Mocha: Type Inference for Java

by

Christopher John Male

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2009

Abstract

Static typing allows type errors to be revealed at compile time, which reduces the amount of maintenance and debugging that must occur later on. However, often the static type information included in source code is duplicate or redundant. Type inference resolves this issue by inferring the types of variables from their usage, allowing much of the type information to be omitted. This thesis formally describes Mocha, an extension to Java, which supports the inference of both local variable and field types. Mocha has unique support for local variables having different types at different program points which allows variables to be retyped due to certain conditional statements. It also includes a procedure for handling the effect Java exceptions have on the type inference process.

Acknowledgments

I would like to thank my partner Lisa for providing constant support during even the most difficult times, Dr David Pearce for once again being an inspirational mentor and my family for the many suggestions they provided.

I am also grateful for the Victoria University Masters with Thesis scholarship without which I would have been unable to undertake this work.

Contents

Chapter 1

Introduction

1.1 Static Typing

A programming language's *type system* defines how *types* are constructed and manipulated in the language. Types are vital to programming languages because they give meaning to statements, expressions and values in the language. With types, a program can be *type-checked* to see if certain properties in a program are true. Two common approaches adopted by programming languages are static and dynamic typing. Examples of statically typed languages are Java, C/C++ and Haskell, while Perl, Python, Ruby and Smalltalk are examples of dynamically typed languages.

In a statically typed language, the type of every variable and expression in a program is known at compile time. Armed with this type information, a compiler for a statically typed language is able to *type-check* source code and report type-errors at compile time. Reporting errors at this stage is considered advantageous since it can reduce the amount of maintenance and debugging that must occur later on.

In a dynamically typed language, the types of variables and expressions are not necessarily known at compile time. Instead they are determined when necessary during runtime. Because the types of variables and expressions are unknown at compile time, compilers of dynamically typed languages are unable to report as many type-errors at that stage. Any type-errors that do occur are reported at runtime instead.

In most statically typed languages, the static type information for a program must be included in its source code. For example, in Java, the static types of every field and local variable must be stated in source code. However often the static type information included is duplicate or redundant. This redundancy in information can increase the verbosity of the source code, which in turn reduces the code's readability and maintainability.

1.2 Local Variable Type Inference

Type inference is a technique that alleviates the burdens found in a statically typed language while maintaining its benefits, by inferring the static types of variables from their usage. This allows the static types to then be omitted from the source code. Type inference is not included in Java, meaning the programmer must provide full type information in all places. This thesis presents Mocha, an extension to Java that supports local variable type inference thereby reducing the verbosity of source code.

As an illustration of the effect Mocha has on the amount of static type information that needs to be included in source code, consider the following simple local variable declaration written in Java:

```
ArrayList<String> list = new ArrayList<String>();
```

In Mocha, the same declaration becomes:

```
var list = new ArrayList<String>();
```

Here, the keyword `var` has replaced `list`'s static type which indicates to Mocha that its static type must be inferred. By removing the redundant static type information, the declaration has become "cleaner" and easier to read.

Key to Mocha is its support for variables having different types at different program points, which allows Mocha to retype variables due to `instanceof` conditionals. Consider the following example:

```
public void f(Object x) {
  if(x instanceof Integer) {
    x.intValue();
  }
}
```

In this example, Mocha has analysed the conditional and has retyped `x` to the type it is an instance of, which in this case is `Integer`. This means that the method `intValue()` can be invoked immediately. If such an example were written in Java, it would be necessary to cast `x` to `Integer` before it could be used. Hence Mocha's retyping has removed the need for a redundant cast.

Other languages such as LinQ, Scala and OCaml also support type inference. However, each requires a local variable to have exactly one static type throughout an entire method. Hence none of these languages are able to retype variables due to conditionals.

1.3 Field Inference

Fields also suffer from redundant static type information and can benefit from type inference. However, because of their very nature, the static types of fields are more difficult to infer. Consequently Mocha's type inference algorithm was extended to include a novel approach for inferring the types of fields based on the values they are assigned in constructors. Consider the following Mocha example:

```
public class FieldInf {
  private var foo;
  public FieldInf(ArrayList<String> a) { foo = a; }
  public FieldInf(LinkedList<String> l) { foo = l; }
}
```

Here, the field `foo` is declared `var`, again indicating to Mocha that its type needs to be inferred. However, unlike local variables, fields can only have a single static type. Because `foo` is assigned multiple values, the inferred type must be a superclass of the types assigned in the constructors. Therefore Mocha would infer the type of `foo` to be `AbstractList-<String>`, which is the direct superclass of `ArrayList` and `LinkedList`.

Mocha's support for both local variable and field type inference reduces the need for redundant and duplicate type information to be included in source code, leaving it less verbose and consequently easier to read and maintain.

1.4 Contributions

The primary contributions of this thesis are:

- The design and implementation of Mocha, an extension of Java which supports the inference of local variable and field types — Chapter 2 and Chapter 3
- An extension to Mocha's local variable type inference algorithm to incorporate the effect of Java exceptions — Chapter 4
- A novel extension to Mocha's local variable type inference algorithm to support the inference of field types — Chapter 4
- The evaluation of Mocha's effectiveness through the compilation of a series of real life benchmarks — Chapter 5

A shortened description of Mocha is also available as a technical report [22]. A release version of Mocha is available along with JKit, an open source kit for developing extensions to the Java language which Mocha is built upon, at `http://homepages.mcs.vuw.ac.nz/~djp/jkit/mocha/`.

Chapter 2

Mocha

Local variable declarations often contain duplicate or redundant type information, which reduces readability and maintainability. In most cases, the static type information can be inferred at compile time. In this chapter a local variable type inference system called Mocha is introduced. Particular attention is given to how local variables can be declared in Mocha, the way in which Mocha handles divergence in program flow and how Mocha analyses `instanceof` conditionals and retypes local variables accordingly. Additionally, Mocha's novel approach to inferring the types of fields is briefly presented, along with a short discussion of the restrictions placed on field retyping.

2.1 Local Variable Type Inference

Mocha removes the need to include redundant type information by the programmer, increasing code readability and maintainability by allowing static types to be omitted from local variable declarations. For example, the Java declaration:

```
ArrayList<Set<Float>> xs = new ArrayList<Set<Float>>();
```

becomes in Mocha:

```
var xs = new ArrayList<Set<Float>>();
```

Here the keyword `var` is used in place of the static type and instead, Mocha infers the type from the right-hand side of the assignment. This reduction in type information makes the program shorter, more readable and hence more maintainable. However, to maintain complete compatibility with existing Java source, Mocha still allows the programmer to provide the static type as before (some situations where this is advantageous will be shown later).

It should be noted that Mocha still requires method parameters and return types be given static types. In Section 2.2.1 and Chapter 4.2, Mocha's support for also allowing fields to be declared with `var` is discussed.

2.1.1 Conditionals

A conditional statement (`if` statement) causes a divergence in program flow. This complicates the type inference process as a variable could be assigned a different type in each flow, as in the following example:

```
var x;
if(...) {
  x = new Float(1.0);
} else {
  x = new Integer(1);
}
... // What is the type of x now?
```

Mocha requires each variable to have exactly one type at any given point in the program, therefore the type of `x` after the `if`-statement should reflect the fact that `x` could be either an `Integer` or a `Float` at that point. One option is simply to give `x` the type `Object` here since both `Integer` and `Float` extend `Object`. However this is not the *most precise type* it could be given by Mocha, since they both extend `Number` (see Figure 2.1).

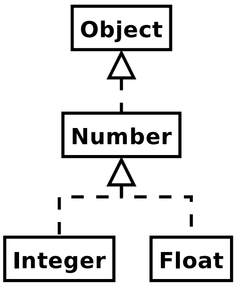


Figure 2.1: Inheritance hierarchy for `Integer` and `Float`. Both extend `Number`, which in turn extends `Object`. Therefore `Number` is the most precise type for a variable which is either an `Integer` or a `Float`.

Instead, Mocha aims to give variables their most precise types in such situations. In this case the most precise type for `x` is `Number`. The notion of *most precise type* is formalised in Section 3.2.4.

2.1.2 Instanceof

Java's conditional operator `instanceof` allows a programmer to check whether an object is an instance of a certain class. The following example illustrates how the `instanceof` conditional can be used:

```
Object obj = ...
if(obj instanceof Integer) {
  // obj is an instance of Integer, but still has type Object
  Integer iobj = (Integer) obj;
  // obj can now be treated as an Integer through iobj
  iobj.intValue();
}
```

At runtime the dynamic type of `obj` is compared with `Integer` to see whether it is actually an instance of that class or one of its subclasses. If this is true, even though `obj` was given the static type `Object` in the source code, the conditional would evaluate true. However, in normal

Java, even if the conditional were to evaluate true, the static type of `obj` would not change. In order to treat it as the type it is known to be (in this case `Integer`), the programmer must explicitly cast the variable to the new type as shown in the example.

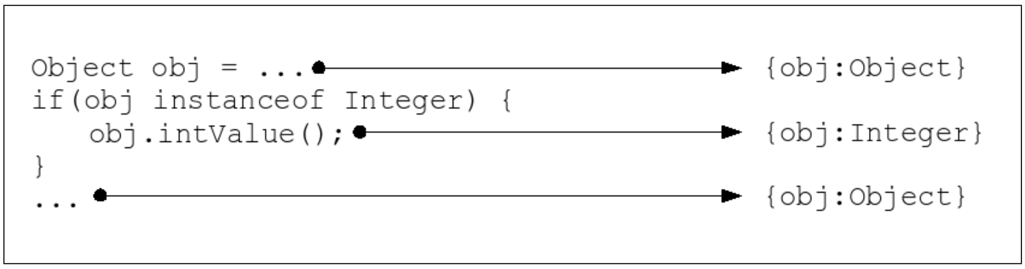
By casting the variable to its new type, the programmer is having to restate what has already been determined to be true. Mocha removes this redundancy by allowing the programmer to use `obj` as if it has acquired the new type `Integer` in the true-branch of the conditional. During its type inference phase, when `instanceof` conditionals are evaluated and where type safe, Mocha's internally recorded type for local variables are changed to the type they are an instance of. Therefore in Mocha the example from before simply becomes:

```
Object obj = ...
if(obj instanceof Integer) {
  // obj can be treated as an Integer straight away
  obj.intValue();
}
```

The variable's new type does not last its entire lifetime however. The variable is only retyped in the true branch of the conditional. Outside of the true branch, the variable maintains its original type, which in the example is `Object`. Figure 2.2 illustrates how `obj`'s type changes over its lifetime due to an `instanceof` conditional by providing the type of `obj` at each point in the code.

2.2 Field Type Inference

While the primary focus of Mocha is local variable type inference, it does include an extension for field type inference. In this section the syntax supported by Mocha for field declarations will be introduced, and the approach used by Mocha to infer the types of fields will be briefly discussed.



```
Object obj = ...
if(obj instanceof Integer) {
   obj.intValue();
}
...
```

Figure 2.2: Example of how a variable's type can change over its lifetime due to an `instanceof` conditional.

After that, the restriction placed on field retyping through `instanceof` conditionals will be described and the use of local variables' instances to bypass the restriction will be evaluated.

2.2.1 Field Declarations

Local variables are not the only variables in normal Java that suffer from redundant type information. For example, a field declaration such as:

```
public class foo {
  HashMap<Integer, ArrayList<String>> f =
    new HashMap<Integer, ArrayList<String>>();
  ...
}
```

could benefit from having its static type inferred. In order to distinguish field declarations from other declarations in a class, Mocha again uses a keyword to replace static type information in field declarations. To maintain consistency with local variables, the keyword `var` is used again. Consequently, the example is simplified in Mocha to:

```
public class foo {
  var f = new HashMap<Integer, ArrayList<String>>();
  ... }
```

Unlike local variables which can have different types at different points in a program, for reasons discussed in Section 2.2.2, the types of fields *must* be fixed. This, coupled with the fact that fields are accessible from multiple methods in a class, means that the static types of fields must be inferred before the local variable type inference process can begin.

Mocha infers the type of a field from the values it is assigned in its class's constructors. This is illustrated in the following example:

```
public class foo {
  public var num;
  public foo(int i) { num = new Integer(i); }
  public foo(float f) { num = new Float(f); }
}
```

As with conditionals, Mocha computes the most precise type from the assignments made in the constructors. In this case, Mocha would infer the type of `i` to be `Number`.

Mocha's field type inference approach is discussed in greater detail in Section 4.2

2.2.2 Field To Local Variable Instance

An interesting issue that affects the `instanceof` retyping introduced in Section 2.1.2 is that field retyping cannot be allowed. To understand this, suppose Mocha allowed fields to be retyped, as in the following example:

```
public class Retype {
  private Object bar;
  public int f() {
    if(bar instanceof Integer) {
      return bar.intValue();
    }
    return 0;
  }
  public void g(Character c) { bar = c; }
}
```

Ideally, bar would be retyped as Integer in the true-branch of the conditional, allowing the method intValue() to be invoked. However this would only be type-safe in a single threaded environment. If a thread were to invoke f at the same time as another thread was invoking g, then bar could be retyped as Integer just as it was assigned the type Character (which would cause a runtime type error). Therefore, to ensure that field types remain consistent in a multi-threaded environment, field types must be fixed.

At the root of this issue is the difference between fields and local variables. Fields are thread-global, meaning multiple threads can affect the same field, while local variables are thread-local meaning a new version exists for each thread. Consequently in order to gain the benefit of a field being retyped in Mocha, it is possible to use a local variable instance of the field instead. In this scenario, a local variable is assigned the field before the field is to be retyped due to an instanceof conditional. The local variable then replaces the field in the conditional and is retyped instead. This process is illustrated in an updated version of the previous example:

```
public class Retype {
  private Object bar;
  public int f() {
    // A local variable instance of bar is created
    var localBar = bar;
    // It then replaces the field in the conditional
    if(localBar instanceof Integer) {
      // localBar is retyped to Integer, so intValue can be invoked
      localBar.intValue();
    }
    return 0;
  }
```

```
    public void g(Character c) { bar = c; }
  }
```

However requiring the programmer to create the local variable goes against Mocha's goal of increasing efficiency. An option that has been considered is whether Mocha could automatically introduce the local variables where necessary. However, this would represent a departure from the semantics of Java and, hence, has not been implemented.

2.3 Example

As mentioned, Mocha improves the readability and maintainability of Java source code. To appreciate this, a more complete example must be considered. The following is a typical example of Java source code. In the example, the method `find` provides support for searching two different data structures, HashMaps and Vectors, using the same search criteria, an integer. For HashMaps, the integer is a key paired with a value, while for Vectors, the integer is an index. The example contains local variables and fields whose static types can be inferred, and an `instanceof` conditional followed by a redundant cast:

```
public class InnerFind {
  private Map<String, Object> map =
        new HashMap<String, Object>();

  public InnerFind(Map<String, Object> m) {
    map = m;
  }

  public Object find(String s, Integer i) {
    Object obj = map.get(s);
    if(obj instanceof Vector) {
      Vector<Object> vec = (Vector<Object>) obj;
      return vec.get(i);
    } else if(obj instanceof HashMap) {
      HashMap<Integer, Object> hmap =
       (HashMap<Integer, Object>) obj;
      return hmap.get(i);
    }
    return null;
  }
}
```

The same example written in Mocha is provided below:

```
public class InnerFind {
  private var map = new HashMap<String, Object>();

  public InnerFind(Map<String, Object> m) {
    map = m;
  }

  public Object find(String s, Integer i) {
    var obj = map.get(s);
    if(obj instanceof Vector) {
      return obj.get(i);
    } else if(obj instanceof HashMap) {
      return obj.get(i);
    }
    return null;
  }
}
```

It is clear that the example written in Mocha is considerably 'cleaner' than the original version. By removing the static type information from the local variable declarations, the declarations are clearer and fewer characters have had to be typed. Furthermore, the retyping of the variable `obj` due to the `instanceof` conditional has saved the programmer from having to create additional variables. Finally, by letting Mocha infer the types of the field `map`, the programmer has again had to type fewer characters, the declarations are again cleaner, and it is unnecessary for the programmer to go back and change the field types if they decide to add additional constructors.

2.3.1 Discussion

It is possible that in certain circumstances Mocha's inference of static types could be detrimental to the readability of source code. Situations can occur where a static type is included by the programmer to communicate something specifically to the reader. An example of this is where a Java class implements multiple interfaces yet the programmer wants a variable assigned an instance of the class to have just one specific type. While providing the specific type as the static type of the variable is not necessary for method call resolution or subtyping, it does communicate to the reader what the programmer intended. By not including the static type, the reader may misinterpret the intention of the programmer.

In practice such situations are incredibly rare, particularly since Mocha still requires method parameters and return types have static types. However if a situation were to occur, the programmer could still include the static type in the source code and Mocha would use this rather than inferring the type.

2.4 Related Work

Various programming languages support local variable type inference. Out of these, LinQ, Scala and OCaml are discussed below. Following that, a summary of other works related to type inference is provided.

2.4.1 LinQ

Language Integrated Query (LinQ) [24] is a Microsoft .Net extension which adds support for object querying. One aspect of LinQ's support for object querying is anonymous types (anonymous since they do not have names). An example of an anonymous type being created in C# is shown below:

```
var x = new {FirstName = "First", LastName = "Last"};
```

As anonymous types do not have names, it would not be possible for the programmer to provide a static type for `x`. Therefore LinQ allows the programmer to use the `var` keyword in place of the static type. The actual static type of the variable would then be inferred from the type it is assigned.

While this process shares many similarities with Mocha's local variable type inference, LinQ only allows a variable to have one static type. Furthermore, a variable whose type is to be inferred must be assigned a value immediately. Assigning the variable another type at a later point does not have any effect on its static type. The implications of this are two fold. First, LinQ *does not* support retyping through `instanceof`-like conditionals (the keyword `as` is used instead `instanceof` in LinQ). Second, the situation described in Section 2.1.1 where a variable whose type is to be inferred is assigned different values in the two branches of a conditional, is also *not* supported.

The only exception to this is through LinQ's support for the ternary operator `?`, which allows conditionals to be inlined as single statements. An example of how the ternary operator can be used with type inference is shown below:

```
public class A { ... }
public class B : A { ... }
public class C : A { ... }
```

```
...
public void f(int i) {
  var x = (i == 0) ? (A) new B() : (A) new C();
}
```

Despite B and C both extending A, in order for A to be the inferred type of x, the programmer must cast both new statements to the type A. If the same example were written in Mocha, this would not be necessary.

2.4.2 Scala

Scala [33] is a language which attempts to bring the functional programming paradigm to Java. Type inference has been part of functional programming languages for many years, so it is unsurprising that it is also supported in Scala in the more reduced form of local variable type inference.

Like LinQ and Mocha, Scala allows the programmer to use the keyword var in place of static types in local variable declarations. However unlike Mocha, Scala restricts each local variable to just one static type during its lifetime. If the static type is to be inferred, then at declaration the variable must be assigned the value the type is to be inferred from.

While Scala does require type-inferred local variables be assigned values at declaration, its support for the functional programming paradigm means that a variable may be assigned the result of a series of statements, rather than just a single expression. This is illustrated in the following example:

```
class A { }
class B extends A { }
class C extends A { }
var y = if(...) {
  new B()
} else {
  new C()
}
```

Like Mocha, Scala does correctly infer the type of y to be the supertype of both B and C, which is A. However, Scala's support for inferring the supertype of two types goes beyond that of Mocha. Consider the following example:

```
trait A { }
trait B { }
class C extends A with B { }
class D extends A with B { }
...
var y = if(...) {
  new C()
} else {
  new D()
}
```

Scala traits are similar to Java interfaces and like interfaces, a Scala class can extend multiple traits. Because both C and D extend A and B, there does not seem to be a unique type that can be given to y. Given a comparable example, Mocha would simply choose either A or B. However Scala creates an *intersection type* which is both A *and* B. The rationale behind Mocha's approach and a discussion on intersection types can be found in Section 3.2.3.

2.4.3 OCaml

OCaml [32] is an object-oriented extension of Caml, a derivative of the ML functional language. In contrast to the other languages discussed, OCaml uses a let based syntax to replace static types in local variable declarations. However like LinQ and Scala, OCaml infers the type of a variable from the value it is immediately assigned at its declaration. Furthermore, a local variable is only allowed a single static type. Since OCaml is a functional language like Scala, the value assigned to a variable may be the result of a series of statements. The following is a simple example of OCaml:

```
class a = object(self) end;;
class b = object(self) end;;
let x = 1;;
let y = if x < 10 then new a else new b;;
```

Here, because the value of x is less than 10, OCaml would infer the type of y to be a.

OCaml's support for type inference is not just limited to local variables. The language includes support for inferring the type of every local variable and expression, including method parameter and return values. The inference algorithm is an interprocedural algorithm based on the Hildney-Milner (HM) algorithm, which involves creating and solving type constraints. Unlike the traditional HM algorithm, OCaml's inference algorithm supports object subtyping, which allows it to handle examples similar to that provided for Scala.

A unique feature of OCaml's inference algorithm is that it infers a great deal of information from the use of operators. For example, the + operator is defined as adding two integers, while the +. operator adds two floats. Consequently, like Java, OCaml does not allow operator overloading.

2.4.4 Java 7

The next version of the Java programming language, Java 7, will include a simple form of type inference for local variable declarations. The new syntax that will be able to used in local variable declarations is illustrated below:

```
HashMap<String, List<String>> map = new HashMap();
```

Here, instead of having to provide the generic types on both sides of the assignment, it is only necessary to provide them in the variable's static type. From the static type, the generics for the newly instantiated HashMap can be easily inferred.

2.4.5 JACK

The Java Annotation Checker (JACK) [23] allows programmers to annotate references with the Java annotation `@NonNull`, which JACK translates into a nullness type and type-checks. The end result is that the programmer can specify which references need to be non-null as to prevent `NullPointerExceptions`. The following is a simple example of it in use:

```
public class NNExample {
  public static void f(@NonNull String s) {
    s.toString();
  }
  public static void main(String[] args) {
    f(null);
  }
}
% jack NNExample
  type-error: NNExample.java:6: parameter 0 not subtype
  of @NonNull String
```

In the example, the programmer has specified that `s` must be non-null so that its dereferencing does not result in an exception. JACK has then identified that the invocation of `f` with the parameter `null` would however result in an exception.

An important feature that influenced Mocha was JACK's analysis of conditionals. Unlike the static types of local variables which can only benefit from the analysis of the `instanceof` conditional, nullness types can benefit from analysing equality and inequality conditionals. An illustration of why is provided below:

```
public void f(String s) {
  if(s != null) {
    s.toString();
  }
}
```

Although s has not been annotated @NonNull, it is clear that it must be non-null when it is dereferenced. Therefore JACK would change the nullness type of s to reflect this knowledge.

2.4.6 Other Work on Type Inference

The most commonly used type inference system was developed by Hindley [14] and Milner [27]. This system, later referred to as *HM*, supports the inference of a completely untyped program. To do so, the system generates type constraints from the program code and solves the constraints using unification. In order to infer types in an entirely untyped program, the inference system must proceed across method invocations. In order to do so, the program's complete call graph for the program must be known. Hence, the *HM* type system is generally not appropriate when modules/classes can be compiled separately. Despite this, many type systems [2, 3, 7, 17, 35, 36, 40] have been developed based on *HM* and are included in many widely used languages such as OCaml.

Local variable type inference based on *dataflow analysis* is already part of Java in the form of the Java Bytecode Verifier [20]. The Bytecode Verifier is responsible for ensuring that the Java bytecode to be executed by the JVM is safe. Since Java bytecode is untyped, the Verifier must infer the types of local variables from the values they are assigned. The algorithm used by the Verifier shares many similarities with that used by Mocha, however there are two prominent differences. First, method resolution occurs in the compilation of Java code, and as such, is not part of the verification process. Second, to handle the problem of the Java class hierarchy not forming a join semi-lattice (this problem is discussed in Section 3.2.3), the Verifier ignores interfaces and instead focuses on the class extends relationship, which does form a join semi-lattice.

Gagnon *et al.* have used type inference to infer the static types of local variables in order to convert Java Bytecode to an intermediate representation [10]. Unlike Mocha, the goal of their type inference system is to infer a single static type for each local variable. In a situation where a local variable is assigned different types at different points in a program (which is allowed since Java Bytecode is untyped), they split the variable into multiple variables, each having one of the assigned types.

Bierman *et al.* formalise the type inference algorithm used in C# 3.0 [4]. The algorithm is considerably different to that used in Mocha, as it employs bidirectional type checking [39], which does not support variables having different types at different program points.

Palsberg *et al.* present an algorithm similar to *HM* for inferring the types of variables and expressions in programs written in a subset of the SmallTalk language [36]. The algorithm involves the creation of the *trace graph* of the program, and type constraints (for each method, and the program as a whole). These constraints are then solved using fixed-point derivation. If a solution exists, then the program is said to be *typable*. Like *HM*, the algorithm does not support classes being compiled in isolation since the entire program is required to generate the complete trace graph.

Wang *et al.* provide a type constraint based algorithm for inferring types in a completely untyped Java program [44]. Like the *HM* type system, the type constraints are generated from the program code and are solved using unification. However the algorithm includes support for data polymorphism, which leads to more precise types being inferred. Despite being an algorithm for Java, no specific attention is given to the effect of exceptions on the inference process.

Lagorio and Zucca [18] use type inference to allow Java programs to be written with *unknown* method parameter and return types. Like other algorithms already discussed, they also generate and solve type constraints. However rather than using a subtype relationship in the creation of the constraints as is done in other algorithms, they focus on whether classes contain certain methods.

Chapter 3

Implementation

This chapter details Mocha's algorithm for inferring the types of local variables. Before that is possible however, JKit [21], a Java development kit on which Mocha is built, is introduced.

3.1 Overview of JKit Compiler

Before Mocha's inference algorithm can begin, the source code that is to be compiled must be parsed and translated into a form that is more appropriate for the algorithm. Mocha's front-end component handles the translation process. This component was built on top of JKit, an open source Java development kit created in conjunction with Dr David Pearce. JKit came out of earlier work also done with Dr Pearce on NonNull Bytecode Verification [23]. As a consequence it is necessary to understand aspects of JKit before proceeding with the explanation of how Mocha is implemented.

Superficially, JKit is just like other Java compilers in that it takes in Java source code and produces Java bytecode. However it was designed with extensibility in mind. Internally, JKit uses a concept called *Pipelines*, which define sets of actions that will occur in the translation of the source code into bytecode. Each action is referred to as a *Stage*. At a minimum, JKit requires a Pipeline to have a Stage for reading/parsing source code,

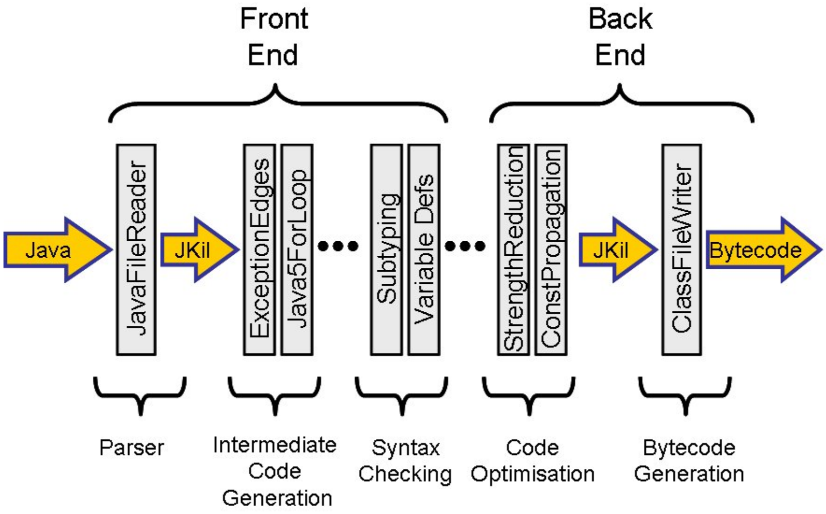


Figure 3.1: Illustration of JKit's default pipeline for compiling normal Java code.

and a Stage for writing the result of the translation. The output does not necessarily have to be Java bytecode, it could be another programming language. Programmers can add or remove additional Stages to tailor JKit to suit their needs.

Figure 3.1 illustrates a typical Pipeline used in JKit which takes normal Java source code and produces Java bytecode in the standard classfile format. As shown, the Pipeline can be conceptually split into two sections, front and back-ends. The remainder of this section will focus on the front-end, particularly the translation of source code into JKit's intermediate language JKil.

3.1.1 Front End

Rather than creating a Java source code parser for JKit, ANTLR [37] was used to automatically produce a parser based on a grammar for standard Java 1.5. In addition to the parsing rules, the parser includes instructions on how to produce an abstract syntax tree (AST) to represent the parsed source code.

ASTs are simple tree data structures which use the parent-child node relationship to represent statements and their expressions. For example, a block of statements would be represented as a `block` node, with each statement as a child. Each statement would in turn be represented by parent and children nodes. Figure 3.2 provides an example of an AST constructed for the following Java code:

```
public Integer find(ArrayList<Object> list) {
  Object obj = list.get(i);
  if(obj instanceof Vector) {
    Vector<Integer> v = (Vector<Integer>) obj;
    return v.get(i);
  }
  return null;
}
```

While ASTs look remarkably similar to the source code from which they are produced, they do not include many language features that are purely syntactic, such as “`;`”, “{” and “}” characters.

The advantage of AST data structures is that they represent source code using an object structure, which can be easily traversed and translated into other data structures.

3.1.2 Intermediate Code Generation

The next step in JKit’s front-end is the generation of an intermediate language representation of the ASTs.

JKit’s intermediate language JKil is an unstructured language which closely resembles Java Bytecode. It includes two primary constructs: statements, *S*, which can change program state and flow; and expressions, *e*, which have no side effects. JKil’s grammar is formally presented in Fig-

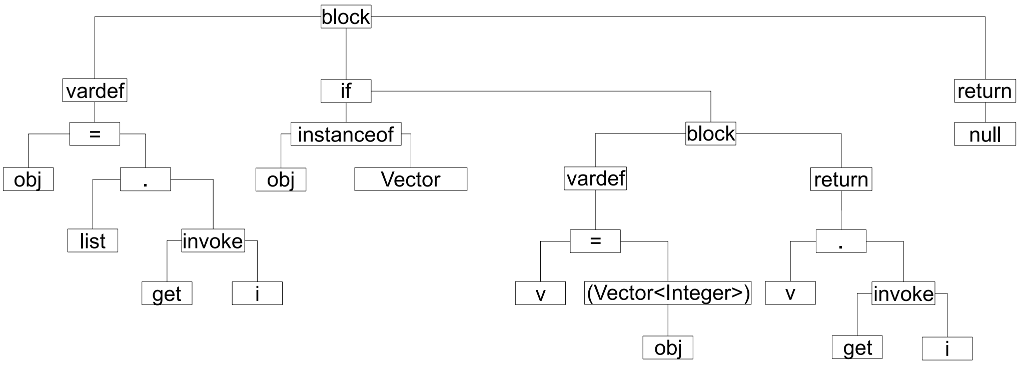


Figure 3.2: An example of an AST constructed during JKit's source code parsing

ure 3.3.

JKil representations are used by JKit since JKil contains fewer language constructs than Java, and those it does contain, more closely match Java bytecode. For example, structured language constructs from Java such as `while` and `for` are represented in JKil as `if` conditions with `goto` statements to produce loops. An example of how a `while` loop is represented in JKil is shown in Figure 3.4.

Another Java construct which is translated by JKit is the ternary operator **?**. In Java, the ternary operator allows `if`-statements to be "inlined" as an expression, rather than being a series of statements.

An example of the ternary operator in use is:

```
return (x != null) ? x : new ArrayList();
```

Although the ternary operator is a useful piece of syntactical sugar, its inclusion in JKil would be redundant since it can be represented using the existing `if` statement. Therefore JKit would translate the example into:

```
L1: if x == null goto L2
    return new ArrayList()
L2: return x
```

v	$\in$	Var	Local variables
c_I	$\in$	$Consts$	Integer Constants $\{\ldots, -1, 0, 1, 2, 3, \ldots\}$
c_R	$\in$	$Consts$	Real Constants $\{\ldots, -1.0, \ldots, 0, \ldots, 1.0, \ldots\}$
c_B	$\in$	$Consts$	Boolean Constants $\{true, false\}$
c_s	$\in$	$Consts$	String Constants {“”, “a”, “ab”, “abc”, ...}
f	$\in$	$Field\ Names$	Field Names
m	$\in$	$Method\ Names$	Method Names
n	$\in$	$Class\ Names$	Class Names
bop	$\in$	*BinOp*	Binary Operators $\{+, *, /, <<, >>, \&\&, \|\|, \&, \|, \ldots\}$
uop	$\in$	*UnOp*	Unary Operators {!,-,∼}
$\texttt{l}_1, \texttt{l}_2$	$\in$	*Lab*	Labels

$$
\begin{array}{lcl}
C & ::= & \texttt{class}\ n\ \texttt{extends}\ n\ \texttt{implements}\ n_0, \ldots, n_x\ \{T f_0, \ldots T f_x\ M_0, \ldots, M_x\} \\
 & | & \texttt{interface}\ n\ \{M_0, \ldots, M_x\} \\
M & ::= & T\ m(T v_0, \ldots T v_x)\ \{T v_0, \ldots T v_x\ S_0, \ldots, S_x\} \\
 & | & T\ m(T v_0, \ldots, T v_x)
\end{array}
$$

$$
\begin{array}{lcl lcl}
S & ::= & [\texttt{l}_1\texttt{:}]\ l = e & e & ::= & uop\ e \\
 & | & [\texttt{l}_1\texttt{:}]\ \texttt{return}\ e & & | & e_1\ bop\ e_2 \\
 & | & [\texttt{l}_1\texttt{:}]\ e.m(e_1, ..., e_n) & & | & e.m(e_1, ..., e_n) \\
 & | & [\texttt{l}_1\texttt{:}]\ \texttt{if}\ e\ \texttt{goto}\ \texttt{l}_2 & & | & \texttt{new}\ n(e_1, ..., e_n) \\
 & | & [\texttt{l}_1\texttt{:}]\ \texttt{lock}\ v & & | & \texttt{new}\ T[e_1] \\
 & | & [\texttt{l}_1\texttt{:}]\ \texttt{unlock}\ v & & | & e\ \texttt{instanceof}\ T \\
 & | & [\texttt{l}_1\texttt{:}]\ \texttt{goto}\ \texttt{l}_2 & & | & (T)\ e \\
l & ::= & v & & | & l \\
 & | & e.f & & | & n \\
 & | & e_1[e_2] & & | & c_I \quad | \quad c_R \quad | \quad c_B \quad | \quad c_S
\end{array}
$$

Figure 3.3: Formal grammar for a subset of JKil. In particular, exceptions, Java generics, inner classes and modifiers are omitted for simplicity.

```
public Integer f(int y,
                 int[] z) {
  Integer i = null;
  int x = 0;
  while(x < 10) {
    if(x == y) {
      i = new Integer(
                 z[x]);
      break;
    } else {
      x = x + 1;
    }
  }
  return i;
}
```

```
public Integer f(int y
                 int[] z)
    Integer i
    int x
    i = null
    x = 0
L1: if x >= 10 goto L3
    if x != y goto L2
    i = new Integer(z[x])
    goto L3
L2: x = x + 1
    goto L1
L3: return i
```

Figure 3.4: Example of a how a Java `while` loop (left) is represented in JKil (right).

Other Language Features

In addition to structured language constructs, many other language features are translated when creating JKil representations. These are summarised in this section.

- **Name Resolution —** An important part of the translation process is name resolution. In ASTs produced by ANTLR, all types, local variable and field names are represented as identifiers. During the translation, the correct context of the identifiers is resolved. For example, if the identifier is the name of a field, then whether it is an instance or a static field, and whether it belongs to the current class, an enclosing class or a super class is determined, since each case is handled differently.

- **Local Variable Shadowing —** Consider the following Java example:

```
public void f() {
  if(...) {
    int i = ...;
  } else {
    int i = ...;
  }
}
```

  To handle local variables of the same name being declared in different scopes, JKit assigns each variable a unique name based upon the scope it was declared in. Hence, the above example would be translated into:

```
public void f()
    int i1
    int i2
L1: if ... goto L2
    i1 = ...
    goto L3
L2: i2 = ...
L3: return
```

- **Inner Classes —** While it is possible to declare a class within another class at the source code level, *inner classes* are considered separate classes in Java Bytecode. Therefore before any translation begins, JKit extracts inner classes from a class's AST and translates them as separate entities.

- **Exceptions —** JKit's support for Java Exceptions is detailed in Section 4.1.

Control Flow Graphs

To construct a JKil representation of an AST, the AST of each method body is translated into a control flow graph (CFG). CFGs are directed graph representations of whole methods which capture the possible paths of executions through them. Figure 3.5 presents the CFG constructed for the `while` loop shown in Figure 3.4. In the CFG, the edges leading from the nodes `x < 10` and `x == y` are labelled **T** and **F**. These labels are used to indicate which edges are the true or false branches of the conditional.

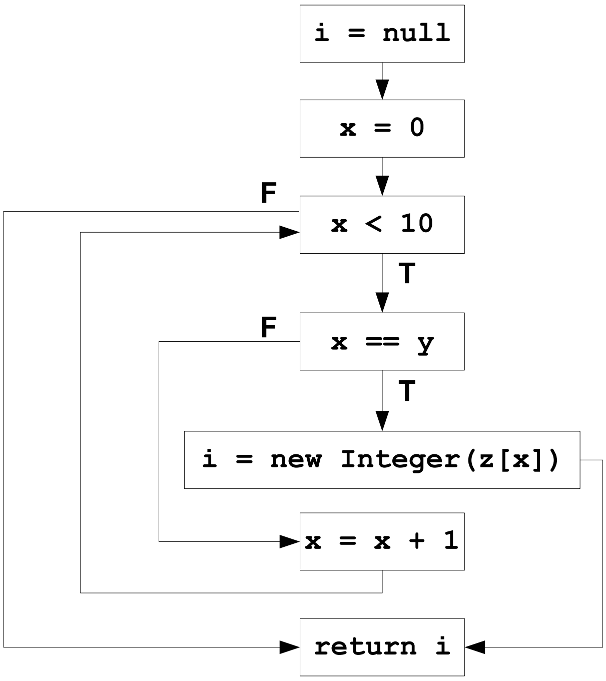


Figure 3.5: An example of a CFG constructed by JKit.

3.1.3 Java Types

In JKil's formal grammar is the concept of a Java type T. JKit supports the full set of Java types, however only a small portion are considered in this chapter.

A type T is defined in JKil as:

$$T \quad ::= \quad \top \mid T\ \texttt{[]} \mid n \mid null \mid int \mid boolean \mid \bot$$

In this definition:

- n represents the name of a class, such as `java.lang.Float`.
- $null$ represents the value `null`.
- The types int and $boolean$ represent the primitive Java types `int` and `boolean` respectively. They are included to illustrate the role of primitive types.
- The special type $\top$ is used where *any* possible values could be present, while $\bot$ is used where *no* possible value could be present.
- Generic types are not included for simplicity.

Supporting these types is a subtype relation. Conceptually, a subtype relation allows certain type properties to be checked. For example, in Java, if a variable `x` has a type X, then it can only be assigned to a variable of type Y if X is a subtype of Y, denoted $X \leq Y$. The subtype relation is a partial order and is defined in Figure 3.6.

The rule that if $T_1 \leq T_2$ then $T_1[] \leq T_2[]$ has often been highlighted as a deficiency in Java as it can lead to the following example:

```
f(Object[] xs) { xs[0] = "Error"; }
...
Integer[] is = new Integer[1];
f(is);
```

$$\frac{n_1 \textbf{ extends } n_2}{n_1 \leq n_2} \qquad \frac{n_1 \textbf{ implements } n_2 \ldots n_n}{n_1 \leq n_2 \ldots n_1 \leq n_n}$$

$$\frac{T_1 \leq T_2}{T_1 \texttt{ [] } \leq T_2 \texttt{ []}}$$

$$\frac{}{\mathit{null} \leq n} \qquad \frac{}{\mathit{null} \leq T \texttt{ []}} \qquad \frac{}{T \texttt{ [] } \leq \textbf{java.lang.Object}}$$

$$\frac{}{T \leq T} \qquad \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3} \qquad \frac{}{T \leq \top} \qquad \frac{}{\bot \leq T}$$

Figure 3.6: Formal definition of JKit's subtyping rules. A class n_1 **extends** or **implements** a class n_2 if it is declared as such in source code. The class **java.lang.Object** is an instance of n and is extended by all classes unless a specific superclass is declared in source code.

Because of the subtyping rule, this example would pass compile time type-checking. However at runtime a type-error would occur. More discussion about the effect of this rule can be found in Chapter 2.5 of Java Generics and Collections [29].

3.1.4 Type Checking

Having created the CFG representations, these are then type-checked by JKit. Type checking Java programs involves checking that numerous properties hold true for the program, including:

- The right hand side (RHS) of an assignment is a subtype of the left hand side (LHS).
- Methods invoked do exist and that the arguments passed are subtypes of those expected.

- Returned values are subtypes of that stated in the method's signature.

However after the CFG construction process, the only type information JKit has is that given in the source code, which is not enough to check these properties. For example, the value returned by a method may be the result of an expression which involves multiple variables. Hence JKit must determine the type of every expression in the CFG before it can be type-checked.

Before presenting the rules used by JKit to determine the types of expressions, the environment Γ must be introduced: Let Γ be an environment which maps variables to their type and the notation $\Gamma \vdash e : T$ mean that under the environment Γ the expression e can be shown to have type T. At the beginning of the type-checking of each method, Γ is initialized to hold the names of all local variables declared in the method and their types. Java's special `this` pointer is also added to Γ with the type of the class to which the method belongs. Only one instance of Γ exists for each method, and the contents of the environment do not change during the type-checking process.

Figure 3.7 formally presents the rules used by JKit to determine the types of expressions. It can be assumed that in JKil all names n are fully qualified, therefore n is a type itself. A number of helper functions were also created to aid the formalisation:

- **meth**$(T, m, T_1, \ldots, T_n)$ returns the result type of the invocation of method m on the receiver type T with argument types $T_1, \ldots, T_n$. To determine the specific method invoked, the function scales T's type hierarchy, looking for methods with the name m that would accept the given argument types. From the list of potential methods, the most applicable one is chosen using criteria detailed in [12], and its return type is returned. If no applicable method is found, then an error is raised.

- **field**(T, f) returns the type of field f belonging to type T or one of its supertypes. The function will scale T's type hierarchy until a field with the name f is found. If no field is found, then an error is raised.

Armed with the types of expressions, JKit is able to type-check statements using the rules presented in Figure 3.8. In the rules, the type OK is used to denote that a statement is well-formed and type safe. Furthermore, the additional helper function **ret**(M) is defined as returning the return type of method M (i.e. the method being type checked).

The following derivation tree illustrates how JKit type checks a simple Java statement. Let the static type of x be int, and the static type of y be $int[]$. Thus $\Gamma = \{x : int,\ y : int[], \ldots\}$.

$$\dfrac{\Gamma \vdash x : int \qquad \dfrac{\dfrac{\Gamma \vdash y : int[] \qquad \Gamma \vdash 3 : int}{\Gamma \vdash y[3] : int}\,\textit{(ARRAYRW)} \qquad \Gamma \vdash 1 : int}{\Gamma \vdash y[3] + 1 : int}\,\textit{(IBINOP)}}{\Gamma \vdash x = y[3] + 1 : OK}\,\textit{(ASGN)}$$

Here, the types of the LHS and RHS of the assignment are inferred using the expression type inference rules. These types are then type-checked using the ASSIGN rule. Since the type of the RHS, int is a subtype of the LHS, int, the statement passes the type-check.

3.2 Mocha Implementation

The type-checking process that has just been outlined was designed to be part of JKit's Pipeline for compiling normal Java source code, and is based on the premise that the static types of each local variable is known. However *this premise does not hold if a program is written in Mocha*. The static types of local variables may have been replaced with `var`, meaning Mocha must infer the actual types before the compilation process can continue.

$$\frac{\{x : T\} \in \Gamma}{\Gamma \vdash x : T}\ [\text{LOCALVAR}] \qquad \frac{c_I \in \{\ldots, -1, 0, 1, 2, 3, \ldots\}}{\Gamma \vdash c_I : int}\ [\text{INT}]$$

$$\frac{c_B \in \{false, true\}}{\Gamma \vdash c_B : boolean}\ [\text{BOOLEAN}] \qquad \frac{\begin{array}{c}\Gamma \vdash e : T_1 \\ T_1 \leq T_2 \vee T_2 \leq T_1\end{array}}{\Gamma \vdash (T_2)e : T_2}\ [\text{CAST}]$$

$$\frac{\Gamma \vdash e_0 : T_0[] \quad e_1 : int}{\Gamma \vdash e_0[e_1] : T_0}\ [\text{ARRAYRW}] \qquad \frac{\Gamma \vdash e_1 : int}{\Gamma \vdash \texttt{new}\, T[e_1] : T[]}\ [\text{NEWARRAY}]$$

$$\frac{\Gamma \vdash e : boolean}{\Gamma \vdash uop\ e : boolean}\ [\text{LUOP}] \qquad \frac{\Gamma \vdash e : int}{\Gamma \vdash uop\ e : int}\ [\text{IUOP}]$$

$$\frac{\Gamma \vdash e_1 : boolean \quad e_2 : boolean}{\Gamma \vdash e_1\ binop\ e_2 : boolean}\ [\text{LBINOP}]$$

$$\frac{\Gamma \vdash e_1 : int \quad e_2 : int}{\Gamma \vdash e_1\ binop\ e_2 : int}\ [\text{IBINOP}] \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \\ \mathbf{field}(T_0, f) : T_{field}\end{array}}{\Gamma \vdash e_0.f : T_{field}}\ [\text{FIELDRW}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \\ \Gamma \vdash e_1 : T_1, \ldots, e_n : T_n \\ \mathbf{meth}(T_0, m, T_1, \ldots, T_n) : T_{ret}\end{array}}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : T_{ret}}\ [\text{INVOKE}]$$

$$\frac{n_0 : T}{\Gamma \vdash \texttt{new}\, n_0(e_1, \ldots, e_n) : T}\ [\text{NEWCLASS}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_0 \\ T_0 \leq T_1 \vee T_1 \leq T_0\end{array}}{\Gamma \vdash e\ \texttt{instanceof}\ T_1 : boolean}\ [\text{INSTANCEOF}]$$

Figure 3.7: Formal definition of JKit's type inference rules for expressions.

$$\frac{\begin{array}{c}\Gamma \vdash l : T_1 \quad e : T_2 \\ T_2 \leq T_1\end{array}}{\Gamma \vdash l = e : OK}\ \text{[ASGN]} \qquad \frac{\Gamma \vdash e : boolean}{\Gamma \vdash \texttt{if}\ e\ \texttt{goto L2:}\ OK}\ \text{[COND]}$$

$$\frac{}{\texttt{goto L2:}\ OK}\ \text{[GOTO]} \qquad \frac{\begin{array}{c}\Gamma \vdash e : T_1 \\ \textbf{ret}(M) : T_{ret} \\ T_1 \leq T_{ret}\end{array}}{\Gamma \vdash \texttt{return}\ e : OK}\ \text{[RET]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : T \\ T \leq \mathbf{java.lang.Object}\end{array}}{\Gamma \vdash \texttt{lock}\ v : OK}\ \text{[LCK]} \qquad \frac{\begin{array}{c}\Gamma \vdash v : T \\ T \leq \mathbf{java.lang.Object}\end{array}}{\Gamma \vdash \texttt{unlock}\ v : OK}\ \text{[ULK]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \\ \Gamma \vdash e_1 : T_1, \ldots, e_n : T_n \\ \mathbf{meth}(T_0, m, T_1, \ldots, T_n) : T_{ret}\end{array}}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : OK}\ \text{[INVOKESTMT]}$$

Figure 3.8: Rules used by JKit to type-check statements. The type OK denotes that the statement is well-formed and type safe.

In order to add Mocha's inference process to JKit without affecting its own ability to compile normal Java source code, Mocha adds its own Pipeline to JKit. The Pipeline is similar to that used by JKit's for compiling Java source code, however there are some important differences:

- JKit's `JavaFileReader` Stage which handles the translation of ASTs into JKil representations, is extended and replaced by Mocha's `MochaFileReader` which supports the keyword `var`.
- Mocha replaces the various stages responsible for type checking statements, expressions and dealing with exceptions with its own imple-

mentation.

This section will focus almost entirely on Mocha's inference algorithm (which is a contribution of this thesis).

3.2.1 Overview

In Java, the type of a local variable is fixed for the entire method. Hence, during type checking, only one instance of Γ is required which contains the types of all local variables. However, in Mocha, the types of local variables are inferred from the values they are assigned, and variables can be retyped through certain conditionals. Therefore the types stored in Γ will change as a program is analysed. Consequently, it is not satisfactory to have just one instance of Γ for each method, instead Mocha associates an instance of Γ at each point in the method (where a point in a method, also referred to as a *program point*, is a single statement represented by a node in the method's CFG). Each instance reflects the *types of variables only at that point*.

The difference between type-checking in Java and in Mocha does not end at how many instances of Γ are used. Consider the following Mocha example:

```
public void f(int y) {
  var x = new Integer(1);
  while(y < 10) {
    x = new Float(y);
    y = y + 1;
  }
  var z = x;
}
```

In the above example, determining the correct type of `x` at the start of the `while` loop requires two passes by Mocha's inference algorithm.

In the first pass, the type of `x` entering the loop is considered, whilst in the second pass the type of `x` coming around the loop is accounted for. Figure 3.9 illustrates this process. In fact, the approach used by Mocha is very similar to the algorithms for *dataflow analysis*, as used in optimising compilers [1, 28].

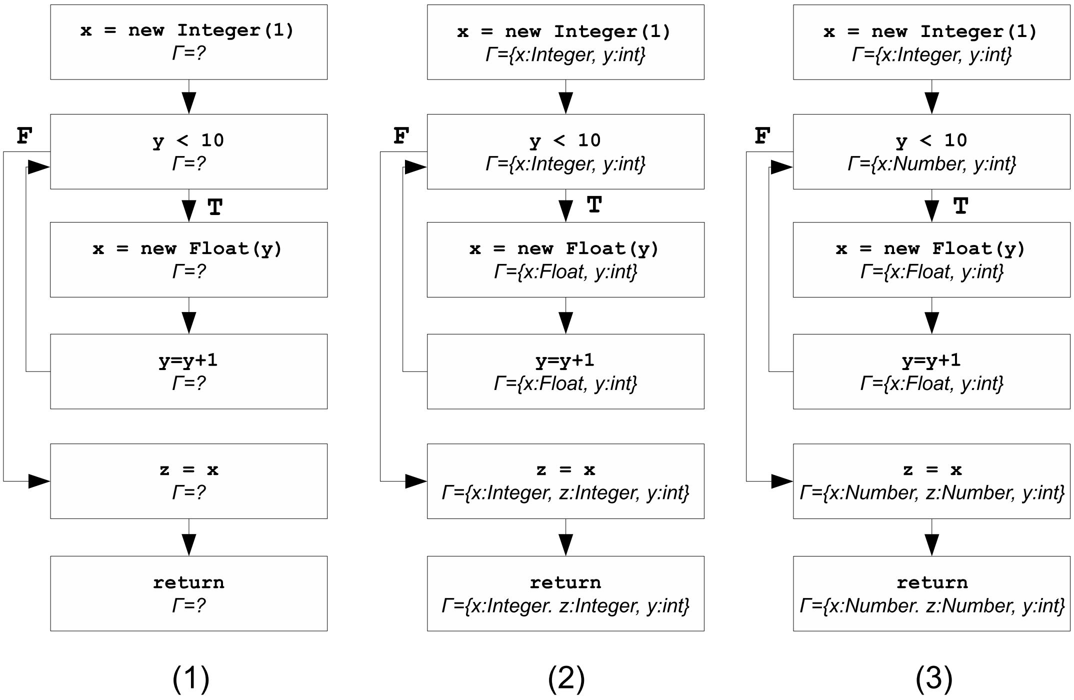


Figure 3.9: Illustration of iterative dataflow analysis. Image (1) presents the CFG and its associated Γ environments, before the algorithm has begun, (2) after the first pass of the algorithm and (3) after the second pass. In (3), the type of `x` at `y < 10`, `z = x` and `return` has been updated to reflect the fact it could be both `Integer` and `Float`.

At the heart of Mocha's dataflow analysis is its *abstract store*, which provides an abstraction of the program store at a particular point in a program. Conceptually, the abstract store is *identical* to Γ, since it simply maps local variables to their types. Therefore Γ will be used to denote the abstract store in the remainder of this chapter. However in addition, the abstract store provides a *join operator* (detailed in Section 3.2.4), which joins two instances of the store together at points where multiple paths of execution meet. At each point in the CFG, the *transfer function* (formalised in Section 3.2.2) changes the types in the incoming abstract store to incorporate the effect the statement at the point would have on the types of the local variables. The resulting store is then propagated onto the point's successors.

3.2.2 Transfer Functions

Mocha's transfer function evaluates the effect a statement at a node in the CFG has on the types in the abstract store. Figure 3.10 formally presents how each statement S is evaluated in the function. Each rule shows how the abstract store, which has come from the node's predecessor (shown on the left hand side of the $\longrightarrow$), is changed by the statement, resulting in the abstract store shown on the right hand side of the $\longrightarrow$. This store is then propagated along the edges from the node to its successors. The notation $\Gamma[v \mapsto T]$ states that the type of the variable v has been replaced by T in Γ. In Figure 3.10, it is assumed that the types of any expressions are inferred using the rules presented in Figure 3.7.

Most statements do not result in a divergence of program flow, hence only one rule is required for each of these statements. However because conditional statements do split the program flow, they need to be evaluated twice, once assuming the conditional evaluated true, and once assuming it evaluated false. The result of these evaluations would be propagated along the true and false branches respectively, coming from the node containing the conditional statement. Only `instanceof` conditionals actually effect Γ in different ways in the true and false evaluations, therefore two rules only are provided for this statement (labelled INSTANCE-OF *(True-Branch)* and *(False-Branch)*). All other conditional statements are evaluated twice using the rule COND.

$$\frac{\Gamma \vdash e : T_0}{v = e : \Gamma \longrightarrow \Gamma[v \mapsto T_0]} \qquad \text{[LVA]}$$

$$\frac{}{\texttt{goto L2} : \Gamma \longrightarrow \Gamma} \qquad \text{[GOTO]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_0 \\ \Gamma \vdash \textbf{field}(T_0, f) : T_{field} \\ T_0 \leq T_{field}\end{array}}{e.f = e : \Gamma \longrightarrow \Gamma} \qquad \text{[FA]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0[] \quad e_1 : int \\ \Gamma \vdash e_2 : T_1 \\ T_1 \leq T_0\end{array}}{e_0[e_1] = e_2 : \Gamma \longrightarrow \Gamma} \qquad \text{[ARW]}$$

$$\frac{\Gamma \vdash e : boolean}{\texttt{if}\ e\ \texttt{goto L2}\text{:}\ \Gamma \longrightarrow \Gamma} \qquad \text{[COND]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T \\ \Gamma \vdash \textbf{ret}(M) : T_{ret} \\ T \leq T_{ret}\end{array}}{\texttt{return}\ e : \Gamma \longrightarrow \Gamma} \qquad \text{[RTRN]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : T \\ T \leq \textbf{java.lang.Object}\end{array}}{\texttt{lock}\ v : \Gamma \longrightarrow \Gamma} \qquad \text{[LCK]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : T \\ T \leq \textbf{java.lang.Object}\end{array}}{\texttt{unlock}\ v : \Gamma \longrightarrow \Gamma} \qquad \text{[ULCK]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \\ \Gamma \vdash e_1 : T_1, \ldots, e_n : T_n \\ \textbf{meth}(T_0, m, T_1, \ldots, T_n) : T_{ret}\end{array}}{e_0.m(e_1, \ldots, e_n) : \Gamma \longrightarrow \Gamma} \qquad \text{[METHOD]}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 : T_1 \\ T_1 \geq T_2\end{array}}{\texttt{if}\ v_1\ \texttt{instanceof}\ T_2\ \texttt{goto L2}\text{:}\ \Gamma \longrightarrow \Gamma[v_1 \mapsto T_2]} \qquad \begin{array}{c}\text{[INSTANCEOF]} \\ \textit{(True-branch)}\end{array}$$

$$\frac{\Gamma \vdash v_1 : T_1}{\texttt{if}\ v_1\ \texttt{instanceof}\ T_2\ \texttt{goto L2}\text{:}\ \Gamma \longrightarrow \Gamma} \qquad \begin{array}{c}\text{[INSTANCEOF]} \\ \textit{(False-branch)}\end{array}$$

Figure 3.10: Formal definition of how each statement is evaluated in Mocha's transfer function.

The most important rule in the figure is LVA, which is the assignment of a value to a local variable. In normal Java, since static types cannot change, assigning a value to a local variable has no effect on its static type. However Mocha allows a local variable to be retyped at any point in a program through assignment statements. This rule is illustrated in diagram (2) from Figure 3.9 where the application of the LVA rule on the statement `x = new Float(y)` results in:

$$\Gamma = \{x : Integer,\ y : int\} \longrightarrow \Gamma' = \{x : Float,\ y : int\}$$

In comparison to LVA, the rules FA and ARW which represent field and array assignment respectively, do not result in the types in Γ being changed. This is because Γ simply holds local variables and their types, not fields nor the contents of arrays.

The other rule which has an impact on the types in the abstract store is INSTANCEOF *(True-Branch)*. As discussed in Section 2.1.2, Mocha analyses `instanceof` statements and retypes local variables to the type they are an instance of. A local variable is only an instance of a type if the condition evaluates true, hence the retyping only occurs along the true-branch. As an illustration of the application of this rule, consider the following simple example:

```
public void f(Object o) {
  if(o instance Integer) { ... }
  ...
}
```

In this example, the application of INSTANCEOF *(True-Branch)* to the statement "`if(o instanceof Integer)`" results in:

$$\Gamma = \{o : Object\} \longrightarrow \Gamma' = \{o : Integer\}$$

While the application of INSTANCEOF *(False-Branch)* to the same statement results in:

$$\Gamma = \{o : Object\} \longrightarrow \Gamma' = \{o : Object\}$$

An interesting restriction placed on the INSTANCEOF *(True-Branch)* rule is that the local variable must be a supertype of the type its being compared too. Although the type-checking rule shown in Figure 3.7 allows the variable to be a subtype, retyping the variable to a supertype would achieve nothing. The purpose of the `instanceof` retyping process is to refine the type information known about a variable, rather than making it more general.

3.2.3 Least Upper Bounds

Dataflow equations combine the components of Mocha's inference algorithm into a complete system. Before presenting these, however, the notion of the *most precise type* which is a supertype of two types must be defined. Ideally, Java's subtype relation would form a semi-lattice, meaning the most precise supertype would be the least upper bound of the two types. The concept of a least upper bound is defined as follows:

Definition 1. (Least Upper Bound) *Type T_3 is the least upper bound of the pair of types T_1 and T_2 iff $T_1 \leq T_3$, $T_2 \leq T_3$ and $\forall T\{T_1 \leq T \wedge T_2 \leq T \Rightarrow T_3 \leq T\}$*

However as discussed by Leroy in [19], since Java classes are able to implement numerous interfaces, there is not always a unique least upper bound. Therefore Java and JKit's subtype relations do not form semi-lattices. The following example illustrates how this complicates Mocha's type inference algorithm:

```
public class Foo {
  private static interface A { ... }
  private static interface B { ... }
  private static class C implements A, B { ... }
  private static class D implements A, B { ... }
```

```
    public void bar() {
      var x;
      if(...) {
        x = new C();
      } else {
        x = new D();
      }
      // What is the type of x? A or B?
    }
  }
```

At the completion of the `if` statement, the most precise type which represents that `x` could be assigned the type `C` or `D`, must be found. However, neither `A` nor `B` satisfies the definition of a least upper bound. To resolve this problem, Mocha includes its own join operator for finding the most precise type. The operator, $\sqcup_T$, like least upper bound, finds a unique type which is a supertype of both joined types and is defined below:

$$T_1 \sqcup_T T_2 = \begin{cases} T_1 & \text{if } T_2 \leq T_1 \\ T_2 & \text{if } T_1 \leq T_2 \wedge T_2 \nleq T_1 \\ T_3 & \text{if } T_3 = T_1 \sqcup_c T_2 \text{ exists and } T_1 \nleq T_2 \wedge T_2 \nleq T_1 \\ T_4 & \text{otherwise, where } T_4 \in T_1 \sqcup_I T_2 \end{cases}$$

In this definition, $T_1 \sqcup_c T_2$ is the *unique* least upper bound of T_1 and T_2 found through the class subtype relation (JKit's subtype relation $\leq$ from Figure 3.6 without the rule that if n **implements** n_1 then $n \leq n_1$) which forms a semi-lattice. In contrast, $T_1 \sqcup_I T_2$ is the set of minimal upper bounds of T_1 and T_2 found through the interface subtype relation (JKit's subtype relation from Figure 3.6 without the rule that if n **extends** n_1 then $n \leq n_1$). It should be noted that the choice of T_4 from the set $T_1 \sqcup_I T_2$ is deterministic.

The following examples illustrate how Mocha finds the least upper bound of two types in different situations. It should be noted that `A`, `B`, `C` and `D` are the classes and interfaces defined in the previous example, and `Float`, `Integer` and `Number` are Java classes:

$$\begin{array}{ll}
\texttt{Number} \sqcup_T \texttt{Float} = \texttt{Number} & (\text{Since } \texttt{Float} \leq \texttt{Number}) \\
\texttt{Integer} \sqcup_T \texttt{Float} = \texttt{Number} & (\text{Since } \texttt{Number} = \texttt{Integer} \sqcup_c \texttt{Float} \\
 & \text{and } \texttt{Float} \nleq \texttt{Integer} \wedge \texttt{Integer} \nleq \\
 & \texttt{Float}) \\
\texttt{C} \sqcup_T \texttt{D} = \texttt{A} & (\text{Since } \texttt{C} \sqcup_I \texttt{D} = \{\texttt{A}, \texttt{B}\})
\end{array}$$

The choice of `A` for the last example is deterministic, meaning that given the types `C` and `D`, `A` will *always* be chosen.

It is possible that when choosing from a set of types, Mocha could choose the wrong type for a variable, or the chosen type could fail to pass type-checking (although this has never been encountered). Therefore future work could consider using *intersection types* [11, 41, 42, 43], to provide a more robust solution. An intersection type is a set of types represented as a single type. When included, intersection types transform Java's subtype relation into a semi-join lattice. Furthermore, if such a situation were to arise, an explicit cast could be inserted into the code to indicate the variable's correct type.

3.2.4 Dataflow Equations

From the definitions of how the statement at an individual node is evaluated, and how Mocha joins two types, it is possible to define the algorithm used by Mocha to evaluate an entire method.

The transfer function for a node in a CFG takes the result of the transfer of the point's predecessors as an input. At join points in the CFG, multiple abstract stores come together and must be reduced to a single abstract store which conservatively approximates the possible program states going into the following statement. As discussed in Section 2.1.1, if a variable has been assigned different types in different flows, then the most precise type which caters for both assigned types is found. Hence Mocha's abstract join operator $\sqcup_\Gamma$ is defined using its type join operator $\sqcup_T$ (defined in Section 3.2.3). The complete definition of the operator $\sqcup_\Gamma$ is provided below:

Definition 2. *Let Γ_1 and Γ_2 be type environments. $\Gamma_1 \sqcup_\Gamma \Gamma_2 = \Gamma$ where $\Gamma(x) = \Gamma_1(x) \sqcup_T \Gamma_2(x)$ for all $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$.*

From this definition, the dataflow equations for a method can be defined:

Definition 3. *Let $G = (V, E)$ be the control flow graph for method M, where V is the set of nodes, and E the set of edges. Then the dataflow equations for each node y in M are given by $M(y) = M(y) \sqcup_\Gamma \bigsqcup_{\Gamma_{x \xrightarrow{l} y \in E}} t(S_y, M(x), l)$.*

In this definition, *t* is the transfer function defined in Figure 3.10, S_y is the statement at the node y, and the edge label *l* separates the true and false edges coming from conditional statements as discussed in Section 3.1.2.

Mocha solves the equations using a *worklist algorithm*. Starting with the method's entry point, a point is evaluated and if the result of the evaluation differs from the point's previous result (or no previous result exists), then the point's successors are added to the worklist. The point at the head of the worklist is then removed, and the process begins again. The algorithm iterates until the result of each node ceases to change (referred to as a *fixed point*), resulting in no further nodes being added to the list. Optimisations and alternatives to this algorithm can be found in [9, 38, 8] and [16, 6, 9] respectively.

3.2.5 Termination

The usefulness of Mocha's inference algorithm would be limited if it did not terminate in all cases, therefore it is necessary to prove termination. As already stated, the algorithm will terminate once the result of the evaluation of each point in the CFG has reached a fixed point [31]. However the following example would seem to be a valid Mocha program where the type of a variable would not reach a fixed point:

```
public Integer f(String s) { return ""; }
public String f(Integer i) { return new Integer(0); }
public void g(int x) {
  var y = "String";
  while(x < 10) {
    y = f(y);
    x = x + 1;
  }
}
```

During the first iteration of the `while` loop, the type of `y` is `String`, resulting in it being assigned the type `Integer`. Because the type of `y` has changed, the loop will be passed over again. This time, `y` will be assigned `String`. Again its type will have changed, so the loop will be passed over yet again. During each pass, the type of `y` would change and a fix point would never be reached.

However Mocha's inference algorithm does terminate when inferring the example, albeit with an error message. This is due to the fact that the algorithm stores the result of a node's evaluation at each of its successors, and joins any subsequent results. Therefore after the first pass over the loop, `y`'s type, `Integer`, will be joined with its previous type `String`, resulting in it having the type `Object`. Because the method `f` has not been overloaded to accept a parameter of type `Object`, Mocha would be unable to resolve the method invocation during the second pass through the loop, and an error would be raised.

Consequently, in order to prove that Mocha's inference algorithm will terminate in all cases, it must be proven that Mocha's join operator $\sqcup_\Gamma$, will reach a fixed point.

Theorem 1. *Mocha's join operator $\sqcup_\Gamma$ will reach a fixed point, meaning Mocha's* *type inference algorithm will always terminate.*

Proof. The notion of a fixed point for the join operator is defined as, there exists an n such that $\Gamma_n \sqcup_\Gamma \Gamma_{n+1} = \Gamma_n$, where n is the number of the passes done by the algorithm, and Γ_n is an instance of Γ at some node. From the definition of $\sqcup_\Gamma$, it is clear that if $\Gamma'_n = \Gamma_n \sqcup_\Gamma \Gamma_{n+1}$, then $\Gamma_n(x) \leq \Gamma'_n(x)$ for all $x \in dom(\Gamma_n)$. If after a certain number of passes by the algorithm $\Gamma_n(x) = \Gamma'_n(x)$ for all $x \in dom(\Gamma_n)$, then a fixed point has been reached. However if there exists an x such that $\Gamma_n(x) \neq \Gamma'_n(x)$, then the algorithm would have to do another pass.

After each pass of the algorithm, the join of the two stores will be found. Because this process uses the type join operator $\sqcup_T$, the types in the resulting store will be *higher* on the type lattice. They will continue to get higher until they reach $\top$, at which point they will cease to change. An abstract store can also be thought of as having reached $\top$ when all the types of its variables have reached $\top$. Once Γ_n reaches $\top$, $\Gamma_n \sqcup_\Gamma \Gamma_{n+1}$ will also be $\top$, hence $\Gamma_n = \Gamma_n \sqcup_\Gamma \Gamma_{n+1}$, and a fixed point has been reached. □

Chapter 4

Extensions

Mocha is not just limited to simple local variable type inference. In this chapter two extensions to Mocha are presented. First, the impact Java exceptions have on local variable type inference is discussed, and Mocha's novel approach for supporting such exceptions is presented. Then, Mocha's unique support for field type inference, as described in Section 2.2.1, is discussed in greater detail.

4.1 Exceptions

Java exceptions complicate local variable type inference. In order to infer the correct types of local variables at every point in a program, edges representing the program flows created by exceptions must have been inserted. However, in order to identify where exceptions are thrown, the types of variables need to be known. In this section, Java exceptions will be explained in greater depth, followed by an explanation of JKit's support for exceptions and a discussion concerning why Mocha cannot use the same approach. Finally, a simple approach for supporting exceptions will be introduced followed by an in-depth discussion of the approach used by Mocha.

4.1.1 Exceptions Explained

Exceptions are designed to provide greater error information and safety in Java programs. At any stage in a program's execution, an exception may be *thrown* either explicitly by the programmer or by the JVM. To facilitate the catching of exceptions, Java provides the `try-catch-finally` statement. Any code that may be the source of an exception such as a method invocation, can be placed within the `try` block. When an exception is thrown, the current path of execution is immediately interrupted. If a catch block for the exception is provided then the code within the block is executed. If no catch block is provided, the method exits. The following example illustrates how a `try-catch` block is used:

```
public String foo(String fname) {
  try {
    File f = new File(fname);
    f.createNewFile();
  } catch(IOException e) {
    return "IOException␣thrown";
  }
  return "File␣Created";
}
```

The method `createNewFile()` can throw an `IOException` in certain circumstances. If an exception were thrown, rather than executing the statement `return "File Created"`, the program would immediately flow into the catch block and the statement `return "IOException thrown"`.

Exceptions can be divided into two categories, *unchecked* and *checked* exceptions. Unchecked exceptions should represent errors that occurred *within* the program itself [5], such as a null reference being dereferenced (which results in a `NullPointerException`). Consequently Java does not require that code that can be the source of unchecked exceptions be placed within a `try` block, and `catch` blocks do not need to be provided for unchecked exceptions that can be thrown during a `try` block.

Checked exceptions generally represent errors that are *outside* of the program's control, such as a non-existent file. If a statement could result in a checked exception being thrown, then the exception must either be re-thrown by the method, or the statement must be placed within a `try` block with a `catch` block for the exception. The previous example of the method `foo` illustrates how the source of a checked exception, `f.createNewFile()`, is placed within a `try` block and how the exception `IOException` is caught. If a method chooses to `throw` a checked exception itself, rather than provide a `catch` block, then it must be stated in its declaration. Hence, if `foo` re-threw the `IOException`, its declaration would become:

```
public void foo(String fname) throws IOException { ... }
```

4.1.2 JKit on Exceptions

JKit does not include `try-catch` blocks in its JKil representation. Instead code from within both kinds of blocks are treated as normal blocks of code. After the translation into JKil has completed, JKit inserts into the CFGs *exceptional edges* from statements that are the source of exceptions, to the code that is executed when the exceptions are thrown. Figure 4.1 presents the CFG constructed by JKit for the method `foo` from the previous example.

Before the exceptional edges can be inserted, JKit must generate the list of exceptions a statement could throw. For most kinds of statements it is trivial to generate the list. For example, field reads and writes can obviously throw a `NullPointerException`. However with method invocations, it is necessary to first resolve the exact method being invoked, so the list of exceptions the method throws can be retrieved.

An important characteristic of JKit's support for exceptions is that it is able to insert all exception edges into a CFG in a single pass. *This is pos-*

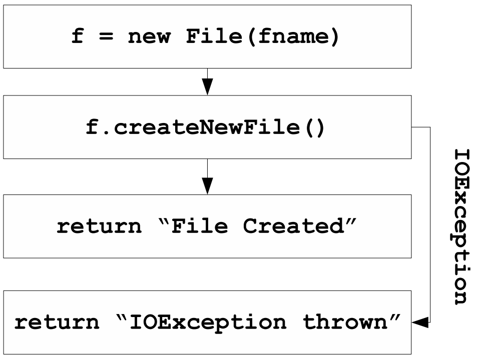


Figure 4.1: CFG which includes an exceptional edge inserted by JKit.

sible since in Java the full type information of every expression and local variable is known. Furthermore, each local variable and field has only one type. Therefore JKit is able to identify which exceptions a statement can throw immediately.

4.1.3 Mocha on Exceptions

In contrast to Java, if a program were written in Mocha then the types of local variables and fields would need to be inferred before the exceptional edges could be inserted. The complicating factor is that in order to infer the types of local variables at every point, the exceptional edges must have been inserted already. This is illustrated in the following example:

```
void f(Integer n) throws IOException { ... }
void f(Float f) { ... }
void g(int y) {
  var x = new Float(0);
  try {
    f(x);
    while(y < 10) {
      x = new Integer(y);
      f(x);
      y = y + 1;
    }
  } catch(IOException e) {
    // What is the type of x here?
    var z = x;
  }
}
```

In order to know the type of `x` in the `catch` block, the exceptional edges must have been inserted. However the type of `x` must first be inferred in order to know which version of `f` is being invoked at the two invocations, since the decision will impact where exceptional edges are inserted. Given this complication, Mocha replaces the various Stages in JKit for dealing with exceptions, with its own exception support procedure.

Simple Approach

A simple approach for supporting exceptions during type inference is to conservatively add exceptional edges first, and then perform type inference. This works well for most statements since, for example, simple assignments do not throw any exceptions and field assignments can only throw `NullPointerExceptions`. However since the exceptions thrown by a method invocation will not be known until the invocation has been resolved, it must be assumed that an invocation can throw any and all possible exceptions. The CFG produced using this approach for the previous example is shown in Figure 4.2.

This approach can be used since it does result in valid types being inferred for local variables in some cases. Given the CFG in Figure 4.2, Mocha would infer the type of `x` at the statement `z = x` to be `Number`. If both invocations of `f` resulted in exceptions being thrown, this would be the correct result. However, in reality only the invocation in the `while` loop could result in an exception being thrown. Therefore inserting edges

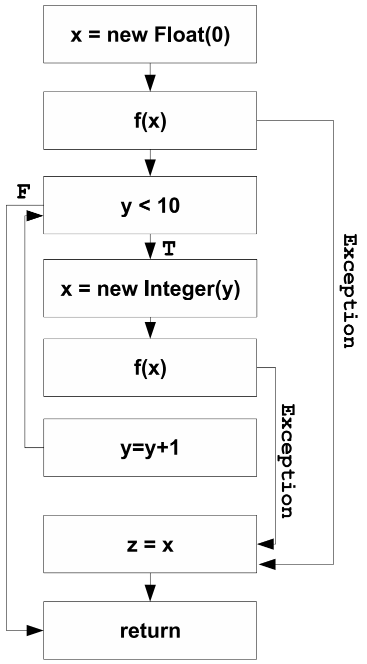


Figure 4.2: Example of a CFG produced using the simple approach to supporting exceptions in type inference. The exceptional edges are labelled `Exception` since the exceptions that are thrown by invocations are unknown and therefore the most general exception must be assumed.

from both invocations of `f` has resulted in Mocha inferring an imprecise type for `x` and `z`. The correct type of `z` should be `Integer`.

Mocha's Approach

The fact that the simple approach can result in imprecise types being inferred means that it is not used by Mocha. Instead Mocha uses an approach for supporting exceptions which is quite different from that used in JKit, and is tightly integrated with the type inference algorithm.

As mentioned, Mocha is unable to identify which exceptions a statement throws until all types of expressions and variables are known, which cannot occur until the exceptional edges have been inserted. To break this

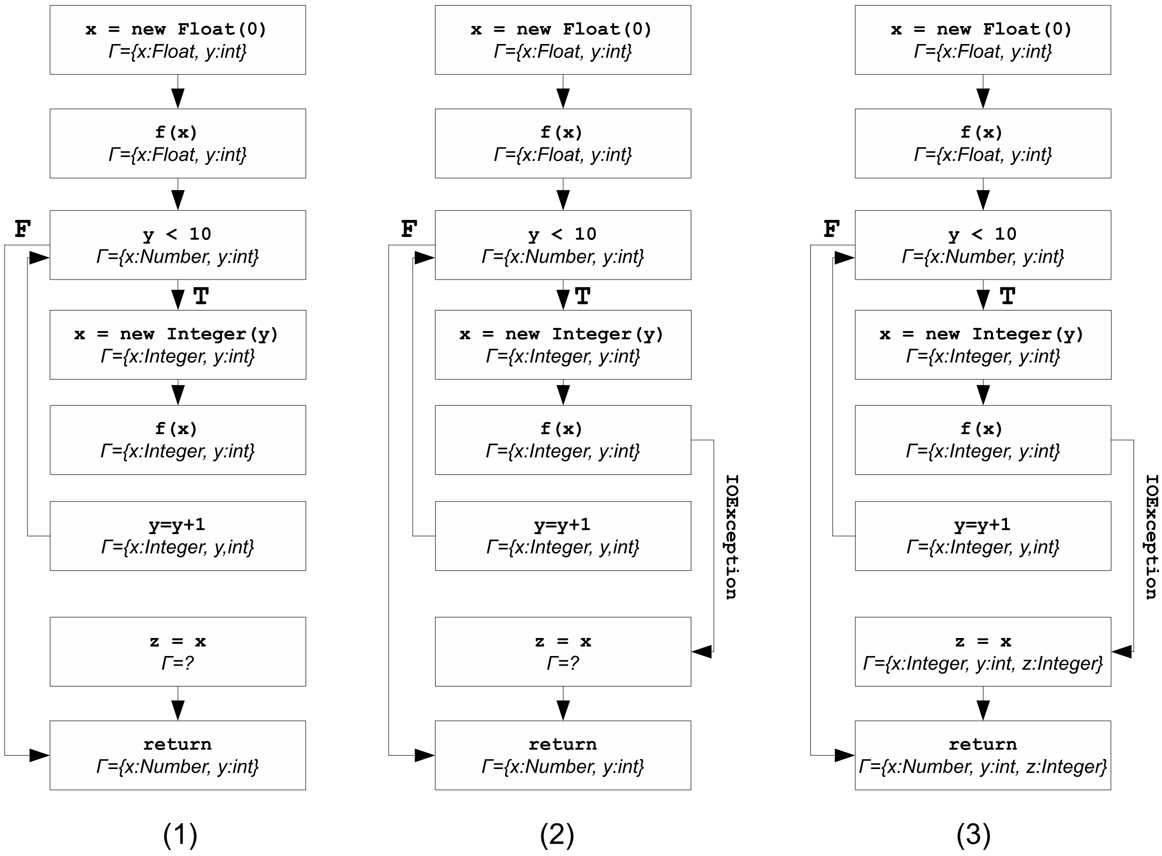


Figure 4.3: Example of a CFG produced using the simple approach to supporting exceptions in type inference.

cycle, Mocha infers the types of local variables at as many nodes as it can in the CFG, then inserts any exceptional edges it can identify, before returning to infer the types of local variables again. This process of inferring types, inserting exceptional edges, and inferring types again continues until no new exceptional edges are inserted.

Figure 4.3 illustrates this process by showing the CFG for the previous example, after the three stages of the process. The CFG after the first stage (1) shows that Mocha has inferred the types of as many variables as it can reach using the process outlined in Chapter 3. However, because the node `z = x` is not yet connected to the CFG, it has not been visited by Mocha and has not had its types inferred. The second CFG (2) shows the CFG after Mocha has inserted the exceptional edge. Unlike with the simple approach, Mocha only inserts an edge from the invocation of `f` in the loop. This is possible since the type of `x` has been inferred and the exact method invoked can be resolved. The last CFG (3) shows the CFG after Mocha's inference algorithm has passed over it again. This time, because the node `z = x` is connected to the rest of the CFG, it is visited, and Mocha is able to correctly infer the type of `z` to be `Integer`.

4.2 Field Type Inference

In Section 2.2.1, a brief overview of Mocha's field type inference approach was given. The motivation given for Mocha inferring the types of fields is that field declarations also suffer from redundant type information which could be inferred by a compiler. However, as mentioned, inferring the types of fields is more difficult than inferring local variable types. Fields belong to classes as a whole, therefore statements in multiple methods can affect the field's type as illustrated below:

```
public class InfExample {
  private Object obj;
  public void f() {
    obj = new Integer(1);
    // obj is an Integer here
  }
  public void g(Character c) {
    obj = c;
    // obj is a Character here
  }
}
```

More importantly though, is the requirement that field types must be known and fixed before Mocha's local variable type inference algorithm can begin (this requirement is discussed in Section 2.2.2).

However a field's type can be unknown during the inference of a method if the field is not used in any statement (other than when it is being assigned a value). Restricting the use of fields in all methods would be unreasonable, therefore Mocha only supports field inference through values assigned in constructors. Of course, a field could be assigned a different value in each of its class's constructors. Since it is necessary to ensure that the inferred type of a field is a supertype of every value it is assigned, the final inferred type of the field is the join $\sqcup_T$ of all the types it is assigned. Hence, let f be a field belonging to class C, whose type needs to be inferred. If in C's constructors f is assigned values with the types $X_0, \ldots, X_{n-1}$, the inferred type of f can be formally defined as:

$$C.f : X_0 \sqcup_T \ldots \sqcup_T X_{n-1}$$

The following simple example illustrates how Mocha infers the type of a field from values assigned in multiple constructors:

```
public class bar {
  private var x;
  public bar() { x = new Integer(0); }
  public bar(float f) { x = new Float(f); }
}
```

The inferred type of `x` would be `Integer` $\sqcup_T$ `Float`, which is `Number`. When the two assignment statements are then type-checked, since `Integer` $\leq$ `Number` and `Float` $\leq$ `Number`, both assignments would be valid.

If a field whose type is to be inferred is used in an expression in a constructor then during the inference of the constructor Mocha would be unable to infer the expression's type and an error would be thrown.

Mocha's field type inference approach is not just limited to instance fields. Static fields can also be declared without a static type and Mocha will infer it from the values the field is assigned. Mocha uses the static constructor of the field's class as the source of values.

4.2.1 Subclasses

Subclasses affect Mocha's field type inference approach in two ways. Firstly, a class may refer to a protected field belonging to its superclass. If the superclass has not yet been compiled, the field's type will be unknown if it is to be inferred. Consequently, if such a scenario occurs, Mocha runs its inference algorithm over the superclass before returning to the subclass. If the superclass cannot be found, or a cyclic relationship between classes exists, an error is raised.

Secondly, a class may assign a value to a protected field belonging to its superclass during its constructor. If the field's type has been inferred by Mocha, then the type will only be based on the values assigned in the superclass's constructors. Therefore it may not be a supertype of the type assigned in the subclass. This is illustrated in the following example:

```
public class foo {
  protected var pfield;
  public foo(Integer i) { pfield = i; }
  public class bar extends foo {
    public bar(Float f) { pfield = f; }
  }
}
```

When compiling the class `foo`, Mocha would infer the type of `pfield` to be `Integer` (since the field is only assigned an `Integer` in `foo`'s constructor). When Mocha then compiles `bar`, since `Float` is not a subtype of `Integer`, the assignment `pfield = o` would fail its type-check. Such a scenario could only be resolved if all the subclasses of `foo` were known when it was compiled, which is an unreasonable expectation. Therefore the only practical solution would be for the programmer to provide the proper static type for `pfield` in its declaration (which is permitted in Mocha).

This is an example of a much wider problem in object-oriented programming referred to as the *fragile base problem*, a discussion of which can be found in Mikhajlov and Sekerinski [26].

4.2.2 Initialisation Methods

A common practice in Java is to separate the process of initialising an object into multiple methods which are called by the constructors. An example of this is provided below:

```
class X {
  T f;
  X() { g(); }
  void g() { f = ...; }
}
```

Currently, Mocha does not include support for a programmer specifying additional methods that should be included in the field inference procedure. Therefore code such as that in the above example will need to be refactored so that the invoked method returns a value that is assigned to the field in the constructor. The refactored example is shown below:

```
class X {
  T f;
  X() { f = g(); }
  T g() { return ...; }
}
```

4.2.3 Code Brittleness

Whilst inferring field types from their assignments in constructors does remove redundant type information from field declarations, it also makes the code that uses type-inferred fields more brittle. Consider the following example:

```
public class foo {
  public var i;
  public foo() { i = new ArrayList<Integer>(); }
}
```

Based on the value that `i` is assigned in the constructor, Mocha would infer the type of `i` as `ArrayList<Integer>` and any code written at this point would assume that was `i`'s type. However if at a later point it was deemed, for performance reasons, that a `LinkedList` would be better, any code that made the previous assumption about `i`'s type, would break. Therefore Mocha's inference approach has meant that changing the implementation of `i` changes its interface too, which goes against the object-oriented principal of *encapsulation*.

To mitigate the effect of the increased code brittleness, `public` fields should not be declared `var`. With `private` and `protected` fields, the effect of the brittleness will be limited to the class to which the field belongs, and any of its subclasses.

Chapter 5

Evaluation

In this Chapter, results of an experimental evaluation of the effectiveness of Mocha are presented.

To evaluate its effectiveness, a number of benchmark programs were compiled using Mocha. The benchmarks were chosen specifically as they are examples of large widely used open-source Java applications. The benchmarks are introduced below:

- **Robocode** — Robocode [30] is a Java game which simulates combat between user-created virtual robots in a graphical 2D environment. Robocode has also been used in the artificial intelligence field as both a teaching and a development tool [15, 13, 34]. The program includes a comprehensive front-end and uses I/O and thread synchronization extensively in the back-end.

- **Javadoc and Javap** — Both Javadoc and Javap are tools distributed along with Sun's Java compiler, javac and have been recently made open-source as part of OpenJDK [25]. Javadoc generates HTML documentation based of comments provided in Java source-code, while Javap presents the information contained in Java classfiles in a more human-readable format.

Benchmark	Lines of Code	Number of Classes	Number of Methods
Robocode	25271	266	2371
Javadoc	5264	24	563
Javap	3650	55	186

Figure 5.1: Benchmark statistics

The number of lines of code, classes and methods for each benchmark is summarised in Figure 5.1.

In the remainder of the chapter, the results of the evaluation of each aspect of Mocha will be presented.

5.1 Local Variables

The most important part of Mocha is its support for local variable type inference. To evaluate the effectiveness of this, each benchmark was compiled and the number of type-inferred local variables was recorded. Initially it was assumed that every local variable could have its type inferred. Therefore the static types of each variable was manually replaced by `var` and the program was compiled using Mocha. If an error occurred during the compilation due to Mocha being unable to infer the type of a local variable, or the inferred type failed to pass type-checking, then the variable's original type was inserted. Only after the compilation succeeded without any errors was the count of inferred variables recorded.

It should be noted that despite being local variables, method parameters were not included in the evaluation. This is because, in Mocha, all method parameter and return types must be given explicitly.

Figure 5.2 is a summary of the number of local variables from each benchmark whose type was successfully inferred by Mocha. The results are presented as a graph in Figure 5.3.

The results indicate that almost all local variables can have their static

Benchmark	Inferred Local Variables	Total Local Variables	Percentage Inferred
Robocode	1538	1801	85
Javadoc	283	322	88
Javap	500	537	93

Figure 5.2: Statistics recorded in the evaluation of Mocha's ability to infer the types of local variables.

types accurately inferred by Mocha. However a small percentage of variables could not have their types inferred, the predominant source of which were `catch` statements. Each `catch` statement includes a local variable which has the static type of the exception the statement is catching. These variables did not have their types inferred due to Mocha not supporting their inference at this point. However it would be possible to infer their types using the kind of type inference algorithm used by Mocha. These variables were included in the experimental results because in the JKil representation of a method they appear just as normal local variables and cannot be separated.

The other sources of uninferred local variables are summarised below:

- Local variables used in anonymous inner classes. Consider the following example:

```
public void f() {
  var o = new Object();
  var x = new X() {
    public String g() {
      return o.toString();
    }
  };
}
```

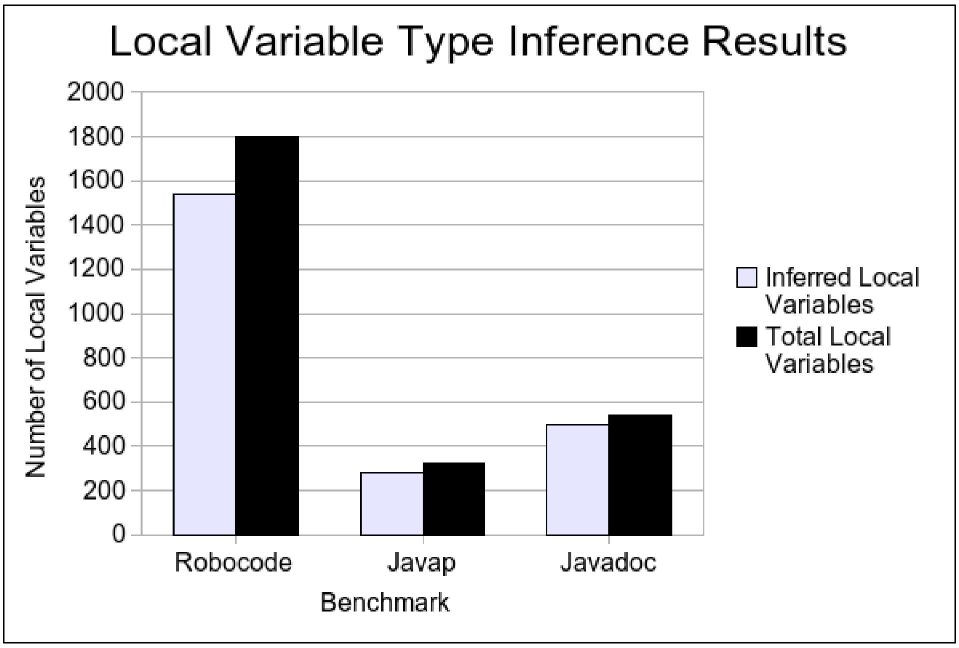


Figure 5.3: Number of type-inferred local variables compared against the total number of local variables in each benchmark.

Before Mocha's inference algorithm can begin, the JKil representation of each class that is to be compiled must have been created by JKit. Since the anonymous inner class in the example would be compiled separately to the class it is declared in, o would be stored as a field in the anonymous class. However since Mocha has not inferred the type of o, the type of the field would not be known and the JKil representation of the anonymous class would not be able to be created. Therefore to resolve this situation, the full static type of o must be given in the source code.

- Methods where the return type is the source of generic type information. It is possible to declare and use a generic method in Java as follows:

```
public static <T> T f() { ... }
...
String x = f();
```

The static type of x, String, is used to determine the type of the generic parameter T in the method f. If the same example was written in Mocha, with the type of x waiting to be inferred, it would not be possible to determine the return type of f. Therefore the type of x could not be inferred and its full static type must be given in the source code.

5.2 Fields

To evaluate the effectiveness of Mocha's support for field type inference, a procedure very similar to that used to evaluate the support for local variable type inference was used. It was assumed that every field in each benchmark could have its static type inferred. Hence the static types of each field was replaced with var. Each benchmark was then compiled and where errors occurred due to Mocha being unable to infer the type of the field, the field's original static type was re-inserted into the source code. Once a benchmark compiled without error, the total number of type-inferred fields was recorded. None of the code in the benchmarks was refactored in order to increase the number of fields that could have their types inferred.

Figure 5.4 is a summary of the number of fields from each benchmark whose type was inferred by Mocha. The results are presented as a graph in Figure 5.5.

Unlike local variables, many fields could not have their static types inferred by Mocha, particularly in the Robocode benchmark. The pattern of deferring the initialisation of fields until they are needed, played a major

Benchmark	Inferred Fields	Total Fields	Percentage Inferred
Robocode	591	1247	47
Javadoc	129	184	70
Javap	586	655	89

Figure 5.4: Statistics recorded in the evaluation of Mocha's ability to infer the types of fields.

role in reducing the number of fields whose types could be inferred. To understand this, consider the following example taken from the Robocode benchmark:

```
private JButton getPredefinedQualityButton() {
  if (predefinedQualityButton == null) {
    predefinedQualityButton = new JButton("Quality");
    predefinedQualityButton.setMnemonic('Q');
    predefinedQualityButton.
          setDisplayedMnemonicIndex(0);
    predefinedQualityButton.
          addActionListener(eventHandler);
  }
  return predefinedQualityButton;
}
```

The field `predefinedQualityButton` is not assigned any value in its classes constructor. Instead it is initialised when it is first accessed through a getter method. Since the field's initialisation occurs outside of the constructor, Mocha is unable to infer the field's static type.

Other reasons for fields not having their types inferred are:

- Assigning the field the value `null` in the constructor. If the only value a field is assigned is `null`, then Mocha would infer the field's type to be *null*, which is not a valid static type. The assignment of

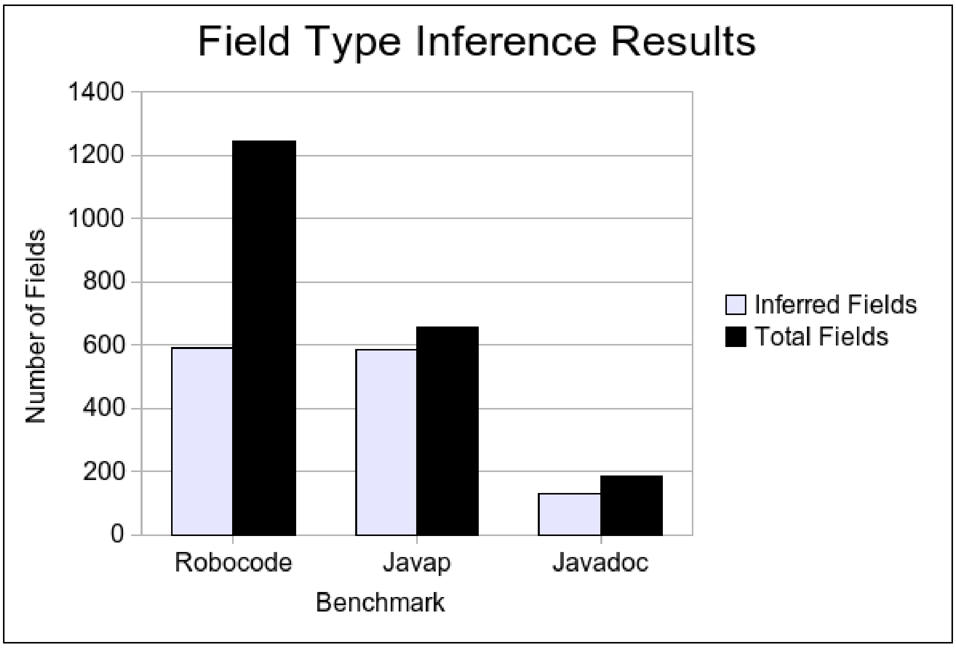


Figure 5.5: Number of type-inferred fields compared against the total number of fields in each benchmark.

`null` is often due to the field's actual value being calculated after object construction.

- The field being used in the constructor. In order for a field to be able to be used in a statement, its type must be known. Hence, if a field is used in a constructor, it cannot be waiting to have its type inferred.

- Mutually dependent objects. Some classes include a *copy constructor*, where the values of fields belonging to an instance of the class are assigned to the same fields of another instance. The following is an example of a copy constructor:

```
public class X {
  private Object o;
  public X(X x) {
    o = x.o;
```

```
        }
    }
```

If `o` were declared `var`, then Mocha would not be able to determine the type of the expression `x.o` since it is a reference to `o` and hence `o`'s type would not be able to be inferred. The only way to break this cycle is to give `o` a static type in the source code.

5.3 Entered Characters

The goal of Mocha was to reduce the amount of redundant and duplicate type inference in local variable and field declarations. While the results in the previous two sections clearly indicate that most local variables and fields can have their types inferred by Mocha, they do not show the actual amount of type information Mocha saves the programmer from having to enter into the source code. To show this, the following formula was applied to each type inferred by Mocha for a local variable or field to determine the number of characters that were saved. Assume that Mocha has inferred a type T:

$$saved(T) = length(T) - pkg(T) - length(var)$$

In this formula,

- $saved(T)$ is the number of characters Mocha has saved the programmer for entering for the type T.
- $length(T)$ is the full length of the type T e.g. for the type `java.util.Vector`, the length is 17 characters.
- $pkg(T)$ is the length of the package information included in the full length of T e.g. for `java.util.Vector` the package information is `java.util` which is 9 characters. This length is subtracted from the

Benchmark	Number of Characters Saved
Robocode	12973
Javadoc	4712
Javap	1126

Figure 5.6: Statistics recorded in the evaluation of how many characters Mocha saves the programmer from having to enter.

total length as it is assumed that all types have associated `import` statements in the source code.

- $length(var)$ is the length of type `var` which is 3 characters. This is also subtracted from the total length since the programmer must give this type to local variables and fields if their real types are to be inferred.

The sum of all the saved characters was recorded for each benchmark and is shown in Figure 5.6.

It is clear from these results that Mocha saves the programmer from having to enter considerable amounts of redundant and duplicate type information. Hence the programmer's efficiency will be greater with the produced source code being 'cleaner', more readable and easier to maintain.

5.4 Instanceof

In order to evaluate the effectiveness of Mocha's `instanceof` retyping which was described in Section 2.1.2, Mocha was extended to identify and provide warning messages for situations where a local variable was being cast to a type it had already been retyped to. An example of such a situation is provided below:

Benchmark	Removed Casts
Robocode	65
Javadoc	14
Javap	0

Figure 5.7: Statistics recorded in the evaluation of Mocha's support for `instanceof` retyping.

```
public void f(Object o) {
  if(o instanceof Integer) {
    Integer i = (Integer) o;
    // o has been retyped but also cast
    // to the type Integer
  }
}
```

Each benchmark was then compiled once so all such situations could be revealed. If any examples were identified, then the redundant cast was removed. If the variable being cast was being assigned a new variable, then all uses of the new variable were replaced by the original. The benchmarks were then recompiled and the number of casts removed was recorded. Situations where a variable was retyped due to an `instanceof` conditional but the variable was then not used, were not identified and were not recorded.

Figure 5.7 is a summary of the number of casts made unnecessary through Mocha's `instanceof` retyping support.

The results suggest that `instanceof` conditionals are in fact very rarely used in Java programs. In fact, the `Javap` benchmark was written without any `instanceof` conditionals. However, in the benchmarks were such conditionals were used, Mocha was able to remove some casts, which would improve the readability and maintainability of the benchmarks.

Benchmark	JKit Time (s)	Std Dev	Mocha Time (s)	Std Dev	Overhead (%)
Robocode	86.33	0.86	95.75	1.54	11
Javadoc	24.41	0.96	27.01	0.53	11
Javap	16.94	0.63	19.86	0.81	17

Figure 5.8: Statistics recorded in the evaluation of the overhead Mocha introduces into the compilation process.

5.5 Time

The final measure of the effectiveness of Mocha is the amount of overhead the type inference procedure introduces into the compilation process. In order to determine how much additional time Mocha adds to the total compilation time, each benchmark was compiled ten times first using JKit, and then using Mocha. The hardware and software specifications used for this evaluation are summarised below:

- Each benchmark was compiled on a dual core laptop with 2GB of RAM, running Ubuntu 7.10.
- Only one instance of the JVM was created when compiling each benchmark.

The time taken over the ten compilations was averaged and summarised in Figure 5.8.

The results reveal that Mocha does not create a substantial overhead in the compilation process. With an average overhead of 13%, even for the biggest compilations, Mocha is still a plausible extension to JKit. The overhead could be further reduced by optimising Mocha's inference algorithm and implementation.

Chapter 6

Conclusion

Static types in Java can reduce runtime errors by allowing many of the errors to be revealed at compile time. However, all too often local variable declarations contain redundant static type information. Type inference, a procedure where the types of variables are inferred from their usage at compile time, resolves this issue as much of the redundant information can be omitted. Because the inference occurs at compile time, the benefits of strong typing are maintained. Mocha is an extension to Java which supports type inference for local variables. However unlike other languages which also support local variable type inference (such as C#, Scala and OCaml), Mocha allows a variable to have different types at different points in the program. This feature is key to Mocha's ability to retype variables through the analysis of `instanceof` conditionals.

Fields also contain duplicate and redundant static type information. However, the inference of field types is more complex than local variables because fields can be affected by multiple methods and threads. Hence Mocha was extended to include a novel approach for inferring the type of a field which uses the values assigned to the field in its class's constructors.

To evaluate the effectiveness of Mocha, a series of real life benchmarks were compiled. The results from this evaluation revealed that Mocha is able to infer the static types of almost all local variables and over half of fields. They also revealed the Mocha's inference reduces the number of characters the programmer has to enter into the source code by many thousands. The overhead of Mocha added to the compilation process however, was just over 10%, making Mocha a viable extension to Java.

6.1 Contributions

The major contributions of this thesis were:

- The presentation of Mocha, an extension to Java that supports local variable type inference based on assigned values and `instanceof` conditionals.
- In depth discussion about the effect Java exceptions have on local variable type inference, and an implementation of Mocha's novel solution for incorporating exceptions into its type inference algorithm.
- The presentation of an extension to Mocha for inferring the types of fields from values they are assigned in their class's constructors.
- The compilation of a series of real life programs using Mocha, which demonstrated the usefulness of type inference and illustrated a number of situations where types of local variables and fields cannot be inferred.

6.2 Future Work

A number of future works have been considered during this thesis. These and more are summarised below:

- **Intersection Types** — As discussed in Section 3.2.3, Mocha's approach of arbitrarily choosing the join of two types from the set of possible types, could lead to an incorrect type being inferred for a variable. Therefore Mocha could be extended to use intersection types in its type hierarchy. This would mean that the hierarchy would be a semi-join lattice, allowing the least upper bound of types to be found.

- **Private Method Inference** — Mocha does not infer the types of method parameters and returns. This is primarily due to the fact that Mocha supports classes being compiled in isolation and it would be necessary to have access to the entire program to fully infer the types of some methods. However private methods belong to a single class. Therefore Mocha's type inference algorithm could be replaced with an inter-procedural constraint based algorithm that supported the full type inference of private methods.

- **Inclusion of additional methods in field type inference** — Currently Mocha only uses the values assigned the constructors of a field's class to infer the field's type. However some methods, such as initialisation methods, could also be included. Therefore Mocha's field type inference support could be expanded to allow the programmer to indicate in the source code which methods they would like included in the inference procedure.

- **Inference of core Java libraries** — Mocha was evaluated by compiling a series of real life benchmarks. However, further testing would confirm the usefulness of Mocha and would also reveal any areas where it could be improved. An ideal candidate for the extended testing would be the core Java libraries, which would fully test each of Mocha's type inference components.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

[3] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *European Conference on Object Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.

[4] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOSPLA)*, pages 479–498. ACM, 2007.

[5] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, May 2008.

[6] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141, 1993.

[7] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications (OOSLA)*, pages 169–184. ACM, 1995.

[8] M. Fahndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 33, pages 85–96, 1998.

[9] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proceedings of the Third International Symposium on Static Analysis (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 189–204. Springer-Verlag, 1996.

[10] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proceedings of the 7th International Symposium on Static Analysis (SAS)*, pages 199–219, London, UK, 2000. Springer-Verlag.

[11] A. Goldberg. A specification of Java loading and bytecode verification. In *Proceedings of the 5th ACM conference on Computer and communications security (CCS)*, pages 49–58. ACM, 1998.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[13] K. Hartness. Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges*, 19(4):287–291, 2004.

[14] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[15] J.-H. Hong and S.-B. Cho. Evolution of emergent behaviors for shooting game characters in robocode. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 634–638. IEEE Press, 2004.

[16] S. Horwitz, A. Demers, and T. Teitebaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[17] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. *SIGPLAN Lisp Pointers*, V(1):193–204, 1992.

[18] G. Lagorio and E. Zucca. Introducing safe unknown types in java-like languages. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC)*, pages 1429–1434. ACM, 2006.

[19] X. Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, 2003.

[20] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[21] C. Male and D. Pearce. Jkit. `http://homepages.mcs.vuw.ac.nz/~djp/jkit/`, 2007.

[22] C. Male and D. Pearce. Mocha: Local type inference for Java. `http://homepages.mcs.vuw.ac.nz/~djp/files/MP08.pdf`, 2008.

[23] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proceedings of the Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes In Computer Science*, pages 229–244. Springer-Verlag, 2008.

[24] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM*

SIGMOD international conference on Management of data (SIGMOD), pages 706–706. ACM, 2006.

[25] S. Microsystem. Openjdk, http://openjdk.java.net/, 2007.

[26] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecure Notes in Computer Science*, pages 355–382. Springer-Verlag, 1998.

[27] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17:363–371, 1978.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[29] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[30] M. Nelson. Robocode, http://robocode.sourceforge.net, 2007.

[31] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[32] Objective Caml. `http://caml.inria.fr/ocaml/`.

[33] M. Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[34] J. O'Kelly and J. P. Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. In *Proceedings of the SIGCSE conference on Innovation and technology in computer science education*, pages 217–221. ACM, 2006.

[35] N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 329–349. Springer-Verlag, 1992.

[36] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, New York, NY, 1991. ACM Press.

[37] T. Parr. Antlr parser generator. `http://www.antlr.org`, 2007.

[38] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, pages 37–42. ACM, 2004.

[39] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[40] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications (OOPSLA)*, pages 324–340. ACM, 1994.

[41] C. Pusch. Proving the soundness of a Java bytecode verifier specification in isabelle/hol. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 89–103. Springer-Verlag, 1999.

[42] Z. Qian. A formal specification of Java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, London, UK, 1999. Springer-Verlag.

[43] R. F. Stark, E. Borger, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*. Springer-Verlag New York, Inc., 2001.

[44] T. Wang and S. F. Smith. Precise constraint-based type inference for java. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 99–117. Springer-Verlag, 2001.