THE UNIVERSITY OF HULL

Mutation Analysis of Dynamically Typed Programs

Being a Thesis submitted for the Degree of

Doctor of Philosophy

in the University of Hull

By

Nabil Abu Hashish

B.Sc., Yarmouk University, Jordan, 1985

Master of Computer Engineering, METU, Turkey, 1988

Department of Computer Science

December 2013

Abstract

The increasing use of dynamically typed programming languages brings a new challenge to software testing. In these languages, types are not checked at compile-time. Type errors must be found by testing and in general, programs written in these languages require additional testing compared to statically typed languages.

Mutation analysis (or mutation testing) has been shown to be effective in testing statically (or strongly) typed programs. In statically typed programs, the type information is essential to ensure only type-correct mutants are generated. Mutation analysis has not so far been fully used for dynamically typed programs. In dynamically typed programs, at compile-time, the types of the values held in variables are not known. Therefore, it is not clear if a variable should be mutated with number, Boolean, string, or object mutation operators.

This thesis investigates and introduces new approaches for the mutation analysis of dynamically typed programs. The first approach is a static approach that employs the static type context of variables to determine, if possible, type information and generate mutants in the manner of traditional mutation analysis. With static mutation there is the danger that the type context does not allow the precise type to be determined and so type-mutations are produced. In a type-mutation, the original and mutant expressions have a different type. These mutants may be too easily killed and if they are then they represent incompetent mutants that do not force the tester to improve the test set. The second approach is designed to avoid type-mutations. This approach requires that the types of variables are discovered. The types of variables are discovered at run-time. Using type information, it is possible to generate only type-correct mutants. This dynamic approach, although more expensive computationally, is more likely to produce high quality, difficult to kill, mutants.

To my beloved Wife: Sahar Atyeh (Abu Hashish)

To my children: Majd, Deya, Mohanad, Ahmad, Juman, and Zaid

To my great Mother

Acknowledgement

I am deeply indebted to my supervisor, Dr. Leonardo Bottaci, who has encouraged me to swim in the ocean of software testing. This research would not have been completed without his continued help, support, and guidance throughout all stages of this research.

I would also like to thank Prof. Yiannis Papadopoulos and Dr. Chandra Kambhampati for their comments and helpful supports. In addition, thanks to all administrative staff in the department of Computer Science, in particular, Colleen Nicholson, Helen El-Sharkawy and Joan Hopper.

I would like to acknowledge Isra Private University, Jordan and the department of computer science, University of Hull for giving me this opportunity and for financial support during my last year of study.

Finally, many thanks should also be extended to my mother, wife and children for their support and patience.

Publications

Nabil Hashish, Leonardo Bottaci (2009). ‘Language Constructs for Generalising Unit Test: Research Proposal’, *TAIC-PART '09 Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques IEEE Computer Society*, Washington, DC, USA.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Research Scope and Context

Software testing (Bertolino 2003) is the process of executing a program with inputs and checking the results. It is an essential part of establishing the quality and reliability of software systems (Hierons 2002). In practice, testing cannot be exhaustive (i.e., exercising the program on every input) and so the main objective of testing software is to find the most faults for a given amount of testing effort. If testing is not exhaustive then some approach to test selection must be used (Yanping Chen et al. 2002). Software test selection approaches are traditionally divided into black-box testing (functional testing) and white-box testing (structural testing) (Sofokleous and Andreou 2008). The black-box approach treats the program as a black box and tests are selected from the specification without using knowledge of the implementation. On the other hand, the white-box approach allows the testers to view the structure of the program under test to examine if their tests cover the code. The tester chooses test case inputs in order to achieve some given structural coverage criteria, e.g. execute every statement or branch in the program, evaluate all logical expressions to both `true` and `false`, execute inequalities at their boundaries, and so on.

Recently, the widespread use of dynamically typed programming languages (Laurence Tratt 2009), JavaScript for example, especially in internet application, imposes a new challenge to software testing. Dynamically typed languages are usually used for scripting programs that do not check or enforce type-safety at compile-time. The JavaScript programming language is typical among dynamic languages (Mikkonen and Taivalsaari 2007). JavaScript is widely used to script html web pages but has also been used to implement large frameworks, e.g. JQuery (Goodman and Morrison 2007).

In dynamically typed programs, values have types but variables do not. The type information of values is available at run-time; no type-checking is done until run-time. Even then, in some programming languages, e.g. JavaScript, extensive automatic type conversion is performed to allow operations to continue and avoid throwing up exceptions. In these languages, many type errors are not detected by the compiler. If the errors are to be found then they must be found during testing. This suggests that dynamic programs require more testing than the equivalent statically typed programs.

1.1.1 Mutation Analysis

Mutation analysis is a technique for assessing the quality of a test set. It is not easy to determine that a program has been well tested. A program may pass every test in a given test set but the program may still contain many faults. Mutation analysis (or mutation testing) is a white-box fault-based coverage criterion proposed for software testing (Hamlet 1977, Demillo and Lipton 1978). The basic idea is to deliberately insert faults into the program under test and check if the faults are detected by any of the tests in the test set. If the tests cannot detect the faults then additional tests are required.

In practice, the process of inserting faults is approximated by a process in which small changes are made to the program under test. The primitive elements of the program are systematically modified or replaced and the set of programs so generated are called mutants. Typically, only a single element of the original program is modified to create a mutant. Syntactically, these mutants closely resemble the original program and are in the neighbourhood of the original program (Untch 2009).

Figure 1.1 shows an example of a program under test and a possible mutant program. Assume the variable `i` holds a number, then the original program may be modified by inserting, for example, the `abs()` function (i.e., modify the operand `i` to `abs (i)`) to produce a mutant program.

```
 //Original Program
function f(i, j) {
  var max;
  if (i > j) {
    max = i;
  }
  else {
    max = j;
  }
  return max;
}
```

```
//mutant Program
function f(i, j) {
  var max;
  if (abs(i) > j) {// i becomes abs(i)
    max = i;
  }
  else {
    max = j;
  }
  return max;
}
```

Figure 1.1: An example of a mutant program generated by inserting the `abs()` function at an occurrence of the variable `i`.

Consider, for example, the execution of a specific test on the original program under test and also on the mutant program. It is not necessary to know the correct output for the test; instead the outputs are compared. If the output produced by the original differs from that of the mutant then at least one of the two programs is incorrect. The mutant is said to be distinguished and such a test is considered informative. In the example program the input `(i = -1, j = 0)` will distinguish the mutant because the original program outputs 0 and the mutant outputs `-1`. If in contrast, the outputs do not differ, both programs may be correct or incorrect. No more is known about the programs than before the execution of the test; consequently the test is considered uninformative. In this case, the mutant is said to be alive.

Mutants are created automatically and it is possible, of course, that some mutants cannot be distinguished from the original program. For example, consider a different program to the program of Figure 1.1, produced by inserting the code `i = i * i;` in front of the if-statement, i.e.

```
i = i * i;
if (abs(i) > j) {  //abs(i) equivalent to i, i always positive
```

then the occurrence of `i` in the if-statement condition is always positive in which case `i` and abs(`i`) always have the same value.

Alive mutant should force the tester to identify either that the two programs are equivalent, i.e., syntactically different but semantically the same, or that the test set is insufficient to detect the change in the mutant program. In this case, the test set should be enhanced by adding new test cases to kill the live mutants.

Mutation analysis relies on the competent programmer hypothesis, i.e. the programmer is assumed to be competent (Acree et al. 1979). Under this hypothesis, the competent programmer produces programs that are either correct or differ from a correct program by a small fault. To test such a program, it is sufficient to establish that the given program does not contain any small fault. In order to assess the fault finding power of the given test set, the program and the neighbourhood programs are all executed on this test set to determine if all nonequivalent programs can be detected.

The idea underlying mutation analysis is that the modifications or changes made by mutation analysis depict faults that competent programmers may introduce. A set of transformation rules (or mutation operators) are applied to a program under test to create a set of variants of the original program, called mutants, each containing a small single syntactic change. Typically, these changes are introduced by modifying operands or replacing operators and operands in the program.

1.2 Motivation

Mutation analysis has been applied to programs written using statically typed procedural programming languages (Demillo et al. 1988, King and Offutt 1991, Offutt et al. 1996a), with some work on the mutation of object-oriented programs (Ma et al. 2005). In all cases, the languages have been statically typed. For statically typed programs, mutation analysis has been shown to be an effective testing method (Wong 1993), but it is not clear how it should be applied to dynamically typed programs.

Mutation analysis depends mainly on the replacement or modification of program elements. Typical mutation operators modify program elements (typically operands or operators) by replacement, insertion or deletion operators. The availability of type information is essential to guide the mutation analysis and prevent generation of type-incorrect expressions within mutants. The insertion of the `abs()` function at a variable `i`, for example, is allowed only if the type of `i` is a number. In a statically typed language, a type-incorrect expression is an error that is detected at compile-time. A mutant that does not compile is clearly not a competent program and so obviously, mutations of statically typed programs should not contain type-incorrect expressions.

In dynamically typed programs, the variables have no-type, only values have type. This raises the problem, for example, of how to ensure that `i` is not mutated to `abs(i)` when the value held in `i` is not a number. One approach towards the lack of type information is to ignore type or equivalently to consider all variables to be of the same type. For example, instead of attempting to apply the `abs()` operator to only number variables, it is applied to all variables.

In a statically typed language, `abs(“-23”)` is type-incorrect and the program will not compile. In a dynamically typed program, such as JavaScript, for example, there are extensive automatic type conversions that allow what would be considered type-incorrect expressions to return a value without stopping the execution of the program. In the case of `abs(“-23”)` there would be an attempt to convert the string to a number. In the case of `“-23”` it would return the value `-23` and 23 would then also be the result of `abs(“-23”)`. In the case of `abs(“hello”)` an attempt to convert the string to a number would fail and return the value `NaN` (Not a Number). This would then also be the result of `abs(“hello”)`.

Using this simple, indiscriminant approach, mutants can be defined statically in the manner of statically typed programs. Mutations are made wherever they are syntactically valid by applying a set of mutation operators to make a small change to the syntax of a program. This approach would be relatively simple to implement but some of the mutants produced will be “type-mutations”, i.e. not only does the mutated expression hold a different value to the original expression, it is also a different type than the original type. The mutation of `"hello"` to `abs("hello")` is an example of a type-mutation. The original expression has type string but the mutated expression has a value of `NaN` and type number (in JavaScript the type of `NaN` is number). Compared to a same type-mutation, it is expected that a type-mutation will be easier to kill. If a mutant is too easily killed then it has little value in forcing the tester to enhance the test set.

The term “type-mutation” is introduced in this thesis to describe the mutation produced when the type of a value in the original program and the mutant at the mutated expression is different. In general, it might be expected that type-mutations are less difficult to kill, i.e. produce a larger change in the program behaviour than traditional value mutations. A mutant that is too easily killed, i.e. incompetent, does not force the tester to enhance the test set. Hence it is not clear if type-mutations are useful. The extent to which type-mutations are incompetent is one of the questions that this thesis investigates.

In any case, the simple static application of mutation operators will lead to an increase in the number of mutants compared to the typed application of mutation operators. For example, assume that there are 8 number mutation operators, 2 string mutation operators and 3 Boolean mutation operators. Consider a program with one number variable, one string variable and one Boolean variable. This makes a total of $1 * 8 + 1 * 2 + 1 * 3 = 13$ mutations when mutations are typed. If mutations are not typed then every mutation is made to every variable and this makes a total of $3 * 8 + 3 * 2 + 3 * 3 = 39$ mutations when mutations are not typed.

The no-typed mutation will result in a greater number of mutations that would be generated if type information were used. The increase in number of mutants will increase the cost of mutation analysis. It is not known if this increase in cost will be significant because the cost of a mutant is dependent on the number of tests required to kill it. The equivalent mutants, the mutants that cannot be killed, are the most expensive because they have been executed with every test and they also require human inspection. The extent of the additional cost is a research question for this thesis.

This extra cost may be cost effective if the additional mutants force the tester to enhance the test set. Whether the additional mutants are cost effective is one of the questions that this thesis investigates.

If type-mutants are likely to be incompetent (too easily killed) it raises the question of how to avoid generating type-mutants. How, for example, will it be possible to avoid mutating the variable `i` to `abs(i),` when `i` is not a number if it is not known whether `i` is a number type? This is the question of how to apply the mutation operators in a type-sensitive manner.

One way to try to discover the type of a variable is to consider the context in which the variable occurs. The context can be used as a heuristic to assume a type. This can be done statically but the assumption may be wrong. In order to apply mutation analysis in a typed manner, the type of the elements in the program under test should be discovered. A simple method of discovering the type of a variable is to test the type of the variable at run-time. Consider, for example, the type-sensitive implementation of the `abs()` mutation operator below:

```
absTypeSensitive(i) {
  if (typeof(i) == ”number”) {
    return abs(i);
  }
  else {
    return i;
  }
}
```

This method discovers the type of the value stored in a variable at run-time. Discovering the type of the value before each time it is mutated is likely to be inefficient because inside a loop the type of `i` may be discovered many times in situations in which there is no possibility that the type can change. It would also be tested by all the number mutation operators. For example, it would be tested for every number mutation of `i`, e.g. `abs(i)`, `add1(i)` (i.e. add 1 to i), `sub1(i)`, etc. Clearly, the values held in a specific occurrence of a variable `i` will not change according to the function that inputs those values.

Since the type of a given variable does not vary with the mutant, it varies only with the program input (i.e., the test), the dynamic mutation approach presented in this thesis executes each test case on the original program under test in order to discover the types of values and variables. This is done before generating mutants. After a test has been executed, the types of values and variables can be known for that test. As a result, the generation of mutants for that test can be done using that type information. In this way the type information is discovered once only.

Figure 1.2 compares the different approaches for generating mutants of statically and dynamically typed programs. For dynamically typed programs, test cases are required in order to define values at variables. The types of these values are then used to define type-specific mutations at those variables. Such mutants are generated by modifying the value of operands with the type compatible mutation operators, replacing a variable with other type compatible variables or constants, and replacing each operator with other type compatible operators in the same way as in traditional mutation analysis.

Statically typed mutation

Program + mutation operators ----------------------→ Set of mutants

Dynamically typed mutation

Program + test cases + mutation operators -------→ Set of mutants

Figure 1.2: The different approaches used to generate mutants of statically and dynamically typed programs.

In dynamically typed mutation, a program produces approximately the same number of mutants as a statically typed program if every variable in the program takes a value of only one type. In addition, when the types of all program variables are known i.e., they have been discovered by execution of the tests, the dynamic mutation analysis is equivalent to statically typed mutation analysis. There is a risk, however, with dynamic mutation analysis, that the tests do not exercise the program fully and not all the types are discovered.

The motivation for this research is to develop mutation analysis approaches for the mutation analysis of dynamically typed programs. The work in this thesis is based on the JavaScript language (Crockford 2008), but the result is expected to apply to programs written in a similar language: Python (Linda Dailey 2007), PhP, Ruby, etc.

1.3 Research Aim

In order to make mutation analysis applicable effectively to dynamically typed programs, the thesis introduces two new approaches to mutation analysis. These two approaches are called static and dynamic mutation approaches. In the static approach, the mutation is done statically as in traditional mutation analysis. With the static approach there is the danger of generating type-mutants but in order to reduce type-mutation, the context in which the mutated element occurs is used to eliminate some redundant mutations and also to heuristically assume a type for a variable in an expression. In the dynamic approach, the type of an element is discovered at run-time and the definition of mutants is then performed with the availability of type information. The dynamic approach allows mutants to be defined in a type-sensitive manner.

1.3.1 Research Questions

Research questions that will guide the research in comparing the two approaches and evaluate the efficiency of each approach for testing dynamically typed programs are given below:

Q1: Using the static mutation approach, what proportion of the mutants are type-mutants? This is the same question as how many mutants are not generated in the dynamic method.

Q2: Using the static mutation approach, what proportion of the type-mutants is incompetent? Are type-mutants more incompetent than mutants where the different values in the original and mutant programs have the same type?

Q3: What is the cost reduction by not generating type-mutants? In other words, how does the cost of the static method compare with the dynamic method?

Q4: How does the choice of static or dynamic mutation affect the number of equivalent mutants generated?

Q5: Which is the most cost effective method?

1.4 Contributions

The difficulties involved in mutation testing of dynamically typed programs refer to the fact that the type information is unknown prior to the program run-time. Mutation testing has been shown to be an effective test coverage criterion but it has not yet been applied to dynamically typed programs.

The thesis investigates and introduces two new approaches for the mutation analysis of dynamically typed programs, which have not been done before. Firstly, a static approach to generate mutants of dynamically typed programs has been developed. This approach does not use run-time type information but does exploit some static information. The type context in which a program element occurs can be used to eliminate some type-mutations and to make heuristic assumptions about the type of the element. This approach is based on making only syntactic changes to the program under test. Secondly, a dynamic approach is adopted to generate type-correct mutants at run-time. This thesis argues that mutation analysis of dynamically typed programs can also be established at run-time by executing test cases against the program under test. The aim is to discover type information at program run-time to avoid type-mutations and, therefore, some of the incompetent mutants. The two approaches are evaluated and compared for the mutants produced for the selected sample programs. The two approaches are suitable for the mutation analysis of dynamically typed programs, but the dynamic approach is shown to be more effective than the static approach.

Moreover, the thesis investigates and answeres the research questions about the effectiveness and the performance of these approaches. The research of this thesis is mainly based on the JavaScript language but these approaches are applicable to the mutation analysis of other similar high-level dynamically typed languages.

1.5 Thesis Outline

The thesis is organized as follows:

Chapter 2 of this thesis presents background information on software testing and mutation analysis of statically typed programs. It also presents an overview of the various techniques that have been used to improve the performance of mutation analysis and reduce the cost and time of generating, compiling and running mutants. The work related to mutation analysis of dynamically typed programs is also reviewed.

Chapter 3 briefly introduces the difference between statically typed programs and dynamically typed programs. It also highlights the typing system and different features of dynamically typed languages, e.g. JavaScript. In addition, the capability of this language to perform implicit type conversion and the use of arithmetic and logical operators in different context are investigated. This is useful to understand the effect of converting the type of an element to another type during the program execution.

Chapter 4 presents a static approach to mutation analysis of dynamically typed programs. It discusses how static mutation analysis can be performed for dynamically typed programs. In addition, the type context of variables in expressions is used to heuristically assume types of these variables to avoid some type-mutations. This chapter, also, collects a set of mutation operators similar to the traditional mutation operators used in the mutation of statically typed programs and proposed new operators that are required for the mutation analysis of dynamically typed programs.

Chapter 5 introduces a dynamic approach for the mutation analysis of dynamically typed programs. This approach requires that the type information is to be discovered during program execution time. This chapter, also, investigates the type discovery methods that can be used effectively for this approach to reveal type-correct information and how the mutation operators are applied in a typed manner at run-time to avoid type-mutations.

Chapter 6 presents an empirical investigation to evaluate the static and dynamic approaches introduced for dynamically typed programs. It also investigates the minimal test sets and the quality of test sets that are mutation adequate in order to reduce the cost of mutation analysis that produced that mutation adequate test set for a dynamically typed program.

Chapter 7 revisits the research questions and summarizes the empirical results obtained of seven JavaScript programs. This is necessary to check the effectiveness of static and dynamic approaches for the mutation analysis of dynamically typed programs introduced in this thesis.

Chapter 8 concludes by summarizing the results of this thesis, contributions of this thesis, limitation and suggestions for future work.

Chapter 2: Mutation Analysis

2.1 Introduction

Mutation analysis (also referred to as mutation testing) is a method used to measure how good a given set of test cases are in detecting potential faults in the program under test (Amman and Offutt 2008). Mutation testing is not directly concerned with testing the program to find faults. Instead of this, it tests the test cases by measuring how good these tests are at detecting changes introduced into the program. If the test cases cannot detect the changes then new tests are added to enhance the test set. This results in improved program testing except equivalent mutants.

The mutation analysis assumption is that the program under test is written by a competent programmer. Mutation testing targets faulty programs (called mutants) that are close to a correct version of the program. A set of synthetic faulty programs (mutants) is generated by introducing small syntactic changes to the program under test. The aim is to produce a test set that shows that the program under test is not equal to any of these mutants.

Mutation testing has been used to test software at various levels, including unit testing, integration testing, system testing and the specification testing (Jia 1996). Unit testing is the most common to mutation analysis (Jia 2009). A unit is a software component that cannot be subdivided into other components. In object-oriented programming, these units typically are classes and methods or functions. Approximately 65% of all bugs can be caught in unit testing (Beizer 1990). Unit testing may be structural (white box) or functional (black box).

2.2 Mutation and Mutant Generation

Usually, the changes are introduced into a program using mutation operators. The purpose of the mutation operators is to generate the set of competent programs that a competent programmer might produce. The collection containing the original program and the mutants is known as the program neighbourhood (Untch 2009).

One way in which mutation operators can be defined is to consider the ways in which the competent programmer may make an error in an expression. The programmer may use the wrong operator; this suggests operator replacement mutations. The programmer may include an operator that should not be present; this suggests operator deletion mutations. The programmer may omit an operator that should be present; this suggests operator insertion mutations. The programmer may use the wrong variable or literal, which suggests operand replacement mutations.

```
//Original Program P

function min(i, j) {
  var m;
  m = 0;
  if (i < j) {
    m = i;
  }
  else {
    m = j;
  }
  return m;
}
```

```
//mutant Program m

function min(i, j) {
  var m;
  m = 0;
  if (i > j) {  // mutant
    m = i;
  }
  else {
    m = j;
  }
  return m;
}
```

Figure 2.1: An example of a mutant produced from a small syntactic change to the original program.

Consider, for example, the program given in Figure 2.1. Assume that this program has been written by a programmer. The programmer may make a mistake in the relational operator in a condition. To check for this, a copy is made of P, called `m` in Figure 2.1, and in this copy the `<` symbol is replaced with `>`. The tester should now find a test input that produces different outputs when executed on `P` and m. This mutant forces the tester to introduce a test which compares the results of a program using < with another program that uses >. Both of these programs cannot be correct if there is a test that produces different outputs.

This way of thinking about mutation operators leads to the idea that a mutation operator should be a minimal change to the original program produced by insertion, replacement or deletion of a single operand or operator. This kind of small syntactic change defines the program neighbourhood.

Another way in which mutation operators can be defined is to consider the coverage that should be achieved by the test set. Mutation operators force the tester to cover certain elements of the program under test. For example, Mothra (King and Offutt 1991) includes a statement deletion operator. This is not because the competent programmer is likely to make a mistake and include a statement that should not be present; instead the statement deletion operator forces the tester to write a test set that has statement coverage. This is because to detect the deletion of a statement the test case must execute that statement. Another example of a mutation operator designed to enhance the test set coverage is the Mothra `zpush()` operator. This is a function that is applied to variables in the program. The mutant is killed only if the argument to `zpush()` is zero. The `zpush()` mutation forces the tester to write a test that sets each variable to zero.

This way of thinking about mutation operators allows more flexibility in the design of mutation operators than the operand and operator insertion, replacement and deletion approach. This kind of mutation operator may make a syntax change that is not small but the aim is always that the resulting program behaviour change is small. The program neighbourhood is the programs that have similar input-output behaviour to the original program. In this thesis, it is this idea of the program neighbourhood that is used when considering mutation operators.

In general, mutations can be classified as follows:

1. Operator mutations: The operator mutations are used to mutate operators in expressions. These operators include assignment, arithmetic, logical, and relational or comparison operators. Mutants are generated by replacing an operator with other compatible operators. For example, each arithmetical operator (`+`, `-`, `*`, `/`, `%`), is replaced with another arithmetical operator. Each relational operator `<` `>` `<=` `>=` `==`

`!=` can be replaced with another relational operator or a binary operator `&&` by `||`, and the assignment operator `+=` can be replaced with `-=, *=, /=, %=.` Moreover, the pre-decrement, pre-increment, post-increment and post-decrement mutations can also be mutated. For example, `++x` is mutated with `--x` (replace operator), `x (remove` operator), and `x++` (move operator from pre to post).

2. Operand mutations: Mutants can be generated by modifying or replacing the value of each operand in an expression. Mutants can be generated by replacing an operand with another operand, variable or literal, usually taken from the same program. Clearly, the context of an operand affects the possible replacements. A variable, for example, if it is on the left hand side of an assignment statement, cannot be replaced with a literal.

3. Other mutations such as `zpush()` and statement deletion designed to achieve a specific coverage criterion.

Mutants can also be classified according to whether they are insertion mutations or replacement mutations. An insertion mutation is performed by inserting a function call around an operand, e.g. `x` is mutated to `abs(x).` To check whether to perform an insertion mutation for a particular operand it is necessary only to check the local context of the operand.

A replacement mutation is performed by replacing one operator or operand with another. To check whether to perform a replacement mutation for a particular operand, it is necessary to check the local context of the operand (e.g. type of operand) and the context of the replacement operand (e.g. type of the replacement operand) for compatibility.

2.3 Mutation Testing Hypotheses

The competent programmer hypothesis was introduced by DeMillo et al. (DeMillo et al. 1978, DeMillo and Lipton 1978). It states that the programmer tends to develop programs that are correct or close to the correct program. One can assume that these programmers make only small faults which are targeted by mutation analysis. Further discussion of the competent programmer hypothesis can be found in Acree et al. work’s (Acree et al. 1979).

Although all of the produced mutants are syntactically similar to the original program, many may not be behaviourally similar. The ideal mutation operator produces only a small semantic change (Offutt et al. 96). A small syntactic change can, however, result in a large semantic change. The produced mutants may be type-incorrect and hence do not survive beyond the compilation stage. These are the so-called still-born mutants (Offutt et al. 1996). Other mutants can be distinguished from the original program by the execution of any test that executes the mutated statement; these are the so-called trivial mutants. Type-incorrect mutants and trivial mutants can easily be detected and are considered to be incompetent, because they would be produced only by an incompetent programmer.

Furthermore, mutation operators tend to produce a small but significant number of mutants that are behaviourally indistinguishable from the given program. These mutants are known as equivalent mutants (see Figure 2.2). Equivalent mutants can require a lot of effort to be devoted to identify them. The presence of both incompetent mutants and equivalent mutants has the effect of reducing the efficiency of mutation analysis and increasing the cost.

2.4 Mutation Assessment and Adequacy Measurement

In mutation testing, every test case `t` is executed against both the original program and the mutants. Test cases are assessed by checking each mutant output with that produced by the original program. If the results produced by a mutant `m` can be distinguished from that of the original program `P` (i.e., a mutant behaves differently from the original program) by at least one test case `t` in a test set `T` such that `P(t)≠ m(t)`, then the mutant `m` is said to be killed by the test case `t`. Moreover, each killed mutant can be investigated to decide whether the program or the mutant or both are incorrect. If `m` is correct then mutation testing has found a fault in `P`. If `P` is correct then the test set has been improved with a new test that checks for a potential fault. Otherwise, if `P` and `m` produce the same output, then the `T` is unable to distinguish `P` from `m` and the mutant is either an equivalent to the original program or the test cases are inadequate to kill that mutant. If both programs produce the same results for all test cases in the input domain, i.e., `P(t)= m(t)`, then the mutant is equivalent to the program and no test case will be able to kill it. If the mutant and the program are not equivalent then new test cases are added to the tests to kill that mutant.

Figure 2.2 shows an example of an equivalent mutant generated by replacing the operand `i` of the original program into the `abs(i)`. The two programs will produce identical output, because `i` equals `abs(i)` for `i >= 0`. Since no test case can kill this mutant, it must be removed and not considered in assessing the adequacy of test data set.

Original program P	*Mutant program m*

`….`
`if(i>=0) {`
`  if(j>=i) {`
`    ….`
`}`
	`….`
	`if(i>=0) {`
	`  if(j>=abs(i)) {`
	`    ….`
	`}`

Figure 2.2: An example of an equivalent mutant

A non-equivalent live mutant offers the tester an opportunity to produce a new test case and improve a test set. A test case that causes one or more mutants to fail is called effective (Offutt et al. 2006).

The test set is assessed by the mutation adequacy score (or mutation score) (Untch 2009). The mutation score is the ratio of killed mutants over the total number of non-equivalent mutants (Jia 2006).

$$\text{Mutation Score(MS)} = \frac{\text{number of killed mutants}}{\text{number of nonequivalent mutants}} \text{ X } 100\%$$

A mutation adequate test set kills all the non-equivalent mutants and has a mutation score of 100%.

2.5 Mutation Analysis Process

The process of mutation analysis is illustrated graphically in Figure 2.3. The process starts by submitting a program P, then a test set T is generated (manually or automatically) to serve as inputs to the original program P. The test set T needs to be executed against the original program P to verify that the output is correct for the test cases. If P is incorrect, an error has been found and the program should be modified and the process is restarted. If the output is correct, mutants are constructed by applying a set of mutation operators.

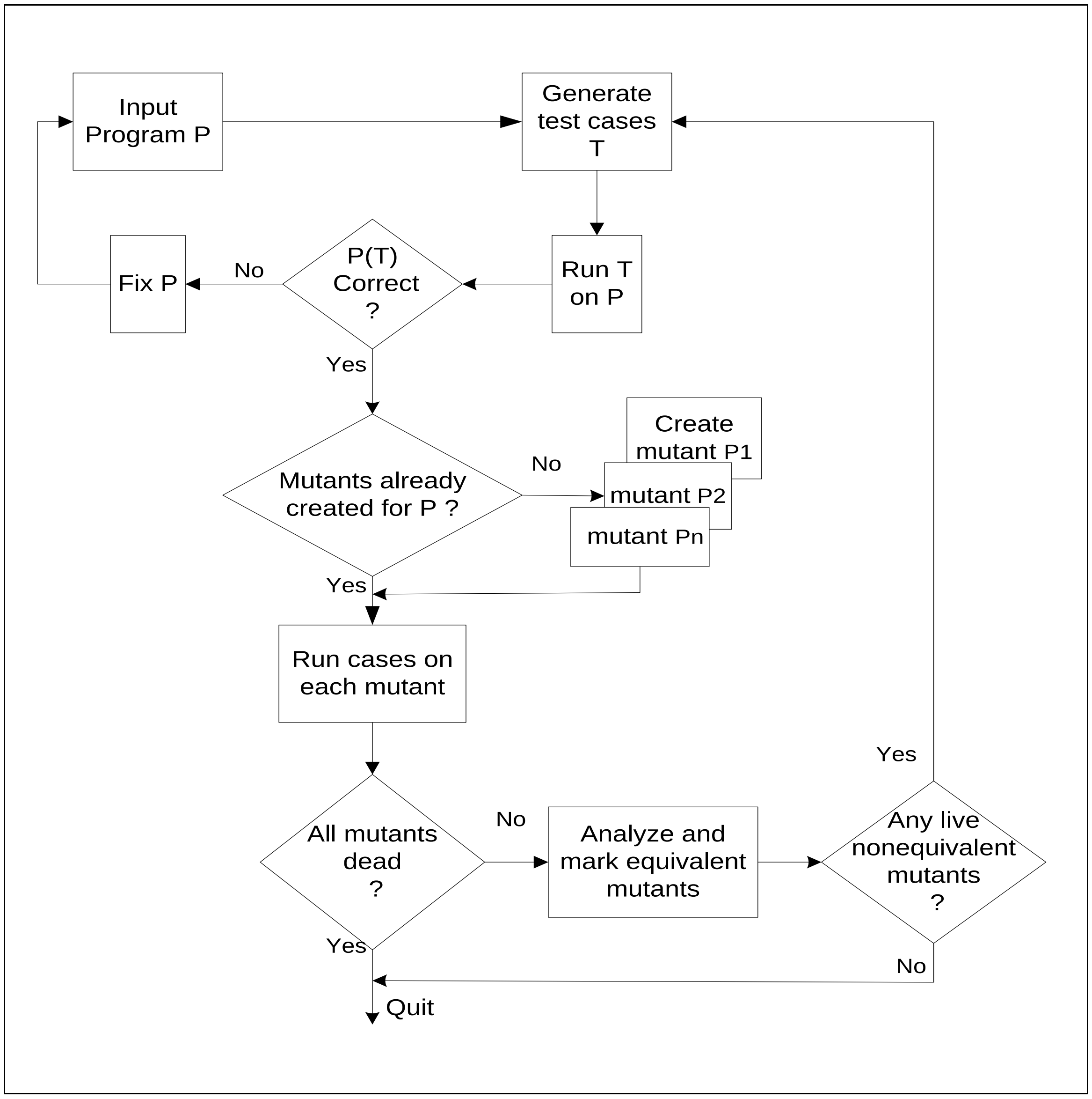


Figure 2.3: The process of mutation analysis

Each mutant `m` is run against one or more test cases in `T`. If the results of a mutant `m` differ from that of `P` on the same test case, the mutant is marked as being killed. Once killed, a mutant is not executed against any additional test cases. After each test case has been executed against each mutant, all the remaining mutants are considered live and should be analysed to determine and remove the equivalent mutants.

In practice, there may still be a few non-equivalent live mutants, but the test set is inadequate to kill them. In this case, a new test case needs to be supplied and the process is continued until a mutation adequate test set is produced.

2.6 The Problems of Mutation Analysis

To generate a large number of mutants, execute each test case on each live mutant and analyse the result requires a lot of time (Polo et al. 2009). One problem that prevents mutation analysis from becoming a practical testing technique is the computational cost of executing the enormous number of mutants against test set. The other problems are related to the required human effort incorporated in using the mutation analysis, for example, the equivalent mutant problem.

2.6.1 Equivalent Mutant Problem

Given even a small program, many mutants are produced and some of these are equivalent. Because of the undecidability of program equivalence, automatically detecting all equivalent mutants is impossible (Offutt and Pan 1997) and the detection of equivalent mutants often involves additional human effort, the result is expensive and time-consuming. How to avoid equivalents mutants is investigated in this thesis.

2.7 Mutation Cost Reduction Techniques

One way to reduce the mutation testing cost is to reduce the number of mutants generated. Many techniques have been used to reduce the number of mutants generated including, mutant sampling, mutant clustering and selective mutation. These techniques seek significant reductions without significant loss of effectiveness. In addition, other techniques are proposed to optimize the running speed of mutants. Some of these techniques include compiler-based testing, byte-code translation, weak mutation and mutant schema generation.

The process of mutant reduction can be summarized as: For a given set of mutants, M, and a set of tests T, $MST(M)$ denotes the mutation score of the test set T applied to mutants M. The mutant reduction problem can be defined as the problem of finding a subset of mutants M' from M, where $MST(M') \approx MST(M)$,i.e., the mutation score of the subset mutants is approximately equal to the mutation score of mutants (Harman et al 2009). These subset M' can be produced in different ways.

2.7.1 Random Selection x%

In this approach, a small percentage of mutants are randomly selected from the entire set possible generated mutants. The mutants are generated first as in traditional mutation analysis and then *x*% of these mutants are then chosen randomly for mutation analysis, and the remaining mutants are ignored (Wong and Mathur 1995).

2.7.2 Mutant Clustering

A clustering algorithm classifies the mutants into different clusters based on the killable test cases. Each mutant in the same cluster is guaranteed to be killed by a similar set of test cases. Only a small number of mutants are selected from each cluster to be used in mutation testing (Shamaila 2008).

2.7.3 Selective Mutation

Selective mutation has been applied to reduce the number of mutants generated, which can be achieved by reducing the number of mutation operators (Untch et al 1993). Mutation operators generate different numbers of mutants, and some mutation operators generate more mutants than others, many of which may be redundant. For example, two mutation operators of the 22 Mothra operators were found to generate about 40% to 60% of all mutants (King and Offutt 1991). In order to reduce the generated mutants, it was suggested to omit two mutation operators which generated most of the mutants (Offutt et al 1996).

Constraint mutation (Mathur et al 1991) is another type of selection strategy based on test effectiveness. It suggested using only two mutation operators: ABS and ROR. Another type of selective mutation was proposed by Mresa and Bottaci (Mresa and Bottaci 1999). Each mutation operator is assigned a score which is computed by its value and cost of detecting equivalent mutants.

Another approach on selective mutation was suggested by Namin and Anderws (Namin and Anderws 2006). They formulated the selective mutation problem as a statistical problem. They applied linear statistical approaches to identify a subset of 28 mutation operators from 108 C mutation operators.

2.8 Reduction of a Mutant Execution Cost

The computational cost of mutation testing can be minimized by optimizing the cost of each mutant execution. However, a test case that kills a mutant must satisfy the following three conditions (Offutt 93):

1. Reachability condition: The test case must cause execution to reach the mutated statement. If the test case does not reach the mutated statement then clearly the output of the mutant must equal the output of the original program because exactly the same statements have been executed.
2. Necessity condition: once the mutated statement is executed, the test case must produce a different data state in the mutant program compared to the original program. If the mutant is to be killed, some variable in the mutant program must be set to a different value compared to the same variable in the original program.
3. Sufficiency condition: the different data state at the mutated statement must be propagated to the output of the program, i.e., at the end of the program, one of the

output variables must have a different value compared to the same variable in the original program.

2.8.1 Strong, Weak, and Firm Mutation

Based on the way in which a mutant is killed during the execution process, mutation testing techniques can be classified into three types, strong mutation, weak mutation and firm mutation. The strong mutation technique (also known as traditional mutation testing) was originally proposed by DeMillo and Lipton (1978). In strong mutation, the mutants are executed against a test case and the produced output is checked after the execution of the program. A mutant is said to be killed only if the output can be distinguished from the output of the original program. This is called 'strong mutation testing' because it places a strong requirement on the used test case.

The weak mutation technique is proposed to reduce the execution time of the mutant (Howden 1982). In weak mutation testing, instead of checking mutants after the execution of the program, the mutants only need to be checked immediately after the execution point of the mutant or a mutated component. A mutant is said to be killed if the value produced at the location of the mutation is different from the corresponding value in the original program. This is less demanding on the test cases because it is always easier to force a program to have a different value at the mutation point than it is to force it to have a different value at the end of the program. Figure 2.4 demonstrates the execution of executing a test case on the program and the generated mutants and compares the two different states to kill mutants. The strong state mutation requires the outputs to be compared after the entire execution of the program and the mutants.

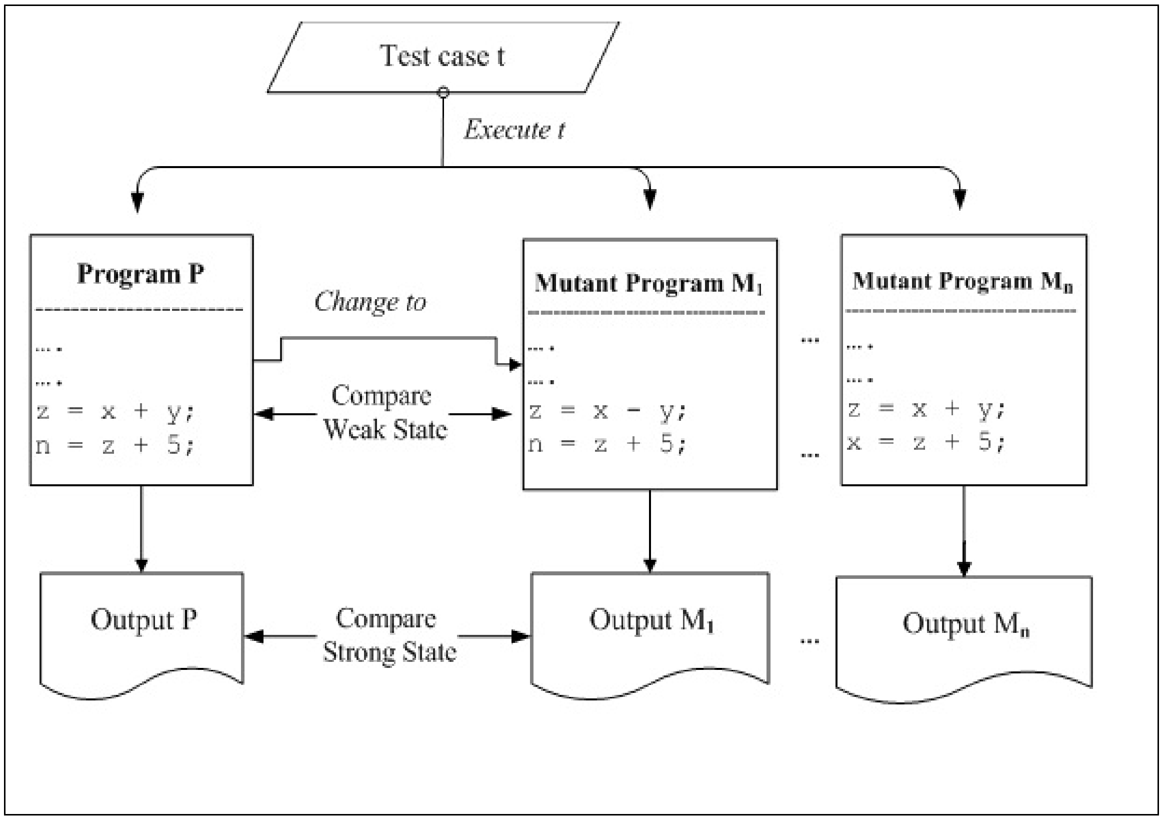


Figure 2.4: Comparison of weak state and strong state mutations

Since mutation testing is concerned with assessing the quality of test cases, one may prefer strong mutation testing to weak mutation testing. To explain how these two techniques affect the quality of test cases, consider the program segment given in Figure 2.5.

Assume, for example, that the statement labeled `2` is mutated by replacing the assignment statement `x = y * 2` with `x = y * 4`. If the program is tested on the test case `z = 10`, then the value of `x` just after statement `2` has been executed is `8`, whereas its value was `4` in the original program. If only the value of `x` at statement `2` is only considered then one could have tested the program with any test case except `z = 0` and would have killed the mutant. However, if the value of `x` is required to be different at the end of the program, then only the test cases where `z` has a value between `4` and `9` would have caught the fault and therefore killed the mutant.

```
  function f(z) {      // test z = 10
    y = 2;
    x = y * 2;     // mutate to x = y * 4;
    if (x < z) {
      x = 1;
    }
    else if (x == z) {
      x = 0;
    }
    else {
      x = x + 1;
    }
    return x;
  }
}
```

Figure 2.5: Program segment to assess the quality of test cases

The advantage of weak mutation is that each mutant does not require a complete execution process. Once the mutated component is executed, one can check for survival. However, as different components of the original program may give different outputs from the original execution, a weak mutation test set can be less effective than a strong mutation test set.

The firm mutation technique is suggested to overcome the shortages of both weak and strong mutations. It compromises between the two techniques, where execution is stopped at some point between the mutated code and the end of the program (Jackson et al. 2001).

2.8.2 Comparing Mutants

An equivalent mutant is a mutant that is syntactically different to the original program but has the same input-output behaviour. Sometimes two mutants of a program are not necessarily equivalent to the original program but they have the same input-output behaviour as each other. In this situation the mutants are equivalent to each other but they are not "equivalent mutants". For example,

```
original      mutant1       mutant 2
x = x - 3;   x = 0 - 3;   x = false - 3;
```

Consider that the statement `x = x - 3;` is present in a program and two mutants are generated as shown above. Because `false` is present in an arithmetical expression it is converted to a number which is zero. This means that mutant1 and mutant2 are equivalent to each other, but `mutant1` and `mutant2` are not equivalent mutants because they are not the same as the original program.

It is beneficial to detect mutants that are equivalent to each other because if a test case is found that kills one of the mutants then it will also kill the other mutant. It is necessary only to execute one of the mutants that are equivalent to each other because the output of one mutant will be equal to the output of every other mutant that is equivalent.

If two programs are equivalent to each other then they must have the same output for every possible input. If two programs have the same output for only some of the possible inputs then the programs are partially equivalent. Sometimes two mutants are partially equivalent. For example

```
original      mutant1       mutant 2
x = x - 3;   x = 0 - 3;   x = sub1(x) - 3;
```

Consider that the statement `x = x - 3;` is present in a program and two mutants are generated as shown above. Assume there is an input that sets `x` always to the value `1`. In this situation `sub1(x)` is the same as `0 false` and so mutant1 and mutant2 are partially equivalent to each other. Another example of two mutants that are partially equivalent to each other is

```
original      mutant1       mutant 2
x = x - 3;   x = 0 - 3;   x = logneg(x) - 3;
```

providing that there is an input that sets `x` always to a non-zero number or any other value which converts to `true`. In this situation `logneg(x)` is always `logneg(true)` which is the same as `false` which in an arithmetical expression is converted to `0` so mutant1 and mutant2 are partially equivalent to each other.

It is beneficial to detect mutants that are partially equivalent to each other because if a test case is found for which the mutants are partially equivalent and the test case kills one of the mutants then it will also kill the other mutant. It is necessary only to execute one of the mutants that are partially equivalent to each other.

2.8.3 Interpretation and Compilation Techniques

The interpreter-based technique was used in the first generation of mutation testing tools (Offutt and King 1987). For example, MOTHRA was the first significant mutation testing tool (Offutt and King 1987, King and Offutt 1991). This tool applies mutation testing to FORTRAN programs. The submitted program is parsed and the mutant maker (mutmake) invoked to generate mutant descriptor records (MDRs) (King and Offutt 1991). Each record describes the mutation to produce a mutant. Furthermore, the content of the MDR for a given variable depends on the type of that variable. In general, a mutant is created by modifying the source code and an interpreter executes the source code directly.

In order to improve speed, the compiler-based technique was suggested to replace the interpreter-based technique, because execution of compiled binary code is much faster than interpretation (Delamaro and Maldonado 1996). In the compiler-based technique, the mutant source is compiled into an executable program first, and then each compiled mutant will be executed by a number of test cases. There is, however, a high compilation cost for large programs (Choi and Mathur 1993).

Another optimization technique that is used in mutation testing is the compiler-integrated technique (DeMillo et al. 1991). It improves the performance of the compiler-based techniques. Because there is only a minor small change between each mutant and the original program, compiling each mutant separately in the compiler-based technique will result in redundant compilation cost. In the compiler-integrated technique, an instrumented compiler is designed to generate and compile mutants. The instrumented compiler generates two outputs from the original program: an executable object code for the original program, and a set of patches for mutants. Each patch contains instructions which can be applied to convert the original executable object code image to executable code for a mutant directly. As a result, this technique can reduce the cost of compilation.

2.8.4 Mutant Schema Generation

The mutant schema generation approach has been used in mutation analysis to reduce the cost of the interpreted techniques (Untch 1992). Instead of compiling each mutant separately, the mutant schema technique generates, from the original program, a special parameterized program called a meta-mutant. By changing the parameters, the meta-mutant can be used to represent all possible mutants. The cost of this technique is composed of a one-time compilation cost and the overall runtime cost. As this meta-mutant is a compiled program, its running speed is faster than the interpreter-based technique (Untch et al. 1993).

In order to explain how a meta-mutant can represent all the functionality of a set of mutants, one should recall that each mutant of a program `P` differs from `P` by only a single syntactic change to some statement in `P`. The way in which these statements are changed is guided by the set of mutation operators used. For example, assume that the statement `z = x + y;` is in the program under test. If the arithmetic operator replacement is applied to replace each occurrence of an arithmetic operator by each of the other possible arithmetic operators (i.e.,-, *, /, and %), then the following four mutants will be produced:

```
z = x - y;        z = x * y;        z = x / y;        z = x % y;
```

All these mutants can be represented as `z = x opr y;` and can be implemented as `z = opr(x, y);` where a parameter `opimutantOperator` selects, within the function `opr,` one of the arithmetic operators. The `opr` function can be implemented as follows:

```
function opr(x, y){
  switch (opimutantOperator){
    case “+”: return x + y;
    case “-”: return x - y;
      . . .
  }
}
```

An `opimutantOperator` parameter is required for each occurrence of an operator in the original program. The same approach can be used to implement operand replacement mutations. A function can be defined to get the value of a variable or some other specified variable depending on the value of a parameter `ximutantVariable` that defines the required variable mutation.

```
function getVal(x) {
  switch (ximutantVariable){
    case “x”:
      return x;
    case “y”:
      return y;
      . . .     // other variables here
  }
}
```

An `ximutantVariable` parameter is required for each occurrence of each variable in the program. With these two functions, it is possible to define a meta-mutant arithmetical expression for the original program expression `x + y` as given below:

```
opr(getVal(x), getVal(y))
```

To mutate the variable on the left-hand-side of an assignment statement, i.e. a variable in an l-value context, it is necessary to replicate the assignment statement. For example, the original program assignment expression `x = x + y;` would be implemented as the meta-mutant statement as follows:

```
switch (ximutantVariable) {
    case “x”:
      x = opr(getVal(x), getVal(y));
      break;
    case “y”:
      y = opr(getVal(x), getVal(y));
      break;
    . . .    // other variables here
  }
}
```

Overall, the meta-mutant is produced by translating each statement of the original program into a corresponding switch or conditional statement.

Object-oriented languages like Java and C++ are structured differently from procedural languages and they contain new features such as encapsulation, inheritance, and polymorphism (Barbey and Strohmeier 1994). These features imposed the need to change the requirements for mutation testing to handle new types of faults. The existing mutation operators for procedural languages are not sufficient for programs written in object-oriented languages and hence a set of class mutation operators (Chevalley et al. 2001) has been introduced.

The most recent work on reduction of the compilation cost is the bytecode translation technique (Ma et al. 2005). In this technique, mutants are generated from the compiled object code of the original program. As a result, the generated ‘bytecode mutants’ can be executed directly without compilation, which reduces the cost of mutant generation. This technique has been used in the mutation analysis of Java programming language. The MuJava tool (Ma et al. 2005) is an example of a mutation system that used bytecode and mutant schema techniques to produce mutants. Each mutation operator is related to one of the six language feature groups. The first four groups are based on language features that are common to all object-oriented languages. The fifth group includes language features that are Java-specific, and the last group of mutation operators are based on common object-oriented programming mistakes (Ma et al. 2005).

However, not all programming languages provide an easy way to manipulate intermediate object code. There are also some limitations for the application of bytecode translation in Java, such as not all the mutation operators can be easily implemented at the bytecode level (Schuler et al. 2009).

2.9 Previous Work on Mutation of Dynamically Typed Programs

Little work on the mutation testing of dynamically typed programs has been found in the literature. Almost all mutation testing tools have been developed for statically typed languages. Mutation analysis for dynamically typed programs has been used for Python and Ruby programs

Due to the unavailability of type information until the execution of the program, mutation operators that are commonly used in statically typed programs are not used. For example, the + operator in a dynamically typed programs can be used in number addition and string concatenation. In order to replace this operator with another arithmetical operator (-, /, *. %), the type information of operands is required to avoid generating type-mutants (Bottaci 2010). Instead the Pester and Heckle system mutate only the typed operators and literals in the program.

Gligoric has developed a mutation system for Smalltalk (SMutant) which is a dynamic language.(Gligoric et al. 2011). Instead of applying mutations statically, SMutant waits until the type information is available (at runtime) and applies mutations dynamically. Little information has been published and it is not clear which mutation operators are applied.

Bottaci (Bottaci 2010) introduced the basic concepts of dynamic mutation. It was at the concept stage at that time and has now been developed into a mutation analysis tool (Tescripta) as a result of work done in this thesis.

In the context of computer security, the problem of cross-site scripting has been investigated by applying a small number of mutation operators to JavaScript and PhP programs (Shahriar et al. 2009).

Chapter 3: Dynamic Type System: JavaScript

3.1 Introduction

JavaScript is a dynamically typed object-based high-level scripting language (Richards et al. 2010). Unlike statically typed programming languages, it allows a programmer to add, at run-rime, properties to an object, new functions and variables, and dynamically determines the data types of values and operations in a program. The following discussion explains the basic elements and features of dynamically typed languages e.g. JavaScript, and how they are different from statically typed languages.

3.2 Statically vs Dynamically Typed Languages

A programming language that provides type information during compile-time is considered a static or statically typed language. In a statically typed language, all variables used in a program are associated with a particular type in the program. In a statically typed language, all variables have either a particular type or a generic type but in either case, no-type error can occur at run-time. Typically, statically typed languages allow type errors to be discovered early in the development cycle. Examples of these languages include Ada, Java, Pascal, C++, C#, etc.

In contrast, a dynamically typed language such as JavaScript, determines the types of values in a program at run-time. Examples of dynamically typed languages include JavaScript, Lisp, Python, PhP, Ruby, etc.

Consider, for example, the JavaScript code given in Figure 3.1, where variables are defined without specifying the type of values that may hold. When the code is run, the definition of `+` takes into account the type of both operands `x` and y, implicitly converting `x` to the string `"8"` and then concatenate the values of `x` and `y` to produce `"855"`. In the first occurrence of x, it holds a number; in the occurrence on the left of the assignment, it holds a string.

```
…
var x = 8;      // x is a number
var y = "55";  // y is a string
x = x + y;      // x assigned string “855”
…
```

Figure 3.1: An example of how variables hold values of different types in a dynamically typed language. In the first occurrence of x, it holds a number, in the occurrence on the left of the assignment, it holds a string.

Figure 3.2 shows an example of a mixed type arithmetical expression that would be a program error in a statically typed language. In JavaScript, there is no exception raised in this case. Instead, an attempt is made to implicitly convert the string to a number by parsing the string and a normal arithmetical operation is performed. Since the string contains non-numeric characters, and therefore, cannot be parsed to a number, the result of this expression will be the value `NaN` (Not a Number). Note that `NaN` is not considered an error and program execution continues normally. If the `NaN` value occurs in a Boolean context then it will be implicitly converted to a Boolean false.

```
…
x = 3 - "hello"; //JavaScript assigns x the NaN value
if (x) {         //no exception here, x is converted to false
  …
```

Figure 3.2: An example of a mixed type arithmetic expression in JavaScript in which no exception is raised

However, execution with values such as `NaN` is not permitted in all dynamically typed languages; Python for example, throws an exception.

3.3 JavaScript Value Types and Variables

In JavaScript, the simple value types include the three value types: number, Boolean, and string. It also contains two special value types `null` and `undefined`. Everything else is considered an object type. Table 3.1 lists these types.

`number`: 64-bit floating point numbers as well as integers. It also includes the special values `NaN` (Not a Number) and Infinity (e.g. divide by zero).

`Boolean`: A value type which is either `true` or `false` value.

`string`: A series of zero or more characters inside quotation marks.

`null`: A value type that can have only the value `null`.

`undefined`: A value type assigned to uninitialized variable or undefined variable used in the program.

`object`: Properties and methods of an object. Also, arrays, functions, and prototypes are objects

Table 3.1: The different types in JavaScript

Usually, variables are defined using the `var` statement. Variables which are not explicitly initialized or used in a program without defining them are given the value `undefined`. A variable can be set to a value of any type of those listed in Table 3.1.

3.4 Operators and Implicit Type Conversion of Value Types

Table 3.2 lists the most common operators available in JavaScript. These operators allow a wide range of implicit type conversions and are influenced by the context. Therefore, the result produced by using these operators depends on the current type of values of operands. For instance, the `+` operator is used for both numeric addition and string concatenation. If both operands are numbers, then the `+` operator performs addition. If any of the two operands is a string value type and the other is not, then the non-string value is converted to a string type and the result is concatenated. Figure 3.3 shows the possible conversion by using some mathematical operators.

Operators					
Arithmetic	Assignment	comparison	logical	Bitwise	Special
+	=	==	&&	&	?:
-	+=	!=	||	^	delete
*	-=	===	!	|	in
/	*=	!==		<<	instanceof
%	/=	>		>>	new
++	%=	<		<<<	this
--		>=		~	eval
		<=			void

Table 3.2: The most common operators used in JavaScript

```
5 + "4";           // "54"
1 + true ;         //  1 + 1 = 2
null + false;      //  0 + 0 = 0
null + true;       //  0 + 1 = 1
null + undefined; // NaN
new Date() + 86400000;   // Tue Apr 19 12:30:52 UTC+0100
                             20118640000
5 - "";       // 5 - 0 = 5
null - "5";  // 0 - 5 = -5
"5" * 5;     // 25
5 - "55";    // -50
5 - "abc";   // NaN
```

Figure 3.3: Some examples of type conversion in the context of mathematical operators

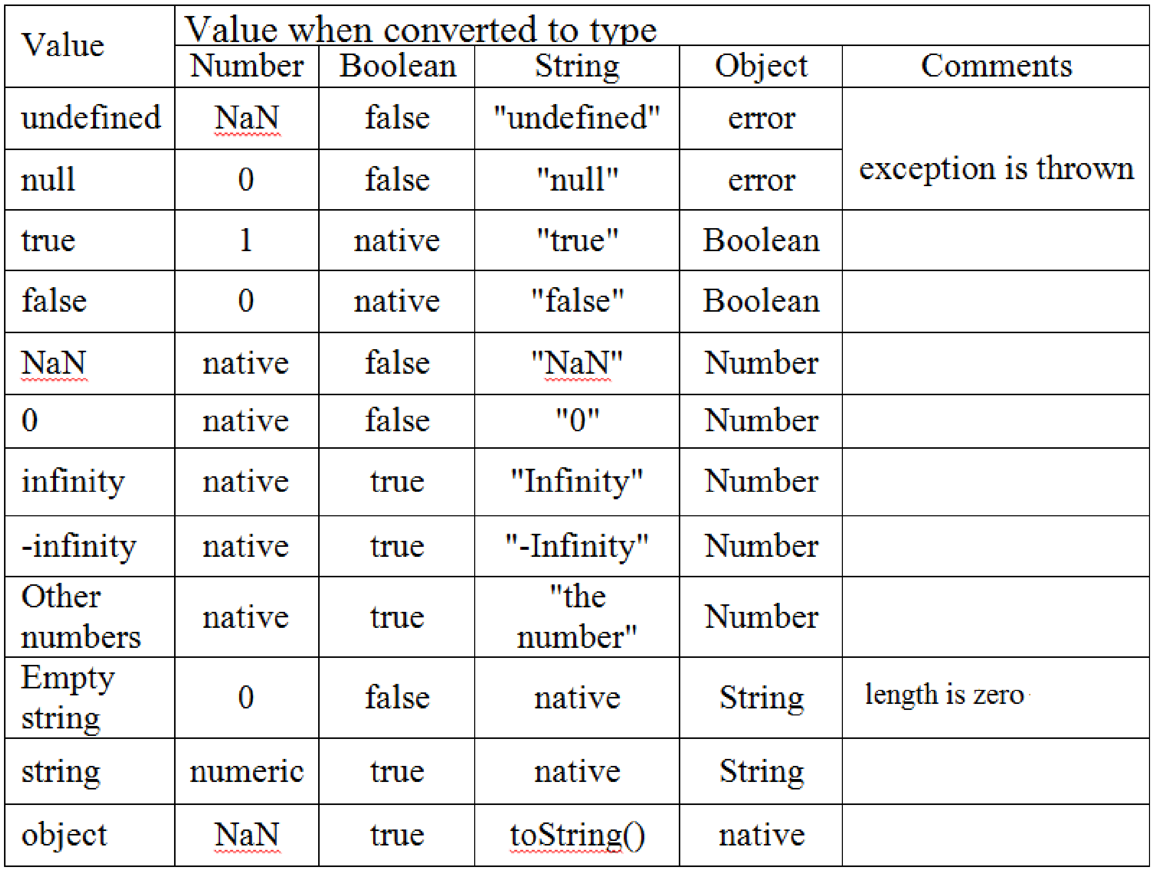
During execution, JavaScript will check the types of values as particular operations are applied and attempt to convert types as necessary. For example, if a string is used in a numeric context, JavaScript will attempt to convert it to a number. If that string contains one or more nonnumeric characters, then it cannot be converted to a number and the result will be `NaN`. Also, `if`, `for`, `while` and `do-while` statements require Boolean values in their condition part, so if other types are present then they are converted to Boolean.

The automatic conversions are wide ranging. Arrays containing a single element are automatically converted to the element. Consider, for example, the code below:

```
var x = [5];
document.writeln(x - 1); //produces 4
document.writeln(x + 1); // produces “51”
x[1] = 9;
document.write(x + 1); // produces “5,91”
document.write(x - 1); // produces NaN
```

Although the array `x` contains a number, the use of the `+` operator will result in producing a string (i.e., concatenate `x` with `1`). On the other hand, if the operator is an arithmetical operator other than the `+` operator and `x` can be parsed to a number then the result will be a number. However, this is not the same if the array contains two or more values. For example, if `x[1] = 9` then subtracting `1` from `x` will result in `NaN`.

In general, JavaScript will automatically perform implicit conversions into values suitable for the context. Table 3.3 shows a set of equivalent values that are used in different contexts.



Value	Value when converted to type				
	Number	Boolean	String	Object	Comments
undefined	NaN	false	"undefined"	error	exception is thrown
null	0	false	"null"	error	
true	1	native	"true"	Boolean	
false	0	native	"false"	Boolean	
NaN	native	false	"NaN"	Number	
0	native	false	"0"	Number	
infinity	native	true	"Infinity"	Number	
-infinity	native	true	"-Infinity"	Number	
Other numbers	native	true	"the number"	Number	
Empty string	0	false	native	String	length is zero
string	numeric	true	native	String	
object	NaN	true	toString()	native	

Table 3.3: Values and type conversion in JavaScript based on context

3.4.1 The "==" and Other Comparison Operators Conversions

JavaScript allows variables and values of any type to be compared. It applies special rules of type conversion when comparison operators are used. Consider, for example, the two conditional expressions

`"0" ? true : false and`

`"0" == true ? true : false`.

The first argument to the conditional expression (expression in front of the ?) is evaluated as a Boolean. If true, the result is the value in front of the '`:`' otherwise the result is the value after the '`:`'. The result of the first expression above is `true` because when a string is converted to a Boolean, only the empty string is `false`. Now consider the second conditional expression. Since `"0"` converts to true, we would expect `"0" == true` to be `true`. In fact this expression is `false`. In the evaluation of the expression `"0" == true`. The string `"0"` is not converted to a Boolean but instead it is converted to a number, i.e. `0. true` is also converted to a number, i.e. `1`, and hence the result is `false`. In JavaScript, the type conversion rules depend on the context in which the value occurs. In particular, in the context of a == expression, the operands are converted to numbers, in the context of a conditional expression or if-statement, the operands are converted to Booleans. Table 3.4 lists the possible type conversion rules with the comparison operators. Note that == is not transitive since

```
"" == 0 is true because "" converts to number 0,
0 == "0" is true
```

Using transitivity, i.e. `a = b` and `b = c => a = c` then from the above two expressions we would expect that `"" == "0"` is `true`. Since they are both strings, no conversion takes place and so the strings are compared directly so

`"" == "0"` is `false.`

`true` is converted to the number 1.
`false` is converted to `0`.
If both or any of the operands are `NaN`, the equality operator returns false.
`undefined` and `null` are equal, when compared in an expression.
`undefined` and `null` are not equal to `0` (zero), "", or false.
compare a string and a number, the string is converted to a number
compare an object and a number, the object is converted to a number
compare an object and a string, the object is converted to a string
compare two objects for equality, the addresses are compared

Table 3.4: Type conversion rules in the context of the comparison operators

The relational comparison operators in JavaScript, i.e. `== != === !== > < >= <=`, compare strings using lexicographical ordering and numbers using number ordering. If a number is compared to a string then they will be converted to the same type before comparing the two values. However, the same is not `true` if a number is compared with a string using `===` or `!==` operators. For example, `"10" === 10` will return `false` because the two values are not of the same type. The === and !== operators are strict and do no-type conversion.

3.4.2 The Logical Operators && and || Conversions

The logical operators in JavaScript can be used with any type. They expect Boolean values and if not, the operands are converted to Boolean. Though the logical operators convert the operands to Boolean, they do not return a Boolean value but the value of their operand itself is returned. Figure 3.4 shows some possible conversion by using some logical operators.

```
var y = 0 || "8";    // y is equal to "8", 2nd operand returned
var y = "5" || "8";  // y is equal to "5", 1st operand returned
var y = 0 && 8;      // y is equal to 0, 1st operand returned
var y = "8" && 5;    // y is equal to 5, 2nd operand returned
var y = "5" && "8";  // y is equal to "8"
var y = 8 && "abc";  // y is equal to "abc",
var y = false && "8";// y is equal to false
```

Figure 3.4: A sample of type conversions using && and || logical operators

3.5 Object Type

There are different sub types of objects that can be used in JavaScript such as arrays and functions. The following discussion will briefly explain the use of these types.

3.5.1 Objects in JavaScript

JavaScript allows the developer to use built-in objects as well as define his own objects. In JavaScript, an object is an unordered collection of name-value pairs. The members of an object can be any type and function members are called methods. There are no classes in JavaScript. To create an object, an object literal expression is used. For example,

```
var address = {num: 22,
               street: “Oxford”,
                city: “London”}
```

In order to access (i.e., get and set) the properties of an object, one can use either the “.” (dot) operator or the “[]” operator.

```
var houseNum = address.num;
var streetName = address[“str” + “eet”];
```

The dot operator expects an object on its left and a property name on its right. An expression can be used with [] operator, e.g. `o.[a+b]`, where `a+b` should produce a string which can be a name of a property of object `o`. Objects are dynamic; new members can be added or existing members can be deleted at run-time.

In JavaScript, member access expressions are calculated at run-time. A possible error is to use the wrong member or property name. If the wrong name is used in the context of an l-value (l-value means left-hand-value i.e. the target of an assignment), and the name is not an existing property name then a new property is implicitly added to the object and becomes the target of the assignment. If the wrong name is used in the context of an r-value (r-value means right-hand-value, i.e. the value to be assigned) and the name is not an existing property name then the property value has the value `undefined`.

3.5.2 JavaScript Arrays

An array is an ordered collection of values. Using an array, a list of different types of values can be grouped in a single variable. In JavaScript, arrays are considered of an object type. Arrays in JavaScript are dynamic and not typed. Unlike many other programming languages, neither the length of an array nor the types of values is fixed. An array can contain numbers, strings, Booleans, objects, functions, arrays or a mixture of them. Arrays have functions and properties associated with them. For example, `length`, `pop()`, `slice()`, etc.

Due to the dynamic capability of arrays, arrays can grow or shrink at run-time; a new element can be added or an existing element can be deleted. The first array element is indexed by 0, but string and negative indices can also be used to index the array. In this respect an array is similar to an object in that elements are indexed by name rather than position.

3.5.3 Prototype Object

There are no class objects in JavaScript. In order for a set of objects to share the same set of properties, the common properties are placed in a shared prototype object. Every object that shares the common properties has a reference to the single prototype object. For example, all array objects have a function property concat(Array) which creates a new array by concatenating the current array with an argument array. This function is a property of the Array prototype object. Every object has a property that refers to the prototype object for that object.

When an object is created it is always created as an instance of an existing object rather than with a class. The prototype property of the new object is set to be the same object as the prototype object of the existing object. Like any object property, the prototype property of an object can be set to any value. When attempting to retrieve the value of a given object property, if the property is not defined directly in the object then the properties of the prototype are searched for the given property. This is known as prototypical inheritance. New properties can be added to the prototype object in which case all objects that link to that prototype object can "inherit" the new property.

Consider, for example, the code

```
var circle = {radius: 2, fill: true};
circle.prototype.getArea =
               function() {
                 return (3.1459 * this.radius * this.radius);
               }
```

The above code will create an object named `circle` with a radius property set to `2` and a fill property set to `true`. The created object can be represented graphically as in Figure 3.5. As shown above, the `circle` object has a link to a default prototype object, Object. The object `Object()` contains several properties and methods that can be referenced and accessed, by prototypical inheritance, from circle.

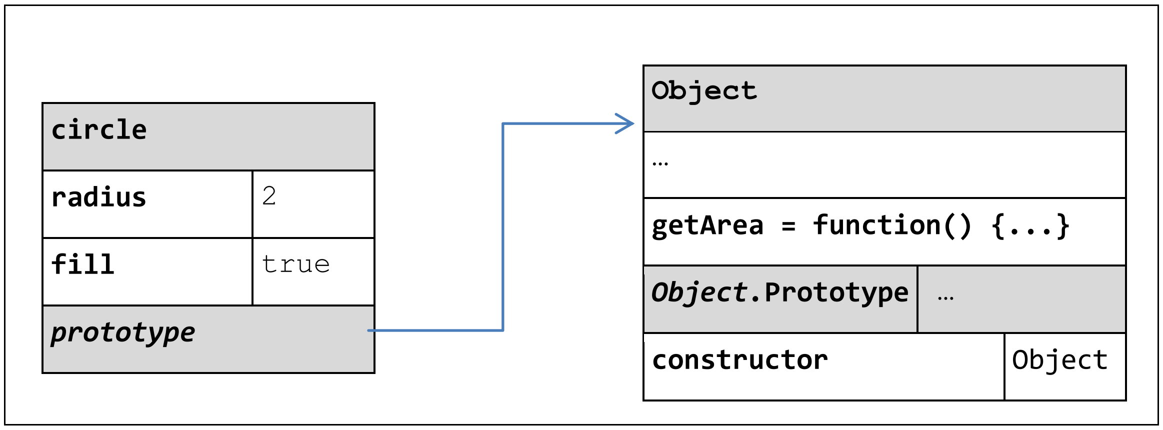


Figure 3.5: The graphical representation of creating an object

Properties can be added to the prototype object. In this example, an area function is added. For example,

```
circle.getArea()  // has the value 3.1459 * 2 * 2
```

Because the `getArea()`function was added to the Object prototype, every object that links to this prototype "inherits" the `getArea()`function, irrespective of whether the property radius is defined for that object.

Because the prototype property of an object can be set like any other property, a prototype object can itself have a prototype. In general, every JavaScript object inherits through a chain of prototypes. The `Object.prototype` is the ultimate base prototype for all prototypes. If a property of an object is to be accessed, JavaScript first checks to see if the property is defined directly in that object. If it is not, it then checks at the object's prototype to see if the property is defined there. If not, then it continues checking at that object's prototype for the property until reaching the `Object.prototype`.

3.5.4 Function Object

JavaScript functions are objects with executable code, i.e. a function, associated with them. A function definition consists of a function statement and a block of statements. A function is executed by an event or by a call to the function. When a function is called, it is not required that you pass the same number of arguments with which it was defined. Extra arguments are ignored. Missing parameters are given the value `undefined`.

3.6 Other Dynamic Programming Languages

In addition to JavaScript, there are other dynamically typed languages. A number of these languages are also widely used for scripting and web computing. These languages include PHP, Python Perl, Ruby, etc. Some of these languages are prototype-based, as is JavaScript, but others are class-based. With some languages that are class-based, class members can be added and deleted at run-time. This makes them very similar to prototype-based languages in terms of how objects can be mutated. The main difference between a class-based object and a prototype-based object is that in the prototype, each object instance contains a copy of both data and function (method) members.

Although dynamically typed languages are soft typing, some languages check the type to ensure that no-type error may occur (e.g. Python), whereas other languages such as PHP, Ruby, Perl and JavaScript are generous and usually allow implicit type conversions to occur among different data types in a program at run-time. Due to the fact that the typing system in the most of these languages is common to JavaScript, the mutation analysis approach introduced in this thesis can also be applied to programs written using other dynamically typed languages. Of course, some of these languages use different types of values than those in JavaScript. In this case, a proper set of mutation operators should be introduced for the mutation analysis of these new types.

Chapter 4: Static Mutation of Dynamically Typed Programs

4.1 Introduction

This chapter collects a set of mutation operators (or mutations) similar to the traditional mutation operators used in the mutation of statically typed programs. These mutations are the mutations that can be applied by making a small change to the syntax of a program. These include the insertion of functions, e.g. unary minus, `abs()`, logical negation etc. They also include the replacement mutations (i.e., replacement of operands and operators).

In a dynamic program, the choice of suitable mutation operators is influenced by the many implicit type conversions that are present in JavaScript. Because of the automatic type conversions, some of the traditional mutation operators become equivalent to each other and hence redundant.

This chapter also describes a static approach to the mutation of dynamic programs that is based on making small changes to the program based only on the syntax of the program; no execution of the program is done to gain type information. In this static mutation approach, mutations are made to all the program elements where some syntactic situation is present.

4.2 Insertion Mutations

An insertion mutation operator inserts a function around a variable. So for example, `x = x + 3;` can be mutated to `x = abs(x) + 3;`

when the `abs()` function is applied to the variable occurrence `x`. The variable occurrence cannot be on the left-hand-side of an assignment, which is called an l-value context. When the value of a variable is used rather than written, e.g. on the right hand side of an assignment, it is called an r-value context and insertion mutations are applied to variables in an r-value context.

The possible operator insertion mutations are the number operator insertions, the Boolean operator insertion and the string operator insertion. The number insertion mutations are:

```
add1:   x mutated to x + 1
sub1:   x mutated to x - 1
neg:    x mutated to -x
abs:    x mutated to Math.abs(x)
negabs: x mutated to -Math.abs(x)
zpush:  x mutated to zpush(x), zpush(x)= x for all x,
                              but mutant killed if x equals 0
```

The numbers `0` and `1` have a special significance in arithmetic expressions and the number mutation operators recognise this. This justifies the `add1` and `sub1` operators. The `zpush` operator insertion was introduced in the Mothra mutation system (King and Offutt 1991). If the number is non-zero then `zpush` does nothing, i.e. returns the argument. If the argument is zero then the mutant is killed. The purpose of this operator is to force the tester to write a test that sets the value of a variable to zero. If this is not possible for the program under test then the mutant cannot be killed and is an equivalent mutant.

In the case of dynamic programs, non-numeric values can be converted to numbers and some of these values (“`0`”, `false` or the empty string) can be converted to zero. This leads to the question of how `zpush` should behave with values that are not numbers but can be converted to number. If `zpush` allows values to be converted to number before testing for zero (i.e. it uses `x == 0`) then it will be easier to kill `zpush` mutants in general. This is because the tester is being forced to write a test that will set a variable to zero, `false` or the empty string. If instead the definition of `zpush` does not allow its argument to be converted to number (i.e uses strict equals, `x === 0`) then the `zpush` mutants are more difficult to kill.

The use of the strict equals test for zero will produce more equivalent mutants than if the `zpush` used a non-strict test for zero. Any mutant where no input can set `x` to a number will be equivalent. In the design of mutation operators it is important to avoid mutation operators that generate many equivalent mutants. Equivalent mutants cannot be killed and usually the tester must detect them manually by inspection. This is time consuming. In the case of the use of the `===` test for zero in `zpush`, compared to the `==` test, all the additional equivalent mutants will be equivalent because no input will set the `zpush` argument to a number. The tester should find it easier to identify this kind of mutant because all that is necessary is to calculate the type of the `zpush` operand rather than its value.

For this reason, the additional type equivalent mutants produced when `zpush` uses the `===` test for zero are not considered a large amount of extra work for the tester. In addition, the use of the `===` test for zero in `zpush` has the benefit that it forces the tester to write a more discriminating test case. For these reasons, the thesis proposes that `zpush` uses the strict equals test for zero.

The single Boolean insertion operator is:

`logneg`: logical negation

so for example,

```
if (x && y) { ...   mutated to      if (!x && y) {  ...
if (x) {  ...      mutated to      if (!x) {  ...
```

Notice that the ! operator converts its argument to a Boolean before negation.

String mutations are not commonly used in existing statically typed mutation systems. Strings are heavily used in JavaScript, however, and so string mutation operators are suitable for this language. Following the example of the number mutations, the empty string is considered a special string in the same way that `zero` is a special number. Using the `zpush` number mutation as an example, this thesis defines `deadOnEmpty(x)` which returns `x` unless `x` is the empty string, in which the mutant is killed.

`deadOnEmpty(x)`: mutant is killed if the argument `x` is the empty string

The `deadOnEmpty(x)` mutation operator forces the tester to write a test that sets the argument variable `x` to the empty string. If the program under test does not allow this then the mutant is equivalent to the original program. Again, if `x` cannot be set to a string then `deadOnEmpty(x)` is an equivalent mutant. The previous discussion about the equivalent mutants produced by `zpush` also applies to `deadOnEmpty` but to the string type. This means that `deadOnEmpty(x)` is defined as `x === ""` and not defined as `x == ""`. If the argument to `deadOnEmpty` cannot be set to a string then `deadOnEmpty` is an equivalent mutant.

The set of possible insertions mutations have been described but not all of them would necessarily be applied to a variable occurrence because of the context of that variable occurrence. The way the context of the variable occurrence determines which mutations are applied is explained later in this chapter.

4.3 Replacement Mutations

Replacement mutations can be divided into mutations that replace operators and mutations that replace operands.

4.3.1 Operator Replacement

Operators are typed. For example, arithmetic operators are applicable only to numbers. To avoid type-mutations, arithmetic operators should be replaced only with other arithmetic operators. This will not introduce type-mutations except that the `+` operator is overloaded between number addition and string concatenation. Any arithmetic operator (except `+`) is replaced with each other arithmetic operator. Replacement of the `+` operator is more difficult. Without type information, it is not known if `+` is an arithmetic operator or a string operator and for some inputs it could be both during a single program execution. Sometimes the type of the `+` operator can be determined from the syntax of the program. For example in `x + "hello" +` is always a string operator. A `+` operator is replaced with other arithmetic operators only if the context in which the operator occurs is not a string context. Later in this chapter, the thesis shows how type-mutations and redundant mutations can be avoided by considering the context in which the operator or operand occurs.

The logical operators `||` and `&&` are replaced with each other only. The relational operators `<`, `<=`, `>`, `>=`, `==`, `!=`, `===`, `!==` are replaced with each other only. In this way type-mutations can be avoided.

The arithmetic assignment operators, e.g. +=, −=, *=, etc. are replaced with other assignment operators. The `=` operator is not replaced with the arithmetic assignment operators.

Each unary operator, e.g. `!, -, ++, --,` etc. is replaced with another unary operator. In addition, `++x` is replaced with `x++, --x` is replaced with `x--` and vice versa. There is no insertion of `++` and `--` operators. In a mutation system for C (Agrawal et al. 1989), the increment and decrement operators, `++` and `--`, are inserted before or after a variable to mutate an expression. Although JavaScript contains these operators, they are not used very much because JavaScript does not use pointer arithmetic. In addition, the operand mutation performed by the `add1` function and the `sub1` function has the same necessity condition as the prefix operators. The means that

```
add1(x)    and    ++x
```

return the same value. However, they are not equivalent to each other because `++x` changes the value of `x`.

4.3.2 Operand Replacement

Replacement of operands includes replacement of operands with literals and the replacement of variables with variables as explained in the following subsections.

4.3.2.1 Replacement with Literal

There is the question of whether, when a literal is replaced with another literal, it should the literal be of the same type, number, Boolean, string, etc. It is obviously possible at compile-time to check that when a literal replaces a literal it has the same type. In addition, it is more likely that a competent programmer will mistake one literal for another literal only if they are of the same type. For these reasons, this thesis proposes that type-mutation of literals is not done. This means that when a literal replaces a literal it must have the same type.

There are some special cases with literals. If two literals are equal then they are not replaced because obviously the mutant is an equivalent mutant. Two values that may be held in program variables are `undefined` or `null.` In deciding whether to replace literals with `undefined` or `null` it should be considered that the replacement of a literal with `undefined` or `null` is likely to produce a reasonably large change in the input-output behaviour of the program and the mutant should be easily killed, i.e. incompetent. An important objective of a mutation method is to avoid incompetent mutants and for this reason literals are not replaced with `undefined` or `null`.

The literals `0` and `1` are often used in programs. Even if these literals do not occur in the function that is to be mutated they are still used to replace variables and other number literals. This means that any number literal can be replaced with `0` and `1`. In this thesis, the two mutation operators which replace a variable or literal with `0` or `1` are called `c0` (constant zero) and `c1` (constant one).

There are two Boolean literal replacement mutation operators. The `cFalse` operator replaces the argument operand with the literal `false`. The `cTrue` operator replaces the argument operand with the literal `true`.

Operands can be replaced with string literals that occur in the program. There is a single string literal replacement operator (`cEmpty`) which replaces an operand with the empty string.

In addition to the primitive type literals, number, Boolean and string, JavaScript has object and array literals. An object or array literal can be quite large. The aim of mutation is to make small changes to a program and so replacement mutations should also be small changes. Instead of replacing a complete object literal, only the variables and primitive data type literals used in the definition of the object literal are replaced or used for replacement. This rule also applies to array literals. In general, the value of an object literal member can be any expression. The leaf members of object literals are values of the primitive data types, number, Boolean and string. Consider, for example, the code given below:

```
var days = [“mon”, “tue”, “wed”, “thu”, “fri”];
  // replace “mon” with “”, “tue”, ... “sizes”,

var small = 3;               // replace with 0, 1, 4, 7

var o = {name: “sizes”,      // replace with “mon”, etc.
          little: small,
         medium: small + 4,  // replace with 0, 1, 3, 7
         large: small + 7};  // replace with 0, 1, 3, 4
```

In the above code, the number literals would replace each other and the string literals would replace each other. Each string in the `days` array is replaced with every other string. The complete array literal or the complete object literal is not replaced.

JavaScript contains a delete operator that deletes an element of an array or object. The delete operator could be used as a mutation operator but it should be used to make only small changes to the program otherwise the mutant is too easily killed. A possible mutation operator is to delete the leaf elements of an array or object. In the following example code

```
var residence = {house: 27,
                  street: “Bayes”,
                 tel: {code: 01536,
                       num: 519146},
                };
```

the three numbers and single string could be deleted to make four mutants. The properties, `house`, `street`, `code` and `num` would each be deleted to create different mutants. An example property deletion mutant is

```
var residence = {street: “Bayes”,
                 tel: {code: 01536,
                       num: 519146},
                };
```

The value of `tel` property is not primitive and so the `tel` property is not deleted.

A very similar result to the delete operator can be achieved by setting a leaf element of an array or object to the value `undefined`. For this reason, no delete mutation operator is defined. Instead the replacement of a leaf element of an array or object with other variables and literals is considered later as a replacement mutation.

4.3.2.2 Variable Replacement with Variable

If `x` and `y` are two variables that occur in a function then the variable replacement mutant is generated by replacing an occurrence of `x` with `y` or replacing an occurrence of `y` with `x`. The replacement is done only if the replacement variable is in scope in the new location. JavaScript has two scopes, local or function scope and non-local or global scope. In a function, a local variable is declared with the `var` statement. The scope of the local variable begins at the `var` statement and ends at the end of the function. For example,

```
function f(x) {
  var k = 0;          // 0 replaced by 1, x.  m not in scope
  k = x;              // x replaced by 0, 1, k and k by x.  m not in scope
  var m = k + 1;      // k replaced by 0, 1, x. 1 replaced by 0, k,x.
  ... x ...  ;        // x replaced by 0, 1, k, m.  m in scope

}
```

The variable occurrence that is used to declare the variable in a `var` statement is not mutated. For example, in the above example, `k` in `var k = 0;` and `m` in `var m = k + 1;` are not mutated. If the variable that is declared in a `var` statement is mutated then it has the effect of removing the original declaration. Removing the declaration of a variable is likely to have a large effect on the behaviour of the program and the aim is to avoid mutants that are likely to be too easily killed, i.e. incompetent.

In general, a variable can hold a value of any type so if a variable replaces a variable or literal, the type of the replacement may not be the same as the type of the original. Because of the implicit type conversions this is not necessarily a program error; it will not necessarily raise an exception and stop the program execution. If a variable occurs in a specific context then the context may indicate the probable type or that a type conversion will be applied if the variable is not of the type required by the context. By making the mutation operators sensitive to the type context of an expression, the number of type-mutations is reduced. The next sections describe how this is done.

4.3.2.3 Other Possible Mutations

JavaScript allows a function to be invoked without checking at compile-time, the number of arguments that are given in a function call. It is, however, possible to call a function by passing any number of arguments that may not be the same as the number of parameters of a function. For example, if the number of parameters exceeds the number of arguments in a function call, then the ignored parameters are assigned the “undefined” value. On the other hand, if the number of arguments is greater than the number of parameters, then the extra arguments will be excluded without causing an error in the program.

It is, however, not obvious that the mutation of the arguments in a function call, perhaps to delete arguments, is a good way to detect errors in the program. In some cases, discovering missing arguments can be considered as a direct static analysis problem. Furthermore, functions in JavaScript can be generated and added to the program at run-time. Thus, the use of static analysis may not be possible to determine all cases of missing arguments.

JavaScript has only function and global scopes. A variable can be used without the need to declare it in the program. If the name of that variable is misspelled later in the program, then this will lead to the implicit declaration of a new variable. Moreover, inside a function, using an undeclared variable will introduce a global variable rather than a local variable. In order to declare a variable in the scope of a program or a function, it is necessary to precede the variable name by the keyword `var`. This suggests an obvious mutation operator in this language feature. The operator would remove the declaration of a variable that had a declaration and add a declaration for a variable that did not. It is not clear, however, how competent such mutants would be. In addition, it is reasonable to adopt a coding rule in which all variables are declared. Tools such as JSLint (Crockford, 2002) are available to enforce such rules. Given the doubtful benefit of variable declaration mutation, and the accessibility of tools to enforce declaration, variable declarations are not considered suitable for mutation.

4.4 Use of Type Context to Selectively Apply Mutation Operators

A syntactic approach towards mutation introduces the possibility of producing type-mutations. Consider a mutation which is a variable occurrence for variable replacement. This could lead to a variable occurrence holding a number replaced with a variable holding a string. This is a type-mutation because at the mutated expression, during execution, the mutant differs from the original program in the type of the value held in the mutated variable occurrence.

Although in general a JavaScript variable can hold values of any type, in some cases, it is possible to determine that a value will either be of a known type or will be converted to that type. For example, in the code

```
x - y
```

the types of `x` and `y` may not be known, but because of the subtraction operation it is known that either they hold numbers or that the values they hold will be converted to number. If the construction of mutants is syntax directed then there is the possibility to recognise the context of a variable occurrence because the context can be recognised from the syntax.

In general, if the type context of an operator or a variable can be known at compile-time then this information should be used to avoid replacement mutations that do not convert to the type of the context.

4.4.1 Checking the Type and Context

To avoid type-mutations and redundant mutations it is necessary to know the type of a value held in a variable occurrence and also the context of the variable occurrence. To determine the context of a variable occurrence is relatively easy. The context is determined by the local syntax of the program. When the program is compiled the syntax tree for the program is constructed. At any variable occurrence node in the syntax tree it is possible to move to the parent node in the tree and the parent of that parent if necessary to determine the language element in which the variable occurrence occurs.

To determine the type of a value held in a variable occurrence is not as easy as checking the context. One way to try to discover the type of a program element is to do type inference (Duggan and Bent 1996). If the types of some elements are known then it is possible to calculate the result types of expressions in which they occur; for example, if `x` and `y` are numbers then `x + y` must be a number. This relies on some initial type information and typed operators. In general, type inference uses data flow analysis (Rapps & Weyuker 1982) to calculate the path of variable occurrences through which data values are transferred during execution. For example, the third occurrence of the variable x in the code below is in an ambiguous context, it is either number or string. From data flow analysis it can be calculated that the value in the third occurrence of x is equal to the value assigned in the previous assignment statement and so is a number.

```
x = x - y;   // x in number context
. . .        // no assignment to x or y in these statements
x + y        // x in number or string context but type is number
```

Type inference is most effective in strongly typed languages. Languages that use type inference are O'Caml (Leroy et al. 2002) and Haskell (Duggan and Bent 1996). In these languages, all variables have a known type at compile time but it is not necessary to declare the type of variables because the type is inferred. Depending on the operations applied to a variable, the inferred type is either a simple type, e.g. int or a polymorphic type, .e.g. List<T> where T is the type parameter and can be any type. In these languages no-type conversions are allowed.

Type inference works from known types but in JavaScript variables are not typed so less type information is known than in a strongly typed language. This makes type inference in a language like JavaScript more complex and not so accurate. Anderson (Anderson 2006) defines a static type inference system for a subset of JavaScript. This subset includes: functions that are used to create objects, members of objects, and members that can be added to objects dynamically, but does not include: libraries of functions, dynamic variable creation, functions as objects, dynamic deletion of members, and prototyping.

The objective of the Anderson type system for JavaScript is to find the most specific type for a variable so that it can be compiled in the most efficient way. This is the most specific type for all the occurrences of the variable. For mutation analysis different type information is required. It is necessary to know the most specific set of types for each variable occurrence because mutations are made at specific occurrences not to the variable as a whole.

In this thesis, it was decided to limit the type analysis to context analysis and not to use type inference. The research necessary to modify a type inference system such as the Anderson system would be data flow analysis research and different from the mutation analysis research which is the aim of this thesis. Another reason for not trying to use type inference is that type inference and data flow analysis is a static technique and cannot take account of impossible to execute paths. The data flow analysis has to assume that data can flow along all the control flow paths present in the program. In practice, some paths may be impossible to execute. For example, an if-condition may check for an exceptional condition that should never occur in a correct program. In general, a static technique is not as accurate as a dynamic technique which uses information from the execution of the program with specific inputs.

It is relatively easy to check the context of a variable because it depends on the syntax of the program and can be checked without execution of the program. An algorithm for finding the context of an operator or operand is included in the Appendix A. This means that some redundant and type-mutants of a program can be avoided without executing the program or any of the mutants. This is a relatively fast way to eliminate type-mutants and redundant mutants.

4.5 Context Sensitive Mutations

This section gives rules for the application of the insertion and replacement mutation operators described in the previous sections. The rules depend only on the syntax of the program and they make use of the context.

4.5.1 Number Context Mutations

A number is obviously in a number context. A variable occurrence is in a number context if at compile-time it can be known that it will be converted to a number if it is not already a number. For example, the variables `x` and `y` are in a number context in the following expressions:

```
x - y
x * y
x / y
x % y
x > 0       //  at least one argument is a number
x != -1     //  at least one argument is a number
x++, x--, ++x, --x,
Math.abs(x), Math.floor(x), etc.
x = x – 3;
```

The variable of the left of an assignment is in a number context if the expression on the right of the assignment is in a number context. In general, the variable of the left of an assignment is in the context of the expression on the right of the assignment.

The operator `+` does not indicate a number context because `+` could be a string concatenation. If an operator is applied to literals then the types are known and so it is possible to determine the type of the operator. For example, in the code

```
x + “world”
```

the `+` operator is a string concatenation because whatever the type of `x`, it will be converted to a string. In general, the `+` is in a number context if both operands are in number contexts. The `+` is in a string context if at least one of the operands is in a string context. For example, in the code

```
(a – x) + (y – b)
```

the `+` is in a number context because the two operands of `+` are both in number contexts. If the `+` is not in a number context or a string context then it is not in any specific type context.

The non-strict relational operators e.g. `<`, `<=`, `>`, `>=`, `==`, `!=`, are in a string context if both of the operands are in a string context. The operators are in a number context if at least one of the operands is in a number context. In Chapter 3 it was explained that `"" == 0` is `true` because "" converts to number `0.` If the operands are both strings then no conversion takes place so `"" == "0"` is `false`. In addition, no conversion takes place for the strict equals test, `===` and `!==`. If the operator is not in a string or number context then it is not in any specific type context.

For an operand in a number context, the mutations are as follows: The number insertion mutations are applied to number literals but not all of the number insertion mutations are applied to all the number literals. Depending on the literal value, some mutations are equivalent or equivalent to each other. For example, if the literal is `0` then `abs(0) = 0`, so this mutant is equivalent to the original program. If the literal is `0` then `add1(0) = 1`, so this mutant is equivalent to the `c1(0) = 1` mutant. Where two mutation operators produce mutants that are equivalent to each other it is not necessary to apply both of them, one is sufficient. The number literals for which not all the number insertion operators are applied are given in Table 4.1.

Literal	Insertion mutation operators applied
`-1`	`sub1`
`0`	`sub1`
`1`	`add1, neg`
`2`	`add1, neg`

Table 4.1: For the literal numbers listed, the number insertion mutation operators that are not equivalent or equivalent to each other when applied to number literals shown. Assumes argument also replaced with constant 0 and 1 where not equivalent.

`add1` is equivalent to `c0` for the -1 argument because the result is `0` and that mutant is also produced by the `c0` mutation operator. `abs` and `negabs` are not applied to number literals, because if the literal is non-negative then `abs` will be an equivalent mutant and if the number is negative then `abs` will be the same as the `negabs` mutation.

Only number insertion mutations are applied to variables in a number context. There is no benefit to be gained from applying Boolean mutation operators `cFalse` and `cTrue` in a number context. `cFalse` replaces an operand with a `false` literal. The `false` will be converted to `0` in a number context. This is the same result as `c0` which replaces an operand with the `0` literal. This means that mutants produced by `cFalse` and `c0` are equivalent to each other in the number context. The mutants produced by the `cTrue` operator and the `c1` operator are also equivalent to each other in the number context.

The mutants produced by the `logneg` operator are not equivalent to any other mutant produced by the number mutation operators. The `logneg` operator can produce mutants that are partially equivalent to mutants produced by other number mutation operators. Recall the example from chapter 2 of two mutants that are partially equivalent to each other, i.e.

```
original       mutant1          mutant 2
x = x - 3;   x = c0(x) - 3;   x = logneg(x) - 3;
```

providing that there is an input that sets `x` always to a non-zero number or any other value which converts to `true`. In this situation `logneg(x)` is always `logneg(true)` which is the same as `false` which in an arithmetical expression is converted to `0` so `c0` and `logneg` can produce mutants that are partially equivalent to each other.

In order to detect that two mutants are partially equivalent to each other it is necessary to detect that the test produces the same output for both mutants. Sometimes this can be done by checking the values that the test produces at the argument to the mutant. For example, in the previous example mutants, it is necessary to test if a test always sets the argument `x` to a non-zero number or any other value which converts to `true,` then the test will produce the same output for both mutants. The value produced in a variable during execution with a specific test cannot be checked statically so it not possible to check if the `logneg` mutant is equivalent to the `c0` mutant. In the next chapter a dynamic method for mutation analysis is presented and in that method the values of arguments to mutation operators are checked at run-time. With dynamic mutation analysis, it can be checked when the `logneg` mutant is equivalent to the `c0` mutant.

Because the `logneg` may not be equivalent to any number mutation operator this is not by itself a sufficient reason to apply the operator in a number context. Notice that when the context is number the `logneg` operator has a function which maps its argument from non-zero to zero and from zero to 1, i.e.

```
logneg(non-zero) = logneg(true) = false = 0
logneg(zero)     = logneg(false)  = true = 1
```

If `logneg` is included as a number mutation operator then it should be justified because it produces a change to the behaviour of the original program that is sufficiently different from that produced by other number mutation operators. In fact, the non-zero part of the `logneg` function is performed by the `c0` function and the zero part of the `logneg` function is performed by the `c1` function. For this reason, it is considered likely that tests that kill the `c0` and the `c1` mutants will also kill the `logneg` mutant applied to a number.

The string mutation operators are not considered very beneficial mutations for a variable in a number context because `deadOnEmpty` is likely to be an equivalent mutant since a variable in a number context is unlikely to hold the empty string. Also replacing a number context variable with the empty string is the same as replacing the variable with `0` (the empty string is converted to `0`) and `c0` already does that mutation so `c0` and `cEmpty` are equivalent to each other. This means that any test that kills the `c0` mutant will also kill the `cEmpty` mutant and vice versa.

If a variable is in a number context then the only literal replacements are number literals. When replacing an operand with a variable the context of the replacement variable should be considered. The type context of a variable occurrence indicates that the variable either will hold a value of a given type or the value will be converted to a given type. It does not indicate the type of the value in the variable occurrence. However, it may be used as a heuristic for the type of the value in the variable.

In the simple case, all the occurrences of a variable are in the same single type context. In this case, it is reasonable to assume that the type of the context is the type of the value in the variable. For example, if all occurrences of a variable y are in arithmetical expressions, i.e. the number context, then it is sensible to assume that y holds a number.

A variable can have more than one occurrence and so the variable can occur in more than one type context.

For example,

```
if (y > 0) {        // y in number context
  . . .
  y + "hello";      // y in string context
  . . .
}
```

The above code shows occurrences of y in number and string context. There are a number of possibilities for the possible types of values held in each occurrence of y. The first occurrence of y probably holds a number but the second may hold any type because they all convert to string although the second occurrence probably does not hold an object or a Boolean.

From experience of JavaScript programs seen during the research for this thesis, most variables that at some occurrence hold a value of a simple type, number, Boolean or string, hold the same type at all occurrences, i.e. hold only one type of value.

If this single type assumption is made for the code example above then it does not restrict the possibilities very much for the type of y. Thus, y could still hold a number, a Boolean or a string. A Boolean is probably unlikely because after conversion the string `"true"` or `"false"` would be concatenated with another string and this does not seem to be a very useful operation but y could still be a number or a string.

The approach to static mutation proposed in this thesis is that the type context should be used to avoid mutations that are known to be type-mutations or have a high probability to be type-mutations and to avoid redundant mutations. If the variable occurrence x to be replaced is in a number context then it should not be replaced with a variable `y` that occurs only in string contexts. From the single type assumption, the variable `y` probably holds a string and only a string.

If the variably `y` is in number and string contexts then the probability of a type- mutation is lower. There is the possibility that when `y` is moved to the location of the variable occurrence to be replaced, x, the replacement variable `y` will hold only a number. In this case there is no type-mutation but otherwise a type-mutation will happen. If there is a reasonable probability that a mutation is not a type-mutation then it is proposed that the mutation should be allowed. The benefit of allowing the mutation is that if it is not a type-mutation and difficult to kill, it will force the tester to improve the test set. The benefit should be balanced against the cost. The cost of allowing type-mutations is not very high. If the type-mutation is incompetent it will be killed at the cost of some additional execution time. On balance, the benefit is considered to be higher than the cost.

A variable occurrence `x` in a number context is therefore replaced with a variable `y` providing all the occurrences of `y` are not in a non-number context. This means that if all occurrences of `y` are in one of the context types, Boolean, string or object then `y` is not used to replace `x`. Notice that this means that if there is at least one number context for `y` or `y` is not in any type context then `y` is used to replace `x`.

4.5.2 Boolean Context Mutations

The Boolean literals `true` and `false` are in the Boolean context. A variable occurrence is in a Boolean context if at compile-time it is known that it will be converted to a Boolean if it is not already a Boolean. A variable is in a Boolean context if it is the single variable expression in an if-statement, a while-statement or a conditional expression. For example,

```
if (x) { . . .
while(x) { . . .
x ? : “x is true” : “x is false”
```

Also, a variable is in a Boolean context if it is the single variable expression in a logical negation expression

```
!x
```

or if it is the left operand of a logical `&&` or `||`. In JavaScript

`x && y` is equivalent to
`if (x) return y; else return x;`

`x || y` is equivalent to
`if (x) return x; else return y;.`

In the above examples, `y` is not in a Boolean context unless it is contained in a conditional expression or statement. So for example,

```
c = x && y;         // y not in Boolean context, y not converted to Boolean
if (x && y) {       // y in Boolean context, y converted to Boolean
```

Only the Boolean mutation operators are applied to operands in a Boolean context. This means that Boolean literals are replaced with the opposite literal. A variable that is in a Boolean context is mutated with the Boolean literal replacement mutations, i.e.

`cFalse`: replace with constant false

`cTrue`: replace with constant true

The Boolean insertion mutation, `logneg`, is also applied to variables in a Boolean context as given below:

```
if (x && y) { ...  mutated to   if (!x && y) {  ...
if (x && y) { ...  mutated to   if (x && !y) {  ...
if (x) {  ...      mutated to   if (!x) {  ...
```

Notice that the ! operator converts its argument to a Boolean before negation.

Notice that any mutant that replaces a variable in a Boolean context with a non-Boolean literal is equivalent to one of the mutants produced by `cTrue` and `cFalse`. The non-Boolean literal will be converted to a Boolean constant `true` or `false`. These replacement mutations will also be produced by the Boolean operators `cFalse` and `cTrue`. In more detail, consider the following example of the occurrence of a variable `x` in an if-statement condition as in the code

```
if (x) { . . .
```

or the code

```
x && y
```

If the type of the value in `x` is not Boolean, it will be converted to Boolean. The mutation that replaces `x` with `0` (`c0`) is identical to the mutation that replaces `x` with `false` (`cFalse`) because `0` is converted to `false`. For the same reason, the mutation that replaces `x` with `1` (`c1`) is identical to the mutation that replaces `x` with `true` (`cTrue`) because `1` is converted to `true`.

A variable occurrence in a Boolean context is replaced by another variable from the program unless the other variable occurs only in non-Boolean type contexts. An exception is that no replacement is done if the result is a Boolean expression containing a Boolean operator with two equal variables. For example,

`x && y` does not mutate to `y && y` or to `x && x`
`x || y` does not mutate to `y || y` or to `x || x`

These mutations are not done because `x && x = x` and so it is equivalent to removing an operand and an operator. This is considered more than a small change to the program and therefore more likely to produce an incompetent mutant.

4.5.3 String Context Mutations

A string literal is in a string context. A variable occurrence is in a string context if at compile-time it is known that it will be converted to a string if it is not already a string. For example, the variables `x` and `y` are in a string context in the statements below:

```
x + "hello”
print(x);
y.charAt(index);
x.substring(index, length);  etc.
```

The operator `+` is in a string context if at least one of the operands is in a string context. In the code below both `+` operators are in a string context.

```
y + (x + "hello”)
```

The non-strict relational operators e.g. `<`, `<=`, `>`, `>=`, `==`, `!=`, are in a string context if both of the operands are in a string context.

The string mutation operator `cEmpty` is applied to every non-empty string literal, i.e. replace the literal with the empty string. The string insertion mutation operator i.e. `deadOnEmpty,` is applied if a variable is in a string context. The string insertion mutation is not applied to operands in any other contexts.

The Boolean operators are not applied in the string context. If the argument value to be mutated is a string then replacement by `true` and `false` is not redundant but they seem to be arbitrary string mutations. The Boolean values as strings, “true” and “false” are not special strings.

A variable occurrence in a string context is replaced with any other variable unless that variable occurs only in any of the non-string type contexts, i.e. number, Boolean or object.

4.5.4 Object, Array, Function and Member Context Mutations

There are two versions of the member or property access expression.

```
x.m
x["m"]
```

where the identifier `m` indicates a property of the object `x`. In the expression `x.m`, `x` must be an object and `m` is a property of `x`. Obviously, in `x.m` the `m` is fixed at compile time. In `x["m"]`, `m` is a computed property access expression and any expression is allowed in `[]`. In the expression `x["m"]`, `x` must be an object but it might also be an Array object. An array is a special kind of object. Arrays in JavaScript need not have number subscripts but can use named properties like objects because the prototype inheritance chain from every Array object eventually leads to the Object prototype.

The variable occurrence `x` in

```
x.m
```

is in an object context. The variable occurrence `x` in

```
x["m"]
```

is in what in this thesis is called an Array context even though `x` might not be an array. The array context is used as a short name for a computed property access expression.

A function is a special kind of object. A function call expression consists of a variable, which should have a function object as a value, followed by parentheses, i.e. `()` containing any arguments. The variable `x` in

```
x( . . .)
```

is in a function context.

The variable occurrence `m` in the expression

```
x.m
```

is in a member context. A property of an object can be another object so these definitions are recursive so for example, in the expression

```
x.y.m
```

x is in an object context, y is in an object context for m and also in a member context for x and m is in a member context.

A variable in an object context is not replaced with a literal object. Literal objects very rarely appear in member expressions. For example,

```
{house: 27,
 street: "Bayes",
 tel: {code: 01536,
       num: 519146},
}.street
```

It would be easier for the programmer to write just that part of the literal object accessed by the property in the member expression, i.e. "Bayes".

If the variable occurrence o is in an object context then it can be replaced with another variable unless the variable occurs only in non-object contexts, i.e. number, Boolean or string. If x is in a function context then it can be replaced with y only if y has an occurrence in a function context. This replacement condition is more strict than the previous types because a variable that is a function object almost always occurs in a function call context. This means that if a variable is not in a function call context then it is probably not a function.

If the variable occurrence m is in a member context, e.g. in the expression x.m, then it can be replaced only with a variable that has at least one occurrence in a member context. This replacement condition is more strict because a variable that is a property name can only occur in a member expression. For example, m can be replaced by n if y.n occurs in the program.

4.5.5 Any-type Context Mutations

A variable need not necessarily occur in a number, Boolean, string or an object context. In some contexts, no particular type can be determined and there is no-type conversion that must happen. If a variable is not in a number, Boolean, string, object, array or member context then it is said to be in the any-type context. For example, in the code

```
x = y;
```

both `x` and `y` are in an any-type context. In an any-type context, it is more likely that the variable `x` will hold a number, Boolean, string or object. `x` is unlikely to hold an array because in both an l-value context and in an r-value context an array usually occurs as an array expression with an index, i.e. `x[i]`. `x` is unlikely to hold a function because a function almost always occurs as a function call expression, i.e. `x()`.

When a variable is not in the context of a specific type it is in an any-type context. The issue is which mutation operators to apply. One possible choice is that a variable in an any-type context is not mutated because of the risk of producing a type-mutation. A type-mutation is likely to be easily killed and so it is likely to have little benefit because it does not force the tester to improve the test set. Although the benefit is likely to be small there is an extra cost in the execution of another mutant. The execution cost is relatively low if the mutant is killed by the first few tests. So the benefit of no mutation is a minor reduction in execution cost but the risk is that some effective mutants are not generated.

On balance, the benefit is considered to be of higher value than the cost. The approach proposed in this thesis is to mutate variables in an any-type context with number and string mutation operators. A variable in an any-type context is mutated by replacement with number literals and `c0` and `c1`. A variable in an any-type context is also mutated by replacement with the string literals which includes `cEmpty`.

The Boolean mutation operators are not applied to operands in an any-type context. The Boolean `logneg` insertion operation is not applied because when the context is not Boolean the `logneg` operator converts its argument to Boolean. As discussed earlier, in any context that is not a Boolean context, Boolean mutations are not considered useful.

A summary of the general rule for application of mutation operator to an operand occurrence is:

1. If an operand occurrence is in a type context `T` where `T` is one of number, Boolean, string or object type then the mutation operators for `T` are applied to the operand occurrence and the operand occurrence is replaced with program literals of type `T` and program variables except those variables that occur only in non-T contexts.
2. If an operand occurrence is in a type context `T` where `T` is one of Array or Function type then the operand occurrence is replaced only with program variables of type T. This means that an array only replaces and array and a function only replaces a function.
3. If an operand occurrence is not in a specific type context then number and string mutation operators are applied to the operand occurrence and the operand occurrence is replaced with number and string literals and program variables except those variables that occur only in non-number or non-string contexts, i.e. only in Boolean or object contexts.

4.6 Example to illustrate the Static Mutation Method

To illustrate how mutation operators would be applied in a program using the static mutation method, consider the example program under test given in Figure 4.1. This program has no purpose but uses variables that hold different types. Suppose that the occurrence of the variable `x` in `x - 3` is to be mutated.

```
function orig(n) {
   var x = 0, i = 0, r = 0;
   for (i = 0; i < n; i++) {
      if (i == 1) {
        x = true;     // test n = 2, 3 executes
      }
      else if (i == 2) {
         x = "hello";   // test n = 3 executes
      }
      r += x - 3;   // mutate x
    }
    return r;
}
```

Figure 4.1: An example used to illustrate the generation of mutants of program elements of different types. The variable x is assigned number, Boolean and string values depending on the input.

If three tests are executed by using test cases (`n = 1`), (`n = 2`) and (`n = 3`) then different types are assigned to this occurrence of `x` in each test. The occurrence of `x` is assigned number, Boolean or string values depending on the input.

Consider first the mutation of `x` in `x - 3` without the use of any type context information. When a no-type context is used, every variable is essentially considered to be in the any-type context. The following mutations are made.

```
add1 sub1 neg abs negabs zpush     number insertion mutations
deadOnEmpty                        string insertion mutations
c0 c1 2 3                          number literal mutations
cEmpty hello                       string insertion mutations
n  i  r                            variable replacement mutations
```

The result of executing the original program with the input `n = 3` is `NaN` because `NaN` is the result of `"hello" – 3.` The results of the mutants with the input `n = 3` are given as in the following:

```
 Mutated statement                  output when killed
r += c0(x) - 3;                  -9
r += c1(x) - 3;                  -6
r += add1(x) - 3;
r += sub1(x) - 3;
r +=  -(x) - 3;
r += Math.Abs(x) - 3;
r +=  -(Math.Abs(x)) - 3;
r += zpush(x) - 3;          exception
r += cEmpty(x) - 3;          -9
r += deadOnEmpty(x) - 3;
r += 2 - 3;                  -3
r += 3 - 3;                   0
r += "hello" - 3;
r += n - 3;                   0
r += i - 3;                  -6
r += r - 3;                -21
```

The mutants with no output shown are all live and all produce the same output as the original, i.e. `NaN`. The reason that the `r += deadOnEmpty(x) - 3` mutant is live is that this mutant is in fact an equivalent mutant since no input can set `x` to the empty string. `r += "hello" - 3;` is also an equivalent mutant because it always produces the output `NaN`. In fact all the live mutants shown above are equivalent.

Consider now the mutation of the same variable occurrence using type context information. The following mutations are made.

```
add1 sub1 neg abs negabs zpush     number insertion mutations
c0 c1 2 3                          number literal mutations
n  i  r                          variable replacement mutations
```

There are no string mutations because the context is recognised to be a number type. This means that 3 less mutants are generated. The mutants that are not generated are shown below

```
      Mutated statement                  output when killed
    r += cEmpty(x) - 3;          -9
    r += deadOnEmpty(x) - 3;
    r += "hello" - 3;
```

The first of these mutants is equivalent to `c0` because the function `cEmpty(x)` always produces the empty string and in a number context the empty string is converted to `0`. The second two mutants are equivalent to the original program. In this example, the advantage of the use of the type context has been demonstrated because some equivalent mutants and mutants equivalent to each other have not been generated.

There are `3` fewer mutants for the variable `x`. If all the operands and operators in the program are mutated for n = 3 without context information then there is a total of 208 mutants, 94 dead and 114 live. With context information, there is a total of 196 mutants, 88 dead and 108 live. There is not much difference in these results. There are about 5% fewer live mutants for the tester to examine. In practice the tester would examine just a few of these live mutants and create an additional test designed to kill these live mutants. The new test would then be executed and the process repeated.

The example program is not a realistic program because the output is `NaN` for any input that executes the `false` branch of the if-statement. It also means that most of the live mutants when executed with `n = 3` are equivalent mutants.

Later in the thesis, in an empirical evaluation, a set of sample programs are mutated and this gives more data about the effectiveness of the static mutation method.

In the next chapter a dynamic technique is presented for the discovery of type information at a variable occurrence. This type information is combined with the context of a variable occurrence.

Chapter 5: Typed Mutation of Dynamically Typed Programs

5.1 Introduction

In the previous chapter, a static approach was taken to the mutation of dynamically typed programs. The rules for the construction of mutants, the mutation operators, are syntax sensitive only and some simple static type information from the context is used to select suitable mutations. It is relatively easy to check the context of a variable because it depends on the syntax of the program and can be checked without execution of the program. This means that some equivalent and type-mutants of a program need not be constructed. This saves the cost of having to execute the mutants and the work of the tester is easier if there are fewer live mutants to examine.

With the static approach, it is possible that some type-mutants will be generated because the type context allows only an assumption about the type of a variable occurrence. If the type of the value held in a variable occurrence is tested at run-time then the type is known accurately and all type-mutations can be avoided. This chapter considers a dynamic approach to the mutation of dynamically typed programs. The aim is to discover information about the values held in particular variables in order to avoid type-mutations and equivalent mutations.

5.2 Typed Application of Mutation Operators

In the static mutation analysis method the choice of mutation operators to apply to an operand depends on the type context of the operand. The choice of variables to replace the operand depends on the context in which the variable occurs. The type context is used as a heuristic for the type of the variable value. With the benefit of type information, the types held at each occurrence of the variable are known precisely. This allows the mutation operator application rule to be sensitive to the type of the value to be mutated as well as the context.

Consider an example of a variable occurrence `x` in an assignment statement such as

```
y = x;
```

There is no specific type context for `x`. Consider that the program has been executed with a test and the types of the values held in `x` are recorded as the test executes. The details of how types are recorded are explained later in this chapter. Suppose that during the execution of the test only number values are held in `x`. In this situation, to avoid type-mutations, only the number mutations should be applied to `x`. In the static mutation method described in the previous chapter, because `x` is not in any specific type context, then number and string mutations would be applied and the string mutations would be type-mutations. So run-time type information makes it possible to avoid type-mutations.

If during the execution of a single test case both number and string values are recorded in `x` then in the dynamic mutation method both number and string mutations can be applied. To avoid type-mutations however, this means that number mutations should not be applied to `x` when `x` holds a string and string mutations must not be applied to `x` when `x` holds a number. This means that the number mutations should be applied only when `x` holds a number and the string mutations should be applied only when `x` holds a string. This means that the algorithm for the dynamic meta-mutant for the statement `y = x;` should be

```
if (mutate(x)) {
  if (typeof(x) == "number") {
    y = NumberMutation(x);
  }
  else if (typeof(x) == "string") {
    y = StringMutation(x);
  }
}
else {
  y = x;  // no mutation of x
}
```

In practice, the types of the values held in `x` are not known until the program is executed on each test case so the meta-mutant must be more general and allow for every type of value in each variable occurrence of the program. This is done by adding a test for the type of a value in each mutation operator. For example the dynamic `add1` and `cEmpty` mutation operators are defined as

```
add1(x) {
  if (typeof(x) == "number") {
    return x + 1;
  }
  else {
    return x;  // no mutation if x not number
  }
}

cEmpty(x) {
  if (typeof(x) == "string") {
    return "";
  }
  else {
    return x;  // no mutation if x not string
  }
}
```

The dynamic meta-mutant for `y = x;` is then defined as

```
switch (mutant) {
  case add1:
    y = add1(x);
    break;
  case cEmpty:
    y = cEmpty(x);
    break;
  default:
    y = x;    // no mutation of x
}
```

By using run-time type information it is possible to avoid generating some equivalent mutants. For example, the mutant `cEmpty`(x) will be equivalent to `x` if `x` never holds a string or always holds the empty string.

When the types of the values held in x are recorded, it is just as easy to test that x is an empty string or a non-empty string as it is to test that x is a string. This is the same as dividing the type of string into two subtypes, empty and non-emtpy string. The benefit of this is that if for a test only the empty string is recorded in x then the cEmpty mutation is equivalent for that test. This means that the cEmpty mutation need not be generated for that test. It is efficient to detect mutants that are partially equivalent to each other so that if one mutant is executed by a test for which the two mutants have the same input-output behaviour then the other mutant execution is not necessary because it will have the same output.

To detect that two mutants are partially equivalent to each other it is necessary to detect that the test produces the same output for both mutants. It is more efficient if two mutants can be detected as equivalent to each other by checking the values that the test produces at the mutated expression. If the value of a variable in the original program is always found to be positive, for example, then the abs() mutation is equivalent and can be avoided.

The following subtypes of the number type are useful for not generating equivalent number mutants.

Subtype	Definition
zero,	variable holds 0
one,	variable holds 1
two,	variable holds 2
negone,	variable holds -1
positive,	variable holds positive number except 1 or 2
negative,	variable holds negative number except -1

The zero subtype can be used to avoid generating the c0, neg, abs and negabs equivalent mutants. In addition, when the value to be mutated is zero, c1 and add1 are partially equivalent to each other. The one subtype can be used to avoid generating the c1, abs and zpush equivalent mutants. In addition, when the value to be mutated is one, c0 and sub1 are partially equivalent to each other. The positive subtype can be used to avoid generating the `abs` equivalent mutant and the `negative` subtype can be used to avoid generating the `negabs` equivalent mutant.

There are two subtypes of Boolean, `true` and `false`. For example, if for a test a variable only holds the Boolean `true` value but not the `false` value then the `cTrue` mutant is equivalent.

The basic difference between the static mutation method of the previous chapter and the dynamic mutation method described in this chapter is that static mutation modifies the syntax of the program. The modification to generate a mutant can be made before the program executes. With the dynamic mutation method, the modification that generates a mutant must be made at run-time because it depends on the type or subtype of the value in the expression to be mutated.

In the static mutation method, in particular contexts, some mutations are equivalent to each other. The effect of the context was described in the previous chapter and it needs to be considered also for dynamic mutation. For example, if the variable `x` is in a Boolean context and a test case places number values in `x` then the mutation of `x` should be determined not just by the number value but also the Boolean context. For example, the mutants produced by the `add1` mutation operator are not equivalent in general. However, if `x` is in a Boolean context and the type of the value held in x, for example, is positive then a positive number and one plus a positive number are both converted to `true` and so are equivalent to each other.

5.3 Mutation Analysis Process for Dynamically Typed Programs

In order to be able to use mutation analysis for dynamically typed programs in a type-sensitive manner, the algorithm given in Figure 5.1 can be used to implement the mutation analysis for a program `P`.

```
// initialise a table of mutants for each operator and operand
foreach (operatorOperand e in P) {
  mutants(e) = {};
}
// generate and execute mutants
foreach (test t) {
  // execute P on t, save output
  orig = execute(P, t); // also record types assigned to all
variables
  foreach (operatorOperand e in P) {
    // define mutants of P for types found with test t
    mutants(e) = mutants(e) union generateMutants(e, t);
  }
  foreach (operatorOperand e in P) {
    foreach (mutant m in mutants(e)) {
      if (not killed(m)) {
        mut = execute(meta(P, m), t);
        if (mut != orig) {
          killed(m) = true;
        }
      }
    }
  }
}
```
Figure 5.1: Dynamic mutation analysis algorithm for generating mutants of a program `P`.

Each test is executed on the original program `P` and the output recorded. The types held in variables are also recorded. Using the type information, the mutants of every statement reached by the test can be generated. If these mutants have not been previously generated then they are added to a global table. Each mutant reached by the test is then executed with the test as input. `meta(P, m)` denotes the meta-mutant parameterised to execute the mutant `m`.

The process of tests enhancement of dynamically typed mutation is depicted in Figure 5.2. Mutants are generated after executing a test case against the program under test to reveal the types of the elements to be mutated and the test output for the original program. The mutants generated from the first test are executed and the produced output is compared with the output of the original program. If the output of the original program is not equal

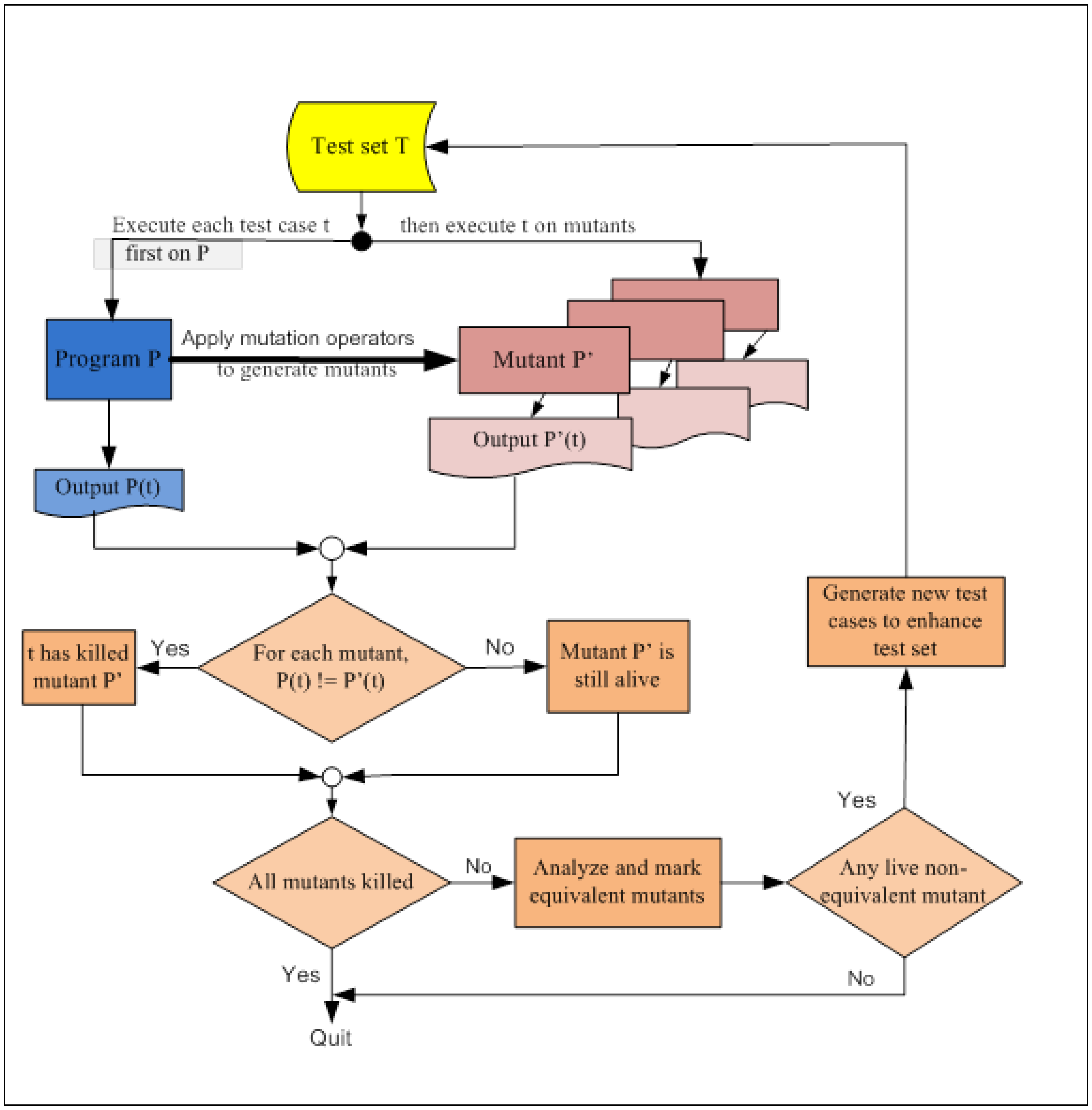

Test set T
Execute each test case t first on P
then execute t on mutants
Program P
Apply mutation operators to generate mutants
Mutant P'
Output P'(t)
Output P(t)
t has killed mutant P'
Yes
For each mutant, P(t) != P'(t)
No
Mutant P' is still alive
Generate new test cases to enhance test set
Yes
All mutants killed
No
Analyze and mark equivalent mutants
Any live non-equivalent mutant
Yes
No
Quit

Figure 5.2: Enhancement of a test set T for a program **P**

to the output of the mutant, or the mutant throws an exception or the mutant program is terminated because it has executed longer than a timeout value then the mutant is killed. The killed mutants are not executed again but the live mutants will remain with the hope that they will be killed by some later test. The adequacy score is calculated as the percentage of non-equivalent mutants that have been killed. The process stops when the score is 100% or before if the test budget has been consumed. In order to improve the adequacy score, new test cases are introduced with the aim of killing more mutants. This may result in generating new mutants because the test may cover new statements or generate new types at variables compared to previous tests.

In the following sections, the mutants of each type of value are described according to the context in which the mutated variable occurs.

5.3.1 Number Context Mutations

If a variable occurrence `x` is in a number context and for a particular test the type recorded at `x` is one of the number subtypes then the mutants generated for each type recorded at `x` are listed below:

type	mutation operator							
	`c0`	`c1`	`add1`	`sub1`	`neg`	`abs`	`negabs`	`zpush`
`zero`		*		*				*
`one`	*		*		*			
`two`	*	*	*		*			
`negone`	*	*		*				
`positive`	*	*	*	*	*			
`negative`	*	*	*	*	*			

If two or more of the subtypes are recorded at a variable then the operators listed for each type are combined. The reason that `abs` and `negabs` are not included is that they generate equivalent mutants or mutants that are equivalent to some other mutant. For example, for the positive type, `neg` and `negabs` are equivalent to each other. Also, when two or more of the subtypes at a variable occurrence contain both positive and negative numbers, e.g. `negone` and `positive`, then the `abs` and `negabs` mutation operators are also applied because these operators no longer generate equivalent mutants.

If a variable occurrence `x` is in a number context and for a particular test the type recorded at `x` is one of the Boolean subtypes then the mutants generated for each type recorded at `x` are listed as in the following:

type	mutation operator		
	cFalse	cTrue	logneg
true	*		
false		*	

Notice that logneg is partially equivalent to both cFalse and cTrue. This is because if a variable always holds a false then cTrue is equivalent to logneg. If both the true and false subtypes are recorded at a variable then the cFalse, cTrue and logneg operators are applied.

If a variable occurrence x is in a number context and for a particular test the type recorded at x is one of the string subtypes then the mutants generated for each type recorded at x are listed below:

Type	mutation operator	
	cEmpty	deadOnEmpty
empty		*
nonempty	*	

If both the empty and nonempty subtypes are recorded at a variable then all the string operators are applied.

If, during the execution of a single test, a variable holds values of two different basic types, e.g. number and Boolean, then both the number and Boolean mutations are applied. Because the mutations are applied to the values, not the variable, number mutations are applied to number values only and Boolean mutations are applied to Boolean values only. This is different to the static mutation method. Recall that in the static mutation method that c1 and cTrue are equivalent to each other in the number context. In the dynamic mutation method, c1 and cTrue are not equivalent to each other in the number context. To explain this consider a variable occurrence x that when executed with a specific test, holds a zero followed by a false. The following is an example program that does this.

```
var i = 0, x = 0, r = 0;
for (i = 0; i < 2; i++) {
  if (i == 0) {
    x = 0;
  }
  else {
    x = false;
  }
  r += x * i;  // x holds both 0 and false
}
```

In the meta-mutant, the statement to be mutated, i.e. `r += x * i;` is replaced with the code shown below.

```
switch (mutant) {
  case c1:
    if (typeof(x) == "number") {
      r += 1 * i;     // constant 1
    }
    else {
      r += x * i;
    }
    break;
  case cTrue:
    if (typeof(x) == "boolean") {
      r += true * i;    // constant true
    }
    else {
      r += x * i;
    }
    break;
 }
```

The `c1` mutation will mutate the zero held in `x` into a one but not mutate the `false` because it is not a number type. This means that the values produced at `x` in the `c1` mutant are `1` followed by `false`. After conversion of the `false` into `0` because of the number context, the values produced at `x` in the `c1` mutant are `1` followed by `0`. The final value of r is therefore `1 * 0 + 0 * 1 = 0`.

The `cTrue` mutation at `x` will not mutate the number but will mutate the `false` into `true`. This means that the values produced at `x` in the `cTrue` mutant are `0` followed by `true`. After conversion of the `true` into `1` because of the number context, the values produced at `x` in the `c1` mutant are `0` followed by `1`. This is a different sequence of values to the `c1` mutant. The final value of r is therefore `0 * 0 + 1 * 1 = 1`. The two mutants produce different outputs, i.e. mutants not equivalent to each other. In general, mutants produced by mutation operators of different basic types are not equivalent to each other.

The variable replacement mutations are also type-sensitive. In a number context, an occurrence of a variable `x` that holds a number type is mutated by replacing the number value in `x` with a number value from some other variable in the program. In the static mutation method, the variable occurrence `x` is replaced with some other variable, say `y`. This means that the value in `x` is replaced with the value in `y` and every access of `x` is replaced with an access of `y` no matter the type in `x` and `y`.

In the dynamic mutation method, only when the type in `x` is equal to the type in `y`, the value in `x` is replaced with the value in y. In more detail, consider again the previous program containing an occurrence of a variable `x` that is to be mutated.

```
var i = 0, x = 0, r = 0;
for (i = 0; i < 2; i++) {
  if (i == 0) {
    x = 0;
  }
  else {
    x = false;
  }
  r += x * i;  // x holds both 0 and false
}
```

The types held in `x` in `r += x * i;` have been recorded as `{zero, false}`. The program also contains some occurrences of another variable `i` and the types recorded at `i` are `{zero, one, two}`. Because `x` holds a number type and `i` holds a number type, `i` is identified as a possible replacement mutation for x. During execution of the replacement mutant, the type of the value in i is checked and if it is a number type then the value in i is copied to x and a value replacement mutation has been performed. If the value held in i is not a number type then no replacement, i.e. no mutation is made.

```
switch (mutant) {
  case replace_i:
    if (typeof(x) == typeof(i)) {
      r += i * i;     // x replaced by i
    }
    else {
      r += x * i;     // x not replaced
    }
    break;
 }
```

In the previous program, the zero in x is replaced but the false in x is not replaced. This ensures that there are no type-mutations produced during variable for variable replacement mutations.

The number subtypes containing just one number, i.e. zero, one, two and negone, are used to avoid equivalent mutants. This means that if the only number type recorded at the variable occurrence x is 2, for example, and the literal 2 appears in the program then x is not replaced with this literal because obviously it is equivalent. Also, if the type at the variable occurrence x is just one of the subtypes, zero, one, two or negone and the type of the replacement variable y is the same type as x then the mutation is equivalent because both x and y hold just one number.

Notice that in the number context, all the mutation operators can be applied if there is a value to be mutated of every type. In the static mutation method, the Boolean operators are not applied in a number context. This is because in the static method a mutation operator is applied statically, i.e. it is a static change to the program and the mutation operator is applied to all the values held in the mutated variable and this means that Boolean mutation operators could be applied to number values. In the dynamic mutation method the mutation operator is applied only to values of the type of the operator. This means that the number context does not affect which mutation operators to apply to a variable.

5.3.2 Boolean Context Mutations

Number values in a Boolean context are converted to Boolean. When number mutation operators are applied to number values, although the value is modified, because of the mutation in a Boolean context, the final result may be equal to the original value. For example, if the `add1` mutation is applied to a positive number, the resulting number is different but both original and mutated numbers are converted to the same value, `true`. Only zero is converted to false.

If a variable occurrence `x` is in a Boolean context and for a particular test the type recorded at `x` is one of the number subtypes then the mutants generated for each type recorded at `x` are listed below:

type	mutation operator							
	c0	c1	add1	sub1	neg	abs	negabs	zpush
zero		*						*
one	*							
two	*							
negone	*							
positive	*							
negative	*							

Notice that there are many fewer mutations than when `x` is in a number context. The mutation operators not selected above produce equivalent mutants in the Boolean context.

If a variable occurrence `x` is in a Boolean context and for a particular test the type recorded at `x` is one of the Boolean subtypes then the mutants generated for each type recorded at `x` are listed as in the following:

type	mutation operator		
	cFalse	cTrue	logneg
true	*		
false		*	

If both the true and false subtypes are recorded at a variable then the cFalse, cTrue and logneg operators are applied.

If a variable occurrence x is in a Boolean context and for a particular test the type recorded at x is one of the string subtypes then the mutants generated for each type recorded at x are listed below:

type	mutation operator	
	cEmpty	deadOnEmpty
empty		*
nonempty	*	

If both the empty and nonempty subtypes are recorded at a variable then all the string operators are applied.

Replacement of values for a variable occurrence in a Boolean context can produce equivalent mutants. For example, assume x holds a number and y holds a different number and the number in y replaces the number in x. In a number context the value of x has changed which means it is possible to kill the mutant. In a Boolean context, if the numbers in x and y are both non-zero then they both converted to true and the value of x has not changed and so the mutant is equivalent. In general, a replacement mutation in a Boolean context is equivalent if the original and the replacement value both convert to the same Boolean value. Below is listed all the values that convert to false.

```
NaN, null, undefined, "", 0, false
```

For a replacement not to be equivalent, a value that converts to true must be replaced with a value that converts to false and vice versa. Replacement mutations that are known to be equivalent are not generated for that test.

5.3.3 String Context Mutations

If a variable occurrence `x` is in a string context and for a particular test the type recorded at `x` is one of the number subtypes then the mutants generated for each type recorded at `x` are listed below

type	mutation operator							
	c0	c1	add1	sub1	neg	abs	negabs	zpush
zero		*		*				*
one	*		*		*			
two	*	*	*		*			
negone	*	*		*				
positive	*	*	*	*	*			
negative	*	*	*	*	*			

These are the same mutations that are applied to a number in a number context. Number values in a string context are converted to string but this does not make any of the number mutations equivalent. If two or more types are recorded at a variable then the operators listed for each type are combined. Also, when two or more types contain both positive and negative numbers, e.g. `negone` and `positive`, then the `abs` and `negabs` mutation operators are also applied.

If a variable occurrence `x` is in a string context and for a particular test the type recorded at `x` is one of the Boolean subtypes then the mutants generated for each type recorded at `x` are listed below:

type	mutation operator		
	cFalse	cTrue	logneg
true	*		
false		*	

If both the `true` and `false` subtypes are recorded at a variable then the `cFalse`, `cTrue` and `logneg` operators are applied.

If a variable occurrence x is in a string context and for a particular test the type recorded at x is one of the string subtypes then the mutants generated for each type recorded at x are listed below:

type	mutation operator	
	cEmpty	deadOnEmpty
empty		*
nonempty	*	

If both the empty and nonempty subtypes are recorded at a variable then all the string operators are applied.

In a string context, an occurrence of a variable x that holds a string type is mutated by replacing the string value in x with a string value from some other variable in the program. Only when both x and the replacement variable hold a string value is replaced. Equivalent mutants for a specific test can be detected if the string type of x is just a single empty string type and the same string subtype is the only type held in the replacement variable.

5.3.4 Object Context Mutations

The variables within an object, known as properties, are accessed using member expressions. A member expression such as obj.mem is mutated by replacing the object variable obj, and replacing the member name mem. For the mutation to be typed, any replacement for obj should be an object with a member called mem.

Any replacement for the member name mem should be the name of another member of obj. If the replacement member name was not a member of obj then the value of the member expression is the special value undefined. The replacement property for mem should also have a value which has the same type as the value of mem. In the example code

```
var residence = {house: 27,
                 street: “Bayes”,
                  tel: {code: 01536,
                        num: 519146},
                };

var school = {name: “Stamford Rd Boys”,
              street: “Stamford Rd”,
             };
```

The expression `residence.street` can be mutated to `school.street` because the object `school` has a property called `street` and the type of the `street` property in the `school` object is the same as the type of the `street` property (string) in `residence`. When the program is executed with a specific test to collect type information, if the value of a variable is an object then the object property names and object property types are recorded. For example, in the above code, the following property types are recorded.

```
{house: number,
 street: string,
 tel: {code: number,
       num: number},
};

{name: string,
 street: string,
};
```

In JavaScript an array access expression is very similar to an object member access expression. There is a dynamic form of object member access expression, e.g. `obj.mem` is equivalent to `obj[“m” + “em”]`. The array index expression or the member expression is mutated in the way that any expression is mutated. The object or array is mutated in the way that the object of a member expression is mutated. Consider the array expression `a[i – 2]`. The expression `i - 2` would be mutated as an arithmetical expression. The array `a` would be replaced by another object with a member indexed by `i – 2` that has the same type.

5.3.5 Any-type Context Mutations

If a variable is not in a number, Boolean, string, object, array or member context then it is said to be in the any-type context. In the any-type context there is no-type conversion that must happen. In the case of the static mutation method there is the risk of producing a type-mutation. In the static mutation method, the Boolean mutation operators are not applied because of the possibility of a type-mutation. In the dynamic mutation method, the types of the values to be mutated are known at run-time and so type-mutations can be avoided. This means that if the value to be mutated is a Boolean then Boolean mutations will be applied.

5.4 Operator Replacement Mutations

The `+` operator is overloaded between number addition and string concatenation. With type information, it is possible to avoid the replacement of a string concatenation `+` by arithmetic for operators such as – for example. The `+` operator is replaced with other arithmetic operators only if both operands are numbers. For example for the statement

```
x = x + y;
```

the context is not string or number and so `+` could be an arithmetical or string operator. In this situation the meta-mutant is defined as

```
switch (mutant) {
  case subtract:
    if (typeof(x) == “number”
        && typeof(y) == “number”) {
      x = x - y;     // + replaced by -
    }
    else {
      x = x + y;     // + not replaced
    }
    break;
  case multiply:
     . . .           // other cases here
 }
```

5.5 Dynamic Mutation Example

To illustrate how typed mutation operators would be applied in a program, consider the example program given in Figure 5.3. Suppose that the occurrence of the variable `x` in `x + 3` is to be mutated. This program is the same as the example program from Chapter 4 but + replaces – so that the type context for + is ambiguous between number and string.

```
function orig(n) {
   var x = 0, i = 0, r = 0;
   for (i = 0; i < n; i++) {
      if (i == 1) {
        x = true;     // test n = 2, 3 executes
      }
      else if (i == 2) {
         x = "hello";   // test n = 3 executes
      }
      r += x + 3;   // mutate x
    }
    return r;
}
```

Figure 5.3: An example used to illustrate the generation of mutants of program elements of different types. The variable `x` is assigned number, Boolean and string values depending on the input.

```
function orig(n n.0) {
   var x x.0 = 0, i i.0 = 0, r r.0 = 0;
   for (i i.1 = 0; i i.2 < n n.1; i i.5 ++) {
      if (i i.3 == 1) {
        x x.1 = true;     // test n = 2, 3 executes
      }
      else if (i i.4 == 2) {
         x x.2 = "hello";   // test n = 3 executes
      }
      r r.1 += x x.3 + 3;   // mutate x
    }
    return r r.2;
}
```

Figure 5.4: Labelled variables occurrences for the program given in Figure 5.3.

In order to mention specific variable occurrences, each variable occurrence in the above program is labelled with a unique label so that each occurrence can be uniquely identified.

The label consists of the variable name followed by a dot '.' and then a number which is incremented for each occurrence. The program with each variable occurrence followed by its label is given in Figure 5.4. It is easier to read the program when the labels are shortened by remove the variable name from the label. For example, the program in Figure 5.5 is more easy to read when shown as in Figure 5.4.

```
function orig(n .0) {
   var x .0 = 0, i .0 = 0, r .0 = 0;
   for (i .1 = 0; i .2 < n .1; i .5 ++) {
      if (i .3 == 1) {
        x .1 = true;     // test n = 2, 3 executes
      }
      else if (i .4 == 2) {
         x .2 = "hello";   // test n = 3 executes
      }
      r .1 += x .3 + 3;   // mutate x
    }
    return r .2;
}
```

Figure 5.5: Labelled variables occurrences for the program given in Figure 5.3 but with the shorter labels that are easier to read.

With the string or number context for `x` in `x + 3` the following mutants are generated using the static mutation method.

```
add1 sub1 neg abs negabs zpush    number insertions
deadOnEmpty                       string insertions
c0  c1  2   3                     replacement with number literal
cEmpty  hello                     replacement with string literal
n   i   r                         replacement with variable
```

Notice that the possible string context for the `+` operator means that the string mutations are included. The objective of the dynamic mutation method is to get type information from the execution of a test case and if possible avoid type-mutations and equivalent mutants.

Assume that the program given in Figure 5.3 is to be executed with the test `n = 1`, `n = 2`, and `n = 3`.The program is executed with the first test (`n = 1`) and the types of the values at each variable occurrence are recorded in a table. Not all variable occurrences are executed when `n = 1` but those that are executed are shown below with the recorded types.

```
x.0    zero
i.0    zero
r.0    zero
i.1    zero
i.2    zero one
n.1    two
i.3    zero
i.4    zero
r.1    zero positive
x.3    zero
i.5    zero
```

The occurrence of `x` in `x + 3`, i.e. `x.3`, holds a `zero` number only. Using this type information it is possible to reduce the mutations at the `x.3` occurrence of the variable `x` from

```
add1 sub1 neg abs negabs zpush     number insertions
deadOnEmpty                        string insertions
c0  c1  2   3                      replacement with number literal
cEmpty  hello                      replacement with string literal
n   i   r                          replacement with variable
```

to the following:

```
sub1 zpush            number insertion mutations
c1 2  3               literal replacement mutations
n  i  r               replacement with variable
```

The mutant `add1` is absent because it is partially equivalent to `c1`, `add1(x) = c1(x)` when `x = 0`. The mutants `neg`, `abs`, and `negabs` are all equivalent to the original program. The mutant `c0` is also equivalent to the original program. For the same reason, the replacement with the string literals is excluded.

Because `x.3` holds a number, it is replaced with the other number literals that occur in the program i.e. `2` and `3`. Because `x.3` holds a number, it is replaced with any other variables in the program that also hold number values, i.e. `n`, `i`, and `r`. These variables have been identified to hold number values because at least one of their occurrences has been recorded as holding a number value. For example, `n.1` hold a number, `i.0` holds a number and `r.1` holds a number.

Notice that the use of type information has avoided the generation of the three string mutations. The `deadOnEmpty(x) + 3;` is an equivalent mutant for this test and so it is beneficial to avoid this mutant for this test. The `cEmpty(x) + 3;` mutant is not equivalent and also the mutant `"hello" + 3;` is not equivalent. Both of these mutants are type-mutations. Type-mutations are generally considered incompetent, i.e. easily killed and both of these mutants are killed by the test `n = 1`. This is consistent with being incompetent.

With the test case `n = 1`, all the dynamic mutants of `x.3` are killed except the replacement by `r` and the replacement by `i`. The replacement by r and the replacement by `i` mutants are equivalent with dynamic mutation. This is because both r and `i` are initialised to `0` and `x` is initialised to zero. Except for `0`, the only other values held in `x.3` are `true` and “`hello`”. These are not number values and so the replacement of `x.3` with `i` and r is not done when `x.3` holds `true` and “`hello`”.

The result of executing the mutants with the input `n = 2` associates the same types to the variables associated with `n = 1` except that `x.1` occurrence is now reached and the Boolean type is also held in `x.1` and `x.3`. The new variable type occurrence information is

```
x.0   zero
i.0   zero
r.0   zero
i.1   zero
i.2   zero  one  two
n.1   two
i.3   zero  one
x.1   zero  true
i.4   zero
r.1   zero  positive
```

```
x.3   zero  true
i.5   zero  one
```

Because the Boolean type is now held in `x.3` Boolean mutations are added to the mutations of the previous test. Only the Boolean subtype `true` is recorded at `x.3` and so only the mutation operator `cFalse` is applied. This is because the `cTrue` mutation will produce an equivalent mutant. This means that the mutants of `x.3` are now

```
  sub1 zpush                  number insertion mutations
  cFalse                      replace Boolean with false
  c1 2  3                     literal replacement mutations
  n  i  r                     replacement with variable
```

The new `cFalse` mutant generated from the intput `n = 2` is also killed by the input `n = 2`. The live mutants are still the replacement by `r` and the replacement by `i`.

When the test, `n = 3` is applied the types recorded at the variable occurrences are

```
x.0   zero
i.0   zero
r.0   zero
i.1   zero
i.2   zero one two positive
n.1   positive
i.3   zero one two
x.1   zero true
i.4   zero two
x.2   strngnonempty true
r.1   zero positive strngnonempty
x.3   zero strngnonempty true
i.5   zero one two
```

There are new types held in `i.2`, `n.1`, `i.3`, `i.4`, `r.1`, `x.3` and `i.5`. The variable occurrence `x.2` is reached for the first time. At `x.3` a non-empty string type is recorded and this causes string mutations to be generated for `x.3`. The mutations at `x.3` for `n = 3` are

```
sub1 zpush                  number insertion mutations
cFalse                      replace Boolean with false
cEmpty                      replace string with empty string
c1 2  3                     literal replacement mutations
n  i  r                     replacement with variable
```

Notice that only the `cEmpty` mutant is generated because it can be predicted from the non-empty string subtype recoded at `x.3` that the `deadOnEmpty` mutant cannot be killed. The output for each killed mutant is shown below

```
r += c1(x) + 3;       8hello3
r += sub1(x) + 3;     6hello3
r += zpush(x) + 3;    killed
r += cfalse(x) + 3;   6hello3
r += cEmpty(x) + 3;   73
r += 2 + 3;           9hello3
r += 3 + 3;           10hello3
r += n + 3;           10hello3
r += i + 3;
r += r + 3;
```

Notice that the `cEmpty` mutant is killed.

5.6 Type Discovery

To apply type-sensitive mutation operators, it is necessary to know the type of program elements. In the previous chapter it was explained why static type inference is not suitable for a program written in JavaScript. In general, type discovery using program execution is more accurate than static type discovery because the type is known for a specific test at run-time. Although it is more accurate, it is less comprehensive because it is limited to the inputs for which the program has been executed. Therefore, the mutants will be generated and executed for the most accurate type that occurs with a given test rather than a possibly more general type that covers all possible inputs. For example, if a variable occurrence `x` is number type for test t1 then only number type-mutants are executed with t1. If the same occurrence of `x` is string type for test t2, then only string mutants are executed for test t2. Static type inference will discover only that `x` is both number type and string type, but not necessarily the inputs for which `x` is only number and only string.

If types of variables are required to compile the code in the most efficient way then static type discovery is most appropriate. It is efficient and can be used on large programs. However, static type discovery is most accurate for strongly typed languages. Mutation analysis is a different situation to compilation. Mutation analysis is a testing process and testing is not expected to be as fast as a program compilation. It is expected that there will be many executions of the program when a program is tested. In general, to determine the type of a variable occurrence using static type inference is less expensive compared to dynamic method of type discovery. However, the cost of mutant execution dominates the cost of type discovery so the saving gained by using static type discovery is likely to be small.

To determine the types of the values held in each variable occurrence of the program instrumentation code is added to the original program. The instrumentation code does not affect the output of the program but instead records the types of values in variables. The type information is collected while the program is executed with a specific test. The extra information for the number subtypes is only slightly more expensive to collect than the simple type number but it is useful to detect mutants that are equivalent to the original program or to each other. The same is `true` for the Boolean and string subtypes.

The program under test can be automatically instrumented. First, each variable occurrence is identified by a label to distinguish the occurrence of the variable. The abstract syntax tree of the program under test is traversed and a unique label is assigned to each variable occurrence encountered during the traversal. For example, to record the type of a value held in each variable occurrence, each assignment statement in the program under test is automatically rewritten as follows

`m = i;` is transformed into the comma expression

```
recType(m, "m.0"), m = recType(i, "i.0"), recType(m, "m.0");
```

The first part of the comma expression, i.e. the function `recType(m, "m.0")`, tests the type of the value in the variable m and stores the result in a table under the key `"m.0"` which is the unique label assigned to the occurrence of the variable `m`.

```
recType(variable, label) {
  typeTable.get(label).unionWith(typeof(variable));
  return variable;
}
```

The first `recType(m, "m.0")` in the comma expression stores the type of the value of the variable m before the assignment to `i`. The reason for this is that m may be a non-local variable and so might be assigned a value outside of the function under test. The second part of the comma expression, i.e.

```
 m = recType(i, "i.0")
```

is evaluated next and this part records the type of the variable `i` under the label `"i.0"` and assigns the value of `recType(i, "i.0"),` which is the same as the value of `i`, to `m`. The final part of the comma expression records the type of the value in `m` after the assignment. In a dynamic language, the type of value in `m` before the assignment may be different to the type in `m` after the assignment.

Expressions that assign values to properties of objects are instrumented in a similar way. Each component of an object member access expression, i.e. in the expression

```
obj.prop
```

the property `prop` of object `obj` is given a unique label.

Although the variable occurrence type table records the set of types stored in a variable occurrence, it is not known when during the execution that each type is assigned. A variable may first hold a number followed by another number followed by a string and then a number again. For this reason, during the execution of the meta-mutant, every time a variable occurrence is used it is necessary to test the type of the value in the variable occurrence in order to select a type-correct mutation.

Consider how the meta-mutant implements a variable replacement mutation. Each r-value variable occurrence that is mutated with a variable replacement is implemented in the meta-mutant as a function coded as follows:

```
function mutateVariable(varName) {
  switch (varName){
    case “x”:
      return x;   // no mutation
    case “y”:
      if (typeof(x) == typeof(y)) {
        return y;   // mutate x to y if same type
       }
       else {
         return x;   // avoid type-mutation
       }
    case “z”:
      . . .  // one case for each variable in
  }          // function under test
}
```

Notice that the function is generated according to a template, i.e. a switch statement. To fill in the template the variables that occur in the program under test should be collected and one switch case generated for each variable.

The “+” operator is distinguished at run-time as a string concatenation operator or an arithmetic operator; depends on the type of one or both operands. If the “+” operator is an arithmetical + then it is replaced with other arithmetical operators. Otherwise, no arithmetical operator replacement should be applied for this operator. This requires a type check at meta-mutant run-time. Consider, for example, the code given in the following:

```
function MutateAdd(x, y) {
  var result = x + y;
  if (result !== NaN
      && typeof(result) == "number") {
    switch (opimutantOperator){  // mutate as arithmetic
      case “+”:
        return x + y;
      case “-”:
        return x - y;
       . . .   // other operators here
  }
  else {
    return result;   // no arithmetic mutation
  }
 }
```

Checking for the type of the result of `x + y` is a convenient way of testing if any of the automatic conversions applied to `x` and `y` fail to produce a number.

Chapter 6: Empirical Investigation

6.1 Introduction

This chapter is intended to evaluate the new mutation analysis methods for dynamically typed programs. The evaluation is done using some example programs. This raises the question of how a mutation analysis method should be evaluated. The purpose of mutation analysis is to force the tester to write a test set that rigorously tests the program. Therefore, a good mutation analysis method forces the tester to write tests that test the program rigorously. At the same time, the mutation analysis should not be more costly than necessary. This means that the mutation analysis methods should be evaluated in terms of the quality of the test sets that are mutation adequate, i.e. test sets that can kill all the non-equivalent mutants and the cost of the mutation analysis that produced that mutation adequate test set.

6.2 Evaluating Cost of Mutation Analysis

The cost of mutation analysis consists of the cost of (a) generating a set of tests, (b) executing the tests on the mutants (c) investigation of the live mutants to check which are equivalent and which can be killed with additional tests. The cost of generating a set of tests depends on the method used. It can be done manually or by automatic test data generation. In this thesis, the tests were partly automatic and partly manual. It is the manual part that is expensive. In this thesis, the cost of tests generation was considered to be proportional to the number of tests generated.

The cost of executing the test set on the mutants is the total number of mutant executions. For an actual program with an actual test set, this can be counted. In general, however, the first test is executed on every mutant. Some of the mutants will be killed by the first test but the second test will be executed on the live mutants and the third test will be executed on the remaining live mutants and so on. In general, suppose that the average proportion of mutants that are still live after the execution of one test is `s, 0 <= s <= 1`. This means that the total number of executions of M mutants for the test set `T` is

$$M + sM + s^2M + . . . + s^TM$$

The first `M` is from the execution of the first test, the `sM` is from the second test and so on. In practice the proportion of mutants that are still live after the execution of one test is not constant and is typically between 20% and 50% for the first test and increases as more tests are executed. As more tests are executed, it is more difficult to kill mutants so s increases. About 5% to 10% of the mutants that are executed are not killed by any test because they are equivalent. When only equivalent mutants are left, the proportion of mutants that are still live after the execution of one test, `s,` is equal to `1`. It is possible that a test kills no mutants, especially if it is executed near the end of the sequence of tests. This is because, even though it can kill some mutants, these mutants have been killed by previously executed tests.

The cost of the investigation of the live mutants is difficult to measure in general. The live mutants are investigated either to manually generate a new test or to check that the mutant is equivalent. This means that the cost of the investigation of the live mutants should count only the investigation of the equivalent mutants because the investigation of the live mutants that generates a new test is already counted in the cost of the manually generated tests. In this thesis, the cost of equivalent mutants is considered as proportional to the number of equivalent mutants.

To summarise, the cost of producing a mutation adequate test set `T` for a program is based on

```
total number of tests
total number of mutant executions
number of equivalent mutants
```

Using these three numbers with some weighting, it is possible to compare the costs of different mutation analysis methods used to produce a mutation adequate test set for the same program.

```
Cost(T) = a * sizeOf(T) + b * MutantExecutions + c * EquivalentMutants
```

The most important cost is the number of equivalent mutants because to identify equivalent mutants is a manual activity. The less important cost is the number of mutant executions because this cost is machine time and not manual. It is not proposed to give specific values for the weightings except that for a typical program we expect:

```
c * EquivalentMutants > a * sizeOf(T) > b * MutantExecutions
```

The weightings would have to be different if a program had no equivalent mutants but this would be very unusual.

6.3 Minimal Test Sets

For a given set of mutants, there are usually many possible different mutation adequate test sets. For example, if any test is added to a mutation adequate test set then the new test set is also mutation adequate. When comparing the cost of producing a mutation adequate test set for a program, it is the lowest possible cost that is required. It is desirable to reduce cost of testing by not generating more tests than necessary. Also, there is a cost for the use and management of a test set that is proportional to the size of the test set. If the program function is modified, for example, then all the tests will need to be checked in case they also need to be modified. For this reason it is desirable to minimise the size of the test set.

In mutation analysis, a test set is non-redundant if no test can be removed without lowering the number of mutants killed. A minimal size mutation adequate test set is the smallest possible test set that is mutation adequate. A minimal test set (Akers et al. 1987) is difficult to generate in general. Even the problem of finding the smallest mutation adequate subset of a given set of tests is difficult. Given a set of tests, a minimal size mutation adequate subset is the smallest possible test subset that is mutation adequate. If a test set is minimal then it is also non-redundant but a test set may be non-redundant and still not be minimal. For example, consider the test set {t1, t2, t3} and the set of mutants {m1, m2, m3}. If t1 kills {m1, m2} and t2 kills {m2, m3} and t3 kills {m1, m2, m3} then

{t1, t2}	non-redundant set of mutation adequate tests
{t3}	minimal set of mutation adequate tests

In general, an algorithm for producing a minimal test set from a given test set has complexity O(2^n) because it is necessary to check every subset of tests of the given test set. In practice, a heuristic algorithm is usually used that is much faster and usually produces near minimal test sets. In this thesis the following algorithm was used.

1. Select tests randomly from the test set and place in a list, i.e. order the tests in a random order.
2. Execute each test in order on the mutants and remove any test that is redundant with respect to the previously executed tests, i.e. it kills no new mutants. For example, consider the test set {t1, t2, t3} above. t1 kills {m1, m2} and t2 kills {m2, m3} and so t2 is not redundant with respect to t1. Then t3 is executed and it kills {m1, m2, m3} but these mutants have already been killed by t1 and t2 and so t3 is redundant with respect to t1 and t2 so t3 is removed.
3. Reverse the order of the remaining tests and repeat the execution and removal of tests as in step 2.
4. Reorder the remaining tests so that the tests in the middle of the list are at the front and the tests at either end are in the middle. For example, <t1, t2, t3, t4, t5, t6> is reordered as <t3, t4, t2, t5, t1, t6>.

The test sets produced by this algorithm are called non-redundant test sequences. The idea behind the heuristic is that different tests are executed first. The earlier a test is executed the more likely it is to end up in the final list. Notice, however, that the above heuristic can fail to find a minimal subset even when the set contains just three tests. For example, consider the test set `{t1, t2, t3}` when `t1` kills `{m1, m2}`, `t2` kills `{m2, m3}` and `t3` kills `{m3, m4}`. The minimal subset is `{t1, t3}` but if the heuristic algorithm executes the three tests in the sequences below

```
<t1, t2, t3>    // first random order
<t3, t2, t1>    // reverse of first order
<t2, t1, t3>    // middle of second order at front
```

no test is removed. To detect that `t2` is redundant with respect to `t1` and `t3` it is necessary to execute `t2` last in the sequence.

6.4 Tescripta Mutation Analysis Tool

Tescripta is a test data generation and mutation analysis tool for JavaScript. Tescripta was initially developed by Dr L Bottaci [Bottaci 2010] and has been adapted as a result of the work done in this thesis to implement static and dynamic mutation analysis. Researchers may obtain Tescripta by contacting Dr Len Bottaci. The tool has documentation in the READ.ME file in the Tescripta solution that explains how to install and use Tescripta using Microsoft Visual Studio. The sample programs in this thesis that are available in the sample folder can be executed using Microsoft Visual Studio. Researchers can add new sample programs by copying and modifying a suitable existing sample program.

Tescripta performs mutation analysis using up to three different methods, labelled `notype` (static), `context` (static) and `runtype` (dynamic). The `mutationMethods` directive specifies one or more methods. Tescripta performs mutation analysis when the `mutationAnalysis` directive is `true`. The programmer should provide some information about the input domain of the program under test to produce correct test cases. In addition, the programmer may also add more tests manually to the generated tests. These tests are collected in a file to test the program. Moreover, input data generation always precedes mutation analysis. In this way, the tests generated can be used for mutation analysis. By default, the tests used to kill mutants are taken from the

`"*.bc.n.tst"` files where `n` is the trial number. All this is documented in the Tescripta Visual Studio solution.

The output of mutation analysis is a file listing the status, live, dead, etc. of each mutant and a file of effective tests, i.e. those tests that killed at least one mutant not killed by any preceding test. The mutant coverage information is printed to the file `programName.mut`, where `programName` is the name of the sample program file and the function under test. The name of the mutated object is printed together with the mutation table for that object. The mutation table lists the possible mutants, each identified by an integer. The table also lists the original object; the integer identifier of the original object is not followed by any character. A sample output that is produced for the min program using the dynamic mutation method is shown in Appendix C.

6.5 Evaluating Quality of Mutation Analysis

The aim is to compare the static mutation analysis method with the dynamic mutation method. In general, consider two mutation methods, A and B for the same program. Assume MA is the mutants generated by the method A and MB the mutants generated by the method B. Let TA be a mutation adequate test set for MA that is also a non-redundant sequence and let TB be a mutation adequate test set for MB that is also a non-redundant sequence. TA is mutation adequate which is shown as Kills (TA, MA) which means that the tests in TA kill all the mutants MA. Let Cost (TA) be the total cost of generating TA and Cost(TB) be the total cost of generating TB.

When comparing A and B, one possibility is that TA costs less than TB but TA is better at killing mutants than TB. This could be written as the condition

$$\text{Cost}(TA) < \text{Cost}(TB) \text{ and } \text{Kills}(TA, MB) \text{ and not } \text{Kills}(TB, MA)$$

TA can be considered to be better than TB because TA kills all the mutants of A and B but TB cannot kill all the mutants of A. This obviously assumes that killing mutants is a good way to evaluate the quality of a test set. More generally, if the above condition applied to many test sets and many programs then it would be strong evidence that A is a better mutation analysis method than B. The condition that both TA and TB should be non-redundant test sequences reduces the risk that TA can kill more mutants than TB because TA is larger than TB. TA should be large enough to kill all the mutants of MA and no larger. Similarly for TB.

Another situation in which TA can be considered to be a better test set than TB is if TA is better at killing mutants but TA costs no more than TB. This means replacing the $<$ in the above condition with a $<=$.

Another situation in which TA can be considered to be a better test set than TB is that as above TA costs less than TB but TA has the same mutant killing ability as TB. This is expressed as

Cost(TA) $<$ Cost(TB) and Kills(TA, MB) and Kills(TB, MA) or
Cost(TA) $<$ Cost(TB) and not Kills(TA, MB) and not Kills(TB, MA)

The comparison of mutation adequate test sets will vary according to the particular tests in the test sets. To check if a condition holds in general, a number of different mutation adequate test sets should be produced for the mutation analysis of a particular program and the average taken across all the test sets. More generally, this should be repeated for a range of programs.

To illustrate the cost and quality measurements described above, consider the following JavaScript program. This program is similar to the example program in the last two chapters.

```
function orig(n) {  // n is integer
  var x = 0, i = 0, y = "", r = 0;
  for (i = 0; i < n; i++) {
    if (i == 1) {
      x = true;
      y = "23";
    }
    else if (i == 2) {
      x = y - 1;     // string y converted to number
    }
    r += x + 3;      // x is number, string and Boolean
  }
  return r;
}
```

The program has no purpose but inputs an integer which determines the number of times a value is added to a result. Mutation analysis was performed on this program using a mutation analysis test tool called Tescripta. Using Tescripta, 30 random tests were generated. These tests are different values of n from [-9, 30]. Mutants were generated using both the static and dynamic methods.

For the static method, the mutation analysis was done with and without type context information to give three methods in all. In the tables below, to save space, the static method without type context information is referred to as 'static' and the static method with type context information is referred to as 'context'.

method	total mutants	%	non-equiv	%	equiv	%	% equiv
static	268	100	228	100	40	100	14.9
context	250	93.2	216	94.7	34	85.0	13.6
dynamic	159	59.3	147	64.5	12	30.0	7.5

For each of the three mutation methods, static with no type context, static with type context, and dynamic, 10 random mutation adequate non-redundant test sequences were generated. When generating test sets, the test set minimisation heuristic described earlier was used. One of the 10 final sets produced was the 3 following tests:

```
n  5  output 82
n  0  output  0
n -1  output  0
```

To produce this set, 27 tests were removed from the 30 random tests to produce the final non-redundant sequence of 3 tests. The total tests and the mutant executions are given in the table below:

method	total tests	mutant executions
static	3	310
context	3	298
dynamic	3	152

From the number of executions it is clear that the cost of the dynamic method is the lowest, the static method without type context is the most expensive and the cost of the static method with type context information is more than the dynamic method but not as high as the static method without context information. Notice that the cost of the dynamic method is about a half of the cost of the other two methods. Notice also that the lower cost of the dynamic method is not all due to the lower number of mutants. If the three methods are compared to each other with the static no type context as 100% then the static-context method produces 94.7% of the mutants and has about the same proportion of executions at 96.1%. The dynamic method produces 64.5% of the mutants but has a lower proportion of executions at 49.0%.

method	non-equiv	% of static	executions	% of static
static	228	100	310	100
context	216	94.7	298	96.1
dynamic	147	64.5	152	49.0

The following table shows how effective are the mutation adequate test sets of one method at killing the mutants produced by other methods.

	kill tests mutation adequate on method		
generate method	static	context	dynamic
static	100	100	100
context	100	100	100
dynamic	100	100	100

The table shows that mutants generated by any of the three methods can all be killed by a mutation adequate test set that is generated by any other method. For this particular program

cost(context) < cost(static) and Kills(context, static) and Kills(static, context)

In this case, static-context is a better method than static because it has a lower cost but the mutation adequate test sets are just as effective. Similarly, for this particular program

Cost(dynamic) < cost(context) and
Kills(dynamic,context) and Kills(context, dynamic)

In this case, dynamic is a better method than static-context because it has a lower cost but the mutation adequate test sets are just as effective.

6.5.1 Competence of Mutants

It is the mutants that are difficult to kill that force the tester to enhance the test set. A good mutation method should produce mutants that are difficult to kill. For a given test set and order of test, the competence of a mutant is defined as the proportion of the tests executed before it is killed when the tests are executed in order. For example, if there are 10 tests and a mutant is killed by the third test then the competence of the mutant is 20% because two out of 10 tests were executed before it was killed. The competence depends on the order that the tests are executed. This is not desirable and is similar to the problem that there are many different possible mutation adequate test sets for a given set of mutants. For the same reason, it would be very costly to calculate the competence for every possible ordering of a test set.

Chapter 7: Evaluation and Discussion

7.1 Introduction

An empirical investigation is required to answer the research questions and check the effectiveness of static and dynamic approaches for the mutation analysis of dynamically typed programs introduced in this thesis. Seven JavaScript programs have been used in the evaluation. These programs were selected for performing experiments to answer the following research questions:

Q1: Using the static mutation approach, what proportion of the mutants are type-mutants? This is the same question as how many mutants are not generated in the dynamic method.

Q2: Using the static mutation approach, what proportion of the type-mutants is incompetent? Are type-mutants more incompetent than mutants where the different values in the original and mutant programs have the same type?

Q3: What is the cost reduction by not generating type-mutants? In other words, how does the cost of the static method compare with the dynamic method?

Q4: How does the choice of static or dynamic mutation affect the number of equivalent mutants generated?

Q5: Which is the most cost effective method?

Typically, the results of these experiments can be used to demonstrate that the new approaches for the mutation analysis of dynamically typed programs can generate mutants in the manner of traditional mutation analysis. A tool called Tescripta, which was designed initially for automatic test data generation and has been extended for the mutation analysis of JavaScript programs, was used to perform the mutation analyses for the selected programs.

7.1.1 Experiments and Evaluation of Results

Seven JavaScript programs were written. The programs were written because it is difficult to find small JavaScript programs that do not execute inside an html web page. The Tescripta tool can execute only stand-alone JavaScript programs. The programs written are considered to be typical of JavaScript programs apart from, instead of input being taken from a form on a web page, the input is provided programmatically by the Tescripta tool. Also instead of writing output to a web page, the output is collected programmatically by the Tescripta tool.

The seven programs all were written to use variables that hold values of different types. Unless a variable holds values of different types then there is no difference in the static and dynamic approaches to mutation analysis and such a program would not be useful for comparing the static and dynamic approaches.

Although some dynamically typed languages, JavaScript for example, allow an operand to hold different types of values, it is noticed by checking a large number of JavaScript programs that this practice is not used very much by JavaScript programmers and did not often happen that an operand was used to hold different types of values. In general, the type of operands in a program depends mainly on the type of inputs supplied during the execution and mixed types in a variable occur mainly in the input to a program, e.g. number or string. The seven sample programs have been written in this style.

The seven programs used in the empirical investigation are listed in Appendix B and the description of each program and the numbers of lines of code (LOC) are summarized in Table7.1.

Program	LOC	Description
min	45	Depending on the types of the inputs, returns the minimum of two numbers or the length of the shortest of two strings or a number if it is less than the length of the second string input or a string if its length is less than the second number input.
boolStringNumber	17	Sets variable x to number, string and Boolean type and adds the value of each x to the result.
wages	39	Calculate wages given salary information and number of hours worked on each of 7 days adding on overtime.
price	62	Input is an array of orders. Each order is object representing an Html input form containing the item ordered, quantity and discount. Result is the cost of the total orders. The inputs may be numbers or strings.
hazard	60	The old game of hazard, played with a pair of dice. The first input declares main followed by a sequence of throws of the dice. Outcomes of the game depending on the value of main, and then the numbers on the pair of dice.
game	131	Board game with a sequence of squares. Player starts at square zero and array of inputs determines moves. When player lands on a square, a unit of energy is consumed and depending on the square various actions take place which might move the player to another square, cause the player to gain or loose energy or kill the player and end the game.
student	35	Input is a set of module marks for a set of students. If input module matches module in marksheet, mark is added otherwise mark for new module.

Table 7.1: Description of programs used in the empirical evaluation

Each of the seven programs was submitted to Tescripta so that Tescripta could generate the mutants of the program. In fact, for each method, static, context and dynamic, Tescripta generates a single program that implements all the mutants of the method. The program is known as a meta-mutant.

To generate the test data to kill the mutants, for each of the seven programs, Tescripta was used to generate branch coverage test data. For each program, 10 random branch coverage test sets were generated. All these 10 test sets were placed in a single test file, one file for each program. This file was used as the test set to kill mutants. If needed, tests were written by hand to kill any live non-equivalent mutants and added to the test set for the program. The equivalent mutants for each program and method were identified.

Table 7.2 shows the number of mutants of each program for each method. For a program and a method, the total of mutants is shown as the average of the killed non-equivalent mutants and the equivalent mutants.

Program	Number of Mutants for each method								
	Static			Context			Dynamic		
	killed	Eqv	Total	killed	Eqv	Total	killed	Eqv	Total
`min`	408	107	515	387	100	487	308	82	390
`boolStringNumber`	236	43	279	224	37	261	174	22	196
`wages`	1482	120	1602	1388	102	1490	1123	56	1179
`price`	1587	214	1801	1437	209	1646	944	101	1053
`hazard`	1031	117	1148	982	117	1099	523	54	577
`game`	3148	462	3610	2930	436	3366	2064	279	2343
`student`	988	93	1081	950	93	1043	456	68	524
Average	888	116	1004	830	109	939	559	66	626

Table 7.2: Sets of mutants generated for the programs used in the empirical evaluation

These sets of mutants given in Table 7.2 are useful to help answer the research questions above.

Q1: Using the static mutation approach, what proportion of the mutants are type-mutants? This is the same question as how many mutants are not generated in the dynamic method.

It can be seen from the results that on average across the seven programs the static method generates the most mutants, the context method generates about 7% fewer mutants and the dynamic method generates about 38% fewer than the static method. Notice that these percentages are roughly constant for all of the seven programs so this is evidence that the percentages generalise across JavaScript programs in general.

Consider the research question

Q2: Using the static mutation approach, what proportion of the type-mutants is incompetent? Are type-mutants more incompetent than mutants where the different values in the original and mutant programs have the same type?

For each element, the mutants that are present in the static method and not present in the dynamic method i.e., type-mutants should be identified and checked. It will take too long to do all of them. The lack of time did not allow a complete analysis of all the mutants that are type-mutants. These have to be inspected by hand and there are many thousands of mutants. For this reason, a sample of mutants was examined for the `min` program. From this sample 100 % of the type- mutants were found to be killed by the first or second test. This is good evidence that type-mutants are incompetent.

The number of tests in a non-redundant test sequence that kills all the non-equivalent mutants for each of the seven programs is shown in Table 7.3. The total number of mutant executions is also shown. For some mutants that are difficult to kill, tests are executed many times. Using Table 7.3, the costs of the three methods can be compared. The data can be used to answer the research questions

Q3: What is the cost reduction by not generating type-mutants? In other words, how does the cost of the static method compare with the dynamic method?

Q4: How does the choice of static or dynamic mutation affect the number of equivalent mutants generated?

Program	Number of adequate tests / method			Number of mutants execution / method			Number of equivalent mutants /method		
	Stc	Ctx	Dyc	Stc	Ctx	Dyc	Stc	Ctx	Dyc
min	20	20	20	1643	1577	1749	107	100	82
boolStringNumber	3	3	4	379	357	310	43	37	22
wages	10	10	10	3550	3280	2538	120	102	56
price	14	13	19	4366	3996	4691	214	209	109
hazard	66	66	66	20587	20399	14197	117	117	54
game	26	26	26	34643	33799	26620	462	438	279
student	6	6	8	1640	1549	1562	93	93	68
Total	145	144	153	66808	64957	51667	1156	1096	670

Table 7.3: Number of mutation adequate tests, number of mutant executions and number of equivalent mutants for each method and for each programs used in the empirical evaluation.

For the `min` program, the dynamic method is considered to have the lowest cost. It was explained in Chapter 6 that the most expensive part of the mutation analysis is the equivalent mutants and that

```
c * EquivalentMutants > b * MutantExecutions
```

The number of tests is the same for all methods so this leaves only the number of equivalent mutants and the number of executions. For the `min` program the number of equivalent mutants for the dynamic method is the smallest.

The table shows that the situation of the `min` program is similar to the other six programs. Notice that for the `boolStringNumber` program in the dynamic method, there is one more test than in the static and context methods. This is because there is a mutant that is killed in the static and context methods; instead the mutant is not defined in the dynamic method because it is type-incorrect.

In general for the seven programs, the number of tests for each method varies very little. The main difference between the methods is that the dynamic method has fewer mutants and fewer equivalent mutants than the other methods. This shows that the dynamic method is the least costly method.

There is the question of the effectiveness of the mutation method. Is the mutation adequate test set of one method able to kill the mutants of another method? In other words, are the mutants generated by a method difficult enough to kill mutants generated by another method. Table 7.4 shows the percentages of mutants killed by a test set. The column shows the method used to produce the mutation adequate test set. The row shows the method used to produce the mutants that are killed by the test set. For each program, the percentages are averaged across the 10 trial and are rounded to the nearest 1%.

Mutant generate method	Mutation adequate test method		
	Static	Context	Dynamic
Static		100	100
Context	100		100
Dynamic	100	100	

Table 7.4: The percentages of mutants killed by a test set. The column shows the method used to produce the mutation adequate test set. The row shows the method used to produce the mutants that are killed by the test set. For each program, the percentages are averaged across the 10 trial and are rounded to the nearest 1%

Consider the research question

Q5: Which is the most cost effective method?

Because the tests of each method can kill the mutants of the other methods, all methods are equally effective. This means that a method must have a lower cost to be cost effective. i.e. method TA is more cost effective than method TB when

$\text{Cost}(TA) < \text{Cost}(TB)$ and $\text{Kills}(TA, MB)$ and $\text{Kills}(TB, MA)$

From Table 7.3, it is clear that the dynamic method is the most cost effective and the context method is the second most cost effective. This is true for all the seven programs of the evaluation and so this suggests that the result can generalise to programs in general.

Chapter 8: Conclusions

8.1 Contributions

There are three main contributions of the thesis.

1. Methods have been developed for the mutation analysis of dynamic programs. Previously, mutation analysis has been applied only to strongly typed programs.
2. The developed methods of mutation analysis have been evaluated empirically using a set of sample programs.
3. There is evidence that the dynamic mutation analysis method is the most cost effective method for dynamic programs.

The thesis investigated and introduced two new approaches for the mutation analysis of dynamically typed programs which have not been done before. Firstly, a static approach to generate mutants of dynamically typed programs has been developed. This approach can be divided into two sub-approaches, static with no context information and static with context information. This approach does not use run-time type information but does use some static information. The type context in which a program element occurs can be used to avoid some type-mutations and to make heuristic assumptions about the type of the element. This approach is based on making only syntactic changes to the program under test. Secondly, a dynamic approach to generate type-correct mutants at run-time. This thesis argued that mutation analysis of dynamically typed programs can also be established at run-time by executing test cases against the program under test. The aim is to discover type information at program run-time to avoid type-mutations and, therefore, some of the incompetent mutants.

Mutation analysis has been shown to be an effective test coverage criterion for statically typed programs. Mutation analysis has not yet been fully applied to dynamically typed programs. Since testing is important for programs written in dynamically typed languages and mutation analysis is a demanding testing criterion, the combination of mutation analysis for dynamically typed programs has the potential to be highly effective.

As discussed in Chapter 4, this thesis considered a small set of mutation operators. Most of these mutation operators have been used in the mutation analysis of statically typed programs but the string and object mutation operators are new. The simple static approach for the mutation of dynamically typed programs is similar to the traditional mutation analysis except that the mutation is done with the assumption that every variable can hold values of all data types. This obviously will result in producing type-mutants. An alternative to this is to use the type context in which the mutated element occurs to eliminate some type-mutations. The static approach checks the type context of variables in an expression and heuristically assumes types of these variables. In this approach, mutants are generated by applying a compatible set of mutation operators in a static manner. It is, however, possible that a variable hold a value of any-type context. In this case, all non-redundant mutations are used to generate mutants.

The dynamic approach is investigated and introduced to generate mutants at run-time by applying the same set of mutation operators that has been introduced for static mutation, but in a typed manner. This is, however, required that test cases are executed against the program under test to discover the type information of operands. Since mutants are generated at run-time in a type-sensitive manner, this approach reduces the number of mutants generated and produces more difficult to kill mutants.

The dynamic approach has been effective by improving the performance and reducing the cost of mutation analysis for dynamically typed programs. There is, however, a risk that a type may not be discovered because different inputs may result in different types. If a type is not discovered then no mutation of that type are performed. Dynamic type discovery allow the application of mutation operators to be more efficient. The dynamic mutation method forces the tester to write tests that set a value of a variable to be of a specific type as well as a specific value.

The static and dynamic approaches are evaluated and compared for the mutants produced for seven sample programs. The dynamic approach has been shown to be the more cost effective than the static approach. Although, in the dynamic mutation approach, time is consumed to determine the type of operands and variables, dynamic mutation has been shown to be efficient when compared to static mutation. As shown earlier in Chapter 7, the results obtained by using Tescripta to perform the mutation analysis for a number of JavaScript programs for the three different mutation methods have shown that the dynamic approach provides an effective way to test dynamically typed programs. It is clear that the cost of the dynamic method is the lowest, the static method without type context is the most expensive and the cost of the static method with type context information is more than the dynamic method but not as high as the static method without context information. Notice that the cost of the dynamic method is about a half of the cost of the other two methods.

Finally, a part of the type-mutants that are present in the static method and not present in the dynamic method are identified and inspected. The lack of time did not allow a complete analysis of all these mutants because they have to be inspected by hand and there are many thousands of mutants. A sample of mutants were inspected and found that it was found that100% of these mutants are easily killed, i.e., incompetent mutants.

8.2 Mapping the Thesis Contribution to Research Questions

The main contribution of this thesis is the development of two new approaches to the mutation analysis of dynamically typed programs. These two mutation analysis approaches, static and dynamic, are also useful to answer the research questions listed in Chapter 1. These questions are concerned with evaluating the advantages and disadvantages of the two approaches to the mutation analysis of dynamically typed programs. In order to compare the approaches, both approaches were used on a sample of example programs. Mutants were generated for each of the seven sample programs using both the static and dynamic methods.

The first research question, Q1, is concerned with type-mutants. The advantage of the dynamic mutation method is that it does not generate type-mutations. The benefit of this advantage depends on how many type-mutants are generated by the static method. If the static method generates only a small number of type-mutants then the mutants generated by the static and dynamic methods will be very similar and it will not be possible to have a significant advantage for dynamic mutation.

Q1: Using the static mutation approach, what proportion of the mutants are type-mutants? This is the same question as how many mutants are not generated in the dynamic method.

The question was answered by calculating the difference between the numbers of mutants generated using the static and dynamic methods. The difference was found to be about 38% fewer using the dynamic method.

The second research question, Q2, is concerned with what proportion of type-mutants is incompetent.

Q2: Using the static mutation approach, what proportion of the type-mutants is incompetent? Are type-mutants more incompetent than mutants where the different values in the original and mutant programs have the same type?

It is reasonable to assume that type-mutations will make a large difference to the behaviour of the program and so it is reasonable to assume that a type-mutant can be easily killed, i.e. incompetent. Incompetent mutants are not useful. Although it is reasonable to assume that type-mutations will make a large difference to the behaviour of the program this assumption must be tested. This assumption was tested by counting how many of the type-mutants are easily killed, i.e. killed by the first or second test. The type-mutants are identified as the mutants present in the static method but not present in the dynamic method. The empirical evaluation of the sample programs shows that all the type-mutants are incompetent. This does not mean that for all programs, every type-mutant is incmpetent but there is a high probability that it is incompetent.

The third and the fourth research questions, Q3 and Q4, are concerned with the reduction of the cost of mutation analysis by not generating type-mutants and equivalent mutants.

Q3: What is the cost reduction by not generating type-mutants? In other words, how does the cost of the static method compare with the dynamic method?

This question was answered by comparing the cost of the mutation analysis using the static method with the cost of mutation analysis using the dynamic method. The experiments with the seven example programs show that the cost of the mutation analysis using the dynamic method is lower than the cost of mutation analysis using the static method.   On average, the cost was found to be about 40% lower

Q4: How does the choice of static or dynamic mutation affect the number of equivalent mutants generated?

Comparing the number of equivalent mutants generated in the static method with the number of equivalent mutants generated in the dynamic method, for the seven example programs, less equivalent mutants were generated in the dynamic method. The equivalent mutants generated using the dynamic method is about 45% fewer than using the static method. The main reduction in cost is due to the lower number of equivalent mutants in the dynamic method. The cost of the equivalent mutants is the dominant cost of mutation analysis.

The fifth research question, Q5, is concerned with the most effective method.

Q5: Which is the most cost effective method?

This question was answered by generating mutants using the tests of each method to kill the mutants of the other method. The results summarized in Table 7.4 shown that all methods are effective. A mutation method of dynamically typed programs that avoids producing type-mutants and reduces the number of equivalent mutants without affecting the effectiveness of mutation analysis is considered as a cost effective method. Therefore, comparing static with dynamic mutation, the dynamic is the most cost effective method

8.3 Limitation and Future Work

The empirical evaluation has been done on only seven programs and therefore more study with a bigger range of programs is needed to fully establish the results of this thesis.

It is not clear how many actual JavaScript programs use variables that hold values of different types. The lower cost of the dynamic method may not be so important if most JavaScript programs use variables like statically typed programs. It could be in real programs that most of the mixed types are number and string and apply mostly to input variables. More research is needed to analyse existing JavaScript programs.

The investigation of the three approaches is based on the JavaScript programming language. This language is widely used in web applications and combines the most common features of other high-level dynamically typed languages. There are other dynamic languages and further work involves applying these new approaches to the mutation analysis of other high-level dynamically typed languages.

Bibliography

Acree, A. T., Budd, T. A., Demillo, R. A., Lipton, R. J. and Sayward, F. G. (1979). *Mutation Analysis*, Atlanta, Georgia: Georgia Institute of Technology.

Adamopoulos, K., Harman, M. and Hierons, R. M. (2004). 'How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution', *Genetic and Evolutionary Computation Gecco 2004 , Pt 2, Proceedings,* 3103, 1338-1349.

Agrawal, H., Demillo, R. A., Hathaway, B., Hsu, W., Krauser, E. W., Martin, R. J., Mathur, A. P. and Spafford, E. (1989). *Design of Mutant Operators for the C Programming Language*, West Lafayette, Indiana: Purdue University.

Akers S.B., Joseph C. and Krishnamurthy B. (1987). 'On the role of independent fault sets in the generation of minimal test sets', *Proc. 1987 Int. Test. Conf.*, pp. 1100-1107.

Alexander, R. T., Bieman, J. M., Ghosh, S., Ji, B. X. and Ieee Computer, S. (2002). 'Mutation of Java objects', *13th International Symposium on Software Reliability Engineering, Proceedings*, 341-351.

Amman, P. and Offutt, J. (2008). *Introduction to software Testing,* Cambridge University Press.

Arcuri, A. and Yao, X. (2007). 'On test data generation of object-oriented software', *TAIC PART 2007 - Testing: Academic and Industrial Conference - Practice and Research Techniques, Proceedings: CO-LOCATED WITH MUTATION 2007*, 72-76.

Artzi, S., Dolby, J., Jensen, S. H., Moller, A., Tip, F. and Ieee (2011). 'A Framework for Automated Testing of JavaScript Web Applications', *2011 33rd International Conference on Software Engineering (Icse)*, 571-580.

Barbey, S. and Strohmeier, A. (1994). 'THE PROBLEMATICS OF TESTING OBJECT-ORIENTED SOFTWARE', *Software Quality Management Ii, Vol 2: Building Quality into Software*, 411-426.

Barbosa, E. F., Maldonado, J. C. and Vincenzi, A. M. R. (2001). 'Toward the determination of sufficient mutant operators for C', *Software Testing Verification & Reliability*, 11(2), 113-136.

Beizer, B. (1990). *Software Testing Techniques,* second ed., London: Van Nostrand Reinhold Company Limited.

Bertolino, A. (2003). 'Software testing research and practice', *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice(ASM'03)*, Taormina, Italy, Springer-Verlag, PP 1-21.

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools,* Addison-Wesley Professional.

Chevalley, P. (2001). 'Applying mutation analysis for object-oriented programs using a reflective approach', *Apsec 2001: Eighth Asia-Pacific Software Engineering Conference, Proceedings*, 267-270.

Choi, B. J. and Mathur, A. P. (1993). 'High-Performance Mutation Testing ', *Journal of Systems and Software*, 20(2), 135-152.

Crockford, D. (2008). *JavaScript: The Good Parts,* O'Reilly Media Inc.

Delamaro, M. E. and Maldonado, J. C. (1996). *Proteum- A Tool For the Assessment of Test Adequacy for C Programs,* translated by New Brunswick, New Jersey: pp. 79-95.

Demillo, R. A., Guindi, K. N., King, K. N., McCracken, W. M. and Offutt, A. J. (1988). 'An Extended Overview of the Mothra Software testing environment', in *Proceedings of SIGSOFT Symposium on Software Testing, Analysis and Verification 2,* (July), 142–151.

Demillo, R. A., Krauser, E. W. and Mathur, A. P. (1991). 'Compiler-Integrated Program Mutation', *Compsac 91 - the Fifteenth Annual International Computer Software & Applications Conference, Proceedings*, 351-356.

Demillo, R. A. and Lipton, R. J. (1978). 'Hints On Test Data Selection - Help For Practicing Programmer', *Computer,* 11(4), 34-41.

Duggan, D. and Bent, F. (1996). 'Explaining type inference', *Science of Computer Programming,* 27(1), 37-83.

Flanagan, D. (2011). *JavaScripts :The Definitive Guide*. O'Reilly Media, Inc.

Gligoric, M., Badame, S. and Johnson, R. (2011). 'SMutant: a tool for type-sensitive mutation testing in a dynamic language', in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, Szeged, Hungary, ACM, pp 424-427.

Goodman, D. and Morrison, M. (2007). *Javascript bible,* sixth ed., New York, NY, USA: John Wiley & Sons, Inc.

Gupta, R., Harrold, M. J. and Soffa, M. L. (1996). 'Program Slicing-Based Regression Testing Techniques', *Software Testing, Validation, and Reliability,* 6(2), 83-111.

Hamlet, R. G. (1977). 'Testing programs with the aid of a compiler', *Ieee Transactions on Software Engineering,* 3(4), 279-290.

Harman, M., Hierons, R. and Danicic, S. (2001). 'The relationship between program dependence and mutation analysis', *Mutation Testing for the New Century,* 24, 5-13.

Hierons, R. M. (2002). 'Comparing test sets and criteria in the presence of test hypotheses and fault domains', *Acm Transactions on Software Engineering and Methodology,* 11(4), 427-448.

Howden, W. E. (1982). 'Completeness Criteria for Testing Elementary Program Functions - Week Mutation Testing and Completeness of Sets', *IEEE Transactions on Software Engineering,* 8(4), 371-379.

IEEE-Standards-Board, Ieee standard for software unit testing. *An American national standard, ansi/ieee std 1008-1987. IEEE Standards: Software Engineering*, Volume two: Process Standards, 1999.

Jackson, D. and Woodward, M. R. (2001). 'Parallel firm mutation of Java programs', *Mutation Testing for the New Century,* 24, 55-61.

Jia, Y. and Harman, M. (2009). 'Higher Order Mutation Testing', *Information and Software Technology,* 51(10), 1379-1393.

Jia, Y. and Harman, M. (2011). 'An Analysis and Survey of the Development of Mutation Testing', *IEEE Transactions on Software Engineering,* 37(5), 649-678.

King, K. N. and Offutt, A. J. (1991). 'A FORTRAN Language System for Mutation-Based Software Testing ', *Software-Practice & Experience,* 21(7), 685-718.

Langdon, W. B., Harman, M. and Jia, Y. (2010). 'Efficient multi-objective higher order mutation testing with genetic programming', *Journal of Systems and Software,* 83(12), 2416-2430.

Leroy, X., Doligez, D., Garrigue, J., R´emy, D. and Vouillon, J. (2002). *The Objective Caml system,documentation and user's manual (release 3.06)*, Tech.rep., INRIA, Rocquencourt, France.

Linda Dailey, P. (2007). 'Developers Shift to Dynamic Programming Languages', *Computer,* 40(2), 12-15.

Ma, Y. S., Offutt, J. and Kwon, Y. R. (2005). 'MuJava: an automated class mutation system', *Software Testing Verification & Reliability,* 15(2), 97-133.

Mathur, Aditya and p. (1991). 'Performance, Effectiveness, and Reliability Issues in software Testing', in *Proceedings of COMPSAC'91, CA: IEEE Press.* pp. 604-605.

Mikkonen, T. and Taivalsaari, A. (2007). *Using JavaScript as a RealProgramming Language, Sun Microsystems Laboratories Technical Report TR-2007-168*.

Mresa, E. S. and Bottaci, L. (1999) 'Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study', *Software Testing, Verification and Reliability,* 9(4), pp. 205-232.

Namin, A. S. and Anderws, L. H. (2006). *Finding sufficient Mutation operators via Variable Reduction,* North Catolina: IEEE Computer Society.

Offutt, A. J. (1992). 'Investigation of the Software Testing Coupling Effect', *ACM Transaction on Software Engineering and Methodology,* 1(1), 5-20.

Offutt, A. J. and King, K. N. (1987). 'A FORTRAN-77 Interpreter for Mutation Analysis', *Acm Sigplan Notices,* 22(7), 177-188.

Offutt, A. J. and Pan, J. (1997). 'Automatically Detecting Equivalent Mutants and Infeasible Paths', *Software Testing, Verifivation and Reliability,* 7(3), 165-192.

Offutt, A. J., Rothermel, L. G., Untch, R. H. and Zapf, C. (1996a). 'An Experimental Determination of Sufficient Mutant Operator', *ACM Transaction on Software Engineering and Methodology*, 21(7), 685-718.

Offutt, A. J., Voas, J. and J., P. (1996b). *Mutation operators for Ada*, George Mason University.

Offutt, A. J. (1992). 'Investigation of the Software Testing Coupling Effect', *ACM Transaction on Software Engineering and Methodology*, 1(1), 5-20.

Polo, M., Piattini, M. and Garcia-Rodriguez, I. (2009). 'Decreasing the cost of mutation testing with second-order mutants', *Software Testing Verification & Reliability*, 19(2), 111-131.

Richards, G., Lebresne, S., Burg, B. and Vitek, J. (2010). 'An Analysis of the Dynamic Behavior of JavaScript Programs', *Acm Sigplan Notices*, 45(6), 1-12.

Rapps, S., and Weyuker, E. J. (1982). 'Data flow analysis techniques for test data selection', In *Proceedings of the 6th international conference on Software engineering*, (pp. 272-278), IEEE Press.

Schuler, D., Dallmeier, V., Zeller, A. and Acm (2009). 'Efficient Mutation Testing by Checking Invariant Violations', *Issta 2009: International Symposium on Software Testing and Analysis*, 69-79.

Shahriar, H., Zulkernine, M. and Ieee (2009). 'MUTEC: Mutation-based Testing of Cross Site Scripting', *2009 Icse Workshop on Software Engineering for Secure Systems*, 47-53.

Shamaila H. (2008). 'Mutation Clustering', *Masters Thesis*, King’s College London, Strand, London.

Sofokleous, A. A. and Andreou, A. S. (2008). 'Automatic, evolutionary test data generation for dynamic software testing', *Journal of Systems and Software,* 81(11), 1883-1898.

Tratt Laurence (2009). 'Dynamically Typed Languages', *Advances in Computers*, Vol. 77, pp. 149-184

Untch, R. H. (1992). *Mutation-Based Software Testing Using Program Schemata,* translated by Raleigh, North Carolina: pp. 285-291.

Untch, R. H. (2009). 'On reduced neighborhood mutation analysis using a single utagenic operator', in *ACM Southeast Regional Conference, Clemson SC,* pp. 19–21.

Untch, R. H., Offutt, A. J. and Harrold, M. J. (1993). *Mutation Analysis Using Mutant Schemata,* translated by Cambridge, Massachusetts: pp. 139-148.

Wong, W. E. (1993). *On Mutation and Data Flow*, PhD thesis. Purdue University.

Wong W. E. and Mathur A. P (1995) . 'Reducing the cost of mutation testing: An empirical study', *JSS,* 31(3):185–196, 1995.

Zhu, H., Hall, P. A. V. and May, J. H. R. (1997). 'Software unit test coverage and adequacy', *Acm Computing Surveys,* 29(4), 366-427.

http://docstore.mik.ua/orelly/webprog/jscript/index.htm, revisited on 4/2/2013

Ivan Moore(2005). Pester. http://jester.sourceforge.net. revisited 5/2/2013

Heckle. http://seattlerb.rubyforge.org/heckle/ revisited 5/2/2013

http://jibbering.com/faq/notes/type-conversion/ Visited 15/12/20012

Appendix A: An Algorithm for checking the context of an operand or operator in a JavaScript program

The algorithm uses the abstract syntax tree of the function under test and traverses the tree, each child node first followed by the node itself.

```
Object PostOrder(Object node) {
      foreach (Object child in children) {
        PostOrder(child);
      }
      return Postprocess(node);
    }

Object Postprocess(Object node) {
      if (node is ConstantWrapper) {
        if (((ConstantWrapper)node).isNumericLiteral) {
          progElem.AddContext(((ConstantWrapper)node).nome, JSType.number);
        }
        else if (((ConstantWrapper)node).value is String
                   || ((ConstantWrapper)node).value is char) {
          progElem.AddContext(((ConstantWrapper)node).nome, JSType.strng);
        }
        else if (((ConstantWrapper)node).value is System.Boolean) {
          progElem.AddContext(((ConstantWrapper)node).nome, JSType.booln);
        }
      }
      else if (node is Lookup) {
        if (!(ParentStack(1) is VariableDeclaration)) {
          ;
        }
      }
      else if (node is Member) {
        Member memNode = (Member)node;
        progElem.AddContext(memNode.nome, JSType.member);
        progElem.AddContext(memNode.rootObject.nome, JSType.objct);
      }
      else if (node is Call) {
        Call callNode = (Call)node;
        if (callNode.inBrackets) {
          progElem.AddContext(callNode.func.nome, JSType.array);
        }
        else {
          progElem.AddContext(callNode.func.nome, JSType.functn);
        }
      }
      else if (node is If) {
        If ifNode = (If)node;
        if (ifNode.condition is Lookup
            || ifNode.condition is Member) {
          progElem.AddContext(ifNode.condition.nome, JSType.booln);
        }
        else if (ifNode.condition is Logical_and
                 || ifNode.condition is Logical_or
                 || ifNode.condition is Comma) {
          BinaryOp cond = (BinaryOp)ifNode.condition;
          progElem.AddContext(cond.operand2.nome, JSType.booln);
        }
      }
      else if (node is While) {
        While wNode = (While)node;
        if (wNode.condition is Lookup
            || wNode.condition is Member) {
          progElem.AddContext(wNode.condition.nome, JSType.booln);
        }
      }
      else if ((node is Logical_and)
               || (node is Logical_or)) {
        BinaryOp bop = (BinaryOp)node;
        if (bop.operand1 is Lookup
            || bop.operand1 is Member) {
          progElem.AddContext(bop.operand1.nome, JSType.booln);
        }
      }
      else if (node is NumericUnary) {
        NumericUnary uop = (NumericUnary)node;
        if (uop.operatorTok == JSToken.Minus) {
          progElem.AddContext(uop.nome, JSType.number);
          if (uop.operand is Lookup
            || uop.operand is Member
            || uop.operand is Plus) {
            progElem.AddContext(uop.operand.nome, JSType.number);
          }
        }
        else if (uop.operatorTok == JSToken.LogicalNot) {
          progElem.AddContext(uop.nome, JSType.booln);
          if (uop.operand is Lookup
            || uop.operand is Member) {
            progElem.AddContext(uop.operand.nome, JSType.booln);
          }
        }
      }
      else if (node is NumericBinary) {  // -, *, /, %
        NumericBinary nop = (NumericBinary)node;
        if (nop.operatorTok == JSToken.Minus
            || nop.operatorTok == JSToken.Multiply
            || nop.operatorTok == JSToken.Divide
            || nop.operatorTok == JSToken.Modulo) {
          progElem.AddContext(nop.nome, JSType.number);
          if (nop.operand1 is Lookup
              || nop.operand1 is Member
              || nop.operand1 is Plus) {
            progElem.AddContext(nop.operand1.nome, JSType.number);
          }
          if (nop.operand2 is Lookup
              || nop.operand2 is Member
              || nop.operand2 is Plus) {
            progElem.AddContext(nop.operand2.nome, JSType.number);
          }
        }
        else {
          throw new Exception("Unknown type of NumericBinary");
        }
      }
      else if (node is Plus) {
        Plus p = (Plus)node;
        if (progElem.GetContext(p.operand1.nome).Equals(JSTypeSet.strng)) {
          progElem.AddContext(p.nome, JSType.strng);
          if (p.operand2 is Lookup
              || p.operand2 is Member) {
            progElem.AddContext(p.operand2.nome, JSType.strng);
          }
        }
        if (progElem.GetContext(p.operand2.nome).Equals(JSTypeSet.strng)) {
          progElem.AddContext(p.nome, JSType.strng);
          if (p.operand1 is Lookup
              || p.operand1 is Member) {
            progElem.AddContext(p.operand1.nome, JSType.strng);
          }
        }
        if ((progElem.GetContext(p.operand1.nome).Equals(JSTypeSet.number))
            && (progElem.GetContext(p.operand2.nome).Equals(JSTypeSet.number))) {
          progElem.AddContext(p.nome, JSType.number);
        }
      }
      else if (node is Relational       // <, <=, >, >=
               || node is Equality) {  // ==, !=
        BinaryOp r = (BinaryOp)node;
        if (progElem.GetContext(r.operand1.nome).Equals(JSTypeSet.number)) {
          progElem.AddContext(r.nome, JSType.number);
          if (r.operand2 is Lookup
              || r.operand2 is Member) {
            progElem.AddContext(r.operand2.nome, JSType.number);
          }
        }
        if (progElem.GetContext(r.operand2.nome).Equals(JSTypeSet.number)) {
          progElem.AddContext(r.nome, JSType.number);
          if (r.operand1 is Lookup
              || r.operand1 is Member) {
            progElem.AddContext(r.operand1.nome, JSType.number);
          }
        }
        if ((progElem.GetContext(r.operand1.nome).Equals(JSTypeSet.strng))
            && (progElem.GetContext(r.operand2.nome).Equals(JSTypeSet.strng))) {
          progElem.AddContext(r.nome, JSType.strng);
        }
      }
      else if (node is Assign) {
        Assign a = (Assign)node;
        if (a.lhside is Lookup) {
          JSTypeSet rhsideContext = progElem.GetContext(a.rhside.nome);
          if (rhsideContext.IsSingleton()) {
            progElem.AddContext(a.lhside.nome, rhsideContext.AsJSType());
          }
          else {
              // TODO investigate this case
          }
        }
      }
```

```
  base.Postprocess(node);  // pop node
  return node;
}
```

Appendix B: Set of the programs used for the empirical investigation

min program

```
/*
Depending on the types of the inputs, returns the minimum of two numbers or the
length of the shortest of two strings or a number if it is less than the length of
the second string input or a string if its length is less than the second number
input.*/

function min(i, j) {
    //Based on example from Kapoor and Bowen
    var m = 0;
    if ((typeof(i) == "number")
        && (typeof(j) == "number")) {
      //#BeginNoMutation
      //print("number number");
      //#EndNoMutation
      if (i < j) {
        m = i;
      }
      else {
        m = j;
      }
    }
    else if ((typeof(i) == "string")
             && (typeof(j) == "string")) {
      //#BeginNoMutation
      //print("string string");
      //#EndNoMutation
      if (i.length < j.length) {
        m = i;
      }
      else {
        m = j;
      }
    }
    else if ((typeof(i) == "number")
             && (typeof(j) == "string")) {
      //#BeginNoMutation
      //print("number string");
      //#EndNoMutation
      if (i < (j - 0)) {   // convert j to number
        m = i;
      }
      else {
        m = j;
      }
    }
    else if ((typeof(i) == "string")
             && (typeof(j) == "number")) {
      //#BeginNoMutation
      //print("string number");
      //#EndNoMutation
      if (i.length < (j + "").length) {      // convert j to string
        m = i;
      }
      else {
        m = j;
      }
    }
    else {
      //#BeginNoMutation
      //print("  null ");
      //#EndNoMutation
      m = null;
    }
    var r = m;
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput(r, 9);
    return r;
    #EndNoMutation
  }
}
```

boolStringNumber2 program

```
/*
Sets variable x to number, string and Boolean type and adds the value of each x to
the result.*/

// function under test
  function boolStringNumber2(n) {
    var x = 0;
    var i = 0;
    var y = "";
    var r = 0;
    for (i = 0; i < n; i++) {
      if (i == 1) {
        x = true;
        y = "23";
      }
      else if (i == 2) {
        x = y - 1;
      }
      r += x + 3;
    }
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput(r, 9);
    return r;
    #EndNoMutation
  }
/*
  // function under test
  function boolStringNumber2(n) {
    var x = 0, i = 0, r = 0;
    var b1, b2;
    for (i = 0; i < n; i++) {
      b1 = i == 1;
      b2 = i == 2;
      if (b1) {
        x = b1;
      }
      else if (b2) {
             x = "hello";
      }
      b1 = b1 || ((x == "hello") && x);
      b2 = b1 || ((x == "hello") && (x && b1));
      r += x + 3 + b1 + b2;
    }
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput(r);
    return r;
    #EndNoMutation
  }
*/
}
```

wages program

```
/*
Calculate wages given salary information and number of hours worked on each of 7
days adding on overtime.*/

// function under test
  function wages(code, level, paygrade, hours) {
    var hoursInDay = 8;
    var salary = 0;
    var rate = 11;
    var allowance = 180;   // before tax
    var year = "first";
    var totalHours = 0;
    var bonus = [80, 160, 240];  // added if work overtime
    var overtime = 0;
    var i = 0;
    if (code == 0) {           // i0
      rate = 7;
    }
    if (level == 4000) {       // i1
      if (paygrade == 5) {     // i2
        year = "second";
      }
    }
    if (year == "first") {        // i3
      rate = rate - 3;
    }
    while (i < 7) {            // w0
      totalHours = totalHours + hours[i];
      if (hours[i] > hoursInDay) {      // i4
        overtime += hours[i] - hoursInDay;
      }
      i += 1;
    }
    salary = totalHours * rate;
    salary += bonus[Math.floor(overtime / 7)];
    if (salary > allowance) {  // deduct tax
      salary -= allowance;     // do not tax allowance
      salary -= salary * 0.2;
      salary += allowance;
    }
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput(salary, 9);
    return salary;
    #EndNoMutation
  }
}
```

price program

```
/*
   * Input is array of orders.
   * Each order is object of form {item: [number|string], discount:
[number|string], units: [number|string]}.
   * Calculate price of given item x units applying any discount.
   * item may be a number, in which case it is a unit price.
   * item may be a string, in which case it is the name of the item and the price
must be looked up.
   * discount may be a number, in which case it is a percentage by which the price
is reduced.
   * discount may be a string, in which case it is a code and must be looked up.
   */
  // function under test
  function price(orders) {
    var order;
    var item;
    var discount;
    var units;
    var cost;
    var totalCost = 0;
    var weeks;
    var weeklyCost;
    var i = 0;
    var j = 0;
    for (i = 0; i < orders.length; i++) {
      // calculate cost of order
      order = orders[i];
      item = order.item;
      discount = order.discount;
      units = order.units;
      //print("item " + item + " dis " + discount + " units " + units);
      cost = itemData[item];
      cost -= cost * discount;
      cost *= units;
      //print(item + " " + itemData[item] + " " + cost);
      if (cost >= 180) {
        // split cost across a number of weeks
        weeks = Math.ceil(cost / 30);
        weeklyCost = cost / weeks;
        cost = new Array(weeks);
        for (j = 0; j < weeks; j += 1) {
          cost[j] = Math.ceil(weeklyCost + (weeklyCost * 0.15));
        }
      }
      if ((typeof(totalCost) == "number")
          && (typeof(cost) == "number")) {
        totalCost += cost;
      }
      else if ((typeof(totalCost) == "number")
               && (typeof(cost) == "object")) {
        cost[0] += totalCost;   // previous cost added to first week cost
        totalCost = cost;
      }
      else if ((typeof(totalCost) == "object")
               && (typeof(cost) == "number")) {
```

```
          totalCost[0] += cost;   // previous cost added to first week cost
        }
        else if ((typeof(totalCost) == "object")                 // i.4
                 && (typeof(cost) == "object")) {
          // add corresponding weekly costs into largest array
          if (totalCost.length >= cost.length) {                 // i.5
            for (j = 0; j < cost.length; j++) {
              totalCost[j] += cost[j];
            }
          }
          else {
            for (j = 0; j < totalCost.length; j++) {
              cost[j] += totalCost[j];
            }
            totalCost = cost;
          }
        }
      }
      #BeginNoMutation
      TESCRIPTAClass.TESCRIPTA.defineOutput(totalCost, 60);
      return totalCost;
      #EndNoMutation
    }
  }
```

hazard **program**

The old game of hazard, played with a pair of dice. The first input declares main followed by a sequence of throws of the dice. Outcomes of the game depending on the value of main, and then the numbers on the pair of dice.

```
//The game of hazard, played with a pair of dice.
function hazard(throws) {  // an initial number which declares main followed by a
sequence of throws of the dice
  // outcomes depending on the value of main, 5 to 9 and then the total thrown
  var rules = {1: "invalidInput",
               2: "invalidInput",
               3: "invalidInput",
               4: "invalidInput",
               5: {2: "outs", 3: "outs", 4: "chance", 5: "nicks", 6: "chance", 7:
"chance", 8: "chance", 9: "chance", 10: "chance", 11: "outs", 12: "outs"},
               6: {2: "outs", 3: "outs", 4: "chance", 5: "chance", 6: "nicks", 7:
"chance", 8: "chance", 9: "chance", 10: "chance", 11: "outs", 12: "nicks"},
               7: {2: "outs", 3: "outs", 4: "chance", 5: "chance", 6: "chance", 7:
"nicks", 8: "chance", 9: "chance", 10: "chance", 11: "nicks", 12: "outs"},
               8: {2: "outs", 3: "outs", 4: "chance", 5: "chance", 6: "chance", 7:
"chance", 8: "nicks", 9: "chance", 10: "chance", 11: "outs", 12: "nicks"},
               9: {2: "outs", 3: "outs", 4: "chance", 5: "chance", 6: "chance", 7:
"chance", 8: "chance", 9: "nicks", 10: "chance", 11: "outs", 12: "outs"}};
  var index = 0;  // index input sequence
  var main;       // first input sets this value
  var thrw;
  var thrwValue = 0;
  var chance = 0;
  var rule;
  main = throws[index];
  rule = rules[main];  // value of main determines interpretation of subsequent
throws
  if (typeof(rule) != "object") {
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput(rule, 12);
    print("rule " + rule);
    return rule;
    #EndNoMutation
  }
  index = index + 1;
  thrw = throws[index];  // next input is a pair of dice values
  thrwValue = thrw.first + thrw.second;
  //print("[" + thrw.first + ", " + thrw.second +"]");
  //print("main: " + main + ", rule " + rule);
  var result = rule[thrwValue];
  if (result == "outs") {
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput("lost", 20);
    return "lost";
    #EndNoMutation
  }
  if (result == "nicks") {
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput("won", 20);
    return "won";
```

```
      #EndNoMutation
    }
    if (result == "chance") {
      chance = thrw.first + thrw.second;
      for (index = 2; index < throws.length; index++) {
        thrw = throws[index];
        //print("[" + thrw.first + ", " + thrw.second +"]");
        thrwValue = thrw.first + thrw.second;
        if (thrwValue == chance) {
          #BeginNoMutation
          TESCRIPTAClass.TESCRIPTA.defineOutput("won", 20);
          return "won";
          #EndNoMutation
        }
        else if (thrwValue == main) {
          #BeginNoMutation
          TESCRIPTAClass.TESCRIPTA.defineOutput("lost", 20);
          return "lost";
          #EndNoMutation
        }
      }
      #BeginNoMutation
      TESCRIPTAClass.TESCRIPTA.defineOutput("outOfInput", 20);
      return "outOfInput";
      #EndNoMutation
    }
    else {
      #BeginNoMutation
      TESCRIPTAClass.TESCRIPTA.defineOutput("invalidInput", 20);
      return "invalidInput";
      #EndNoMutation
    }
  }
  }
```

game program

```
/* Board for game is a sequence of squares.
   Player starts at square zero and array of inputs determines moves.
   When player lands on a square, a unit of energy is consumed and:
     - "safe", get next input
     - "mine", player dead, game over
     - "action":
        - "goback", move back prescribed number of squares
        - "food", acquire additional energy
        - "fight", consume given amount of energy, if energy = 0 player dead,
            game over */

  function game(moves) {
    var board = [
      "safe",                           // first square must be safe
      {action: "food", amount: 12},     // 1
      "mine",
      {action: "food", amount: 8},
      "mine",
      "safe",                           // 5
      {action: "goback", dist: 3},
      "mine",                                  // v14
      {action: "food", amount: 13},
      {action: "goback", dist: 11},
      "mine",                           // 10     v21
      "safe",                                  // v22
      {action: "food", amount: 8},
      {action: "food", amount: 7},
      {action: "goback", dist: 4},
      {action: "fight", amount:  7},     // 15
      {action: "fight", amount: 27},
      {action: "fight", amount: 17},
      "safe",                                  // v41
      {action: "goback", dist: 5},
      {action: "goback", dist: 5},       // 20
      "safe",                                  // v48
      {action: "fight", amount: 5},
      {action: "goback", dist: 3},
      {action: "fight", amount: 33},
      "safe",                            // 25    v58
      {action: "fight", amount: 35},
      {action: "fight", amount:  5},
      {action: "goback", dist: 6},
      "mine",
      {action: "fight", amount: 2},      // 30
      "safe",
      "mine",
      {action: "fight", amount: 19},
      "safe",
      {action: "goback", dist: 11},      // 35
      "safe",
      {action: "goback", dist: 11},             //  v83
      "mine",
      {action: "goback", dist: 3},
      {action: "fight", amount: 9},      // 40
      {action: "goback", dist: 13},
      "safe",
      {action: "goback", dist: 11},
      {action: "fight", amount: 9},
      {action: "goback", dist: 18},      // 45
      "mine",
      "safe",
      {action: "fight", amount: 9},
      "mine",
      {action: "goback", dist: 6},      // 50
      {action: "fight", amount: 7},
      "mine",
      "end"
      ];
    var gameOver = false;
    var energy = 5;  // must not fall below 0
    var energyIsEmpty = false;  // true when energy = 0
    var fullEnergy = 50;
    var energyIsFull = false;  // true when energy = fullEnergy
    var index = -1;
    var square = 0;
    var squareValue = board[square];
    while (!gameOver) {
      if (squareValue == "safe") {
        // get next input
        index += 1;
        if (index == moves.length) {
          gameOver = true;
        }
        else {
          square += moves[index];
          energy -= 1;
          if (square >= board.length) { // board overflow, reverse move
            square = (board.length - 1) - (square - (board.length - 1));
            energy -= 1;
            if (square < 0) {  // board underflow, goto start, infeasible
              square = 0;
              energy -= 1;
            }
          }
          if (energy < fullEnergy) {
            energyIsFull = false;
          }
          squareValue = board[square];
          //print("moves[" + index + "] = " + moves[index] + ", square = " + square
+ ", " + stringSquare(squareValue));
        }
      }
      // move square if not safe
      else if ((typeof squareValue) == "object") {
        if (squareValue.action == "goback") {
          square -= squareValue.dist;
          if (square < 0) {
            square = - square;
          }
          squareValue = board[square];
          //print("back to square = " + square + ", " + stringSquare(squareValue));
        }
        else if (squareValue.action == "food") {
          energy += squareValue.amount;
          if (energy >= fullEnergy) {
            energyIsFull = true;
          }
          squareValue = "safe";
          //print("energy = " + energy + (energyIsFull?" energyIsFull":""));
        }
        else if (squareValue.action == "fight") {   // f infeasible
          energy -= squareValue.amount;
          if (energy < fullEnergy) {
            energyIsFull = false;
          }
          if (energy == 0) {
            energyIsEmpty = true;
          }
          if (energy < 0) {
            gameOver = true;
          }
          squareValue = "safe";
          //print("energy = " + energy);
        }
      }
      else if (squareValue == "mine") {
        gameOver = true;
      }
      else if (squareValue == "end") {    // f infeasible
        gameOver = true;
      }
    }
    //if (energyIsEmpty) {
      //print("energyIsEmpty");
    //}
    //if (energyIsFull) {
      //print("energyIsFull");
    //}
    if (squareValue == "end") {
      //print("end " + energy);
      //if (energyIsFull) {
        //print("energyIsFull");
      //}
      #BeginNoMutation
      TESCRIPTAClass.TESCRIPTA.defineOutput("end" + energy + energyIsFull +
energyIsEmpty, 20);
      return "end" + energy;
      #EndNoMutation
    }
    else {
      //print("gameOver");
      #BeginNoMutation
      TESCRIPTAClass.TESCRIPTA.defineOutput("gameOver" + energy + energyIsFull +
energyIsEmpty, 26);
      return "gameOver" + energy;
      #EndNoMutation
    }
  }
}
```

Student program

Input is a set of module marks for a set of students. If input module matches module in marksheet, mark is added otherwise mark for new module.

```
// function under test
  function student(students) {
    var markSheet = [
      {name: "John", marks: [{mod: "0101", mark: 30}, {mod: "0102", mark: 35}]},
      {name: "Jane", marks: [{mod: "0101", mark: 40}, {mod: "0200", mark: 20}]},
      {name: "Bill", marks: [{mod: "0104", mark:  0}, {mod: "0200", mark: 70}]}];
    var name;
    var mod;
    var mark = 0;
    var markOld = 0;
    var studnt;
    var studentMarks;
    var markSheetRow;
    var studentMark;
    var i, j, k = 0;
    for (i = 0; i < students.length; i++) {
      studnt = students[i];
      name = studnt.name;
      mod = studnt.mod;
      mark = studnt.mark;
      for (j = 0; j < markSheet.length; j++) {
        if (name == markSheet[j].name) {
          markSheetRow = markSheet[j];
          studentMarks = markSheetRow.marks;
          for (k = 0; k < studentMarks.length; k++) {
            studentMark = studentMarks[k];
            if (mod == studentMark.mod) {
              markOld = studentMark.mark;
              studentMark.mark = markOld + mark;
            }
          }
        }
      }
    }
    #BeginNoMutation
    TESCRIPTAClass.TESCRIPTA.defineOutput(markSheet);
    return markSheet;
    #EndNoMutation
  }
}
```

Appendix C: Sample outputs produced by Tescripta using Dynamic method for the `min` program

min program after instrumentation. Labels are added to identify every object in the program. Assignment statement are labelled as s0, s1 ,…, s12. The if statements are labelled as i0, i1, …, i7

```
function min(i_, j_)
  var i;
  i .0  =  .0 i_ .0  s0;
  var j;
  j .0  =  .1 j_ .0  s1;
  var m;
  m .0  =  .2 0 v0  s2;
  if  i0 ((typeof i .1  ==  .0 "number" v1) && .0 (typeof j .1  == .1
 "number" v2)) {
    if  i1 (i .2  <  .0 j .2) {
      m .1  = .3 i .3  s3;
    }
    else {
      m .2  = .4 j .3  s4;
    }
  }
  else if  i2 ((typeof i .4 == .2 "string" v3) &&  .1 (typeof j .4  ==  .3 "string"
v4)) {
    if  i3 (i .5 .length m0  < .1 j .5 .length m1) {
      m .3  = .5 i .6  s5;
    }
    else {
      m .4  = .6 j .6  s6;
    }
  }
  else if  i4 ((typeof i .7 == .4 "number" v5) &&  .2 (typeof j .7  ==  .5 "string"
v6)) {
    if  i5 (i .8  <  .2 (j .8  -  .0 0 v7)) {
      m .5  =  .7 i .9  s7;
    }
    else {
      m .6  =  .8 j .9  s8;
    }
  }
  else if  i6 ((typeof i .10 == .6 "string" v8) && .3 (typeof j .10  ==  .7
"number" v9)) {
    if  i7 (i .11 .length m2 < .3 (j .11  +  .0 "" v10).length m3) {
      m .7  =  .9 i .12  s9;
    }
    else {
      m .8  =  .10 j .12  s10;
    }
  }
  else {
    m .9  =  .11 null  v11  s11;
  }
  var r;
  r .0  =  .12 m .10  s12;
  TESCRIPTAClass.TESCRIPTA.defineOutput(r, 9);
```

v1: the second occurrence of a literal
==.0: the first occurrence of ==
j.2: the third occurrence of j
m.1: the second occurrence of m

<.1 the second occurrence of <
m0: the first occurrence of an object member
=.5: the sixth occurrence of =

The tester defines the output of min to be r

```
  return r;
}
```

Test number followed by the values of argument i and argument j and the output r

```
(Test 0 i -4 j  output -4 execCount 8 )
(Test 1 i abc j ab output ab execCount 18 )
(Test 2 i 10 j 10 output 10 execCount 6 )
(Test 3 i 5 j 0 output 0 execCount 9 )
(Test 4 i aa j bb output bb execCount 13 )
(Test 5 i 4 j 5 output 4 execCount 23 )
(Test 6 i number j 4 output 4 execCount 30 )
(Test 7 i 0 j 1 output 0 execCount 42 )
(Test 8 i true j false output null execCount 31 )
(Test 9 i Od+ j 1629 output Od+ execCount 35 )
(Test 10 i -33 j -383 output -383 execCount 15 )
(Test 11 i  j -383 output  execCount 3 )
(Test 12 i 4 j  output  execCount 4 )
(Test 13 i JHw j 563 output 563 execCount 20 )
(Test 14 i 0 j ab output ab execCount 41 )
(Test 15 i  j  output  execCount 7 )
(Test 16 i ab j abc output ab execCount 19 )
(Test 17 i -44 j -40 output -44 execCount 21 )
(Test 18 i  j ]h" output  execCount 34 )
```

Set of mutation adequate tests used to kill the mutants of min program.

Statistical information of the method used to generate mutants and a summary of the number of mutants killed (k, t or x), missed(m) live, type incorrect(i), not reached(n), equivalent(e). Also, number and time of executions to perform the mutation analysis.

```
1 Total runtype mutants 429, dead(k, t or x) 308, missed(m) 82, type incorrect(i)
39, not reached(n) 0, equiv(e) 0, execs 1705, time 00:00:05.2413223
```

Element label

A mutant generated for s0 element

```
s0 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

  i .0  =  .0 i_ .0  s0; s0
  var j;   1, 0 k null 1 k null 2 k null 3 k null 4 k null 5 k null 6 k null 7 k
   null 8 m 9 k null 10 k null 11 k null 12 k null 13 k null 14 k null 15 k null 16
   k null 17 k null 18 k null, comp 0.1

i.0 i  none i
    0  0,  total 0 dead 0 miss 0
```

The original expression (i.e., before mutation)

Test number

```
s1 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

   j .0  =  .1 j_ .0  s1; s1
  var m;   1, 0 k null 1 k null 2 k null 3 k null 4 k null 5 k null 6 k null 7 k
```

```
 null 8 m 9 k null 10 k null 11 k null 12 k null 13 k null 14 k null 15 k null 16 k
 null 17 k null 18 k null, comp 0.1

j.0 j i   none j
    0 1k  0,  total 1 dead 1 miss 0
```

Total number of mutants generated for s2.
0 missed (live) mutant
1 killed (dead) and

```
  j .0  =  .1 j_ .0  s1; j.0
  i .0  =  .1 j_ .0  s1;   1, 0 i 1 k null 2 k null 3 k null 4 k null 5 k null 6 i
           7 k null 8 m 9 i 10 k null 11 i 12 i 13 i 14 i 15 k null 16 k null 17 k
           null 18 k null, comp 0.4

s2 expn deleteExpn  none expn
   0    1m          0,  total 1 dead 0 miss 1
```

The first mutant is missedm (m)

```
  m .0  =  .2 0 v0  s2; s2
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
       v2)) {   1, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 m 10 m 11 m 12 m 13 m 14
       m 15 m 16 m 17 m 18 m, comp 1.0

m.0 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0
```

Mutants generated for m.0 by replacing m with i and j.

```
  m .0  =  .2 0 v0  s2; m.0
  i .0  =  .2 0 v0  s2;   1, 0 k  1 i 2 k 0 3 m 4 i 5 k 0 6 i 7 m 8 i 9 i 10 m 11 i
                        12 m 13 i 14 m 15 i 16 i 17 k -40 18 i, comp 0.8
  j .0  =  .2 0 v0  s2;   2, 0 i 1 i 2 k 0 3 m 4 i 5 k 0 6 k 0 7 m 8 i 9 k 0 10 k
            33 11 m 12 i 13 k 0 14 i 15 i 16 i 17 m 18 i, comp 0.7
```

Competence of mutants

```
v0 0 c1 sub1 zpush i  j  m   none 0
   0 1m 2m   3x    4m 5m 6i  0,  total 6 dead 1 miss 4 tin 1

  m .0  =  .2 0 v0  s2; v0
  m .0  =  .2 c1(0 v0 ) s2;   1, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 m 10 m 11 m
                        12 m 13 m 14 m 15 m 16 m 17 m 18 m, comp 1.0
  m .0  =  .2 sub1(0 v0 ) s2;   2, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 m 10 m 11
                        m 12 m 13 m 14 m 15 m 16 m 17 m 18 m, comp 1.0
  m .0  =  .2 zpush(0 v0 ) s2;   3, 0 x 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 x 10 x 11
                         x 12 x 13 x 14 x 15 x 16 x 17 x 18 x, comp 0.0
  m .0  =  .2 i v0  s2;   4, 0 m 1 i 2 m 3 m 4 i 5 m 6 i 7 m 8 i 9 i 10 m 11 i 12 m
                        13 i 14 m 15 i 16 i 17 m 18 i, comp 1.0
  m .0  =  .2 j v0  s2;   5, 0 i 1 i 2 m 3 m 4 i 5 m 6 m 7 m 8 i 9 m 10 m 11 m 12 i
                       13 m 14 i 15 i 16 i 17 m 18 i, comp 1.0
  m .0  =  .2 m v0  s2;   6, 0 i 1 i 2 i 3 i 4 i 5 i 6 i 7 i 8 i 9 i 10 i 11 i 12 i
                        13 i 14 i 15 i 16 i 17 i 18 i, comp 1.0
```

Type-incorrect mutant(i)

```
&&.0 && ||  none &&
     0  1k  0,  total 1 dead 1 miss 0

  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { &&.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) ||  .0 (typeof j .1  ==  .1 "number"
v2)) {   1, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12 m 13 m
14 m 15 m 16 m 17 m 18 m, comp 0.9

==.0 ==  != <  <= >  >=  none ==
     0  1k 2k 3m 4k 5k  0,  total 5 dead 4 miss 1
```

Mutants of the first occurrence of ==

```
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { ==.0
  if  i0 ((typeof i .1  !=  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   1, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 k 1629 10 k
null 11 k -383 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.6
  if  i0 ((typeof i .1  <  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   2, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m 10 k null
11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof i .1  <=  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   3, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 m 10 m 11 m 12 m 13 m 14 m 15 m
16 m 17 m 18 m, comp 1.0
  if  i0 ((typeof i .1  >  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   4, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 k 1629 10 k
null 11 k -383 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.6
  if  i0 ((typeof i .1  >=  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   5, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12 m 13 m
14 m 15 m 16 m 17 m 18 m, comp 0.9

i.1 i c1 sub1 zpush c0 add1 neg abs negabs cfalse ctrue logneg cEmpty deadOnEmpty
number string     j   m    none i
    0 1k 2k   3x    4k 5m   6k  7k  8k     9k     10k   11k    12k    13x
14k    15k    16k 17k 18k  0,  total 18 dead 17 miss 1
                                                      Mutant killed by exception

  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { i.1
  if  i0 ((typeof c1(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   1, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12 m 13 m
14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof sub1(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   2, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12
m 13 m 14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof zpush(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   3, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 x 8 m 9 m 10 m 11 m 12 m 13 m 14
x 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof c0(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   4, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12 m 13 m
14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof add1(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   5, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 m 10 m 11 m 12 m 13 m 14
m 15 m 16 m 17 m 18 m, comp 1.0
  if  i0 ((typeof  - (i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   6, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12
m 13 m 14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof Math.Abs(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   7, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12
m 13 m 14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof  - (Math.Abs(i .1)) ==  .0 "number" v1) &&  .0 (typeof j .1  ==
.1 "number" v2)) {   8, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383
12 m 13 m 14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof cfalse(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   9, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m 10
k null 11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof ctrue(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   10, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m
10 k null 11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
```

```
  if  i0 ((typeof !(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   11, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m 10 k null
11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof cEmpty(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   12, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m
10 k null 11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof deadOnEmpty(i .1) ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   13, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 m 10 m 11 x 12 m 13 m
14 m 15 x 16 m 17 m 18 x, comp 0.8
  if  i0 ((typeof "number" .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   14, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m
10 k null 11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof "string" .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
"number" v2)) {   15, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m
10 k null 11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof "" .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   16, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m 10 k null
11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
  if  i0 ((typeof j .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   17, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12 m 13 m
14 m 15 m 16 m 17 m 18 m, comp 0.9
  if  i0 ((typeof m .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) {   18, 0 m 1 m 2 m 3 m 4 m 5 m 6 m 7 m 8 m 9 k 1629 10 m 11 k -383 12 m 13 m
14 m 15 m 16 m 17 m 18 m, comp 0.9
                                                          string mutations of
                                                          “number” string
v1 number cEmpty string    i  j  m   none number
   0      1k     2k     3k 4m 5m 6i  0,  total 6 dead 3 miss 2 tin 1

  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { v1
     if  i0 ((typeof i .1  ==  .0 cEmpty("number" v1)) &&  .0 (typeof j .1  ==  .1
   "number" v2)) {   1, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m
   10 k null 11 m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
     if  i0 ((typeof i .1  ==  .0 "string" v1) &&  .0 (typeof j .1  ==  .1 "number"
   v2)) {   2, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 k 1629 10
   k null 11 k -383 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.6
     if  i0 ((typeof i .1  ==  .0 "" v1) &&  .0 (typeof j .1  ==  .1 "number" v2))
   {   3, 0 m 1 m 2 k null 3 k null 4 m 5 k null 6 m 7 k null 8 m 9 m 10 k null 11
   m 12 m 13 m 14 m 15 m 16 m 17 k null 18 m, comp 0.7
     if  i0 ((typeof i .1  ==  .0 i v1) &&  .0 (typeof j .1  ==  .1 "number" v2)) {
   4, 0 i 1 m 2 i 3 i 4 m 5 i 6 m 7 i 8 i 9 m 10 i 11 m 12 i 13 m 14 i 15 m 16 m 17
   i 18 m, comp 1.0
     if  i0 ((typeof i .1  ==  .0 j v1) &&  .0 (typeof j .1  ==  .1 "number" v2)) {
   5, 0 m 1 m 2 i 3 i 4 m 5 i 6 i 7 i 8 i 9 i 10 i 11 i 12 m 13 i 14 m 15 m 16 m 17
   i 18 m, comp 1.0
     if  i0 ((typeof i .1  ==  .0 m v1) &&  .0 (typeof j .1  ==  .1 "number" v2)) {
   6, 0 i 1 i 2 i 3 i 4 i 5 i 6 i 7 i 8 i 9 i 10 i 11 i 12 i 13 i 14 i 15 i 16 i 17
   i 18 i, comp 1.0

   ==.1 == != <  <= >  >=  none ==
        0  1k 2k 3m 4k 5m  0,  total 5 dead 3 miss 2

  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { ==.1
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  !=  .1 "number"
v2)) {   1, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10 k null
11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
```

```
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  <  .1 "number"
v2)) {   2, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10 k null
11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  <=  .1 "number"
v2)) {   3, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14 m 15 n
16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  >  .1 "number"
v2)) {   4, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10 k null
11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  >=  .1 "number"
v2)) {   5, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14 m 15 n
16 n 17 m 18 n, comp 1.0

j.1 j c1 sub1 zpush c0 add1 neg abs negabs ctrue cEmpty deadOnEmpty number string
i   m    none j
    0 1m 2m   3x    4m 5m   6m  7m  8m     9k    10k    11x         12k    13k
14k 15m 16m  0,  total 16 dead 7 miss 9

  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { j.1
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof c1(j .1) ==  .1 "number"
v2)) {   1, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14 m 15 n
16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof sub1(j .1) ==  .1
"number" v2)) {   2, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14
m 15 n 16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof zpush(j .1) ==  .1
"number" v2)) {   3, 0 m 1 n 2 m 3 x 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14
m 15 n 16 n 17 m 18 n, comp 0.9
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof c0(j .1) ==  .1 "number"
v2)) {   4, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14 m 15 n
16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof add1(j .1) ==  .1
"number" v2)) {   5, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14
m 15 n 16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof  - (j .1) ==  .1
"number" v2)) {   6, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14
m 15 n 16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof Math.Abs(j .1) ==  .1
"number" v2)) {   7, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14
m 15 n 16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof  - (Math.Abs(j .1)) ==
.1 "number" v2)) {   8, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n
14 m 15 n 16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof ctrue(j .1) ==  .1
"number" v2)) {   9, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10
k null 11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof cEmpty(j .1) ==  .1
"number" v2)) {   10, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n
10 k null 11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof deadOnEmpty(j .1) ==  .1
"number" v2)) {   11, 0 x 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 x 13 n
14 m 15 n 16 n 17 m 18 n, comp 0.9
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof "number" .1  ==  .1
"number" v2)) {   12, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n
10 k null 11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
```

```
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof "string" .1  ==  .1
"number" v2)) {   13, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n
10 k null 11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof "" .1  ==  .1 "number"
v2)) {   14, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10 k null
11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof i .1  ==  .1 "number"
v2)) {   15, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14 m 15 n
16 n 17 m 18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof m .1  ==  .1 "number"
v2)) {   16, 0 m 1 n 2 m 3 m 4 n 5 m 6 n 7 m 8 n 9 n 10 m 11 n 12 m 13 n 14 m 15 n
16 n 17 m 18 n, comp 1.0

v2 number cEmpty string    i  j  m   none number
   0      1k     2k     3k 4i 5m 6i  0,  total 6 dead 3 miss 1 tin 2

  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "number"
v2)) { v2
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1
cEmpty("number" v2))) {   1, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8
n 9 n 10 k null 11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "string"
v2)) {   2, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10 k null
11 n 12 m 13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 "" v2)) {
3, 0 m 1 n 2 k null 3 k null 4 n 5 k null 6 n 7 k null 8 n 9 n 10 k null 11 n 12 m
13 n 14 m 15 n 16 n 17 k null 18 n, comp 0.7
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 i v2)) {
4, 0 i 1 n 2 i 3 i 4 n 5 i 6 n 7 i 8 n 9 n 10 i 11 n 12 i 13 n 14 i 15 n 16 n 17 i
18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 j v2)) {
5, 0 m 1 n 2 i 3 i 4 n 5 i 6 n 7 i 8 n 9 n 10 i 11 n 12 m 13 n 14 m 15 n 16 n 17 i
18 n, comp 1.0
  if  i0 ((typeof i .1  ==  .0 "number" v1) &&  .0 (typeof j .1  ==  .1 m v2)) {
6, 0 i 1 n 2 i 3 i 4 n 5 i 6 n 7 i 8 n 9 n 10 i 11 n 12 i 13 n 14 i 15 n 16 n 17 i
18 n, comp 1.0

<.0 < == != <= >  >=  none <
    0 1k 2k 3m 4k 5k  0,  total 5 dead 4 miss 1

    if  i1 (i .2  <  .0 j .2) { <.0
    if  i1 (i .2  ==  .0 j .2) {   1, 2 m 3 m 5 k 5 7 k 1 10 m 17 k -40, comp 0.5
    if  i1 (i .2  !=  .0 j .2) {   2, 2 m 3 k 5 5 m 7 m 10 k -33 17 m, comp 0.7
    if  i1 (i .2  <=  .0 j .2) {   3, 2 m 3 m 5 m 7 m 10 m 17 m, comp 1.0
    if  i1 (i .2  >  .0 j .2) {   4, 2 m 3 k 5 5 k 5 7 k 1 10 k -33 17 k -40, comp
0.2
    if  i1 (i .2  >=  .0 j .2) {   5, 2 m 3 k 5 5 k 5 7 k 1 10 k -33 17 k -40, comp
0.2

i.2 i c1 sub1 zpush c0 add1 neg abs negabs j  m    none i
    0 1k 2m   3x    4k 5k   6k  7k  8k     9k 10k  0,  total 10 dead 9 miss 1

    if  i1 (i .2  <  .0 j .2) { i.2
    if  i1 (c1(i .2) <  .0 j .2) {   1, 2 m 3 m 5 m 7 k 1 10 m 17 k -40, comp 0.7
    if  i1 (sub1(i .2) <  .0 j .2) {   2, 2 m 3 m 5 m 7 m 10 m 17 m, comp 1.0
    if  i1 (zpush(i .2) <  .0 j .2) {   3, 2 m 3 m 5 m 7 x 10 m 17 m, comp 0.8
    if  i1 (c0(i .2) <  .0 j .2) {   4, 2 m 3 m 5 m 7 m 10 m 17 k -40, comp 0.8
    if  i1 (add1(i .2) <  .0 j .2) {   5, 2 m 3 m 5 k 5 7 k 1 10 m 17 m, comp 0.7
    if  i1 ( - (i .2) <  .0 j .2) {   6, 2 m 3 k 5 5 m 7 m 10 m 17 k -40, comp 0.7
    if  i1 (Math.Abs(i .2) <  .0 j .2) {   7, 2 m 3 m 5 m 7 m 10 m 17 k -40, comp
0.8
    if  i1 ( - (Math.Abs(i .2)) <  .0 j .2) {   8, 2 m 3 k 5 5 m 7 m 10 m 17 m,
comp 0.8
    if  i1 (j .2  <  .0 j .2) {   9, 2 m 3 m 5 k 5 7 k 1 10 m 17 k -40, comp 0.5
    if  i1 (m .2  <  .0 j .2) {   10, 2 m 3 m 5 m 7 m 10 m 17 k -40, comp 0.8

j.2 j c1 sub1 zpush c0 add1 neg abs negabs i  m    none j
    0 1k 2k   3x    4k 5m   6k  7k  8k     9k 10k  0,  total 10 dead 9 miss 1

    if  i1 (i .2  <  .0 j .2) { j.2
    if  i1 (i .2  <  .0 c1(j .2)) {   1, 2 m 3 m 5 k 5 7 m 10 k -33 17 m, comp 0.7
    if  i1 (i .2  <  .0 sub1(j .2)) {   2, 2 m 3 m 5 k 5 7 k 1 10 m 17 m, comp 0.7
    if  i1 (i .2  <  .0 zpush(j .2)) {   3, 2 m 3 x 5 m 7 m 10 m 17 m, comp 0.8
    if  i1 (i .2  <  .0 c0(j .2)) {   4, 2 m 3 m 5 k 5 7 k 1 10 k -33 17 m, comp
0.5
    if  i1 (i .2  <  .0 add1(j .2)) {   5, 2 m 3 m 5 m 7 m 10 m 17 m, comp 1.0
    if  i1 (i .2  <  .0  - (j .2)) {   6, 2 m 3 m 5 k 5 7 k 1 10 k -33 17 m, comp
0.5
    if  i1 (i .2  <  .0 Math.Abs(j .2)) {   7, 2 m 3 m 5 m 7 m 10 k -33 17 m, comp
0.8
    if  i1 (i .2  <  .0  - (Math.Abs(j .2))) {   8, 2 m 3 m 5 k 5 7 k 1 10 m 17 m,
comp 0.7
    if  i1 (i .2  <  .0 i .2) {   9, 2 m 3 m 5 k 5 7 k 1 10 m 17 k -40, comp 0.5
    if  i1 (i .2  <  .0 m .2) {   10, 2 m 3 m 5 k 5 7 k 1 10 k -33 17 m, comp 0.5

s3 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

      m .1  =  .3 i .3  s3; s3
    else {   1, 5 k 0 7 m 17 k 0, comp 0.3

m.1 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0

      m .1  =  .3 i .3  s3; m.1
      i .1  =  .3 i .3  s3;   1, 5 k 0 7 m 17 k 0, comp 0.3
      j .1  =  .3 i .3  s3;   2, 5 k 0 7 m 17 k 0, comp 0.3

i.3 i c1 sub1 zpush c0 add1 neg abs negabs j  m    none i
    0 1k 2k   3x    4k 5k   6k  7k  8k     9k 10k  0,  total 10 dead 10 miss 0

      m .1  =  .3 i .3  s3; i.3
      m .1  =  .3 c1(i .3 ) s3;   1, 5 k 1 7 k 1 17 k 1, comp 0.0
      m .1  =  .3 sub1(i .3 ) s3;   2, 5 k 3 7 k -1 17 k -45, comp 0.0
      m .1  =  .3 zpush(i .3 ) s3;   3, 5 m 7 x 17 m, comp 0.7
      m .1  =  .3 c0(i .3 ) s3;   4, 5 k 0 7 m 17 k 0, comp 0.3
      m .1  =  .3 add1(i .3 ) s3;   5, 5 k 5 7 k 1 17 k -43, comp 0.0
      m .1  =  .3  - (i .3 ) s3;   6, 5 k -4 7 m 17 k 44, comp 0.3
      m .1  =  .3 Math.Abs(i .3 ) s3;   7, 5 m 7 m 17 k 44, comp 0.7
      m .1  =  .3  - (Math.Abs(i .3)) s3;   8, 5 k -4 7 m 17 m, comp 0.7
      m .1  =  .3 j .3  s3;   9, 5 k 5 7 k 1 17 k -40, comp 0.0
      m .1  =  .3 m .3  s3;   10, 5 k 0 7 m 17 k 0, comp 0.3

s4 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0
```

```
      m .2  =  .4 j .3  s4; s4
  }   1, 2 k 0 3 m 10 k 0, comp 0.3

m.2 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0

      m .2  =  .4 j .3  s4; m.2
      i .2  =  .4 j .3  s4;   1, 2 k 0 3 m 10 k 0, comp 0.3
      j .2  =  .4 j .3  s4;   2, 2 k 0 3 m 10 k 0, comp 0.3

j.3 j c1 sub1 zpush c0 add1 neg abs negabs i  m    none j
    0 1k 2k   3x    4k 5k   6k  7k  8k     9k 10k  0,  total 10 dead 10 miss 0

      m .2  =  .4 j .3  s4; j.3
      m .2  =  .4 c1(j .3 ) s4;   1, 2 k 1 3 k 1 10 k 1, comp 0.0
      m .2  =  .4 sub1(j .3 ) s4;   2, 2 k 9 3 k -1 10 k -384, comp 0.0
      m .2  =  .4 zpush(j .3 ) s4;   3, 2 m 3 x 10 m, comp 0.7
      m .2  =  .4 c0(j .3 ) s4;   4, 2 k 0 3 m 10 k 0, comp 0.3
      m .2  =  .4 add1(j .3 ) s4;   5, 2 k 11 3 k 1 10 k -382, comp 0.0
      m .2  =  .4  - (j .3 ) s4;   6, 2 k -10 3 m 10 k 383, comp 0.3
      m .2  =  .4 Math.Abs(j .3 ) s4;   7, 2 m 3 m 10 k 383, comp 0.7
      m .2  =  .4  - (Math.Abs(j .3)) s4;   8, 2 k -10 3 m 10 m, comp 0.7
      m .2  =  .4 i .3  s4;   9, 2 m 3 k 5 10 k -33, comp 0.3
      m .2  =  .4 m .3  s4;   10, 2 k 0 3 m 10 k 0, comp 0.3

&&.1 && ||  none &&
     0  1k  0,  total 1 dead 1 miss 0

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { &&.1
  else if  i2 ((typeof i .4  ==  .2 "string" v3) ||  .1 (typeof j .4  ==  .3
"string" v4)) {   1, 0 k  1 m 4 m 6 m 8 m 9 k 1629 11 k -383 12 m 13 m 14 m 15 m 16
m 18 m, comp 0.8

==.2 == != <  <= >  >=  none ==
     0  1k 2k 3k 4k 5m  0,  total 5 dead 4 miss 1

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { ==.2
  else if  i2 ((typeof i .4  !=  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   1, 0 k  1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.5
  else if  i2 ((typeof i .4  <  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   2, 0 k  1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.5
  else if  i2 ((typeof i .4  <=  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   3, 0 k  1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 0.9
  else if  i2 ((typeof i .4  >  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   4, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  >=  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   5, 0 m 1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 1.0

i.4 i c1 sub1 zpush c0 add1 neg abs negabs cfalse ctrue logneg cEmpty deadOnEmpty
number string     j   m    none i
```

```
    0 1k 2k   3x    4k 5m   6k  7k  8k     9k     10k   11k    12k    13x
14k    15k    16k 17k 18k  0,  total 18 dead 17 miss 1

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { i.4
  else if  i2 ((typeof c1(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   1, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof sub1(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   2, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof zpush(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   3, 0 m 1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 x 15 m 16 m 18 m,
comp 0.9
  else if  i2 ((typeof c0(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   4, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof add1(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   5, 0 m 1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 1.0
  else if  i2 ((typeof  - (i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   6, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof Math.Abs(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==
.3 "string" v4)) {   7, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof  - (Math.Abs(i .4)) ==  .2 "string" v3) &&  .1 (typeof j .4
==  .3 "string" v4)) {   8, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m
15 k null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof cfalse(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   9, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof ctrue(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   10, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof !(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   11, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof cEmpty(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   12, 0 k  1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 0.9
  else if  i2 ((typeof deadOnEmpty(i .4) ==  .2 "string" v3) &&  .1 (typeof j .4
==  .3 "string" v4)) {   13, 0 m 1 m 4 m 6 m 8 m 9 m 11 x 12 m 13 m 14 m 15 x 16 m
18 x, comp 0.8
  else if  i2 ((typeof "number" .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   14, 0 k  1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 0.9
  else if  i2 ((typeof "string" .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   15, 0 k  1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 0.9
  else if  i2 ((typeof "" .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   16, 0 k  1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 0.9
  else if  i2 ((typeof j .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   17, 0 k  1 m 4 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 m 16 m 18 m,
comp 0.9
```

```
  else if  i2 ((typeof m .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   18, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6

v3 string cEmpty number    i  j  m   none string
   0      1k     2k     3k 4k 5k 6i  0,  total 6 dead 5 miss 0 tin 1

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { v3
  else if  i2 ((typeof i .4  ==  .2 cEmpty("string" v3)) &&  .1 (typeof j .4  ==
.3 "string" v4)) {   1, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "number" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) {   2, 0 k  1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k
null 16 k null 18 k null, comp 0.5
  else if  i2 ((typeof i .4  ==  .2 "" v3) &&  .1 (typeof j .4  ==  .3 "string"
v4)) {   3, 0 m 1 k null 4 k null 6 m 8 m 9 m 11 m 12 m 13 m 14 m 15 k null 16 k
null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 i v3) &&  .1 (typeof j .4  ==  .3 "string" v4))
{   4, 0 i 1 k null 4 k null 6 m 8 i 9 m 11 m 12 i 13 m 14 i 15 k null 16 k null 18
k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 j v3) &&  .1 (typeof j .4  ==  .3 "string" v4))
{   5, 0 m 1 k null 4 k null 6 i 8 i 9 i 11 i 12 m 13 i 14 m 15 k null 16 k null 18
k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 m v3) &&  .1 (typeof j .4  ==  .3 "string" v4))
{   6, 0 i 1 i 4 i 6 i 8 i 9 i 11 i 12 i 13 i 14 i 15 i 16 i 18 i, comp 1.0

==.3 == != <  <= >  >=  none ==
     0  1k 2k 3k 4k 5m  0,  total 5 dead 4 miss 1

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { ==.3
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  !=  .3
"string" v4)) {   1, 0 n 1 k null 4 k null 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14
n 15 k null 16 k null 18 k null, comp 0.5
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  <  .3
"string" v4)) {   2, 0 n 1 k null 4 k null 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14
n 15 k null 16 k null 18 k null, comp 0.5
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  <=  .3
"string" v4)) {   3, 0 n 1 m 4 m 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14 n 15 m 16
m 18 m, comp 0.8
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  >  .3
"string" v4)) {   4, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  >=  .3
"string" v4)) {   5, 0 n 1 m 4 m 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 m 16 m 18 m,
comp 1.0

j.4 j c0 c1 add1 sub1 neg abs negabs zpush ctrue cEmpty deadOnEmpty number string
i   m    none j
    0 1k 2k 3m   4k   5k  6k  7k     8m    9k    10k    11x         12k    13k
14k 15k 16k  0,  total 16 dead 14 miss 2

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { j.4
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof c0(j .4) ==  .3
"string" v4)) {   1, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
```

```
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof c1(j .4) ==  .3
"string" v4)) {   2, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof add1(j .4) ==  .3
"string" v4)) {   3, 0 n 1 m 4 m 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 m 16 m 18 m,
comp 1.0
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof sub1(j .4) ==  .3
"string" v4)) {   4, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof  - (j .4) ==  .3
"string" v4)) {   5, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof Math.Abs(j .4) ==
.3 "string" v4)) {   6, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof  - (Math.Abs(j .4))
==  .3 "string" v4)) {   7, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n
15 k null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof zpush(j .4) ==  .3
"string" v4)) {   8, 0 n 1 m 4 m 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 m 16 m 18 m,
comp 1.0
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof ctrue(j .4) ==  .3
"string" v4)) {   9, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof cEmpty(j .4) ==  .3
"string" v4)) {   10, 0 n 1 m 4 m 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14 n 15 m 16
m 18 m, comp 0.8
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof deadOnEmpty(j .4)
==  .3 "string" v4)) {   11, 0 n 1 m 4 m 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 x 16 m
18 m, comp 0.9
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof "number" .4  ==  .3
"string" v4)) {   12, 0 n 1 m 4 m 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14 n 15 m 16
m 18 m, comp 0.8
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof "string" .4  ==  .3
"string" v4)) {   13, 0 n 1 m 4 m 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14 n 15 m 16
m 18 m, comp 0.8
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof "" .4  ==  .3
"string" v4)) {   14, 0 n 1 m 4 m 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14 n 15 m 16
m 18 m, comp 0.8
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof i .4  ==  .3
"string" v4)) {   15, 0 n 1 m 4 m 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14 n 15 m 16
m 18 m, comp 0.8
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof m .4  ==  .3
"string" v4)) {   16, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k
null 16 k null 18 k null, comp 0.6

v4 string cEmpty number    i  j  m   none string
   0      1k     2k     3k 4k 5k 6i  0,  total 6 dead 5 miss 0 tin 1

  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"string" v4)) { v4
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
cEmpty("string" v4))) {   1, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n
15 k null 16 k null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3
"number" v4)) {   2, 0 n 1 k null 4 k null 6 m 8 n 9 k 1629 11 k -383 12 n 13 m 14
n 15 k null 16 k null 18 k null, comp 0.5
```

```
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3 ""
v4)) {   3, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k null 16 k
null 18 k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3 i v4))
{   4, 0 n 1 k null 4 k null 6 m 8 n 9 m 11 m 12 n 13 m 14 n 15 k null 16 k null 18
k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3 j v4))
{   5, 0 n 1 k null 4 k null 6 i 8 n 9 i 11 i 12 n 13 i 14 n 15 k null 16 k null 18
k null, comp 0.6
  else if  i2 ((typeof i .4  ==  .2 "string" v3) &&  .1 (typeof j .4  ==  .3 m v4))
{   6, 0 n 1 i 4 i 6 i 8 n 9 i 11 i 12 n 13 i 14 n 15 i 16 i 18 i, comp 1.0

<.1 < == != <= >  >=  none <
    0 1k 2k 3k 4k 5k  0,  total 5 dead 5 miss 0

    if  i3 (i .5 .length m0  <  .1 j .5 .length m1) { <.1
    if  i3 (i .5 .length m0  ==  .1 j .5 .length m1) {   1, 1 m 4 k aa 15 m 16 k
abc 18 k ]h", comp 0.4
    if  i3 (i .5 .length m0  !=  .1 j .5 .length m1) {   2, 1 k abc 4 m 15 m 16 m
18 m, comp 0.8
    if  i3 (i .5 .length m0  <=  .1 j .5 .length m1) {   3, 1 m 4 k aa 15 m 16 m 18
m, comp 0.8
    if  i3 (i .5 .length m0  >  .1 j .5 .length m1) {   4, 1 k abc 4 m 15 m 16 k
abc 18 k ]h", comp 0.4
    if  i3 (i .5 .length m0  >=  .1 j .5 .length m1) {   5, 1 k abc 4 k aa 15 m 16
k abc 18 k ]h", comp 0.2

m0 i.length  none i.length
   0         0,  total 0 dead 0 miss 0


i.5 i j  m   none i
    0 1i 2i  0,  total 2 dead 0 miss 0 tin 2

    if  i3 (i .5 .length m0  <  .1 j .5 .length m1) { i.5
    if  i3 (j .5 .length m0  <  .1 j .5 .length m1) {   1, 1 i 4 i 15 i 16 i 18 i,
comp 1.0
    if  i3 (m .5 .length m0  <  .1 j .5 .length m1) {   2, 1 i 4 i 15 i 16 i 18 i,
comp 1.0

m1 j.length  none j.length
   0         0,  total 0 dead 0 miss 0


j.5 j i  m   none j
    0 1i 2i  0,  total 2 dead 0 miss 0 tin 2

    if  i3 (i .5 .length m0  <  .1 j .5 .length m1) { j.5
    if  i3 (i .5 .length m0  <  .1 i .5 .length m1) {   1, 1 i 4 i 15 i 16 i 18 i,
comp 1.0
    if  i3 (i .5 .length m0  <  .1 m .5 .length m1) {   2, 1 i 4 i 15 i 16 i 18 i,
comp 1.0

s5 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

      m .3  =  .5 i .6  s5; s5
    else {   1, 16 k 0 18 k 0, comp 0.0
```

```
m.3 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0

      m .3  =  .5 i .6  s5; m.3
      i .3  =  .5 i .6  s5;   1, 16 k 0 18 k 0, comp 0.0
      j .3  =  .5 i .6  s5;   2, 16 k 0 18 k 0, comp 0.0

i.6 i cEmpty deadOnEmpty number string    j  m   none i
    0 1k     2x          3k     4k     5k 6k 7i  0,  total 7 dead 6 miss 0 tin 1

      m .3  =  .5 i .6  s5; i.6
      m .3  =  .5 cEmpty(i .6 ) s5;   1, 16 k  18 m, comp 0.5
      m .3  =  .5 deadOnEmpty(i .6 ) s5;   2, 16 m 18 x, comp 0.5
      m .3  =  .5 "number" .6  s5;   3, 16 k number 18 k number, comp 0.0
      m .3  =  .5 "string" .6  s5;   4, 16 k string 18 k string, comp 0.0
      m .3  =  .5 "" .6  s5;   5, 16 k  18 m, comp 0.5
      m .3  =  .5 j .6  s5;   6, 16 k abc 18 k ]h", comp 0.0
      m .3  =  .5 m .6  s5;   7, 16 i 18 i, comp 1.0

s6 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

      m .4  =  .6 j .6  s6; s6
  }   1, 1 k 0 4 k 0 15 k 0, comp 0.0

m.4 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0

      m .4  =  .6 j .6  s6; m.4
      i .4  =  .6 j .6  s6;   1, 1 k 0 4 k 0 15 k 0, comp 0.0
      j .4  =  .6 j .6  s6;   2, 1 k 0 4 k 0 15 k 0, comp 0.0

j.6 j cEmpty deadOnEmpty number string    i  m   none j
    0 1k     2x          3k     4k     5k 6k 7i  0,  total 7 dead 6 miss 0 tin 1

      m .4  =  .6 j .6  s6; j.6
      m .4  =  .6 cEmpty(j .6 ) s6;   1, 1 k  4 k  15 m, comp 0.3
      m .4  =  .6 deadOnEmpty(j .6 ) s6;   2, 1 m 4 m 15 x, comp 0.7
      m .4  =  .6 "number" .6  s6;   3, 1 k number 4 k number 15 k number, comp 0.0
      m .4  =  .6 "string" .6  s6;   4, 1 k string 4 k string 15 k string, comp 0.0
      m .4  =  .6 "" .6  s6;   5, 1 k  4 k  15 m, comp 0.3
      m .4  =  .6 i .6  s6;   6, 1 k abc 4 k aa 15 m, comp 0.3
      m .4  =  .6 m .6  s6;   7, 1 i 4 i 15 i, comp 1.0

&&.2 && ||  none &&
     0  1m  0,  total 1 dead 0 miss 1

  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) { &&.2
  else if  i4 ((typeof i .7  ==  .4 "number" v5) ||  .2 (typeof j .7  ==  .5
"string" v6)) {   1, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0

==.4 == != <  <= >  >=  none ==
     0  1k 2k 3m 4k 5m  0,  total 5 dead 3 miss 2

  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) { ==.4
```

```
  else if  i4 ((typeof i .7  !=  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   1, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof i .7  <  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   2, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof i .7  <=  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   3, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof i .7  >  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   4, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof i .7  >=  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   5, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0

i.7 i c1 sub1 zpush c0 add1 neg abs negabs cfalse ctrue logneg cEmpty deadOnEmpty
number string     j   m    none i
    0 1m 2m   3x    4m 5m   6m  7m  8m     9k     10k   11k    12k    13x
14k    15k    16k 17k 18m  0,  total 18 dead 10 miss 8

  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) { i.7
  else if  i4 ((typeof c1(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   1, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof sub1(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   2, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof zpush(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   3, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 x, comp 0.9
  else if  i4 ((typeof c0(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   4, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof add1(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   5, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof  - (i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   6, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof Math.Abs(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==
.5 "string" v6)) {   7, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof  - (Math.Abs(i .7)) ==  .4 "number" v5) &&  .2 (typeof j .7
==  .5 "string" v6)) {   8, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0
  else if  i4 ((typeof cfalse(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   9, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof ctrue(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   10, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof !(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   11, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof cEmpty(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   12, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof deadOnEmpty(i .7) ==  .4 "number" v5) &&  .2 (typeof j .7
==  .5 "string" v6)) {   13, 0 m 6 m 8 m 9 m 11 x 12 m 13 m 14 m, comp 0.9
  else if  i4 ((typeof "number" .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   14, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof "string" .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   15, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof "" .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   16, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof j .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   17, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6
  else if  i4 ((typeof m .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) {   18, 0 m 6 m 8 m 9 m 11 m 12 m 13 m 14 m, comp 1.0

v5 number cEmpty string    i  j  m   none number
   0      1k     2k     3k 4m 5k 6i  0,  total 6 dead 4 miss 1 tin 1
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 
"string" v6)) { v5 
  else if  i4 ((typeof i .7  ==  .4 cEmpty("number" v5)) &&  .2 (typeof j .7  == 
.5 "string" v6)) {   1, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 
0.6 
  else if  i4 ((typeof i .7  ==  .4 "string" v5) &&  .2 (typeof j .7  ==  .5 
"string" v6)) {   2, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "" v5) &&  .2 (typeof j .7  ==  .5 "string" 
v6)) {   3, 0 k null 6 m 8 m 9 m 11 m 12 k null 13 m 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 i v5) &&  .2 (typeof j .7  ==  .5 "string" v6)) 
{   4, 0 i 6 m 8 i 9 m 11 m 12 i 13 m 14 i, comp 1.0 
  else if  i4 ((typeof i .7  ==  .4 j v5) &&  .2 (typeof j .7  ==  .5 "string" v6)) 
{   5, 0 k null 6 i 8 i 9 i 11 i 12 k null 13 i 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 m v5) &&  .2 (typeof j .7  ==  .5 "string" v6)) 
{   6, 0 i 6 i 8 i 9 i 11 i 12 i 13 i 14 i, comp 1.0 

==.5 == != <  <= >  >=  none == 
     0  1k 2k 3m 4k 5m  0,  total 5 dead 3 miss 2 

  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 
"string" v6)) { ==.5 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  !=  .5 
"string" v6)) {   1, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  <  .5 
"string" v6)) {   2, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  <=  .5 
"string" v6)) {   3, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  >  .5 
"string" v6)) {   4, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  >=  .5 
"string" v6)) {   5, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0 

j.7 j c0 c1 add1 sub1 neg abs negabs zpush ctrue cEmpty deadOnEmpty number string 
i   m    none j 
    0 1k 2k 3m   4k   5k  6k  7k     8m    9k    10m    11x         12m    13m    
14m 15k 16k  0,  total 16 dead 10 miss 6 

  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 
"string" v6)) { j.7 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof c0(j .7) ==  .5 
"string" v6)) {   1, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof c1(j .7) ==  .5 
"string" v6)) {   2, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof add1(j .7) ==  .5 
"string" v6)) {   3, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof sub1(j .7) ==  .5 
"string" v6)) {   4, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof  - (j .7) ==  .5 
"string" v6)) {   5, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof Math.Abs(j .7) == 
.5 "string" v6)) {   6, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 
0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof  - (Math.Abs(j .7)) 
==  .5 "string" v6)) {   7, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, 
comp 0.6 
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof zpush(j .7) ==  .5 
"string" v6)) {   8, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0 
```

```
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof ctrue(j .7) ==  .5
"string" v6)) {   9, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof cEmpty(j .7) ==  .5
"string" v6)) {   10, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof deadOnEmpty(j .7)
==  .5 "string" v6)) {   11, 0 x 6 n 8 n 9 n 11 n 12 x 13 n 14 m, comp 0.8
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof "number" .7  ==  .5
"string" v6)) {   12, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof "string" .7  ==  .5
"string" v6)) {   13, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof "" .7  ==  .5
"string" v6)) {   14, 0 m 6 n 8 n 9 n 11 n 12 m 13 n 14 m, comp 1.0
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof i .7  ==  .5
"string" v6)) {   15, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof m .7  ==  .5
"string" v6)) {   16, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6

v6 string cEmpty number    i  j  m   none string
   0      1k     2k     3k 4i 5k 6i  0,  total 6 dead 4 miss 0 tin 2

  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"string" v6)) { v6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
cEmpty("string" v6))) {   1, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null,
comp 0.6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5
"number" v6)) {   2, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 ""
v6)) {   3, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 i v6))
{   4, 0 i 6 n 8 n 9 n 11 n 12 i 13 n 14 i, comp 1.0
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 j v6))
{   5, 0 k null 6 n 8 n 9 n 11 n 12 k null 13 n 14 k null, comp 0.6
  else if  i4 ((typeof i .7  ==  .4 "number" v5) &&  .2 (typeof j .7  ==  .5 m v6))
{   6, 0 i 6 n 8 n 9 n 11 n 12 i 13 n 14 i, comp 1.0

<.2 < == != <= >  >=  none <
    0 1k 2k 3m 4k 5k  0,  total 5 dead 4 miss 1

    if  i5 (i .8  <  .2 (j .8  -  .0 0 v7)) { <.2
    if  i5 (i .8  ==  .2 (j .8  -  .0 0 v7)) {   1, 0 k  12 m 14 m, comp 0.7
    if  i5 (i .8  !=  .2 (j .8  -  .0 0 v7)) {   2, 0 m 12 k 4 14 k 0, comp 0.3
    if  i5 (i .8  <=  .2 (j .8  -  .0 0 v7)) {   3, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  >  .2 (j .8  -  .0 0 v7)) {   4, 0 k  12 k 4 14 m, comp 0.3
    if  i5 (i .8  >=  .2 (j .8  -  .0 0 v7)) {   5, 0 k  12 k 4 14 m, comp 0.3

i.8 i c1 sub1 zpush c0 add1 neg abs negabs j  m    none i
    0 1k 2m   3x    4k 5m   6k  7k  8k     9i 10k  0,  total 10 dead 7 miss 2 tin 1

    if  i5 (i .8  <  .2 (j .8  -  .0 0 v7)) { i.8
    if  i5 (c1(i .8) <  .2 (j .8  -  .0 0 v7)) {   1, 0 k  12 m 14 m, comp 0.7
    if  i5 (sub1(i .8) <  .2 (j .8  -  .0 0 v7)) {   2, 0 m 12 m 14 m, comp 1.0
    if  i5 (zpush(i .8) <  .2 (j .8  -  .0 0 v7)) {   3, 0 m 12 m 14 x, comp 0.7
    if  i5 (c0(i .8) <  .2 (j .8  -  .0 0 v7)) {   4, 0 k  12 m 14 m, comp 0.7
    if  i5 (add1(i .8) <  .2 (j .8  -  .0 0 v7)) {   5, 0 m 12 m 14 m, comp 1.0
    if  i5 ( - (i .8) <  .2 (j .8  -  .0 0 v7)) {   6, 0 k  12 k 4 14 m, comp 0.3
    if  i5 (Math.Abs(i .8) <  .2 (j .8  -  .0 0 v7)) {   7, 0 k  12 m 14 m, comp
0.7
```

```
    if  i5 ( - (Math.Abs(i .8)) <  .2 (j .8  -  .0 0 v7)) {   8, 0 m 12 k 4 14 m,
comp 0.7
    if  i5 (j .8  <  .2 (j .8  -  .0 0 v7)) {   9, 0 i 12 i 14 i, comp 1.0
    if  i5 (m .8  <  .2 (j .8  -  .0 0 v7)) {   10, 0 k  12 m 14 m, comp 0.7

-.0 - +  *  /  %   none -
    0 1m 2m 3k 4k  0,  total 4 dead 2 miss 2

    if  i5 (i .8  <  .2 (j .8  -  .0 0 v7)) { -.0
    if  i5 (i .8  <  .2 (j .8  +  .0 0 v7)) {   1, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (j .8  *  .0 0 v7)) {   2, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (j .8  /  .0 0 v7)) {   3, 0 k  12 m 14 m, comp 0.7
    if  i5 (i .8  <  .2 (j .8  %  .0 0 v7)) {   4, 0 k  12 m 14 m, comp 0.7

j.8 j cEmpty deadOnEmpty number string    i  m   none j
    0 1m     2x          3k     4k     5m 6i 7i  0,  total 7 dead 3 miss 2 tin 2

    if  i5 (i .8  <  .2 (j .8  -  .0 0 v7)) { j.8
    if  i5 (i .8  <  .2 (cEmpty(j .8) -  .0 0 v7)) {   1, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (deadOnEmpty(j .8) -  .0 0 v7)) {   2, 0 x 12 x 14 m, comp
0.3
    if  i5 (i .8  <  .2 ("number" .8  -  .0 0 v7)) {   3, 0 k  12 m 14 m, comp 0.7
    if  i5 (i .8  <  .2 ("string" .8  -  .0 0 v7)) {   4, 0 k  12 m 14 m, comp 0.7
    if  i5 (i .8  <  .2 ("" .8  -  .0 0 v7)) {   5, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (i .8  -  .0 0 v7)) {   6, 0 i 12 i 14 i, comp 1.0
    if  i5 (i .8  <  .2 (m .8  -  .0 0 v7)) {   7, 0 i 12 i 14 i, comp 1.0

v7 0 c1 sub1 zpush i  j  m   none 0
   0 1m 2m   3x    4m 5i 6m  0,  total 6 dead 1 miss 4 tin 1

    if  i5 (i .8  <  .2 (j .8  -  .0 0 v7)) { v7
    if  i5 (i .8  <  .2 (j .8  -  .0 c1(0 v7))) {   1, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (j .8  -  .0 sub1(0 v7))) {   2, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (j .8  -  .0 zpush(0 v7))) {   3, 0 x 12 x 14 x, comp 0.0
    if  i5 (i .8  <  .2 (j .8  -  .0 i v7)) {   4, 0 m 12 m 14 m, comp 1.0
    if  i5 (i .8  <  .2 (j .8  -  .0 j v7)) {   5, 0 i 12 i 14 i, comp 1.0
    if  i5 (i .8  <  .2 (j .8  -  .0 m v7)) {   6, 0 m 12 m 14 m, comp 1.0

s7 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

      m .5  =  .7 i .9  s7; s7
    else {   1, 0 k 0, comp 0.0

m.5 m i  j   none m
    0 1k 2i  0,  total 2 dead 1 miss 0 tin 1

      m .5  =  .7 i .9  s7; m.5
      i .5  =  .7 i .9  s7;   1, 0 k 0, comp 0.0
      j .5  =  .7 i .9  s7;   2, 0 i, comp 1.0

i.9 i c0 c1 add1 sub1 neg j  m   none i
    0 1k 2k 3k   4k   5k  6i 7k  0,  total 7 dead 6 miss 0 tin 1

      m .5  =  .7 i .9  s7; i.9
      m .5  =  .7 c0(i .9 ) s7;   1, 0 k 0, comp 0.0
      m .5  =  .7 c1(i .9 ) s7;   2, 0 k 1, comp 0.0
      m .5  =  .7 add1(i .9 ) s7;   3, 0 k -3, comp 0.0
      m .5  =  .7 sub1(i .9 ) s7;   4, 0 k -5, comp 0.0
      m .5  =  .7  - (i .9 ) s7;   5, 0 k 4, comp 0.0
      m .5  =  .7 j .9  s7;   6, 0 i, comp 1.0
      m .5  =  .7 m .9  s7;   7, 0 k 0, comp 0.0

s8 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

      m .6  =  .8 j .9  s8; s8
  }   1, 12 k 0 14 k 0, comp 0.0

m.6 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0

      m .6  =  .8 j .9  s8; m.6
      i .6  =  .8 j .9  s8;   1, 12 k 0 14 k 0, comp 0.0
      j .6  =  .8 j .9  s8;   2, 12 k 0 14 k 0, comp 0.0

j.9 j cEmpty deadOnEmpty number string    i  m   none j
    0 1k     2x          3k     4k     5k 6i 7i  0,  total 7 dead 5 miss 0 tin 2

      m .6  =  .8 j .9  s8; j.9
      m .6  =  .8 cEmpty(j .9 ) s8;   1, 12 m 14 k , comp 0.5
      m .6  =  .8 deadOnEmpty(j .9 ) s8;   2, 12 x 14 m, comp 0.5
      m .6  =  .8 "number" .9  s8;   3, 12 k number 14 k number, comp 0.0
      m .6  =  .8 "string" .9  s8;   4, 12 k string 14 k string, comp 0.0
      m .6  =  .8 "" .9  s8;   5, 12 m 14 k , comp 0.5
      m .6  =  .8 i .9  s8;   6, 12 i 14 i, comp 1.0
      m .6  =  .8 m .9  s8;   7, 12 i 14 i, comp 1.0

&&.3 && ||  none &&
     0  1m  0,  total 1 dead 0 miss 1

  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { &&.3
  else if  i6 ((typeof i .10  ==  .6 "string" v8) ||  .3 (typeof j .10  ==  .7
"number" v9)) {   1, 6 m 8 m 9 m 11 m 13 m, comp 1.0

==.6 == != <  <= >  >=  none ==
     0  1k 2k 3m 4k 5m  0,  total 5 dead 3 miss 2

  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { ==.6
  else if  i6 ((typeof i .10  !=  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   1, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  <  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   2, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  <=  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   3, 6 m 8 m 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  >  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   4, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  >=  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   5, 6 m 8 m 9 m 11 m 13 m, comp 1.0

i.10 i cfalse ctrue logneg cEmpty deadOnEmpty number string    j  m    none i
     0 1k     2k    3k     4m     5x          6m     7m     8m 9k 10k  0,  total 10
dead 6 miss 4
```

```
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { i.10
  else if  i6 ((typeof cfalse(i .10) ==  .6 "string" v8) &&  .3 (typeof j .10  ==
.7 "number" v9)) {   1, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof ctrue(i .10) ==  .6 "string" v8) &&  .3 (typeof j .10  ==
.7 "number" v9)) {   2, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof !(i .10) ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   3, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof cEmpty(i .10) ==  .6 "string" v8) &&  .3 (typeof j .10  ==
.7 "number" v9)) {   4, 6 m 8 m 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof deadOnEmpty(i .10) ==  .6 "string" v8) &&  .3 (typeof j .10
==  .7 "number" v9)) {   5, 6 m 8 m 9 m 11 x 13 m, comp 0.8
  else if  i6 ((typeof "number" .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==
.7 "number" v9)) {   6, 6 m 8 m 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof "string" .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==
.7 "number" v9)) {   7, 6 m 8 m 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof "" .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   8, 6 m 8 m 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof j .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   9, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof m .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   10, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2

v8 string cEmpty number    i  j  m   none string
   0      1k     2k     3k 4k 5i 6i  0,  total 6 dead 4 miss 0 tin 2

  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { v8
  else if  i6 ((typeof i .10  ==  .6 cEmpty("string" v8)) &&  .3 (typeof j .10  ==
.7 "number" v9)) {   1, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "number" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) {   2, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "" v8) &&  .3 (typeof j .10  ==  .7 "number"
v9)) {   3, 6 k null 8 m 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 i v8) &&  .3 (typeof j .10  ==  .7 "number"
v9)) {   4, 6 k null 8 i 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 j v8) &&  .3 (typeof j .10  ==  .7 "number"
v9)) {   5, 6 i 8 i 9 i 11 i 13 i, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 m v8) &&  .3 (typeof j .10  ==  .7 "number"
v9)) {   6, 6 i 8 i 9 i 11 i 13 i, comp 1.0

==.7 == != <  <= >  >=  none ==
     0  1k 2k 3m 4k 5m  0,  total 5 dead 3 miss 2

  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { ==.7
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  !=  .7
"number" v9)) {   1, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  <  .7
"number" v9)) {   2, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  <=  .7
"number" v9)) {   3, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  >  .7
"number" v9)) {   4, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  >=  .7
"number" v9)) {   5, 6 m 8 n 9 m 11 m 13 m, comp 1.0

j.10 j c0 c1 add1 sub1 neg abs negabs zpush ctrue i   m    none j
```

```
     0 1m 2m 3m   4m   5m  6m  7m     8m    9k    10k 11m  0,  total 11 dead 2 miss
9

  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { j.10
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof c0(j .10) ==  .7
"number" v9)) {   1, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof c1(j .10) ==  .7
"number" v9)) {   2, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof add1(j .10) ==  .7
"number" v9)) {   3, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof sub1(j .10) ==  .7
"number" v9)) {   4, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof  - (j .10) ==  .7
"number" v9)) {   5, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof Math.Abs(j .10) ==
.7 "number" v9)) {   6, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof  - (Math.Abs(j
.10)) ==  .7 "number" v9)) {   7, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof zpush(j .10) ==
.7 "number" v9)) {   8, 6 m 8 n 9 m 11 m 13 m, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof ctrue(j .10) ==
.7 "number" v9)) {   9, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof i .10  ==  .7
"number" v9)) {   10, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof m .10  ==  .7
"number" v9)) {   11, 6 m 8 n 9 m 11 m 13 m, comp 1.0

v9 number cEmpty string    i  j  m   none number
   0      1k     2k     3k 4k 5i 6i  0,  total 6 dead 4 miss 0 tin 2

  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"number" v9)) { v9
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
cEmpty("number" v9))) {   1, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7
"string" v9)) {   2, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7 ""
v9)) {   3, 6 k null 8 n 9 k null 11 k null 13 k null, comp 0.2
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7 i
v9)) {   4, 6 m 8 n 9 k null 11 k null 13 k null, comp 0.4
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7 j
v9)) {   5, 6 i 8 n 9 i 11 i 13 i, comp 1.0
  else if  i6 ((typeof i .10  ==  .6 "string" v8) &&  .3 (typeof j .10  ==  .7 m
v9)) {   6, 6 i 8 n 9 i 11 i 13 i, comp 1.0

<.3 < == != <= >  >=  none <
    0 1k 2k 3k 4k 5k  0,  total 5 dead 5 miss 0

    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) { <.3
    if  i7 (i .11 .length m2  ==  .3 (j .11  +  .0 "" v10).length m3) {   1, 6 m 9
k 1629 11 k -383 13 k JHw, comp 0.3
    if  i7 (i .11 .length m2  !=  .3 (j .11  +  .0 "" v10).length m3) {   2, 6 k
number 9 m 11 m 13 m, comp 0.8
    if  i7 (i .11 .length m2  <=  .3 (j .11  +  .0 "" v10).length m3) {   3, 6 m 9
m 11 m 13 k JHw, comp 0.8
    if  i7 (i .11 .length m2  >  .3 (j .11  +  .0 "" v10).length m3) {   4, 6 k
number 9 k 1629 11 k -383 13 m, comp 0.3
```

```
    if  i7 (i .11 .length m2  >=  .3 (j .11  +  .0 "" v10).length m3) {   5, 6 k
number 9 k 1629 11 k -383 13 k JHw, comp 0.0

m2 i.length  none i.length
   0         0,  total 0 dead 0 miss 0


i.11 i j  m   none i
     0 1i 2i  0,  total 2 dead 0 miss 0 tin 2

    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) { i.11
    if  i7 (j .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) {   1, 6 i 9 i
11 i 13 i, comp 1.0
    if  i7 (m .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) {   2, 6 i 9 i
11 i 13 i, comp 1.0

m3  none
(j .11  +  .0 "" v10).length
   (j


+.0 + -  *  /  %   none +
    0 1k 2k 3k 4k  0,  total 4 dead 4 miss 0

    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) { +.0
    if  i7 (i .11 .length m2  <  .3 (j .11  -  .0 "" v10).length m3) {   1, 6 m 9 k
1629 11 k -383 13 m, comp 0.5
    if  i7 (i .11 .length m2  <  .3 (j .11  *  .0 "" v10).length m3) {   2, 6 m 9 k
1629 11 k -383 13 m, comp 0.5
    if  i7 (i .11 .length m2  <  .3 (j .11  /  .0 "" v10).length m3) {   3, 6 m 9 k
1629 11 k -383 13 m, comp 0.5
    if  i7 (i .11 .length m2  <  .3 (j .11  %  .0 "" v10).length m3) {   4, 6 m 9 k
1629 11 k -383 13 m, comp 0.5

j.11 j c0 c1 add1 sub1 neg abs negabs zpush i  m    none j
     0 1k 2k 3m   4m   5k  6m  7k     8m    9i 10k  0,  total 10 dead 5 miss 4 tin
1

    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) { j.11
    if  i7 (i .11 .length m2  <  .3 (c0(j .11) +  .0 "" v10).length m3) {   1, 6 m
9 k 1629 11 m 13 m, comp 0.8
    if  i7 (i .11 .length m2  <  .3 (c1(j .11) +  .0 "" v10).length m3) {   2, 6 m
9 k 1629 11 m 13 m, comp 0.8
    if  i7 (i .11 .length m2  <  .3 (add1(j .11) +  .0 "" v10).length m3) {   3, 6
m 9 m 11 m 13 m, comp 1.0
    if  i7 (i .11 .length m2  <  .3 (sub1(j .11) +  .0 "" v10).length m3) {   4, 6
m 9 m 11 m 13 m, comp 1.0
    if  i7 (i .11 .length m2  <  .3 ( - (j .11) +  .0 "" v10).length m3) {   5, 6 m
9 m 11 m 13 k JHw, comp 0.8
    if  i7 (i .11 .length m2  <  .3 (Math.Abs(j .11) +  .0 "" v10).length m3) {
6, 6 m 9 m 11 m 13 m, comp 1.0
    if  i7 (i .11 .length m2  <  .3 ( - (Math.Abs(j .11)) +  .0 "" v10).length m3)
{   7, 6 m 9 m 11 m 13 k JHw, comp 0.8
    if  i7 (i .11 .length m2  <  .3 (zpush(j .11) +  .0 "" v10).length m3) {   8, 6
m 9 m 11 m 13 m, comp 1.0
    if  i7 (i .11 .length m2  <  .3 (i .11  +  .0 "" v10).length m3) {   9, 6 i 9 i
11 i 13 i, comp 1.0
```

```
    if  i7 (i .11 .length m2  <  .3 (m .11  +  .0 "" v10).length m3) {   10, 6 m 9
k 1629 11 m 13 m, comp 0.8

v10   deadOnEmpty number string i  j  m   none
    0 1x          2k     3k     4k 5i 6i  0,  total 6 dead 4 miss 0 tin 2

    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "" v10).length m3) { v10
    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 deadOnEmpty("" v10)).length m3) {
1, 6 x 9 x 11 x 13 x, comp 0.0
    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "number" v10).length m3) {   2, 6
k number 9 m 11 m 13 k JHw, comp 0.5
    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 "string" v10).length m3) {   3, 6
k number 9 m 11 m 13 k JHw, comp 0.5
    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 i v10).length m3) {   4, 6 k
number 9 m 11 m 13 k JHw, comp 0.5
    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 j v10).length m3) {   5, 6 i 9 i
11 i 13 i, comp 1.0
    if  i7 (i .11 .length m2  <  .3 (j .11  +  .0 m v10).length m3) {   6, 6 i 9 i
11 i 13 i, comp 1.0

s9 expn deleteExpn  none expn
   0    1k          0,  total 1 dead 1 miss 0

      m .7  =  .9 i .12  s9; s9
    else {   1, 9 k 0 11 k 0, comp 0.0

m.7 m i  j   none m
    0 1k 2k  0,  total 2 dead 2 miss 0

      m .7  =  .9 i .12  s9; m.7
      i .7  =  .9 i .12  s9;   1, 9 k 0 11 k 0, comp 0.0
      j .7  =  .9 i .12  s9;   2, 9 k 0 11 k 0, comp 0.0

i.12 i cEmpty deadOnEmpty number string    j  m   none i
     0 1k     2x          3k     4k     5k 6i 7i  0,  total 7 dead 5 miss 0 tin 2

      m .7  =  .9 i .12  s9; i.12
      m .7  =  .9 cEmpty(i .12 ) s9;   1, 9 k  11 m, comp 0.5
      m .7  =  .9 deadOnEmpty(i .12 ) s9;   2, 9 m 11 x, comp 0.5
      m .7  =  .9 "number" .12  s9;   3, 9 k number 11 k number, comp 0.0
      m .7  =  .9 "string" .12  s9;   4, 9 k string 11 k string, comp 0.0
      m .7  =  .9 "" .12  s9;   5, 9 k  11 m, comp 0.5
      m .7  =  .9 j .12  s9;   6, 9 i 11 i, comp 1.0
      m .7  =  .9 m .12  s9;   7, 9 i 11 i, comp 1.0

s10 expn deleteExpn  none expn
    0    1k          0,  total 1 dead 1 miss 0

      m .8  =  .10 j .12  s10; s10
  }   1, 6 k 0 13 k 0, comp 0.0

m.8 m i  j   none m
    0 1i 2k  0,  total 2 dead 1 miss 0 tin 1

      m .8  =  .10 j .12  s10; m.8
      i .8  =  .10 j .12  s10;   1, 6 i 13 i, comp 1.0
      j .8  =  .10 j .12  s10;   2, 6 k 0 13 k 0, comp 0.0
```

```
j.12 j c0 c1 add1 sub1 neg i  m   none j
     0 1k 2k 3k   4k   5k  6i 7k  0,  total 7 dead 6 miss 0 tin 1

      m .8  =  .10 j .12  s10; j.12
      m .8  =  .10 c0(j .12 ) s10;   1, 6 k 0 13 k 0, comp 0.0
      m .8  =  .10 c1(j .12 ) s10;   2, 6 k 1 13 k 1, comp 0.0
      m .8  =  .10 add1(j .12 ) s10;   3, 6 k 5 13 k 564, comp 0.0
      m .8  =  .10 sub1(j .12 ) s10;   4, 6 k 3 13 k 562, comp 0.0
      m .8  =  .10  - (j .12 ) s10;   5, 6 k -4 13 k -563, comp 0.0
      m .8  =  .10 i .12  s10;   6, 6 i 13 i, comp 1.0
      m .8  =  .10 m .12  s10;   7, 6 k 0 13 k 0, comp 0.0

s11 expn deleteExpn  none expn
    0    1k          0,  total 1 dead 1 miss 0

    m .9  =  .11 null  v11  s11; s11
  var r;   1, 8 k 0, comp 0.0

m.9 m i  j   none m
    0 1i 2i  0,  total 2 dead 0 miss 0 tin 2

    m .9  =  .11 null  v11  s11; m.9
    i .9  =  .11 null  v11  s11;   1, 8 i, comp 1.0
    j .9  =  .11 null  v11  s11;   2, 8 i, comp 1.0

v11    none
    0  0,  total 0 dead 0 miss 0


s12 expn deleteExpn  none expn
    0    1k          0,  total 1 dead 1 miss 0

  r .0  =  .12 m .10  s12; s12
  TESCRIPTAClass.TESCRIPTA.defineOutput(r, 9);   1, 0 k  1 k  2 k  3 k  4 k  5 k  6
k  7 k  8 k  9 k  10 k  11 k  12 k  13 k  14 k  15 k  16 k  17 k  18 k , comp 0.0

r.0 r i  j  m   none r
    0 1k 2k 3k  0,  total 3 dead 3 miss 0

  r .0  =  .12 m .10  s12; r.0
  i .0  =  .12 m .10  s12;   1, 0 k  1 k  2 k  3 k  4 k  5 k  6 i 7 k  8 i 9 k  10
k  11 k  12 i 13 i 14 i 15 k  16 k  17 k  18 k , comp 0.3
  j .0  =  .12 m .10  s12;   2, 0 i 1 k  2 k  3 k  4 k  5 k  6 k  7 k  8 i 9 i 10 k
11 i 12 k  13 k  14 k  15 k  16 k  17 k  18 k , comp 0.2
  m .0  =  .12 m .10  s12;   3, 0 k  1 k  2 k  3 k  4 k  5 k  6 k  7 k  8 k  9 k
10 k  11 k  12 k  13 k  14 k  15 k  16 k  17 k  18 k , comp 0.0

m.10 m c1 sub1 zpush c0 add1 neg abs negabs cEmpty deadOnEmpty number string     i
j   r    none m
     0 1k 2k   3x    4k 5k   6k  7k  8k     9k     10x         11k    12k    13k
14k 15k 16i  0,  total 16 dead 15 miss 0 tin 1

  r .0  =  .12 m .10  s12; m.10
  r .0  =  .12 c1(m .10 ) s12;   1, 0 k 1 1 i 2 k 1 3 k 1 4 i 5 k 1 6 k 1 7 k 1 8 i
9 i 10 k 1 11 i 12 i 13 k 1 14 i 15 i 16 i 17 k 1 18 i, comp 0.5
  r .0  =  .12 sub1(m .10 ) s12;   2, 0 k -5 1 i 2 k 9 3 k -1 4 i 5 k 3 6 k 3 7 k -
1 8 i 9 i 10 k -384 11 i 12 i 13 k 562 14 i 15 i 16 i 17 k -45 18 i, comp 0.5
```

```
  r .0  =  .12 zpush(m .10 ) s12;   3, 0 m 1 i 2 m 3 x 4 i 5 m 6 m 7 x 8 i 9 i 10 m
11 i 12 i 13 m 14 i 15 i 16 i 17 m 18 i, comp 0.9
  r .0  =  .12 c0(m .10 ) s12;   4, 0 k 0 1 i 2 k 0 3 m 4 i 5 k 0 6 k 0 7 m 8 i 9 i
10 k 0 11 i 12 i 13 k 0 14 i 15 i 16 i 17 k 0 18 i, comp 0.6
  r .0  =  .12 add1(m .10 ) s12;   5, 0 k -3 1 i 2 k 11 3 k 1 4 i 5 k 5 6 k 5 7 k 1
8 i 9 i 10 k -382 11 i 12 i 13 k 564 14 i 15 i 16 i 17 k -43 18 i, comp 0.5
  r .0  =  .12  - (m .10 ) s12;   6, 0 k 4 1 i 2 k -10 3 m 4 i 5 k -4 6 k -4 7 m 8
i 9 i 10 k 383 11 i 12 i 13 k -563 14 i 15 i 16 i 17 k 44 18 i, comp 0.6
  r .0  =  .12 Math.Abs(m .10 ) s12;   7, 0 k 4 1 i 2 m 3 m 4 i 5 m 6 m 7 m 8 i 9 i
10 k 383 11 i 12 i 13 m 14 i 15 i 16 i 17 k 44 18 i, comp 0.8
  r .0  =  .12  - (Math.Abs(m .10)) s12;   8, 0 m 1 i 2 k -10 3 m 4 i 5 k -4 6 k -4
7 m 8 i 9 i 10 m 11 i 12 i 13 k -563 14 i 15 i 16 i 17 m 18 i, comp 0.8
  r .0  =  .12 cEmpty(m .10 ) s12;   9, 0 i 1 k  2 i 3 i 4 k  5 i 6 i 7 i 8 i 9 k
10 i 11 m 12 m 13 i 14 k  15 m 16 k  17 i 18 m, comp 0.7
  r .0  =  .12 deadOnEmpty(m .10 ) s12;   10, 0 i 1 m 2 i 3 i 4 m 5 i 6 i 7 i 8 i 9
m 10 i 11 x 12 x 13 i 14 m 15 x 16 m 17 i 18 x, comp 0.8
  r .0  =  .12 "number" .10  s12;   11, 0 i 1 k number 2 i 3 i 4 k number 5 i 6 i 7
i 8 i 9 k number 10 i 11 k number 12 k number 13 i 14 k number 15 k number 16 k
number 17 i 18 k number, comp 0.5
  r .0  =  .12 "string" .10  s12;   12, 0 i 1 k string 2 i 3 i 4 k string 5 i 6 i 7
i 8 i 9 k string 10 i 11 k string 12 k string 13 i 14 k string 15 k string 16 k
string 17 i 18 k string, comp 0.5
  r .0  =  .12 "" .10  s12;   13, 0 i 1 k  2 i 3 i 4 k  5 i 6 i 7 i 8 i 9 k  10 i
11 m 12 m 13 i 14 k  15 m 16 k  17 i 18 m, comp 0.7
  r .0  =  .12 i .10  s12;   14, 0 m 1 k abc 2 m 3 k 5 4 k aa 5 m 6 i 7 m 8 i 9 m
10 k -33 11 m 12 i 13 i 14 i 15 m 16 m 17 m 18 m, comp 0.8
  r .0  =  .12 j .10  s12;   15, 0 i 1 m 2 m 3 m 4 m 5 k 5 6 m 7 k 1 8 i 9 i 10 m
11 i 12 m 13 m 14 m 15 m 16 k abc 17 k -40 18 k ]h", comp 0.7
  r .0  =  .12 r .10  s12;   16, 0 i 1 i 2 i 3 i 4 i 5 i 6 i 7 i 8 i 9 i 10 i 11 i
12 i 13 i 14 i 15 i 16 i 17 i 18 i, comp 1.0
```