Implementing *Fickle*

Christopher Anderson

cla97@doc.ic.ac.uk

Supervisor: Sophia Drossopoulou

Second Marker: Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
London
SW7 2BZ

October 2, 2001
Abstract

*Fickle* is an exciting development in object oriented programming, characterized by the ability of an object to change its class membership at run-time.

We develop an implementation *Carmela*, that compiles *Fickle* into Java. We prove the correctness through a formalization of the mapping. We demonstrate the correctness of *Carmela* through a test suite.

We discuss the issues most important to the development of *Carmela*. We also present some novel extensions to *Fickle*. 
Acknowledgments

My very greatest thanks go to Dr. Sophia Drossopoulou. She has provided me with more time than could have ever been expected. Her enthusiasm, guidance, friendship, teaching and support have been instrumental in the research and development of Carmela. I would also like to thank Susan Eisenbach for her support and acting as second marker.

I would also like to thank Davide Ancona, Ferruccio Damiani, Paola Giannini and Elena Zucca for their feedback and allowing me to work with them on the translation paper.

Finally I would like to thank my friends and family for putting up with me talking about Fickle, non stop for nine months!!

---

1Sources and updates to Carmela can be found at http://www.dec.ied.ac.uk/~cla97/
# Contents

1 Introduction
   1.1 Dynamic Object Re-classification .............................................. 7
   1.2 My Contribution ............................................................... 7
   1.3 Report Structure .............................................................. 8

2 Background
   2.1 Philosophy of Dynamic Object Re-classification .......................... 9
      2.1.1 Class-based languages .................................................. 9
      2.1.2 Evolutionary Objects ................................................... 11
   2.2 Dynamic Object Re-classification - A direct solution .................. 11
      2.2.1 Dynamically typed solutions ........................................... 12
      2.2.2 Statically Typed solutions ............................................ 12
   2.3 Fickle-2000 - Current Version ............................................ 14
   2.4 Making Fickle programs run ................................................ 15
      2.4.1 Compilation of Fickle ................................................... 15
      2.4.2 Interpreting Fickle ..................................................... 16
      2.4.3 Preprocessing ........................................................... 16
   2.5 Implementation Language ................................................... 16

3 Specification and Tools
   3.1 Lexical Analysis/ Parsing .................................................... 18
      3.1.1 Selection of lexer/parser generator .................................. 18
      3.1.2 Specification for Lexer/Parser ...................................... 20
   3.2 Attribution/Semantic Analysis .............................................. 21
      3.2.1 Specification of Semantic Analyzer .................................. 21
   3.3 Code Generation ............................................................. 23
      3.3.1 Specification of Code Generation ................................... 23
   3.4 Miscellaneous issues ....................................................... 24

4 Semantic Analysis ............................................................... 25
   4.1 Semantic Analysis ........................................................... 25
      4.1.1 Core Data Structures .................................................. 26
      4.1.2 Symbol Collection ..................................................... 28
      4.1.3 Attribution ............................................................. 31

5 Code Generation ................................................................. 36
   5.1 Overview ................................................................... 36
   5.2 Object Layout - A rough guide ............................................. 36
   5.3 The changing face of this .................................................. 36
      5.3.1 Format for freshness ................................................... 37
   5.4 The translation of this ..................................................... 40
      5.4.1 Aliasing issues with this ............................................. 41

3
5.5 Formal Description of Translation ........................................ 43
  5.5.1 Programs .......................................................... 43
  5.5.2 Classes .......................................................... 44
  5.5.3 Method declarations ............................................. 45
  5.5.4 Expressions ..................................................... 45
5.6 Translation of the Flip/Flop Example .................................... 46

6 Alternative Translations of Fickle into Java ............................... 48
  6.1 Reference Tracking .................................................. 48
     6.1.1 Advantages ................................................... 49
     6.1.2 Disadvantage .................................................. 49
  6.2 'The Italian Approach' .............................................. 49
     6.2.1 Translation of the Flip/Flop Example ....................... 50
     6.2.2 Advantages ................................................... 51
     6.2.3 Disadvantages ................................................. 51
  6.3 Method Splitting .................................................... 51
     6.3.1 Advantages ................................................... 54
     6.3.2 Disadvantages ................................................. 54

7 Proving Carmela Correct .................................................. 55
  7.1 Preservation of static correctness ................................. 55
  7.2 Preservation of dynamic semantics .................................. 55
  7.3 Dead Objects ....................................................... 56

8 Extending the underlying language ....................................... 57
  8.1 Local Variables .................................................... 57
     8.1.1 Adding local variable support to Carmela .................. 57
     8.1.2 Issues with local variables ................................ 58
  8.2 Exceptions ........................................................ 58
     8.2.1 Re-classification of the exception object ................... 58
     8.2.2 Re-classification of the handler object ..................... 59
     8.2.3 Discussion .................................................... 59

9 Extensions of Fickle ..................................................... 61
  9.1 State Types for Local/Method Parameter Variables .................. 61
     9.1.1 Origins of Aliasing - The Enemy ............................ 61
     9.1.2 FickleIII ...................................................... 63
     9.1.3 Why not fields? .............................................. 64
     9.1.4 Comparison of approaches ................................... 65

10 Testing Carmela .................................................................. 67
  10.1 The Test Suite ....................................................... 67
     10.1.1 The Hive File .................................................. 67
     10.1.2 Test Result Output ........................................... 69

11 Evaluation ...................................................................... 70
  11.1 Conclusions .......................................................... 70
     11.1.1 Is the implementation correct? .............................. 70
     11.1.2 Were the extensions useful? ................................ 70
     11.1.3 Would I do things differently next time? .................. 71
     11.1.4 What was the most challenging part? ....................... 71
     11.1.5 Formal Methods ................................................. 71
  11.2 Software Engineering Aspects ........................................ 71
     11.2.1 Design Methodologies ......................................... 71
11.2.2 Development Tools and Environment ........................................ 71
11.2.3 Code Organization ................................................................. 72
11.3 Achievements .................................................................................. 72
   11.3.1 Carmela ............................................................................... 72
   11.3.2 Formalization ....................................................................... 72
   11.3.3 Test Suite ............................................................................ 73
   11.3.4 Extensions ........................................................................... 73
   11.3.5 Publication of a paper ............................................................. 73
11.4 Further work .................................................................................... 73
   11.4.1 Formalization ....................................................................... 73
   11.4.2 Abstract Methods ................................................................ 73
   11.4.3 Constructors ....................................................................... 73
   11.4.4 Object Tables ..................................................................... 74

A User Guide .......................................................................................... 75
   A.1 Using Carmela .......................................................................... 75
   A.2 Running the test suite ............................................................... 75

A Fickle Grammar used for SableCC .................................................... 76

B Proofs of Static and Dynamic Correctness ........................................... 90

C Test Cases .......................................................................................... 97
   C.1 Static Cases ............................................................................. 97
      C.1.1 Class Declarations ............................................................. 97
      C.1.2 Overriding Members ......................................................... 104
      C.1.3 Expressions - Conditional ............................................... 110
      C.1.4 Expressions - Assignment ............................................... 113
      C.1.5 Expressions - Field Access .............................................. 120
      C.1.6 Expressions - Return Expressions .................................... 123
      C.1.7 Expressions - Methods .................................................... 124
      C.1.8 Expressions - Re-classification ....................................... 128
   C.2 Dynamic Cases .......................................................................... 131
List of Figures

1.1 UML diagram for Adult and Child with superclass Person .................. 7
1.2 Overview of compilation ................................................. 8
4.1 The Symbols for the FlipFlop example. Only salient fields shown ......... 27
4.2 The symbols with their type objects. Only salient fields shown .......... 28
4.3 The objects for phase one. Only salient fields show .................... 29
4.4 The objects for phase two. Only salient fields show .................... 33
4.5 Phase three objects for class Main with only phase two objects for the other classes. Only salient fields show ............................................. 34
4.6 An overview of the type inference process ................................ 34
4.7 Type checking the expression this.flipper = new Off(); ............... 35
5.1 Rough object layout before and after translation .......................... 36
5.2 The example with redirections shown after each of lines 5 to 8 are executed . 38
5.3 The example with current fields shown. Only state after lines 5-7 are executed is shown .......................................................... 39
5.4 The four states of objects .................................................. 40
5.5 Objects after reclassification to D in foo(). ............................. 41
5.6 The translation of this.b1++ ................................................. 41
5.7 Objects of class B, C and Store after line 4 of change() ................. 43
5.8 Objects of class B, C and Store after line 6 of change() ................. 43
5.9 Objects of class B, C and Store after line 6 of change() with the fixed translation 44
6.1 The objects and their references in memory ............................... 48
8.1 Local Variable Declaration Expression .................................... 57
9.1 State Class to State Class Assignment .................................... 66
9.2 Method Call ..................................................................... 66
9.3 Well Formed Programs ......................................................... 66
Chapter 1

Introduction

1.1 Dynamic Object Re-classification

In class-based, object-oriented programming languages such as Java, an object's behavior is determined by its class. Thus, adults who work in the day and children who play are best described through distinct classes Adult and Child with different `daytime()` methods. We might express the situation with a UML diagram as given in figure 1.1 with Person being the superclass of both Adult and Child. However, this approach does not scale when objects change classification. For example, an object of class Child 'growing up' to become an object of class Adult. In [20] a language called Fielder is described that allows objects to change class membership dynamically. Fielder is an imperative class-based language, where classes are types and subclasses are subtypes, and where methods are defined inside classes are selected depending on the class of the receiver. Object re-classification is achieved by explicitly changing the class membership of objects.

![UML diagram for Adult and Child with superclass Person](image)

Figure 1.1: UML diagram for Adult and Child with superclass Person

1.2 My Contribution

This project is about the implementation of a compiler for the language Fielder. The tool developed is a compiler, called Carmela, that takes programs written in Fielder (with file extension .fielder). The program is type checked against the rules given in [20], then rewritten into corresponding java file(s). The files created can then be compiled using the standard Java compiler, javac. Figure 1.2 shows the overall process.

Carmela is comprised of:

- Lexical analyzer and parser generated by SableCC[8].
- Semantic analyzer that checks that the Fielder program is well formed with respect to the rules given in [20].
- Code generator that produces Java code that can be compiled with a Java compiler.

Considerable work was done in transforming the constructs in Fickle into Java constructs. Of particular interest is how re-classification is modelled in the translated Java classes. We represent a re-classifiable object in Fickle with potentially more than one Java object. When a translated re-classifiable object is in one state e.g. Child it will be represented by an object of type Child, the currently active object. When the Child object is re-classified to an Adult, another object in the translated world will be created and this will become the currently active object. When a re-classifiable object is created in the translated world it becomes the anchor for its clients. When (and if) the anchor is re-classified it will redirect its users to the new currently active object. At this point the anchor only acts as redirection for its clients.

A formalization of the mapping from Fickle into Java is given. Furthermore, we theorize about the static and dynamic correctness of this mapping. The process of formalization was iterative and uncovered some subtle problems (later fixed!) with early attempts at a mapping.

Fickle is then extended to include features such as: local variables, state types for method parameters/local variables and exceptions.

A test suite was developed to test Carmela. The suite is capable of producing test results in HTML (for viewing in web browsers) and Latex (for inclusion in this report). It is completely automated.

1.3 Report Structure

The remainder of this report is structured as follows. Chapter 2 introduces class based programming then Fickle. Chapter 3 gives a specification of Carmela and the tools that will be used for development. Chapter 4 gives the description of how the semantic analysis is done in Carmela. Chapter 5 describes the mapping from Fickle into Java, with chapter 7 theorizing about the correctness of the mapping. Chapter 9 gives the extensions to Fickle developed. Chapter 10 describes the test suite and chapter 11 evaluates the project.
Chapter 2

Background

We first give an introduction to class based programming then describe *Fickle*, with a brief overview of various possible implementations.

2.1 Philosophy of Dynamic Object Re-classification

2.1.1 Class-based languages

In a class-based language everything is centered around the class and their description of objects. For a full description of object theory please refer to [2]. A class describes the structure of all objects generated from that class. An example of a popular class based language is Java[13]. The notion of a class in Java ranges from classes detailing fields and methods (code) to classes being partially described with methods and code to be specified by another class. Java also has the concept of a class with no fields or code, this being called an Interface. The class declaration below could be used to describe a Person.

```java
abstract class Person {
    String name;
    int age;

    abstract void awake(State s);
    void sleep(State s) {
        age+=1;
    }
}
```

Note that the class is abstract and no specification of `awake()` is given. This must be provided by some subclass(s) of Person. The method having the same name as the class is a constructor and will be executed when objects of this class are instantiated.

Subclasses and Inheritance

One of the exiting features of a class-based language is the ability to incrementally describe a class. This is done through the mechanism of subclassing and inheritance. Fields from a subclass add to those in the superclass. Methods in the subclass may override those in the superclass. Furthermore the subclass may be obliged to implement or provide code for methods in its superclass. In the Person example below we have the abstract method `awake()` which all non abstract classes must provide code for. Note that any number of classes can subclass a class. However Java restricts the number of classes that a class can subclass itself to one of zero (in which case the class is a
subclass of object). The non-abstract classes declared below are subclasses of Person and both provide code for the abstract method and hence are not declared abstract themselves.

```java
class Child extends Person {
    Toys myToys;

    void awake(State s) {
        if (s.weatherIsGood()) {
            // Play outside with toys
        } else {
            // Sulk...
        }
    }
}

class Adult extends Person {
    Money myMoney;
    Job myJob;

    void awake(State s) {
        if (s.isWorkDay()) {
            myJob.goToWork(this);
        } else {
            myMoney.spend();
        }
    }
}
```

The above example demonstrates how subclassing can be used to give distinguish behavior. The awake() method of a Person depends on the implementing class. In an language such as C distinct behavior would have been derived from case or conditional statements. Naturally, explicit behavioral selection can be done in object orientated languages as well but is strongly discouraged. This Java example demonstrates:

```java
void doSomething(Person p) {
    // Do something with a person

    if (cl instanceof Child) {
        // Do something with Child ...
    } else if (cl instanceof Adult) {
        // Do something with Adult...
    }
}
```

The main problem with explicit conditional behavioral selection is that if new subclasses of Person are added methods such as doSomething(...) would have to be changed. In a large piece of software such changes can quickly become very difficult.
2.1.2 Evolutionary Objects

Having distinct code relevant to an object is advantageous from the point of maintainability and readability but what of object evolution? In the example above how would one model the transition from being a Child to becoming an Adult? For example when the age of a Child reaches a certain predetermined value the transformation to being an Adult could occur. Java has no direct way of expressing such a transformation. Possible solutions are:

Class Consolidation

When an Object may be of class A or B etc. Create another class that consolidates the two classes e.g. AorB. A flag is then required within the object to select the appropriate behavior, for example:

class ChildOrAdult {
    boolean amIaChild;
    Toys myToys;
    Money myMoney;
    Job myJob;

    void awake(State s) {
        if (amIaChild) {
            // Child Code
        } else {
            // Adult code
        }
    }

    void growUp() {
        amIaChild = false;
    }
}

This solution negates many of the advantages to object oriented programming and 'pollutes' one object with another. For example the Adult has no need for Toys. Also if there are a large number of distinct object a class consolidation can result in a large ill defined class.

Handler Object

This solution uses a proxy handler object which forwards requests to the relevant classes. While this keeps a clear separation between classes creating the handler object is extra work for the programmer.

2.2 Dynamic Object Re-classification - A direct solution

While the above ideas on emulating evolutionary objects are interesting they do not address the problem directly. There is a broad separation between the solutions; those that are statically typed and those that are dynamically typed.
2.2.1 Dynamically typed solutions

We give three examples of languages that allow the class ownership of to be changed dynamically without static typing: Smalltalk[10], Python[1] and Lisp[16]. It is common for these languages to give some semblance of type safety through run-time checks, not compile/link-time. For more information on each approach please read the references. Dynamic typing is not discussed further in this report.

2.2.2 Statically Typed solutions

Plain Java - Dynamic Class Loaders

In [19] a detailed description of the Java class loading mechanism is given and how it can be used to facilitate the dynamic loading of a classes into the Java Virtual machine (JVM). The solution is type safe as additional checks are performed at link time. Hence Java ensures that the dynamic class loading will not violate the type safety of the JVM.

However [19] does not address the problems that may arise from changing the definition of a class while running in the JVM, namely:

- Live objects of the class must be migrated because the new class may have more/less instance fields etc.
- A method in the class to be reloaded may be running.

Fortunately, these issues can be to a certain extent bypassed by coding style and organization. By effectively wrapping the 'dynamic' class with a handler object and using reflection it is possible to achieve class reloading. For example:

```java
class Handler {
    private Object toBeReloaded;
    private String className;

    public void invokeMethod(...) {
        Class c = toBeReloaded.getClass();
        Method m = c.getMethod("Meth1", ...);
        m.invoke(toBeReloaded);
    }

    public void dynamicReload(String newClassLoaderLocation) {
        InternalClassLoader loader = new ClassLoader(newClassLoaderLocation);
        Class c = cl.loadClass(className);
        toBeReloaded = c.newInstance();
    }
}
```

This solution, while being type safe, is not a far cry from the handler object described above. Furthermore, it requires more work on behalf of the programmer. However it does address the needs of a server that requires upgrading without being closed down, for example.

Fickle-99 - Original Paper

Fickle-99[20] was the forerunner to Fickle-2000[6] which was presented at FOOL2001. Although both solve the same problem their approaches are different. Conceptually [6] is more complicated
Implementing Fickle

2.2. Dynamic Object Re-classification - A direct solution

than [20].

Fickle-99 has the concept of Mutatable and Immutable classes. Immutable types correspond to standard Java classes and primitive classes. Mutatable types correspond to families of possible classes of which an object may mutate. We now give an example shown in the original paper to demonstrate Fickle-99:

Person peter;

class Person extends Object {
    int byYear;
    [Std, Empl] friend;
    Person set() { peter := this }
}

class Student extends Person {
    int marks;
    String hobby;
    mut Employee employ() {
        int i:=this.marks;
        this!!Employee;
        this.sal := i;
        this;
    }
    id study() { this; }
}

class Employee extends Person {
    int sal;
    dual int fare() { 25 }
    mut Boss promote() {
        this!!Boss;
        this.hobby:= "Trains";
        this;
    }
    mut Student study() {
        this!!Student;
        this;
    }
    id employ() { this; }
}

class Boss extends Employee {
    String hobby;
    dual int fare() { this.sal / 20; }
    mut Employee demote() { this!!Employee; this;
}

The syntax is very similar to Java with the necessary extensions to support reclassification. These constructs relate to method declarations, reclassification operator and declaration/initializers. In Fickle-99 there are four different types of method:

- Standard Methods - Require immutable receiver, example Person set()
- Mutating Methods - Mutate the the class of their receiver, hence requiring mutable receivers, example `mut Employee demote()`.
- Dual Methods - Receiver can be mutable or immutable, example `dual face()`.
- Identity Methods - Return their receiver, example `id study()`.

To declare a variable/field that can mutate (a mutable type) one uses the syntax `ClassN[Class1, Class2, ...]` var1. Where `ClassN` is a member of `[Class1, Class2, ...]` or just `[Class1, Class2, ...]` var2. The former states that var1 is initially of type Class1 but can mutate to any of the `[Class1, Class2, ...]`. In terms of typing for method calls and field access only the current class `ClassN` is considered. With the later declaration all `[Class1, Class2, ...]` are considered. Mutated and Immutable types are not subtypes. For example:

```r
Employee chris; [Employee] john; Boss judie; juddy := new
Boss; chris := judie
```

If `[Employee]` were a subtype of `Employee` then the assignment `john := chris` would be legal, which it is not.

The reclassification operator in `Fickle-99` is of the form `!!classN which (when executing in a mutable context) transforms the receiver to `classN`.

While `Fickle-99` does directly allow dynamic object reclassification it is more complicated and less flexible than its replacement `Fickle-2000`.

### 2.3 Fickle-2000 - Current Version

We now refer to `Fickle-2000` as `Fickle`. `Fickle` is an imperative, typed, class-based, object oriented language similar to Java. However `Fickle` supports a re-classification operation that can change the class membership of an object, similar to that of `Fickle-99`. `Fickle` achieves this behavior by adding two new types of class:

- **state classes** - Describe objects that can be re-classified.
- **root classes** - Are the abstract super class of state classes. Define methods and fields common to their sub classes.

A type system has been developed for `Fickle` by [20] and shown to be sound. One of the aims of this report is to describe an implementation of `Fickle` as one has not been fully developed yet.

Declaration of variable/fields is simpler, than in `Fickle-99`. Instead of the 'bounded' style declarations for mutable objects one declares a variable of a root type. Also method declaration have changed to include a set of root classes called the effect. As shown below.

\[ \text{t m}(t_1 x) \phi \{ e \} \]

The effect \( \phi \), indicates the abstract state classes of all objects that may be re-classified by invoking this method. An example will clarify.

```r
root class FlipFlop {
    int counter; //Assume will be initialized to false
    void flip() {FlipFlop} { } 
}
```

```r
state class On extends FlipFlop {
    void flip() {FlipFlop} { 
```
Implementing *Fickle*

```java
    this.counter++;
    this!!Off; // We are now off!
}
}

state class Off extends FlipFlop {
    void flip() {FlipFlop} {
        this.counter++;
        this!!On; // We are now on!
    }
}

class Main {
    FlipFlop flipper; // We declare to be of type FlipFlop

    void main() {
        this.flipper = new Off();
        while(true) {
            this.flipper.flip();
        }
    }
}

This example models a flip-flop. The type of flipper alternates between On and Off. The class Main declares the field flipper to be of type FlipFlop.

### 2.4 Making *Fickle* programs run

There are three main possibilities for the implementation of *Fickle*. These are:

- **Compilation** - Transformation of the code into native assembly that will be executed directly by the host processor.

- **Interpretation** - Run the *Fickle* programs on some form of virtual machine

- **Preprocessing** - Converting *Fickle* into another high-level language.

#### 2.4.1 Compilation of *Fickle*

This solution would require the transformation of the *Fickle* program into a native binary object for a chosen architecture e.g. Intel. The advantages and disadvantages of this solution are:

- **Advantages** - There are considerable performance gains to be obtained from running native code.

- **Disadvantages** - There is a large amount of runtime support required to facilitate the execution of *Fickle* programs. This support includes runtime allocation and deallocation of objects. Complex method invocation issues with late binding.

With these points taken into account it was decided that a compilation approach would be beyond the scope of the project.
2.4.2 Interpreting Fickle

The execution of a Fickle program on some form of virtual machine would mirror the methodology used in Java. With Java the source text is transformed into byte codes that can be executed on a suitable Java Virtual Machine (JVM) [22]. It is not the Java source that is interpreted but the byte code it has been transformed into by the Java compiler. The byte code is a platform independent representation of a Java program with constructs to support run time memory allocation/deallocation and method dispatch etc. Hence, the following possibilities exists for interpretation of Fickle programs.

Modification of an existing interpreter

An obvious candidate would be the Java Virtual Machine as Fickle is very similar to Java albeit less complicated. Such a solution would require the following steps:

- Extending the byte code syntax used by the JVM to support re-classification. This would require acquisition of the source of a JVM and making the necessary modifications.
- Creation of a compiler that would transform Fickle into the new extended byte code.

We believe that such an approach is disadvantaged from many perspectives. Firstly modification of the JVM will be a non trivial task requiring deep knowledge of its workings. Secondly there would be compatibility issues with the new byte code not working on existing JVMs.

Creating a new Virtual Machine

Firstly it would be required to create an instruction set for the virtual machine. This would have to handle the requirements of object allocation/deallocation, method calling etc. Secondly a virtual machine for the instruction set.

Again this solution would be a large undertaking and not within the time constraints of the project.

2.4.3 Preprocessing

As stated previously, Fickle is deliberately similar to Java in many respects. It could be considered a 'cut down' version of Java with object re-classification. Hence, a solution would be to transform the Fickle source into plain Java source for execution on the JVM. Therefore it would be required to create a pre-processor to convert non-Java constructs, namely re-classification structures into standard Java. Once this has been done the converted Fickle source can be compiled and executed on any JVM.

With a view to keep the project within the time constraints of such a project this solution will be used. Hence a Carmela will transform a Fickle program into a Java program for execution on a JVM. However, it would more appropriate to refer to Carmela as a compiler as it performs semantic analysis.

2.5 Implementation Language

Java will be used as the implementation language for a Carmela. This is reinforced by the elegance and efficiency of the current version of javac [17] designed by Martin Odersky, which is written entirely in Java. Advantages of using Java:

- Java is an object oriented language and so supports an intuitive level of abstraction.
- Java has a large library of classes that contain many useful routines useful in compiler construction.
Disadvantages of using Java:

- There can be performance issues with Java especially the time taken to initialize the virtual machine.
Chapter 3

Specification and Tools

In this section any references to Fickle will imply the incarnation as described in [20]. The phases of the transformation from Fickle to Java will be as follows.

1. Lexical analysis of Fickle source.
2. Parsing of tokens from lexical analyzer and production of an Abstract Syntax Tree (AST).
3. The AST will be type checked in accordance with the typing rules of Fickle.
4. Transformation of Fickle constructs into standard Java constructs.

3.1 Lexical Analysis/Parsing

The word parsing is used loosely to also encompass the lexical analysis of the source, although the two phases are distinct. The lexical analysis as described by [3]:

The lexical analyzer is the first phase of the compiler. Its main task is to read the input characters and produce a sequence of tokens.

With the definition of parsing:

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar of the source language.

The task of generating a lexical analyzer and parser are often done using third party tools, such as Lex/Yacc. These tools take a grammar in a predefined format usually BNF and generate code for a lexer/parser that can be used as a front for the compiler. We believe that such a generation tool should be used, furthermore the code produced should be Java. By producing Java code it will integrate with the rest of the project, as it too will be written in Java. A critique of current lexer/parser generators that produce Java is given below.

3.1.1 Selection of lexer/parser generator

This section focuses on the selection of a tool for parser generation in Java.

The traditional tools for generating lexers and parser for languages such as C were Lex and YaCC. With the arrival of C++ came PCCTS (Purdue Compiler Construction Tool Set). Naturally both tool sets have been ported to Java and are called JLex/CUP and ANTLR respectively. A completely new parser generator named SableCC[8] is also available offering a very modern design pattern orientated approach. Please refer to [9] for more information relating to design patterns.

The following parser generators where considered for use with the Fickle compiler. The exception being the Java port of Lex/YACC, JLex/CUP which is critiqued in [8] and is included for completeness here.
Implementing *Fickle*

### 3.1. Lexical Analysis/Parsing

**JLex/CUP**

JLex is a Java port of the popular Unix tool Lex. Lex is used to produce a lexer for a given language specification. CUP[11] (Constructor of Useful Parsers) is a Java port of the Unix tool YACC. CUP will produce an LALR(1) parser. Combined the tools can be used to create a complete parser or a front-end to a compiler.

**Advantages:**

- JLex supports macros to simplify the specification of complex regular expressions.
- The parser generated by CUP is LALR(1) which generally means a larger selection of grammars can be used namely LL(k) grammars are subsets of LALR(k) grammars. Hence left recursive grammars can be used.
- The source code to JLex and CUP are available.

**Disadvantages:**

- Java supports the 16bit Unicode standard in source files, JLex only supports 8bit characters. Although Unicode support is not essential in Fickle, it may limit future extendability.
- It is up to the programmer to link the code produced by JLex to that of CUP.
- CUP offers no support for building an Abstract Syntax tree and leaves the responsibility to the user.

The major drawback of CUP is that it does not produce an Abstract Syntax Tree (AST) and this would make it very difficult to create a multi pass compiler for *Fickle*.

**ANTLR (ANother Tool for Language Recognition)/JavaCC**

ANTLR is a port of a C++ parser generator tool Purdue Compiler Construction Tool Set PCCTS[18]. It was ported to Java by [12]. Sun Microsystems have developed a similar tool called JavaCC, the only difference being copyright issues. It was developed in response to the complexity of using Lexx/Yacc. It generates LL(1) recursive descent parsers.

**Advantages:**

- Both tools have a strong integration between the parser and the lexer.
- Can be used to generate an Abstract Syntax Tree (AST).
- Support for Extended BNF.
- The tools are popular and well supported.

**Disadvantages:**

- The class of grammars supported by the tools LL(1) cannot be left recursive. Hence grammatical transformations may need to be done.
- The integrity of the AST is left to the programmer. The AST does not use object oriented methodologies to protect against degenerate transformations.
SableCC

SableCC [8] was developed at McGill University, Montreal as a new compiler framework for building compilers in Java. The most notable feature is that design patterns have been used throughout to take advantage of the Java type system. Furthermore SableCC specifications of grammar do not contain any action code, action code is delegated to new classes.

Advantages:

- The AST produced is strictly typed and hence self-preserving.
- The lexer and parser are well integrated.
- Support for a larger class of grammars, LALR(1).

Disadvantages:

- The ‘strictness’ of the AST can result in complex visitor based code to infer things about the parsed source files.
- By not allowing subclassing of the nodes (because they are declared final) it can be difficult to associate data about a node during later phases of compilation.

We believe that SableCC will be the best tool for generation of lexer/parser. Its support for a larger class of grammars coupled with good design principles makes it a good choice.

3.1.2 Specification for Lexer/Parser

The SableCC tool be used to generate a parser/lexer for the Field language given below in BNF.

| progr  ::=  class*
| class  ::=  [root | state] class c extends c field* meth*  
| field  ::=  type f  
| meth   ::=  type m (type x) eff e  
| type   ::=  bool | c  
| eff    ::=  c*  
| e      ::=  if (e) e else e | var := e | e | sVal  
| var    ::=  e | e.f  
| sVal   ::=  true | false | null  
| c      ::=  c | c' | c'' ...  
| f      ::=  f | f' | f'' ...  
| m      ::=  m | m' | m'' ...  

Important Considerations

- Use of an appropriate naming scheme when naming productions in the specification file for SableCC. This is a consideration because production names become class names. Large ‘unobvious’ class names will be difficult to work with in the source code.

Secondary Considerations

- Create the specification with a view for possible extensions to the grammar at later stages.
- Make the grammar more Java like, with semi colons etc. This is more of a cosmetic consideration.
3.2 Attribution/Semantic Analysis

Once a Fickle source file has been parsed an Abstract Syntax Tree will have been produced. This tree must have a number of semantic checks performed on it before code generation can proceed. These checks address the following questions:

- **Type Checks** - Can a proof be created using the type rules given for the given Fickle source?
- **Uniqueness Checks** - Are all the class names unique. Furthermore, are field and method names unique with respect to each other? More formally:

\[
\forall c : p = p_1 \text{ [root | state] class c extends } c' \{ \ldots \} p_2, \quad p = p_3 \text{ [root | state] class c extends } c'' \{ \ldots \} p_4 \implies p_1 = p_3, p_2 = p_4
\]

\[
\forall f : p = p_1 \text{ [root | state] class c extends } c \{ \text{def} s_1 \ t \ f \ \text{def} s_2 \} p_2, \quad p = p_1 \text{ [root | state] class c extends } c \{ \text{def} s_3 \ t' \ f \ \text{def} s_4 \} p_2 \implies \text{def} s_1 = \text{def} s_3, \text{def} s_2 = \text{def} s_4
\]

\[
\forall m : p = p_1 \text{ [root | state] class c extends } c' \{ \text{def} s_1 \ t \ m(t_1) \phi \{ e \} \ \text{def} s_2 \} p_2, \quad p = p_1 \text{ [root | state] class c extends } c' \{ \text{def} s_3 \ t' \ m(t_1) \phi \{ e' \} \ \text{def} s_4 \} p_2 \implies \text{def} s_1 = \text{def} s_3, \text{def} s_2 = \text{def} s_4
\]

- **Well-formed related checks** - Are special keywords such as this used as member names appropriately.

- **Well-formed programs** - Is the overall program well formed, are there any cycles in the class graph? More formally:

\[
\vdash P \diamond_a
\]

\[
\vdash P \diamond_a
\]

\[
C(P, c) |\text{root | state} | \text{class c extends } c' \{ \ldots \}
\]

\[
\forall f : F(D(P, c, f) = t_0 \implies P \vdash t_0 \diamond_{vt} \text{ and } F(P, c', f) = \text{udf}
\]

\[
\forall m : MD(P, c, m) = t m(t_1) \phi \{ e \} \implies P \vdash t \diamond_{vt}
\]

\[
P \vdash t \diamond_{vt}
\]

\[
P \vdash t_1 \diamond_{vt}
\]

\[
P \vdash \phi \diamond
\]

\[
P, t_1 \times c \text{ this } \vdash e : t' \parallel c'' | \phi
\]

\[
P \vdash t' \leq t
\]

\[
\phi' \subseteq \phi
\]

\[
R(P, c) = R(P, c')
\]

\[
M(P, c', m) = \text{udf} \text{ or } (M(P, c', m) = t m(t_1) \phi' \{ e' \} \text{ and } \phi \subseteq \phi')
\]

\[
P \vdash c \diamond
\]

The derivation of an answer to the above questions broadly constitute to the semantic analysis of the abstract syntax tree. That is, calculating the type of every node of the abstract syntax tree or assigning a type error if there is a problem.

### 3.2.1 Specification of Semantic Analyzer

The semantic analyzer is arranged into two phases: symbol collection and attribution. Data structure for the following concepts will be required:

- **Symbols** in the source file for example, class, method, field etc.
- **Types**
• Symbol Table

• Scope, regions of visibility within classes. These act as symbol storage.

Symbol Collection

The procedure for placing symbols into the symbol table in an object-oriented language differs from that of an imperative language. The main complication being the consideration of inheritance and overriding. With this in mind the symbol collection has been arranged into two phases. Each class (entered into the symbol table) will have a scope that contains its members (fields and methods). However before the symbols contained within a class can be entered into the scope, the supertype must be determined. The reason for this is that when processing a class definition, it will be necessary to assert:

• Overridden methods have the correct signature.

• Field definitions are not overridden, Fickle does not support field hiding.

• The modifiers of the class are appropriate with respect to its supertype. For example the superclass of a state class must state or abs-state.

Once all the symbols have been entered into the appropriate place and in the correct order, type checking of expressions can commence.

Type Checking of expressions

All the expressions within the source must be type checked. Incidentally expressions can only occur within the body of methods. In [20] there are typing rules given for expression, there is one rule for each type of expression. The rules are of the form:

Condition$_1$ for expression$_1$
Condition$_2$ for expression$_2$
......
Condition$_n$ for expression$_n$.

Condition$_{n+1}$ holds for expression$_n$.

For the condition below the line to hold all the conditions above the line must hold. Determining whether a condition holds may result in application of other rules. Hence checking the rules is essentially a recursive process. Below is an example of a rule in Fickle for conditional statements:

\[
\begin{align*}
\text{P,F} & \vdash e : \text{bool} \mid c \mid \phi \\
\text{P,F}[\text{this} \rightarrow c] & \vdash e_1 : t_1 \mid c_1 \mid \phi_1 \\
\text{P,F}[\text{this} \rightarrow c] & \vdash e_2 : t_2 \mid c_2 \mid \phi_2 \\
\text{P} & \vdash t_1 \leq t_2 \text{ for } i \in 1, 2 \\
\text{P,F} & \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t \mid c_1 \cup p \cup c_2 \mid \phi \cup \phi_1 \cup \phi_2
\end{align*}
\]

The rules can be obtained from [20]. The process of type checking will be one of recursive descent through the AST representing the expression. For example consider the expression:

\[
\text{if } (x.m(\text{this})) \{ 1; \} \text{ else } \{ 2; \}
\]

At each node a rule will be applied. Any type information needed to apply the rule will be obtained by further invocations of the type checker on the children of the node in question. Hence, for the above, an invocation of the type checker for conditional statements would occur, followed by a recursive call for the test expression \(x.m(\text{this})\). Followed by calls for the true and false branches 1 and 2 respectively. The representation of types will also be required to store the effects of expressions as these will be required during the calculations.
Important Considerations

- The semantic analyzer must report the presence of errors and where they are located.
- Make every effort to continue in the presence of errors unless they are absolutely catastrophic.

Secondary Considerations

- The semantic analyzer should operate in a reasonable amount of time and use a reasonable quota of memory.
- Production of debug information would be useful.
- Some information such as effects will not be required after analysis, this should be deleted to free memory.

3.3 Code Generation

Once the abstract syntax tree has been type checked and there are no type errors code must be produced. As discussed above, the Fickle re-classification features will be transformed into regular Java constructs. The transformed source files can then be compiled using the Java compiler javac.

3.3.1 Specification of Code Generation

There are many possible solutions for emulating the dynamic re-classification of Fickle objects in Java. These include:

- Double Dispatch - All references to mutable objects will point to a top-level object. The top-level object holds references to the Fickle state class object.
- Reference Counting - Works by keeping track of every reference to a Fickle object so they can be updated when the object mutates.

With reference counting field and method calls will be more efficient as they will be directly on the object in question, rather than through a top level object. However object mutation would be more time consuming than in the double dispatch case. With these factors taken into consideration the double dispatch method is favored.

Double Dispatch

With double dispatch objects of state class will be represented by two objects, hence the name double dispatch. There will be an outer ‘wrapper’ class that will represent the root classes. This class will contain a field that points to a state class (a subclass of the root class). When re-classification occurs the field will be overwritten with a new state class object. Any method calls and field accesses will be forwarded through the wrapper class to the field containing the reference to the state class.

Important Considerations

- The code must be correct and accurately model the operational behavior of Fickle.

Secondary Considerations

- The code produced should be as readable as possible
- Use an appropriate naming scheme for automatically generated classes. (This could be configurable).
3.4 Miscellaneous issues

No entry point into a *Fickle* program has been specified yet. We propose a *special* class named `Main` that acts as an entry point for execution. The `Main` class has the form:

```java
class Main {
    fields

    void main() {

    }
}
```

Furthermore, the syntax of *Fickle* could be extended to allow blocks of code to be *passed through* unchecked by the Java compiler. This will allow for basic input/output via Java’s `System.in/System.out` mechanisms. Pass through declarations have the form:

```java
Java {
    // Potentially any Java code here....
}
```

These two extensions will allow for effective testing of *Fickle* programs.
Chapter 4

Semantic Analysis

We now detail the design of Carmela's semantic analyzer. Each phase of the analysis is detailed below with a description of how it is implemented in the Carmela compiler. To illustrate the issues we follow the Flip/Flop example through the compiler.

```java
root class FlipFlop {
    int counter; // Assume will be initialized to false
    void flip() {FlipFlop} { }
}

state class On extends FlipFlop {
    void flip() {FlipFlop} {
        this.counter++;
        this!!Off; // We are now off!
    }
}

state class Off extends FlipFlop {
    void flip() {FlipFlop} {
        this.counter-=2;
        this!!On; // We are now on!
    }
}

class Main {
    FlipFlop flipper; // We declare to be of type FlipFlop
    
    void main() {
        this.flipper = new Off();
        while(true) {
            this.flipper.flip();
        }
    }
}
```

4.1 Semantic Analysis

Once the Abstract Syntax Tree (AST) has been generated we analyze it to check that it represents a well formed program. This process consists of:
Implementing Fickle

4.1. Semantic Analysis

- Find all the classes declared in the source files.
- Determine the hierarchical relationships between these classes.
- Determine members of each class.
- Type check the bodies of the methods in these classes.

The main classes implementing the above:

- SymbolCollection - For each of the entities in the Fickle program i.e. classes, methods and fields, builds an internal representation through an object of class Symbol explained in section 4.1.1.

- Attribution - Infers the types of AST nodes. There are two additional helper classes used here:
  - NameResolver - Given a string identifying an entity return the internal representation of it.
  - TypeProcessor - Helper class for the Attributor.

There is a number of data structures that are used by all of the above classes, these are detailed next.

4.1.1 Core Data Structures

Compiler Contexts/Environments

There is a certain amount of state information required during the semantic analysis. This includes which class/method we are currently processing. To represent this state we use the concept of a context object. There is a duality in the use of context objects in that the same objects can be used as contexts for both class and method processing. The contexts are linked together to form a nesting. The most important items stored are references to the AST node representing the current class/method respectively. Note that the contexts are not essential as such, but represent a convenience. In Carmela they are used to determine which class/method is currently been processed. It is the task of the SymbolCollection class to act as factory for production of these contexts. As the AST is being traversed the context is carried along and used as required. The SymbolCollector produces a list of contexts for use by the Attributor, so that it knows which classes to type check. So in this case they are used to describe a unit of work with all the information to undertake that work.

Symbols

The entities within a Fickle program are defined as being the classes, methods and fields. For each of these entities Carmela builds an internal representation through an object of class Symbol. For example, consider the class FlipFlop:

```java
class FlipFlop {
    int counter;
    void flip() {FlipFlop} { }
}
```

Figure 4.1.1 shows the symbol objects created. Note in particular an object of type ClassSymbol representing the class FlipFlop, VarSymbol for counter and MethodSymbol for flip(). The following information is stored within Symbol objects:

- Name - A string representing the name given in the source for the entity. For example, the name field of the Symbol for the counter field would be 'counter'.

26
Flags - Are used to mark the Symbol as being in a particular state. For example, Symbols are locked during symbol collection.

Owner - Methods and Fields are 'owned' by the class in which they are defined. This is useful when a VarSymbol or MethodSymbol has been returned by NameResolution and we need to determine which class it was defined in. Classes do not have an owner as they are top-level. Note that if Fiddle allowed inner classes then it would be owned by its enclosing class.

Kind - An optimization to broadly identify the 'kind' of the Symbol. For example, is the symbol a method, variable etc. The main kinds are:
- NIL - A 'bottom' kind. Used during Attribution.
- MTD - A method.
- VAR - A field, formal parameter or local variable.
- TYP - A type e.g. FlipFlop.

Completer - Explained in section 4.1.2.

Type - A reference to a Type object representing the type of the Symbol. Types are explained next.

The Error class is a subclass of Symbol and represents semantic errors.

Types
This class represents an abstraction of types and signatures. Every entity is allocated a Type object or one of its subclasses. The Type classes used by Carmela are:

- ClassType - Classes are types in Fiddle. Objects of ClassType and its subclasses represent these types. There is a field superType representing the superclass of the class. ClassType objects provide methods to determine whether a type is a subtype of another and the least root class of a class.

- StateClassType - A subclass of ClassType for state classes.

- RootClassType - A subclass of ClassType for root classes.

- MethodType - A method signature composed of return type, formal parameter types and effect.
• ErrorType - Represents an error type, this is the type of the error symbol.

• void, null, boolean, int, noType - Primitive types, these are static instances of the Type class.

Finally the class FickleType represents two of the components of the Fickle type triple. Namely the first and third components. Type checking in Fickle has the form:

\[ P, \Gamma \vdash e : t | c [\phi] \]

Where \( P \) is the Fickle program, \( \Gamma \) is the environment, \( t \) is the type of possible values returned by the evaluation of \( e \), the class \( c \) is the class of this after the evaluation of \( e \), and \( \phi \) conservatively estimates the re-classification effect of the evaluation of \( e \) on objects. We represent \( P \) through the AST, \( \Gamma \) through the Scope and \( t \) and \( \phi \) through FickleTypes. The component \( c \) is stored in the state of the Attribute.

![Figure 4.2: The symbols with their type objects. Only salient fields shown.](image)

Figure 4.2 shows the FlipFlop example from above with the added Type objects for each of the symbols.

**Scope**

Scope represents a region of visibility within the program. Each ClassSymbol has a scope that represents its members (fields and methods) and each method has a scope representing its parameters and local variables. The scope is represented by a Java `HashMap`.

### 4.1.2 Symbol Collection

We now show the process of building a representation of the source program in Carmela. An important concept in this process is completion on demand.

**Completion on demand: Completer objects**

Each symbol in Carmela can have a Completer object. This object is given the responsibility of finishing off the representation of the symbol. In Carmela completer objects are only used for ClassSymbols but the concept is generic. When information is required from a symbol this is a cue for it to complete itself before returning the information required (as it will not exist until the symbol is completed!). Note that the completion of one symbol may cause the completion of another and so on. This technique is used when discovering the relationships between classes.
Phase 1 - Finding the classes

The first task in symbol collection is to discover which classes are in the source program. This is easily determined as they are just nodes in the AST that are discovered as the tree is visited. For each *Fickle* class a *ClassSymbol* is allocated. The constructor in turn creates an instance of a *ClassType*, *RootClassType*, or *StateClass* as appropriate. The two hold references to each other. Furthermore each symbol is given a completer. All the newly created *ClassSymbol* objects are placed in a list for the next phase. There is no subclass relationship in place yet. Figure 4.3 shows the objects created for the Flip/Flop example. *Carmeda* will report an error if the class names are not unique. If this phase is successful we have established that the class names in the *Fickle* program are unique. More formally:

\[
\forall c : \quad p = p_1 \quad \text{root | state } \quad \text{class } c \text{ extends } \quad \text{class } c' \{ ... \} \quad p_2,
\]

\[
p = p_3 \quad \text{root | state } \quad \text{class } c \text{ extends } \quad \text{class } c'' \{ ... \} \quad p_4
\]

\[\implies p_1 = p_3, p_2 = p_4\]

![Diagram of the objects for phase one. Only salient fields show.](image)

**Figure 4.3:** The objects for phase one. Only salient fields show.

Phase 2 - Building the class graph

Once all the classes have been allocated the relationships in terms of inheritance must be determined. The use of the *Completer* objects is essential at this stage. The process begins by invocation of the *complete()* method of the first class symbol on the list. This in turn starts the completion of the symbol. The completion process involves the following:

1. The class symbol is marked as *LOCKED*. This is used to detect cycles in the class graph.
2. The AST node for the class symbol is checked for the presence of an *extends* clause.
3. If there is no superclass then the superType field for the type of the class symbol is set to Type::noType a static member of Type.

4. If there is no superclass but the flags indicate that the class is state, then an error is produced indicating the presence of an orphan state class.

5. If there is an extends clause the Attrbutor is invoked to determine the type of the extends clause.

6. Once the type of the superclass has been found (note it could be a Type::typeError) the superType field of the type of the current symbol is set.

7. In the process of determining the supertype of the current class the complete method of the Symbol for the supertype will have been invoked and so on.

8. If at any time a supertype is found and it is already LOCKED then there is a cycle in the class graph and an error is produced.

9. The class symbol is UNLOCKED and placed on a queue ready for member collection.

At this point we have determined the class graph and reported any problems in its structure. More formally, we have established the ⊑ relation and determined whether ⊢ P ⊑ u holds for the Fickle program P:

\[ \vdash P \bowtie u \]

\[ \vdash P \bowtie u \]

\[ P \vdash \text{Object} \sqsubseteq \text{Object} \]

\[ P \vdash c \sqsubseteq c' \]

\[ P \vdash c' \sqsubseteq c'' \]

\[ P \vdash c \sqsubseteq c'' \]

\[ \forall c, c' : P \vdash c \sqsubseteq c' \text{ and } P \vdash c' \sqsubseteq c \implies c = c' \]

\[ \mathcal{C}(P, c) = \text{class } c \text{ extends } c' \{ \ldots \} \implies \mathcal{C}(P, c') = \text{class } c' \ldots \]

\[ \mathcal{C}(P, c) = \text{state class } c \text{ extends } c' \{ \ldots \} \implies \mathcal{C}(P, c') = \text{state class } c' \ldots \]

The next stage is to determine the members of the classes. It should be clear that the class graph must be known before members can be collected so that we can correctly determine overridden members. Diagram 4.4 shows the objects with the newly found links for this phase.

Phase 3 - Member collection

The final stage of collection is the acquisition of methods and fields within the declared classes. This is done by the ClassMemberCollection class. Each of the field and method entities in the AST is given a VarSymbol and MethodSymbol respectively. The symbols are placed in the scope of the class being analyzed. Note that the method bodies are not analyzed at this point. The Attribution class is used to determine the types of fields and methods returning the appropriate instance of a Type object. The only checks done at this point are that of uniqueness of method and field names. Therefore if this phase is successful we have established that the program has unique definitions. More formally we have established, for Fickle program P that \( \vdash P \bowtie u \), where:
Implementing *Fickle*

4.1. Semantic Analysis

\[ \forall c \colon p = p_1 \rightarrow \begin{array}{l}
\text{root} \quad \text{state} \quad \text{class} \quad c \text{ extends } c' \{ \ldots \} \quad p_2, \\
p = p_3 \rightarrow \begin{array}{l}
\text{root} \quad \text{state} \quad \text{class} \quad c \text{ extends } c' \{ \ldots \} \quad p_4 \\
\end{array} \\
\Rightarrow \begin{array}{l}
p_1 = p_3, p_2 = p_4 \\
\end{array} \]

\[ \forall f \colon p = p_1 \rightarrow \begin{array}{l}
\text{root} \quad \text{state} \quad \text{class} \quad c \{ \text{def}_1 t \ f \ \text{def}_2 \} \quad p_2, \\
p = p_1 \rightarrow \begin{array}{l}
\text{root} \quad \text{state} \quad \text{class} \quad c \{ \text{def}_3 t' \ f \ \text{def}_4 \} \quad p_2 \\
\end{array} \\
\Rightarrow \begin{array}{l}
\text{def}_1 = \text{def}_3, \text{def}_2 = \text{def}_4 \\
\end{array} \]

\[ \forall m \colon p = p_1 \rightarrow \begin{array}{l}
\text{root} \quad \text{state} \quad \text{class} \quad c' \{ \text{def}_1 t \ m(t_1 x) \ \phi \{ e \} \ \text{def}_2 \} \quad p_2, \\
p = p_1 \rightarrow \begin{array}{l}
\text{root} \quad \text{state} \quad \text{class} \quad c' \{ \text{def}_3 t' \ m(t'_1 x) \ \phi' \{ e' \} \ \text{def}_4 \} \quad p_2 \\
\end{array} \\
\Rightarrow \begin{array}{l}
\text{def}_1 = \text{def}_3, \text{def}_2 = \text{def}_4 \\
\end{array} \]

\[ \vdash P \circ \alpha \]

At this point the source program will be represented in *Carmales* as shown in Figure 4.5. The symbol collection is complete and type checking of the method bodies can proceed. All the class symbols have been placed on a queue awaiting their method bodies being analyzed.

### 4.1.3 Attribution

The Attribution class is responsible for taking nodes of the AST and a context and inferring a type. If it finds a type error, it produces a type error object and continues as best possible. The main points of interest in this process are: name resolution and error recovery and detection. It must be noted that at this point in the process the structure of the class graph and class members are known. This information was established with the aid of the Attribution class. However, now the method bodies must be analyzed in accordance with the *Fickle* typing rules. Figure 4.6 gives a bird’s eye view of the process.

#### Name Resolution

Given a string and a kind the `NameResolver` returns the associated `Symbol` object. The kind is used to narrow the search space, for example if the kind is `TYP` (for a type) then the given string must identify a class. Therefore the search can be restricted to the symbol table.

If the kind of the identifier is `VAR` or `MTD` for variable or method the search is slightly more complex. The search for variables starts with the local variables and formal parameters of the current method body, followed by fields in the current class. These search regions are called `type sites`. From the type site it is possible to extract a scope and hence a repository of defined `Symbols`. Type sites are a natural way of looking at name resolution take for example the contrived expression.

```
this. a1. b2. c3. d4. foo;
```

Firstly a search would be done in the type site for `this`, which is the current class (the current class can be extracted from the context). Once `a1` has been found a search in the type site for `b2` would commence and so on. This way, name resolution is broken into stages and this in turn makes error detection much easier. In this example because of the initial `this` the search would ignore local variables and formal parameters. However, if the expression had been `a1. b2. c3. d4`, the search would have started with locals i.e. at the scope of the method body. Naturally when searching for fields the search must go up through the class hierarchy until the field is found.

#### Name Resolution Errors

When name resolution fails to find a `Symbol` for a given entity a `NameResolutionError` is given. `NameResolutionErrors` are a subclass of `Symbol` that have the extra capability of being able to report what type of naming error they represent. There are the following `NameResolutionErrors`:

- `VarNotFound` - When a formal parameter or field was not found.
• **TypeNotFound** - When a type is not found. For example, declaring something to be of a type that does not exist.

• **MethodNotFound** - When a method is not found with the correct signature or name.

• **NoSuch** - When a field is accessed without the this.

Before the NameResolver returns a Symbol back to the Attributor it is checked to see whether it is a NameResolutionError. In this case the reportError() method is called and an error message produced. Naturally the message is appropriate for the type of error and also includes the position.

### Analysis of method bodies

The final stage of the semantic analysis is the type checking of the method bodies. We will illustrate with a statement from the Flip/Flop example:

```java
this.flipper = new Off();
```

As explained earlier the Attributor is responsible for inferring types for nodes of the AST. These types are FickleTypes in that they contain a type and an effect. In some contexts the type of an expression is expected to be a sub-type of another. Therefore we infer the type of the expression, then check to see whether this type is a sub-type of the required type. For example, using the type rules for `Fickle` we obtain the following type for the above expression.

```
P, f ⊩ this: Main | Main | φ
P, f[term]Main ⊩ new Off(): Off | Off | φ
\( \mathcal{P}(P, f, \phi) = \text{Flipper} \)
P ⊩ \text{Flipper}
```

Here we derive the type of the left hand side of the expression to be `Flipper` and the right hand side to be `Off`. Furthermore we require the right hand side to be sub-type of the left hand side, \( P ⊩ \text{Off} \leq \text{Flipper} \). The Attributor firstly infer the type of the left hand side then the right hand side. A parameter to the call for inferring the type of the right hand side would be the required sub-type `i.e. Flipper`. Once the type of the right hand side had been determined a check is performed to ensure that it is indeed a sub-type of `Flipper`. If not, a type error will be produced.

Figure 3.7 shows the course of events as the expression is type checked. The boxes on the right show the expected type and if used in the next recursive call the returned type of the sub expression. If the symbol collection and attrribution are successful then the program is well formed.

More formally we would have established that \( ⊢ P \circ \), where:

\[
\begin{align*}
\vdash P & \circ_a \\
\mathcal{C}(P, c) &= [\text{root} | \text{state}] \text{class c extends c' }\{\ldots\} \\
\forall t: \mathcal{P}(P, c, t) = t_0 & \implies P \vdash t_0 \circ_{vt} \quad \text{and} \quad \mathcal{P}(P, c', t') = \text{Udf} \\
\forall m: \mathcal{M}(P, c, m) = t \mathcal{m}(t_1 x) \left\{ e \right\} & \implies \\
& P \vdash t \circ_{vt} \\
& P \vdash t_1 \circ_{vt} \\
& P \vdash \phi \circ \\
& P, t_1 x : c \text{ this }\vdash e: t \circ e' \circ d' \\
& P \vdash t' \leq t \\
& d' \subseteq d \\
& \mathcal{R}(P, c) = \mathcal{R}(P, c') \\
& \mathcal{M}(P, c', m) = \text{Udf} \quad \text{or} \quad (\mathcal{M}(P, c', m) = t \mathcal{m}(t_1 x) \left\{ e \right\} \quad \text{and} \quad \phi \subseteq d') \\
\vdash P \circ \\
\forall c: \mathcal{C}(P, c) \neq \text{Udf} & \implies P \vdash c \circ \\
\vdash P \circ
\end{align*}
\]

32
Figure 4.4: The objects for phase two. Only salient fields show.
Figure 4.5: Phases three objects for class Main with only phase two objects for the other classes. Only salient fields show.

```
void main() {
    flipper = new Off();
    while(true) {
        flipper.flip();
    }
}
```

Figure 4.6: An overview of the type inference process
Figure 4.7: Type checking the expression `this.flipper = new Off();`
Chapter 5

Code Generation

After successful semantic analysis Carmela generates Java code representing the Fickle program. We first demonstrate and justify the principles of code generation with small examples, then we return to the Flip/Flop example and describe how it would be translated.

5.1 Overview

The code generator takes each Fickle class and translates it into a Java class. This translation preserves the inheritance hierarchy. We view Fickle as subset of Java enriched with reclassification. Therefore, one of the aims of translation is to map the reclassification into Java, i.e. model the reclassification operator. We propose a translation that uses a minimal amount of typing information, compared with [5] that uses the type of an expression for the translation of certain constructs.

5.2 Object Layout - A rough guide

Each Fickle class is translated into one Java class. Furthermore, object layout remains unchanged apart from the extra code required to support reclassification. This is shown in figure 5.1. Note that fields are left unchanged but method bodies are translated.

![Diagram showing class B extends A with fields and methods before and after translation](image)

Figure 5.1: Rough object layout before and after translation

5.3 The changing face of this

We use the term standard class to denote a class that is not root or state. We use the term client to denote the user of an object. Potentially a standard class can have some subclasses that are reclassifiable. Consider the following example:

36
class D { // Standard class
    int counter;
    void foo() {A} { counter++;}
}

root class A {
}

state class B extends A {
    void foo() {A} { counter+=2; this!!C;}
}

state class C extends A {
    void foo() {A} { counter+=4; this!!B;}
}

Now consider the following example annotated with line numbers:

1. D d = new D();
2. d.foo(); // method body in D executed
3. d = new A();
4. d.foo(); // method body in D executed
5. d = new B();
6. d.foo(); // method body in B executed
7. d.foo(); // method body in C executed
8. d.foo(); // method body in B executed

The first two calls to foo() execute the code in class D. But the code executed in the last three calls to foo() alternates between the implementations in B and C. Because each Field class is represented by one Java class each reclassification by foo() will create a new object on the heap (more on this in the next section). Therefore, the translation must ensure that the method calls and field access made by clients are directed to the correct object. Hence an extra level of indirection will be required when accessing the members of an object to ensure the right one is used.

5.3.1 Follow me for freshness

The objects referred to in this section are in Java. When reclassification occurs we don't find all clients of the reclassified object and point them to a new object. Instead we guide clients to the correct object through an extra level of indirection. Looking at the example above, after line 5 d is pointing to an object of class B. After line 6 the object pointed to by d has been reclassified to an object of class C. The next call to foo() of line 7 must execute the code in class C. We must redirect the call because d is still pointing to an object of class B. So we have:

d. <Some Redirection> .foo();

Figure ?? shows the redirection. The dotted line emanating from the end of the d pointer shows where method calls and field accesses will be redirected to. So as shown after line 6 there is a redirection pointing to an object of class C. Therefore the call to foo() in line 7 will execute the foo() method declared in class C. The objects in gray are never again used by the client for method call or field access. They are stale and will eventually be garbage collected, with exception of the first object that is still pointed to by d and is acting as an anchor.
So we have the concept of a currently active object. When an object is first created it is the currently active object. It will execute the member accesses of its clients. If this object is reclassified another object will be created of the appropriate type, this object will be the currently active object. For example in figure 3.8 after line 5 the first object is the currently active object and has type B. After line 6 the first object is no longer the currently active object. The newly created object of class C (the result of the reclassification) becomes the currently active object.

However, clients are always pointing to the first object i.e. the one created by new. We call this first object the anchor. Any references to an object within the Fickle code will be translated into references to an anchor object. The translation of this is slightly different and detailed in 5.4. The anchor must provide a means of accessing the currently active object. When a class is translated it is given an extra field called current that will hold a reference to the currently active object. When an object is first created the current field points to the object itself. The type of the field current will be the same as the class it is declared in. The next section explains how the current field is kept up-to-date.

Now all client accesses to an object’s fields and methods are through the current field of the anchor object. So the example above would be translated thus:

1. D d = new D();
2. d.current.foo(); // method body in D executed
3. d = new A();
4. d.current.foo(); // method body in D executed
5. d = new B();
6. d.current.foo(); // method body in B executed
7. d.current.foo(); // method body in C executed
Implementing Fickle

5.3. The changing face of this

8. d.current.foo(); // method body in B executed

....

....

Diagram 5.3 shows the state of affairs after execution of lines 5-7. In line 6 the access $d.current$ will point back to the object itself, i.e. $d = d.current$ as expected, because no reclassification has taken place yet. So $d.current.foo()$ will execute the method $foo()$ in class B. However, after execution of line 6, a reclassification has taken place and the current field of the anchor object points to an object of class C. Therefore the access $d.current$ of line 7 now correctly points to and hence execute the method $foo()$ in class C.

![Diagram 5.3: The example with current fields shown. Only state after lines 5-7 are executed is shown.]

To summarize, an object in the translated Fickle can be in four different states:

- Anchor and currently active object - An object is in this state if it has been created by a new expression within the Fickle resource program and has not been reclassified. For example, consider $D d = new B()$: the object pointed to by $d$ is both the anchor and the currently active object.

- Anchor - An object is in this state if it has been created by a new expression within the Fickle resource program and has been subsequently reclassified. Consider the state when execution reaches line 7 in the example above: $d$ points to an object that is an anchor but is not the currently active object. The current field points to another object.

- Currently active object - If an object is created as the result of reclassification and can be reached by the current field of the anchor

- Stale - The only way for an object to be in this state is for it to have been created by a reclassification and then a subsequent reclassification has occurred. Hence it is not reachable by any means.

Diagram 5.4 shows objects in each of the states above. In the translated code, to ensure a client is using the currently active object, all field and method accesses are through the current
field of the receiver. So a \textit{Fickle} field \texttt{e.f} is translated into Java as \texttt{e.current.f}. Similarly for \texttt{e.m(e0)} we have \texttt{e.current.m(e0)}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure54.png}
\caption{The four states of objects.}
\end{figure}

### 5.4 The translation of this

During the execution of a method the class of the executing object may change, therefore the translation of this must be able to find the currently active object. For example, consider root class \texttt{A} with state subclasses \texttt{B,C,D} and method \texttt{foo()} and fields \texttt{a1,b1,c1,d1} in class \texttt{A,B,C,D} respectively.

```java
state class B extends A {
  int b1;

  void foo() {A} {
    this.b1++;  // field only in B
    this!!C;
    this.c1++;  // field only in C
    this!!D;
    this.d1++  // field only in D
    this!!B;
    this.a1++;  // field in A
  }
}
```

Furthermore, when \texttt{foo()} is called on an object of class \texttt{B}, it may have already been reclassified many times. So that the translation of \texttt{this} can find the currently active object, we add a field, \texttt{anchor}, to each class. The field \texttt{anchor} stores a pointer back to the anchor object. With the exception of state classes. The type of the field \texttt{anchor} will be the same as the class it is declared in. Figure 5.5 shows the objects just after the reclassification to class \texttt{D} in method \texttt{foo()}. 

\textit{Fickle} \texttt{this} will be translated into \texttt{this.anchor.current}. However, a cast will be required so
that members declared in state classes can be accessed e.g. b1 in state class B. The anchor field is not declared in translated state classes but can point to objects of state class. Namely, we don't know which state class it will be, so anchor has to have the type of the root class. Hence, to access members declared in the state class we cast down to the state class type. Consider the first line of the foo() method from above. It would be translated as shown in figure 5.6. Note that the current after the translation of this is redundant but all member accesses go through this redirection as explained in 5.2. To eliminate the extra current would require knowing the type of the receiver of member accesses. This translation uses minimal typing information in favor of occasional redundancy.

```
((B) this.anchor.current)).current.b1++
```

Figure 5.6: The translation of this.b1++

### 5.4.1 Aliasing issues with this

Unfortunately, there exist situations where the translation of this can create aliases to an object that can become state. Consider the following example, which shows a Store object capable of holding references to an object of class A. The method body of change() in class B is annotated with line numbers. We include a Main class which creates an instance of B and calls the change() method.
class Store {
    A val;

    void setVal(A a) {
        this.val = a;
    }

    A getVal() {
        return this.val;
    }
}

type class A {
    int a1;
    void change() {A} { }
}

state class B extends A {
    void change() {A} {
        // Set the store up.
        1. Store s = new Store();
        2. this!IC;
        3. this.a1 = 5;
        4. s.setVal(this);
        5. this!IB;
        6. this.a1 = 10;
        7. System.out.println("Inside a1 has val:"+this.a1);
        8. System.out.println("Stored A has a1 val:"+s.getVal().a1);
    }
}

state class C extends A {
}

class Main {
    void main() {
        A a;
        a = new B();
        a.change();
    }
}

After line 5 the Store object s contains an alias to this. The value of a1 has been set to 5. Diagram 5.7 shows the objects after line 4 of method change. Note that the Store object has an alias to the currently active object of type C. Diagram 5.8 shows the objects after line 5 in change(). The cross indicates the problem alias stored within the Store object. After the reclassification of line 5 the currently active object is now of type B, but the val field in Store still points to the C object. Hence lines 7 and 8 output:
Inside A a1 has val:10
Stored A has a1 val:5

This is incorrect, but its behavior is consistent with the state of the objects in 5.8. The reference to this inside the Store object becomes stale after the reclassification of line 5. When the execution of s.getVal().a1 occurs the stale objects a1 is outputted with value 5. The value should have been 10, which a1 had been set to in the currently active object of type B.
Fixing the problem

The aliasing problem occurred because the translation of this, takes us the currently active object. While this is correct for field accesses and method call, it is incorrect for the cases where a reference is store: When this can be aliased, for example, in the right hand side of an assignment, or as a parameter to a method call, the translation must be amended. The translation of this in these cases should be just anchor. Therefore, when the aliased this is accessed, as in a.getVal().a1, the currently active object will be used. Diagram 5.9 shows the objects after the corrected translation of this. Note the alias in the Store object now points to the anchor like all the other clients. Therefore, the translation of expressions must take the aliasing potential of this into account and translate accordingly, more in section 5.5.4.

5.5 Formal Description of Translation

5.5.1 Programs

The translation of a Fickle program $p$ is defined by the translation w.r.t. $p$ of all classes declared in $p$. The parameter $p$ is needed since translation of expressions depends on their types (in particular, for method invocation and field selection) and on names of root classes (in particular, constructor invocation and this).

$$[p]_{\text{prog}} = [\text{class}_1]_{\text{stmt}(p)\ldots [\text{class}_n]_{\text{stmt}(p)}$$
5.5.2 Classes

The translation of a class \( c \) is shown below. Note that \( c \) can be root.

\[
\begin{align*}
\text{[root class } c \text{ extends } & \{ t_1, f_1; \ldots, t_m, f_m; \text{meth}_1, \ldots, \text{meth}_n \}]_{\text{class}}(p) = \\
\text{[class } c \text{ extends } & \{ t_1, f_1; \ldots, t_m, f_m; \text{meth}_1, \ldots, \text{meth}_n \}]_{\text{class}}(p) = \\
\text{class } c \text{ extends } & \{ t_1, f_1; \ldots, t_m, f_m; \\
\text{[meth}_1]_{\text{root}}(p,c) \ldots [\text{meth}_n]_{\text{root}}(p,c) \\
c() | \\
\begin{align*}
\text{current} &= \text{this}; \\
\text{anchor} &= \text{this}; \\
\end{align*}
\}
\end{align*}
\]

\( c \text{ current}; \\
\text{c anchor}; \\
\text{void setCurrent}(c \text{ old})|
\begin{align*}
\text{old.current} &= \text{this}; \\
\text{anchor.current} &= \text{old} \\
\text{super.setCurrent(old); if cl-object} \\
\end{align*}
\]

The translation of a state class \( c \) produces one constructor as with normal and root classes. Furthermore, a new method static \( \text{create}(\mathcal{R}(p,c) \text{ old}) \) is added to \( c \). This method is static and is used to create an instance of \( c \). The parameter taken by \( \text{create} \) is the least root class. It represents an already existing object of type \( \mathcal{R}(p,c) \), of which all fields contained will be copied into the new object created. This method can be thought of as a cloning method where \( \text{old} \) is the original and next will be the clone.
Implementing Fieldt

5.5. Formal Description of Translation

\[
\text{state class } c \text{ extends } c' \{ t_1 f_1: \ldots t_m f_m; \text{ meth}_1 \ldots \text{meth}_n \}\]_{\alpha, (p)} = \\
\text{class } c \text{ extends } c' \{ t_1 f_1: \ldots t_m f_m; \]_{\alpha, (p)} \text{ meth}_1 \ldots \text{meth}_n [\alpha, (p, c)] \\
\text{c}() \{ \\
\text{current} = \text{this}; \\
\text{anchor} = \text{this}; \\
\} \\
\text{static create}(\mathcal{R}(p, c)) \text{ old} \{ \\
\text{c next} = \text{new c}(); \\
\text{old.saveCurrent}(); \\
\forall f \in f_1 \text{ next} : \text{f} = \text{old} \text{f}; \\
\text{return next;} \\
\}
\]

5.5.3 Method declarations

The translation of method definitions leaves the signature unmodified and translates the body. Effects are simply ignored. Since the translation of expressions depends on their types, the program \( p \) and the environment \( \gamma \) must be passed as parameters to the translation function.

\[
[t \; m(t' \; x) \; \phi[c]]_{\alpha, (p, c)} = t \; m(t' \; x) \{ [e]_{\alpha, (p, \gamma)} \}
\]

where \( \gamma = t' \; x \; c \) this.

5.5.4 Expressions

Our translation of expressions is position dependent, because of the issues outlined in 5.4.1. We define an expression as being \( lhs \) if it cannot be aliased. For example, consider the expression \( e \cdot f \). The overall expression \( e \) is \( lhs \) but it may be composed of sub-expressions that are not. An expression is \( rhs \) if it can be aliased. For example, consider the expression \( e_0 \cdot f = e_1 \). The result of evaluating \( e_1 \) can be aliased and therefore is \( rhs \). Sub-expressions of an \( rhs \) expression may be \( lhs \). Consider, \( (e_0 \cdot f = e_1) \cdot f = e_2 \). The expression \( (e_0 \cdot f = e_1) \) is \( lhs \) but the sub expression \( e_1 \) is \( rhs \). Translation of \( lhs \) expressions is defined by \( \llbracket \cdot \rrbracket_{ls} \\
\) and \( rhs \) expressions by \( \llbracket \cdot \rrbracket_{rs} \). Translation of a method body starts with \( \llbracket \cdot \rrbracket_{ls} \). The definition of translation for the following expressions is the same for \( lhs \) and \( rhs \).

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \} = [if \; e_1 \; \text{then} \; e_2 \; \text{else} \; e_3]_{\alpha, (p, \gamma)} = \\
\text{if } \{ [e_1]_{\alpha, rs}(p, \gamma) \} \{ [e_2]_{\alpha, rs}(p, \gamma) \} \text{ else } \{ [e_3]_{\alpha, rs}(p, \gamma) \} \\
\{ e_1; e_2 \}_{\alpha, rs}(p, \gamma) = [e_1; e_2]_{\alpha, rs}(p, \gamma) = \{ e_1 \}_{\alpha, rs}(p, \gamma); \{ e_2 \}_{\alpha, rs}(p, \gamma) \\
\{ x \}_{\alpha, rs}(p, \gamma) = \{ x \}_{\alpha, rs}(p, \gamma) = x \\
\{ \text{true} \}_{\alpha, rs}(p, \gamma) = \{ \text{true} \}_{\alpha, rs}(p, \gamma) = \text{true} \\
\{ \text{false} \}_{\alpha, rs}(p, \gamma) = \{ \text{false} \}_{\alpha, rs}(p, \gamma) = \text{false} \\
\{ \text{null} \}_{\alpha, rs}(p, \gamma) = \{ \text{null} \}_{\alpha, rs}(p, \gamma) = \text{null} \\
\{ \text{new } c \}_{\alpha, rs}(p, \gamma) = \{ \text{new } c \}_{\alpha, rs}(p, \gamma) = \text{new } c() \\
\{ e_1 := e_2 \}_{\alpha, rs}(p, \gamma) = \{ e_1 := e_2 \}_{\alpha, rs}(p, \gamma) = \{ e_1 \}_{\alpha, rs}(p, \gamma); \{ e_2 \}_{\alpha, rs}(p, \gamma) \\
\{ e \cdot f \}_{\alpha, rs}(p, \gamma) = \{ e \cdot f \}_{\alpha, rs}(p, \gamma) = \{ e \}_{\alpha, rs}(p, \gamma); \{ f \}_{\alpha, rs}(p, \gamma) \\
\{ e_1; m(e_2) \}_{\alpha, rs}(p, \gamma) = \{ e_1; m(e_2) \}_{\alpha, rs}(p, \gamma) = \{ e_1 \}_{\alpha, rs}(p, \gamma); \{ m(e_2) \}_{\alpha, rs}(p, \gamma) \\
\]

45
5.6 Translation of the Flip/Flop Example

The flip/flop example would be translated by Carmeda into the following Java code:

class On extends FlipFlop {

    // Fickle Auto Gen Stuff

    public On() {
        current = this;
        anchor = this;
    }

    On current;

    public static On createOn(FlipFlop _FlipFlop) {
        On _On = new On();
        _On.counter = _FlipFlop.counter;
        _FlipFlop.setCurrentThis(_On);
        return _On;
    }

    // End of Fickle Auto Gen Stuff

    public void flip() {
        this.current.counter++;
        (Off.createOff(((On)anchor.current)).
    }

}

class Off extends FlipFlop {

    // Fickle Auto Gen Stuff

    public Off() {
        current = this;
        anchor = this;
    }

    Off current;

    public static Off createOff(FlipFlop _FlipFlop) {
        Off _Off = new Off();
        _Off.counter = _FlipFlop.counter;
    }
Implementing *Fickle*

```java
_FlipFlop.setCurrentThis(_Off);
return _Off;
}

// End of Fickle Auto Gen Stuff

public void flip() {
   this.current.counter = this.current.counter + 2;
   (On.createOn(((Off)anchor.current)));
}
}

class FlipFlop {

   // Fickle Auto Gen Stuff

   public FlipFlop() {
      current = this;
      anchor = this;
   }

   FlipFlop current;
   FlipFlop anchor;

   void setCurrentThis(FlipFlop _FlipFlop) {
      _FlipFlop.anchor = anchor;
      anchor.current = _FlipFlop;
   }

   // End of Fickle Auto Gen Stuff

   int counter;

   public void flip() {
   }
}
```
Chapter 6

Alternative Translations of Fickle into Java

6.1 Reference Tracking

Earlier work on mapping *Fickle* into Java [15] suggested a technique called reference tracking. Instead of using indirection, every reference to a translated *Fickle* object is tracked using a *tracker object*. When mutation occurs all references are updated, to ensure referential integrity. Consider the following example, assuming definitions of classes A, B, C respectively:

```java
void foo() {
    A r1;
    A r2;
    A r3;
    B r4;
    C r5;
    C r6;

    r1 = new A();
    r2 = r1;
    r3 = r2;
    r4 = new B();
    r5 = new C();
    r6 = r5;
}
```

![Diagram of objects and their references in memory.](image)

Figure 6.1: The objects and their references in memory.

Diagram 6.1 shows the objects in memory with the references after `foo()` has executed. References have to be updated in the following situations:
6.2. 'The Italian Approach'

A slight variation of the mapping used in Carmela is given in [5]. Each re-classifiable Fickle object \( o \) is represented through a pair of Java objects \( [w,i] \). The object \( w \) is called the wrapper and provides the (non-mutable) \textit{identity} of \( o \). The object \( i \) is called the \textit{implementor} and provides the (mutable) \textit{behavior} of \( o \). In Carmela, \( w \) would be the \textit{anchor} object and \( i \) would be the \textit{currently active object}. Furthermore, the translation used in Carmela allows the same object to be both the \textit{anchor} and the \textit{currently active object}, until re-classification occurs. [5] always uses two objects. Both approaches use extra fields in the translated objects to ensure clients are using the \textit{currently active object}. 

6.1.1 Advantages

- Field access and method calls are faster because there is no indirection.

6.1.2 Disadvantage

- No mechanism exists for calculating, storing or passing the \( l \) value for a field within Java. Unless the literal name and literal enclosing class name for a field is known at compile-time and built into a mechanism for updating or reading from the field, then access to that field is impossible.

- Re-classification and object creation are slow because all references to the re-classified object must be updated.
6.2.1 Translation of the Flip/Flop Example
The Flip/Flop example used in previous chapters would be translated as follows using the mapping in [5].

```java
class FickleObject {
    FickleObject implementor;
    FickleObject trueThis;
    FickleObject() { creates instances of non-state classes
        implementor=this;
        trueThis=this;
    }
    FickleObject(FickleObject oldImp) { // re-classifies objects
        implementor=oldImp;
        trueThis=oldImp.trueThis;
        trueThis.implementor=this;
    }
}

class FlipFlop extends FickleObject {
    int counter; // Assume will be initialized to false
    void flip() {FlipFlop} { }
    FlipFlop() {}
    FlipFlop(FlipFlop imp) {
        trueThis = this;
        implementor=imp;
        imp.trueThis=this;
    }
    FlipFlop(FickleObject oldImp) {
        super(oldImp);
        counter = ((FlipFlop oldImp)).counter;
    }
}

class On extends FlipFlop {
    void flip() { {
        ((On) thisThisImplementor).counter++;
        new Off(trueThisImplementor); // We are now off!
    }
    On() {}
    On(FickleObject oldImp) {
        super(oldImp);
    }
}

class Off extends FlipFlop {
    void flip() { {
        ((On) thisThisImplementor).counter+=2;
        new On(trueThisImplementor); // We are now on!
    }
    Off() {}
    Off(FickleObject oldImp) {
        super(oldImp);
    }
}
```

50
Implementing *Fickle*  

6.3. Method Splitting

```java
}

class Main {
    FlipFlop flipper; // We declare to be of type FlipFlop
    public static void main(String args[]) {
        this.flipper = new FlipFlop(new Off());
        while(true) {
            ((FlipFlop) this.flipper.implementor).flip();
        }
    }
}
```

6.2.2 Advantages
- The translated objects have less fields to support the reclassification. In *Carmela* each translated object has a current field and all non state class have an anchor field.

6.2.3 Disadvantages
- There are more objects because each *Fickle* object is always represented by a pair of objects. In *Carmela* there is one object for each *Fickle* object until a re-classification occurs.
- A cast is required for field access and method call, because the implementor field (equivalent to *Carmela*'s current) is of type *Fickle*Object. *Carmela* has a current field in every translated object removing the need for the cast and using overloading instead.

6.3 Method Splitting

One of the main design considerations of the translation described in chapter 5, was the continuation of method execution after re-classification. There, execution in the target class was emulated using the current and anchor fields.

An alternative to this approach, would be to really continue execution in the target class after re-classification. This can be achieved by a technique which we call method splitting, proposed by Sophia Drossopoulos. After re-classification, a method call is made to the newly created currently active object, to continue execution of the method there. Therefore, each method declaration in *Fickle* would be translated into a series of methods. The number of translated methods per *Fickle* method are equal to the number of re-classifications in the *Fickle* method. Consider a slight variation of the class *Off* from the flip/flop example.

```java
state class Off extends FlipFlop {
    void flip() {FlipFlop} {
        this.counter+=1;
        this!!On; // We are now on!
        this.counter+=1;
    }
}
```

The method flip() still increments counter by 2 but, it is now done in two stages, before and after the re-classification this!!!On. So for an object of type *Off* executing flip(), the first expression this.counter+=1; we have this of type *Off*. After the re-classification, the third expression has this of type *On*. *Carmela* would generate the following code for the class *Off*.

```java
class Off extends FlipFlop {
```
// Fickle Auto Gen Stuff

public Off() {
    current = this;
    anchor = this;
}

Off current;

public static Off createOff(FlipFlop _FlipFlop) {
    Off _Off = new Off();
    _Off.counter = _FlipFlop.counter;
    _FlipFlop.setCurrentThis(_Off);
    return _Off;
}

// End of Fickle Auto Gen Stuff

public void flip() {
    this.current.counter = this.current.counter + 1;
    (On.createOn(((Off)anchor.current)));
    ((On) anchor.current).current.counter = ((On) anchor.current).current.counter + 1;
}

}

We see that initially, the access to counter in flip() requires no casting as this has type Off. After the re-classification, the code generated must mimic execution as if it were in class On. Therefore, in the final translated expression in flip() we see accesses to this represented by the expression ((On) anchor.current) as described in chapter 5.

In the translated class Off, with method splitting, the third expression would be replaced by a call to a split method in class On. The Fickle class Off, would be translated as shown below. Note that the translation of On must also be given as execution of flip() in Off continues in On and similarly for flip() in On and continuation in class Off.

class Off extends FlipFlop {

    // Fickle Auto Gen Stuff

    public Off() {
        current = this;
        anchor = this;
    }

    Off current;

    public static Off createOff(FlipFlop _FlipFlop) {
        Off _Off = new Off();
        _Off.counter = _FlipFlop.counter;
    }
public void flip() {
    this.counter = this.counter + 1;
    On.createOn(this);
    flipOffToOn1(); // Call to split method
}

// Split method for method flip() in On
public void flipOnToOff1() {
    // Nothing here!!
}

class On extends FlipFlop {

    // Fickle Auto Gen Stuff

    public On() {
        current = this;
        anchor = this;
    }

    On current;

    public static On createOn(FlipFlop _FlipFlop) {
        On _On = new On();
        _On.counter = _FlipFlop.counter;
        _FlipFlop.setCurrrentThis(_On);
        return _On;
    }

    // End of Fickle Auto Gen Stuff

    public void flip() {
        this.current.counter++;
        Off.createOff(this);
    }

    // Split method for method flip() in Off
    public void flipOffToOn1() {
        // We continue flip() from class Off here
        this.counter = this.counter + 1;
    }
}
6.3.1 Advantages

- The translation of this is greatly simplified because, execution of methods occurs in objects of the correct class, rather than being simulated.

6.3.2 Disadvantages

- Causes an explosion of methods within the translated classes.

- 'Pollutes' classes with code from other classes. In the above example if 0ff where to change we would have to recompile 0n as well. So in general we lose separate compilation of the translated classes.

- Translating complex expressions involving many re-classification is non-trivial. Consider:

\[(\text{this}!!\text{C.c1} = \text{this}!!\text{B.b1}) = \text{this}!!\text{D.d1};\]
Chapter 7

Proving *Carmela* Correct

We now formalize some properties of the translation given in chapter 5. The proofs are given in appendix B. The theorems and definitions are similar to those given in [5].

7.1 Preservation of static correctness

**Theorem 1.** For any *Fickle* program P, if P is well-typed (in *Fickle*), then $[P]_{\text{jos}}$ is well-typed (in Java).

In order to be proved, the claim of the theorem must be extended to all subterms of P. The strengthened claim can be proved by induction on the typing rules. We look at the theorem for expressions as it is the most interesting.

The translation preserves types up to state classes, in the following sense: if a *Fickle* expression $e$ has type $t$ w.r.t. a program $P$ and an environment $\Gamma$, and $e$ is translated into a Java expression $e'$ that has type $t'$ w.r.t. $p$ and $\Gamma$, then $t = t'$, when $t$ is not a state class, and $P \vdash t' \leq R(P,t)$ when $t$ is a state class. For the Java subset obtained from the translation we can use the *Fickle* type system, so that for any well-typed Java expression $e$ we can derive judgments of the form $P, \Gamma \vdash e : t \mid \Gamma(\text{this}) \parallel \emptyset$, where $t$ is the type of $e$. The fact that the type of this remains the same, and the set of effects is empty indicate that $e$ contains no re-classifications. Theorem 1 for expressions can be formalized as follows:

For any *Fickle* expression $e$, program $P$, environment $\Gamma$, if

- $P, \Gamma \vdash e : t \mid \Gamma(\text{this}) \parallel \emptyset$, and
- $[[e]]_{\text{jos}}, \epsilon(P, \gamma) = e'$ or $[[e]]_{\text{jos}}, \epsilon(P, \Gamma) = e'$, and
- $[P]_{\text{jos}} = P$,

then

- $P, \Gamma \vdash e' : e' \mid \Gamma(\text{this}) \parallel \emptyset$ where $P \vdash e' \leq R(P,t)$ and if $e \neq \text{this}$ or $e' = [\text{this}]_{\text{jos}}$, then $e' = t$.

7.2 Preservation of dynamic semantics

We wish to show that the semantics of expressions is preserved by the translation. The semantics of *Fickle* given in [20] rewrites pairs of expressions and stores into pairs of values and stores. The rewriting is defined by the judgement $e, \sigma \gamma \vdash v, \sigma'$. The program $P$ provides the definition of classes used in the expression. The subset of Java that the translation maps to, is slightly
different to \textit{Fickle}. In particular a distinction is made between statements and expressions, there are constructors and static fields and methods. However, the semantics of \textit{Fickle} given in [20] could easily be adapted to handle these constructs. Rules for casting are not required because the typing ensures that it would be applied to objects that have the target type.

We introduce a relation between stores \(P \vdash \sigma \triangleright_T \sigma'\) that states the fact that the store \(\sigma'\) is the 'translation' of the store \(\sigma\). Informally this means that an object \(o\) of class \(c\) in \(\sigma\) corresponds to an object \(o'\) in \(\sigma'\) that is an instance of the translation of the class \(c\). Both the store \(\sigma\) and the store \(\sigma'\) are assumed to agree (as defined in [20]) with the relative environments and programs.

\textbf{Definition 1}. Let \(P, \Gamma \vdash \sigma \otimes \sigma'\) and \([P], \Gamma \vdash \sigma'\otimes\). We say that \(\sigma'\) in \(\sigma'\) corresponds to \(\sigma\) w.r.t. \(P, \Gamma\), and write \(P, \sigma, \sigma' \triangleright_T \nu\), if either

- \(\nu = \nu' = \text{true}\), or \(\nu = \nu' = \text{false}\), or \(\nu = \nu' = n\) (for some integer \(n\)), or \(\nu = \nu' = \text{null}\), or
- \(\nu = i, \nu' = \iota, \sigma(i) = [f_1 := v_1, \ldots, f_r := v_r]^{c'}\),
  \(\sigma'(\iota') = [\ldots, \text{anchor:} \iota', \text{current:} \cdot\cdot\cdot]^{c''}\) where \(P \vdash \iota' \leq R(P, c)\), and
  \(\sigma'(\iota') = [f_1 := v_1, \ldots, f_r := v_r, \text{anchor:} \iota', \text{current:} \cdot\cdot\cdot]^{c''}\), and
  for all \(i, 1 \leq i \leq r, P, \sigma, \sigma' \triangleright_T \nu_i\), and
  if \(c\) is not a state class, then \(\iota' = i\).

If \(c\) is not a state class, then \(R(P, c) = c\). In definition 2 we introduce the notion of correspondence between stores.

\textbf{Definition 2}. Let \(P, \Gamma \vdash \sigma \otimes \sigma'\) and \([P], \Gamma \vdash \sigma'\otimes\). We say that store \(\sigma'\) corresponds to \(\sigma\) w.r.t. \(P\), and write \(P, \sigma, \sigma' \triangleright_T \sigma'\), if

1. \(P, \sigma, \sigma' \triangleright_T (\sigma(x)) \triangleright_T \sigma'(x)\).
2. \(P, \sigma, \sigma' \triangleright_T (\sigma(\text{this})) \triangleright_T (\sigma'(\text{this}))(\text{anchor})\), and
3. for all \(i\) if \(\sigma(i)\) is defined there is a unique \(i\) such that \(P, \sigma, \sigma' \triangleright_T i\).

The last condition of the definition asserts that there is an injection between address defined in \(\sigma\) and the addresses defined in \(\sigma'\).

\textbf{Definition 3}. We say a mapping from addresses to addresses, \(T' : \text{ext } T \iff \forall x, T'(x) \neq Udf \implies T(x) = T'(x)\).

\textbf{Theorem 2}. For a well-typed expression \(e\), stores \(\sigma_0\), and \(\sigma_1\), such that \(P, \Gamma \vdash \sigma_0 \otimes \sigma_1\) and \(P \vdash \sigma_0 \triangleright_T \sigma_1\).

\[
e, \sigma_0 \triangleright_T \nu, \sigma_0'\]

if and only if \([e], \sigma_1 \triangleright_T \nu', \sigma_1'\]

And there exists a \(T'\) such that \(T' : \text{ext } T\) and \(P \vdash \sigma_0' \triangleright_T \sigma_1'\) and \(P, \sigma_0', \sigma_1' \triangleright_T \nu\).

\subsection{7.3 Dead Objects}

A stale object is never accessed and can be garbage collected. More formally, an object at \(\iota\) is dead in state \(\sigma\) iff \(\forall \nu \neq \nu' \forall \sigma(\iota')(f) \neq \nu\) and \(\sigma(x)(f) \neq \nu\).
Chapter 8

Extending the underlying language

8.1 Local Variables

A very conservative extension to \textit{Fickle} is to allow for local variable declarations. We define a local variable as a variable declared within the body of some method. For example, in the class \texttt{Foo} below, \texttt{j} in method \texttt{m1} is a local variable.

class Foo {
    int m1(int i) {
        int j; // A local variable!
        j = 0;
        j = i*2;
        return j;
    }
}

We restrict local variables to be of non-state class type as with all declarations in \textit{Fickle}. Figure 8.1 shows the new typing rule, the operational semantics obviously remain unchanged.

8.1.1 Adding local variable support to \textit{Carmela}

To add local variable support to \textit{Carmela} the following was done:

- The grammar file was changed to allow variable declarations within method bodies. The declarations are syntactically identical to field declarations.

- The semantic analyzer was extended, to add local variables to the scope of the enclosing \texttt{MethodSymbol}.

Name resolution was unchanged, because local variables share the same scope as the formal parameters of methods. Therefore, when an identifier is encountered the search starts with the local variables which includes the method formal parameters.

\[
\begin{array}{c}
\frac{P \vdash t \text{ var}}{P, \Gamma \vdash x : t}
\end{array}
\]

\[
\begin{array}{c}
\frac{P, \Gamma \vdash x : t \\ \Gamma \vdash \text{ var}}{P, \Gamma \vdash t \, \text{ var} :: e : t'}
\end{array}
\]

Figure 8.1: Local Variable Declaration Expression

57
8.1.2 Issues with local variables

Carmela does not do data flow analysis, and therefore will not detect local variables that are used before they are initialized. For example:

```java
class Foo {
    int m(int i) {
        int j; // A local variable!
        return j+2; // Error j has not been initialized
    }
}
```

However, when the translated class is compiled with javac a warning will be produced. Although this is undesirable, it is beyond the scope of the project to do data flow analysis of this kind. For a good introduction to data flow analysis see [4].

8.2 Exceptions

Exceptions and exception handling are useful when developing robust programs with reliable error detection. For a formal treatment of exceptions in Java refer to [7] or the language specification [13]. In Java, exceptions are objects of the class Throwable or one of its subclasses. When an exception is thrown control is passed to the nearest catch block for that exception in the dynamic call stack. We consider the implications of adding exceptions to Fickle, in particular we discuss the interesting cases when

- The exception object is re-classified after it has been thrown.
- Within the catch or finally block re-classification occurs.

To discuss these cases we extend the syntax of Fickle with try, catch and throw keywords as in Java.

8.2.1 Re-classification of the exception object

A normal class in Fickle can have subclasses that are of state class type. Therefore, the object thrown as a result of an exception could be re-classified in a handler. Consider the following example, which assumes the existence of the predefined Exception class.

```java
root class FickleException extends Exception {
    void report() {FickleException} { }
}

state class FreshFickleException extends FickleException {
    void report() {FickleException} {
        System.out.println("A problem has occurred...");
        this!!WornOutFickleException;
    }
}

state class WornOutFickleException extends FickleException {
    void report() {FickleException} {
        System.out.println("Sorry I am too tired to tell you anything!");
    }
}
```

58
// A catch block in some method m1()

.... } catch (FickleException e) { // Line 1
  e.report(); // Line 2
  throw e // Throw upwards // Line 3
}

In the example if the handler is executed (due to an exception of FickleException or subclass) the report() method will be called. This will produce a report, then re-classify the exception object. The exception is then thrown again with the throw keyword. When exception reaches the next appropriate handler in the dynamic chain and the report() method is called, the exception will produce a different report to reflect the fact it has been 'used'.

8.2.2 Re-classification of the handler object

There is also the possibility that the object handling the thrown exception is re-classified. An object might re-classify itself into some 'broken' state when an exception occurs. Consider a modification to the Flip/Flop example.

class FlipFlopException extends Exception { }

class FlipFlop {
  int counter; //Assume will be initialized to false
  void flip() {FlipFlop} { }
}

state class Off extends FlipFlop {
  void flip() {FlipFlop} {
    try {
      if (this.counter++ > MAXFLIPS) {
        throw new FlipFlopException();
      }
      this!!Off; We are off!
      catch(FlipFlopException e) {
        this!!BrokenFlipFlop;
      }
    }
  }
}

state class BrokenFlipFlop extends FlipFlop {
  void flip() {FlipFlop} {
    System.out.println("Broken, get a new one!");
  }
}

In this example after the Flip/Flop has flipped too many times a FlipFlopException is thrown. In the handler block the Flip/Flop is re-classified into a BrokenFlipFlop. When clients invoke flip() on a BrokenFlipFlop an error is reported.

8.2.3 Discussion

Although these example are contrived they do demonstrate the possibilities of coupling re-classification with exceptions. While no attempt has been made to extends Carmela with exceptions, we do not believe it would be that difficult. The only Fickle complication would be in the finally clause
of the handler. Because we cannot know at compile time which catch block will be executed we re-classify the receiver to the least root on entry to the finally block.
Chapter 9

Extensions of Fickle

9.1 State Types for Local/Method Parameter Variables

Fickle only allows the receiver to be of a state class and to be re-classified. However it would be beneficial in certain cases, to use state classes as types for local variables and method parameters. In [14] a LinkedList example is given where casting is required, if parameters could have state class type, this would not be necessary. We now discuss why Fickle only allowed the receiver to have state class type.

9.1.1 Origins of Aliasing - The Enemy

The reason for restricting where state classes can appear as types is aliasing. The two sources of aliasing in Fickle are assignment and method call.

Explicit aliasing through assignment

The following example demonstrates the danger of state classes as types when aliasing can occur through assignment.

```java
root class A {
    int f1;
}

state class B extends A {
    int f2;
}

state class C extends A {
    int f3;
}

class D {
    void m1() {
        B b1, b2; // Line 1
        b1 = new B(); //Line 2
        b2 = b1; // Line 3
        b1!!C; // Line 4
        b2.f2; // Line 5 Field not found!! stuckError!
    }
}
```
Fickle prevented this by never allowing state classes as types except for this. Hence, the declaration on line 1 would be illegal. We show that state classes as types are dangerous only in combination with unchecked aliasing between variables of state class types. In essence, once the assignment on line 3 has occurred the declared type of b2 is too specific, and hence dangerous, as shown by line 5.

**Implicit aliasing through method call**

When a method is invoked there is aliasing between the caller’s parameters and the callee’s formal parameters, as shown in this example:

```java
root class A {
    int fa1;
}

state class B extends A {
    int fbi;
    void makeIntoB(C input) {A} {
        input!!B;
    }
}

state class C extends A {
    int fci;
}

class D {
    void m1() {A} {
        B b1 = new B(); // Line 1
        C c1 = new C(); // Line 2
        b1.makeIntoB(c1); // Line 3
        c1.fci; // Line 4 - Field not found!! stuckError!
    }
}
```

In this case the alias is between c1 in class D and input in class B. Hence on line 4 c1 is no longer pointing to an object of type C but B. Furthermore, the receiver and the arguments can be aliases of each other, consider the following example:

```java
root class A {
    int fa1;
    void m1() {A} {}
    void m2(B b) {A} {}
}

state class B extends A {
    int fbi;
    void m1() {} {
        this.m2(this); // Line 0
    }
    void m2(A a) {
        this!!C; // Line 1
    }
}
```

62
state class C extends A {
    int fc1;
}

In this example the formal parameter a and this are aliases of each other.

9.1.2 *Fickle*III*

We propose an extension to *Fickle* that allows method parameters and local variables to be of state class type. An alternative approach, giving a similar extension to *Fickle*, is given in [21]. Both approaches are compared in section 9.1.4.

Our approach uses the type system to prevent the possibility of a *stuckError*. When aliasing occurs we promote types to prevent the problems described above. Our solution does not require any alias tracking in the sense of section 6.1. We suggest that when aliasing between values of state classes type occurs (assuming they are subclasses of each other and hence the assignment is valid) that the type of the left hand side is promoted to the least root class of its type. Each of the aliasing potentials given above is considered below with relation to the solution.

**Explicit Aliasing through assignment**

In the case of explicit aliasing through assignment it is the entity on the right hand side that can potentially cause problems later on. Informally when the entity on the right hand side is re-classified the type of the alias on the left hand side is 'left behind'. This example demonstrates.

```java
root class A {
    int fa1;
}

state class B extends A {
    int fb1;
}

state class C extends A {
    int fc1;
}

class D {
    void m1() {
        B b1,b2; // Line 1
        b1 = new B(); //Line 2
        b2 = b1; // Line 3 b2 promoted to type R(p,B)=A.
        b1!!C; // Line 4
        b2.fbl; // b2 is now of type A and hence this line is a
               // type error.
    }
}
```

By promotion of the type on the left hand side of the assignment on line 3, the danger of the alias is removed. Namely the type of the variable on the left hand side is now preemptively widened. The most specific widened type being the root class common to each side (which will be the same root class). Hence no matter how much the type of the right hand side changes between state classes (within the typing rules of *Fickle*) the type of the left hand side will never become too specific and so execution will not refer to members that no longer exist. This follows from the fact that
the least root class of a set of state subclasses only gives access to all the
members common to those subclasses i.e. declared in the root class. Promotion
restricts the type of the right hand side enough to prevent *stuckErrors* i.e
field/method not found. The typing rule is shown in figure 9.1. Note that
the rule does not take into account local variables, such an extension would not
be difficult.

**Implicit Aliasing through method call**

As with the suggested solution for assignment, type widening can be used to allow safe usage of
state class typed variables in method calls. We suggest that if the effect of the method call is
applied to each method parameter after execution of the method we will have type safety. Hence
for the example above we have:

class D {
    void m1() {A} {
        B b1 = new B(); // Line 1
        C c1 = new C(); // Line 2
        b1.makeInB(c1); // Line 3
        c1.fci; // The effect of makeInB() is {A}, this applied
        // c1 gives the type {A} @ C = A. Hence c1 now has
        // type A, and this line is a type error.
    }
}

To cover the possibility of the receiver being an alias of the arguments. We stipulate that if they
are of state class type they are not subclasses of each other. The typing rule is shown in figure
9.2.

**9.1.3 Why not fields?**

We do not allow fields to be of state class type, as their lifetime exceeds method activation. For
easy example, consider:

root class A {
   void change() {A} {} 
}

state class B extends A {
    int b1;
    void change() {A} { this!!C;}
}

state class C extends A {
    int c1;
    void change() {A} { this!!B;}
}

class Danger {
    B myB; // Illegal
    void lookAtB() { myB.b1; }
}

// d is a Danger, b is a B, d.myB is alias of b;
1. b.change();
2. d.lookAtB(); //Error
If the declaration of field myB where legal, in the context of line 1, b could be an alias of d.myB, for example, through execution of b = new B(); d := new Danger(); d.myB := b. Then, execution of line 1 would re-classify the object bound to d.myB to C, and the field access d.myB.b1 in lookAtB() would raise a fieldNotFoundException error. Therefore, fields may not be declared to be of state class type.

### 9.1.4 Comparison of approaches

The major difference between both approaches is that, FickleII applies the effect of a method to this and all parameters when method call occurs. FickleIII is more conservative and only promotes when aliasing can occur. Consider the following class definitions:

```
root class A {
    int a1;
}

state class B extends A {
    int b1;
    void foo() {A} {
        this!!C;
    }
}

state class C extends A {
    int c1;
    void foo() {A} {
        this!!B;
    }
}
```

The table below shows execution of some method m1() in both FickleII (left side) and FickleIII (right side). In particular we see that FickleII is cautious and re-classifies all parameters after the call to foo(). FickleIII in this example not promote x3, we see this as an advantage over FickleII.

<table>
<thead>
<tr>
<th>FickleII</th>
<th>FickleIII</th>
</tr>
</thead>
<tbody>
<tr>
<td>void m1(A x1,B x2,C x3)</td>
<td>void m1(A x1,B x2,C x3)</td>
</tr>
<tr>
<td>1. x2.b1++; // Correct x2 is a B</td>
<td>x2.b1++; // Correct x2 is a B</td>
</tr>
<tr>
<td>2. x2.foo(); // Re-classifies x2 and all aliases</td>
<td>x2.foo(); // Re-classifies x2</td>
</tr>
<tr>
<td>3. x2.b1++; // Type incorrect, x2 is an A</td>
<td>x2.b1++; // Type incorrect, x2 is an A</td>
</tr>
<tr>
<td>4. x3.c1++; // Type incorrect, x3 is an A</td>
<td>x3.c1++; // Type correct, x3 is still C</td>
</tr>
</tbody>
</table>

The table below shows the execution of some method m1(), again in both approaches. Here FickleII re-classifies x1 after line 2 so that the field access x1.b1++ becomes a type error. FickleII does not and so x1.b1++ would be legal. Here FickleII has the advantage over FickleIII.

<table>
<thead>
<tr>
<th>FickleII</th>
<th>FickleIII</th>
</tr>
</thead>
<tbody>
<tr>
<td>void m1(B x1,B x2)</td>
<td>void m1(B x1,B x2)</td>
</tr>
<tr>
<td>1. x2.b1++; // Correct x2 is a B</td>
<td>x2.b1++; // Correct x2 is a B</td>
</tr>
<tr>
<td>2. x1 := x2; // Type correct</td>
<td>x1 := x2; // Type correct</td>
</tr>
<tr>
<td>3. x1.b1++; // Type correct, x3 is still a B</td>
<td>x1.b1++; // Type incorrect, x1 now an A</td>
</tr>
</tbody>
</table>

There are tradeoffs between both solutions and in [21] there is a proof that their solution is sound. We have not proved the solution given here due to time constraints.
\[
\begin{align*}
P, \Gamma & \vdash x : c \iff c \in \{\} \\
P, \Gamma[\text{this} \rightarrow \phi] & \vdash e : c_1 \mid c' \mid \phi \\
P \vdash c_1 \leq c_0 \\
\mathcal{R}(P, c_0) = \mathcal{R}(P, c_1) \\
P \vdash c_0 \|_a \\
P \vdash c_1 \|_a
\end{align*}
\]

Figure 9.1: State Class to State Class Assignment

\[
\begin{align*}
P, \Gamma & \vdash e_0 : c \mid c_0 \|_0 \\
P, \Gamma[\text{this} \rightarrow \phi] & \vdash e_1 : t_1' \mid c_1 \|_1 \\
\mathcal{M}(P, \phi_1 \|_p c, m) & = t \cdot m(t_1 x) \| \{ \ldots \} \\
P \vdash t_1' \leq t_1
\end{align*}
\]

Figure 9.2: Method Call

\[
\begin{align*}
P \vdash \phi \|_a \\
\mathcal{C}(P, c) = [\text{root} \mid \text{state} \mid \text{class} c \text{ extends } c' \{ \ldots \} \\
\forall f : \mathcal{F}\mathcal{D}(P, c, f) = t_0 \Rightarrow P \vdash t_0 \|_{st} \quad \text{and} \quad \mathcal{F}(P, c', f) = \mathcal{Udf} \\
\forall m : \mathcal{M}(P, c, m) = t \cdot m(t_1 x) \| \{ e \} \Rightarrow \\
P \vdash t \|_{t_1} \\
P \vdash t_1 \|_t \\
P \vdash \phi \|_\phi \\
P \vdash t \| \phi \|_{c, p} \quad P \vdash t_1 \|_{s} \implies P \vdash c \not\leq t_1, P \vdash t_1 \not\leq c \\
P, t_1 x, c \; \text{this} \vdash e \mid t' \| \{(x, t''), (\text{this}, c')\} \|_\phi \\
P \vdash t' \leq t \\
\phi' \subseteq \phi \\
\mathcal{R}(P, c) = \mathcal{R}(P, c') \\
\mathcal{M}(P, c', m) = \mathcal{Udf} \quad \text{or} \quad (\mathcal{M}(P, c', m) = t \cdot m(t_1 x) \| \{ e' \} \quad \text{and} \quad \phi \subseteq \phi')
\end{align*}
\]

Figure 9.3: Well Formed Programs
Chapter 10

Testing Carmela

When designing a language it is not uncommon for the designers to produce a series of test cases for the language. These cases are roughly separated into two distinct types: those that should compile and those that should not. Hence, the author of a compiler for a given language should run the test cases through the compiler. Those test cases intended to compile correctly should compile and produce the outcome expected and those not intended to compile should produce an appropriate error message.

The test cases should be thorough and test all aspects of the language both static and dynamic. Hence they can categorized as follows:

- Static test that is well formed which results in successful compilation.
- Static test that is not well formed which results in a compilation error.
- Dynamic test to verify that a Fickle construct (e.g. re-classification) has been implemented correctly.

10.1 The Test Suite

An automated test suite was developed to test Carmela. The suite produces the test results in a variety of formats including HTML and Latex. The suite operates as follows:

1. The test hive file is loaded into memory (The format of this file is explained in the next section).
2. The test case source file is written to disk.
3. The test case source file is compiled with Carmela, and the output recorded.
4. If the test is dynamic the Java files created by Carmela are compiled with javac and executed. The results of execution are recorded.

10.1.1 The Hive File

The test suite takes as input a hive file which contains:

- The Fickle source for each test.
- A heading to indicate:
  - The class of test, e.g. expressions, methods.
  - File name prefix for the type of test.

67
- A flag that states whether the test should:
  * Compile successfully only (\texttt{StaticSuccess}).
  * Should not compile successfully (\texttt{StaticFail}).
  * Should compile and run (with Java) successfully (\texttt{Dynamic}).

- A description of the test

**File Format**

We give the format of the hive file and an example to clarify.

```verbatim
***
Filename prefix
\texttt{StaticSuccess} \texttt{StaticFail} \texttt{Dynamic}
**

@@
Test Description
%%
Test Code
&

... @@

@@
Test Description
%%
Test Code
&

There can be many test cases within each test header section. The following example is taken from the hive file used to test \texttt{Carmela}. It shows some of the test cases for class declarations. The Latex output for these test cases can be seen in appendix C.

***TClassDeclSuccess
Class Declarations
@StaticSuccess **

@@ Normal class declaration
%%
class TNorm1 { }
&

@@ Abstract state class declaration
%%
abs-state class TCAbs1 { }
&

@@ Normal class extending normal
%%
class TNorm1 { }
class TNorm2 extends TNorm1 { }
&

```
10.1.2 Test Result Output

As the test suite runs it produces output. This includes an index file and a result file for each test case. The index file links together individual test case result files. All output is directed through subclasses of the \texttt{TestWriter} class. Currently, subclasses include \texttt{HTMLTestWriter} and \texttt{LatexTestWriter}. Each subclass provides methods to output the following:

- Header and footer of index file
- Write index item
- Header and footer of test case header
- Header and footer of source code
- Header and footer of \texttt{Carmela} output
- Header and footer of \texttt{javac} output
- Header and footer for test run by Java.
- Write a line of output.

Each of the output methods is given a Java \texttt{OutputStream} with which to do the output. For example, the prototypes of the output methods for the header and footer of the index file are:

```java
public void writeIndexHeader(PrintStream out);

public void writeIndexFooter(PrintStream out);
```

With such a generic output system it was possible to produce HTML test results for viewing with a web browser and also the Latex source for appendix C of this report.
Chapter 11

Evaluation

We evaluate the work undertaken and highlight any limitations and problems. We show the major achievements of the project and discuss further work that could be undertaken. Software engineering aspects of the project are also discussed.

11.1 Conclusions

11.1.1 Is the implementation correct?

The correctness of the implementation has been approached from two perspectives:

- Theoretical - The theorems of chapter 7 show that the code generation as defined in chapter 5 is correct for the cases considered in appendix B. This does not prove that the implementation is correct, rather that the definition of code generations is correct. We make the assumption that Carmela implements the definition of code generation correctly.

- Test Suite - The test suite was used throughout the development of Carmela to test each language construct as it was implemented. Naturally, the test suite also developed while Carmela was being developed.

11.1.2 Were the extensions useful?

We consider each of the extensions proposed in chapters 8 and 9.

Local Variables

Perhaps the most conservative of the language extensions proposed, but very useful when writing programs. From a theoretical perspective the changes to Fickle were minor, with one extra typing rule. Adding support to Carmela was equally as easy, with section 8.1 describing the details. By default, local variable support is activated when compiling Fickle programs with Carmela.

Exceptions

Although only discussed in section 8.2, we believe that exceptions would be a worthwhile extension to the language. The ability to re-classify the handling object, could be useful when ensuring clients cannot use an object after a problem has occurred.

State types for parameters and local variables

The importance of this extension is allied by its inclusion in FickleIII[21].
11.1.3 Would I do things differently next time?

A lot of time was spent examining the source code of the Java compiler, javac, and using it as a basis for Carmela's type checker. This resulted in a sound design, with room for extensions in the future. However Java is a more complex language than Fickle with features like: constructors, exceptions, static methods and classes, threads, packages etc. This in turn, meant that a considerable amount of time was required to fully understand the operation of the compiler. Scaling this understanding for developing Carmela's type checker was difficult. However, the final product is reliable and extendible with the rich features seen in Java.

It would have been nice to have completed more of the proofs but the tight time constraints meant this was not possible. However, formulation of the theorems and definitions required considerable amounts of thinking and highlighted many issues.

11.1.4 What was the most challenging part?

The most challenging coding was during the development of the type checker. Many hours were spent researching and understanding the operation of the Java compiler, javac. Especially difficult was error handling.

The most interesting and challenging aspect of the project was developing the formalization of the mapping from Fickle into Java. Much was learned about taking an intuitive idea, formalizing it and then proving that it is correct. The formalization step helps pin down all of the details in preparation for the proof. Also enjoyable was the alias control research. This novel approach of extending Fickle started as a rough idea and developed into an interesting theory.

11.1.5 Formal Methods

The use of rigorous mathematical theories has been a guiding light throughout the development of Carmela. The use of formalisms for code generation was instrumental in finding some of the subtler mistakes. The aliasing problem of 5.4.1 was discovered while formulating the invariants of chapter 7.

11.2 Software Engineering Aspects

11.2.1 Design Methodologies

The visitor pattern[9] was used throughout the design of Carmela. The reason for this was:

- SableCC produces visitors to traverse the abstract syntax tree. Therefore, the SymbolCollector, Attributes and CodeGenerator classes all inherit from these generated classes to become visitors themselves.

- The Java compiler, javac also uses the visitor pattern. It has been conjectured that this is why the latest version of javac is faster than previous versions, which did not use visitors.

The visitor pattern gives a clean separation between code for the nodes of the AST and code that traverses the AST nodes. Embedding code for type checking and code generation within the AST nodes would mean more work if the nodes changed.

11.2.2 Development Tools and Environment

Editor

The text editor Visual SlickEdit v4 was used to develop the code for Carmela. It support standard features such as syntax highlighting etc. One of its useful features is Auto Tagging of source code.
This means that the code for the completed sections of *Carmela* and the JDK sources were tagged and auto-completion of field access and method calls could be done. We believe that this feature saved much time.

**Compilers**

*Carmela* was developed using the IBM JDK 1.3. This is regarded as the fastest JVM and is freely available. No features of JDK 1.3 were directly used and therefore *Carmela* will run on JDK 1.2.

**Build Environment**

Two scripts were used for building and backing-up the sources of *Carmela*. The first, `make.bat` compiled the source tree and produced a jar file (*Fickle.jar*) for distribution. The second compiled the sources and created jar file (*FickleSource.jar*) which contained the sources. This was used to hack the project up.

### 11.2.3 Code Organization

The code is separated into six sub packages of the toplevel package `uk.ac.ic.doc.cl.a97.carmela`, these are:

- **code** - Data structures used during compilation.
- **compiler** - The classes that support the phases of compilation.
- **parser** - Classes for the parser generated by SableCC.
- **utils** - Classes for the log and some tree manipulation classes.
- **testsuite** - Classes to support the test suite.
- **resources** - Text file containing the warning and error messages.

### 11.3 Achievements

There were many achievements both theoretical and practical during the development of the project.

#### 11.3.1 *Carmela*

The development of a compiler for the language *Fickle* capable of generating Java source that can be compiled and used. Important features:

- **Speed of operation** - *Carmela* is very fast and operates in under 1-2 seconds. Running the test suite takes around 8 seconds on average.
- **Accurate error reporting** - The error reporting in *Carmela* is as informative as *javac*.
- **Error Recovery** - *Carmela* will continue as best possible in the presence of errors. Furthermore, *Carmela* avoids reporting errors as a result of other errors.

#### 11.3.2 Formalization

We developed a formalism for the translation from *Fickle* into Java, which was used to theorize about the preservation of static and dynamic correctness.
11.3.3 Test Suite
The test suite is fully automated and produces the test results in a variety of formats, including HTML and Latex. The test cases are thorough and cover a wide range of cases. Throughout the development of Carmela the number of test cases increased as potential problems were ironed out. This is especially apparent in the dynamic test cases which were almost developed in parallel with the code generator.

11.3.4 Extensions
We added support for local variables to Carmela. We extended Fickle so that state classes could be used as types for method parameters and local variables. We discussed the salient issues concerned with adding exception support to Fickle.

11.3.5 Publication of a paper
The mapping from Fickle into Java described in section 6.2 came from the paper [5]. This paper was co-written by the author after working with Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, Paola Gianmini and Elena Zucca. Although both mappings are different, they share many similarities because of the discussions had while writing the paper.

11.4 Further work
Although the project has come to an end, the work will still continue. Fickle continues to be developed by Sophia Drossopoulou et al.

11.4.1 Formalization
Because of time constraints and the experience of the author the proofs of appendix B are not complete for all cases. A considerable amount of time was required to formulate the theorems. While the proofs are important, it is journey towards them where the insight and enlightenment occurs. Carmela has benefited from the time and consideration taken on the parts of the proof completed. The incomplete parts should be considered work in progress.

The state classes for parameters and local variables work, lacks any formal proof. We are encouraged by the proofs in [21] and intend to continue research in this area to fully prove the ideas developed.

11.4.2 Abstract Methods
Carmela does not support abstract methods, which would have been useful when declaring the root classes. Instead, empty method bodies are given in the case of void methods, and a single return statement is given within a non void method. The Attribute would check subclasses of root classes to ensure they implement all the abstract methods declared. Implementing abstract methods would have been fairly straightforward, but was left for future versions of Carmela.

11.4.3 Constructors
Support for constructors was considered, but would have required a fairly sizable amount of work. At a mundane level, the Attribute would have to do checks with the constructors declared in a particular class with respect to its superclasses. The rules regarding constructors are documented in [13]. A more interesting aspect is linking constructors with the re-classification operation. We
implementing Fickle

11.4. Further work

proposes that the re-classification operator could be extended to take parameters, that would map to a constructor in the target class. For example, consider the following re-classification expression.

\[
\text{this}!!\!C(x_1, \ldots, x_n);
\]

We intend that this would be re-classified to \(C\) and furthermore, the constructor \(C(x_1, \ldots, x_n)\) would be used to initialize the fields in \(C\). Naturally, the initialization of fields in \(C\) could be done after the re-classification, but this syntax is more elegant.

11.4.4 Object Tables

A completely different approach to making Fickle programs run was to modify the Java Virtual Machine (JVM), this was discussed briefly in section 2.4.2. The specification of the JVM in [22] gives a small clue as to how re-classification could be added to the JVM. To quote:

"In Sun's current implementation of the Java Virtual Machine, a reference to a class instance is a pointer to a handle that is itself a pair of pointers: one to a table containing the methods of the object and a pointer to the Class object that represents the type of the object, and the other to the memory allocated from the Java heap for the object data."

Therefore, in principle to change the class of an object in the JVM we could change the first pointer to a different method table, which in turn points to a different Class object. Care would have to be taken to make sure the allocated memory was sufficient.

We believe that with such little documentation of the inner workings of the JVM and the complexity of the source code, implementing Fickle in this way could be quite hard. Furthermore, there is no guarantee that a particular JVM represents its objects in this way, we quote:

"Other Java Virtual Machine implementations may not use techniques such as inline caching rather than method table dispatch, and they may or may not use handles"
Appendix A

User Guide

We give a brief description of how to use Carmela and the test suite.

A.1 Using Carmela

Unlike Java, Carmela does not support multiple source files, so all class declarations must be in a single file with the extension .ickle. To compile this source file the following must be done:

- Add the Fickle.jar to the CLASSPATH.
- Using the provided script, carmela.bat enter the following at the command line:
  
carmela <FileName>

This will produce a set of java files (one for each class declared in the Fickle source). These can be compiled using javac. Note that the option -compile can be used to automatically compile the Java files (if a Java compiler can be found). A trace detailing the internal operation of Carmela can be obtained by using the -trace option.

A.2 Running the test suite

Before the test suite can be run Fickle.jar must be added to the CLASSPATH. A test hive file (as described in chapter 10) is provided with Carmela. Output can be in HTML or Latex format. To start the test suite enter the following at the command line:

    testcarmela -output [HTML | Latex] TestSuite.hive

This will run the test suite and produce a directory for each test. There will be an appropriately named .index file for the output style chosen.
Appendix A

Fickle Grammar used for SableCC

/
* Fickle Grammar for SableCC
* Christopher Anderson
* November 2000
*
* Package uk.ac.ic.doc.cla97.carmela.parser;
/

/* ********************************************************************
* Helpers
* ********************************************************************/

unicode_input_character = [0..0xffff];

ht = 0x0009;
lf = 0x000a;
ff = 0x000c;
cr = 0x000d;
sp = ' ';

line_terminator = lf | cr | cr lf;
input_character = [unicode_input_character - [cr + lf]];

not_star = [input_character - '*'] | line_terminator;
not_star_not_slash = [input_character - ['*' + '/']] | line_terminator;

unicode_letter =

[0x0041..0x005a] | [0x0061..0x007a] | [0x00aa..0x00aa] | [0x00b5..0x00b5] |
[0x00ba..0x00ba] | [0x00c0..0x00d6] | [0x00d8..0x00f6] | [0x00f8..0x01f5] |
[0x01fa..0x0217] | [0x0250..0x02a5] | [0x02b0..0x02b8] | [0x02c1..0x02c2] |
[0x02d0..0x02d1] | [0x02e0..0x02e4] | [0x037a..0x037a] | [0x0386..0x0386] |
[0x0388..0x038a] | [0x038c..0x038c] | [0x038e..0x03a1] | [0x03a3..0x03ce] |
[0x03d0..0x03d6] | [0x03da..0x03da] | [0x03dc..0x03dc] | [0x03de..0x03de] |
[0x03e0..0x03e0] | [0x03e2..0x03f3] | [0x0401..0x040c] | [0x040e..0x044f] |
[0x0461..0x046c] | [0x04e5..0x0481] | [0x0490..0x04c4] | [0x04c7..0x04c8] |
[0x04cb..0x04cc] | [0x04d0..0x04e3] | [0x04ee..0x04f5] | [0x04f8..0x04f9] |
[0x0531..0x0556] | [0x0559..0x0559] | [0x0561..0x0567] | [0x056d0..0x05ea] |
[0x05f0..0x05f2] | [0x0621..0x063a] | [0x0640..0x064a] | [0x0671..0x06b7] |
[0x06ba..0x06be] | [0x06c0..0x06ce] | [0x06d0..0x06d3] | [0x06d5..0x06d5] |
| [0x06e5..0x06e6] | [0x0905..0x0939] | [0x093d..0x093d] | [0x0958..0x0961] |
| [0x0986..0x098c] | [0x098f..0x0999] | [0x0993..0x0993] | [0x099a..0x09bf] |
| [0x09b2..0x09b9] | [0x09be..0x09f9] | [0x09d7..0x09d7] | [0x09e6..0x09e6] |
| [0x09f0..0x09f1] | [0x0a06..0x0a0a] | [0x0a0f..0x0a10] | [0x0a13..0x0a28] |
| [0x0a2a..0x0a30] | [0x0a32..0x0a32] | [0x0a35..0x0a36] | [0x0a38..0x0a39] |
| [0x0a59..0x0a5c] | [0x0a5e..0x0a60] | [0x0a72..0x0a74] | [0x0a85..0x0a8b] |
| [0x0a8d..0x0a94] | [0x0a93..0x0a98] | [0x0aa3..0x0aab] | [0x0aab..0x0aab] |
| [0x0ab2..0x0ab3] | [0x0aab..0x0aab] | [0x0abd..0x0abd] | [0x0ae0..0x0ae0] |
| [0x0b05..0x0b06] | [0x0b0f..0x0b10] | [0x0b13..0x0b2b] | [0x0b2a..0x0b30] |
| [0x0b32..0x0b33] | [0x0b36..0x0b39] | [0x0b3d..0x0b3d] | [0x0b6c..0x0b6d] |
| [0x0b6f..0x0b6f] | [0x0b85..0x0b8a] | [0x0b8e..0x0b90] | [0x0b92..0x0b95] |
| [0x0b99..0x0b9a] | [0x0b9c..0x0b9c] | [0x0b9e..0x0b9f] | [0x0ba3..0x0ba4] |
| [0x0ba8..0x0baa] | [0x0bba..0x0bba] | [0x0bb7..0x0bb9] | [0x0c05..0x0c0c] |
| [0x0c0e..0x0c10] | [0x0c2a..0x0c33] | [0x0c35..0x0c39] | [0x0c91..0x0c93] |
| [0x0c60..0x0c61] | [0x0c85..0x0c8c] | [0x0c8e..0x0c90] | [0x0c92..0x0ca8] |
| [0x0caa..0x0cb3] | [0x0cba..0x0cb9] | [0x0cde..0x0cde] | [0x0ce0..0x0ce1] |
| [0x0d05..0x0d0c] | [0x0d0e..0x0d10] | [0x0d12..0x0d28] | [0x0d2a..0x0d39] |
| [0x0d50..0x0d51] | [0x0d51..0x0d59] | [0x0e30..0xe30] | [0x0e32..0xe33] |
| [0x0e40..0xe46] | [0x0e81..0xe82] | [0x0e84..0xe84] | [0x0e87..0xe88] |
| [0x0e8a..0xe8a] | [0x0e8d..0xe8d] | [0x0e94..0xe97] | [0x0e99..0xe9f] |
| [0x0ea1..0xea3] | [0x0ea5..0xea5] | [0xea7..0xea7] | [0xea9..0xea9] |
| [0x0ead..0xea6] | [0x0ebe..0xeb0e] | [0xebe2..0xeb3] | [0xebd..0xebd] |
| [0xece0..0xece4] | [0xece6..0xecc] | [0xeedc..0xedd] | [0xef40..0xef47] |
| [0xef49..0xef69] | [0xefa0..0xefc5] | [0xe10d..0xe10f] | [0xe110..0xe119] |
| [0xe115f..0xe11a] | [0xe11a..0xe11f] | [0xe1e0..0xe1eb] | [0xe1eb..0xe1e9] |
| [0xef0..0xef15] | [0xef18..0xef1f] | [0xef20..0xef45] | [0xef48..0xef4d] |
| [0xef60..0xef7f] | [0xef59..0xef59] | [0xef5b..0xef5b] | [0xef5d..0xef5e] |
| [0xef7f..0xef7d] | [0xef80..0xef84] | [0xef86..0xef8c] | [0xefbe..0xefbe] |
| [0xefc2..0xef4] | [0xefc6..0xefcc] | [0xefd0..0xefd3] | [0xefdb..0xefdb] |
| [0xefe0..0xefc] | [0xefe2..0xff4] | [0xff6..0xffc] | [0xff7f..0xff7f] |
| [0xff02..0xff02] | [0xff07..0xff07] | [0xff1a..0xff1a] | [0xff1f..0xff1f] |
| [0xff11..0xff1d] | [0xff24..0xff24] | [0xff26..0xff26] | [0xff28..0xff28] |
| [0xff2a..0xff31] | [0xff33..0xff38] | [0xff3a..0xff3a] | [0xff3d..0xff3d] |
| [0xff01..0xff04] | [0xff07..0xff07] | [0xff0f..0xff0f] | [0xff10..0xff19] |

```
unicode_digit =
  [0x0030..0x0039] | [0x0660..0x0669] | [0x06f0..0x06f9] | [0x0966..0x096f] |
  [0x0966..0x096f] | [0x0a6..0xa6f] | [0x0a6..0xa6f] | [0x0b66..0x0b6f] |
  [0x0b67..0x0b6f] | [0x0c6..0xc6f] | [0x0c6..0xc6f] | [0x0d66..0x0d6f] |
  [0x0e60..0xe659] | [0x0e0d..0xe0e9] | [0x0f0..0xf0f2] | [0xffd0..0xffd7] |

java_letter = unicode_letter | \"\`\" | \`\';
java_letter_or_digit = unicode_letter | unicode_digit | \`\" | \`\';

non_zero_digit = [\'1\'..\'9\'];
digit = [\'0\'..\'9\'];
hex_digit = [\'0\'..\'9\'] | [\'a\'..\'f\'] | [\'A\'..\'F\'];
```
Implementing Flode

```c
octal_digit = ['0'..'7'];
zero_to_three = ['0'..'3'];

decimal_numeral = '0' | non_zero_digit digit*;
hex_numeral = '0' {'x' | 'X'} hex_digit+
octal_numeral = '0' octal_digit+

integer_type_suffix = 'l' | 'L';
exponent_part = ('e' | 'E') ('+' | '-')? digit+

float_type_suffix = 'f' | 'F' | 'd' | 'D';

single_character = [input_character - ['''' + '\']];
octal_escape = '\\' (octal_digit octal_digit? | zero_to_three octal_digit octal_digit octal_digit);
escape_sequence = '\b' | '\t' | '\n' | '\f' | '\r' | '\"' | '\' | '\\' | octal_escape;
string_character = [input_character - ['"' + '\']] | escape_sequence;

/*********************************************************************************/
/* Tokens */
*********************************************************************************/

white_space = (sp | ht | ff | line_terminator)*;

traditional_comment = '/\* not_star+  )*+(not_star_not_slash not_star*  '*)+*/';
documentation_comment = '/\*\*  )*+(not_star_not_slash not_star*  '*)+*/';

end_of_line_comment = '//' input_character* line_terminator?;

abs_state = 'abs-state';
state = 'state';
abstract = 'abstract';
default = 'default';
if = 'if';
private = 'private';
throw = 'throw';
boolean = 'boolean';
do = 'do';
implements = 'implements';
protected = 'protected';
throws = 'throws';
break = 'break';
double = 'double';
import = 'import';
public = 'public';
transient = 'transient';
byte = 'byte';
else = 'else';
instanceof = 'instanceof';
return = 'return';
try = 'try';
case = 'case';
extends = 'extends';
```
int = 'int';
short = 'short';
void = 'void';
catch = 'catch';
final = 'final';
interface = 'interface';
static = 'static';
volatile = 'volatile';
char = 'char';
finally = 'finally';
long = 'long';
super = 'super';
while = 'while';
class = 'class';
float = 'float';
native = 'native';
switch = 'switch';
const = 'const';
for = 'for';
new = 'new';
synchronized = 'synchronized';
continue = 'continue';
goto = 'goto';
package = 'package';
this = 'this';

true = 'true';
false = 'false';
null = 'null';
l_parenthese = '(';
r_parenthese = ')';
l_brace = '{';
r_brace = '}';
l_bracket = '[';
r_bracket = ']';
semicolon = ';';
comma = ',';
dot = '.';

assign = '=';
mutate = '!!';
lt = '<';
gt = '>'; 
complement = '!';
bit_complement = '~';
question = '?';
colon = ':';

eq = '==';
lteq = '<=';
gteq = '>=';
neq = '!=';
and = '&&';
or = '||';
plus_plus = '++';
minus_minus = '--';

plus = '+';
minus = '-';
star = '*' ;
div = '/';
bit_and = '&';
bit_or = ' | ';
bit_xor = ' ^ ';
mod = '%';
shift_left = '<<<';
signed_shift_right = '>>';
unsigned_shift_right = '>>>';

plus_assign = '+=';
minus_assign = '-=';
star_assign = '*=';
div_assign = '/=';
bit_and_assign = '&=';
bit_or_assign = '|=';
bit_xor_assign = '^=';
mod_assign = '%=';
shift_left_assign = '<==';
signed_shift_right_assign = '>>=';
unsigned_shift_right_assign = '>>>=';

decimal_integer_literal = decimal_numeral;
hex_integer_literal = hex_numeral integer_type_suffix?;
octal_integer_literal = octal_numeral integer_type_suffix?;

floating_point_literal =
digit+ '.' digit* exponent_part? float_type_suffix? |
'.' digit+ exponent_part? float_type_suffix? |
digit+ exponent_part float_type_suffix? |
digit+ exponent_part float_type_suffix?

cr = new line
character literal = "'" ( single_character | escape_sequence ) "'";
string literal = '"" escape_sequence '"";

ident = java_letter java_letter_or_digit*;
sop = 'System.out.println';
javalit = 'Java';

/************************************************************
* Ignored Tokens
************************************************************/
Implementing Field

/**********************************************************
 * Productions
 *******************************************************/
Productions

/**********************************************************
 Root element
 *******************************************************/
progr =
     class_declaration*;

/**********************************************************
 Types, Values, and Variables
 *******************************************************/
type =
    {primitive_type}
       primitive_type |

    {reference_type}
       ident    |

    {void}
       void;

primitive_type =

    {boolean}
       boolean  |

    {int}
       int;

ident_list =
    {elem}
       ident  |

    {list}
       ident_list comma ident;

/**********************************************************
 Names
 *******************************************************/
name =
    {simple}
Implementing_Fiddle

ident |

{qualified}
  qualified;

qualified =
  name dot ident;

******************************************************************************
Lexical Structure
******************************************************************************

literal =

{integer_literal}
  integer_literal |

{boolean_literal}
  boolean_literal |

{null_literal}
  null_literal;

******************************************************************************
Class modifiers
******************************************************************************

modifier =

{abs_state}
  abs_state |

{state}
  state;

******************************************************************************
Class declaration
******************************************************************************

class_declaration =
  modifier? [t_class]:class ident P.super? class_body;

super =
  extends ident;

class_body =
  l_brace class_member_declaration* r_brace;
Implementing Field

class member declaration =
  {field declaration}
  variable declaration semicolon |

  {method declaration}
  method declaration;

//----------------------------------------------------------------------------
Variable declarations
//----------------------------------------------------------------------------

variable declaration =
  type ident;

//----------------------------------------------------------------------------
Method declarations
//----------------------------------------------------------------------------

method declaration =
  type ident l_parenthese formal_parameter_list? r_parenthese eff? method_body;

formal_parameter_list =
  {params}
    variable_declaration |

  {params_list}
    formal_parameter_list comma variable_declaration;

eff = l_brace ident_list? r_brace;

method_body =
  {block}
    block |

  {semicolon}
    semicolon;

//----------------------------------------------------------------------------
19.11 Grammar from 14: Blocks and Statements 14
//----------------------------------------------------------------------------

block =
  l_brace block_statement* r_brace;

block_statement =
  {statement}
    statement |
Implementing field

```
{local_var} 
    variable_declaration;

statement = 
    {statement_short} 
        statement_short | 
    {if_then_else_statement} 
        if_then_else_statement | 
    {java_block} 
        java_block;

java_block = sop l_parenthese java+ r_parenthese semicolon;

java = 
    {string_exp} 
        string_literal plus out_state java_follow? | 
    {string} 
        string_literal;

class = 
    {class_name} 
        name | 
    {primary} 
        primary;

java_follow = plus java;

statement_short = 
    {block} 
        block | 
    {empty_statement} 
        empty_statement | 
    {expression_statement} 
        expression_statement | 
    {return_statement} 
        return_statement | 
    {throw_statement} 
        throw_statement | 
    {try_statement} 
        try_statement;

empty_statement = 
    semicolon;
```
Implementing *Field*

expression_statement =
    statement_expression semicolon;

statement_expression =
    {assignment}
    assignment |

    {pre_increment}
    pre_increment |

    {pre_decrement}
    pre_decrement |

    {post_increment}
    post_increment |

    {post_decrement}
    post_decrement |

    {method_invocation}
    method_invocation |

    {this_mutate}
    this_mutate |

    {new_class}
    new_class;

if_then_else_statement =
    if l_parentheses expression r_parentheses [t_stars]:statement else [f_stars]:statement;

statement_expression_list =
    {statement_expression}
    statement_expression |

    {statement_expression_list}
    statement_expression_list comma statement_expression;

return_statement =
    return expression? semicolon;

throw_statement =
    throw expression semicolon;

try_statement =
    {try}
    try block catch_clause+ |

    {finally}
Implementing *Fieldte*

```java
try block catch_clause* P.finally;

catch_clause =
    catch 1_parenthese variable_declaration r_parenthese block;

finally =
    T.finally block;

/*****************************/
19.12 Grammar from 15: Expressions 15
/*****************************/

primary =
    {literal}
        literal |
    {this}
        this |
    {bracket}
        l_parenthese expression r_parenthese |
    {new_class}
        new_class |
    {field_access}
        field_access |
    {method_invocation}
        method_invocation |
    {this_mutate}
        this_mutate ;

new_class =
    new ident l_parenthese r_parenthese ;

argument_list =
    {expression}
        expression |
    {argument_list}
        argument_list comma expression;

this_mutate =
    this mutate ident;

field_access =
    primary dot ident ;
```
method_invocation =
    {name}
    name dot ident l_parenthese argument_list? r_parenthese |
    {primary}
    primary dot ident l_parenthese argument_list? r_parenthese ;

postfix_expression =
    {primary}
    primary |
    {name}
    name |
    {post_increment}
    post_increment |
    {post_decrement}
    post_decrement;

post_increment =
    postfix_expression plus_plus;

post_decrement =
    postfix_expression minus_minus;

unary_expression =
    {pre_increment}
    pre_increment |
    {pre_decrement}
    pre_decrement |
    {plus}
    plus unary_expression |
    {minus}
    minus unary_expression |
    {postfix_expression}
    postfix_expression ;

pre_increment =
    plus_plus unary_expression;

pre_decrement =
    minus_minus unary_expression;

mult =
    {unary_expression}
    unary_expression |
\{\text{star}\}
  \text{mult} \ \text{star} \ \text{unary\_expression} \ |

\{\text{div}\}
  \text{mult} \ \text{div} \ \text{unary\_expression} \ |

\{\text{mod}\}
  \text{mult} \ \text{mod} \ \text{unary\_expression};

\text{add} =
  \{\text{mult}\}
    \text{mult} \ |

  \{\text{plus}\}
    \text{add} \ \text{plus} \ \text{mult} \ |

  \{\text{minus}\}
    \text{add} \ \text{minus} \ \text{mult};

\text{relational} =
  \{\text{add}\}
    \text{add} \ |

  \{\text{lt}\}
    \text{relational} \ \text{lt} \ \text{add} \ |

  \{\text{gt}\}
    \text{relational} \ \text{gt} \ \text{add} \ |

  \{\text{lteq}\}
    \text{relational} \ \text{lt\_eq} \ \text{add} \ |

  \{\text{gteq}\}
    \text{relational} \ \text{gt\_eq} \ \text{add} \ ;

\text{equality} =
  \{\text{relational}\}
    \text{relational} \ |

  \{\text{eq}\}
    \text{equality} \ \text{eq} \ \text{relational} \ |

  \{\text{neq}\}
    \text{equality} \ \text{neq} \ \text{relational};

\text{cond\_and} =
  \{\text{equality}\}
    \text{equality} \ |

  \{\text{test}\}
    \text{cond\_and} \ \text{and} \ \text{equality};
cond_or =
    {cond_and}
    cond_and |

    {test}
    cond_or or cond_and;

assignment_expression =
    {cond_or}
    cond_or |

    {assignment}
    assignment;

assignment =
    {ident}
    ident assign assignment_expression |

    {field_assign}
    primary dot ident assign assignment_expression |

    {name}
    name dot ident assign assignment_expression ;

equation =
    assignment_expression;

costant_expression =
    expression;

/********************************************************************************/

Literals
/* *******************************************************************************/

   boolean_literal =
    {true} true |
    {false} false;

   null_literal =
    null;

   integer_literal = decimal_integer_literal;
Appendix B

Proofs of Static and Dynamic Correctness
Implementing Field
Implementing Field
Appendix C

Test Cases

This section of the report was automatically generated. Each test case has the following format:

FileName.fickle
[Test description]
Source
[Source for test case]
Carmela output
[Output from Carmela]
Java files produced
[Java files generated]

C.1 Static Cases

C.1.1 Class Declarations

Type Correct Cases

TClassDeclSuccess1.fickle
Normal class declaration
Source
1. class TClassDeclSuccess1 { }

Carmela output

Compilation successful.
Java files produced
TCDNorm1.java

TClassDecSuccess2.fickle
Abstract state class declaration
Source
1. root class TCRoot1 { }

Carmela output
Compilation successful.

Java files produced
TCRoot1.java

TClassDecSuccess3.fickle
Normal class extending normal
Source
1. class TCDNorm1 { }
2. class TCDNorm2 extends TCDNorm1 { }

Carmela output
Compilation successful.

Java files produced
TCDNorm1.java
TCDNorm2.java

TClassDecSuccess4.fickle
State class extending root class
Source
1. root class TCRoot1 { }
2. state class TCSstate1 extends TCRoot1 { }

Carmela output
Compilation successful.
Java files produced
TCRoot1.java
TCState1.java

TClassDeclSuccess5.fickle

State class extending state class

Source
1. root class TCRoot1 { }
2. state class TCState1 extends TCRoot1 { }
3. state class TCState2 extends TCState1 { }

Carmela output

Compilation successful.

Java files produced
TCRoot1.java
TCState1.java
TCState2.java

Type Incorrect Cases

TClassDeclFail1.fickle

Orphan state class

Source
1. state class TCState1 { } // Error

Carmela output

Error Found: Orphan state class TCState1, must extend a root or state class.
Line: 1
Column: 13
state class TCState1 { } // Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced
TClassDecFail2.fickle

Normal class extending state

Source
1. root class TCRoot1 { }
2. state class TCState1 extends TCRoot1 { }
3. class TCNorm1 extends TCState1 { } //Error

Carmela output

Error Found: Expecting non-state, non-root class.
Line: 3
Column: 23
class TCNorm1 extends TCState1 { } //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TClassDecFail3.fickle

Normal class extending root class

Source
1. root class TCRoot1 { }
2. class TCNorm1 extends TCRoot1 { } //Error

Carmela output

Error Found: Expecting non-state, non-root class.
Line: 2
Column: 23
class TCNorm1 extends TCRoot1 { } //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TClassDecFail4.fickle
State class extending normal class

Source
1. class TCNorm1 { }
2. state class TCState1 extends TCNorm1 { } //Error

Carmela output

Error Found: State classes can only extend state and root classes.
Line: 2
Column: 30
state class TCState1 extends TCNorm1 { } //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TClassDeclFail5.fickle

Root class extending root class

Source
1. root class TCRoot1 { }
2. root class TCRoot2 extends TCRoot1 { }

Carmela output

Error Found: Expecting non-state, non-root class.
Line: 2
Column: 28
root class TCRoot2 extends TCRoot1 { }

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TClassDeclFail6.fickle

Root class extending state class

Source
1. root class TCRoot1 { }
2. state class TCState1 extends TCRoot1 { }
3. root class TCRoot2 extends TCState1 { } //Error
Implementing *Fickle*  

C.1. Static Cases

---

*Carmela* output

Error Found: Expecting non-state, non-root class.  
Line: 3  
Column: 28  
root class TCRoot2 extends TCState1 { } //Error

There was 1 error and 0 warnings during compilation.

---

Java files produced

No .java files produced

---

**TCClassDeclFail7.fickle**

Class extending undefined class

Source

1. class TCNorm1 extends Judy { } //Error  
2. state class TCState1 extends John { } //Error  
3. root class TCRoot1 extends Chris { } //Error

---

*Carmela* output

Error Found: Cannot find type Judy.  
Line: 1  
Column: 23  
class TCNorm1 extends Judy { } //Error

Error Found: Cannot find type John.  
Line: 2  
Column: 30  
state class TCState1 extends John { } //Error

Error Found: Cannot find type Chris.  
Line: 3  
Column: 28  
root class TCRoot1 extends Chris { } //Error

There were 3 errors and 0 warnings during compilation.

---

Java files produced

No .java files produced
Implementing Fickle

C.1. Static Cases

TClassDecFail8.fickle

Cycle in class graph

Source

1. class TNorm1 extends TNorm3 { }
2. class TNorm2 extends TNorm1 { }
3. class TNorm3 extends TNorm2 { }

Carma output

Error Found: Cycle found in class graph.
Line: 2
Column: 23
class TNorm2 extends TNorm1 { }

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TClassDecFail9.fickle

Non-unique class names

Source

1. class TNorm1 { }
2. class TNorm1 { } //Error

Carma output

Error Found: Duplicate class declaration.
Line: 2
Column: 7
class TNorm1 { } //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced
C.1.2 Overriding Members

Type Correct Cases

**TOverrideSuccess1.fickle**

Standard method override

Source
1. class TCNorm1 {
2.     void m1() {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6.     void m1() {
7.         }
8. }

*Carma* output

Compilation successful.

Java files produced

TCNorm1.java
TCNorm2.java

**TOverrideSuccess2.fickle**

Method override in sub-sub-class

Source
1. class TCNorm1 {
2.     void m1() {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6. }
7. class TCNorm3 extends TCNorm2 {
8.     void m1() {
9.         }
10. }

*Carma* output

Compilation successful.

Java files produced

TCNorm1.java
TCNorm2.java
TCNorm3.java

TOverrideSuccess3.fickle

Standard method override with effects

Source

1. class TCNorm1 {
2.     void m1() {TCRoot1} {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6.     void m1() {TCRoot1} {
7.         }
8. }
9. root class TCRoot1 {

Carmela output

Compilation successful.

Java files produced

TCNorm1.java
TCRoot1.java
TCNorm2.java

TOverrideSuccess4.fickle

Standard method override with implicit effects

Source

1. class TCNorm1 {
2.     void m1() {TCRoot1} {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6.     void m1() {} {
7.         }
8. }
9. root class TCRoot1 {

Carmela output

Compilation successful.
Java files produced
TCNorm1.java
TCRoot1.java
TCNorm2.java

Type Incorrect Cases

TMethodDecFail1.fickle

Method override with different signature

Source
1. class TCNorm1 {
2.     void m1() {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6.     void m1(int a) {
7.         }
8. }

Carmela output

Error Found: Method m1 overrides a method in a superclass but has a different signature.
Line: 6
Column: 14
    void m1(int a) {

There was 1 error and 0 warnings during compilation.

Java files produced
TCNorm1.java

TMethodDecFail2.fickle

Method override in sub-sub-class with different signature

Source
1. class TCNorm1 {
2.     void m1() {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6. }
7. class TCNorm3 extends TCNorm2 {
8.     void m1(int a) {
9.         }
10. }
Carmela output

Error Found: Method m1 overrides a method in a superclass but has a different signature.
Line: 8
Column: 14
    void m1(int a) {

There was 1 error and 0 warnings during compilation.

Java files produced

TCNorm1.java
TCNorm2.java

TMethodDecFail3.fickle

Overidden method with more effects

Source
1. class TCNorm1 {
2.     void m1() {TCRoot1} {
3.         
4.     }
5. class TCNorm2 extends TCNorm1 {
6.     void m1() {TCRoot1,TCRoot2} {
7.         
8.     }
9. root class TCRoot1 { }
10. root class TCRoot2 { }

Carmela output

Error Found: The effect of m1 is not a subset of the effect in its superclass.
Line: 6
Column: 14
    void m1() {TCRoot1,TCRoot2} {

There was 1 error and 0 warnings during compilation.

Java files produced

TCNorm1.java
TCRoot1.java

TMethodDecFail4.fickle
Overridden method with unknown effect

Source

```java
1. class TCNorm1 {
2.     void m1() {TCRoot1} {
3.         }
4. }
5. class TCNorm2 extends TCNorm1 {
6.     void m1() {TCRoot1,TCRoot2} { //Error
7.         }
8. }
9. root class TCRoot1 {
```

Carmela output

Error Found: Cannot find type TCRoot2.
Line: 6
Column: 28
   void m1() {TCRoot1,TCRoot2} { //Error
```

Error Found: Only root classes may appear in the effect clause of a method declaration.
Line: 6
Column: 28
   void m1() {TCRoot1,TCRoot2} { //Error
```

Error Found: The effect of m1 is not a subset of the effect in its superclass.
Line: 6
Column: 14
   void m1() {TCRoot1,TCRoot2} { //Error
```

There were 3 errors and 0 warnings during compilation.

Java files produced
No .java files produced

TMethodDeclFail6.fickle

Field override

Source

```java
1. class TCNorm1 {
2.     int f1;
3.     void m1() {
4.         }
5. }
6. class TCNorm2 extends TCNorm1 {
```
7. int f1; // Error
8. void m1() {
9.   }
10. }

Carmela output

Error Found: Field f1 already defined in a superclass. No overriding of fields.
Line: 7
Column: 13
int f1; // Error

There was 1 error and 0 warnings during compilation.

Java files produced
TNorm1.java

TMethodDecFail7.fickle

Field override in sub-sub-class

Source
1. class TNorm1 {
2.   int f1;
3.   void m1() {
4.     }
5. }
6. class TNorm2 extends TNorm1 {
7. }
8. class TNorm3 extends TNorm2 {
9.   int f1;  //Error
10.   void m1() {
11.     }
12. }

Carmela output

Error Found: Field f1 already defined in a superclass. No overriding of fields.
Line: 9
Column: 13
int f1; //Error

There was 1 error and 0 warnings during compilation.

Java files produced
TNorm1.java
TNorm2.java
C.1.3 Expressions - Conditional

Type Correct Cases

```
TEExpCondSuccess1.fickle

Test expression is boolean

Source

1. class CTNorm1 {
2.     int m1Norm() {} {
3.         if (true) { //Could be false too
4.             return 1;
5.         } else {
6.             return 0;
7.         }
8.     }
9. }
```

Carmela output

Compilation successful.

Java files produced

CTNorm1.java

```
TEExpCondSuccess2.fickle

Re-classification in test expression

Source

1. root class TCRoot1 {
2.     void m1Abs() {} { }
3. }
4.
5. state class TCState1 extends TCRoot1{
6.     int m1State1() {} {
7.         if (this.m2State1())
8.             // At this point this is of type TCRoot1 as
9.             // the effect of m2State1() estimates.
10.             {
11.                 this.m1Abs();
12.                 return 1;
13.             }
14.         else
15.             {
16.                 this.m1Abs();
17.                 return 0;
18.             }
```
Implementing Fickle

C.1. Static Cases

19. }
20. }
21. }
22. boolean m2State1() {TCRoot1} {
23. this!!TCState2;
24. return true;
25. }
26. } 27. }
28. state class TCState2 extends TCRoot1{
29. void m1State2() { }
30. }

Carmela output

Compilation successful.

Java files produced

TCRoot1.java
TCState1.java
TCState2.java

Type Incorrect Cases

TExpCondFail1.fickle

Re-classification with invalid expression in conditional blocks

Source

1. root class TCRoot1 {
2. void m1Abs() {} }
3. }
4. state class TCState1 extends TCRoot1{
5. int m1State1() {TCRoot1} {
6. if (this.m2State1())
7. // At this point this is of type TCRoot1 as
8. // the effect of m2State1() estimates.
9. {
10. this.m2State1(); // Error
11. return 1;
12. }
13. else {
14. return 0;
15. }
16. }
17. }
18. boolean m2State1() {TCRoot1} {
19. this!!TCState2;
20. return true;
21. }
22. }
23. state class TCState2 extends TCRoot1 {
24.     void m1State2() { }
25. }

Carmela output

Error Found: Cannot find method m2State1 in TCRoot1.
Line: 10
Column: 26
    this.m2State1(); // Error

There was 1 error and 0 warnings during compilation.

Java files produced
TCRoot1.java

TExpCondFail2.fickle

Wrongly typed test expression

Source
1. class CTNorm1 {
2.     int m1Norm() {} {
3.         if (2) { // Error
4.             return 1;
5.         } else {
6.             return 0;
7.         }
8.     }
9. }

Carmela output

Error Found: Incompatible types expecting boolean but found int.
Line: 3
Column: 21
    if (2) { // Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced
C.1.4 Expressions - Assignment

Type Correct Cases

TExpAssignSuccess1.fickle

Literal integer assignment

Source
1. class TCNorm1 {
2.     int f1;
3.     void m1Norm1() {} {
4.         this.f1 = 1;
5.     }
6. }

Carmela output

Compilation successful.

Java files produced
TCNorm1.java

TExpAssignSuccess2.fickle

Field to field integer assignment

Source
1. class TCNorm1 {
2.     int f1;
3.     int f2;
4.     void m1Norm1() {} {
5.         this.f1 = 10;
6.         this.f2 = this.f1;
7.     }
8. }

Carmela output

Compilation successful.

Java files produced
TCNorm1.java

TExpAssignSuccess3.fickle
Literal boolean assignment

Source
1. class TCNorm1 {
2.     boolean f1;
3.     void mINorm1() {} {
4.         this.f1 = true;
5.     }
6. }

Carmela output

Compilation successful.

Java files produced
TCNorm1.java

TExpAssignSuccess4.fickle

Field to field boolean assignment

Source
1. class TCNorm1 {
2.     boolean f1;
3.     boolean f2;
4.     void mINorm1() {} {
5.         this.f1 = true;
6.         this.f2 = this.f1;
7.     }
8. }

Carmela output

Compilation successful.

Java files produced
TCNorm1.java

TExpAssignSuccess5.fickle

Field of class type assignment under new

Source
1. class TCNormAux { }
2. class TCNorm1 {
3.     TCNormAux f1;
4.     void miNorm1() {} {
5.         this.f1 = new TCNormAux();
6.     }
7. }

_Carmela_ output

Compilation successful.

Java files produced

TCNormAux.java
TCNorm1.java

TExpAssignSuccess6.fickle

Field to field of class type assignment

Source

1. class TCNormAux { }
2. class TCNorm1 {  
3.     TCNormAux f1;
4.     TCNormAux f2;
5.     void miNorm1() {} {  
6.         this.f1 = new TCNormAux();
7.         this.f2 = this.f1;
8.     }
9. }

_Carmela_ output

Compilation successful.

Java files produced

TCNormAux.java
TCNorm1.java

TExpAssignSuccess7.fickle

Subclass of field assignment under new

Source

1. class TCNormAux1 { }
2. class TCNormAux2 extends TCNormAux1 { }
3. class TCNormAux3 extends TCNormAux2 { }
4. class TCNorm1 {  
5.     TCNormAux1 f1;
6.     void m1Norm1() {} {
7.         this.f1 = new TCNormAux3();
8.     }
9. }

Carmela output

Compilation successful.

Java files produced
TCNormAux1.java
TCNorm1.java
TCNormAux2.java
TCNormAux3.java

TExpAssignSuccess8.fickle

Sub class of field to field assignment

Source
1.     class TCNormAux1 {
2.         class TCNormAux2 extends TCNormAux1 {
3.             class TCNormAux3 extends TCNormAux2 {
4.                 class TCNorm1 {
5.                     TCNormAux1 f1;
6.                     TCNormAux3 f2;
7.                     void m1Norm1() {} {
8.                         this.f2 = new TCNormAux3();
9.                         this.f1 = this.f2;
10.                     }
11.                 }
12.             }
13.         }
14.     }

Carmela output

Compilation successful.

Java files produced
TCNormAux1.java
TCNorm1.java
TCNormAux2.java
TCNormAux3.java

Type Incorrect Cases

TExpAssignFail1.fickle
Null assignment to integer field

Source

1. class TCNorm1 {
2.     int f1;
3.     void miNorm1() {} {
4.         this.f1 = null; //Error
5.     }
6. }

Carmela output

Error Found: Incompatible types expecting int but found [null].
Line: 4
Column: 21
this.f1 = null; //Error

There was 1 error and 0 warnings during compilation.

Java files produced

No .java files produced

TExpAssignFail2.fickle

Class assignment to integer field

Source

1. class TCNormAux1 { }
2. class TCNorm1 {
3.     int f1;
4.     void miNorm1() {} {
5.         this.f1 = new TCNormAux1();
6.     }
7. }

Carmela output

Error Found: Incompatible types expecting int but found TCNormAux1.
Line: 5
Column: 21
this.f1 = new TCNormAux1();

There was 1 error and 0 warnings during compilation.

Java files produced

TCNormAux1.java
TExpAssignFail3.fickle

Boolean assignment to integer field

Source
1. class TCNrm1 {
2.     int f1;
3.     void m1Nrm1() {} {
4.         this.f1 = true;
5.     }
6. }

*Carmela output*

Error Found: Incompatible types expecting int but found boolean.
Line: 4
Column: 21
    this.f1 = true;

There was 1 error and 0 warnings during compilation.

Java files produced

No .java files produced

TExpAssignFail4.fickle

Null assignment to boolean field

Source
1. class TCNrm1 {
2.     boolean f1;
3.     void m1Nrm1() {} {
4.         this.f1 = null; //Error
5.     }
6. }

*Carmela output*

Error Found: Incompatible types expecting boolean but found [null].
Line: 4
Column: 21
    this.f1 = null; //Error

There was 1 error and 0 warnings during compilation.
Java files produced
No .java files produced

TExpAssignFail5.fickle

Class assignment to boolean field

Source
1. class TNormAux1 { }
2. class TNorm1 {
3.     boolean f1;
4.     void mINorm1() {
5.         this.f1 = new TNormAux1(); //Error
6.     }
7. }

Carmela output

Error Found: Incompatible types expecting boolean but found TNormAux1.
Line: 5
Column: 21
    this.f1 = new TNormAux1(); //Error

There was 1 error and 0 warnings during compilation.

Java files produced
TNormAux1.java

TExpAssignFail6.fickle

Integer assignment to boolean field

Source
1. class TNorm1 {
2.     boolean f1;
3.     void mINorm1() {
4.         this.f1 = 1; //Error
5.     }
6. }

Carmela output

Error Found: Incompatible types expecting boolean but found int.
Line: 4
Column: 21

119
Implementing Fickle

C.1. Static Cases

this.f1 = 1; //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TExpAssignFail7.fickle

Incompatible class assignment to field

Source
1. class TCNormAux1 { }
2. class TCNormAux2 { }
3. class TCNorm1 {
4.     TCNormAux1 f1;
5.     void m1Norm1() {} {
6.         this.f1 = new TCNormAux2(); //Error
7.     }
8. }

Carmela output

Error Found: Incompatible types expecting TCNormAux1 but found TCNormAux2.
   Line: 6
   Column: 21
     this.f1 = new TCNormAux2(); //Error

There was 1 error and 0 warnings during compilation.

Java files produced
TCNormAux1.java
TCNormAux2.java

C.1.5 Expressions - Field Access

Type Correct Cases

TExpFieldAssSuccess1.fickle

Single level select field expression

Source
1. class TCNorm1 {
2.     int f1;
3.     void m1() {} {

120
4.      this.f1++;
5.   }  
6. }

*Carmela* output

Compilation successful.

Java files produced

TCNorm1.java

TExpFieldAssSuccess2.fickle

Double level select field expression

Source

1. class TCNormAux1 {
2.   int f1;
3. }
4. class TCNorm1 {
5.    TCNormAux1 f1;
6.    void m1() {} {
7.        this.f1 = new TCNormAux1();
8.        this.f1.f1++;
9.    }  
10. }

*Carmela* output

Compilation successful.

Java files produced

TCNormAux1.java
TCNorm1.java

*Type Incorrect Cases*

TExpFieldAssFail1.fickle

Field does not exist.

Source

1. class TCNorm1 {
2.    void m1() {
3.        this.f1++; //Error
4.    }  
5. }
Carmela output

Error Found: Cannot find identifier f1 in TCNorm1.
Line: 3
Column: 18
        this.f1++; //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TEExpRetLitSuccess1.fickle

Boolean return

Source
1. class TCNorm1 {
2.     boolean m1() {} {
3.         return true;
4.     }
5. }

Carmela output

Compilation successful.

Java files produced
TCNorm1.java

TEExpRetLitSuccess2.fickle

Integer return.

Source
1. class TCNorm1 {
2.     int m1() {} {
3.         return 1;
4.     }
5. }

Carmela output

Compilation successful.
Java files produced
TCNorm1.java

C.1.6 Expressions - Return Expressions
Type Incorrect Cases

TExpRetLitFail1.fickle

Type incorrect return, integer expected.

Source
1. class TCNorm1 {
2.     int m1() {
3.         return true; //Error
4.     }
5. }

Carmela output

Error Found: Incompatible types expecting int but found boolean.
Line: 3
Column: 20
    return true; //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced

TExpRetLitFail2.fickle

Type incorrect return, boolean expected.

Source
1. class TCNorm1 {
2.     boolean m1() {
3.         return 1;
4.     }
5. }

Carmela output

Error Found: Incompatible types expecting boolean but found int.
Line: 3

123
Implementing 

\[ \text{Column: 20} \]
\[
\begin{align*}
\text{return 1;} 
\end{align*}
\]

There was 1 error and 0 warnings during compilation.

Java files produced

No .java files produced

**C.1.7 Expressions - Methods**

*Type Correct Cases*

TExpMethodSuccess1.fickle

Method call with literal parameter

**Source**

1. root class TCRoot1 {
2. int f1;
3. }
4. state class TCState1 extends TCRoot1 {
5. int m1(int x) {TCRoot1} {
6. this.m2(1);
7. return 1;
8. }
9. }
10. int m2(int x) {TCRoot1} {
11. this!!TCState2;
12. return 1;
13. }
14. }
15. state class TCState2 extends TCState1 {
16. }

**Carmaela output**

Compilation successful.

Java files produced

TCRoot1.java
TCState1.java
TCState2.java

TExpMethodSuccess2.fickle
C.1. Static Cases

Method call with class parameter

Source

1. root class TCRoot1 {
2.     int f1;
3. }
4. state class TCState1 extends TCRoot1 {
5.     int m1(int x) {TCRoot1} {
6.         this.m2(this);
7.         return 1;
8.     }
9. }
10. int m2(TCRoot1 x) {TCRoot1} {
11.     this!!TCState2;
12.     return 1;
13. }
14. }
15. state class TCState2 extends TCState1 {
16. }

Carmela output

Compilation successful.

Java files produced

TCRoot1.java
TCState1.java
TCState2.java

TExpMethodSuccess3.fickle

Method call with subclass parameter

Source

1. class TCNrm1 {
2. }
3. root class TCRoot1 extends TCNrm1 {
4.     int f1;
5. }
6. state class TCState1 extends TCRoot1 {
7.     int m1(int x) {TCRoot1} {
8.         this.m2(this);
9.         return 1;
10.     }
11. }
12. int m2(TCNrm1 x) {TCRoot1} {
13.     this!!TCState2;
14.     return 1;
15. }

125
16. }
17. state class TCState2 extends TCState1 {
18. }

Carmela output

Compilation successful.

Java files produced
TCNorm1.java
TCRoot1.java
TCState1.java
TCState2.java

Type Incorrect Cases

TEExpMethodFail1.fickle

Method call with invalid literal parameter

Source
1. root class TCRoot1 {
2.     int f1;
3. }
4. state class TCState1 extends TCRoot1 {
5.     int m1(int x) {TCRoot1} {
6.         this.m2(2); // Error
7.         return 1;
8.     }
9. }
10. int m2(boolean x) {TCRoot1} {
11.     this!!TCState2;
12.     return 1;
13. }
14. }
15. state class TCState2 extends TCState1 {
16. }

Carmela output

Error Found: A method with signature m2(int) was not found in TCState1.
Line: 6
Column: 22
    this.m2(2); // Error

There was 1 error and 0 warnings during compilation.
Java files produced

TCRoot1.java

TExpMethodFail2.fickle

Method call with invalid class parameter

Source
1. class TCNorm1 {
2. }
3.
4. root class TCRoot1 extends TCNorm1{
5.   int f1;
6. }
7. state class TCState1 extends TCRoot1 {
8.   int m1(int x) {TCRoot1} {
9.     this.m2(new TCNorm1()); //Error
10.    return 1;
11. }
12.
13.   int m2(TCRoot1 x) {TCRoot1} {
14.     this!!TCState2;
15.    return 1;
16. }
17. }
18. state class TCState2 extends TCState1 {
19. }

Carmela output

Error Found: A method with signature m2(TCNorm1) was not found in TCState1.
Line: 9
Column: 22
   this.m2(new TCNorm1()); //Error

There was 1 error and 0 warnings during compilation.

Java files produced

TCNorm1.java
TCRoot1.java

TExpMethodFail3.fickle

Wrong number of arguments

Source
1. root class TCRoot1 {
2. int f1;
3. }
4. state class TCState1 extends TCRoot1 {
5.     int m1(int x) {TCRoot1} {
6.         this.m2(2, 2); // Error
7.         return 1;
8.     }
9. }
10. int m2(int a1) {TCRoot1} {
11.     this!!TCState2;
12.     return 1;
13. }
14. }
15. state class TCState2 extends TCState1 {
16. }

Carmela output

Error Found: Wrong number of arguments for the call to m2, there should be 1.
Line: 6
Column: 22
   this.m2(2, 2); // Error

There was 1 error and 0 warnings during compilation.

Java files produced
TCRoot1.java

C.1.8 Expressions - Re-classification

Type Correct Cases

TExpReclassSuccess1.fickle

Re-classification from one class to another

Source
1. root class TCRoot1 {
2.     int f1;
3.     void m1() {TCRoot1} {
4. }
5. state class TCState1 extends TCRoot1 {
6.     void m1() {TCRoot1} {
7.         this!!TCState2;
8.     }
9. }
10. state class TCState2 extends TCState1 {
11. }
Carmela output

Compilation successful.

Java files produced
TCRoot1.java
TCState1.java
TCState2.java

TEExpReclassSuccess2.fickle

Re-classification from one class to another and back

Source
1. root class TCRoot1 {
2.       int f1;
3.       void m1() {TCRoot1} {} }
4. }
5. state class TCState1 extends TCRoot1 {
6.       void m1() {TCRoot1} {
7.             this!!TCState2;
8.             this!!TCState1;
9.         }
10. }
11. state class TCState2 extends TCState1 {
12. }

Carmela output

Compilation successful.

Java files produced
TCRoot1.java
TCState1.java
TCState2.java

Type Incorrect Cases

TEExpReclassFail1.fickle

Effect not correct for re-classification

Source
1. root class TCRoot1 {
2.       int f1;
3.       void m1() {} {}
4. }
5. state class TCState1 extends TCRoot1 {
6.     void m1() {} {
7.         this!!TCState2;   //Error
8.     }
9. }
10. state class TCState2 extends TCState1 {
11. }

Carmela output

Warning: The effect for method m1 does not contain type TCRoot1
Line: 7
Column: 13
     this!!TCState2;   //Error

There where 0 errors and 1 warnings during compilation.

Java files produced
TCRoot1.java
TCState1.java
TCState2.java

TExpReclassFail2.fickle

Receiver not a state class

Source
1. class TCNorm1 {
2.     void m1() {} {
3.         this!!TCState2; //Error
4.     }
5. }

Carmela output

Error Found: The receiver of re-classification must be a state class.
Line: 3
Column: 17
     this!!TCState2; //Error

There was 1 error and 0 warnings during compilation.

Java files produced
No .java files produced
TExpReclassFail3.fickle

Different least root classes

Source

1. root class TCRoot1 {
2.     void m1() {} {
3. }
4.
5. root class TCRoot2 {
6.     int f1;
7.     void m1() {} {
8. }
9. state class TCState1 extends TCRoot1 {
10.    void m1() {} {
11.        this!!TCState2; //Error
12. }
13. }
14. state class TCState2 extends TCRoot2 {
15. }

Carmela output

Error Found: The receiver and target do not have the same least root class.
Line: 11
Column: 17
     this!!TCState2; //Error

There was 1 error and 0 warnings during compilation.

Java files produced
TCRoot1.java
TCRoot2.java

C.2 Dynamic Cases

TDynamic1.fickle

Check transmission of fields under re-classification

Source

1. root class TCRoot1 {
2.     void m1(VarStorage vS) {TCRoot1} {} 
3. }
4.
5. state class TCState1 extends TCRoot1 {
Implementing *Fieldte*  

C.2. Dynamic Cases

6. 
   void m1(VarStorage vs) {TCRoot1} {
7.      this!!TCState2;
8.      vs.var1 = 10; // Indicates we are in TCState1
9.   }
10. }
11. }
12. state class TCState2 extends TCRoot1 {
13.      void m1(VarStorage vs) {TCRoot1} {
14.         vs.var1 = 20; // Indicates we are in TCState2
15.      }
16.   }
17. }
18. class Main {
19. }
20. }
21.   boolean test() {
22.      TCRoot1 a1;
23.      VarStorage vs1;
24.      VarStorage vs2;
25.      a1 = new TCState1();
26.      vs1 = new VarStorage();
27.      vs2 = new VarStorage();
28.      vs1.init(); vs2.init();
29.      vs1.var2 = 10;
30.      a1.m1(vs1);
31.      // The execution of method m1 in TCState1
32.      // should have set vs1.var1 = 10;
33.      vs2.var2 = 20;
34.      a1.m1(vs2);
35.      // a1 should have been re-classified by first call
36.      // to m1 and hence vs2.var1 should be set 20 this
37.      // time
38.      if (vs1.var1 == vs1.var2 && vs2.var1 == vs2.var2)
39.         { System.out.println("Test Passed");
40.            return true;
41.         } else {
42.            System.out.println("Test Failed");
43.            return false;
44.         }
45.     }
46.   }
47. }
48. }
49. class VarStorage {
50.      int var1;
51.      int var2;
52.      int var3;
53.     }
54.     void init() {} {
55.      // We make sure that initially these values
56.      // are different.
57.      this.var1 = 1;
58.      this.var2 = 2;
Implementing Fickle

60.     this.var3 = 3;
61. }
62.
63.     boolean check2() {} {
64.         if (this.var1 == this.var2) {
65.             System.out.println("Passed");
66.             return true;
67.         } else {
68.             System.out.println("Failed, results:"+this.var1+","+this.var2);
69.             return false;
70.         }
71.     }
72. }
73.     boolean check3() {} {
74.         if (this.var1 == this.var2 && this.var1 == this.var3) {
75.             System.out.println("Passed");
76.             return true;
77.         } else {
78.             System.out.println("Failed");
79.             System.out.println("Results:"+this.var1+","+this.var2+"","+this.var3);
80.             return false;
81.         }
82.     }
83. }
84. }

Carmela output

Compilation successful.

Java files produced

TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic2.fickle

Check Transmission of fields under re-classification

Source

1. root class TCRoot1 {
2.     void m1(VarStorage vs) {TCRoot1};
3.     void m2(VarStorage vs) {};
4. }
5.
6. state class TCState1 extends TCRoot1 {
7.     void m1(VarStorage vs) {TCRoot1} {
8.         this!!TCState2;
9.    this.f1State2 = 10;
10.   vS.var1 = this.f1State2;
11. }
12. void m2(VarStorage vS) {} {} 
13. }
14. state class TCState2 extends TCRoot1 {
15.    int f1State2;
16.    void m1(VarStorage vS) {TCRoot1} {}
17.    void m2(VarStorage vS) {} {
18.        vS.var2 = this.f1State2;
19.    }
20. }
21. class Main {
22.    boolean test() {} {
23.        TCRoot1 t1;
24.        VarStorage vS;
25.        t1 = new TCState1();
26.        vS = new VarStorage();
27.        vS.init();
28.        t1.m1(vS);
29.        t1.m2(vS);
30.        return vS.check2();
31.    }
32. }
33. class VarStorage {
34.    int var1;
35.    int var2;
36.    int var3;
37. }
38. void init() {} {
39.    // We make sure that initially these values
40.    // are different.
41.    this.var1 = 1;
42.    this.var2 = 2;
43.    this.var3 = 3;
44. }
45.    boolean check2() {} {
46.        if (this.var1 == this.var2 ) {
47.            System.out.println("Passed");
48.            return true;
49.        } else {
50.            System.out.println("Failed, results:"+this.var1+","+this.var2);
51.            return false;
52.        }
53.    }
54. }
55. }
56. }
57. boolean check3() {} {
58.    if (this.var1 == this.var2 && this.var1 == this.var3) {
59.        System.out.println("Passed");
60.        return true;
63. } else {
64.     System.out.println("Failed");
65.     System.out.println("Results:"+this.var1+","+this.var2+","+this.var3);
66.     return false;
67. }
68. }
69.
70. }
71.

Carmella output

Compilation successful.

Java files produced
TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic3.fickle

Checking preservation of field values in root

Source

1. root class TCRoot1 {
2.     int f1Abs;
3.     void m1(VarStorage vS) {TCRoot1}{}
4. }
5.
6. state class TCState1 extends TCRoot1 {
7.     void m1(VarStorage vS) {TCRoot1}{
8.         this.f1Abs = 10; // Set initial value
9.         vS.var1 = this.f1Abs;
10.        this!!TCState2; // Now in TCState2
11.        vS.var2 = this.f1Abs;
12.     }
13. }
14.
15. state class TCState2 extends TCRoot1 {
16. }
17.
18. class Main {
19.
20.    boolean test() {} {
21.        TCRoot1 t1;
22.        VarStorage vS;
23.        t1 = new TCState1();
24.        vS = new VarStorage();
25.    }
26. }
27.
28.
Implementing Field

```java
25.     vs.init();
26.     ti.ml(vs);
27.     return vs.check2();
28. }
29. }
30. }
31. class VarStorage {
32.     int var1;
33.     int var2;
34.     int var3;
35. }
36.     void init() {} {
37.         // We make sure that initially these values
38.         // are different.
39.         this.var1 = 1;
40.         this.var2 = 2;
41.         this.var3 = 3;
42. }
43.     boolean check2() {} {
44.         if (this.var1 == this.var2) {
45.             System.out.println("Passed");
46.             return true;
47.         } else {
48.             System.out.println("Failed, results:" + this.var1 + "," + this.var2);
49.             return false;
50.         }
51.     }
52. }
53. }
54.     boolean check3() {} {
55.         if (this.var1 == this.var2 && this.var1 == this.var3) {
56.             System.out.println("Passed");
57.             return true;
58.         } else {
59.             System.out.println("Failed");
60.             System.out.println("Results:" + this.var1 + "," + this.var2 + "," + this.var3);
61.             return false;
62.         }
63.     }
64. }
65. }
66. }
```

Carmela output

Compilation successful.

Java files produced

TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic4.fickle

Checking preservation of field values in root class with two re-classifications.

Source
1. root class TCRoot1 {
2.    int f1Abs;
3.    void m1(VarStorage vS) {TCRoot1{ }
4.    }
5. }
6. state class TCState1 extends TCRoot1 {
7.    void m1(VarStorage vS) {TCRoot1{
8.        this.f1Abs = 10; // Set initial value
9.        vS.var1 = this.f1Abs;
10.       this!!TCState2; // Now in TCState2
11.       vS.var2 = this.f1Abs;
12.       this!!TCState1; // Back in TCState1
13.       vS.var3 = this.f1Abs;
14.     }
15. }
16. }
17. state class TCState2 extends TCRoot1 {
18. }
19. class Main {
20. }
21.   boolean test() {} {
22.     TCRoot1 t1;
23.     VarStorage vS;
24.     t1 = new TCState1();
25.     vS = new VarStorage();
26.     vS.init();
27.     t1.m1(vS);
28.     return vS.check3();
29. }
30. }
31. }
32. class VarStorage {
33.    int var1;
34.    int var2;
35.    int var3;
36. }
37.   void init() {} {
38.     // We make sure that initially these values
39.     // are different.
40.     this.var1 = 1;
41.     this.var2 = 2;
42.     this.var3 = 3;
43. }
44. }
45.   boolean check2() {} {
46. if (this.var1 == this.var2) {
47.     System.out.println("Passed");
48.     return true;
49. } else {
50.     System.out.println("Failed, results:"+this.var1+","+this.var2);
51.     return false;
52. }
53. }
54.
55. boolean check3() {
56.     if (this.var1 == this.var2 && this.var1 == this.var3) {
57.         System.out.println("Passed");
58.         return true;
59.     } else {
60.         System.out.println("Failed");
61.         System.out.println("Results:"+this.var1+","+this.var2+"","+this.var3);
62.         return false;
63.     }
64. }
65. }
66. }

Carmela output

Compilation successful.

Java files produced
TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic5.fickle

Normal class as superclass with field access under aliasing

Source
1. class TCNorm1 {
2.     int f1Norm1;
3. }
4. root class TCRoot1 extends TCNorm1 {
5. }
6.
7. state class TCState1 extends TCRoot1 {
8. }
9.
10. class Main {
11.     boolean test() {
12.         VarStorage vS;
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
Implementing Field

C.2. Dynamic Cases

13. TNorm1 n1;
14. TRoot1 a1;
15. vS = new VarStorage();
16. vS.init();
17. a1 = new TCState1();
18. n1 = a1;
19. n1.f1Norm1 = 10;
20. vS.var1 = n1.f1Norm1;
21. vS.var2 = a1.f1Norm1;
22. return vS.check2();
23. }
24. } 
25. class VarStorage {
26. int var1;
27. int var2;
28. int var3;
29. 
30. void init() {} {
31. // We make sure that initially these values 
32. // are different.
33. this.var1 = 1;
34. this.var2 = 2;
35. this.var3 = 3;
36. }
37. 
38. boolean check2() {} {
39. if (this.var1 == this.var2) {
40. System.out.println("Passed");
41. return true;
42. } else {
43. System.out.println("Failed, results:"+this.var1+","+this.var2);
44. return false;
45. }
46. }
47. 
48. boolean check3() {} {
49. if (this.var1 == this.var2 & this.var1 == this.var3) {
50. System.out.println("Passed");
51. return true;
52. } else {
53. System.out.println("Failed");
54. System.out.println("Results:"+this.var1+","+this.var2+","+this.var3);
55. return false;
56. }
57. }
58. 
59. }

Carmela output

Compilation successful.

Java files produced
Implementing Fickle

C.2. Dynamic Cases

TCNorm1.java
TCRoot1.java
TCState1.java
Main.java
VarStorage.java

TDynamic6.fickle

Normal class as superclass with subclass re-classification with field access of normal class.

Source

1. class TCNorm1 {
2.    int f1Norm1;
3. }
4.
5. root class TCRoot1 extends TCNorm1 {
6.    void m1(VarStorage vS) {TCRoot1};
7. }
8.
9. state class TCState1 extends TCRoot1 {
10.   void m1(VarStorage vS) {TCRoot1} {
11.      this.f1Norm1 = 10;
12.      vS.var1 = this.f1Norm1;
13.      this!!TCState2;
14.      vS.var2 = this.f1Norm1;
15.  }
16. }
17.
18. state class TCState2 extends TCRoot1 {
19.   void m1(VarStorage vS) {TCRoot1} {
20.   }
21. }
22.
23. class Main {
24.     boolean test() {
25.        VarStorage vS;
26.        TCRoot1 a1;
27.        vS = new VarStorage();
28.        vS.init();
29.        a1 = new TCState1();
30.        a1.m1(vS);
31.        return vS.check2();
32.     }
33. }
34. }
35. class VarStorage {
36.    int var1;
37.    int var2;
38.    int var3;
39. }
40.     void init() {} {
41.       // We make sure that initially these values
42.       // are different.
43.     }
Implementing *Fickle*  

C.2. Dynamic Cases

```java
43.    this.var1 = 1;
44.    this.var2 = 2;
45.    this.var3 = 3;
46. }
47.
48.    boolean check2() {} {
49.        if (this.var1 == this.var2) {
50.            System.out.println("Passed");
51.            return true;
52.        } else {
53.            System.out.println("Failed, results:"+this.var1+","+this.var2);
54.            return false;
55.        }
56.    }
57.
58.    boolean check3() {} {
59.        if (this.var1 == this.var2 && this.var1 == this.var3) {
60.            System.out.println("Passed");
61.            return true;
62.        } else {
63.            System.out.println("Failed");
64.            System.out.println("Results:"+this.var1+","+this.var2+"","+this.var3);
65.            return false;
66.        }
67.    }
68.
69. }

Carmela output

Compilation successful.

Java files produced

TCNorm1.java
TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic7.fickle

Check correct method called after re-classification

Source

1. root class TCRoot1 {
2.     void mi(VarStorage vS) {TCRoot1} {}  
3. }
4.
5. state class TCState1 extends TCRoot1 {

141
6.    void m1(VarStorage vs) {TCRoot1} {
7.        vs.var1 = 10; // Indicates we are in TCState1
8.        this!!TCState2;
9.    }
10. }
11. }
12. state class TCState2 extends TCRoot1 {
13.    void m1(VarStorage vs) {TCRoot1} {
14.        vs.var1 = 20; // Indicates we are in TCState2
15.    }
16. }
17. }
18. class Main {
19.    
20.        boolean test() {
21.            TCRoot1 a1;
22.            VarStorage vs1;
23.            VarStorage vs2;
24.            a1 = new TCState1();
25.            vs1 = new VarStorage();
26.            vs2 = new VarStorage();
27.            vs1.init(); vs2.init();
28.            vs1.var2 = 10;
29.            a1.m1(vs1);
30.            // The execution of method m1 in TCState1
31.            // should have set vs1.var1 = 10;
32.            vs2.var2 = 20;
33.            a1.m1(vs2);
34.            // a1 should have been re-classified by first call
35.            // to m1 and hence vs2.var1 should be set 20 this
36.            // time
37.            if (vs1.var1 == vs1.var2 && vs2.var1 == vs2.var2) {
38.                System.out.println("Test Passed");
39.                return true;
40.            } else {
41.                System.out.println("Test Failed");
42.                return true;
43.            }
44.        }
45.    }
46. }
47. }
48. }
49. }
50. class VarStorage {
51.    
52.        int var1;
53.        int var2;
54.        int var3;
55.    }
56.    void init() {} {
57.        // We make sure that initially these values
58.        // are different.
59.        this.var1 = 1;
60.        this.var2 = 2;
Implementing Fickle

C.2. Dynamic Cases

60.     this.var3 = 3;
61. }
62. 
63.     boolean check2() {} {
64.         if (this.var1 == this.var2) {
65.             System.out.println("Passed");
66.             return true;
67.         } else {
68.             System.out.println("Failed, results:"+this.var1+"","+this.var2);
69.             return false;
70.         }
71.     }
72. 
73.     boolean check3() {} {
74.         if (this.var1 == this.var2 && this.var1 == this.var3) {
75.             System.out.println("Passed");
76.             return true;
77.         } else {
78.             System.out.println("Failed");
79.             System.out.println("Results:"+this.var1+"","+this.var2+"","+this.var3);
80.             return false;
81.         }
82.     }
83. 
84. }

Carmela output

Compilation successful.

Java files produced

TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic8.fickle

Check method parameter under re-classification

Source
1. root class TCRoot1 {
2.     int f1Abs;
3.     void mi(VarStorage vS, int x) {TCRoot1}{}
4. }
5. 
6. state class TCState1 extends TCRoot1 {
7.     void mi(VarStorage vS, int x) {TCRoot1}{}
8.     vS.var1 = x;
9. this!!TCState2; // Now in TCState2
10. vS.var2 = x;
11. }
12. }
13. }
14. state class TCState2 extends TCRoot1 {
15. }
16. class Main {
17. }
18. boolean test() {} {
19. TCRoot1 t1;
20. VarStorage vS;
21. t1 = new TCStatei();
22. vS = new VarStorage();
23. vS.init();
24. t1.m1(vS,5);
25. return vS.check2();
26. }
27. }
28. class VarStorage {
29. int var1;
30. int var2;
31. int var3;
32. }
33. void init() {} {
34. // We make sure that initially these values
35. // are different.
36. this.var1 = 1;
37. this.var2 = 2;
38. this.var3 = 3;
39. }
40. }
41. boolean check2() {} {
42. if (this.var1 == this.var2 ) {
43. System.out.println("Passed");
44. return true;
45. } else {
46. System.out.println("Failed, results:"+this.var1+","+this.var2);
47. return false;
48. }
49. }
50. }
51. boolean check3() {} {
52. if (this.var1 == this.var2 & this.var1 == this.var3) {
53. System.out.println("Passed");
54. return true;
55. } else {
56. System.out.println("Failed");
57. System.out.println("Results:"+this.var1+"","+this.var2+","+this.var3);
58. return false;
59. }
60. }
61. }
62. }
Implementing Fickle

C.2. Dynamic Cases

Carmina output

Compilation successful.

Java files produced

TCRoot1.java
VarStorage.java
TCState1.java
TCState2.java
Main.java

TDynamic9.fickle

Alias Testing

Source

1. class Store {
2.    A val;
3.    
4.        void setVal(A a) {
5.            this.val = a;
6.        }
7.        
8.        A getVal() {
9.            return this.val;
10.        }
11.        
12.        void setInternal(int i) {
13.            this.val.ai = i;
14.        }
15.        
16.    }
17.    
18.    root class A {
19.        int ai;
20.        boolean change() {A} { return false; }
21.    }
22.    
23.    state class B extends A {
24.        boolean change() {A} {
25.            // Set the store up.
26.            Store s;
27.            s = new Store();
28.            this!!B.ai++;
29.            this!!C;
30.            this.ai = 5;
31.            s.setVal(this);
32.            this!!C;
33.            this!!B;
34.            (this).ai = 10;
Implementing Fickle

```java
35.     System.out.println("al from this is:"+this.al);
36.     System.out.println("al from store object is:"+s.getVal().al);
37.     return this.al == s.getVal().al;
38. }
39.
40. }
41.
42. state class C extends A {
43. }
44.
45. class Main {
46.     boolean test() {
47.         A a;
48.         a = new B();
49.         return a.change();
50.     }
51. }
```

Carmela output

Compilation successful.

Java files produced

Store.java
A.java
B.java
C.java
Main.java

TDynamic10.fickle

Alias Testing with re-classification passed as parameter.

Source

```java
1. class Store {
2.     A val;
3. }
4.     void setValue(A a) {
5.         this.val = a;
6.     }
7.     A getValue() {
8.         return this.val;
9.     }
10. }
11. }
12.     void setInternal(int i) {
13.         this.val.al = i;
14.     }
15. }
16. }
```
Implementing Fields

C.2. Dynamic Cases

17.
18. root class A {
19.     int a;
20.
21.     boolean change() {A} { return false; }
22. }
23.
24. state class B extends A {
25.     boolean change() {A} {
26.         // Set the store up.
27.         Store s;
28.         s = new Store();
29.         this!!B;
30.         this!!C;
31.         this.a1 = 5;
32.
33.         s.setVal(this!!B);
34.         this!!C;
35.         this!!B;
36.         (this).a1 = 10;
37.         System.out.println("a1 from this is:"+this.a1);
38.         System.out.println("a1 from store object is:"+s.getVal().a1);
39.         return this.a1 == s.getVal().a1;
40.     }
41. }
42. }
43.
44. state class C extends A {
45. }
46.
47. class Main {
48.     boolean test() {
49.         A a;
50.         a = new B();
51.         return a.change();
52.     }
53. }

Carmela output

Compilation successful.

Java files produced

Store.java
A.java
B.java
C.java
Main.java
Bibliography


