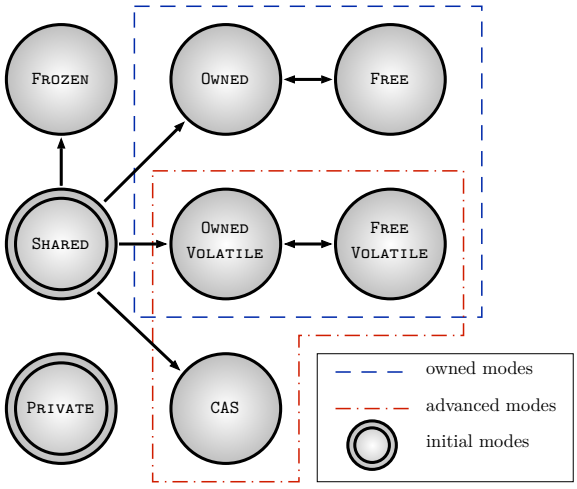

Figure 1-1: The Tscript modes..

intents of the programmer. These intents are specified through modes. Every memory block exists in one of eight access modes. These modes specify how a block of memory can be used at run-time. The modes are Private, Shared, Owned, Free, Owned_Volatile, Free_Volatile, CAS and Frozen, as shown in Figure 1-1. Memory can transition between these modes only in carefully constrained ways according to the semantics of the intent system. Since Tscript is object-based, these memory blocks are objects. Accesses involve reading or writing object properties (e.g. writing a field or calling a method) and mode transitions. Each of these accesses requires a memory check at runtime.

Violations of intent are expressed through the IntentError runtime exception. These violations may occur due to illegal memory accesses, invalid mode transitions, or other multithreading errors. When created, an object is in either Private or Shared mode. Private objects are those intended for only the current thread, whereas Shared ones may be used by multiple threads. Shared objects may only be shared after the owner transitions the object into Frozen mode, for immutable objects, or Owned mode for most other uses (advanced usage might involve a transition to Owned_Volatile or CAS as well). Once in owned mode, an object can transition back and forth between Owned and Free as threads use it. This is analogous to locking: a thread "locks" the object by transitioning it to Owned mode, and unlocks by transitioning back to Free mode. Note that a thread can only lose access to an object by itself taking an action on the object.

As stated before, memory accesses require runtime checks by the thread accessing the object. We call these memory checks *ownership checks*. Ownership checking of all property accesses is potentially expensive. Property accesses happen very frequently in JavaScript (and thus, Tscript) programs, so eliminating unnecessary checks is desirable. In this thesis, we seek to reduce the number of ownership checks through an iterative algorithm leveraging both dataflow analysis and type inferencing.

CHAPTER 2

Related Work

Starkiller by Salib [2004] seeks to improve the performance of Python programs through a combination of static compilation and static type inference. The author felt that such techniques would help address some fundamental obstacles in Python code compilation, such as dynamic inheritance and method-binding. The Starkiller algorithm is a flow-insensitive, concrete type inferencing algorithm that performs whole program analysis. Salib constructs a so-called dataflow network for types consisting of nodes and constraints which seek to simulate the way in which types can change. The success of this work in utilizing static type inference in a very dynamic language like Python demonstrated the feasibility of similar type inferencing in Tscript. Given that Tscript is free from many of the difficulties that plague static compilation in Python, like arbitrary code execution and polymorphism, it seemed likely that we would achieve even greater success.

Cannon [2005] sought to answer a slightly less ambitious question than Starkiller. He wished to discover how utilizing only type inference of atomic types might aid compilation of Python code. The type inferencing algorithm performs its type inferencing on bytecode, rather than on source code. The algorithm uses dataflow analysis to determine possible types for each variable, giving each a set of possible types it can take on. It handles both conditionals and iteration statements, as well as exception-handling. The type inferencing is performed using a type interpreter which sought to mimic the Python interpreter. The interpreter was passed bytecode emitted from the Python interpreter and modified this code to use type-specific bytecodes. This work did not show the same success than Starkiller achieved. The type-specific bytecodes seemed to have little to no benefit, or even net loss, on code performance. The author reached the conclusion that type inferencing seemed infeasible for Python.

JavaScript has historically been a relatively slow, interpreted language. With the advent of the V8 engine by Google, JavaScript began being compiled to machine-specific assembly code [Conrad, 2012]. With this came a major boost in performance. V8 has essentially two compilers: the basic, unoptimized compiler, called full-code-gen, and the optimizing compiler, known as Crankshaft. This optimizing compiler acts as the main focus of our discussion. The optimizations in this compiler are based on similar work by Hölzle [1995] on a SELF compiler.

The SELF compiler goes in a radically different direction than Starkiller. Rather than doing the bulk of the work at compile-time, it saves the heavy optimizations for the runtime system. The compiler by Hölzle uses a technique called adaptive recompilation for this purpose. Adaptive recompilation allows the runtime system to inform the compiler that a section of code should be recompiled. After recompilation, the runtime system reintegrates the optimized code in a process called on-stack replacement. This allows optimized code to take the place of already-running unoptimized code. The other key technique is called type feedback.

Type feedback allows type information from the runtime system to propagate to the compiler. Propagating type information in this way lets the compiler disambiguate operations like dynamic function calls. Before an optimization like inlining can be performed, the compiler must know the precise types of all objects involved in a function call. This introduces another facet of the optimizing SELF compiler, namely, polymorphic inline caching. Since a given function might be called with parameters of different types, and inlining is distinct to one specific definition, they needed a way to store already-inlined function bodies. The polymorphic inline cache serves this purpose. Then, on a cache miss, the compiler can create a function stub that can be inlined as needed. These three components serve as the backbone for the optimizing SELF compiler.

CHAPTER 3

Approach

We present an algorithm to eliminate unneeded ownership checks utilizing both type inference and dataflow analysis.

Before that, however, we briefly elaborate on ownership checks. As we stated earlier, an ownership check must be performed each time a property of an object is accessed or modified. To remove some of these checks, one can seek to "remember" previous checks so that all objects that were previously checked need not be checked again unless a previous check was somehow invalidated. We will discuss the specifics in Section 3.4 on page 15, but it is first important to mention the motivation behind type inference.

Between two given ownership checks on the same object variable by a thread, one of two things may occur that can affect the second one: the variable may be redefined, or the thread may lose access to the object it references. If the object variable is redefined or the object it references takes a new mode locally, it is fairly simple to detect. The difficulty occurs when either case occurs within a function call. Function calls are often far removed from the function bodies that they refer to. This is especially true in a dynamic language like Tscript where a given variable can point to multiple different function bodies at run-time.

Thus, we compute and use reaching definitions (Section 3.2 on page 9) and type inferencing (Section 3.1 on the next page, Section 3.3 on page 13) to try to determine the function bodies referred to by each function call. This gives us more information about the ways in which these function calls affect a given ownership check. To such an end, we present the idea of a "safe" function: one which can have no impact on any ownership checks of an already checked variable.

3.1 Initial Type Inference

Our algorithm begins with an initial type inference step. The purpose is twofold: propagation of variable declarations and population of the symbol table. Before discussing the details, we briefly outline the type system. Every expression in a Tscript program has a type that we assign at compile-time. This type represents the union of all types that the expression could potentially assume. We require this conservative estimate because a given expression may take on different types based on dynamic runtime behavior. The possible types are: Boolean, Number, String, Object, Function, and Unknown. Unknown is considered equivalent to the union of all other types. Though ECMAScript defines an additional type, undefined, our type inference system does not use it. In this work, it provides no additional information beyond Unknown.

Object and Function types are the more interesting types. The Object type represents a mapping from property names to types. Properties in JavaNcript are similar to public methods or fields in an object oriented language. For example, we might have an object p with properties x and y of type Number to represent a point. Interestingly, ECMAScript semantics establishes Function as a sub-type of Object. The Function type has two parts: return type and safety. The return type is self-explanatory. Function safety refers to how calling a function will affect object ownership for this thread. We call a function safe if it calls no unsafe functions and does not modify any variables of outside scope. We discuss it further in Section 3.3 on page 13.

Before the initial type inferencing step, all expressions are assigned the type Unknown. The inferencer then takes effect, traversing down the AST to the leaf nodes. At this level, the AST contains literals (e.g. an integer or string literal) and identifiers (variables). Initially, we know nothing about identifiers, as we do not know what types might be assigned to them later on. For each literal node, we annotate it with its type. These are usually known trivially because the literal itself contains all the information needed to know its type. Object and function literals are the exception. In the case of an object literal, we may not know the types of all the properties. For a function, we do not necessarily know either its safeness or return type. In these cases, the type is filled only with what is known at this time. Later iterations may determine more information.

The traversal of the AST is bottom-up, so we propagate type information from the leaf nodes to the operator nodes. However, in the vast majority of programs, this initial type inference will leave much of the type information unknown. This is due to the identifiers referring to potentially unknown variables. To partially assuage this difficulty, we introduce a symbol table to the type inferencer. The strategy here is to also implement a form of flow-insensitive type inferencing.

The symbol table is a mapping from identifiers to type and scope depth. Scope refers to the lexical scope a given identifier is in. Scope is an integer representation of the lexical scope that lets us track the level of nesting of an identifier. We consider all statements in the so-called main block to be of the highest scope. Each function traversed during type inferencing will increase the depth by one. Thus, a function defined inside a globally-scoped function (a doubly-nested function) will introduce a new scope that is two less than the global scope. We use scope to determine uses of global (or higher-scoped) variables that have important connotations for function safety. Scoping introduces an important feature in the symbol table: each scope gets its own mapping from identifiers to types.

Each time an identifier is assigned to, we have two cases. The first case is a *var* statement. This statement is equivalent to a variable declaration statement in a more traditional language. A *var* statement contains both an identifier (the name being defined) and an optional initialization portion. Here, we add to the symbol table an entry containing the identifier name and the type of the expression which is assigned, or Unknown if no initialization (again, ECMAScript would require undefined here).

In the other case for identifiers, we are assigning to a previously-defined variable. Here, we update the symbol table with the union of the known type of that identifier and the static type of the new expression being assigned to it. Using this approach, we will know after the initial type inferencing all possible values any given identifier can refer to. However, we wish to do better. We wish to determine, at each use of an identifier, only those types which "reach" that use. The types it may have at later points will have no impact on the current type it may assume. We can solve this using reaching definitions, a form of dataflow analysis.

3.2 Reaching Definitions Dataflow

Dataflow analysis is a compiler optimization technique that allows information to "flow" through a whole procedure, or even a whole program (in a limited context). It does this by approximating the runtime behavior of the program at compile-time. Dataflow analysis is an iterative algorithm. It works by solving a set of equations called the dataflow equations. These equations are specific to the type of problem being solved. In our case, this problem is reaching definitions.

Specifically, we wish to determine which definitions for a variable reach any given use of the variable (referred to hence as a "use") in a block of code. In our case, a definition reaches a given use if there is a path from the definition to the use in which the definition is not "killed". A definition is killed if another definition of the variable occurs between itself and a use. The reaching definitions dataflow carries a set of definitions to all uses in the program, with each definition being propagated as far as possible.

To support dataflow the AST is transformed into a control-flow graph (CFG). A CFG is an intermediate representation of the code that lets us reason about it in blocks known as basic blocks. A basic block is a sequence of instructions (AST nodes in our case) which always execute as a single block. Thus, if one statement in a block is executed, all will be executed. The CFG is a graph of basic blocks whose edges model the ways in which execution can flow from one block to another. Each basic block must have one or more incoming edges and one or more outgoing edges, except in the case of the initial or final blocks, which have zero incoming and zero outgoing edges, respectively. Figure 3-1 on the following page shows the source-code for an example Tscript program. The CFG for this program is presented in Figure 3-2 on page 11.

In Figure 3-3 on page 11, we present the dataflow equations for reaching definitions analysis. The set $GEN[b]$ is the set of all definition that have been "generated" in the block b. Each "gen" is also a "kill" in reaching definitions. The reason for this is that any definition will invalidate at this point in the program all previous definitions of the same variable. $KILL[b]$ is the set of all previous definitions which are killed in block b. $REACH_{in}[b]$ and $REACH_{out}[b]$ refer to all definitions that reach when entering and exiting the block, respectively. We calculate $REACH_{in}$ using set union

```
var x = { a: 5, b: 1.0 };
var getPi = function () { return 3.14; };
var f = function () {
    getPi();
    return { g: function () { x = { a: "not a" }; },
             h: function (n) { return n * n; } };
};

x.a += 1;
var anF = f();
console.log(getPi());

while (x.a > 1) {
    var t = anF.h(x.a);
    console.log(t);
    x.a -= 1;
}

console.log(x.a);
x.a = t;
anF.g();
console.log(x.a);
```

Figure 3-1: A simple example of reaching definitions.

because we wish to take all reaching definitions from all previous blocks. Any of these could reach this block, regardless of which predecessor they come from.

These equations are solved using an algorithm called iterative dataflow analysis. We present the pseudo-code in Figure 3-4 on page 12. First, we preprocess each block to compute the GEN and $KILL$ sets, as these do not change once initially computed. Then we repeatedly solve the two $REACH$ equations until we discover nothing new, a state known as equilibrium. To better understand reaching definitions, we will work through the algorithm using our example program. First, we pre-process each block, calculating the GEN and $KILL$ sets. Note that preprocessing does not change any types, but merely computes definitions.

In the first block, we notice four gens, one for each *var* statement. The first is x, for which we create a definition whose type is an Object with two properties a and b whose types are both Num-

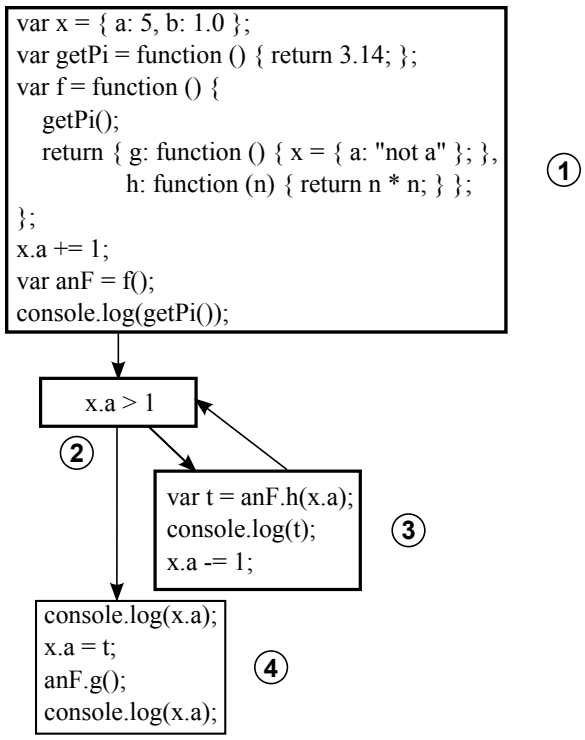

Figure 3-2: Control flow graph for program in Figure 3-1.

$$REACH_{in}[b] = \bigcup_{p \in pred[b]} REACH_{out}[p]$$

$$REACH_{out}[b] = GEN[b] \cup (REACH_{in}[b] - KILL[b])$$

Figure 3-3: The dataflow equations for reaching definitions.

ber. Similarly, $getPi$ defines a safe Function with return type Number. The definition generated for f is also a Function, but one that is unsafe (since it calls an unknown and presumed-unsafe function) and returns an Object containing two function values. For anF, a definition is created with Unknown type. Note that the definition retains a pointer to the node for $f()$, which becomes important later.

Block 2 has no gens, but blocks 3 and 4 do. In block 3, a definition is generated for t whose type is Unknown, but which contains the expression $anF.h(x.a)$. A more interesting case is the definition for $x.a$. For property accesses, we track both the identifier (x) and the property name (a). When it comes time to figure out a type, we look up the property name in all types that

```
N := number of blocks - 1
for i := 0 to N
    preprocessBlock(i)
changed := true
while (changed)
    changed := false
    for i := 0 to N
        recompute ReachIn(i)
        recompute ReachOut(i)
        if ReachOut(i) changed then
            changed := true
```

Figure 3-4: Pseudo-code for iterative dataflow analysis.

reach and union all the types we find. So we create a definition for $x.a$ that is of type Number (the integer literal 1 must have this type). Block 4 contains one gen: another definition mapping $x.a$ to Number, specifically the variable t. Recall that the $KILL$ sets are identical to GEN sets in reaching definitions.

	$REACH_{in}[b]$			
Iteration	b_1	b_2	b_3	b_4
Init	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x, getPi, f, anF, t, x.a_0\}$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_0\}$
4	$\emptyset$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_0\}$
	$REACH_{out}[b]$			
Iteration	b_1	b_2	b_3	b_4
Init	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{x, getPi, f, anF\}$	$\emptyset$	$\{t, x.a_0\}$	$\{x.a_1\}$
2	$\{x, getPi, f, anF\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{t, x.a_0\}$	$\{x.a_1\}$
3	$\{x, getPi, f, anF\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_1\}$
4	$\{x, getPi, f, anF\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_0\}$	$\{x, getPi, f, anF, t, x.a_1\}$

Table 3.1: Reaching definitions iterative dataflow analysis.

We present the results of iterative analysis in tabular form in Table 3.1. To be concise, we represent each definition as the identifier that represents it. Note that each iteration will consider the out sets from the previous iteration. We initially start with all $REACH_{in}$ and $REACH_{out}$ empty. In the first iteration, the GEN sets of each block propagate into their respective out sets.

To calculate $REACH_{out}$, we note that we union the GEN set of the block with its $REACH_{in}$ set, minus everything that is in kill. Since we knew nothing about any $REACH_{in}$ set, we discover nothing more on this iteration.

On the next iteration, we are able to compute $REACH_{in}[2]$ by looking at the $REACH_{out}$ sets of its predecessors (blocks 1 and 3). We discover that all the definitions from blocks 1 and 3 reach block 2. Since block 2 has no $KILL$s, each of these definitions becomes part of $REACH_{out}[2]$. In the third iteration, the gens from block 1 propagate through to blocks 3 and 4 through block 2. The gens in block 3 also get to block 4. Block 4, however, has one killed definition, $x.a$. This kill means that the block 3 definition of $x.a$ will not propagate through to $REACH_{out}[4]$. However, it does reach the beginning of the block.

The important end result of this analysis is that we can determine the types of all function calls in this example. The main reason we figure out reaching definitions is to disambiguate function calls. This, along with the next step in our algorithm, will help us determine which functions are safe.

3.3 Iterative Type Inference

The iterative portion of type inferencing is an extension to the initial step in Section 3.1. The main goal of this part is to determine safety of each function. We determine function safety based on two criteria: function calls and assignments to out of scope variables. For the first case, a function is unsafe if it calls a known unsafe function or an unknown function. Known unsafe functions are those defined in the library that change modes of an object, or ones that call such a function. The other criterion can invalidate checks in a different way. In JavaScript-like languages, variables are accessible within higher-depth environments, becoming global variables in a sense. Thus a function call far from the initial declaration of a variable could change the value referenced by that variable. So if we have checked a variable x at the lowest depth and call a function f which contains a definition $x = 5$, then we cannot carry through the old check on x, as it now refers to a different value than before.

A complication exists with how to handle library functions. Library functions (i.e. those made available through *import* statements) are pre-compiled and loaded into the "global object" as soon as execution begins. Since they are pre-compiled, these functions cannot be handled in the same manner as non-library functions. To handle this, we use the following approach. For each library imported, an AST node is generated. Our type inference sees an import statement node and figures out which library file is needed. From this, we can determine a path to look up a special file we generated called an opt file. This opt file is placed with the compiled libraries and contains a listing of all functions defined in that library. These function names are paired with either "safe" or "unsafe". From this file, we load into the symbol table a new object type with a property for each function. This maps directly to the use of a library in a Tscript program (i.e. "Thread" references an object in the global object). Currently, we create these files by hand, but they could be created while compiling the library. However, the built-in libraries written in Java would still need to be created by hand.

Dataflow analyses like reaching definitions only work within a single function, or intraprocedurally. However, we are seeking to extend our ability to reason beyond this limitation. Connecting function definitions to function calls is a form of interprocedural analysis. This analysis is done in an iterative fashion, since one function can contain a call to a second function. The safety of the first function depends on the safety of the second function which we may not determine until a later iteration. We essentially run the same initial type inference repeatedly. We terminate this once no further type information can be determined.

We reexamine the example Tscript code from Section 3.2 on page 9 after iterative type inferencing. Iterative type inferencing occurs over two iterations. Just after reaching definitions, the function f is presumed unsafe. In the first iteration, we discover that f is safe, since we now see that $getPi$ is safe. Each iteration essentially unrolls one layer of nested function calls. Since the only function with a nested call is f, we determine the safety of all functions after this first iteration. The final iteration merely finishes propagating this safety information through the rest of the program, but finds no changes and terminates.

3.4 Ownership Check Dataflow

Specifically, we wish to determine which ownership checks are "live" at the end of a block of code. In our case, live means that an ownership check has already been performed on a given variable. Then after our analysis, we know at any point which variables have currently been checked.

$$LIVE_{in}[b] = \bigcap_{p \in pred[b]} LIVE_{out}[p]$$
$$LIVE_{out}[b] = GEN[b] \cup (LIVE_{in}[b] - KILL[b])$$

Figure 3-5: The dataflow equations for ownership checking.

In Figure 3-5, we present the equations for our ownership check analysis. The set $GEN[b]$ is the set of all variables that have been checked in the block b. $KILL[b]$ is the set of all variables that may have been redefined in b. $LIVE_{in}[b]$ and $LIVE_{out}[b]$ refer to all variables whose checks are live when entering (or exiting) the block.

This dataflow is a type of forward analysis, meaning we carry ownership checks through the CFG from start to end. We use set conjunction rather than set union when calculating $LIVE_{in}$, as an ownership check must come from all possible branches into the given block in order to be certain the check will have been performed. Recall that our runtime system must perform an ownership check on all property accesses in case the current thread does not have access to the given object. Each time a property access is encountered, we "generate" a check for the variable being accessed. Each time an assignment statement is encountered, we "kill" previous checks to the variable being assigned to. If a check for a variable is "live" when encountering a new property access, we do not need to perform another ownership check. This dataflow analysis will tell us which checks are live after every basic block. We leave this information in the AST to inform code generation.

Continuing our example in Figure 3-1 on page 10, we discuss the results after ownership check removal. The ownership checks performed are highlighted in blue for removed checks and red for performed checks. Note that, of 13 total checks, we were able to remove seven. Note also that

"put" checks, like on line 16, are separate from "get" checks, and one type of check is not equivalent to the other. Lines 9 through 16 demonstrate an important success: the four function calls between the two put checks on x, being known to be safe, do not kill the first check, allowing it to propagate through to the checks on lines 16 and 20. However, since $anF.g$ is an unsafe function, the get checks on *console* and x do not propagate through to line 22.

CHAPTER 4

Results and Analysis

4.1 Ownership Check Removal

	static checks		dynamic checks		expression node count
	get	put	get	put	
	percent removed (of optimal)		percent removed		
pi	12 (3)	0 (0)	112	25	163 nodes
	33%		1%	0%	74% unambiguous
countWords	46 (10)	4 (1)	326000	32000	266 nodes
	100%	100%	49%	$< 1\%$	59% unambiguous
jacobi	77 (20)	12 (3)	37.1 million	2.88 million	600 nodes
	60%	0%	39%	0%	50% unambiguous
LongestWords	57 (25)	4 (0)	257000	60600	319 nodes
	28%	0%	16%	0%	45% unambiguous
LongestWords2	42 (15)	3 (0)	223000	17000	238 nodes
	53%	0%	17%	0%	53% unambiguous

Table 4.1: Results of our check removal algorithm. The numbers in parentheses represent highest number of checks that could be removed.

We measure the results of our algorithm by running it on five Tscript programs written by other group members. We measure performance by comparing the number of checks removed versus the number of checks performed in both static and dynamic contexts. For the static counts, we measure the amount removed as a percentage of the optimal amount that could be removed as determined by manual inspection of the code. By optimal, we mean number of checks that could be removed assuming perfect knowledge of safe functions. We present this optimum number of checks in parentheses in the table. Removing a check statically may result in a much larger reduction

at runtime. We wish to know how many checks our algorithm can remove. Runtime performance is not helpful currently, as the compiler is not designed to output particularly efficient code. The other aspect of the algorithm is the coverage of the type inferencer. To measure this, we counted the number of expression nodes that had unambiguous type. The results for all five benchmark programs are presented in Table 4.1 on the preceding page.

The first benchmark program we use is a Tscript program which computes an estimate for pi. The source code is presented in Figure 5-1 on page 25. It uses a number of Tscript libraries. Our optimizer at compile-time was only able to remove 1 check of 12, a third of optimal. This number matched the number of dynamic get checks with one removed out of 112. The poor performance on this benchmark program is an artifact of the way the pi program is written. It works by first calculating a few values that it needs, such as number of intervals and the interval width for the estimation. This first part generates several checks which we had hoped would carry through the rest of the program. However, the next part creates multiple threads and gives each one a function to run. Because the work is done within a new function, our approach is unable to eliminate the checks performed in this inner function. Moreover, it is not even conceptually possible to bring the old checks into the function, since the code is run by a new thread. Thus the new thread must check each object again to ensure it has access. Additionally, we see few objects used in the program.

The next benchmark, countWords, is a program that counts the occurrences of certain words in a file. We present the source code in Figure 5-3 on page 26. The number of compile-time checks removed was as good as possible though only 10 of 50 get checks were removed. This number amounts to all 100% of optimal removal. The statistics for the runtime checks also showed success. There were 159000 of 326000 get checks removed at runtime, reduced nearly by half. This success can perhaps be explained by two factors. First, much more work was done outside the "work function" of the threads. This allowed more checks to propagate through a larger area of code. The second factor was the stronger reliance on arrays. Multiple checks removed were in iterations over arrays. Thus a single check removed at compile-time turned into thousands at runtime.

There were also four put checks removed of 32039. The problem here is that the put checks are performed on a particular array object inside a loop that our dataflow cannot optimize away. Were

there a put on that object *before* the loop as well, we would have been much more successful.

Third, we optimized jacobi, a Tscript program simulating heat distribution on a two-dimensional plane. The source is listed in Figure 5-6 on page 28. The first thing to note is the frequent use of two-dimensional array accesses. In addition, the amount of work performed in the program is fairly evenly split between the "main" program and the work function performed by each thread. Given the success of the *countWords* benchmark, the outlook for success seems hopeful. At compile-time, our algorithm was able to remove 12 of 89 get checks. This amounts to three fifths of the optimal amount of get checks removed, worse than *countWords* but twice as good as *pi*. The number of runtime checks performed dwarfed all of the other benchmarks, with 37.1 million get checks and 2.88 million put checks. The optimized code removed nearly 40% of the runtime get checks. The runtime behavior was excellent, but the static behavior seems a bit lackluster given that only a fifth were removed. One major reason for this is the use of multi-dimensional arrays. Conceptually, two checks must be performed for each access of both dimensions of the array. However, we can only remove at maximum one of these statically because it is often difficult to track checks on the "inner" array access, even with more type information. A major reason for this is that arrays in JavaScript are fully dynamic and may have their inner types altered at run-time. Secondly, we note that being unable to carry checks into the work function means that there are a certain number of checks which must be performed again, as in *pi*.

The fourth benchmark is LongestWords (Figure 5-10 on page 32), a simple multithreaded program that reports the longest 20 words among multiple files. This benchmark showed the worst performance of all. Only 7 of 57 static get checks were removed, around a quarter of the optimal amount. At runtime, this amounted to 42000 of 257000 checks removed (or 17%). The style of this program is much different from the others, featuring multiply-nested functions, a "constructor"-like function, *newPrioritized*, and multi-dimensional arrays. The *newPrioritized* function actually demonstrates a particular success in the algorithm: we were able to apply knowledge of types gained from an inner function to uses outside that function. The heavy use of function calls is the most likely cause of the poor performance, however, as many of these functions were seen incorrectly as unsafe. These functions were called on objects returned by other library functions. For example,

the reader will note that the method *readLine* returns a String object which itself has methods. However, our algorithm only saw an identifier, say *curLine*, with no known type. Thus a method *curLines.split* was seen as an unknown, unsafe function.

Finally, we examine LongestWords2, which we present in Figure 5-13 on page 34. This program sought to solve the same task as the previous benchmark, except using a ReaderWriterLock. The style is much closer to that of the first three benchmarks, with preparatory work performed before splitting off into multiple worker threads. At compile-time, we see 8 of 45 get checks removed, a bit over half of optimal. At run-time, 17% of 223000 get checks were removed. The problem here is with chains of property accesses (i.e. $a.b.c$). Like with multidimensional arrays, our approach is unable to reason about the inner property ($(a.b).c$). This program featured multiple function calls like that which we had to assume were unsafe.

4.2 Type Inferencing

The coverage of our type inferencer showed greater success than the ownership check removal. On average, more than 50 percent of all expression nodes in the AST had an unambiguous type after type inferencing. There are three main reasons we did not cover more expression nodes. First, we had no knowledge of the types returned by library functions beyond function safety. Second, our approach did not allow us to determine the inner types of arrays and other objects. Third, by not reasoning about function arguments, we were unable to apply type knowledge at call sites to variables inside function bodies.

CHAPTER 5

Conclusions

5.1 Conclusion

We sought to reduce the amount of ownership checks performed using a combination of type inferencing and dataflow analysis. We focused on removing both get and put checks at compile-time. There were 4 to 12 times as many get checks as put checks at runtime, with an even higher ratio of static get checks. Most of the checks our algorithm removed were get checks. At compile-time, we were able to remove around an eighth of the get checks, on average. However our algorithm removed slightly more than half of the get checks which could have been removed given full type knowledge. This translated to around 28 percent of get checks removed at runtime, with the best benchmarks (*jacobi* and *countWords*) removing close to half.

One might ask how many runtime checks are performed in library code versus user code, as the former checks are invisible to our optimizer. The problem is that we have no easy way to make a distinction between the two at runtime. In our experience, the programs with the largest number of checks were those with heavy use of arrays. If certain library routines also use arrays heavily, this might skew the results toward library code performing more checks. Without more experimentation, the answer is not clear.

Our approach struggled with optimizing checks in loops. The problem is that at run-time, we may or may not be able to remove a check inside a loop, and often not at all in the condition expression. Because we did no unfolding of loops, having no check just before a loop meant we would always have to perform the inner check. However, conceptually, the check in either the condition or body would only need to be performed once. With proper handling of loops, such as

lifting an inner check to outside the loop when possible, we could potentially raise our removal rate at run-time much closer to 100%.

The algorithm performed general type inferencing, but we mostly used this for the purpose of function safety. However, though we did not use all of the type information, it certainly seems like it would be helpful for future work in Tscript. The system we designed for storing type information in so-called "opt" files allowed us to bring information about Tscript libraries into our type system. Though we generated these by hand, the system can be extended to generate these files automatically.

5.2 Future Work

Our algorithm showed some success, but there are many extensions to be desired. First, we had some limitations to our handling of property accesses. Our type system was unable to track properties added to objects dynamically. In addition, property access "chains", including multidimensional array accesses, led to the inner types being opaque. Tracking the types of these inner accesses, especially for arrays, might improve ownership check removal significantly. Second, we would like to unfold loops as mentioned above.

Another thing we are missing is a good understanding of how most people write JavaScript and how conducive this style might be to our approach. A study of varying styles might lead to a better understanding of which are easiest to reason about. Given this information, we might better understand how (or if) a programmer can be encouraged to write code in a way that enables the greatest amount of ownership check removal.

As we mentioned earlier, automatic generation of opt files would likely be largely beneficial. As well as tracking function safety, we might also wish to track the types returned by library functions. This could inform user programs and enable further compiler optimizations.

Fourth, we decided to forgo type inferencing of function arguments due to lack of time. Tracking the types of these can help in at least two situations: carrying checks on arguments to function calls into function bodies and providing knowledge of function safety when passing functions as

arguments. Solving this might improve ownership check removal significantly.

Finally, our algorithm only focused on a subset of the checks that the Tscript system performs. There are other checks that are performed even more often than ownership checks, particularly environment checks. Environments are shared pseudo-objects which contain all Tscript objects declared within a particular scope. Each time any object is accessed, a check must be performed on the environment "object". Future work in progress seeks to address these checks.

BIBLIOGRAPHY

Brett Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005.

Michel Charpentier and Phil Hatcher. Intentional concurrent programming. Working paper, 2014.

Jay Conrad. A tour of V8. `http://www.jayconrod.com/posts/51/a-tour-of-v8-full-compiler`, 2012.

Urs Hölzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1995.

Michael Salib. Starkiller: A static type inferencer and compiler for Python. Master's thesis, MIT, 2004.

APPENDIX

```
// £Id: pi.ts 288 2014-11-04 01:41:53Z pjh £

// Computes the value of pi as the area under the curve 4/(1+x^2) on
// the interval [0,1] using numeric integration. Threads are used to
// perform the integration in parallel.
//
// Uses a CountDownLatch to synchronize the threads. And an AtomicNumber
// to protect the integration sum, which is computed by multiple threads.
//
// The number of threads to be used should be provided as the only
// command-line argument.

import edu.unh.cs.tscript.concurrent.CountDownLatch as CountDownLatch;
import edu.unh.cs.tscript.concurrent.AtomicNumber as AtomicNumber;
import edu.unh.cs.tscript.concurrent.Thread as Thread;
import edu.unh.cs.tscript.util.Argv as Argv;

if (Argv.length == 0)
{
  throw "Usage: ts pi.ts numberOfThreads";
}

var numberOfThreads = Number(Argv[0]);
if (numberOfThreads <= 0)
{
  throw "illegal number of threads (" + numberOfThreads + ")";
}

console.log("Using " + numberOfThreads + " threads");

var numberOfIntervals = 50000000;

var widthOfInterval = 1.0 / numberOfIntervals;

// number of intervals per thread
var chunk = numberOfIntervals / numberOfThreads;

// logical thread IDs less than split have one extra interval
var split = numberOfIntervals % numberOfThreads;
if (split == 0)
{
  split = numberOfThreads;
  chunk -= 1;
}


// AtomicNumber for the threads to add their partial sum to
var globalSum = AtomicNumber.newAtomicNumber(0.0);

// latch to coordinate the threads
var latch = CountDownLatch.newCountDownLatch(numberOfThreads);
```

Figure 5-1: Program to estimate pi.

```
// create and start the threads
for (var i = 0; i < numberOfThreads; i++)
{
  Thread.newThread(
    gfunction(id, split, chunk, widthOfInterval, latch, globalSum) {
      var low;              // first interval to be processed
      var high;             // first interval *not* to be processed
      var localSum = 0.0;   // sum for intervals being processed
      var x;                // mid-point of an interval

      if (id < split)
      {
         low = (id * (chunk + 1));
         high = low + (chunk + 1);
      }
      else
      {
         low = (split * (chunk + 1)) + ((id - split) * chunk);
         high = low + chunk;
      }

      x = (low+0.5)*widthOfInterval;
      for (var i = low; i < high; i++)
      {
         localSum += (4.0/(1.0+x*x));
         x += widthOfInterval;
      }

      // update the global sum using the AtomicNumber
      globalSum.addAndGet(localSum);

      // countdown the latch
      latch.countdown();

    },
    i,
    split,
    chunk,
    widthOfInterval,
    latch,
    globalSum
  ).start();
}
latch.await();
console.log("Estimation of pi is " +
  (globalSum.get() * widthOfInterval));
```

Figure 5-2: Program to estimate pi (part 2).

```
// £Id: countWords.ts 287 2014-11-03 03:11:21Z pjh £

// Simple app for testing use of a latch.
//
// Two files are read. The first file is a sequence of words to be
// searched for, one word per line. The second file is then read
// and searched for words in the first file. Words are formed from
// the lines of the second file by "splitting" the lines on spaces.
// The count for each word is reported when EOF is reached
// for the second file.
//
// The filenames of the two files are taken from the commandline.

import edu.unh.cs.tscript.concurrent.CountDownLatch as CountDownLatch;
import edu.unh.cs.tscript.concurrent.Thread as Thread;
import edu.unh.cs.tscript.util.Argv as Argv;
import edu.unh.cs.tscript.io.TextFile as TextFile;

if (Argv.length != 2)
{
  throw "Usage: countWords.ts fileWithSearchWords fileToBeSearched";
}

// index
var i;
```

Figure 5-3: Program to count words in a file.

```
// read the first file and build an array of words to be searched for
var searchWord = new Array(20);
var fp = TextFile.openRead(Argv[0]);
for (i = 0; (searchWord[i] = fp.readLine()) != null; i++)
{
  searchWord[i] = searchWord[i].toLowerCase();
}
fp.close();

var numberOfSearchWords = i;

// will use one thread per search word to do the searching
var numberOfThreads = numberOfSearchWords;

var latch1 = CountDownLatch.newCountDownLatch(numberOfThreads + 1);
var latch2 = CountDownLatch.newCountDownLatch(numberOfThreads + 1);

// create an array for storing the input words
var words = Array.createShared(0);

// create an array to store the counts computed by the threads
var count = Array.create(numberOfThreads);

// create objects, one per thread, and store in the counts array
for (var i = 0; i < numberOfThreads; i++)
{
  count[i] = { SHARED: true, value: 0 };
  count[i].setOwnedMode();
  count[i].release();
}

// create and start the threads
for (i = 0; i < numberOfThreads; i++)
{
  Thread.newThread(
    gfunction(id, searchWord, words, count, latch1, latch2) {

      // get ownership of my count to keep master from peaking at it early
      if (!Object.acquire(count)) throw "can't acquire count (" +id+ ")";

      // wait for the words to be ready
      latch1.countdown();
      latch1.await();

      // search for matches
      for (var i = 0; i < words.length; i++)
      {
        if (words[i] == searchWord) count.value += 1;
      }

      // release ownership of my count
      count.release();

      // countdown on second latch
      latch2.countdown();
    },
    i,
    searchWord[i],
    words,
    count[i],
    latch1,
    latch2).start();
}

// read the second file
fp = TextFile.openRead(Argv[1]);
i = 0;
var line;
while ((line = fp.readLine()) != null)
{
  var w = line.split(" ");
  for (var j = 0; j < w.length; j++)
  {
    words[i] = w[j].toLowerCase();
    i++;
  }
}
fp.close();
```

Figure 5-4: Program to count words in a file (part 2).

```
// freeze the array to be searched
words.setFrozenMode();

// countdown and wait on first latch
latch1.countdown();
latch1.await();

// countdown and wait on second latch
latch2.countdown();
latch2.await();

// run through the count array and display the words and their counts
for (i = 0; i < numberOfSearchWords; i++)
{
  Object.acquire(count[i]);
  console.log(searchWord[i] + ": " + count[i].value);
}
```

Figure 5-5: Program to count words in a file (part 3).

```
// £Id: jacobi.ts 365 2015-04-09 23:35:50Z pjh £

// This program implements the Jacobi algorithm to find the
// steady-state temperature distribution on an insulated
// two-dimensional plate, given constant boundary conditions.
//
// The north, west and east boundaries are set to 0 degrees. The
// south boundary is set to 100 degrees. The interior points are
// initialized to 50 degrees.
//
// The below comments were lifted from the original C code. I
// am currently getting 208 for SIZE = 16, but I do not get
// the expected answer for SIZE = 128, but the original C
// code does not get that answer either now. I am going to
// just let this be for now.
//
// For SIZE = 16, the program should print:
// There are 208 cells cooler than 50.00 degrees.
// For SIZE = 128, the program should print:
// There are 11638 cells cooler than 50.00 degrees.

import edu.unh.cs.tscript.concurrent.Thread as Thread;
import edu.unh.cs.tscript.concurrent.AtomicNumber as AtomicNumber;
import edu.unh.cs.tscript.concurrent.CountDownLatch as CountDownLatch;
import edu.unh.cs.tscript.concurrent.CyclicBarrier as CyclicBarrier;
import edu.unh.cs.tscript.util.Argv as Argv;

if (Argv.length == 0)
{
  throw "Usage: ts jacobi.ts numberOfThreads";
}

var numberOfThreads = Number(Argv[0]);
if (numberOfThreads <= 0)
{
  throw "illegal number of threads (" + numberOfThreads + ")";
}

console.log("Using " + numberOfThreads + " threads");

// constants
var SIZE = 128;
var TEMP =  50.0;
var EPSILON = 0.1;

// loop indices
var i, j;
```

Figure 5-6: Implementation of the Jacobi algorithm for temperature distribution.

```
// create and initialize two 2D arrays
var newGrid = Array.createShared(SIZE);
var oldGrid = Array.createShared(SIZE);
for (i = 0; i < SIZE; i++)
{
  oldGrid[i] = Array.createShared(SIZE);
  newGrid[i] = Array.createShared(SIZE);
  if (i == 0)
  {
    for (j = 0; j < SIZE; j++)
    {
      oldGrid[i][j] = 0.0;
      newGrid[i][j] = 0.0;
    }
  }
  else if (i == (SIZE - 1))
  {
    for (j = 0; j < SIZE; j++)
    {
      oldGrid[i][j] = 100.0;
      newGrid[i][j] = 100.0;
    }
  }
  else
  {
    oldGrid[i][0] = 0.0;
    newGrid[i][0] = 0.0;
    oldGrid[i][SIZE-1] = 0.0;
    newGrid[i][SIZE-1] = 0.0;
    for (j = 1; j < SIZE-1; j++)
    {
      newGrid[i][j] = 50.0;
    }
  }
  newGrid[i].setVolatile();
  newGrid[i].setOwnedMode();
  newGrid[i].release();
  oldGrid[i].setVolatile();
  oldGrid[i].setOwnedMode();
  oldGrid[i].release();
}
newGrid.setFrozenMode();
oldGrid.setFrozenMode();

// map rows to threads
// logical thread IDs less than split have one extra row
// do not distribute the top and bottom rows
var chunk = Math.floor((SIZE - 2) / numberOfThreads);
var split = (SIZE - 2) % numberOfThreads;
if (split == 0)
{
  split = numberOfThreads;
  chunk -= 1;
}

// used to compute the largest change in temp over the grid
var maxError = AtomicNumber.newAtomicNumber(0.0);

// latch for main thread to wait on
var latch = CountDownLatch.newCountDownLatch(numberOfThreads);

// used to control termination
var terminate = Object.createShared(Object.prototype);
terminate.flag = false;
terminate.setVolatile();
terminate.setOwnedMode();
terminate.release();

// for debugging: the number of cells with temp < TEMP
var coolCells = AtomicNumber.newAtomicNumber(0);
```

Figure 5-7: Implementation of the Jacobi algorithm for temperature distribution (part 2).

```
// need to make the barrier inside a function so that maxError
// can be captured in a frozen environment
var makeBarrier = function (maxError, EPSILON, terminate)
{
  // cyclic barrier to synchronize threads at end of each time step
  return CyclicBarrier.newCyclicBarrier(numberOfThreads, function ()
    {
      console.log("maxError is " + maxError.get());
      if (maxError.get() < EPSILON)
      {
        Object.acquire(terminate);
        terminate.flag = true;
      }
      else
      {
        maxError.set(0.0);
      }
    }
  );
};
var barrier = makeBarrier(maxError, EPSILON, terminate);

// create and start the threads
for (var i = 0; i < numberOfThreads; i++)
{
  Thread.newThread(
    gfunction(id, newGrid, oldGrid, maxError, coolCells, barrier, terminate,
      latch, split, chunk, SIZE, TEMP)
    {
      // loop indices
      var i, j;

      // determine which rows are assigned to this thread
      var low, high;
      if (id < split)
      {
         low = (id * (chunk + 1));
         high = low + (chunk + 1);
      }
      else
      {
         low = (split * (chunk + 1)) + ((id - split) * chunk);
         high = low + chunk;
      }
      // need to add 1 because top row is not to be assigned
      low += 1;
      high += 1;

      // need to acquire the rows assigned to this thread
      for (i = low; i < high; i++)
      {
        Object.acquire(oldGrid[i]);
        Object.acquire(newGrid[i]);
      }

      var localTerminate = false;

      // iterate until max difference across plate is below EPSILON
      while (!localTerminate)
      {
        // rotate references to the two 2D arrays
        var tmp;
        tmp = oldGrid;
        oldGrid = newGrid;
        newGrid = tmp;

        var localMaxError = 0.0;
        var change;
```

Figure 5-8: Implementation of the Jacobi algorithm for temperature distribution (part 3).

```
        // process rows assigned to this thread
        for (i = low; i < high; i++)
        {
          for (j = 1; j < SIZE-1; j++)
          {
            newGrid[i][j] = (oldGrid[i-1][j] + oldGrid[i+1][j] +
              oldGrid[i][j+1] + oldGrid[i][j-1]) / 4.0;
            change = oldGrid[i][j]-newGrid[i][j];
            if (change < 0) change = -change;
            if (localMaxError < change) localMaxError = change;
          }
        }

        // contribute local max error to the global max error
        var done = false;
        while (!done)
        {
          done = true;
          var oldMax = maxError.get();
          if (oldMax < localMaxError)
          {
            if (!maxError.compareAndSet(oldMax, localMaxError))
            {
              done = false;
            }
          }
        }
        // wait for all other threads to finish the timestep
        barrier.await();

        // now check termination condition
        localTerminate = terminate.flag;
      }

      // now compute the number of local cool cells and update
      // the global count
      var localCount = 0;
      for (i = low; i < high; i++)
      {
        for (j = 1; j < SIZE-1; j++)
        {
          if (newGrid[i][j] < TEMP) localCount += 1;
        }
      }
      coolCells.addAndGet(localCount);

      // signal the main thread
      latch.countdown();

    },
    i,
    newGrid,
    oldGrid,
    maxError,
    coolCells,
    barrier,
    terminate,
    latch,
    split,
    chunk,
    SIZE,
    TEMP
  ).start();
}

// need to block for threads to finish
latch.await();

// need to add the cool cells in the north, east and west boundaries
console.log("There are " + (coolCells.get() + SIZE + SIZE + SIZE - 2) +
  " cells cooler than " + TEMP + " degrees");
```

Figure 5-9: Implementation of the Jacobi algorithm for temperature distribution (part 4).

```
// £Id£
// This is a multithreaded application derived from a CS520 assignment.
// The program will accept a sequence of names of English text files as its
// only command-line arguments. The goal of the program is to find the twenty
// longest words that appear in every file. (That is, to be reported a word
// must appear in each file and it must be one of the twenty longest of such
// words.) If there are ties for position twenty on the list of longest words,
// then it reports all the words that tie. So, it may report more than twenty
// words, because of ties for the last position on the list. It might report
// fewer than twenty words if there are fewer than twenty words that are
// common to all files. Also note that words less than eight characters in
// length will be ignored. Also, since the input files are in English, it
// will assume that no word will be longer that fifty characters, since the
// longest word in the English language is only 45 letters long. Words longer
// than fifty characters are ignored, since they are probably not real
// words. Therefore, the program will actually report the twenty longest
// words that appear in every file and that are at least eight characters
// long and no more than fifty characters long.
//
// If the user does not specify at least one file to be processed, the
// program terminates with an appropriate message.
//
// A word starts with a letter (either uppercase or lowercase) and continues
// until a non-letter (or EOF) is encountered. Non-words in the file will
// simply be ignored. Once a word is identified, all uppercase letters are
// converted to lowercase before it is processed.
//
// Therefore, "elephant's" will be two words, "elephant" and "s", and since
// "s" is less than eight characters long, it will be ignored. Likewise,
// "double-precision" will be two words, "double" and "precision", and since
// "double" is less than eight characters long, it will be ignored.
//
// The output will be unsorted, will be printed to stdout, and will consist
// of one word per line.
//
// The main thread reads the first file and builds a data structure to
// represent the words seen in that file. Then a set of threads are
// created to read and process the remaining files, with each thread
// processing one particular file. The threads update the data structure
// created by the first thread. When they are done, then the main thread
// prints the twenty longest words found in all files.

import edu.unh.cs.tscript.concurrent.Thread as Thread;

import edu.unh.cs.tscript.io.TextFile as TextFile;

import edu.unh.cs.tscript.util.Argv as Argv;

import edu.unh.cs.tscript.util.PriorityQueue as PriorityQueue;

import edu.unh.cs.tscript.concurrent.AtomicNumber as AtomicNumber;

import edu.unh.cs.tscript.concurrent.Executor as Executor;

var numFiles = Argv.length;
var reader = { SHARED: true };
var i = 0;
while(i < numFiles){
    reader[i] = TextFile.openRead(Argv[i]);
    i = i + 1;
}

// The 'Prioritized' Object that will be used in testing.
var newPrioritized = function(x, p){
    var state = {
        SHARED: true,
        value: x,
        priority: p
    };

    // Returns the priority
    var getPriority = function(){
        return state.priority;
    };
```

Figure 5-10: Find the twenty longest words in multiple files.

```
    // Returns the value
    var getValue = function(){
        return state.value;
    };

    // Returns the value
    var toString = function(){
        return "(v:" + getValue() + ",p:" + getPriority() + ")";
    };

    return {
        getPriority: getPriority,
        getValue:getValue,
        toString: toString
    };
};

var runApp = function(){
    var wordQueue = PriorityQueue.newPrivateQueue();
    var wordArray = { SHARED: true };
    var curLine = reader[0].readLine();
    while(curLine != null){
        var words = curLine.split("[^a-zA-Z]+");
        var i = 0;
        while(i < words.length){
            words[i] = words[i].toLowerCase();
            if(words[i].length >= 6 && wordArray[words[i]] == undefined){
                wordQueue.add(newPrioritized(words[i], words[i].length));
                wordArray[words[i]] = AtomicNumber.newAtomicNumber(1);
            }
            i = i + 1;
        }
        curLine = reader[0].readLine();
    }
    wordArray.setFrozenMode();

    var executor = Executor.newExecutor(numFiles);

    var makeTask = gfunction(wa, textFile) {
        return function() {
            var curWordArray = { SHARED: true };
            var curLine = textFile.readLine();
            while(curLine != null){
                var words = curLine.split("[^a-zA-Z]+");
                var i = 0;
                while(i < words.length){
                    words[i] = words[i].toLowerCase();
                    if(words[i].length >= 6){
                        if(wa[words[i]] != undefined && curWordArray[words[i]] == undefined){
                            wa[words[i]].getAndIncrement();
                            curWordArray[words[i]] = 1;
                        }
                    }
                    i = i + 1;
                }
                curLine = textFile.readLine();
            }
        };
    };

    var args = {PRIVATE: true};
    var i = 1;
    while(i < numFiles){
        executor.submit(makeTask(wordArray, reader[i]));
        i = i + 1;
    }
    executor.shutdown();
    executor.awaitTermination();
```

Figure 5-11: Find the twenty longest words in multiple files (part 2).

```
    var numWords = 0;
    var curWord = wordQueue.poll().getValue();
    var prevLength = curWord.length;
    i = 0;
    while(numWords < 20 || prevLength == curWord.length){
        if(wordArray[curWord].get() == numFiles){
            numWords = numWords  + 1;
            console.log(curWord);
        }
        prevLength = curWord.length;
        curWord = wordQueue.poll().getValue();
    }
};

runApp();
```

Figure 5-12: Find the twenty longest words in multiple files (part 3).

```
// £Id: LongestWords2.ts 293 2014-11-14 18:30:05Z pjh £
//
// The names of text files are given on the command line. The goal
// is to find the longest word that appears at least once in each file.
// All words of the longest length that appear in all files are reported.
//
// A word starts with a letter (either uppercase or lowercase) and continues
// until a non-letter (or EOF) is encountered. Non-words in the file should
// simply be ignored. Once a word is identified, convert all uppercase letters
// to lowercase before you process it. Words with a length less than 8
// or greater than 50 are ignored.
//
// A thread is created for each file. One object is allocated and
// shared by all threads to track the words. Each word found in some
// file is inserted into object, with the word being used as a property
// name. The value associated with the property is a bit vector, one
// bit per file. If the word has been seen in a file, the file's bit will
// be set to 1; otherwise the bit is 0.
//
// This app is intended to test the ReaderWriterLock. Threads will get
// a reader lock to see if they have already seen a word. If not, then
// they get a writers lock to set their bit (and perhaps also create
// the property if the word has not been seen by any thread yet).

import edu.unh.cs.tscript.util.Argv as Argv;
import edu.unh.cs.tscript.concurrent.Thread as Thread;
import edu.unh.cs.tscript.io.TextFile as TextFile;
import edu.unh.cs.tscript.concurrent.CountDownLatch;
import edu.unh.cs.tscript.concurrent.ReentrantReadWriteLock;

var numFiles = Argv.length;

if (numFiles < 1) throw "Usage: countWords2.ts file1 file2 ...";

// using bit vectors with one bit per file
// Number (i.e. a double) can only exactly represent integers up to 53 bits
if (numFiles > 50) throw "can only handle up to 50 files";

// open each file
var fp = { SHARED: true };
var i = 0;
while(i < numFiles){
    fp[i] = TextFile.openRead(Argv[i]);
    i = i + 1;
}
```

Figure 5-13: Find the longest word that appears in each of multiple files.

```
// create an object to be used to collect the counts
var dictionary = { SHARED: true };
dictionary.setOwnedMode();

// create a latch to coordinate the threads
var latch = CountDownLatch.newCountDownLatch(numFiles);

// create a reader-writer lock to protect the counts
// newLock returns an object with two properties
//   lock: the lock itself
//   proxy: the proxy for the protected object
// the lock itself is another object that contains two properties
//   readLock: the lock for reading
//   writeLock: the lock for writing
var pair = ReentrantReadWriteLock.newLock(dictionary);
var rwlock = pair.lock;
var proxy = pair.proxy;

// create and start the threads
for (var i = 0; i < numFiles; i++)
{
  Thread.newThread(
    gfunction(id, fp, rwlock, dictionary, latch) {
      var line;
      while ((line = fp.readLine()) != null)
      {
        var w = line.toLowerCase().split("[^a-z]");
        for (var i = 0; i < w.length; i++)
        {
          var word = w[i];
          //console.log(id + ": " + word + " " + (word.length));
          if (word.length < 8 || word.length > 50) continue;

          // first grab read lock
          rwlock.readLock.lock();
          //console.log(id + ": got read lock");
          if (dictionary[word] == undefined ||
              !(dictionary[word] & (1 << id)))
          {
            // release reader lock and grab writer lock
            // someone could write to my word before I get the writer lock
            rwlock.readLock.unlock();
            //console.log(id + ": released read lock");
            rwlock.writeLock.lock();
            //console.log(id + ": got write lock");

            // insert the word, if necessary
            // and mark my file's bit
            if (dictionary[word] == undefined)
            {
              dictionary[word] = (1 << id);
            }
            else
            {
              dictionary[word] |= (1 << id);
            }

            // release writer lock
            rwlock.writeLock.unlock();
            //console.log(id + ": released write lock");
          }
          else
          {
            // release reader lock
            rwlock.readLock.unlock();
            //console.log(id + ": released read lock");
          }
        }
      }
      //console.log(id + ": finished");
      latch.countdown();
```

Figure 5-14: Find the longest word that appears in each of multiple files (part 2).

```
    },
    i,
    fp[i],
    rwlock,
    proxy,  // send the proxy, not the real object
    latch
  ).start();
}

// wait for the threads to finish
latch.await();

// get ownership of dictionary
// it must be available or it's an error
if (!Object.acquire(dictionary))
{
  throw "dictionary not available";
}

// find the longest word(s) that were seen in every file
var longestLength = 0;
var longestWords;
for (var w in dictionary)
{
  if (dictionary[w] == ((1 << numFiles) - 1))
  {
    //console.log(w + ": " + w.length);
    if (w.length > longestLength)
    {
      //console.log("new winner");
      longestWords = Object.create(Object.prototype);
      longestLength = w.length;
    }
    if (w.length == longestLength)
    {
      //console.log("added");
      longestWords[w] = w;
    }
  }
}

for (var w2 in longestWords)
{
  console.log(w2);
}
```

Figure 5-15: Find the longest word that appears in each of multiple files (part 3).