

**POSSIBILITY SET SEMANTICS: ERROR DETECTION
AND CODE OPTIMIZATION**

By

BRIAN JOSEPH SULLIVAN

Bachelor of Science
Oklahoma State University
Stillwater, Oklahoma
1993

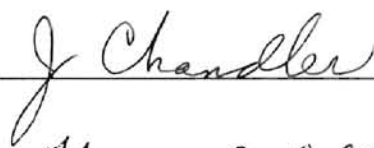
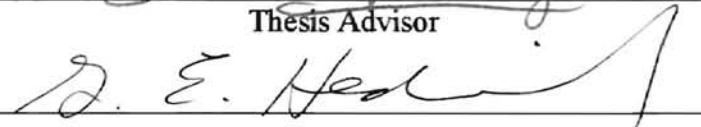
Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1996

POSSIBILITY SET SEMANTICS: ERROR DETECTION
AND CODE OPTIMIZATION

Thesis Approved:



Thesis Advisor



Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to thank my advisor Dr. K. M. George for his assistance in constructing this thesis. Also, I would like to thank my other committee members Dr. Hedrick and Dr. Chandler.

In addition, I would like to express my gratitude to my family and friends for their support during the creation of this thesis.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	REVIEW OF THE LITERATURE	3
2.1	Errors	3
2.1.1	Lexical Errors	4
2.1.2	Syntax Errors	5
2.1.2.1	Panic Mode	5
2.1.2.2	Phrase-Level Recovery	5
2.1.2.3	Error Productions	6
2.1.2.4	Global Correction	7
2.1.2.5	Recovery Without Correction	7
2.1.3	Semantic Errors	8
2.1.3.1	Type Checking	8
2.1.3.2	Data Flow Inconsistencies	9
2.1.3.3	Lint	9
2.1.4	Semantic Errors Resulting in Exceptions	10
2.1.4.1	Current Treatment of Exceptions	10
2.1.4.2	Treatment of Exceptions in this Thesis	11
2.1.5	Preserving Error Producing Behavior	12
2.2	Code Optimization	13
2.2.1	Constant Propagation	13
2.2.2	Uninitialized Variable Access	15
2.2.3	Range Checking	16
III.	POSSIBILITY SET SEMANTICS	19
3.1	Notation for Possibility Set Semantics	19
3.2	Virtual Machine Notation	21
3.3	Basic statements	22
3.3.1	begin_procedure	22
3.3.2	end_procedure	23
3.3.3	declare x of type T	24
3.3.4	kill x	25

3.3.5	undeclare v	25
3.3.6	$y \leftarrow x$	26
3.3.7	cast source x to destination y	26
3.4	Arithmetic Instructions	27
3.4.1	$x \leftarrow x + y$	28
3.4.2	$x \leftarrow x + y$ on overflow goto z	28
3.4.3	$y \leftarrow y / x$ on divzero goto z	30
3.4.4	bound low w, check x, high y on error label z	30
3.5	Comparisons	32
3.5.1	$b \leftarrow (x < y)$	33
3.5.2	$b \leftarrow x$ and y	35
3.5.3	if/else/endif	39
3.6	Branching	41
3.6.1	Forward jump: goto S	41
3.7	Looping	43
3.7.1	Backward jump: goto Z	44
IV.	USAGE OF POSSIBILITY SET SEMANTICS	47
4.1	Interprocedural Analysis and Globals	47
4.2	Error Detection and Code Optimization	49
4.2.1	Error Detection	49
4.2.1.1	Division by zero	49
4.2.1.2	Overflow	50
4.2.1.3	Invalid array index	50
4.2.1.4	Invalid parameter to function	50
4.2.1.5	Possible loss of information	50
4.2.1.6	Reference to uninitialized variable	51
4.3	Code Optimization	52
4.4	Practicality	53
4.4.1	Determination of Definite Errors in Loop Analysis	53
4.4.2	Early Termination of Loop Analysis by Expanding Possibilities	54
4.4.3	Conservation of Space	55
4.4.4	Limit to Complexity	55
V	CONCLUSION	57
5.1	Future Work	57
5.2	Conclusions	58
REFERENCES		59

APPENDIX A-- GLOSSARY	62
APPENDIX B-- ALGORITHMS IN C AND PSS ANALYSIS	64

LIST OF TABLES AND FIGURES

Table	Page
1. PL/I Error Conditions	11

Figure	Page
1. Example of <code>end_procedure</code> from C	24
2. Example of assignment using <code>cast</code> from C	27
3. Example of greater-than-or-equal (<code>>=</code>) from C	35
4. Example of logical <i>and</i> where x and y are distinct.	37
5. Example of logical <i>and</i> where x and y are the same variable.	38
6. Example of a forward <code>goto</code> from C	43
7. Example of prototype from C-like language	48

CHAPTER I

INTRODUCTION

Traditionally, error detection by the compiler occurs mainly at the lexical and syntactic levels. Lexical and syntactic analysis have been investigated thoroughly and have well established means of determining errors (see section 2.1). Just as the syntactic level builds upon the foundation of lexical detection and requires more analysis, the semantic level also builds upon the syntactic level and requires yet more analysis (section 2.1.3).

The semantic level has a larger area to cover as each language seems to have more diverse semantics. Some languages, such as PL/I and Ada, may allow the user to trap run-time errors with *ON ERROR* statements or exception handling, but the compiler gives no notice of where those errors may occur at compile-time other than in very obvious cases. Utilities such as lint more rigorously apply type checking of parameters and results as well as indicate the portability of the code [Darw91]. Safety checks must be placed around each possible offender to control run-time errors. The compiler provides no notification that an error may occur at such points. Such possible points of run-time errors are too numerous to be worth mentioning by the compiler as every division by a variable and every array indexed by a variable are candidates for compiler warnings.

This thesis addresses the problem of statically detecting the possible locations of run-time errors. The techniques presented in the thesis mimic the method used by some

human programmers when analyzing code. A procedural language's source code, or more typically its intermediate code, is analyzed using alternate semantics in which each variable represents not a value but rather a set of possible values. To be practical the compiler overdetects possible errors and to be conservative it does not optimize everything that could be optimized if all information were retained. But the technique flags all places where certain run-time errors such as division by zero, array bound violations, arithmetic over/underflows and uninitialized variables may occur.

Some optimizations are made which otherwise cannot occur. Checks for run-time errors which may be detected by this analysis can be eliminated wherever the analysis determines that the run-time errors cannot occur. A different method of constant propagation, more appropriately constant replacement, is presented. As the possible values of expressions are known, some expressions may be downgraded in byte length.

CHAPTER II

REVIEW OF THE LITERATURE

Current literature discusses error detection of all types in compilers at all levels. The literature contains much information on the well-defined areas, but the less well-defined areas merely indicative of errors also are discussed. Some general code optimizations are discussed as found in the literature.

2.1 Errors

Errors occur at the lexical level by the insertion and/or deletion of characters leading to strings that do not match any of the lexical analyzer's valid tokens. Syntactic errors may occur through flaws in the programmer's grammatical constructions as well as through the causes above. Semantic errors have valid syntax, but invalid meaning; the extent of information included in a language's grammar separates the syntactic and semantic levels. Logical errors are beyond the scope of this thesis.

2.1.1 Lexical Errors

Error detection at the purely lexical level is rather simple. The lexical analyzer uses regular expressions and a table in order to determine whether the source input is a valid token. The token may not be valid at higher levels, but the determination of the existing token is straightforward. When no available expression or table entry describes the string encountered, there are several methods of recovery available.

First, the compiler may simply stop upon receiving an invalid token. This is certainly the easiest action, but for such a simple error the compiler can generally continue to provide the programmer more information.

Second, the compiler may insert or delete characters, replace incorrect characters by correct characters, or exchange adjacent characters [Aho88]. If the compiler inserts characters into the input stream to correct the input, it should take care not to insert characters indefinitely thus creating an infinite loop. As the number of possibly valid characters is limited, each character "is examined in sequence against a list of all possible characters. During this examination a new list of all possible next characters is built. When the end of the current list is reached, the new list becomes the current list, the character is obtained, and the process continues" [Thom68]. All such modification should only be for the purpose of continuing the compilation; the compiler should notify the programmer of such changes and they should be fixed by the programmer before compilation should produce executable code.

2.1.2 Syntax Errors

Error detection at the syntactic level is more involved, especially if meaningful error messages are to be produced and if the compiler is to continue compiling after detecting the error. As an example in LR(1) parsers, the parsing table and stack are used in the detection of errors; so long as the parser may proceed using given productions all is well, but if there is no match then the parser enters into an error state [Geor85].

Upon the detection of an error, there are several methods of recovery available.

2.1.2.1 Panic Mode

If the parser discovers a syntax error, *panic mode* recovery deletes tokens from its consideration until some form of synchronization is possible. This recovery "attempts to move forward in the input stream far enough that the new input is not adversely affected by the older input" [Levi92]. In many languages, synchronization may occur when an *end* token is reached to pair against an already recognized *begin* token [Aho88]. The token to regain synchronization varies with the language and the context. A left parenthesis may synchronize with a right parenthesis, or a statement may regain synchronization upon reading an end of statement marker such as ';'. If synchronization is not achieved correctly, errors can cascade [Levi92]. This technique never enters an infinite loop, but it does not give the programmer the most information possible as all possible error information between points of synchronization is lost [Aho88].

2.1.2.2 Phrase-Level Recovery

Phrase-level recovery attempts a local recovery by using simple heuristics determined by the compiler writer to "correct" the code allowing the compiler to continue.

It isolates "an 'error phrase', which is then replaced by a suitable 'reduction goal'" [Sipp83]. It may use the remaining input as the base upon which it can concatenate another string making the remaining input valid. It may replace similar characters such as a comma or colon by a semicolon, or delete or insert end of statement markers [Aho88]. This form of error correction must be ensured not to enter an infinite loop by always inserting another character in the input. Its greatest weakness lies in dealing with the situation in which the error occurred before the point of detection [Aho88].

2.1.2.3 Error Productions

Error productions involve the foresight of the compiler writer in noting those errors of construction which will be common in the language. Just as the compiler has productions for comparisons, assignments, etc., it may have productions for constructs which produce errors [Aho88]. These productions may actually produce code corresponding to what the programmer wanted and simply warn the programmer that the construct is invalid or it may just give a more explicit error message. If it were known that people moving from Pascal to C often used Pascal style pointers, the appropriate productions may be included, automatically replacing the error with its corresponding C construct and giving an error message notifying the programmer of the correct usage. This often seems to be done with ANSI C compilers upon encountering K&R C constructs. Some systems such as those proposed by Sippu and Soisalon-Soininen would create error productions based upon the findings of the stack in phrase level recovery [Sipp83].

2.1.2.4 Global Correction

Global correction attempts to use an algorithm "to find the minimal sequence of changes to obtain a globally least-cost correction" [Aho88]. Given that there is an error in the source code and the low likelihood of the correction being that which the programmer actually intended, the high cost of such algorithms prevents their general use [Aho88].

Backhouse discusses locally least-cost error recovery for LL(1) parsers [Back84]. The cost of replacing terminal symbols and determining the min and max follow costs are discussed therein. The scheme presented uses a parameter table rather than searching for the follow set; it tries to decrease the size of the parameter table through various techniques including the definition of equivalence classes for terminals. Unfortunately, its definition of equivalence classes is not solvable in general for LR(1) parsers [Back84].

Traviolia discusses a new syntax-error recovery scheme for LR(k) parsers called Least-Cost LR(k) Early's Algorithm [Trav91]. Traviolia's algorithm is $O(n)$ for correct input, causing little concern for correct compilation, and it is effective though potentially costly, $O(n^5)$, in the worst-case [Trav91]. This worst-case bound is still better than some previous algorithms [Trav91].

2.1.2.5 Recovery Without Correction

Recovery from syntactic errors without an attempt at correction is another method, one promoted by Richter [Rich85], in which a normal LR parser would process the source code language until an error is reached. At that point, a suffix parser is then invoked which may determine intervals in which errors may occur. This

method of detection does not necessarily detect all errors, most notably it only detects the innermost error in an interval thus possibly skipping mismatched parenthesis errors.

2.1.3 Semantic Errors

Semantic errors are errors in meaning. The programmer creates a program which may be syntactically correct yet does not perform the desired operation. As an example, a compiler cannot generally detect that the desired function of a program is to sort input when the programmer actually creates a program to play Othello. A compiler may respect assertions, and a programmer may enter preconditions and post-conditions for a block of code [Marc86]. A programmer can work backwards from the desired result finding the necessary condition to ensure its evaluation, repeatedly applying such steps until the precondition is derived [Babe87]. “The semantics of a programming language can be defined via a hypothetical machine which interprets the programs of that language”[Bjoe82]; a modification of such semantics for a language leads to the results contained herein. Yet if such assertions and any other extra materials match the program, and still the program does not match the programmer's intention, nothing can detect the error. When the program and its associated materials is consistent and still incorrect, there is no handle for the compiler to discover that the program is incorrectly specified.

Yet a compiler can detect certain inconsistencies in the program.

2.1.3.1 Type Checking

The type checking component of analysis is concerned with determining whether a program conforms to the type system. This may apply to the built-in operations such as

addition and subtraction and other programmer defined types and operations. Type errors are improper usages that violate the rules defining the type system of the language. For example, type mismatch of the left-hand side and the right-hand side of an assignment statement may be a type error in a language.

Type errors may be considered syntax errors if the language defines the type structure at the syntactic level [Wijn76], or it may be considered to be at the semantic level if the language syntax does not invoke type. Algol 68 includes sufficient information within its language definition to determine that types are syntactic in nature [Wijn76], whereas in Lisp an addition of two members of a list may or may not have valid meaning; it may have meaning during one pass of a loop and lose its meaning during the next pass.

2.1.3.2 Data Flow Inconsistencies

Systems such as Dave, developed and described by Fosdick and Osterweil, try to determine inconsistencies in data-flow [Fosd76]. Dave finds definitions, references, and undefinitions (the invalidation of data). A definition followed by another definition is an inconsistency in that the first definition would not be used. Similarly, a definition followed by an undefinition would make the definition useless. And naturally an undefinition followed by a reference would mean that the reference would be invalid.

2.1.3.3 Lint

Lint, a utility provided on UNIX systems, provides another form of error checking. Lint tries to determine some common coding problems and tries to determine if the code has the same semantics on other machines, its portability [Darw91]. Again, the checks made by the program fall under the category of inconsistency. It can determine if

arguments passed to functions are not always the same. It can determine if a function's returned value is used in some instances, but ignored in other instances.

2.1.4 Semantic Errors Resulting in Exceptions

Semantic errors as referred to in this section describe those errors made by the programmer which would generally not be detectable at compile time, but rather they occur at run-time as run-time errors or exceptions. The most general case of those errors which could be detected by the techniques presented are those in which functions are applied to invalid values. This includes functions such as division where the divisor is zero, the array indexing functions with invalid indexes, addition and other arithmetic operators with values inducing over/underflow, and certain inconsistencies of programmers' code when interacting with language supplied functions and specially annotated user functions.

2.1.4.1 Current Treatment of Exceptions

Currently, the two most common approaches to these forms of errors are letting the error occur and allowing the operating system to attempt a recovery or allowing the programmer or compiler to designate exception handlers [Rich85]. No notice is given to the programmer of the places that these exceptions would be needed even when such exceptions are supported by the language.

The PL/I language has many separate ON conditions which would catch different types of run-time errors (see Table 1 below). Some conditions in PL/I are always active and some are active unless disabled and some are active only when enabled. The

condition may even be signaled directly without the actual cause of the condition occurring. The subject of the ON condition may continue to the next statement or halt the entire program if no ON condition is supplied, depending on the condition [Lech68].

Table 1. PL/I Error Conditions [Lech68]

CONVERSION	Illegal conversion of character string data.
FIXEDOVERFLOW	Result of fixed point arithmetic exceeds places.
OVERFLOW	Value of float's exponent exceeds 2^{127} .
SIZE	Assignment of a value too large for the variable.
UNDERFLOW	Value of float's exponent smaller than 2^{-128} .
ZERODIVIDE	Division by zero in fixed or floating point.
SUBSCRIPTRANGE	An index to an array falls out of declared bounds.

2.1.4.2 Treatment of Exceptions in this Thesis

This thesis shows that by maintaining a set of possible values for each expression that the compiler may discover those places where certain run-time errors cannot occur. By knowing where they definitely will not occur, the compiler may forego the placement of checks surrounding the otherwise suspect code.

The reduction in possible places for run-time errors makes the location of remaining possible exception points valuable to the programmer. The programmer could then make the decision to have the compiler eliminate automatic exception points, instead

placing the exceptions only in the points with possible meaning to the program. If the program were free from such possible warnings and errors, then the programmer could be assured that in a reliable system the program could not suffer from the run-time errors discussed in this thesis. As discussed within this thesis, a reliable system is one in which all effects on the program may be reliably predicted given the information within the program and correct information regarding its environment. For example, if the possible return values from the operating system were found to be incorrect or if another program could overwrite the execution space of the program in question then the system would not be "reliable". Such knowledge would be extremely valuable to those programming for mission critical applications.

2.1.5 Preserving Error Producing Behavior

If the error-producing behavior of a program needs to be maintained, then a greater amount of information retained may help with optimizations. Normally, a compiler is only concerned with reproducing the behavior of a correct program. If a program crashes with an error, compilers typically allow differing behavior. If the behavior of program which crashes were to be preserved, optimizations which rely upon any form of code motion would normally be disabled. That is, if the unoptimized code executes a then b , and a and b each would cause a different run-time error, then the positions of a and b may not be altered. Optimizing the code could execute b before a therefore creating a different run-time error, causing the error producing behavior of the program to change.

Code motion is disallowed across warnings discovered by this thesis if the error-producing behavior must be preserved.

If the run-time error producing portion of the code must be maintained, then it eliminates many possible optimizations. Portions of the program may be marked as safe in that they would be known not to cause run-time errors. Code between warnings generated by this thesis can be marked safe. Any portion so marked could then be fully optimized. Although Aiken et al. speak of marking code as safe in functional languages, by knowing where an error cannot occur, benefits may be achieved in procedural languages as well [Aike95].

2.2 Code Optimization

Code optimization in compiling actually refers to improving the time performance of code. If all statements from a high level language were to be translated as directly as possible to a lower-level intermediate or assembler language, the produced code would generally be of poor quality compared to what a human could code by hand. Code optimization attempts to bring the produced code closer to the code a competent person could produce by hand.

2.2.1 Constant Propagation

One of the most common and useful code optimization techniques is that of constant folding and propagation. In constant folding, when all the inputs of an

expression are constant, then the expression is computed at compile-time and directly replaced by the constant [Clic95]. In constant propagation, the definition of a variable with a constant value allows its uses to be replaced by the constant.

Static Single-Assignment form allows each assignment to have its own name allowing the compiler to find more easily the occurrence of each value [Bran94]. Then if one occurrence of the name has a constant value, all occurrences have the same constant value.

Constants may be propagated even across procedures, although such interprocedural propagation is not very common [Metz93].

Aho, Sethi and Ullman show constant propagation in terms of dataflow analysis [Aho88]. The values, transfer functions, meet operation, etc. for constant propagation are shown. Simple modifications of constants are allowed as in "when x is defined by d : $x:=x+1$, and x had a constant value before assignment, it does so afterward" [Aho88]. Yet it reduces all sets of values to the single special value, *nonconst*. "The value *nonconst* would be assigned to variable x if, say, during data-flow analysis we discovered two paths along which the values 2 and 3, respectively, were assigned to x , or a path along which the previous definition of x was a read statement" [Aho88]. The values 2 and 3 would be irrevocably lost, subsumed into the special *nonconst*. If a later control construct split the flow with a statement such as:

```
if(x≠2) output(x); else output(10-x);
```

then both opportunities for constant replacement would be lost.

Constant propagation within the framework of this thesis will be done quite differently. As each variable would have a set of possible values, it follows directly that if there is only one possible value then that value may replace the original reference. This form of replacement does not depend on the existence of a constant's assignment to a variable at any point. As many values in the sets as practical would be retained by the compiler. Partitioning of the set by comparisons could occur as in the following, assuming x is an integer:

```
if((x>9) && (x<11)) output(x);
```

Assuming ten (10) is ever a possibility, the output if any would certainly be 10, yet the constant assignment of 10 will not be made using techniques found in the literature. (If x could not be 10, then the code could safely be removed as dead code.) As more constants will now be discovered at compile time, especially including such useful constants for optimization such as zero (0) and one (1), more benefits of constant folding for addition, multiplication and division will arise [Bidw86].

2.2.2 Uninitialized Variable Access

A special value denoting the uninitialized state may be assigned to each declared variable. Whenever this value is accessed, it may be flagged as accessing an uninitialized variable. As the set of possible values would be maintained, the variable may be initialized

in one branch of the control flow, but not in another, and when the control flows rejoin it would still be detected as a possibly illegal access. Note that it could not be sure to be illegal; program input could conceivably never go through any route that would not initialize the variable, yet the possibility would exist and the programmer should be notified.

2.2.3 Range Checking

When an array is accessed, care must be taken that the index is within the declared bounds. If the index exceeds the array bounds (either above or below), then other unknown portions of data may be overwritten, the program's code may be modified without the programmer's knowledge, and even other programs' code and data may be modified on some systems.

Although obvious that this must be prevented, the overhead to ensure prevention is extraordinarily large. Gupta mentioned that execution times when run-time bounds checks are in place may double [Gupt93], and Chin and Goh state that such bounds checks may add up to 50% overhead [Chin95]. Although suitable during debugging to ensure that test cases by the programmer do not cause boundary faults, such overhead may not be acceptable in production programs.

If the compiler takes no notice of the array other than that it exists and that it must be checked at run-time, then each array access would also require an index check against both upper and lower bounds. As array accesses may often be within loops and other important areas of the program, it is clear why bound checks may hinder programs

seriously. Some of the propagation methods applied to constants have also been applied to bound checks.

Although some computers may have a single instruction capable of checking both upper and lower bounds, most typically the upper and lower bounds may be treated separately as one may be optimized, but the other may not. Local analysis can discover when bound checks are identical and it can discover when one bound check is subsumed by another [Gupt93]. Naturally, such values for ranges may be checked globally as well as locally. Range checks for arrays may be hoisted from within a loop to outside its bounds with modifications [Gupt93][Chin95]. Asuru and Hedrick have shown that both common subexpression elimination, which may be applied to range bounds, and code hoisting may be done simultaneously without separate passes leading to greater efficiency [Asur93]. All such checks are adaptations of more traditional optimizations for other constructs modified for ranges. Just as a constant may be propagated when its value has not changed, a bound check may be propagated when its index has not changed, leaving it definitely safe within another bound check. Just as other code may be hoisted from within a loop, a bound check may be hoisted as well. These techniques may eliminate a large number of checks, anywhere from 42% to 100% in small sample programs tested by Gutpa [Gupt93].

By maintaining the set of all possible values for each variable, much of range checking comes for free at compile time; when the set of possible values for an index contains no values outside the range, then no run-time bound check code must be produced. Rather, when it detects that there may be a boundary violation, the compiler

may warn the programmer. The programmer may insert his own manual bound check (such as a simple if/then/else clause) or allow the compiler to insert one at such remaining points.

CHAPTER III

POSSIBILITY SET SEMANTICS

In this chapter, a new method for static checking of certain runtime errors, namely Possibility Set Semantics (PSS), is presented. Possibility sets will be defined by the actions of a virtual machine described in the sections below. Virtual machine instructions are specified along with their semantics. The notation used to define the possibility sets is described for the general case of high-level procedure-oriented languages and the included virtual machine. A sufficient number of statements to demonstrate each class of statements in the virtual machine is introduced with explanation. Possible compiler actions such as issuance of error messages also are given along with PSS definitions.

3.1 Notation for Possibility Set Semantics

Possibility set semantics may be discussed in the abstract independent of any particular form or in the specific form of a language. The notation for possibility sets themselves and their most general use will first be discussed.

Variables will be written in italicized lower-case in the text. Values, the data which may be stored in variables, will be described in the text using either numbers or variables.

For each variable x , there is an associated possibility set for x . Variables will be written in lowercase and the possibility set for the associated variable will be written in uppercase. For example, assigning the value one (1) to the variable x would result in $X=\{1\}$.

The possibility set for a variable may typically contain any value which may be assigned to the variable. It may also contain special values not possible in the value semantics. One such special value, represented using the empty set (\emptyset), stands for an uninitialized variable.

Boolean variables have their own possibility sets, indicated in the manner above, but they often also have extra associated sets. These sets associate possibility sets with the 'true' or positive possibility and the 'false' or negative possibility. These extra associated sets will be noted with a subscripted T and F for the true/positive and false/negative sets, respectively. For example, a boolean variable b will have its own possibility set called B , which may hold T, F, or \emptyset . Where B is true, B_T contains the appropriate possibility sets; where B is false, B_F is similarly in effect.

Variables as discussed within this thesis have types. For this thesis's purposes, the most important aspect of a type is the values which it may contain. This set of values for any variable will be represented as the capital letter R with a subscripted variable name. The set of values which the variable x may contain would be represented as R_x .

3.2 Virtual Machine Notation

The virtual machine which defines possibility sets is presented below. The language is designed to conform to standard notation wherever possible.

The statements have both value semantics and possibility set semantics. Both are presented, the value semantics first and then the possibility set semantics.

The definitions for the value semantics closely follow the expected definitions where possible. A program counter, PC, tracks which statement will be executed. Since at no point is the PC explicitly set to a numeric value, the instruction size and other implementation details of the machine itself are irrelevant to the discussion. Some instructions do refer to a word size when dealing with values at the bit level, but the actual word size is not specified.

The definitions for the possibility set semantics use the notation given above in section 3.1 where possible. In addition, there is the concept that the analysis moves from one statement to another. This is embodied in the analysis counter, AC, analogous to the program counter, PC, for value semantics. AC is used in loop analysis. Separate terms for the instruction pointer for value semantics and possibility set semantics are used in order to avoid confusion when they do not coincide.

At times additional sets may be defined beyond the expected set or sets associated with each variable. The \mathcal{V} (script V) set is the most prevalent of these sets. It contains ordered pairs containing all visible variables and their possibility sets. As variables enter and leave scope, they may be included in \mathcal{V} or excluded from \mathcal{V} . Unless specified

otherwise, it is considered that all validly referenced possibility sets are retrieved from \mathcal{V} . \mathcal{V}_X represents the set of visible variables and possibility sets at point X, where X is typically represented as a label.

$\mathcal{V} = \{ (x, \{1\}), (y, \{2\}), (z, \{3,4,5,10,11,12\}) \}$ would mean that $X = \{1\}$, $Y = \{2\}$, and $Z = \{3,4,5,10,11,12\}$. The variables x , y , and z would be the only variables that could validly be referenced. The use of any other variable would be the use of an undeclared variable.

3.3 Basic statements

The following statements form the core outline of many languages, plus a few miscellaneous instructions. The semantics is defined in terms of the effect the statements have on \mathcal{V} and the value of \mathcal{V} at those points.

3.3.1 begin_procedure

The value semantics of this typically would be to manage the stack frame and perform any system dependent initializations for the routine.

The possibility set semantics for this initializes \mathcal{V} . \mathcal{V} is the special set containing all visible variables. As there are no variables yet defined other than possibly globals, \mathcal{V} should be set equal to Globals, the set of all global variables. If there are no globals, then

Globals={ } and therefore $\mathcal{V}=\{\}$ upon **begin_procedure**. Further discussion of globals will be delayed until section 4.1.

3.3.2 end_procedure

The value semantics of this typically would be to restore the stack and perform any system dependent terminations for the routine before returning. This may correspond to an implied end of procedure or an explicit return within a procedure.

The possibility set semantics for this is:

$$\mathcal{V}=\{\}$$

That is, all previously declared variables and their possibility sets are no longer available.

This is important considering that analysis continues.

For example, if **end_procedure** is present within an **if's true** clause, \mathcal{V} would be emptied. This would leave only the *false* clause's possibilities to continue onward to recombine at the **endif** with the empty \mathcal{V} (see Figure 1 below).

```

// Assume X={1..10}
if(x<5) // Splits the X set
{
... // X={1..4}
return; // an end_procedure
//  $\mathcal{V}=\{\}$ , so there is no X now
}
else
{
... // X={5..10}
}
// X= $\{\} \cup \{5..10\}=\{5..10\}$ 
// the true clause's values had been removed by the
// return and so were not present to be unioned.

```

Figure 1. Example of end_procedure from C

3.3.3 declare x of type R

The value semantics of this may vary. The allocation of space for the variable may be merged into the `begin_procedure` statement. Some languages may wish to initialize x to zero or to another initialization value. Such an initialization should be done as a separate statement. Naturally, the program translating the intermediate code to the target's native code may recombine the statements, if possible, on the target machine.

The possibility set semantics for this follows the actual meaning of the declare:

$$\mathcal{V} = \mathcal{V} \cup \{ (x, \{\emptyset\}) \}$$

It places x in \mathcal{V} , setting x 's possibility set to include only \emptyset , the special symbol representing an uninitialized variable.

The parameter R is a set of all possible values for the type, the maximum set of possibilities for x . This may be referred to as R_x .

3.3.4 **kill** x

This statement has no explicit counterpart in normal execution. However, it may be an implicit action caused by other statements. It invalidates the use of x , marking it to be undefined. A language may define that this operation should be performed upon a loop index after the loop or other similar operations.

Its definition in possibility set semantics is straightforward:

$$X = \{\emptyset\}$$

Any references of the variable after this, but before other assignments, will be tagged as references to an uninitialized variable.

3.3.5 **undeclare** v

This is a more severe counterpart to **kill**. Rather than just declaring a variable's use to be undefined, but redefinable, this eliminates all possible references to the variable short of a redeclaration. If a loop's index is declared within the loop, but not capable of being referenced outside the loop, this statement may be used. Additionally, if a variable is valid only within a subblock, then this statement may be used at the end of the subblock to invalidate its declared variables.

The possibility set semantics for this instruction removes v from \mathcal{V} :

$$\mathcal{V} = \{(x, Y) \mid \forall x, (x, Y) \in \mathcal{V} \wedge x \neq v\}$$

3.3.6 $y \leftarrow x$

The value semantics for **assignment** copies the value from x to y .

The possibility set semantics correspond exactly:

$$Y = X$$

3.3.7 *cast source x to destination y*

The value semantics correspond to an assignment of a value of one type to a value of a different type. Any value which may be held in y is copied from x .

The possibility set definition corresponds closely. Any element that may be assigned to the destination will be assigned; any element that may not be assigned to the destination flag a warning:

$$Y = \{x \mid \forall x, (x \in X \wedge x \in R_Y)\}$$

$$Y' = \{x \mid \forall x, (x \in X \wedge x \notin R_Y)\}$$

If Y' is not empty, then a warning should be issued explaining the possible loss of information. If Y is empty, then the assignment cannot execute correctly and the warning should instead be upgraded to an error.

If loss of information is allowed without incident in the language, then the information from Y' must be merged with Y, eliminating Y'. How that would be done would be dependent upon the language, but one common possibility would be to union Y with the lowest order bytes of Y'. Figure 2 gives an example illustrating this situation.

```
int L;          // Assume RL={-32768..32767}
char C;        // Assume RC={-128..127}
L=function();  // Assume L={-1000..100}
C=(char)L;     // Warning: C={-128..100}, C'={-1000..-129}
               // If the lowest order bytes of C' are merged with C,
               // then C={-128..127}
```

Figure 2. Example of assignment using cast from C

3.4 Arithmetic Instructions

The arithmetic instructions generally deal with addition, subtraction, multiplication, division and other similar instructions. They may overflow, underflow or be applied over invalid values. Typical instructions are described in the following subsections.

3.4.1 $x \leftarrow x + y$

The value semantics add the contents of x to the contents y , leaving the result in y . Overflow is allowed, leaving the result present in the lowest order byte(s). For example, assuming integer variables capable of holding 0 to 255:

```
                                ; assume r#.b refers to byte registers with  $R_R = \{0..255\}$   
r1.b  $\leftarrow$  #255                ; assign constant 255 to register byte variable 1  
r2.b  $\leftarrow$  #1                   ; assign constant 1 to register byte variable 2  
r1.b  $\leftarrow$  r1.b + r2.b           ; add allowing overflow  
                                ; r1.b would be left with the value 0.
```

As overflow is allowed, no run-time arithmetic error could occur; the lowest-order bytes are assumed to catch the overflow.

The possibility set semantics follow closely:

$$Y = \{ (x+y) \text{ bit-and } (2^{\text{bits-in-word}} - 1) \mid (\forall x, x \in X) \wedge (\forall y, y \in Y) \}$$

3.4.2 $x \leftarrow x + y$ on overflow goto z

In value semantics, this would add the contents of x to the contents of y , leaving the result in y if no overflow occurs. If an overflow occurs, then control branches to label z . If no destination is supplied, it is assumed the target machine would provide a default destination for the trap.

The possibility set semantics for this is the first to be introduced which may detect an overflow exception:

$$Y = \{x+y \mid (\forall x, x \in X) \wedge (\forall y, y \in Y) \wedge ((x+y) \in R_Y)\}$$

$$Y' = \{x+y \mid (\forall x, x \in X) \wedge (\forall y, y \in Y) \wedge ((x+y) \notin R_Y)\}$$

If $Y' = \{\}$, remove on overflow error clause.

If $Y \neq \{\}$ and $Y' \neq \{\}$, issue warning.

If $Y = \{\}$ and $Y' \neq \{\}$, issue error.

Y is then the resultant set of possible values, and Y' is the set of overflowed results. If Y' is empty, then no overflow can occur so the error clause may be eliminated. If Y is empty, then there are no possible results other than overflow; this should flag an error rather than a warning. If both Y and Y' are non-empty then a warning should be issued to the programmer indicating a possible overflow condition; the set Y' may be displayed to the programmer if more information is desired.

3.4.3 $y \leftarrow y / x$ on divzero goto z

The value semantics divides y by x , trapping to z on a division by zero.

This possibility set semantics for division follow the above addition closely:

$$Y = \{y/x \mid (\forall y, y \in Y) \wedge (\forall x, (x \in X) \wedge (x \neq 0))\}$$

If $X = \{0\}$, flag a division by zero error.

If $0 \in X$, flag a warning of possible division by zero.

If $0 \notin X$, remove the on error clause.

If the only possibility for x is zero (0), then the instruction can never execute safely so an error should be issued. If x can never be 0, then the instruction will always execute safely and the error clause may be safely eliminated. If x can be 0, but is not necessarily 0, then the code should compile correctly, but the user should be warned that the program may crash with a division by zero error.

3.4.4 **bound** *low w, check x, high y on error label z*

The value semantics for this would check the value of x against the low value w and the high value y ensuring that x is valid. If x is not within the bounds, it traps to z . This instruction typically would bound an array index. Although this may seem to have more in common with the comparisons below, it is simpler in scope and falls more easily under arithmetic operations.

The possibility set semantics for this checks that no value in X exceeds the given bounds.

$$X = \{x \mid \forall x, x \in X \wedge (x \geq w \text{ and } x \leq y) \}$$

$$X' = \{x \mid \forall x, x \in X \wedge (x < w \text{ or } x > y)\}$$

If $X' = \{\}$, remove statement.

If $X' \neq \{\}$ and $X \neq \{\}$, flag a warning that array bounds may be exceeded.

If $X' \neq \{\}$ and $X = \{\}$, flag an error that array bounds will be exceeded.

X' is the set of values which would exceed the bounds. If there are no values which would exceed the bounds, then the statement will never cause an action in value semantics and thus may safely be removed.

If $X' \neq \{\}$, then there are possible values which would exceed the bounds and the statement may be executed in value semantics. At least a warning should be issued. If there are no valid entries, $X = \{\}$, then an error should be issued because the exception would always be raised in value semantics.

Other arithmetic operations behave similar to the statements described above and so their description is omitted.

3.5 Comparisons

In the above statements, the results of statements were analogous to the normal execution of the statement--uniform members of a set, a null result, or errors. For example, addition of integers yielded only integers or an error. Boundary checks have no effect or produce an error. Many similar statements follow directly with little modification. They are similar to each other in one very important aspect: they need not lose any information that a normal execution would not lose.

Comparisons are quite different. In value semantics, they compare two values yielding a boolean. This boolean must be maintained in possibility set semantics, but there is also a second level of information produced. For the boolean to be *true* or *false*, only certain combinations of values are possible. Possibility set semantics partitions each of the compared expressions into those possibilities which may yield the *true* boolean (the T possibility set) and those possibilities which may yield the *false* boolean (the F possibility set).

Optimally, all possible partitionings should be maintained and this partitioning should be performed once for each time the statement could be executed, splitting the control flow of analysis repeatedly. Obviously, if the executing program could execute the comparison an infinite number of times, it would not be possible to split the control flow of analysis an infinite number of times. And such partitionings into *true* and *false* could not be perfect as some elements would be members of both T and F sets. Most often the

resultant boolean will be {T,F} and the T and F sets will not be disjoint. Splitting control flow will not be considered further.

3.5.1 $b \leftarrow (x < y)$

The values semantics for this compares x to y , yielding the boolean *true* if $x < y$ or the boolean *false* if $x \geq y$.

The possibility set semantics for this instruction is more complex and produces less precise information. First, some intermediate sets are introduced to simplify the semantics. T_x is the set of values from X which holds when the condition is *true*, and F_x is the set of values from X which holds when the condition is *false*. T_y and F_y are analogous.

$$T_x = \{x \mid \forall x, x \in X \wedge x < Y_{\max}\}$$

$$T_y = \{y \mid \forall y, y \in Y \wedge y \geq X_{\min}\}$$

$$F_x = \{x \mid \forall x, x \in X \wedge x \geq Y_{\min}\}$$

$$F_y = \{y \mid \forall y, y \in Y \wedge y < X_{\max}\}$$

The possibility set semantics for $b \leftarrow (x < y)$ is defined by the following:

$$B = \{C \mid C = \{\} \cup (\{F\} \text{ if } (F_x \cup F_y) \neq \{\}) \cup (\{T\} \text{ if } (T_x \cup T_y) \neq \{\})\}$$

$$B_T = \{(x, T_x), (y, T_y)\}$$

$$B_F = \{(x, F_x), (y, F_y)\}$$

$$\mathcal{V} = \mathcal{V} \cup \{(bt, B_T), (bf, B_F)\}$$

As an example, consider $X=\{4..9\}$ and $Y=\{1..7\}$:

$$T_x=\{4..6\}$$

$$T_y=\{4..7\}$$

$$F_x=\{4..9\}$$

$$F_y=\{1..7\}$$

$$B=\{T,F\}$$

In the above example, the T for x and y yield some information, but the F for x and y do not narrow the possibilities. If the comparison were the other direction with the same data, then F would offer more information while T would yield no more information. This narrowing of possibilities is what allows constants to emerge even when not explicitly defined. Although not perfect, the information retained is yet useful.

The T and F sets are maintained with the resulting boolean B. This information is used by statements such as **and** and **or**. This is extra information, not associated with the variable at run-time, and it is ignored when actual code is finally produced.

Similar comparison statements follow easily from the above statement as seen in Figure 3.

```

positive=(x>=1); // Assume X={-128..127}
                // Positive={T,F},
                // PositiveT={X={-128..0}}
                // PositiveF={X={1..127}}

```

Figure 3. Example of greater-than-or-equal (>=) from C

3.5.2 $b \leftarrow (x \text{ and } y)$

The value semantics assigns the result of 'logical and' of x and y to b .

The actual boolean is easy to obtain, but more operations are required for the booleans' extra information. It performs a modified intersection of the *true* components of the sources and a modified union of the *false* components. The possibility set semantics for **and** is shown below:

$$B = \{ C \mid C = \{ \} \cup (\{ T \} \text{ if } (T \in (X \cap Y))) \cup (\{ F \} \text{ if } (F \in (X \cup Y))) \}$$

$$B_F = \{ (r, P \cup Q) \mid \forall r ((r, P) \in X_F, \text{ else } P = \{ \}) \wedge ((r, Q) \in Y_F, \text{ else } Q = \{ \}) \}$$

$$B_T = \{ (r, P \cap Q) \mid \forall r ((r, P) \in X_T, \text{ else } P = Q) \wedge ((r, Q) \in Y_T, \text{ else } Q = P) \}$$

The above definitions create new possibility sets for both the *true* and *false* conditions.

They take advantage of the combined information if the referenced variables are present in both sets. If the referenced variables are not in both sets, the possibility sets transfer directly to the new combined set.

An example of the logical **and** where x and y are distinct variables is shown in Figure 4.

See Figure 5 for an example of the logical **and** where x is the same variable as y . In Figure 5, although no constant of 7 actually appears in the code, the methods allow the possibility sets to be partitioned enough to discover that x may only be 7 when the comparisons are true. If 7 is not a possibility before the comparisons, then the comparison would instead reduce to a definite false.

```

; assuming X={1..10} and Y={1..10}

b1 ← x < #5

; B1={T,F}

; B1_T={{x,{1..4}}}

; B1_F={{x,{5..10}}}

b2 ← y < #5

; B2={T,F}

; B2_T={{y,{1..4}}}

; B2_F={{y,{5..10}}}

b ← b1 and b2

; B={T,F}

; B_T={{x,{1..4}},{y,{1..4}}}

; B_F={{x,{5..10}},{y,{5..10}}}

```

Figure 4. Example of logical *and* where *x* and *y* are distinct.

```

; assuming X={1..10}

b1 ← x < #8

; B1={T,F}

; B1_T={{x,{1..7}}}

; B1_F={{x,{8..10}}}

b2 ← x > #6

; B2={T,F}

; B2_T={{x,{7..10}}}

; B2_F={{x,{1..6}}}

b ← b1 and b2

; B={T,F}

; B_T={{x,{7}}}

; B_F={{x,{1..6,8..10}}}

```

Figure 5. Example of logical *and* where x and y are the same variable.

3.5.3 if/else/endif

```
P:  
if (boolean x) then goto destination R  
Q:  
... true clause ...  
Q':  
else  
R:  
... false clause ...  
R':  
endif  
S:
```

where P, Q, Q', R, R', and S are labels representing execution points. The value semantics for the statement would check the current condition of x and then branch or not according to its value. Assuming that there can be no branches into or out of the conditional structure other than the defined entry and exit points, then the following holds. If x is *true*, then $PC \leftarrow Q$. And when the PC reaches the **else** statement, $PC \leftarrow S$. If x is *false*, then $PC \leftarrow R$. Execution would then fall through to S after the **endif** statement.

3.5.3 if/else/endif

```
P:  
if (boolean x) then goto destination R  
Q:  
... true clause ...  
Q':  
else  
R:  
... false clause ...  
R':  
endif  
S:
```

where P, Q, Q', R, R', and S are labels representing execution points. The value semantics for the statement would check the current condition of x and then branch or not according to its value. Assuming that there can be no branches into or out of the conditional structure other than the defined entry and exit points, then the following holds. If x is *true*, then $PC \leftarrow Q$. And when the PC reaches the **else** statement, $PC \leftarrow S$. If x is *false*, then $PC \leftarrow R$. Execution would then fall through to S after the **endif** statement.

The possibility set semantics for this statement must analyze both the *true* and *false* portions. Q through Q' use X_T , and R through R' use X_F . In the following, recall that \mathcal{V}_P , \mathcal{V}_Q , \mathcal{V}_R and \mathcal{V}_S refer to \mathcal{V} at points P, Q, R, and S respectively. For the *true* clause, the T sets override the normal values, so \mathcal{V} is assigned as follows:

$$\mathcal{V}_Q = \{(u, V) \mid \forall u, (u, V) \in X_T, \text{ else } (u, V) \in \mathcal{V}_P\}$$

That is, if a variable is mentioned in X_T , then that is the relevant set, else the normal set from before the *if* is still used. If a variable is not mentioned in the *if* comparison, then there is no more specific information available for it than its normal set.

The *else* clause also has its F sets override \mathcal{V} in the *false* clause.

$$\mathcal{V}_R = \{(u, V) \mid \forall u, (u, V) \in X_F, \text{ else } (u, V) \in \mathcal{V}_P\}$$

After the end of each clause, the control flows and hence possibility sets recombine:

$$\mathcal{V}_S = \{(u, V \cup W) \mid \forall u, ((u, V) \in \mathcal{V}_Q \wedge (u, W) \in \mathcal{V}_R)\}$$

That is, if the end of the *true* clause leaves with $X=\{1\}$ and the end of the *false* clause leaves with $X=\{0\}$, then **endif** will produce $X=\{0,1\}$.

3.6 Branching

Branching deals with all transfers of control other than the implicit flow of control from one instruction to the following instruction. It includes the error flow of control possible from *on error* conditions in previous statements.

3.6.1 Forward jump: goto *destination S*

P:

Forward jump: goto *S*

Q:

... code ...

R:

S:

Value semantics would define a forward **goto** as an unconditional $PC \leftarrow S$, where *S* is a point in the program which has not yet been processed.

The possibility set semantics for this instruction transfers all possible values from the forward **goto** point to the label point. This must accommodate the fact that there may be code at point *R* which would continue at the label as well. The possibility set semantics for this forward **goto** is defined precisely as shown:

$$\mathcal{V}_S = \{(x, Y \cup Z) \mid \forall x, (x, Y) \in \mathcal{V}_P \text{ or } (x, Z) \in \mathcal{V}_R\}$$

$$\mathcal{V}_Q = \{\}$$

This states that all sets existing before the forward **goto** and all sets existing before **label** must be unioned together at the **label** point. Q is defined to be a label which cannot be the target of a branch; another label may immediately follow Q and be the target of a branch. As any code following the forward **goto** will not be executed unless there is a branch into the code and such a branch is disallowed, \mathcal{V}_Q is set equal to the empty set. By assigning \mathcal{V}_Q to be empty, it may be unioned at an **endif** or at another label. Figure 6 provides an example.

The backward **goto** statement is covered below within the looping section.

```

// X={1..10}  The initial assumption.
if(x<5)
{
// BT={XT={1..4}} BF={XF={5..10}}, B={T,F}
// X={1..4}  This set is carried forward to the label.
goto test;
// X={}      The set is cleared after the label.
}
else
{
// X={5..10}
...
}
// X={} union {5..10}={5..10}
// Union true and false clauses at the endif
...
test:
// X={1..4} union {5..10}={1..10}
// Union pre-label and post-label possibility sets.

```

Figure 6. Example of a forward goto from C

3.7 Looping

Any goto with a destination which has already been analyzed is effectively a possible loop.

3.7.1 Backward jump: goto Z

P:

Z:

Q:

... code ...

R:

Backward jump: goto Z

S:

where P, Q, R, S, and Z are labels. The value semantics define **goto** as an unconditional $PC \leftarrow Z$ where $Z < R$.

The possibility set semantics for this instruction are the most intricate. This must re-evaluate portions, but it cannot do so indefinitely. The analysis method presented terminates regardless of the code within the loop, but the analysis is not quick. In a practical implementation, some modifications to be explored in sections 4.4.1 and 4.4.2 may be made if speed is desired above accuracy.

S is defined as a point which cannot be the target of a branch; another label may be placed after S to be the target of a branch. Clearly, the statement after the **goto**, labeled S:, cannot be executed as it cannot be a branch target, so \mathcal{V}_S is set to the empty set:

$$\mathcal{V}_S = \{\}$$

On the first pass of analysis, before the **goto** is discovered, the \mathcal{V}_P from before the destination label continues onward into the section labeled Q. On subsequent passes, \mathcal{V}_R from immediately before the **goto** must be transferred to the destination label.

On the first pass,

$$\mathcal{V}_Q = \mathcal{V}_P$$

$$\mathcal{V}_{Q'} = \mathcal{V}_P$$

On subsequent passes,

$$\mathcal{V}_Q = \mathcal{V}_R$$

If $\mathcal{V}_{Q'} \neq \mathcal{V}_Q$, then $\mathcal{V}_Q = \mathcal{V}_{Q'} \cup \mathcal{V}_Q$ and $AC \leftarrow z$,

else continue analysis past **goto** and set $AC \leftarrow S$.

The $\mathcal{V}_{Q'}$ above is introduced in order to provide a halting condition for the **goto**. $\mathcal{V}_{Q'}$ tracks all possibilities that have been analyzed in the loop. If more possibilities have been discovered, then $\mathcal{V}_{Q'}$ will differ from \mathcal{V}_Q . If no new possibilities have been determined during a pass, then no new possibilities will ever be created, and the loop may stop.

An infinite loop may merely assign a value to a variable and then unconditionally loop backwards. As all data which may be handled by this analysis is finite, the analysis loop will eventually discover them all and the analysis will stop.

For example, assume that a byte may contain any value from 0 through 255. If a loop always increased a byte, then after 256 passes the values would repeat. When the values repeat, \mathcal{V}_Q would no longer change. The analysis loop would then stop, and processing would continue past the `goto`.

Note that the analysis of statements within a loop may change as the statements are analyzed repeatedly. Therefore, any warnings, removals and optimizations may not be performed until the final pass has been completed. Each destination must maintain a continuously unioned set of possibilities to prevent invalid optimizations. The possibility sets created by statements within the loop must be considered to be only intermediate possibility sets until the loop ends. The actual possibility set used for error detection and optimization is the union of all intermediate possibility sets created during the loop passes. If a warning can occur in any pass of analysis, then the warning needs to be issued, but errors signifying the absence of valid data may become warnings instead after further analysis. The actual issue of such errors should be delayed until all loop analysis has finished.

As this is the most time consuming by far of all operations, considerations to practicality of speed in favor of accuracy are possible and will be covered in the next chapter.

CHAPTER IV

USAGE OF POSSIBILITY SET SEMANTICS

The possibility sets as defined in the previous chapter allow sufficient information to be retained from analysis to allow a greater amount of error detection and code optimization. The error detection follows naturally from the presence or absence of *on error* clauses and uninitialized variables after the final analysis. Code optimization follows from the narrowing of possibility sets to only one choice, removal of statements involved in error checking, transformation of integers into more efficient forms which can still hold all needed data, and removal of dead code which emerged through the lack of possibility sets. Some examples of this analysis is shown in Appendix B.

4.1 Interprocedural Analysis and Globals

Interprocedural analysis using possibility set semantics capable of retaining all conceivable information is not yet practical even in most cases where possible. The predominant reason behind this is that information is flowing both ways, to the function through its parameters and from the function through its return value. Each function would need to be analyzed and re-analyzed repeatedly.

Even if the above possibility is true, some benefits are available. System and language functions may be prototyped with the permissible parameters and all possible return values. This would allow the discovery of inconsistencies in the program, most notably in the handling of error results. Figure 7 illustrates this case.

```
int getchar(void) = -1..255; // Assume new prototype style in language library
int putchar(int=0..255);    // Assume new prototype style in language library

int c;
c=getchar();                // getchar() returns 0-255 or an error -1, so C={-1..255}
putchar(c);                 // putchar(c) expects 0-255, the -1 in C would be invalid
```

Figure 7. Example of prototype and usage from C-like language.

Without further information about globals, each function entry and each program call may redefine any and all globals to any possible value for the globals' types. System calls may be assumed not to redefine globals unless noted in the prototype, but user calls are difficult to handle.

It would be a simple task to determine which globals are defined in each function and its called functions, but it would not be quick to discover the values assigned to the globals. If it has been determined that a certain global is not defined in a called function (or any function it calls, etc.) then it may remain unchanged after the call. Without that assurance, the global needs to be set back to its maximum set of possibilities after each non-system call.

If extensions are added to the language to define what possibilities will be passed to functions and returned from functions, then that information could be used by this

analysis. The information declared by the user may be in error, so it should not be used if all possible warnings are desired.

4.2 Error Detection and Code Optimization

The technique described above supplies more information than would otherwise be available. That information may be used to detect some errors which otherwise would be caught only at run-time. It may also be used to remove some safety checks which have become redundant and to discover some otherwise hidden constants. The information above has already described many of the warnings and errors; the optimizations should be apparent after realizing which new information is available. The error detection and optimizations will be summarized below.

4.2.1 Error Detection

The following summarizes some of the error conditions that may be detected by using possibility set semantics.

4.2.1.1 Division by zero

Any division by a variable x where $0 \in X$ and $X \neq \{0\}$ is a division by zero warning.

Any division by a variable x where $X = \{0\}$ is a division by zero error.

4.2.1.2 Overflow

Anytime an operation upon the possibility sets yields a set with values not within the desired result set, an overflow warning will be flagged. If there are no possible valid results, then an error message would be issued.

4.2.1.3 Invalid array index

Any **bound** statements left after analysis should be flagged as warnings for invalid array indexes.

4.2.1.4 Invalid parameter to function

Anytime a function parameter's possibility set does not represent a subset of the declared possibility set, the compiler should warn of an invalid parameter to the function. This can detect a variety of errors in which special values within a type's possibility set represent exceptional conditions such as -1 representing end-of-file. Anytime a declared function has a set of normal values and an error result, and the error result is not valid in a subsequent function call, this warning would be issued. In a way, this is a more generalized overflow error for functions rather than for internally defined statements.

4.2.1.5 Possible loss of information

Of the explored statements, this applies only to the **cast** statement. In converting a variable from a type with more precision to a type with less, it may be determined that information may be lost. This could occur when there are members of a possibility set for a *long* variable which could not fit into a destination *short*. Or it could occur when converting floating point variables containing possible fractions when being converted to integers.

4.2.1.6 Reference to uninitialized variable

Anytime a variable with a possibility set containing \emptyset is referenced, this warning should be issued. This will detect not only those times an uninitialized variable is referenced in the initial basic block, but also those cases where several **if/else/endif** clauses and other control flow statements have been executed. It may possibly give warnings too often, but it will not miss any uninitialized local variables.

It does not miss any cases because the special value \emptyset is contagious. All operations other than direct assignment propagate \emptyset . Even after a direct assignment, a control construct may union the valid possibility set with another possibility set containing \emptyset . In such a circumstance, the analyzer knows that there may be execution sequences which do not initialize the variable. If a previously initialized variable, x , has an uninitialized variable, y , assigned to it, then x is also uninitialized.

Uninitialized global variables are not discovered without interprocedural analysis. In this situation, local variables passed by reference are effectively global after the procedure call.

4.3 Code Optimization

If checks were to be placed around every division previously, then those checks may be removed if there is no division by zero warnings. Similar optimizations of code to perform redundant safety checks may be removed and considered code optimizations.

Just as a constant has a possibility set containing only the constant, if a set has only one member (other than \emptyset), then it may be replaced by this value. That is, even if the constant was never explicitly assigned to a variable, if the variable's possibility set narrows enough to leave only one value possible, then the variable may safely be replaced by that value. This narrowing generally would occur through the **if/else/endif** clause.

A case statement in a language which requires a constant for the **case** would often result in the replacement of the **case** variable by its constant value.

If a variable has been declared to be a certain type, yet its possibility sets show that it has been used without need of extended precision then its type could be changed to a type with less precision. If a variable has been declared *long*, yet its values are only one through ten, then it may safely be changed into a *char*. The largest restriction upon this is when the variable must be accessed externally.

4.4 Practicality

The above techniques illustrate ways to gain information about the source program. But that information comes at a cost, time and space. A program without loops can be analyzed very easily and without much increased cost in terms of resources.

4.4.1 Determination of Definite Errors in Loop Analysis

One of the most important factors in easing time restrictions is the notion that once a definite error has occurred within a loop, the loop's analysis may be terminated. Only those sets determined up until the time of the error should be carried past the loop, if any. It should be repeated that this extra loop termination condition only applies to certain, definite errors. If any possibility is valid, the loop analysis should continue to keep the information. This helps greatly with loops that access information in arrays. Once the loop index reaches a value which would provoke an invalid array index error, the loop analysis may terminate.

A loop spanning the entire set of values for a type, but indexing an array valid only for a few index values, may quickly be terminated rather than have useless analysis performed upon it. For an example of this, the reader is referred to the third example in Appendix B.

This technique is covered in this section rather than within the looping section because in some rare case it may not be desirable. Although an index accessing an array should not violate the array bounds, conceivably in some programs it could violate the

bounds and still produce correct results. This would occur if the violation only altered memory not protected by the operating system and not used otherwise in the system. Yet it may be the wish of the programmer to have more errors or warnings found within the loop after the array bound violation. Nonetheless, it is safe to have loop analysis termination upon error be the default. A flag to deactivate the loop error termination condition could be provided in case it would ever be desirable.

4.4.2 Early Termination of Loop Analysis by Expanding Possibilities

Another factor in limiting the time spent in the analysis of the program could be the early termination of loop analysis by expanding the possibilities for variables. This would not produce the same results, giving more possible values to variables than would otherwise be found, but it could be effective in quicker analysis of some code during development. This could be provided as an option along with a maximum number of loop passes.

A set maximum number of passes may be specified; any variable x still changing after n passes may be safely replaced by the full set of possibilities for its type, R_x . And then the loop may continue to be processed again with the same maximum condition until the loop stabilizes. This method will terminate analysis of a loop after a reasonable number of passes even when the loop has many variables which constantly change, and it still maintains generally good information about a loop. Other safe possibilities to speed the analysis of the loop are also possible.

4.4.3 Conservation of Space

To conserve space, not all information about a possibility set must be saved. Without possibility set semantics or other optimizations, all variables effectively have a possibility set equal to that of their type. In this manner, no information whatsoever is known about the variables and no optimizations will be performed. Expanding the number of values within a possibility set is always the safe option.

Much of the work done in a program is done with the smallest values. A bit vector of eight bytes could contain the set of values -128 to 127 for signed types, or 0 to 255 for unsigned types. All possibilities for the low values would thus be maintained. Aside from the low values, a few ranges could be maintained. The less critical or the more uniform the values, the more useful this type of modification would be. Arrays should find this type of treatment very useful. Explicitly named variables within the procedure would find it less useful.

With these modifications, I believe this form of analysis to be useful and practical. This technique certainly requires more resources than the typical compilation, but generally the development platform is at least as powerful as the platform upon which the program will be executed and should be capable of spending the required time on analysis.

4.4.4 Limit to Complexity

The analysis is at most within the same order of complexity as the analyzed program. Any program which is feasible to execute is feasible to analyze. This can be seen in that a basic block to execute in the final code is still a basic block during analysis.

The only time analysis loops back is when the program loops back. Without performing the analysis between procedures, once a procedure has been analyzed it need not be re-analyzed. And loops which become infinite in the executed program terminate under analysis, so regardless of how intricate the analyzed code, the possibility semantics analysis will terminate. The analysis of an individual statement may take longer than the execution of the same statement, but only by a constant factor. This initial investment in analysis can avoid unexpected errors in programs used frequently or used for a long duration.

CHAPTER V

CONCLUSION

5.1 Future Work

One obvious extension to the virtual machine would be the modification of names to indicate the declaration level. This virtual machine is meant to be a starting point for various procedural languages. If both possibility sets semantics and value semantics are synchronized, extension to this machine is possible. The new statement must first be defined for value semantics. After value semantics are determined, the implications for all inputs and outputs must be considered to determine the corresponding definition for possibility sets.

Interprocedural analysis should be developed to gain as much information as possible from the interaction of procedures, their parameters, and their results. Any information so gathered, even quite limited information, would be helpful in narrowing the possibility sets. Information from this analysis could be used in other error detection and optimization, and information from other forms of analysis could be useful in possibility set semantics.

5.2 Conclusions

For the uses of error detection and some forms of code optimization, the maintenance of sets of possible values is not currently being performed. Although unrealistic to maintain thousands of separate values in an actual set for a compiler, typically a set of at most one element is being maintained. All others merely become the set of an unknown number of values having no error detection or optimization value. In the thesis, it has been shown how to maintain more information about the value which a variable may hold at each point in the program. Surely the maintenance of more information than simply *nonconst* would be of use in the detection of errors and optimization of code. The more information available to the programmer about where a program may experience an exception, the better able the programmer will be to correct those errors before they may occur.

An optimizing compiler including this analysis would generally not apply this analysis during each compilation. It would be most useful when the program crashes for an unseen reason and before release in order to discover any hidden flaws. When there are a lot of resources invested in the success of a project, it is important to ensure the quality of the product in every way possible. Hopefully this technique will be a useful tool in the assurance of quality software.

References

- [Aho88] Aho, A.V., Sethi, R. and Ullman, J.D. 1988. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, pp. 88, 164-165.
- [Aike95] Aiken, A., Williams, J.H. and Wimmers, E.L. 1995. "Safe: A Semantic Technique for Transforming Programs in the Presence of Errors." *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 1, pp. 63-84.
- [Asur93] Asuru, J. M. and Hedrick, G. E. 1993. "Joint Common Subexpression and Code Hoisting Optimization." Oklahoma State University, Stillwater, Oklahoma, pp. 1-23.
- [Babe87] Baber, R. L. 1987. *The Spine of Software: Designing Provably Correct Software Theory and Practice*. John Wiley & Sons Ltd, New York, New York, pp. 64-66.
- [Back84] Backhouse, R. 1984. "Global Data Flow Analysis Problems Arising in Locally Least-Cost Error Recovery." *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 2, pp. 192-214.
- [Bidw86] Bidwell, D.R. 1986. *Comparison of Optimization Techniques in Code Generation*. Illinois Institute of Technology, Ph.D. Dissertation. University Microfilms International.
- [Bjoe82] Bjoener, D. and Jones, C. B. 1982. *Formal Specification and Software Development*. Prentice-Hall International, New Jersey, p. 15.
- [Bran94] Brandis, M.M. and Moessenboeck, H. 1994. "Single-Pass Generation of Static Single-Assignment Form for Structured Languages". *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6, p. 1685.
- [Chin95] Chin, W. and Goh, E. 1995. "A Reexamination of 'Optimization of Array Subscript Range Checks'". *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 2, pp. 217-227.
- [Clic95] Click, C. 1995. "Global Code Motion Global Value Numbering." *ACM Sigplan Notices*, Vol. 30, No. 6, pp. 247-257.

- [Darw91] Darwin, Ian F. 1991. *Checking C Programs with lint*. O'Reilly & Associates, Inc., Sebastopol, CA, pp.16-23.
- [Fosd76] Fosdick, L.D. and Osterweil, L.J. 1976. "Data Flow Analysis in Software Reliability." *Computing Surveys*, Vol. 8, No. 3. pp. 305-330.
- [Geor85] George, K.M. and Hedrick, G.E. 1985. "A Note on Error Recovery in LR(1) Parsers." Oklahoma State University, Stillwater, Oklahoma, pp. 1-6.
- [Gupt93] Gupta, R. 1993. "Optimizing Array Bound Checks Using Flow Analysis." *ACM Letters on Programming Languages and Systems*, Vol. 2, Nos. 1-4, pp. 135-150.
- [Lech68] Lecht, C.P. 1968. *The Programmer's PL/I: A Complete Reference*. McGraw-Hill Book Company, New York, New York, pp. 156-157, 401-403.
- [Levi92] Levine, John R., Mason, T., and Brown, D. 1992. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, pp. 248-249.
- [Marc86] Marcotty, M. and Ledgard, H. 1986. *Programming Language Landscape: Syntax, Semantics, and Implementation*, Science Research Associates, Inc., Chicago, Illinois, pp. 110-112.
- [Metz93] Metzger, R. and Stoud, S. 1993. "Interprocedural Constant Propagation: An Empirical Study." *ACM Letters on Programming Languages and Systems*, Vol. 2, Nos. 1-4, pp. 231-232.
- [Rich85] Richter, H. 1985. "Noncorrecting Syntax Error Recovery." *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 478-489.
- [Sipp83] Sippu, S. and Soisalon-Soininen, E. 1983. "A Syntax-Error-Handling Technique and Its Experimental Analysis." *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 4, pp. 656-679.
- [Thom68] Thompson, K. 1968. "Regular Expression Search Algorithm." *Communications of the ACM*, Vol. 11, No. 6, pp. 419-422.
- [Trav91] Traviolia, M. L. 1991. *Improved Regionally Least-Cost Syntax Error Recovery*. Oklahoma State University, Stillwater, Oklahoma, pp. 1-163.
- [Wijn76] Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, Sintzoff, M.,

Lindsey, C. H., Meertens, L. G. L. T. and Fisker, R. G. 1976. *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag, New York, New York.

APPENDIX A

GLOSSARY

ANSI C: American National Standards Institute C. The official, standardized version of C.

K&R C: Kernighan and Ritchie C. The original de facto standard C language as designed by its creators, Brian Kernighan and Dennis Ritchie.

Possibility Set: A set associated with a variable x at a point p in the program's code containing all values possible for the variable x at the point p during all possible execution paths regardless of input. The virtual machine in Chapter 3 defines possibility sets precisely.

Possibility Set Semantics: The creation and analysis of possibility sets as presented in this thesis.

PSS: *See Possibility Set Semantics.*

R_v : The set of values which the variable v is capable of containing according to its type.

V_T : The set of values in effect when the variable v is true.

V_F : The set of values in effect when the variable v is false.

\mathcal{V} (Script V): The master set of ordered pairs containing all currently visible variables and their associated possibility sets.

APPENDIX B

Algorithms in C and PSS Analysis

Example 1

Sorting Algorithm

Original code in C:

```
// Simple sort
int n,i,j,temp;
int s[10];

n=10;

for(i=0;i<n;i++)
    s[i]=input();

for(i=0;i<n-1;i++)
    for(j=1;j<n;j++)
        if(s[i]>s[j])
        {
            temp=s[i]; s[i]=s[j]; s[j]=temp;
        }

for(i=0;i<n;i++)
    output(s[i]);
```

PSS Analysis:

declare n of {-32768..32767}	N={ \emptyset }
declare i of {-32768..32767}	I={ \emptyset }
declare j of {-32768..32767}	J={ \emptyset }
declare temp of {-32768..32767}	Temp={ \emptyset }
declare s[0..9] of {-32768..32767}	S[0]=S[1]=S[2]=S[3]=S[4]= S[5]=S[6]=S[7]=S[8]=S[9]={ \emptyset }
n ← #10	N={10}

<pre> ; INPUT LOOP </pre>	<p>Assuming for this example that the for statement always processes at least once. Although not true for C, it is for some languages. The next examples assume the normal interpretation of C's for statement.</p>
<pre> i ← #0 label loop: bound #0,i,#9 on error e call input to s[i] i ← i + #1 on overflow v b ← i < n if b then goto loop else endif </pre>	<pre> I={0} Pass 0 Pass 1 ... Pass 9 I={0} I={1} I={9} Bound is never violated, so may be removed S[i]={-32768..32767} ... I={1} I={2} I={10} {1}..{10} are subsets of R_i, so overflow may be removed. B={T} B={T} B={F} By constant replacement: b ← i < #10 B={T} B={T} B={F} V={I={0},B={T},...} ... </pre>
<pre> ; SORT LOOP OUTER i ← #0 outertemp ← n outertemp ← outertemp - #1 on overflow v label loopouter: </pre>	<pre> I={0} Outertemp={10} Outertemp={9} {9} is a subset of R_{Outertemp}, so remove overflow. </pre>
<pre> ; SORT LOOP INNER j ← #1 label loopinner: </pre>	<pre> J={1} </pre>
<pre> bound #0,i,#9 on error e bound #0,j,#9 on error e </pre>	<pre> I={0}..{8} Bound is never violated, so may be removed J={1}..{9} Bound is never violated, so may be removed </pre>
<pre> b ← s[i] > s[j] if b then bound #0,i,#9 on error e temp ← s[i] </pre>	<pre> S[i]={-32768..32767}, S[j]={-32768..32767} B_T={S[i]={-32767..32767}, S[j]={-32768..32766}} B_F={S[i]={-32768..32767}, S[j]={-32767..32767}} B={T,F} I={0}..{8} Bound is never violated, so may be removed. Temp={-32767..32767} </pre>

<pre> bound #0,j,#9 on error e </pre>	<pre> J={1}..{9} <-- 10 times Bound is never violated, so may be removed. </pre>
<pre> bound #0,i,#9 on error e </pre>	<pre> I={0} Bound is never violated, so may be removed. </pre>
<pre> s[i] ← s[j] bound #0,i,#9 on error e </pre>	<pre> S[i]={-32768..32766} I={0} Bound is never violated, so may be removed </pre>
<pre> s[j] ← temp else endif </pre>	<pre> S[j]={-32767..32767} </pre>
<pre> ; END INNER LOOP j ← j + #1 on overflow v </pre>	<pre> J={2}..{10}, {2}..{10}, {2}..{10}, {2}..{10}, {2}..{10} are subsets of R_i, so overflow may be removed. J={2}..{9}, N={10} B_T={J={0}..{9},N={10}} B_F={} B={T} On the pass with J=10, J={10}, N={10} B_F={J={10},N={10}} B_T={} B={F} </pre>
<pre> if b then goto loopinner else endif </pre>	
<pre> ; END OUTER LOOP i ← i + #1 on overflow v </pre>	<pre> I={1}..{9} {1}..{9} are subsets of R_i, so overflow may be removed. I={1..8}, Outertemp={9} B_T={I={1..8},Outertemp={9}} B_F={}, B={T} I={9}, Outertemp={9} B_T={}, B_F={I={9},Outertemp={9}}, B={F} </pre>
<pre> if b then goto loopouter else endif </pre>	
<pre> ;OUTPUT LOOP i ← #0 </pre>	<pre> I={0} Pass 0 Pass 1 ... Pass 9 </pre>
<pre> label oloop: bound #0,i,#9 on error e </pre>	<pre> I={0}..{9} Bound is never violated, so may be removed </pre>
<pre> call output with s[i] i ← i + #1 on overflow v </pre>	<pre> S[i]={-32768..32767} ... I={1}..{10} {1}..{10} are subsets of R_i, so overflow may be removed. </pre>

<pre> b ← i < n if b then goto oloop else endif </pre>	<pre> B={T}...B={T}, final pass: B={F} less I than #10 to b by constant replacement B={T}...B={T}, final pass: B={F} $\mathcal{V} = \{I=\{0\}, B=\{T\}, \dots\} \dots$ </pre>
---	--

In the above, all array bounds have been checked at compile-time. No boundary violations occur and all boundary checks can be removed safely. The code associated with the boundary check would be removed by more standard optimizations as it would then be dead code.

All additions and subtractions yield results within the types to which they are assigned. Therefore, all overflow checks can be removed safely.

No references are made to variables with possibility sets containing \emptyset , so no uninitialized variable references occur.

All error conditions above have been italicized and safely removed. It is safe to say that if the input and output routines are safe, then this routine is safe from the exceptions noted in this thesis.

Example 2

Faulty Average of Input Values

Original Code from C:

```
int n;
int i;
int a[10];
int sum;
int avg;

n=input();

for(i=0;i<n;i++)
    a[i]=input();

sum=0;

for(i=0;i<n;i++)
    sum+=a[i];

avg=sum/n;
output(sum); // Output the sum of the input values
output(avg); // Output the mean average of the input values
output(a[0]); // Output the first value from the list
output(a[n-1]); // Output the last value from the list
```

PSS Analysis:

declare n of {-32768..32767}	N={ \emptyset }
declare i of {-32768..32767}	I={ \emptyset }
declare a[10] of {-32768..32767}	A[0]=A[1]=...=A[9]={ \emptyset }
declare sum of {-32768..32767}	Sum={ \emptyset }
declare avg of {-32768..32767}	Avg={ \emptyset }
call input to n	N={-32768..32767}
i ← #0	I={0}
label loop1:	
b ← i >= n	0: B _T ={N={0}}, B _F ={N={1..32767}}, B={T,F}
	1: B _T ={N={0..1}}, B _F ={N={2..32767}}, B={T,F}
	2: B _T ={N={0..2}}, B _F ={N={3..32767}}, B={T,F}
	...
	32767: B _T ={N={0..32767}}, B _F ={}, B={F}

if b then	$\{0\}.. \{32766\}$, $B=\{T,F\}$, so process both T and F
	$\{32767\}$, $B=\{F\}$, so process only false (to endif)
bound #0,i,#9 on error b	$\{0\}.. \{9\}$: Passes $\{10\}.. \{32766\}$: Fails <i>Boundary violation is possible.</i>
call input to a[i]	$A[-32768..32766]=\{-32768..32767\}$
$i \leftarrow i + \#1$ on overflow b	$\{1\}.. \{32767\}$ <i>Overflow is not possible, so may be removed.</i>
goto loop1	
else	
endif	
	$I=\{0..32767\}$ $A[0..32766]=\{-32768..32767,\emptyset\}$ $A[32767]=\{\emptyset\}$ Note that $A[10..32767]$ are not declared.

Since the else clause is missing, its possibility sets are effectively the same as if it were present, but null. In this case, the else clause is the same as the sets before the if clause, other than the variable i.

If the rules of analysis allow all processing on a sequence to stop if a definite error occurs, then the above would stop processing as soon as $I=\{10\}$ occurred at the bound statement. All T and F possibilities would then immediately be unioned together at the endif. In this case, the analysis would proceed much more quickly.

sum \leftarrow #0	Sum= $\{0\}$
$i \leftarrow$ #0	$I=\{0\}$
label loop2:	
$b \leftarrow i \geq n$	0: $B_T=\{N=\{0\}\}$, $B_F=\{N=\{1..32767\}\}$, $B=\{T,F\}$ 1: $B_T=\{N=\{0..1\}\}$, $B_F=\{N=\{2..32767\}\}$, $B=\{T,F\}$ 2: $B_T=\{N=\{0..2\}\}$, $B_F=\{N=\{3..32767\}\}$, $B=\{T,F\}$... 32767: $B_T=\{N=\{0..32767\}\}$, $B_F=\{\}$, $B=\{F\}$
if b then	$\{0\}.. \{32766\}$, $B=\{T,F\}$, so process both T and F $\{32767\}$, $B=\{F\}$, so process only false (to endif)
bound #0,i,#9 on error b	$\{0\}.. \{9\}$: Passes $\{10\}.. \{32766\}$: Fails <i>Boundary violation is possible.</i>
sum \leftarrow sum + a[i] on overflow ov	0:Sum= $\{-32768..32767\}$ 1:Sum= $\{-32768..32767\}$, possible overflow ... 32767:Sum= $\{-32768..32767\}$, possible overflow <i>Overflow is possible for all but first pass.</i>
$i \leftarrow i + \#1$ on overflow ov	$\{1\}.. \{32767\}$
goto loop2	

endif	
avg ← sum	Avg={-32768..32767}
avg ← avg / n on error z	N={-32768..32767}, Avg={-32768..32767}
	0 ∈ N, so Division by zero error is possible.
call output with sum	
call output with avg	
bound #0,#0,#9 on error b	0 is within 0..9, so within bounds. Always within bounds, so may be removed.
call output with a[0]	∅ ∈ A[0], so Possible read of uninitialized variable.
temp ← n	Temp={-32768..32767}
temp ← temp - #1 on overflow ov	Temp={-32768..32766}
	Overflow is possible (-32768-1=-32769 ∉ R _{Temp})
bound #0,temp,#9 on error b	Temp={-32768..32766} is not a subset of {0..9}. Boundary violation is possible.
call output with a[temp]	There exists a[Temp] with ∅ as member, Possible read of uninitialized variable.

The above example has C code which looks reasonable, yet which has many possibilities for run-time errors. The first problem yielding the errors is that the input value to n determining the number of entries is unbounded by the code. Any value including zero, even negative numbers, may be supplied, yet the code does not check for reasonable values. When the average is finally calculated at the end, the division can cause problems if the number of values is zero (0).

The output of a[0] with no values input would yield garbage. The output of the supposedly last value in the list is worse; it could conceivably reference a negative index as well as garbage.

By calculating the sum of the values in a single variable of the same type as the input, the sum could easily overflow causing incorrect results.

Example 3

Corrected Average of Input Values

The code below performs the same function as example 2, but without many of the associated problems.

Original code from C:

```
int n;
int i;
int a[10];
long sum;
int avg;

n=input();

if((n>=1)&&(n<=10))
{
    sum=0;
    for(i=0;i<n;i++)
    {
        a[i]=input();
        sum+=a[i];
    }

    avg=sum/n;
    output(sum); // Output the sum of the input values
    output(avg); // Output the mean average of the input values
    output(a[0]); // Output the first value from the list
    output(a[n-1]); // Output the last value from the list
}
```

PSS Analysis:

```
declare n of {-32768..32767}          N={∅}
declare i of {-32768..32767}          I={∅}
declare a[0..9] of {-32768..32767}    A[0]=A[1]=...=A[9]={∅}
declare sum of {-2147483648..2147483647} Sum={∅}
declare avg of {-32768..32767}        Avg={∅}

call input to n                        N={-32768..32767}

b1 ← n ≥ #1                            B1_T={N={1..32767}}, B1_F={N={-32768..0}}
b2 ← n ≤ #10                           B2_T={N={-32768..10}}, B2_F={N={11..32767}}
```

b3 ← b1 and b2	B3 _T ={N={1..10}}, B3 _F ={N={-32768..0,11..32767}}
if b3 then	
	N={1..10}
sum ← #0	Sum={0}
i ← #0	I={0}
label loop1:	
b ← i ≥ n	0: B _T ={}, B _F ={N={1..10}}, B={F}
	1: B _T ={N={1}}, B _F ={N={2..10}}, B={T,F}
	...
	9: B _T ={N={1..9}}, B _F ={N={10}}, B={T,F}
	10: B _T ={N={1..10}}, B _F ={}, B={T}
if b then	Pass 10 avoids goto-backwards, going directly to endloop1.
bound 0,i,9 on error b	I={0}..{9}
	<i>Boundary is never violated, so may be removed.</i>
call input to a[i]	A[i]={-32768..32767}
sum ← sum + a[i]	
on overflow v	0: Sum={-32768..32767}
	1: Sum={-32768*2..32767*2}
	..
	9: Sum={-32768*10..32767*10}
	<i>Always within R_{Sum}, so overflow may be removed.</i>
i ← i + #1 on overflow v	I={1}..{10}
	<i>Never overflows, so may be removed.</i>
goto loop1	
else	
endif	
	Assume temp is a long.
temp ← sum	Temp={-327680..327670}
temp ← temp / n on error e	N={1..10}, Temp={-327680..327670}
	<i>N never contains 0, so on error may be removed.</i>
cast temp to avg	Avg={-327680..327670}, not within R _{Avg} , so
	<i>Possible loss of information.</i>
	Avg={-32768..32767}
call output with sum	
call output with avg	
bound #0,#0,#9 on error b	<i>Always within bounds, so may be removed.</i>
call output with a[0]	
temp ← n	Temp={1..10}
temp ← temp - #1	
on overflow v	Temp={0..9}
	<i>Never overflows, so may be removed.</i>
bound #0,temp,#9 on error b	Temp={0..9},

Always within bounds, may be removed.

call output with a[temp]

endif

In the above version, the only remaining warning is a possible loss of information remains from casting a long to an int. It cannot determine that the loss of information cannot actually occur, so it warns about it regardless.

The only time that the long could be 327680 is if 32768 is input 10 times; in that case n would equal 10, and the division would bring the result into the range of int. The program cannot correlate the n value with the possible ranges; it must assume a division by 1 is possible even when the sum was 327680. Note that this is the safe alternative. Giving a superfluous warning is better than missing a warning.

VITA 

Brian Joseph Sullivan

Candidate for the Degree of

Master of Science

Thesis: POSSIBILITY SET SEMANTICS: ERROR DETECTION AND
CODE OPTIMIZATION

Major Field: Computer Science

Biographical:

Education: Graduated from Broken Arrow High School, Broken Arrow, Oklahoma, in May, 1990; received Bachelor of Science degree in Computing and Information Science from Oklahoma State University, Stillwater, Oklahoma in December, 1993. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 1996.

Experience: Employed at Creative Labs, Inc. as a technical support agent, Lead Agent, and Agent in Charge of the Sound group, 1993 to 1996.