A QUANTITATIVE ANALYSIS OF WHETHER UNIT TESTING OBVIATES STATIC TYPE CHECKING FOR ERROR DETECTION

A Project

Presented

to the Faculty of

California State University, Chico

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Spring 2011

A QUANTITATIVE ANALYSIS OF WHETHER UNIT TESTING OBVIATES STATIC TYPE CHECKING FOR ERROR DETECTION

A Project

by

Evan R. Farrer

Spring 2011

APPROVED BY THE DEAN OF GRADUATE STUDIES
AND VICE PROVOST FOR RESEARCH:

_____
Katie Milo, Ed.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

_____
Abdel-Moaty Fayek
Graduate Coordinator

_____
Anne Keuneke, PhD, Chair

_____
Abdel-Moaty Fayek

PUBLICATION RIGHTS

DEDICATION

For Emily, who has tirelessly maintained the household while I've frolicked in academia. I'll fix the sprinklers now; I promise.

ACKNOWLEDGMENTS

A special thanks must be given to my Graduate Advisory Committee members, Dr. Anne Keuneke and Professor Abdel-Moaty Fayek, who provided constant feedback, encouragement and correction. Their advice not only strengthened this research and paper, but also taught me valuable lessons, which will aid me in my future endeavors.

I must also thank my co-workers at Applied Signal Technology, Inc. who provided early encouragement and feedback as I was exploring and developing ideas on this research.

Finally, I must thank my wife Emily, who not only endured constant ramblings about a topic she did not care to understand, but she also acted as my editor. She corrected countless grammatical and punctuation errors. Any occurrences of sentences in this paper that are grammatically correct and have correct punctuation are because of her diligence. All errors in this paper are mine and mine alone.

TABLE OF CONTENTS

LIST OF FIGURES

PAGE

FIGURE

PAGE

FIGURE

ABSTRACT

A QUANTITATIVE ANALYSIS OF WHETHER
UNIT TESTING OBVIATES STATIC TYPE
CHECKING FOR ERROR DETECTION

by

© Evan R. Farrer 2011

Master of Science in Computer Science

California State University, Chico

Spring 2011

Unit testing and static type checking are tools for ensuring defect free software. Unit testing is the practice of writing code to test individual units of a piece of software. By validating each unit of software, defects can be discovered during development. Static type checking is performed by a type checker that automatically validates the correct typing of expressions and statements at compile time. By validating correct typing, many defects can be discovered during development. Static typing also limits the expressiveness of a programming language in that it will reject some programs which are ill-typed, but which are free of defects.

Many proponents of unit testing claim that static type checking is an insufficient mechanism for ensuring defect free software; and therefore, unit testing is still required if static type checking is utilized. They also assert that once unit testing is

utilized, static type checking is no longer needed for defect detection, and so it should be eliminated.

The goal of this research is to explore whether unit testing does in fact obviate static type checking in real world examples of unit tested software.

CHAPTER I

INTRODUCTION

Background

A major concern for computer programmers is ensuring that software they create is free from defects. Common tools for ensuring defect free software are unit testing [1] and static type checking [2].

Unit testing is a process for testing individual units of code to ensure that their behavior is correct. The first step in the process of unit testing is to identify a unit of code. A unit of code could be a function, a method of an object, or a small class. Then one or more tests are written for the identified unit. Each test will verify that for some predefined input, the actual output corresponds to the expected output. It is unlikely that the tests for a particular unit will validate all possible inputs and their corresponding outputs. The tests, however, at a minimum should test the edge cases. For example, if one wanted to write unit tests for a function that performs multiplication on two integers, it would be impractical to test every possible integer input combination and verify the correct output. It would, therefore, be advisable to test the edge cases for multiplication such as combinations of 0, 1, large numbers, and negative numbers. See Figure 1 for an example of unit testing for the multiply function.

Once sufficient tests have been written for a given unit, the process is repeated for another identified unit. A common goal when writing unit tests is to achieve full code

```
1  #!/usr/bin/python
2
3  import unittest,random,sys
4
5  def multiply(a,b):
6      """ Multiply two numbers together. """
7      return a*b
8
9  class TestMultiply(unittest.TestCase):
10
11     def setUp(self):
12         random.seed()
13
14     def rand(self):
15         return random.randint(0, sys.maxint)
16
17     def test_multiply(self):
18         """ Test multiply function """
19
20         # 0 multiplied with any number is still 0
21         self.assertTrue(0==multiply(0,0))
22         self.assertTrue(0==multiply(0,1))
23         self.assertTrue(0==multiply(0,self.rand()))
24
25         # 1 multiplied with any number is that number
26         self.assertTrue(0==multiply(1,0))
27         self.assertTrue(1==multiply(1,1))
28         a = self.rand()
29         self.assertTrue(a==multiply(1,a))
30         self.assertTrue(-a==multiply(1,-a))
31
32         # Test negative number rules
33         p = self.rand()
34         n0 = -self.rand()
35         n1 = -self.rand()
36         n2 = -self.rand()
37         self.assertTrue(0>=multiply(p, n0))
38         self.assertTrue(0<=multiply(n1,n2))
39
40         # a*b == b*a
41         b = self.rand()
42         self.assertTrue(multiply(a,b)==multiply(b,a))
43
44         # A hard-coded tests
45         self.assertTrue(99==multiply(11, 9));
46
47 if __name__ == '__main__':
48     unittest.main()
```

Fig. 1. Example of unit testing the multiply function

coverage. Full code coverage indicates that the unit tests will execute each line of the program's source code at least once. Unit tests are executed to discover any regression defects whenever the software has been modified. With test-first development, the unit

tests are implemented before the unit of code that they test is written. This helps ensure that the unit tests are complete and that the behavior of the unit of code has been fully specified before it is written. A major benefit of unit testing is catching defects while the software is being developed and detecting errors that may be introduced by later enhancements to the software.

The second tool, static type checking, automatically validates the correct typing of expressions and statements at compile-time. Software that complies with the rules of a given type system are called well-typed. Software that fails to comply with the type system rules are called ill-typed. Ill-typed software is rejected by the type checker and will not be compiled or executed. Static type checking is beneficial in that it can detect at compile-time many errors that would be manifest at run-time. This enables the programmer to fix the errors before the software fails at run-time.

Static type checking does have its limitations. One such limitation is that it can sometimes reject ill-typed software that would never fail at run-time. This limitation may force programmers to avoid some programming constructs or have to rewrite portions of a program for the sole purpose of appeasing the overly restrictive type checker. Figure 2 shows an example of a type error that would not fail at run-time.

An alternative type checking approach is dynamic type checking. With dynamic type checking, all type checks happen at run-time. Run-time type checking ensures that no programs are unnecessarily rejected, but it also forgoes the safety of compile-time type checking. By omitting compile-time type checking many type errors will manifest themselves at run-time instead of at compile-time [2].

```
1  struct _bar
2  {
3    int x;
4  } bar;
5
6  int foo(int a)
7  {
8    /* The "b" variable is guaranteed to be
9       assigned an even number */
10   int b = 2*a;
11
12   /* If the "b" variable is even then line
13      #17 will be executed */
14   if (0 == b % 2)
15   {
16     /* The next line will always be executed */
17     return b;
18   }
19   else
20   {
21     /* The C type checker will report a
22      * type error on the following line
23      * because it is returning a C structure
24      * instead of an integer, but this line
25      * will never be executed. */
26     return bar;
27   }
28 }
```

Fig. 2. An example of a benign type error

Problem Statement

Because some error detection can be done by both unit testing and static type checking, some proponents of dynamic type checking claim that static type checking is not needed [3]. The rationale of this argument is based on the observation that static type checking alone is insufficient to detect all errors that could be found with unit testing. For example, a function that is intended to multiply two integers, but instead performs addition on the integers will be well-typed even though the results are incorrect. Since static type checking is insufficient to validate program correctness, unit testing is still needed. Once unit testing has been employed, dynamic typing advocates claim that the

static type checking is no longer needed because unit testing will implicitly validate that the software is well-typed [3].

For instance, by writing unit tests to validate the output of a function that multiplies two integers, one has implicitly validated that the output of the multiply function is an integer. The act of unit testing statically typed software results in two separate and redundant mechanisms for ensuring that the software is well-typed. Because static type checking both rejects software that would never cause a run-time error and provides insufficient error detection, dynamic typing proponents argue it makes sense to eliminate the static type checking and rely solely on unit testing [3].

A counter claim to the above argument could be made by advocates of static type checking that some unit tests only validate whether the unit is well-typed. This validation can be performed automatically by static type checking: programmers would not need to write as many unit tests if static type checking is utilized. Additionally, in practice, full code coverage may not always be achieved with unit testing [1]. By utilizing static type checking, some errors may be detected in the portions of the software that are not covered by comprehensive unit tests.

Purpose

The purpose of this project is to determine some of the costs and benefits of applying static type checking to unit tested software. This may aid developers in determining whether they should utilize static type checking with their software projects. It may also aid programming language designers in determining whether adding static type checking to a programming language would be beneficial. In order to measure these

factors, this project aims to answer the following questions about real unit tested software written in a dynamically typed programming language.

Do unit tests in practice negate the error detection benefits of static type checking?: This can be answered by verifying that dynamically typed, unit tested software is free from type errors.

Do programmers frequently write unit tests that would not be needed if static type checking was applied?: This would indicate that programmers are manually type checking portions of their programs that could be automatically type checked with static typing.

Do programmers commonly use programming constructs which would be rejected by static type checking but would not result in a run-time failure?: If dynamic programming constructs are commonly used, then it may not be possible to statically type check these programs without restricting the programming languages' expressiveness.

The answers to these questions help illuminate some of the costs and benefits of static type checking unit tested software for the purpose of error detection.

Limitations

It would be possible for programmers to write unit tests that would catch every error that could by caught by a type checker. This could be accomplished writing unit tests that implicitly or explicitly validate the type safety of every statement and expression in the software. It is likely, though, that real-world examples of unit testing lack this degree of thoroughness. This study is therefore limited to examining real-world examples of unit tested software to see whether type errors exist after unit testing.

The software chosen for examination included only projects with less than 2000 source lines of code. This restriction enabled a greater number of projects to be examined. It is unknown whether the results of this study would vary if larger projects were examined. It is hoped that in the future other researchers will repeat this study on larger software projects.

There are other benefits to unit testing beyond the ability to detect errors in software, such as better API design. There are also other benefits to static type checking beyond their ability to detect errors in software, such as more efficient execution. While those benefits are important they fall outside the scope of this study. For the remainder of this paper, any discussions of the benefits of unit testing and static type checking will refer solely to their ability to detect software errors.

CHAPTER II

LITERATURE REVIEW

There are several studies on the ability of static type checking to detect and, thereby, reduce errors in software. Gannon [4], Hanenberg [5], Prechelt, and Tichy [6] each conducted experiments where the participants were asked to solve a programming assignment in dynamically and statically typed programming languages. In each experiment, the resulting programs were analyzed for defects. Hanenberg's experiment involved having participants write programs in one of two programming languages that were identical except one was statically typed and required explicit type declarations. Hanenberg concluded that there was no significant reduction in defects in the statically typed implementations over the dynamically typed implementations. Gannon's experiment compared implementations in two similar but distinct programming languages. One language was statically typed and included some higher level abstractions, while the other was type-less. Gannon discovered a reduced defect count in the statically typed implementation. Prechelt and Tichy's experiment was to have participants write programs that interacted with a complicated API in either ANSI C where the compiler type checks function interfaces or K&R C where it does not. They concluded that there was a reduction in defects when using static type checking when interacting with an unfamiliar API.

There also exist several papers describing efforts to add static typing to dynamically typed programming languages. Cannon's [7] work showed that it may not be feasible to apply static typing to some dynamically typed programming languages for the purpose of improving performance without sacrificing language flexibility. Others such as Chen et al. [8], Furr, Foster, and Hicks [9], Hamlet [10], Henglein and Rehof [11] show that it is possible to apply static type checking for the purpose of error detection to dynamically typed languages. They applied static type checking by utilizing a combination of annotations, type inference [12], or by programmatically translating programs from a dynamically typed programming language (Scheme) to a statically typed programming language (ML).

There is also a good deal of research on the benefits of unit testing. Ellims, Bridges, and Ince [13] measure the effects of applying unit testing to three distinct automotive applications. In all three cases, it was determined that unit testing discovered defects that were not found through other testing means. Their study provides valuable insight into the benefit of unit testing real-world software.

Madeyski [14], Muller, and Hagner [15] performed experiments to determine whether different development practices would improve the effectiveness of unit testing in detecting errors. Madeyski's experiment was with pair programming, where Muller and Hagner focused on test-first development. The researchers concluded that neither test-first development nor pair programming positively affected the ability of unit tests to detect program errors.

Simons and Thomson [16] discuss the proper way of measuring the effectiveness of unit testing. They argue that neither path and branch coverage nor the automation of generating unit tests are effective measurements. They assert that these measurements sidestep the core issue which is whether the unit tests properly test for correct behavior. They suggest that mutation testing is better because it tests whether random changes to the code are detected by the unit tests.

The benefits of static type checking and unit testing have been thoroughly researched in isolation. A lack of published materials on whether there is any benefit to static type checking when unit testing is utilized indicates that this topic has not been systematically studied.

CHAPTER III

METHODOLOGY

The basic process for determining the costs and benefits of applying static type checking to unit tested software was to first find examples of unit tested software written in a dynamically typed programming language; second, translate the software from the dynamically typed programming language to a statically typed programming language; and third, note any defects discovered by the static type checker during the translation process.

In order to simplify the translation process, it was decided that all software projects selected for study should be limited to a single dynamically typed programming language. Programs would then be translated into a single statically typed programming language.

The criteria for choosing the dynamically typed programming language of the software projects to study were:

- The language should be dynamically typed
- The language should have strong support for and a strong culture of unit testing
- There should be a large corpus of open-source software freely available for study
- The language should be well known and considered a good language among unit-testing and dynamic typing proponents

There are several programming languages that satisfy the above criteria; however, Python [17] was chosen over the other languages due to the author's familiarity.

The criteria for choosing the statically typed programming language were as follows:

- The language should be statically typed
- The language should be available on the same platforms as the previously selected Python programming language
- The language should be strongly typed
- The language should be popular and considered a good language among static typing proponents

Haskell [18] was chosen as a language that satisfies the above criteria.

The Python software projects for this study were located by searching on the Bitbucket [19] and the Google Project Hosting [20] source code hosting websites. These sites were used because they provide a wide selection of open source Python software projects. Individual projects on these sites were located by searching for "pure python" and "python libraries" using each site's built in search capabilities. The "pure python" search term was used to try to eliminate Python projects that incorporated C or C++ code along with the Python code in the software. Since the purpose of this experiment was to test dynamically typed programming languages, testing a project that included code from the statically typed C or C++ programming languages could taint the results. The "python libraries" search term was used with the assumption that software libraries would be

more likely to have comprehensive unit tests than software applications. Both of these search terms resulted in several pages of matching projects on each site. From the returned search results, individual software projects were reviewed from randomly selected pages. The project source code was downloaded for the software projects that appeared to have unit tests and were written in pure Python. After the source code was downloaded, it was further analyzed using the cloc [21] utility. The cloc [21] utility is designed to count source code lines of code of a software project and to report which programming languages are used in the software. The cloc [21] utility helped verify that the selected software projects were written completely in Python and that each contained fewer than 2000 lines of code. Finally, the projects source code was manually examined to see if the project utilized some form of unit testing. When a project was found that passed the above tests, the translation process began.

The translation process was the most time consuming and challenging aspect of this study. Great care had to be taken to ensure that the translated software accurately modeled the semantics of the original software. Ensuring an accurate translation was especially challenging due to the use of Python and Haskell as the respective dynamically typed and statically typed programming languages.

Python is predominantly an object-oriented programming language, while Haskell is a purely functional programming language. Due to the different paradigms, the style of programming varies greatly between these two programming languages. Despite these differences, every effort was made during the translation process to not only completely preserve program semantics, but to also preserve as much syntactic similarity

as possible. Maintaining syntactic similarity was important for ensuring that a direct translation was achieved and that type errors were not accidentally introduced or removed due to unnecessary deviations from the original software. The syntactic similarities in the translation may also facilitate future audits by researchers who want to validate the results of this study.

The first challenge that was encountered when translating Python code to Haskell was how to represent a Python class in Haskell. A simple solution for representing a class is to define a new data type that contains fields for each of the Python classes' data members. Haskell allows developers to define new data types using the `data` keyword which defines a new type and also a value constructor for creating values of that type. Values created with the value constructor represent the Python objects in the Haskell translation. The Python class' methods were defined by writing Haskell functions that took a value of the defined data type as the first argument. See Figures 3 and 4 for an example of a simple Python class and the respective Haskell representation.

Python's class inheritance was simulated in Haskell by defining a single data type with multiple value constructors. Each value constructor creates a value with distinct fields, but the values created by each value constructor all have the same type. Each Haskell value constructor contains fields for either the base class' or a subclass' data members.

When a Python subclass is defined base class methods can be overridden. The decision on whether to call the base class variant or the subclass variant of the methods is determined at run-time via dynamic dispatch. This process is simulated in Haskell by

```
1  #!/usr/bin/python
2
3  # A simple class in Python
4  class car():
5    # The __init__ method
6    def __init__(self, color):
7      self.color = color
8
9    # The color method
10   def color(self):
11     return self.color
12
13   # The mpg method
14   def mpg(self):
15     return 45
16
17 # Construct a car object then call the mpg method.
18 if __name__ == '__main__':
19     c = car('red')
20     # Note that car.mpg(c) is the same as c.mpg()
21     print car.mpg(c)
```

Fig. 3. A simple Python class

```
1  #!/usr/bin/runghc
2
3  -- A simple class in Haskell
4  data Car = Car String
5
6  -- Function for constructing a new car
7  -- This takes the place of the __init__ method
8  car color = Car color
9
10 -- The color function
11 color (Car color) = color
12
13 -- The mpg function
14 mpg (Car _) = 45
15
16 -- Construct a car object then call the mpg method.
17 main =
18     let c = car "red"
19     in print (mpg c)
```

Fig. 4. A translation of the Python class to Haskell

defining a single function which contains the functionality of both the base class and subclass methods. Which functionality is executed is determined at run-time by using pattern matching to select the desired functionality based on which value constructor was used to create the object. See Figures 5 and 6 for an example of Python inheritance and method overriding along with the respective Haskell translation.

```
1  #!/usr/bin/python
2
3  # A simple class in Python
4  class car():
5    # The __init__ method
6    def __init__(self, color):
7      self.color = color
8
9    # The color method
10   def color(self):
11     return self.color
12
13   # The mpg method
14   def mpg(self):
15     return 45
16
17 # Basic inheritance
18 class truck(car):
19   # The overridden mpg method
20   def mpg(self):
21     return 14
22
23 # Construct a truck object then call the mpg method.
24 if __name__ == '__main__':
25     t = truck('red')
26     print truck.mpg(t)
```

Fig. 5. An example of simple Python inheritance and method overriding

Another challenge in translating from Python to Haskell is that Python allows for mutation where Haskell is a purely functional language and, therefore, does not allow its variables to be mutated. This limitation was most frequently encountered when translating Python methods that modify the values of their data members. Mutation of an object's data members was simulated in Haskell by having the translated method return a

```
1  #!/usr/bin/runghc
2
3  -- A simple class in Haskell
4  data Car = Car String
5           | Truck String -- Basic inheritance
6
7  -- Function for constructing a new car
8  -- This takes the place of the __init__ method
9  car color = Car color
10
11 -- The color function
12 color (Car color) = color
13
14 -- The mpg function
15 -- Both the truck and car variants are handled here
16 mpg c = case c of
17            (Car _) -> 45
18            (Truck _) -> 14
19
20 -- Function for constructing a new truck
21 -- This takes the place of the __init__ method
22 truck color = Truck color
23
24 -- Construct a car object then call the mpg method.
25 main =
26     let t = truck "red"
27     in print (mpg t)
```

Fig. 6. A translation of the Python classes and methods to Haskell

new copy of the object with updated data member values. See Figures 7 and 8 for an example of a Python method that mutates its data members and the respective Haskell translation.

In the above example the Python decrement method only returns the updated integer value. The Haskell version returns a tuple containing the updated integer value along with a value that represents the modified `counter` object. While at first it could seem counterproductive to modify the type signature of a method when the goal is to detect type errors, in practice it was simple to adapt the calling code to accommodate the additional return value. The change to the method type signature did not seem to hinder the detection of type errors.

```
1  #!/usr/bin/python
2
3  # A simple class in Python
4  class counter():
5    # The __init__ method
6    def __init__(self, count):
7      self.count = count
8
9    # The decrement method
10   def decrement(self):
11     self.count -= 1
12     return self.count
13
14   # The value method
15   def value(self):
16     return self.count
17
18 # Create a counter and decrement the value twice
19 if __name__ == '__main__':
20     c = counter(9)
21     x = c.decrement()
22     y = c.decrement()
23     print counter.value(c)
```

Fig. 7. A python class with a method that mutates its data members

```
1  #!/usr/bin/runghc
2
3  -- A simple class in Haskell
4  data Counter = Counter Integer
5
6  -- Function for constructing a new counter
7  -- This takes the place of the __init__ method
8  counter count = Counter count
9
10 {- The decrement method returns
11 a tuple containing the current count and a new
12 data type value representing the mutated state -}
13 decrement (Counter count) = (count-1, (Counter (count-1)))
14
15 -- The value method
16 value (Counter count) = count
17
18 -- Create a counter and decrement the value twice
19 main = do
20 let c = counter 9
21     (x,c') = decrement c
22     (y,c'') = decrement c'
23     in print(value c'')
```

Fig. 8. A translation of the Python class in Haskell

It was also problematic to translate Python functions and methods that had default arguments. Haskell does not support the notion of default arguments. This was resolved by defining new Haskell functions for each variation of the required arguments. Each function variation was given a slightly different name from the original, and the appropriate function was called from other parts of the code. See Figures 9 and 10 for an example of a Python function with default arguments and the respective Haskell translation.

```
1  #!/usr/bin/python
2
3  # Generate RGBA color
4  def makeRgb(r,g,b,a=1.0):
5    return (r,g,b,a)
```

Fig. 9. A python function with a default argument

```
1  #!/usr/bin/runghc
2
3  -- Generate RGBA color
4  {- This variant of the function only takes
5     three arguments the 4th argument is
6     defaulted to 1.0 -}
7  makeRgb r g b = makeRgb' r g b 1.0
8
9  -- Generate RGB colors
10 {- This variant of the function takes all
11    four arguments -}
12 makeRgb' r g b a = (r,g,b,a)
```

Fig. 10. A Haskell translation of a function with a default argument

The final challenge in translating the Python code to Haskell was the use of Python's rich set of built in libraries. In some cases, Haskell provided a similar library

that could be used as a drop in replacement. When a similar Haskell library did not already exist, a replacement had to be implemented by defining Haskell functions and data structures that duplicated the Python interfaces. Many of these functions and data structures were able to be used in the translation of more than one programming project.

Many of these translation strategies ignore many of the more powerful and idiomatic mechanisms (such as monads and type classes) of Haskell which would likely have resulted in a simpler translation that was less syntactically similar to the original Python code. By using these strategies, the resulting translation was syntactically similar to the Python code and, therefore, easier to verify that a correct translation was made.

While each project was being translated, the Haskell code was continuously checked for type errors using the Haskell type checker. When the type checker reported a type error, the translation process was stopped, and the nature of the error was examined. The purpose of the examination was to determine whether the type error was benign or whether the type error could be manifest at runtime. This determination was made by analyzing the original Python software and by attempting to write a Python unit test that would trigger a runtime error. Once the nature of the error had been determined, the Haskell version of the program was minimally modified to remove the type error and the translation process continued.

When the translation of a software project was completed, each individual unit test was manually examined to see if it could be eliminated from the statically typed version without sacrificing software verification.

CHAPTER IV

RESULTS

In order to study potential error detection benefits of static type checking, four software projects were translated from Python to Haskell. During the translation process it became clear that a full understanding of the effect of applying static type checking to dynamically typed software would require a detailed description of each defect along with the scenarios that would cause each defect to be manifest at run time. The results and description of each project translation are detailed below.

The Python NMEA Tookit

The first project that was studied was the Python NMEA Toolkit [22]. The Python NMEA Toolkit is a library for communicating with GPS devices using the line oriented NMEA protocol. During the translation of this toolkit, several type errors were discovered. The first type error that was discovered by the translation process was in the `parse_data` method of the `Gps` class. See Figure 11 for relevant code from the `parse_data` method [22].

The `parse_data` method reads in all available NMEA sentences from an input device and then creates a `Sentence` object for each NMEA sentence. If a NMEA sentence is malformed, a `ParseError` exception is raised. The exception is caught by an exception handler within the `parse_data` method that tries to call a method named

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         gps.py
# Class:        Gps
# Method:       parse_data

83  def parse_data(self):
...
93      lines = self.port.read_buffered()
94      for line in lines:
95          try:
96               sentence = Sentence(line)
97          except ParseError, ex:
98               self.error_message(str(ex))
99          else:
...
```

Fig. 11. The *parse_data* method from the Python NMEA Toolkit

error_message. The *error_message* method is not defined, so a Python *AttributeError* is raised. Because the *AttributeError* is raised, all remaining available NMEA sentences are discarded. If the type error was corrected by defining the *error_message* method, only the malformed NMEA sentence would be discarded, and the rest of the sentences could be processed. One could argue that the original developer expected future developers who use this library to subclass the *Gps* class and provide a custom *error_message* method. This is an unlikely scenario, however, because the requirement to add a custom *error_message* method is not documented in the code. Furthermore, early revisions of the *Gps* class did include an implementation of the *error_message* method. The code was later re-factored, and the *error_message* method and references to it were removed. The most likely explanation is that this error was introduced during the refactoring process. While the Python NMEA Toolkit [22] did provide unit tests, none of the tests detected this defect. Had the unit tests included a test for malformed NMEA sentences, this type error would

have almost certainly been discovered. The Haskell type checker was able to discover this error at compile time.

The second type error was found in the *_parse_GSV* method of the *Gps* class. See Figure 12 for relevant code from the *_parse_GSV* method [22].

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         gps.py
# Class:        Gps
# Method:       _parse_GSV

175  def _parse_GSV(self, sentence):
176      """ Parse "GPS Satellites in View" sentence """
177      totalMsgs = sentence.get_int(0)
178      msgNumber = sentence.get_int(1)
179      totalSats = sentence.get_int(2)
180      if msgNumber < totalMsgs:
181          satRange = 4
182      else:
183          satRange = totalSats - ((msgNumber - 1) * 4)
...
```

Fig. 12. The *_parse_GSV* method from the Python NMEA Toolkit

The method takes in a *Sentence* object and uses the *Sentence* object's *get_int* method to retrieve the first three NMEA sentence fields. The *get_int* method returns the fields as integers unless the field is empty in which case the method may return the Python *None* object. See Figure 13 for the *get_int* method [22].

The *_parse_GSV* method uses the values returned by *get_int* in mathematical expressions. Because basic mathematical operators are not defined for the Python *None* object, empty fields may result in the raising of the *TypeError* exception. Because only empty fields in a NMEA sentence will cause the type error to be manifest at run time, full unit test code coverage of the *_parse_GSV* method may not

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         parse.py
# Class:        Sentence
# Method:       get_int

104  def get_int(self, index, default=None):
105      """ Get an int item """
106      value = self._words[index]
107      if len(value) == 0: return default
108      try:
109          return int(value)
110      except ValueError:
111          raise ParseError("Word is not an int")
```

Fig. 13. The *get_int* method from the Python NMEA Toolkit

have detected this defect. The Haskell type checker detected the type error at compile time.

The third type error discovered by the translation process is in the *Sentence* class' *get_velocity* method. See Figure 14 for the *get_velocity* method [22].

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         parse.py
# Class:        Sentence
# Method:       get_velocity

122  def get_velocity(self, index, default=None):
123      """ Get a velocity item """
124      return velocity(self.get_float(index, default))
```

Fig. 14. The *get_velocity* method from the Python NMEA Toolkit

The *get_velocity* method retrieves a NMEA sentence field using the *get_float* method. See Figure 15 for the *get_float* method [22].

The *get_velocity* method uses the results of the *get_float* call to construct a *velocity* object. See Figure 16 for relevant code from the *velocity* class [22].

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         parse.py
# Class:        Sentence
# Method:       get_float

113     def get_float(self, index, default=None):
114         """ Get an float item """
115         value = self._words[index]
116         if len(value) == 0: return default
117         try:
118             return float(value)
119         except ValueError:
120             raise ParseError("Word is not a float")
```

Fig. 15. The *get_float* method from the Python NMEA Toolkit

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         _types.py
# Class:        velocity

164 class velocity(float):
165     """ Speed value (default is knots to match nmea spec) """
166
...
```

Fig. 16. The *velocity* class declaration from the Python NMEA Toolkit

The velocity class inherits from *float* and must be constructed with either a number or a string representation of a number. The *get_float* method like the *get_int* method can return a Python *None* object if the sentence field is empty. This results in a type error in the *get_velocity* method when it attempts to construct a *velocity* object with the Python *None* value. Like the previous type error full code coverage of the *get_velocity* method would not guarantee that this type error would be discovered. The Haskell type checker detected the type error at compile time.

The final type errors in the Python NMEA Toolkit [22] are triggered when calling the *fileno*, *read* or *write* methods on a closed *TcpPort* object. The *close* method of the *TcpPort* class closes the underlying socket device and sets the *self.sock* member variable to *None*. See Figure 17 for the *close* method [22].

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         tcpport.py
# Class:        TcpPort
# Method:       close

44  def close(self):
45      """ Close the nmea port """
46      if self.sock:
47          self.sock.close()
48          self.sock = None
```

Fig. 17. The *close* method of the *TcpPort* class from the Python NMEA Toolkit

The *fileno*, *read* and *write* methods all may raise the *AttributeError* exception because all of these methods assume that *self.sock* is a valid socket and has not been assigned the *None* value. It is interesting to note that the *close* method does check for a *None* value, and so it is safe to call the *close* method on a closed *TcpPort* object. This type error is also interesting in that it will only be manifest if a developer chooses to call these methods after the connection has been closed. In other words, only by misusing the API will the error be manifest. While it is possible that raising the *AttributeError* is the behavior intended by the original developer, it can be argued that the developer should have provided a more meaningful error message. Even though it is impossible to know whether the current error handling mechanism is intentional, it is interesting to note that the Haskell type system would force the developer to consider the case of *self.sock* having a *None* value and to

explicitly handle this scenario. This restriction by the Haskell type system would have likely resulted in a more descriptive error message. The code for the *SerialPort* class contains the same defects as the *TcpSocket* class.

Unless the unit tests called one of these methods on a closed *TcpSocket* object, it is likely that full code coverage unit testing would not have discovered this defect. The Haskell type system was able to discover this defect. There is a run time error in the program that can be eliminated when static type checking is applied. The *update* method of the *satellite* class was written to explicitly throw a *ValueError* if it is passed an argument that is not either a *satellite* object or a tuple. See Figure 18 for the *update* method of the *Sentence* class [22].

```
# Project:      Python NMEA Toolkit
# Revision:     23:c3b4b4c61e3d
# File:         _types.py
# Class:        satellite
# Method:       update

144 def update(self, value):
145     if isinstance(value, tuple):
146         (prn, elevation, azimuth, snr) = value
147         self.prn = prn
148         self.elevation = elevation
149         self.azimuth = azimuth
150         self.snr = snr
151     elif isinstance(value, satellite):
152         self.prn = value.prn
153         self.elevation = value.elevation
154         self.azimuth = value.azimuth
155         self.snr = value.snr
156     else:
157         raise ValueError
```

Fig. 18. The *update* method of the *Sentence* class from the Python NMEA Toolkit

The Haskell type checker ensures that this method is called with values of the appropriate type and so not only is the `ValueError` eliminated, the code can be simplified to remove the explicate type checks.

After the Python NMEA Toolkit [22] was translated, the unit tests were examined. It was determined that two of the unit tests could safely be removed because they only tested for type safety. These two unit tests account for 8.7% of the unit tests.

The Python NMEA Toolkit [22] did not use programming constructs which would be rejected by the Haskell type system, but would not result in a runtime error.

It is clear that for the NMEA Toolkit, the unit tests did not negate the defect detecting benefits of static type checking. There were three type errors that were discovered that could be triggered due to malformed NMEA sentences and six type errors that could be triggered by misusing the API. Of these type errors, only one would be guaranteed to be discovered if the unit tests had full code coverage. There was additionally one run time error that could be eliminated and two unit tests which were not needed once static typing was applied. The toolkit did not utilize any dynamic code constructs that resulted in either benign type errors or code that was cumbersome to translate into a statically typed programming language.

MIDIUtil

The second project that was examined is the MIDIUtil [23] library. The MIDIUtil library is a Python library for writing MIDI files. A couple of type errors were discovered during the translation process. The first discovered type error was found in the `__eq__` method of the `GenericEvent` class. The `__eq__` method performs a

comparison on *GenericEvent* class and all known subtypes. It contains specialized code for the *ControllerEvent* subclass that compares a field named *parameter2*. This field does not exist in the *ControllerEvent* class, and so an *AttributeError* is raised. See Figure 19 for relevant code from the *__eq__* method [23].

```
# Project:      MIDIUtil
# Revision:     r10
# File:         MidiFile.py
# Class:        GenericEvent
# Method:       __eq__

56 def __eq__(self, other):
...
90     if self.type == 'controllerEvent':
91         if self.parameter1 != other.parameter1 or \
92             self.parameter2 != other.parameter2 or \
93             self.channel != other.channel or \
94             self.eventType != other.eventType:
95             return False
```

Fig. 19. The *__eq__* method of the *GenericEvent* class from the MIDIUtil library

While the full history of the MIDIUtil library [23] is not available, it appears that the *parameter2* member previously existed and was removed in later revisions. This library did provide unit tests, but none of the tests detected this defect. If the unit tests had compared two *ConrollerEvent* objects for equality, this defect would have been detected. The Haskell type checker was able to detect this error.

The second type error in this library was found in the *deInterleaveNotes* method of the *MIDITrack* class. See Figure 20 for relevant code from the *deInterleaveNotes* method [23].

```
# Project:      MIDIUtil
# Revision:     r10
# File:         MidiFile.py
# Class:        MIDITrack
# Method:       deInterleaveNotes

549 def deInterleaveNotes(self):
...
558  tempEventList = []
559  stack = {}
560
561  for event in self.MIDIEventList:
562
563    if event.type == 'NoteOn':
564      if stack.has_key(str(event.pitch)+str(event.channel)):
565        stack[str(event.pitch)+str(event.channel)].append(event.time)
566      else:
567        stack[str(event.pitch)+str(event.channel)] = [event.time]
568      tempEventList.append(event)
569    elif event.type == 'NoteOff':
570      if len(stack[str(event.pitch)+str(event.channel)]) > 1:
571        event.time = stack[str(event.pitch)+str(event.channel)].pop()
572        tempEventList.append(event)
573      else:
574        stack[str(event.pitch)+str(event.channel)].pop()
575        tempEventList.append(event)
576    else:
577      tempEventList.append(event)
...
```

Fig. 20. The *deInterleaveNotes* method of the *MIDITrack* class from the MIDIUtil library

The *deInterleaveNotes* method processes all the events in the *self.MIDIEventList* list. When a *NoteOn* event is encountered, information about the event is recorded in the *stack* dictionary object. When the corresponding *NoteOff* event is encountered, the information is retrieved from the *stack* dictionary. This method assumes that the *NoteOn* event comes before the *NoteOff* event in the *self.MIDIEventList* list. If a *NoteOff* event came before the corresponding *NoteOn* event then the *len* function would be called on a *None* value and a *TypeError* would be raised. The *processEventList* method places the *NoteOn* and *NoteOff* objects on the *self.MIDIEventList* list and then attempts to ensure

that *NoteOn* events come before *NoteOff* events by sorting all events by their *time* field in descending order. See Figure 21 for the relevant code from the *processEventList* method [23].

```
# Project:      MIDIUtil
# Revision:     r10
# File:         MidiFile.py
# Class:        MIDITrack
# Method:       processEventList

293 def processEventList(self):
...
303   for thing in self.eventList:
304     if thing.type == 'note':
305       event = MIDIEvent()
306       event.type = "NoteOn"
307       event.time = thing.time * TICKSPERBEAT
308       event.pitch = thing.pitch
309       event.volume = thing.volume
310       event.channel = thing.channel
311       event.ord = 3
312       self.MIDIEventList.append(event)
313
314       event = MIDIEvent()
315       event.type = "NoteOff"
316       event.time = (thing.time + thing.duration) * TICKSPERBEAT
317       event.pitch = thing.pitch
318       event.volume = thing.volume
319       event.channel = thing.channel
320       event.ord = 2
321       self.MIDIEventList.append(event)
...
379     else:
380       print "Error in MIDITrack: Unknown event type"
381       sys.exit(2)
...
383   # Assumptions in the code expect the list to be time-sorted.
384   # self.MIDIEventList.sort(lambda x, y: x.time - y.time)
385
386   self.MIDIEventList.sort(lambda x, y:\
                                int( 1000 * (x.time - y.time)))
387
388   if self.deinterleave:
389     self.deInterleaveNotes()
```

Fig. 21. The *processEventList* method of the *MIDITrack* class from the MIDIUtil library

The *time* field of the *NoteOff* event is guaranteed to be greater than the *time* field of the *NoteOn* object as long as the *duration* field is positive. Because

the code does not prevent a user of the API from passing a negative `duration` field, it is possible for the `NoteOff` event to come before the `NoteOn` event and therefore cause the `TypeError` exception to be raised. It is likely that the original developer simply did not consider the effect of the user passing a negative `duration` value as a negative duration is nonsensical. The Haskell type system caught this type error and forces the developer to explicitly handle this error condition. While this library did have unit tests, none of them caught this error. Even if this library had full unit test code coverage, this defect would not have been caught unless one of the unit tests explicitly tested a negative `duration` value.

A runtime error is eliminated when static type checking is applied is in the `processEventList` method. This method selects behavior based on the string value of the `GenericEvent.type` data member. See Figure 21 for the relevant code from the `processEventList` method. If the `self.eventList` contains an object of type `GenericEvent`, but is not handled in this method, then the `processEventList` will print an error message and exit. Haskell's static type checker ensures at compile time that all `GenericEvent` variants are handled by the `processEventList` method.

After the MIDIUtil [23] library was translated, the unit tests were examined. It was determined that none of the unit tests could be safely removed when static type checking was applied.

The MIDIUtil [23] library uses the `struct.pack` and `struct.unpack` methods. These methods serialize Python values to binary data and binary data to Python

values respectively. The methods determine how to serialize the data based on a format string. Because the format string directs the serialization process, the type of the arguments to `struct.pack` and the type of the return value from from `struct.unpack` may vary. The variance in the types of arguments and return values resulted in the Haskell translation of these methods being rejected by the Haskell type checker even though the usage of these methods would not result in runtime errors. In order to work around this limitation, several Haskell `pack` and `unpack` methods were defined that each serialize a different data type. The pack and unpack methods are composed in the same order that is specified in the Python format string. While this approach does differ slightly from the Python mechanism, it is no less dynamic than the Python version. Hard-coding serialization methods into Haskell is no less flexible than hard-coding a format string in Python.

For the MIDIUtil [23], unit testing did not negate the benefits of static type checking. While one of the type errors would have been discovered with full code coverage of unit testing, the second type error would likely have gone un-detected. Additionally, a runtime error was able to be eliminated.

GrapeFruit

The third project that was translated was the GrapeFruit [24] color manipulation library. GrapeFruit is a library for translating between various color systems (RGB, HSL, CMY, etc.). No type errors were discovered during the translation of this library. There was, however, one runtime error that could be eliminated with the application of Haskell's static type checking. The `Color` class' `__init__` method

explicitly raises a *TypeError* exception if the *values* argument is not a tuple. See Figure 22 for the relevant code from the *__init__* method. The Haskell static type checker can prevent this programming error at compile time.

```
# Project:      GrapeFruit
# Revision:     r31
# File:         grapefruit.py
# Class:        Color
# Method:       __init__

273   def __init__(self, values, mode='rgb', alpha=1.0, wref=_DEFAULT_WREF):
...
287     if not(isinstance(values, tuple)):
288       raise TypeError, 'values must be a tuple'
...
```

Fig. 22. The *__init__* method of the *Color* class from GrapeFruit [24]

When the GrapeFruit [24] library's unit tests were examined, a single unit test in the *testEq* method was discovered that could be removed after static typing was applied. This test simply verified that a string representation of a color tuple was not equivalent to the *Color* object in the *self.rgbCol* member. This test accounts for only 0.008% of the total unit tests. See Figure 23 for the unit test code [24].

```
# Project:      GrapeFruit
# Revision:     r31
# File:         grapefruit_test.py
# Class:        ColorTest
# Method:       testEq

208   def testEq(self):
...
213     self.assertNotEqual(self.rgbCol, '(1.0, 0.5, 0.0, 1.0)')
```

Fig. 23. The *testEq* method of the *ColorTest* class from the GrapeFruit library

This library did not use programming constructs that would not cause a runtime failure, but would be rejected by the Haskell type system.

The GrapeFruit [24] library's unit tests did appear to negate the benefits of static type checking as no type errors were found. There was, however, a single runtime error that could be translated to a compile time error and a single unit test could be eliminated.

PyFontInfo

The final project that was translated was the PyFontInfo [25] library. This library is used to extract header information from font files.

The first type errors that were discovered when translating this library to Haskell are found in the *parseChildren* methods of the *TABLE_RECORD* and *NAME* classes. See Figures 24 and 25 for the *TABLE_RECORD* and *NAME* class' *parseChildren* methods [25].

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         PFI_Tables.py
# Class:        NAME
# Method:       parseChildren

303     def parseChildren(self, fp):
304         for i in range(0, self.data.count):
...
```

Fig. 24. The *parseChildren* method of the *NAME* class from the PyFontInfo library

The *parseChildren* methods reference data members of the object assigned to *self.data*. The *self.data* member is created in the *TTF_HEADER* *parse* method. See Figures 26 for the *TTF_HEADER* *parse* method [25].

While the *parse* method does call *parseChildren* after the *self.data* member has been created, if the *parseChildren* method were to be called directly

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         PFI_Tables.py
# Class:        TABLE_RECORD
# Method:       parseChildren

91     def parseChildren(self, fp):
92         for i in range(0, self.data.numTables):
...
```

Fig. 25. The *parseChildren* method of the *TABLE_RECORD* class from the PyFontInfo library

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         PFI_Tables.py
# Class:        TTF_HEADER
# Method:       parse

25     def parse(self, fp):
...
35         self.data = self.tpl._make(x)
36         self.parseChildren(fp)
...
```

Fig. 26. The *parse* method of the *TTF_HEADER* class from the PyFontInfo library

without calling *parse* first, Python would raise an *AttributeError* when *self.data* was referenced. It is likely that the original developer intended for the *parseChildren* methods to only be called from the *parse* method, but neglected to enforce the restriction. The Haskell type system requires that the *self.data* member always exist and that the *parseChildren* methods handle the case where *self.data* attribute has not been initialized. The provided unit tests did not catch these errors. Even if full unit test code coverage had been provided, it would be possible for these type errors to go undetected.

The next set of type errors are found in the *getPANOSE*, *getOS2*, *getHead*, *getNames* methods of the *PyFontInfo* class. See Figures 27, 28, 29 and 30 for the *getPANOSE*, *getOS2*, *getHead*, *getNames* methods of the *PyFontInfo* class [25].

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         __init__.py
# Class:        PyFontInfo
# Method:       getPANOSE

68     def getPANOSE(self):
...
78         return self.os2.PANOSE
```

Fig. 27. The *getPANOSE* method of the *PyFontInfo* class from the PyFontInfo library

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         __init__.py
# Class:        PyFontInfo
# Method:       getOS2

80     def getOS2(self):
...
86         return self.os2.data._asdict()
```

Fig. 28. The *getOS2* method of the *PyFontInfo* class from the PyFontInfo library

The *getPANOSE* and *getOS2* methods each reference the *self.os2* member which is initialized in the *PyFontInfo* *__init__* method if an "*OS/2*" record exists in the font file. See Figure 31 for the *__init__* methods of the *PyFontInfo* class [25].

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         __init__.py
# Class:        PyFontInfo
# Method:       getHead

88     def getHead(self):
...
94         return self.head.data._asdict()
```

Fig. 29. The *getHead* method of the *PyFontInfo* class from the PyFontInfo library

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         __init__.py
# Class:        PyFontInfo
# Method:       getNames

96     def getNames(self):
...
103         ret = self.name.UNICODE
...
```

Fig. 30. The *getNames* method of the *PyFontInfo* class from the PyFontInfo library

If an "OS/2" record does not exist and either the *getPANOSE* or *getOS2* methods are called, an *AttributeError* will be raised. The *getHead* and *getNames* methods reference the *self.head* and *self.name* members respectively. These members are initialized to *None*, but may be set to a *HEAD* or *NAME* object in the *PyFontInfo* *__init__* method if a "head" or "name" record exists. If the respective records do not exist, and the *getHead* or *getNames* methods are called, an *AttributeError* will be raised. Even if full unit test code coverage had been provided, it would be possible for these defects to go undetected.

There is a runtime error that can be eliminated when static type checking is applied. The *__init__* method of the *PyFontInfo* class takes an argument that must be a *string* or a file-like object. If the argument is not one of these types, a

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         __init__.py
# Class:        PyFontInfo
# Method:       __init__

34     def __init__(self, f):
...
36         self.head = None
37         self.name = None
...
39         try:
40             if type(f) == str:
41                 fp = open(f, 'rb')
42             elif type(f) == file:
43                 fp = f
44             else:
45                 raise PFI_Exceptions.BadFileType(f)
46         except IOError:
47             raise
...
51         self.DefinedRecords = [i for i in table.children_map]
52         if 'head' in self.DefinedRecords:
...
54             self.head = PFI_Tables.HEAD()
...
57         if 'name' in self.DefinedRecords:
...
59             self.name = PFI_Tables.NAME()
...
62         if 'OS/2' in self.DefinedRecords:
...
64             self.os2 = PFI_Tables.OS_2()
...
```

Fig. 31. The *__init__* method of the *PyFontInfo* class from the PyFontInfo library

PFI_Exceptions.BadFileType exception is raised. While this check happens at runtime in the Python code, the Haskell type checker can perform this check at compile time.

An additional error that was detected during the translation of the Python code to Haskell is found in the *HMTX* class. The *HMTX* class has a *fmt* data member that is used as a format string for a Python *struct.unpack* call. See Figure 32 for the *HMTX* class [25].

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         PFI_Tables.py
# Class:        HMTX

191 class HMTX(TTF_HEADER):
...
198     fmt = '>?[]h[]'
...
202
```

Fig. 32. The *HTMX* class from the PyFontInfo library

The format string specified in the *HMTX* class is invalid, which result in a *struct.error* runtime error in the Python code. Due to the decision to use a sequence of unpacking functions in Haskell instead of a Python format string, this defect is changed from a runtime error to a compile time error in the Haskell translation. While this error in Python is not a type error, the restrictions of the Haskell static type checker required an implementation that transforms the runtime error into a type error. As was pointed out before, hard-coding a series of Haskell functions to unpack the binary data is no less inflexible than hard-coding a Python format string.

After examining the unit tests, it was determined that a single unit test could be eliminated. See Figure 33 for the eliminated unit test [25].

The test attempted to construct a *PyFontInfo* object with an integer argument. *PyFontInfo* then calls *open* using the integer as the filename. This unit test is designed to make sure the PyFontInfo raises a Python *TypeError* exception when constructed with an integer argument. In the Haskell translation this is a type error.

The unit tests provided by the PyFontInfo [25] did not negate the benefits of static type checking. There were six type errors that were discovered by the Haskell type

```
# Project:      PyFontInfo
# Revision:     6:290ce911500b
# File:         test.py

23 test_files = []
...
31 test_files.append(14)
32
33 for f in test_files:
...
36     try:
37         info = PyFontInfo(f)
...
```

Fig. 33. The eliminated PyFontInfo library unit test

checker and two runtime errors that could be removed. There was a single unit test that only tested type safety that could be eliminated.

Just like the MIDIUtil [23] library, the PyFontInfo [25] library uses the *struct.pack* and *struct.unpack* methods, which require some modification to work around the limitations of static type checking. In this library the work-around resulted in transforming the Python runtime error to a type error. Just like the MIDIUtil [23] library, the Haskell version is no less dynamic than the Python version.

CHAPTER V

CONCLUSION

The translation of these four software projects from Python to Haskell proved to be an effective way of measuring the effects of applying static type checking to unit tested software. In the studied software projects, there were a total of four unit tests that could be removed from the Haskell translation because their sole function was to test for type safety. Because only a small subset of the total unit tests could be eliminated, it appears that the developers of these software projects did not spend a lot of time duplicating static type checking in the unit tests.

None of the studied software projects utilized Python's dynamic features in a way that made it difficult to translate the software to a statically typed programming language. Two of the software projects did use Python's `struct.pack` and `struct.unpack` which were initially rejected by Haskell's static type checker. However, with some minor modifications to the translation, it was possible to create type safe alternatives to these methods.

The consequence of these first two observations is that all of the software projects could have easily been implemented in either a dynamically or statically typed programming languages with only minor differences.

The final question this study attempted to address is whether unit testing in practice is a good substitution for static type checking. There were a total of 17 type

errors that were discovered when translating the four software projects. Only two of the 17 type errors were guaranteed to be discovered with full unit test code coverage. Based on these results, the conclusion can be reached that while unit testing can detect some type errors, in practice it is an inadequate replacement for static type checking.

The GrapeFruit [24] library stands out from the other studied projects in that it did not have any type errors. There are two notable factors that could contribute to the type safety of the GrapeFruit [24] library. First, the library's unit test were more extensive than the other projects. As noted above, unit testing is not a replacement for static type checking, but full code coverage unit tests will detect some type errors. There is a second and perhaps more important difference between the GrapeFruit [24] library and the other software projects. The GrapeFruit [24] library was written such that each of the library's variables will only be assigned values of a single type. Likewise, each of the library's methods will only return values of a single type. In statically typed programming languages, the restriction of variables and return values to a single type is enforced by the type checker. Fifteen of the 17 type errors that were found in the other software projects were caused by variables that would hold or methods that would return values of differing types. By restricting the values held in variables and the values returned by methods to a single type, the developer of the GrapeFruit [24] library forfeited the benefits of dynamic typing in exchange for a the possibility of fewer code defects.

CHAPTER VI

FUTURE RESEARCH

There are several opportunities for future research related to this study. Because the manual translation was time consuming and labor intensive, this study was limited to studying four software projects with less than 2000 lines of source code. Additional insights may be gained by studying a larger number of software projects including projects that contain more than 2000 lines of code. The efficiency of translating software from Python to Haskell may be improved by investing time in researching ways to fully or partially automate the translation process.

During the translation process, it was discovered that it was beneficial to have both the Python code and the Haskell translation open side-by-side to ensure that a correct translation was being made. It was also beneficial to receive constant feedback on whether the Haskell translation was syntactically correct. This was accomplished by writing simple shell scripts that would constantly compile the Haskell translation and provide auditory feedback when the Haskell translation failed to compile. Future researchers may want to create similar tools to aid the translation process.

Another interesting opportunity for future study would be to translate dynamically typed software projects into several different statically typed programming languages. Due to the variance of static type systems between different programming

languages, some type systems may be better suited than others at discovering defects in unit tested dynamically typed programming languages.

No effort was put into understanding how static type checking and unit testing affects the speed of software development, or if either defect detecting system aided the developer in understanding the source code or designing the software. All of these issues may contribute to the overall cost and benefits of both unit testing and static type checking.

It is hoped that this research project not only provides valuable insight into the practical limitations of unit testing as a replacement for static type checking, but also encourages further research into this topic.

REFERENCES

REFERENCES

[1] H. Zhu, P. A. Hall, and J. H. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[2] L. Cardelli, "Type Systems," *The Computer Science and Engineering Handbook*, A. B. Tucker, ed. CRC Press, pp. 2208–2236, 1997.

[3] J. Spolsky and B. Eckel, "Strong Typing vs. Strong Testing," in *The Best Software Writing I*. Apress, pp. 67–77, 2005.

[4] J. Gannon, "An Experimental Evaluation of Data Type Conventions," *Communications of the ACM*, vol. 20, no. 8, pp. 584–595, 1977.

[5] S. Hanenberg, "Doubts About the Positive Impact of Static Type Systems on Programming Tasks in Single Developer Projects - an Empirical Study," *ECOOP 2010 Object-Oriented Programming*,. Lecture Notes in Computer Science, T. DHondt, ed. Springer Berlin / Heidelberg, vol. 6183, pp. 300–303, 2010.

[6] L. Prechelt and W. F. Tichy, "A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking," *IEEE Transactions on Software Engineering*, vol. 24, pp. 302–312, 1998.

[7] B. Cannon, "Localized Type Inference of Atomic Types in Python," Master's thesis, California Polytechnic State University, San Luis Obispo, 2005.

[8] L. Chen, B. Xu, T. Zhou, and X. Zhou, "A Constraint Based Bug Checking Approach for Python," *COMPSAC '09: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, pp. 306–311, 2009.

[9] M. Furr, J. D. An, J. S. Foster, and M. Hicks, "Static Type Inference for Ruby," *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA: ACM, pp. 1859–1866, 2009.

[10] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977.

[11] F. Henglein and J. Rehof, "Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML," *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*. ACM Press, pp. 192–203, 1995.

[12] D. Duggan and F. Bent, "Explaining Type Inference," *Science of Computer Programming*, vol. 27, no. 1, pp. 37–83, 1996.

[13] M. Ellims, J. Bridges, and D. C. Ince, "Unit Testing in Practice," *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, pp. 3–13, 2004.

[14] L. Madeyski, "Impact of Pair Programming on Thoroughness and Fault Detection Effectiveness of Unit Test Suites," *Software Process Improvement and Practice*, vol. 13, no. 3, pp. 281–295, 2008.

[15] M. M. Muller and O. Hagner, "Experiment About Test-First Programming," *IEEE Proceedings - Software*, vol. 149, no. 5, pp. 131–136, 2002.

[16] A. J. Simons and C. D. Thomson, "Benchmarking Effectiveness for Object-Oriented Unit Testing," *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. Washington, DC, USA: IEEE Computer Society, pp. 375–379, 2008.

[17] “Python Language Reference,” Retrieved September 11, 2010 from the World Wide Web: http://docs.python.org/reference.

[18] S. Marlow, ed., *Haskell 2010 Language Report*. Cambridge University Press, 2010.

[19] “Bitbucket,” Retrieved September 1, 2010 from the World Wide Web: https://bitbucket.org.

[20] “Project Hosting on Google Code,” September 1, 2001 from the World Wide Web: http://code.google.com/hosting.

[21] A. Danial, “Count Lines of Code,” Sourceforge, 2011. Retrieved September 10 from the World Wide Web: http://cloc.sourceforge.net.

[22] T. Savage, “Python NMEA Toolkit”, Google Project Hosting, 2009. Retrieved September 20, 2010 from the Internet via Mercurial: https://python-gpsd.googlecode.com/hg/. rev. 23:c3b4b4c61e3d.

[23] M. Wirt, “MIDIUtil”, Google Project Hosting, 2010. Retrieved October 05, 2010 from the Internet via Subversion: http://midiutil.googlecode.com/svn/trunk. rev. r10.

[24] X. Basty, “GrapeFruit”, Google Project Hosting, 2008. Retrieved October 16, 2010 from the Internet via Subversion: http://grapefruit.googlecode.com/svn/trunk. rev. r31.

[25] S Zong Chen, “PyFontInfo”, Bitbucket, 2010. Retrieved November 2, 2010 from the Internet via Mercurial http://bitbucket.org/sirpengi/pyfontinfo. rev. 6:290ce911500b.