Richard Thrippleton

# Theory and implementation of side-effect free Java

Computer Science Tripos, Part II

Churchill College

2003

# Proforma

| | |
|---|---|
| Name: | **Richard Thrippleton** |
| College: | **Churchill College** |
| Project title: | Theory and implementation of side-effect free Java |
| Examination: | Computer Science Tripos, Part II, June 2003 |
| Approximate Word Count: | 11000 |
| Project Originator: | Richard Thrippleton |
| Project Supervisor: | Dr Tim Harris |

## Original aims

This project aims to extend the Java language to allow methods to be marked as side-effect free, with respect to existing program state, and to produce/modify a compiler to respect this language extension. By marking a method with the appropriate keyword to assert that it is side-effect free, the compiler would attempt to prove this assertion, and treat failure to construct a proof as being like any other compiler error due to illegal Java.

## Work completed

- The official Java compiler from Sun has been modified to accept this language extension.
- A theoretical background to the theorem prover implemented within the compiler has been produced.
- An extra type of (useful) assertion has been added to the language extension and theorem prover.
- An extension to the original project has been implemented to allow side-effect free assertions to be added to methods in the standard Java API.
- The compiler has been tested on a wide variety of code examples, and found to be widely applicable and useful.

## Special difficulties

None

# Declaration of Originality

I Richard Thrippleton of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extend for a comparable purpose.

Signed

Date

# Contents

# List of Figures

# Acknowledgements

I would like to thank my supervisor, Tim Harris, for all his advice and support throughout this project

For his assistance in comprehending the source code to Sun's Java compiler, I would also like to thank Alan Lawrence

# Chapter 1

# Introduction

In the world of high level programming languages, in addition to increasing ease of implementing high level concepts, one language goal is restricting the environment in such a way as to clamp down on programmer error to the benefit of the resulting program. A trivial example of this is array bounds checking; in the Java language, attempting to write to or read from an index beyond the array's size will consistently cause a comprehensive runtime error, as opposed to the undefined behaviour in languages such as C or C++ that could go undetected on a non-deterministic basis.

Of more relevance to this project is typing; in a typesafe language such as Java, a programmer can be confident that if a non-null variable is of type T, then the value of that variable will indeed behave like type T, be it a HashMap or a String. It is considerably more difficult for a programmer to use a variable/object in a mistaken fashion if they know with certainty what the type is.

This project and the resulting implementation is a form of behavioural typing; similar to the Java typechecker's confidence guarantees about types, it ensures that any method declared to be side-effect free can be treated with absolute confidence as being side-effect free by the programmer attempting to invoke it. In slightly more detail, this means that the method's invocation has no effects that modify existing program data; it may return and create new data, but nothing else. This is one of the properties of a *function*, as described in general mathematics or in functional programming languages. To more experienced programmers, the use of this is quickly apparent; in a language such as Java, the effects of all invocations must be carefully considered by the conscientious programmer, but with liberal use of functions (the term that shall be used from now on for side-effect free methods) where appropriate, the journey to full confidence in the correct operation of a program is eased.

A small example of what is and is not a function:

```
class number
{
    public number(int v)
    {
        value=v;
    }
```

```
    //Doubles the value of this number
    public void doubleMe()
    {
        value=value*2;
    }

    //Returns a new number that is double the value of this one
    public number doubleFunction()
    {
        return new number(value*2);
    }

    public int value;
}
```

The class "number" shown above encapsulates an integer, and can perform one operation; doubling the value. "doubleFunction" is a function, because it does not modify anything that the program had access to before invocation. One can quite happily invoke this method on one object as much as you wish, and no existing data will be modified. "doubleMe" on the other hand changes the object it is invoked upon, a noticeable side-effect.

The ML language is one example of a functional language; when refs are discounted, all of its functions are in fact functions as defined above. The only effect their invocation has on program behaviour is on that which uses the return value[1]. Much past experience, both in demanded coursework and in personal interest use has shown that writing bug free ML for a particular task is that much easier than with a non-functional language; there is no *hidden flow of data*, and no unexpected or accidental surprises that a function is able to cause. That said, writing useful programs in such a stateless environment can prove awkward, in a way that they would not in a procedural language with side effects such as Java. With this in mind, a hybrid of their best features is justified; the fluid programming environment of Java, with the safety guarantees of a functional language to be applied as much as the programmer sees fit (i.e. until it becomes awkward and contrived).

In the remainder of this dissertation, the implementation of such behavioural typing is realised as a modified compiler

Section 2 describes the planning and research, decisions on how the behavioural typing is to be realised

Section 3 presents the work that was actually done in this realisation

Section 4 attempts to judge how effective the project was at allowing a programmer to correctly use functions

[1] When one discounts non-halting functions, which are not an issue covered by this project

Section 5 draws conclusions from the evaluation, both a recap of success and a critical self assessment with suggestions on how the project could have been done better

# Chapter 2

# Preparation

Before code was written, or theory developed, it was necessary to prepare; I needed to know exactly what needed doing, how I might do it, and also learn about previous related work, knowledge of which could possibly aid/optimise the implementation.

## 2.1 Requirements analysis

As this project can be seen as a Java typechecker for a specific category of behavioural types, some requirements can be inferred from other typecheckers; these are, obviously, almost as numerous as compilers. One requirement that became rapidly apparent from both programming experience and from consultation with my supervisor, is the requirement that the checker have the property informally known as "safety". Further background to this theory was later taken from Part II of the Computer Science course[6]. Safety concerns itself with the difference between what is known about types of symbols in a program at runtime, and what is known at the time of static analysis (i.e. at compile time). In a language as complex as Java, it is not possible to create a static checker so powerful that it can determine for all cases the most specific type that a symbol may have at runtime, as that would often entail complete program analysis, so it will *err on the side of correct typechecking*. This is safety of type judgement, and can be expressed as

"s typechecks as type T" ⇒ "`s instanceof T` will always evaluate to true at runtime"

whereas the reverse is not necessarily so

"`s instanceof T` will always evaluate to true at runtime" ⇒ "s typechecks as type T"

This translates simply to the safety requirement that the implementation of the functional extension must conform to. That is to say, any method declared to be a function must indeed behave like a function, yet a method that behaves like a function may not necessarily be permitted by the compiler to call itself a function. Such a type judgement system therefore must be *sound* but not necessarily *complete*.

There is a need to precisely define the behaviour of a "function"; the official Java typing can be formally defined, so no less should be expected of added behavioural typing. Ambiguity is not appropriate within a structured safe language such as Java. It seems unlikely that one can create a definition that is both formal in nature and is exactly equivalent to "all subsequent behaviour of the program, save that depending on the return value of the method, will be totally unaffected by invocation of a function", so to keep safety properties, the resulting definition used at compile time will be one that is logically stronger (a stronger statement being one that implies the other but not equivalent).

While it might not appear relevant at first glance, a compiler still has a user (or "developer" if you prefer) interface, and as such should follow good HCI practices; efficiency in the software development process is just as important as, if not more so than John Q Public's desktop productivity tasks. As the behavioural typechecker will be implemented as an extension to an existing Java compiler, an important HCI heuristic is that of consistency. The checker should be consistent with developer's intuition about "normal" Java compiler behaviour; being a typechecker, this will mean being entirely silent when the behavioural typechecking succeeds, and outputting an error when it fails, usually to console and in the same format as other Java compilation errors. Additionally, the error output should be clear, and correspond to the programmer's way of thinking as opposed to being composed of internal implementation oriented terms.

## 2.2 Research

This section covers similar behavioural type systems that have previously been implemented. The purpose is to inspire and lend past experience to the implementation of the functional extension.

As described in the Standard C++ Bible[4], the `const` keyword describes a similar form of behavioural typing to functions. Any value typed as being `const` is defined as being *immutable*. For primitive values, such as `int`, `char`, `bool` etc., this is enforced by the compiler forbidding assignments to them, and this suffices. For aggregates (class instances), on top of these constraints, the compiler forbids methods being called upon them as they could change the instance internally. The exception to this is methods declared as being `const`; this declares that the methods will not modify fields of the instance they are invoked upon. The `const` modifier in this context is itself the behavioural typing. This holds some similarity to side-effect free methods, and some useful ideas can be picked out;

- A `const` method containing invocations to other `const` methods trusts these declarations at checking time, allowing them to be invoked without harming its own `const`ness, forsaking any further program analysis. It is easily apparent how this recursive trust concept can be applied to function methods.

- For all methods that dispatch dynamically (`virtual` methods) on runtime type,

`const` must be inherited[1]. That is to say, if type $U$ is a subtype of type $T$, then if virtual method $T :: m$ is `const` then $U :: m$ must also be `const`. Else otherwise the `const` guarantees would not hold, as invoking $m$ on something typed as $T$ might well invoke non-`const` $U :: m$ where it was expected to be `const` based on `const` $T :: m$. The lesson to learn here is that the inheritance/subtyping properties of behavioural typing must be considered in the realm of object orientation.

The Eiffel language[7] was an insightful case study into behavioural typing in a language. Defining the behaviour of a method in Floyd-Hoare logic[3], it is not constrained to a limited range of behavioural types as other languages are. This is not exactly a static typechecker as this project is, as the preconditions and postconditions are only checked at runtime. In this context though, this can be overlooked and treated as an implementation detail that doesn't detract from the significance and utility of the principles behind it. This flexible typing system also copes well with subtyping; methods in a subtype are only allowed to have stronger postconditions and weaker preconditions than the methods that they override, if any. This is in keeping with the Hoare logic axioms of postcondition weakening and precondition strengthening, inferring the properties of a method in the supertype from the behavioural properties of the same method in the subtype. In this way, the "Is-A" property of subtyping is obeyed; when $T$ is a supertype of $U$, one can properly say that "$U$ is a $T$" because you can treat it like a $T$ and it will act like a $T$. $U :: m$ will still be satisfied by the stronger preconditions put in place for $T :: m$, and postconditions of $U :: m$ will satisfy the weaker postconditions demanded of $T :: m$.

- Again, one must take care to consider subtyping when adding behavioural types to a language.

- Floyd-Hoare logic is a powerful foundation for any sort of behavioural typing, doubly so as there are already known and very flexible methods for reasoning about behavioural subtyping within it.

### Floyd-Hoare logic

Although Floyd-Hoare logic is referenced in the bibliography([3]), it is utilised sufficiently in this section to justify a brief summary of what it is and does.

Floyd-Hoare logic is a way of reasoning about the behaviour of a given command/sequence of commands in an arbitrary programming language. Behaviour is asserted in the form of a Hoare triple. The most generalised Hoare triple looks like this:

$$\vdash \{P\}C\{Q\}$$

[1]This is not strictly true; if you try to override a const method with a non-const, it will just treat the non-const method as being a brand new method defined only in the subclass; calls that are expected to be to a const method of the same name will still go to the parent class. But this is effectively the same as forbidding overriding

$P$ and $Q$ are boolean expressions, evaluating to either true or false, and conventionally contain some assertion about program state, like the value of variables or relationships among them. $C$ is a command in the programming language. The above statement means

> "In all interpretations, if $P$ is true at a certain time, and $C$ is then immediately executed after that time, then $Q$ will hold true immediately after $C$ has finished executing if it halts"

More complex examples can be found in the referenced course notes, but the concept can be demonstrated with a simple example:

$$\vdash \{X = 2\}X = X + 1; \{X = 3\}$$

This just shows how the value of the variable $X$ will change if you increment it by 1 when it holds the value 2. If X starts of with any other value, that is undefined behaviour as the above rule doesn't cover it. To generalise the above, Hoare logic allows one to instantiate arbitrary variables that are not necessarily part of the program state, only relevant to the logic and rules, known as ghost variables. These are traditionally lower case:

$$\vdash \{X = n\}X = X + 1; \{X = n + 1\}$$

Floyd-Hoare logic is most useful when used with inference rules, logic that infers a conclusion given some hypotheses. An inference rule has the meaning that if the hypotheses of the rule are true, then the conclusion is true. For example:

$$\frac{\vdash \{P\}C_1\{Q\} \ , \vdash \{Q\}C_2\{R\}}{\vdash \{P\}C_1; C_2\{R\}}$$

This is the *Sequencing* rule of Floyd-Hoare logic, and means that if $C_1$ and $C_2$ are executed in sequence with precondition $P$ holding, then postcondition $R$ holds if you can find $Q$ such that $C_1$ and $C_2$ execute individually as $\vdash \{P\}C_1\{Q\}$ and $\vdash \{Q\}C_2\{R\}$ respectively.

More elaborate inference rules, more specifically related to Java and functional behaviour were eventually used in the implementation of this project.

## 2.3 Development tactics

### 2.3.1 Resources

Implementing an entire Java compiler with functional extensions was out of the question due to time constraints, but source code to other major Java compilers exists which can be extended; then the work is cut down to the more feasible task of implementing the functional extensions. Sun's `javac` and IBM's `jikes` were both contenders; both of these were obtained and examined, and found to be acceptable with

respect to ease of understanding for the purposes of modification. The decision was eventually made in favour of Sun's `javac`, primarily because I knew for certain that modification was actually feasible; I am in contact with a colleague who modified this Java compiler for a previous part II project[1], which also covered typechecking. I had no such assurances for `jikes`, but knew that my colleague could be contacted in the event of trouble modifying the `javac` source.

The language of implementation was chosen to be standard Java. I do not believe it to be the ideal language for implementing a typechecker/prover, but it is required that the project be integrated with the compiler; Sun's `javac` is itself written in Java, a language which does not cleanly interface with others[2] Before the project I was already very familiar with the Java language, so the rate of implementation was not likely to be hindered by a language learning curve.

The hardware and software requirements for development are not particularly demanding. Enough filespace to hold the `javac` source (less than 10 megabytes) plus the source code I myself developed, and a Java development environment (for my particular development tastes, an unmodified standard Java(1.3) compiler and the `gvim` editor). I always intended to use my own PC for this, with thorough planning to ensure that work would not be significantly set back in the worst case scenario of its total destruction. Before I wrote a single line of code, a script was installed on my computer that automatically backed up all work to another filesystem on my computer, as well as to the Pelican server run by Computing Services. This was run daily. In the event of disaster, I resolved that I would retrieve my work off of Pelican or whatever remained of my hard drive and use the Linux PWF system to continue my work. I would work at either one of the workstations in the laboratory booted into Linux, or using a public college workstation as a remote X terminal for running my development efforts off of `{linux2,linux3}.pwf.cl.cam.ac.uk`. The PWF system has enough personal filespace for my project, as well as all the software required[3]. I consider this a more than adequate contingency plan, coping as it does with total destruction of the primary development platform without the loss of any more than 24 hours of work.

### 2.3.2 Procedure

I expected much of the work to be in constructing the formalities of side-effect free code, and theories/rules to be applied against code claiming to be side-effect free. The implementation of these rules into code was planned to resemble the rules and notation used in the theory as closely as possible; with rules already in place, it was reasoned that they could be implemented in programming one at a time, and tested against the rules, each of which would be an unambiguous model of how the code should behave, giving confidence that the implementation followed the rules as de-

[2]Passing data structures, especially those as complex as syntax trees through JNI to another language does not make for clean code. Sharing data structures across the Java/non-Java boundary is most certainly not transparent, requiring explicit translation.

[3]Except for `gvim`, but installation of this into my personal space is an easy matter

fined. To wit, the Iterative development model would be followed, with a very short write-compile-test cycle. The task at hand would allow this model to work very well, with the advantage that errors can be tracked down to the chunk of code recently worked upon, as opposed to potentially being anywhere in the code if found in one final testing stage.

Where many tried and tested development models exist for producing a software product from an informal specification (in the case of industry, the type of document a manager would deliver to his/her development team), similar methodologies are not so numerous for development of formal theories. I did not take this as being much of an issue, as the idea of correctness in theory development is far less fleeting. As one develops a theory, one can check that it is true, owing to the precision of logic and mathematics; I planned to do just this as I developed the theoretical backing for this project. No such concept of correctness exists in the traditional models of software development. At this time, I did not know what form of logic and reasoning I would be applying, as I considered that choice to be dependent on the definition of side-effect free that I would eventually decide upon.

The most difficult software challenge I expected to encounter was in integrating the prover/typechecker into `javac`. Despite it being well documented, it is still an enormously large and complex source tree, developed with years of manpower. I did not expect that I would comprehend it in its entirety in the time allowed. The intention was to have as much of the project code disjoint from the `javac` source tree as possible, analysing as little of it as possible, only enough to locate appropriate "hook" points within the compile process where my methods could be invoked.

The plan in short:

Define "functional" in formal terms

↓

Develop formal rules and notation with the power to prove that a method is "functional" or not

↓

Analyse `javac` source code with a view to locating the appropriate "hooks" where the concept of functions in the language can be added

↓

Rewrite rules into procedural code that applies them

↓

Integrate rule based code into the whole compiler using the hooks

### 2.3.3 Plan of testing

I did not expect to be able to produce a thorough plan of testing at this stage, as much of the testing would involve passing source code utilising the behavioural typing extension through the compiler; this domain of testing would be dependent on the decisions and definitions made at implementation time with respect to functions.

However, there are still elements of the testing plan that can be and were set down at preparation time:

- Verify that the modified compiler still behaves identically to `javac` when given trivial and non-trivial normal Java code
- Utilise the language extension to demonstrate and successfully compile non-contrived functional style code; that is to say, ensure that the project fulfils its purpose

## 2.4 Examining `javac`

Although at this point the theory had not been defined, examination of `javac` was useful preparatory work, if only to determine what was feasible.

To this end, I had to traverse most of the `javac` source code, although not in a great amount of detail. A cursory examination showed that it roughly approximated the

`Tokenise` → `Parse` → `Generate intermediate code` → `Generate target code`

model of compilation[6], except that the last two stages are collapsed into one, as JVM bytecode is used as both an intermediate and target machine code.

What was found:

- Code being compiled is formed into a syntax tree(Figure 2.1), each node of which contains an abundance of information, more than enough to reason about in detail
- Method keywords will need to be added, however the implementation is approached, to allow programmers to declare a method as functional: the tokenizer is a work of clarity, and adding keywords for this should be no difficulty whatsoever
- A clearly documented method for outputting errors in the program being compiled was found: it handles line number and method name reporting with little effort required from the caller

## 2.5 Summary of preparation

- Soundness in the behavioural type judgements should be guaranteed, not necessarily completeness
- The behavioural type “function” should be formalised
- Some HCI requirements should be considered during implementation

For example:

```
while(x)
{
    if(y)
    {
        x=x+1;
    }
    else
    {
        aMethod(y);
    }
}
```

Becomes the syntax tree:

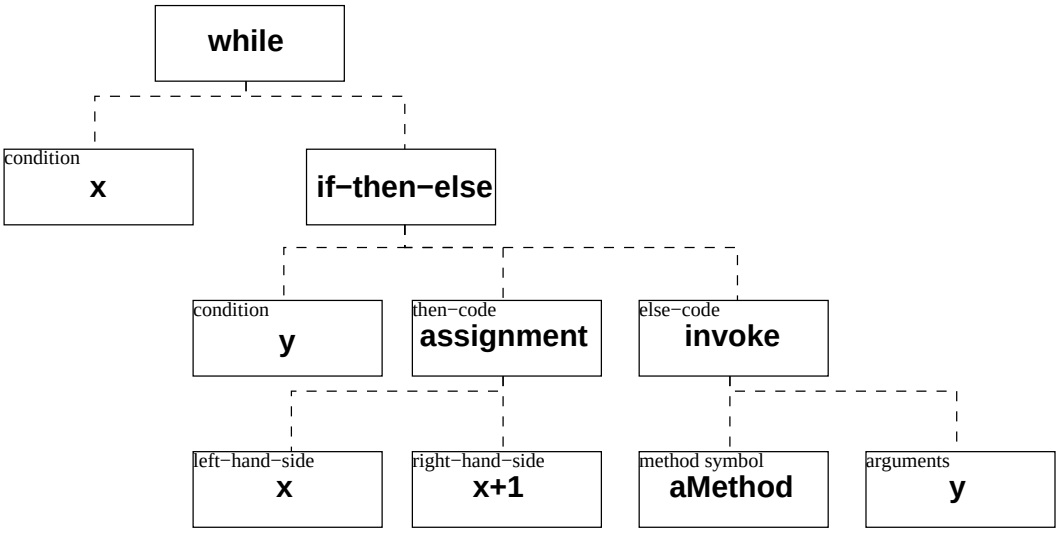


Figure 2.1: An example syntax tree within `javac`

- The checker should be implemented within Sun's `javac` (Java Compiler)
- The iterative development model is to be used in conjunction with the code mirroring the theory's notation and logic

# Chapter 3

# Implementation

This chapter describes the creation of a theory that can determine if Java methods are "side-effect free", as well as the integration of this theory into a Java compiler which applies it to specific methods at compile time.

## 3.1 Theory

This section describes the definitions, notations and proof rules that were created before any code was written, for the task of checking the behavioural type of `function` against the implementation of the method that is asserted by the programmer to be a `function`.

### 3.1.1 Defining functions

Creating a precise definition of "side effect free" ("function") was the first task on the agenda; all other tasks were dependent on this. Returning to the introduction,"the function's invocation has no effects that modify existing program data; it may return and create new data, but nothing else". The sort of data(or program state) constraints demanded by a function are illustrated in Figure 3.1. Note that this does not preclude functions from creating extra state, that is, creating `new` objects and modifying them within its own execution, as this extra state did not exist before invocation and hence evades the definition.

With this in mind, I started to design a more formal logic and notation for reasoning about program state, utilising set theory. The core of this is defined in Figure 3.2.

Lending from the example of Eiffel's behavioural typing, the definition of a function seemed appropriate to express in Hoare logic. A small amount of thought lead to the following definition:

$$\{\texttt{State} = s\}\texttt{Function call}\{\texttt{State} \supseteq s\}$$

$s$ is defined as the `State` before invocation, and it is asserted that the same $s$ exists

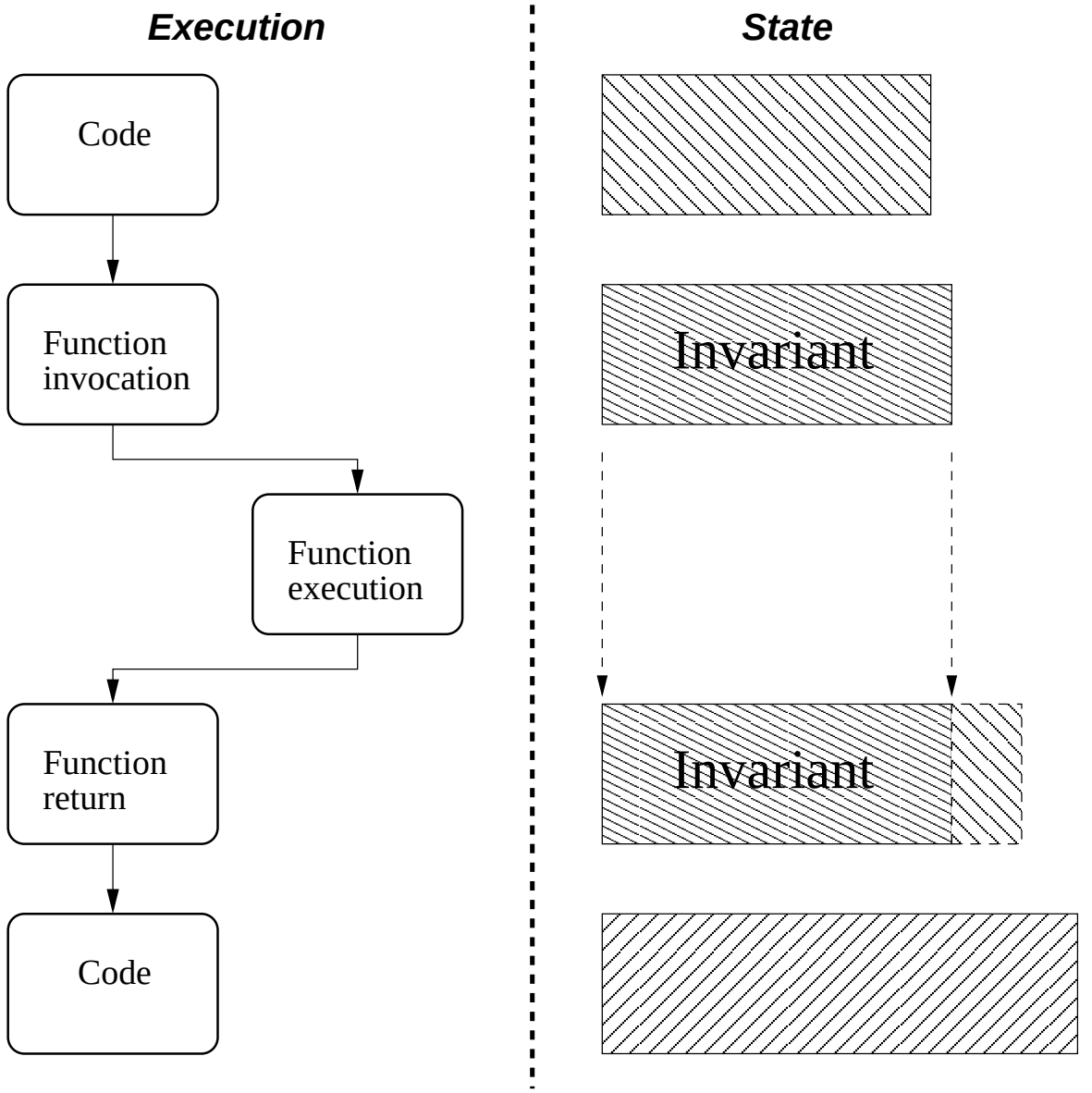


Figure 3.1: Illustrating program state informally against the execution of a function

- `fields`: the set of all member fields of all Java objects that have been instantiated thus far and all static fields of all loaded Java classes.
- `values`: the set of all possible primitive values in Java; bytes, integers, characters, floating point numbers and object references.
- `State` : `fields` $\rightarrow$ `values`. A property of the Java environment at a certain point in time; it is a *function* from each existing variable to its values.
- For any Java object $o$, it follows from the above that $o \subseteq$ `State`

Figure 3.2: The State notation

as a subset of the `State` after invocation (i.e. all fields existing before invocation have the same value before and after).

### 3.1.2 Local variable tracking

To arrive at the decision that a method satisfies the given Hoare triple for a `function`, a proof system was needed to operate over the code implementing the method. As the definition was defined in Hoare logic, it followed that the proof rules should be defined similarly; if Hoare triples asserting conditions about `State` for each Java programming structure can be written, then they can be applied recursively on the method to see if the Hoare triple matches the function definition.

When the first simple rules were being devised, it was quickly realised that there was a danger of making rules that, while safe, simply didn't allow the programmer enough power. They would of course, make sure that non-functional code was not allowed to be accepted as being a function (no false-positives), but a great deal of code that always behave functionally would also not be accepted as as a function (too many false-negatives). That is, *safe* but not *complete*. One example of such a rule follows, for an assignment:

$$\{\texttt{State} = s\}\text{lhs} \ = \ \text{rhs}\{\texttt{State} = t\}$$

It "taints" `State` to being a new value, with no extra information given; the reasoning is that an assignment of a value might modify existing state, so this is sound (if it doesn't actually modify state, then this is still sound as $t \neq s$ is not necessarily asserted). So if an assignment were the only command in the function, judgement could only say that $t$ was *not necessarily* equivalent to, or a superset of $s$, so it could not accept this assignment as functional. The flaw here is easily apparent; $lhs$ might be a field in an object that has just been instantiated with `new` $()$, and $rhs$ is just a constant so the assignment would have functional behaviour in all cases, as it does

```
public void function aFunction()
{
   Thing loc=new Thing();
   l.a=2;
   l.b=4;
}
```

Figure 3.3: A function that relies on local variable tracking to be accepted

not modify any `State` that existed in the precondition. This rule is naïve and simply not powerful enough for real functional code.

A distinction needs to be drawn between objects that are obviously not subsets of the `State` in the precondition and those that are, to allow sound and useful assignment rules. Attention also needs to be paid to local variables; it is common idiom to assign object references to local variables to access them later. It would be desirable for the code in Figure 3.3 to be accepted as a function. If the values of local variables were tracked in preconditions and postconditions in a similar manner to `State` assignments and similar Java constructs can be reasoned about with far more context than the previous simpler scheme, hopefully resulting in less false-negatives.

The notation for local variable tracking is defined in Figure 3.4.

### 3.1.3 Extending Hoare logic for unusual control flow

The next step was in dealing with the shortcomings of Hoare logic with respect to unusual control flow. As soon as it was decided to introduce local variable tracking, these shortcomings became apparent. When one has a Hoare triple $\{P\}C\{Q\}$, $Q$ only describes the conditions that must hold when control flow goes normally through $C$ and exits normally e.g. in the case of a sequence, the last instruction executed is the last instruction in the sequence if $Q$ is definitely to hold. Were $C$ to contain a `break` or `throw` , control flow would leave abnormally, and re-enter elsewhere in the method possibly. This is unhelpful within a method that we are trying to reason about for functionality; we have no potential precondition for the code that abnormal control flow will re-enter at. While the logic could be modified to treat the postcondition as holding for normal exit *or* exceptional exit, this weakens the logic far too much and would make for exceedingly weak preconditions at re-entry points. Inspired by the work of Michael Norrish in reasoning about the C language[8], an extended Hoare logic was defined, with multiple postconditions that hold for each way of exiting the code being reasoned about (Figure 3.5).

By observing that control flow can leave a method by either running to the end of code, a `return` statement or a `throw` , but not by `break` or `continue` statements,

- `vars`: the set of all local variables currently within scope
- `values` needs defining further.
  `values` $= \{$`Unknown`, `undefined/NULL`, `reference to` $x\}$
- `Unknown` : a value that is not known at the time of proof
- `undefined/NULL` : the "uninitialised" value taken by new variables that have not yet been assigned to, or an object reference to `null`
- `reference to dom`$(x)$ : an object reference to $x$ such that $x \subseteq$ `State` (Parameterising references by only the domain of the `State` relation allows the references to remain valid while field values change)
- `Local` : `vars` $\rightarrow$ `values`. A property of the Java environment at a certain point within execution of a method, concerning the value taken by each local variable currently within scope

Figure 3.4: Logic for local variable tracking

we can rewrite the earlier `function` definition into the extended Hoare logic to get this:

$\{$`State` $= s\}$
`Function call`
$\{$`State` $\supseteq s\}\{$`EX:State` $\supseteq s$, `BREAK:f`, `RETURN:f`, `CONT:f`$\}$

`BREAK:f` and `CONT:f` are simply present to assert that control flow cannot leave a function invocation by those routes. Leaving via the other routes (exception throwing and explicit returns) still respects the fact that a function must be side-effect free.

### 3.1.4 The `eval`() notation

A final piece of notation that the proof rules require is, `eval`(), which takes any Java expression and maps it to `values`, essentially evaluating the expression as specifically as possible with the information available (that that information being the context of the conditions in the current proof rule).

`eval`() handles local variables:

$$v \in \texttt{dom(Local)} \Rightarrow \texttt{eval}(v) = \texttt{Local}(v)$$

Also new objects:

$$\texttt{eval(new } \mathit{Class}(\ldots)) = \texttt{reference to } n$$

$$\begin{array}{l}\{P\}\\ C\\ \{Q\}\{\mathsf{EX}:R, \mathsf{BREAK}:S, \mathsf{RETURN}:T, \mathsf{CONT}:U\}\end{array}$$

If logical statement $P$ is true, immediately before $C$ executes, then if program flow exits $C$ normally, statement $Q$ will hold. If program flow leaves via an exception being thrown, $R$ will hold. If exit from $C$ is caused by a `break` , $S$ will hold. If a `return` statement causes exit of this code block, $T$ will hold. And finally, if a `continue` statement causes an exit, $U$ will hold.
There exist some convenient abbreviation rules for this notation:
In

$$\begin{array}{l}\{P\}\\ C\\ \{Q\}\{E\}\end{array}$$

$E$ is an abbreviation for the full $\{\mathsf{EX}:E_e, \mathsf{BREAK}:E_b, \mathsf{RETURN}:E_r, \mathsf{CONT}:E_c\}$, where one does not wish to reason about each of the exceptional control flow conditions separately. If used elsewhere in the same inference rule, the meaning of any operation on it is to unfold into constituent parts and apply individually:

$$\{E \vee F\} \Leftrightarrow \{\mathsf{EX}:E_e \vee F_e, \mathsf{BREAK}:E_b \vee F_b, \mathsf{RETURN}:E_r \vee F_r, \mathsf{CONT}:E_c \vee F_c\}$$

The rule of postcondition weakening in Hoare logic also applies to the extra postconditions in the same way it does to the postconditions for normal exiting of control flow. So for example:

$$\frac{\vdash \begin{array}{l}\{P\}\\ C\\ \{Q\}\{\mathsf{EX}:E, F\}\end{array} \quad , \; E \Rightarrow E\prime}{\vdash \begin{array}{l}\{P\}\\ C\\ \{Q\}\{\mathsf{EX}:E\prime, F\}\end{array}}$$

Figure 3.5: The extended Hoare logic used in this project

However, this is in the context of the proof rule for instantiating objects:

$$\frac{\begin{array}{l}\{P[\texttt{State}\prime/\texttt{State}], \texttt{State} = \texttt{State}\prime \cup n\} \\ \vdash\ (\texttt{reference to dom(n)}) \cdot \texttt{<init>}(\texttt{C1}, \ldots, \texttt{Cm}) \\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\} \\ \vdash\ \texttt{new}\ \mathit{Class}(\texttt{C1}, \ldots, \texttt{Cm}) \\ \{Q\}\{E\}\end{array}}$$

Also, `null` has special meaning:

$$\texttt{eval}(\text{null}) = \texttt{undefined/NULL}$$

If none of these rules match the expression, there is the default evaluation, which is both true and less helpful:

$$\texttt{eval}(E) = \texttt{Unknown}$$

Complementing evaluation of references, an extra dereferencing function '!' is briefly utilised in the proof rules:

$$\texttt{!(reference to dom}(o)) = o$$

### 3.1.5 Special casing for constructors

Another important issue not covered so far is the consequences of instantiating objects within methods, where the object has a constructor method[1]. While the rules have not been set down thus far, it is intuitively obvious that with the system we have so far, the constructor cannot be a normal (non-functional method), otherwise this would prevent the enclosing method from being passed as a function; as far as the proof system knows, calling of any non-function could taint all state arbitrarily. This however implies a useless constructor; constructors have no return values, so their only useful effects involve modifying state, usually that belonging to the newly instantiated object that the constructor[2] has been invoked upon. While there may exist any number of ways of allowing useful constructors through special casing for constructors, it was decided to exploit this issue to further increase the power[3] of the function proof system by adding an extra behavioural type to go alongside `function`; the `mutator` type. It is expected that all constructors useful within a function will be of type `mutator`, with the additional power of allowing other arbitrary methods to be typed as `mutator`s.

[1] As a refresher, a constructor in Java is a method that is called upon an object immediately after it is instantiated. The identity and arguments of this method are selected from the constructors in that class and determined by how the `new` ...() statement is used. A constructor is typically used to initialise an object into a consistent state.

[2] Or more correctly, the appropriate `<init>` method within the class; a Java `new` operation involves allocation of a blank object followed by invocation of `init` upon it

[3] As elsewhere in this dissertation, this refers to decreasing the number of false-negatives judged by the system; thus allowing more code to pass as functional while still being sound

Informally, a `mutator` may only modify the object that it is being instantiated upon. The behaviour of a `mutator` is set down more formally below:

$$\begin{array}{ll} & \{\texttt{State} = s \cup t\} \\ \vdash & \texttt{Mutator call upon object } t \\ & \{\texttt{State} \supseteq s\}\{\texttt{EX:State} \supseteq s, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\} \end{array}$$

By splitting initial state into two distinct chunks, and declaring that one of them is the object that the mutator is being invoked upon, the above asserts that only part of the state need be invariant, while allowing the object that the `mutator` is being invoked upon to be modified without violating the specified behaviour. For the sake of consistency, if a `mutator` is a static method, we shall take $t = \emptyset$ in the above rule.

At this point, it is appropriate to say that `function` is a behavioural subtype(Section 2.2) of `mutator`. It is enough to show that a `function` *is a* `mutator`. The proof for this is in Figure 3.6.

To fit entirely normal Java methods (those that the programmer hasn't asserted behaviour for), we have the following definition:

$$\begin{array}{ll} & \{\texttt{State} = s \cup t\} \\ \vdash & \texttt{Normal method call on object } t \\ & \{\texttt{State} \supseteq \emptyset\}\{\texttt{EX:State} \supseteq \emptyset, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\} \end{array}$$

That is, a normal method with no assertions about side effects or otherwise is treated as unreliable, and can potentially modify any and all elements of `State`. Completing the subtyping relations, it can be shown (Figure 3.7) that `mutator` is a behavioural subtype of normal methods. Knowing that `function` *is a* `mutator`, and `mutator` *is a* normal method, this subtyping is transitive, with `function` also being a subtype of normal methods.

### 3.1.6 The proof rules

This section contains a selection of inference rules in Floyd-Hoare logic over Java; this selection is considered to contain the more interesting rules. The remainder are covered in Appendix A. All of these rules together are sufficient to reason about any Java method and judge if it is a `mutator` or `function`.

#### Function invocation

This rules specifies the behaviour of a statement invoking a `function`.

$$\frac{\vdash M \text{ is a } \texttt{function}, \quad \begin{array}{l} \{P\} \\ \vdash \texttt{O}; \texttt{arg1}; \texttt{arg2}; \ldots; \texttt{argn} : \\ \{Q\}\{\texttt{EX:}E, \texttt{BREAK:}F, \texttt{RETURN:}G, \texttt{CONT:}H\} \end{array}}{\begin{array}{ll} & \{P\} \\ \vdash & \texttt{O}.M(\texttt{arg1}, \ldots, \texttt{argn}) \\ & \{Q\}\{\texttt{EX:}E \vee Q, \texttt{BREAK:}F, \texttt{RETURN:}G, \texttt{CONT:}H\} \end{array}}$$

Start with the assertion:

$$M \text{ is a } \texttt{function}$$

By the definition allows us to infer:

$$\begin{array}{ll} & \{\texttt{State} = s\} \\ \vdash & \texttt{call}\ M \\ & \{\texttt{State} \supseteq s\}\{\texttt{EX:State} \supseteq s, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\} \end{array}$$

Knowing that $M$ must be being invoked on a subset of $s$ (either an object, or $\emptyset$ for static methods), we split `State` into chunks such that $s = t \cup u$, with $u$ as the object $M$ is being invoked upon and $t$ being the remainder of `State`:

$$\begin{array}{ll} & \{\texttt{State} = t \cup u\} \\ \vdash & \texttt{call}\ M \text{ upon } u \\ & \{\texttt{State} \supseteq t \cup u\}\{\texttt{EX:State} \supseteq t \cup u, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\} \end{array}$$

Following on, postcondition weakening with $\texttt{State} \supseteq s \cup t \Rightarrow \texttt{State} \supseteq s$ allows us to infer:

$$\begin{array}{ll} & \{\texttt{State} = t \cup u\} \\ \vdash & \texttt{call}\ M \text{ upon } u \\ & \{\texttt{State} \supseteq t\}\{\texttt{EX:State} \supseteq t, \texttt{BREAK:f}, \texttt{RETURN:State} \supseteq t, \texttt{CONT:f}\} \end{array}$$

Which leads us on to:

$$M \text{ is a } \texttt{mutator}$$

By its definition.
Note that this process cannot be reversed; $\texttt{State} \supseteq s \Rightarrow \texttt{State} \supseteq s \cup t$ is not true. So we cannot show that a `mutator` is a `function`.

Figure 3.6: Proving that `function` is a behavioural subtype of a `mutator`

Starting with,

$$M \text{ is a } \texttt{mutator}$$

By definition,

$$\begin{array}{rl} & \{\texttt{State} = s \cup t\} \\ \vdash & \texttt{call } M \text{ upon } t \\ & \{\texttt{State} \supseteq s\}\{\texttt{EX:State} \supseteq s, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\} \end{array}$$

Postcondition weakening using $\texttt{State} \supseteq s \Rightarrow \texttt{State} \supseteq \emptyset$,

$$\begin{array}{rl} & \{\texttt{State} = s \cup t\} \\ \vdash & \texttt{Normal method call on object } t \\ & \{\texttt{State} \supseteq \emptyset\}\{\texttt{State} \supseteq \emptyset\} \end{array}$$

By definition,

$$M \text{ is a normal method}$$

Figure 3.7: Proving that `mutator` is a behavioural subtype of the normal method

The Hoare triple of the invocation is composed from the sequencing of evaluation of all the arguments (that any one of them could taint state or modify local variables should not be overlooked), plus the expression evaluating to the object being invoked upon, followed by the actual function call. The function call leaves invariant[4] all the variables in our logic (`State` and `Local`) and so the postcondition of the total invocation is in fact just the precondition of the call (the postcondition of the argument evaluation), $Q$.

The abnormal exit conditions are simply those of the argument and expression evaluation, disjuncted(logical or applied) with those of the call itself (exception) which due to the invariant nature of the call is again its precondition, $Q$.

The $M$ mentioned in the top part of the rule refers to a method within a particular class. That class can only be deduced to be the type of `O`, whereas at runtime it may in fact be a subclass. This implies the possibility that it may not be method $M$ being called, but one in a subclass overriding it. Hence, this rule can only hold with the further constraint that *all methods overriding a* `function` *must themselves be a* `function`.

[4]The earlier definitions only mention state, not mentioning `Local`, the other variant in our logic. However, Java local variables are scoped by the method that is currently executing, so $M$ could not possibly modify `Local` in the context of the calling method

### Function implementation

This rule specifies what properties must hold for the code within a method if it is to behave as a function.

$$\frac{\begin{array}{l}\{\texttt{State} = s \cup t,\\ \texttt{Local} = \{(\texttt{this}, \texttt{reference to}\ t), (\texttt{arg1}, \texttt{undefined/NULL}), \ldots,\\ (\texttt{argn}, \texttt{undefined/NULL})\}\}\\ \vdash\ M \text{ body code}\\ \{\texttt{State} \supseteq s \cup t\}\{\texttt{EX:State} \supseteq s \cup t, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\}\end{array}}{\vdash M \text{ is a } \texttt{function}}$$

Use of this proof rule in the implementation of the typechecker does not need to prove the implementation (the top of the proof rule) of $M$ when it comes across an invocation within another method that is currently being checked, but instead assumes that the method has been implemented this way if it has been declared as such by the programmer, in the source code. This is not a weakness; if it is not actually implemented correctly, it will be caught out later by the checker. This also applies to the mutator implementation rule.

### Mutator invocation

This rule specifies the behaviour of the code implementing a method if it is to behave as a mutator when called, and also the behaviour of code invoking such a method.

$$\frac{\vdash M \text{ is a } \texttt{mutator}\ , \quad \begin{array}{l}\{P\}\\ \vdash\ \texttt{O}; \texttt{arg1}; \texttt{arg2}; \ldots; \texttt{argn} :\\ \{Q\}\{\texttt{EX:}E, \texttt{BREAK:}F, \texttt{RETURN:}G, \texttt{CONT:}H\}\end{array} \quad , \quad Q\prime \equiv Q[\texttt{State}\prime/\texttt{State}] \wedge \texttt{State} \supseteq (\texttt{State}\prime \setminus\ !(\texttt{eval}(O)))}{\begin{array}{l}\{P\}\\ \vdash\ \texttt{O}.M(\texttt{arg1}, \ldots, \texttt{argn})\\ \{Q\prime\}\{\texttt{EX:}E \vee Q\prime, \texttt{BREAK:}F, \texttt{RETURN:}G, \texttt{CONT:}H\}\end{array}}$$

That is, the sequencing of evaluating $O$ and the arguments, but this time the call itself is not invariant, so that needs composing into the postconditions of the invocation triple. Specifically, $Q$ with knowledge of state reduced further[5], as the object being invoked upon may or may not have been tainted by the mutator, as specified (Section 3.1.5).

The same warnings and demands about subclassing and overriding of methods as for `functions` also apply here for `mutators` with the added freedom that the overriding methods may be `functions` rather than just `mutators` due to behavioural subtyping; remember that `function` *is a* `mutator`(Figure 3.6).

[5]The renaming of `State` to `State`$\prime$ in Q was a logical idiom merely used to discard any assertions $Q$ was making about state, and to use it to construct a new value of `State` from the old one.

### Mutator implementation

This rule specifies how the body code of a method must behave if it is to be accepted as a `mutator`.

$$\frac{\begin{array}{l}\{\texttt{State} = s \cup t,\\ \texttt{Local} = \{(\texttt{this}, \texttt{reference to } t), (\texttt{arg1}, \texttt{undefined/NULL}), \ldots,\\ (\texttt{argn}, \texttt{undefined/NULL})\}\}\\ \vdash\ M \text{ body code}\\ \quad \{\texttt{State} \supseteq s\}\{\texttt{EX:State} \supseteq s, \texttt{BREAK:f}, \texttt{RETURN:f}, \texttt{CONT:f}\}\end{array}}{\vdash M \text{ is a } \texttt{mutator}}$$

### Normal method invocation

Specified here is the behaviour of code invoking a normal (that is, not a `function` or `mutator`) method.

$$\frac{\vdash M \text{ is a normal method}, \quad \begin{array}{l}\{P\}\\ \vdash\ \texttt{O}; \texttt{arg1}; \texttt{arg2}; \ldots; \texttt{argn} :\\ \quad \{Q\}\{\texttt{EX:}E, \texttt{BREAK:}F, \texttt{RETURN:}G, \texttt{CONT:}H\}\end{array} \quad , \quad Q\prime \equiv Q[\texttt{State}\prime/\texttt{State}] \wedge \texttt{State} \supseteq \emptyset}{\begin{array}{l}\{P\}\\ \vdash\ \texttt{O}.M(\texttt{arg1}, \ldots, \texttt{argn})\\ \quad \{Q\prime\}\{\texttt{EX:}E \vee Q\prime, \texttt{BREAK:}F, \texttt{RETURN:}G, \texttt{CONT:}H\}\end{array}}$$

### Sequencing

This rule is concerned with the behaviour of a flat sequence of statements.

$$\frac{\begin{array}{l}\{P\}\\ \vdash\ \texttt{C1}\\ \{Q_1\}\{E_1\}\end{array} \quad , \quad \begin{array}{l}\{Q_1\}\\ \vdash\ \texttt{C2}\\ \{Q_2\}\{E_2\}\end{array} \quad , \ \ldots\ , \quad \begin{array}{l}\{Q_{n-1}\}\\ \vdash\ \texttt{Cn}\\ \{Q_n\}\{E_n\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash\ \texttt{C1}; \texttt{C2}; \ldots; \texttt{Cn}\\ \quad \{Q_n\}\{E_1 \vee E_2 \vee \ldots \vee E_n\}\end{array}}$$

For the preconditions and postconditions, this is simply the (extended) sequencing rule of Floyd-Hoare logic. For the abnormal exit conditions, it is trivial that if any of the statements within the sequence may leave the sequence with exceptional control flow, with the given conditions being true, then exceptional control flow will leave the entire sequence with any one of these being true, hence the disjunctions.

### Local variable definition

$$\begin{array}{l}\{P\}\\ \vdash\ \textit{Type}\ V;\\ \{P[\texttt{Local}\prime/\texttt{Local}] \wedge \texttt{Local} = \texttt{Local}\prime \cup (V, \texttt{undefined/NULL})\}\{\text{f}\}\end{array}$$

When a local variable is declared, the only difference to any asserted condition is that the `Local` relation has an extra entry in it, for the variable and its value (which in Java is `undefined/NULL`).

**Local assignment**

$$\frac{\begin{array}{l}\{P\}\\ \vdash\ \texttt{C}\\ \{Q\}\{E\}\end{array}\ ,\ V \in \texttt{dom(Local)}}{\begin{array}{l}\{P\}\\ \vdash\ V = \texttt{C};\\ \{Q[\texttt{Local}\prime/\texttt{Local}] \wedge \texttt{Local} = \texttt{Local}\prime \cup (V, \texttt{eval(C)})\}\{E\}\end{array}}$$

Assigning the result of an expression to a local variable will obviously be composed of the evaluation of the expression, followed by setting the variable. Hence the same conditions as the evaluation, with the postcondition modified such that any statement asserting the value of `Local` has this value updated to reflect a `Local` with the new value of the variable in it.

**New object**

$\texttt{eval(new } \mathit{Class}\texttt{(C1,...,Cm))} = \texttt{reference to dom(n)}$

$$\frac{\begin{array}{l}\{P[\texttt{State}\prime/\texttt{State}], \texttt{State} = \texttt{State}\prime \cup n\}\\ \vdash\ (\texttt{reference to dom(n)}) \cdot \texttt{<init>(C1,...,Cm)}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash\ \texttt{new } \mathit{Class}\texttt{(C1,...,Cm)}\\ \{Q\}\{E\}\end{array}}$$

This is essentially sequencing the allocation of a new object ($n$) with calling the constructor method on it.

**Throw**

$$\frac{\begin{array}{l}\{P\}\\ \vdash\ \texttt{C}\\ \{Q\}\{\texttt{EX:}E\prime, E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash\ \texttt{throw C}\\ \{\text{f}\}\{\texttt{EX:}Q \vee E\prime, E\}\end{array}}$$

**Remaining rules...**

The rules shown here are just a small subset of all those used in the project. The remaining rules can be found in Appendix A.

### 3.1.7 The proof algorithm

The previous section contained rules that stood on their own, defining truth in the context of this behavioural type system. This section defines how the rules can be used together to determine algorithmically the behaviour of a method with respect to `State`. The design goal of the algorithm; it has to look at the body code of any method that a programmer has declared to be a `function` or `mutator` and attempt to prove the hypothesis (top half) of the appropriate implementation rule(3.1.6) to prove the conclusion that "$M$ is a `function`/`mutator`".

When one has a rule of the form

$$\frac{A_1, A_2, \ldots, A_n}{B}$$

, where $A_1$ to $A_n$ and $B$ are statements in logic, showing that all $A_1$ to $A_n$(your hypotheses) are true is sufficient to show that $B$ (your conclusion) is true. If the hypotheses themselves are not axioms (that is, by definition true within the current system of logic), then they themselves must be conclusions in another rule, and hence your proof must recurse up along an expanding tree of rules, terminating with axioms or other truths at the nodes. Only then can you say that $B$ is true.

This lends itself easily to a structurally recursive algorithm, a procedure that works upon a conclusion, and recursively executes itself upon the hypotheses, with the base case of axioms and truth. The procedure needs to return more information than just truth though; take for example the sequencing rule, in which the hypotheses need to be evaluated with preconditions based on the postconditions of a previous hypothesis. In fact, the procedure does not have to make a decision about truth[6], instead returning postconditions that *make* it true. This can be satisfied by having the reasoning procedure return the postconditions (normal and abnormal exit) of the code/rule it has been given. All the rules in this system allow this; they are all capable of resolving postconditions given a precondition and Java code to reason over.

[6]With the exception of the root of the inference tree, which will be either `function` implementation or `mutator` implementation. Here, truth is resolved outside the scope of the `REASON` procedure by supplying the appropriate preconditions and checking that the postconditions match those demanded by the implementation rule.

The following pseudo-code illustrates this algorithm:

```
procedure REASON(precondition , java_expression)
returns (normal_postcondition,exception_condition,
         return_condition,break_condition,
         continue_condition,eval() value)
{
    rule = the rule containing java_expression as its conclusion

    if (rule is an axiom)
    {
        calculate final_postconditions
        return final_postconditions , eval(java_expression)
    }
    else
    {
        foreach (sub_expression) in (hypotheses of rule)[]
        {
            expression = sub_expression from java_expression
            pre = appropriate precondition for sub_expression
            postconditions[expression] = REASON(pre , expression)
        }
        calculate final_postconditions from postconditions[]
        return final_postconditions , eval(java_expression)
    }
}
```

For convenience, it also returns the result of `eval()` on the Java expression; `eval()` evaluates in the context of a rule, so would not be easily implemented in a flat procedure taking an expression.

## 3.2 Compiler integration

With the theory in place, and an algorithm to make it usable, the next step was to apply it to adding the behavioural types of `function` and `mutator` to the Java compiler[7]

### 3.2.1 Bridging between Javac and the typechecker

As mentioned in Section 2.3.2, the implementation was planned to have distinct separate code modules that just hooked into `javac` at clearly defined points, remaining separate to avoid the difficulty of comprehending it in its entirety. The issue at the

[7]The source code to Sun's Java 1.4 compiler(`javac`) was obtained for this purpose, and modified in the implementation of this behavioural typechecker

first stage of implementation was to find the appropriate locations in `javac` for these hooks. In placing these hooks, the following requirements were relevant:

- The checker/prover needs to be able to get hold of an entire method implementation for all methods declared as `function` or `mutator`
- The programmer needs keywords to declare methods to have these special behavioural types
- The type of the methods needs to persist into the bytecode so that functions and mutators not in the current compilation run (i.e. with source code available to the compiler) can still be invoked usefully
- The rules regarding overriding(Section 3.1.6) of `function`s and `mutator`s need to be enforced
- Failure to prove that a method is a `function` or `mutator`, depending on what the programmer asserted, should bring up an error message in the same format as usual `javac` error messages, explaining the failure with as much detail as is needed for the programmer to correct it

### Passing methods to the prover

The prover requires some handle to the method that gives the behavioural type of it, as well as all the code, based on the requirements of the proof algorithm(Section 3.1.7). The entire syntax tree of a method, as constructed by the parser was found to be entirely suited to that. Just passing that to the proof method was sufficient; it carries data about the behavioural type of the method(see the next two paragraphs for more details), as well as all the source code in a tree-like structure that is ideally suited to the algorithm. As implemented for this project, the algorithm resembles a tree traversal algorithm.

The appropriate hook which calls the checker with the method should obviously be after the syntax tree has been constructed, and ideally after as error checking as possible has been done upon it; the proving algorithm assumes *that the Java it works upon is actually correct and legal*.

### Keywords

The tokenizer was the first point of attack here. Adding the tokens “function” and “mutator” was simply a matter of seeing how the other tokens were defined in the source code, and mimicking that style(see Figure 3.8 for an abstract view of this). The parser was also modified to convert the “function” and “mutator” tokens into the appropriate method flags when creating the root node for a method in the parse tree (it creates a parse tree for each class, with methods as children, each of which encapsulate all data about that method including attributes). See the next paragraph for more information about attributes.

The grammar rules for a Java method are illustrated here:

```
     method  →  attributes type identifier ( arguments ) { body }

 attributes  →  attribute attributes
             |  attribute

  attribute  →  public
             |  private
             |  protected
             |  abstract
             |  static
             |  native

       type  →  Details not necessary here

 identifier  →  Details not necessary here

  arguments  →  Details not necessary here

       body  →  Details not necessary here
```

For adding the required keywords to the language, no fundamental grammar changes were required. Only the list of tokens under `attribute` needed modifying:

```
 attribute  →  function
            |  mutator
            |  public
            |  private
            |  protected
            |  abstract
            |  static
            |  native
```

Figure 3.8: Adding the keyword to the Java method grammar

In the surrounding context of the `public interface Flags`, each attribute number should be $i^2$ for bit $i$. The extra attributes shown here took bits 13 and 14 respectively in the attributes bitfield.

```
//RT
//The method flags for side-effect free functions (functions)
//and modifying only this methods (mutator)
int SEFF=8192;
int MOT=16384;
```

Figure 3.9: Adding to the table of recognised method flags

### Persistence into bytecode

For each method in a classfile, there is a bitfield of flags, that are set to 1 or 0 for each attribute that the method could have. These are used for such things as `public`, `synchronized`, and `final`. It was decided to treat `function` and `mutator` as yet more method attributes, and added them to the table as shown in Figure 3.9.

This took advantage of the fact that `javac` already handles loading and saving all of these flags from classfiles.

### Overriding rules

For the rules to hold, overriding a method in a subclass has to be constrained, as explained in the proof rules:

- A `mutator` or normal method may not override a `function`
- A normal method may not override a `mutator`

This was a matter of adding a facility in `javac` that already enforced similar constraints. Specifically, a stage in the compiler ensures that access controls (regarding {`public`,`private`,`protected`}) do not become more restricted in overriding methods. The `mutator` and `function` attributes of a method are just extra attributes in the same bitfield as the access modifiers, and hence all that was required was for existing code to be copied and modified to respect these.

### Error messages

Like much other functionality that was added to `javac`, error messages were added by taking advantage of the existing system for reporting error messages. This had a twofold advantage, in that it was quicker and easier to program, and also the error messages were guaranteed to be in a standard form that Java developers were already familiar with from using `javac`.

### 3.2.2 The checker

The typechecker (or prover) resembled the proof algorithm(Section 3.1.7) as closely as possible. Additionally, the object orientation used matched the notation in the proof rules fairly closely; `State` and `Local` have corresponding classes, whose instances are composed to form `Condition` classes for the precondition and postconditions. The reasons for the implementation/theory are sketched out in Section 2.3.2. As well as holding the actual values in their instances, they also have methods defined on them for logical operations, such as state tainting, disjunctions and adding elements to the sets.

The following were specified in detail based on the logic/notation, before any code was written for the checker.

#### GlobalState

This is analogous to logical statements asserting the value of `State`. It contains a set of `StateChunks` which are subsets of `State` union-ed together. A small difference is in how the rules taint subsets of `State`. Where some of them transform `State` $= s \cup t$ into `State` $\supseteq s$ for example, the `GlobalState` object fills in the "gap" with an extra subset of `State` that is not necessarily equal to $t$ but with the same domain, a tainted version of $t$, $t\prime$ for example.

`class GlobalState`

| | |
|---|---|
| GlobalState() | This constructor creates an empty GlobalState, meaning `State` $= \emptyset$ |
| get(StateChunk what) | Returns the `GlobalValue` of a subset of this `GlobalState` which has the domain given by the `StateChunk` |
| taint(StateChunk what,Tree where) | Returns a new `GlobalState` identical to this one, except the subset given by the given `StateChunk` is tainted |
| taintAll(Tree where) | Like taint(...) but taints all `StateChunks` |
| disjunct(GlobalState with,Tree where) | Returns a new `GlobalState` that is the disjunction between this and with, that is, the assertion `State` $=$ this $\vee$ `State` $=$ with |
| add() | Returns a new `GlobalState` which has an extra StateChunk added to it. When called upon the GlobalState for `State` $= s$, returns the GlobalState for `State` $= s \cup t$ |
| latest() | Returns the `StateChunk` that was last added to this one |
| toString() | Returns a string representation of what this object says about the value of `State` |

Some of the methods specified here also contain parameters of type `Tree`. This is

a node of a syntax tree within `javac`, and is passed as a parameter to say which bit of the code was being reasoned about when a new `GlobalState` was created. Inspection of this value allows the checker to determine which part of the method caused `function` or `mutator` constraints to be violated by way of `State`/`GlobalState` tainting.

#### StateChunk

This is a part of the domain of `State` as asserted in a `GlobalState`.

`class StateChunk`

| | |
|---|---|
| StateChunk() | This constructor creates a new `StateChunk` |
| toString() | Returns the unique name of this `StateChunk` |

#### GlobalValue

This represents the range of a `StateChunk` in a `GlobalState`, in only as much detail as tainting. That is, a tainted version of this is *not necessarily* equal in value to the original.

`class GlobalValue`

| | |
|---|---|
| GlobalValue() | This creates a `GlobalValue`, not necessarily equal to any other |
| taint(Tree where) | This returns a tainted version of this `GlobalValue` |
| taint_val | This integer records how much this `GlobalValue` has been tainted, 0 indicating that it is the original value |
| tainted_where | This value is of type `Tree` and records which bit of code was being reasoned about when this was last tainted |

#### LocalState

This is the local variables equivalent of `GlobalState`, representing an assertion about the value of `Local`.

`class LocalState`

| | |
|---|---|
| LocalState() | Constructor creating an empty `LocalState`, which asserts an empty `Local` set |
| disjunct(LocalState with) | The `LocalState` equivalent of `GlobalState.disjunct(...)` |
| assertValue(String variable,LocalValue val) | Returns a new `LocalState` identical to this except that the value of the named local variable is set as given |
| assertValue(String variable,Set vals) | Like the above `assertValue(...)`, except it asserts that the variable may be one of several values, as a disjunction; `Local`(variable) $=$ vals$[0] \vee$ `Local`(variables) $=$ vals$[1] \vee \ldots$. |
| getValue(String variable) | Returns a set of possible values this assertion claims for the given local variable |
| toString() | Returns a string representation of what this object claims about the value of `Local` |

**LocalValue**

This class represents values that local variables can take, as in Figure 3.4.

`class LocalValue`

| | |
|---|---|
| LocalValue(StateChunk ref) | Creates a `reference to` ref, referring to the domain of `State` referred to by ref |
| getRef() | Gets the `StateChunk` that this refers to, if it is a `reference to` |
| toString() | Returns a string describing what this value is |
| UNKNOWN | This static member constant represents `Unknown` |
| UNDEF_NULL | This static member constant represents `undefined/NULL` |

### 3.2.3 Non-invasive API tagging

Due to a small amount of slack time after implementing the core of this project, a useful extension was added. So far, use of the entire Java API has had to be ignored when one implements a function, as none of the API calls are themselves functions;

we have seen before that a `function` cannot itself be allowed to invoke a normal method.

Within the system so far, any `function`s compiled to a classfile would be labelled as such in the classfile, and treated as such when linked to in future compiles. It follows that as the Java API is itself a collection of classfiles, these could be modified to assert that certain API methods are `function`s or `mutator`s. The potential existed to write a classfile processor which took in the API classes and outputted them modified, with the appropriate methods tagged as having such special behaviour. This method of tagging the API was dismissed for two reasons; it would require far more time than was remaining, learning how to manipulate classfiles, and also would require tainting an entire Java installation which is not ideal when there is a much less invasive alternative.

The solution that was actually implemented was rather simple. A small configuration file contained a table identifying which API methods behaved as if they were `function`s or `mutator`s. The modified compiler reads this as necessary, and a modification to `javac`s classloader ensures that a loaded method appears to be a `function` or `mutator` to the rest of the compiler.

The file format for this configuration file (known as "annotation-table") is human readable and can contain comments that are ignored by preceding a line with a # symbol. A sample annotation-table file follows:

```
#We want to use System.out.println for debugging messages!
java.io.PrintStream println(java.lang.String) function

#We want to throw exceptions; allow us to construct them
java.lang.IllegalArgumentException <init>(java.lang.String) mutator
```

With this file in place, `function`s can throw `IllegalArgumentException`s without failing due to the compiler believing that the exception constructor taints state (if it were a normal method). The commonly used "System.out.println(...)" can also be used for debugging messages.

There is perhaps a problem of soundness in this scheme, in that the developer who is adding to the annotation table for their own convenience is taken on trust. If they have added to the table claiming that a given method is a `function` but it doesn't actually behave as one, then a `function` will be allowed to invoke it and then not necessarily behave as a `function`. This was considered to be a worthwhile risk when set against the convenience of the annotation-table, that is, it would be fairly difficult to write real usable `function`s if the API couldn't be used.

### 3.2.4 Debug mode

This is not relevant to the "finished product" as such, but was useful for the incremental testing and evaluation. While usually the compiler should keep silent about its internal details when it accepts a `function` or `mutator` with errors, and only give

a short helpful error message, enabling debug mode prints out the pre and postconditions of any method passing through the prover. An example of a simple function "myfunction" that creates an object and modifies one of its fields, then returns a value is shown below:

```
Now reasoning about myfunction...
Precondition:
State = c  U  d
Local(this) = ref(d)   Local(arg2) = Unknown   Local(arg1) = Unknown

Postconditions:
false
EX: State = c  U  e  U  d
Local(this) = ref(d)   Local(arg2) = Unknown   Local(arg1) = Unknown
BRK: false
CONT: false
RET: State = c  U  e'  U  d
Local(this) = ref(d)   Local(arg2) = Unknown   Local(arg1) = Unknown
eval: [Unknown]
```

## 3.3 Use of the system

This section is a brief guide to using `function`s and `mutator`s within side-effect free Java, from the perspective of a programmer.

### 3.3.1 Creating functions and mutators

Any method you wish to have the behavioural type `function` must have the keyword "function" in its declaration, before the name and the return type, in the same place as method attributes like "public" and "abstract". The program will only compile if such methods do not modify any fields in objects and classes existing at method invocation time. If it compiles, then "function" can be taken as a guarantee of such behaviour wherever it is invoked from.

With a `mutator` type method, the usage is very much the same, with the keyword "mutator", except that the programmer is also allowed to modify fields of the object being invoked upon.

### 3.3.2 Error messages

The modified compiler has three new error messages to inform the programmer of why a compile failed due to misuse of the language extension.

- "Overriding method has incompatible behavioural type" - the programmer sees this if a `function` or `mutator` is overridden in a subclass, in a way violating the rules in Section 3.2.1

- "Could not prove that ... was a function: program state tainted" - if the checker cannot prove that the given method is a function, this message is associated with the first line number that causes a violation of the behaviour specified for a `function`; e.g. an assignment to a field in an existing object
- "Could not prove that ... was a mutator: program state tainted" - the same as above, but for `mutator` methods

# Chapter 4

# Evaluation

Evidence that the project works correctly is exhibited here. Some tests here were planned in advance, in the plan of testing(Section 2.3.3), and others were devised as a result of decisions made at implementation time.

## 4.1 Behaviour of the compiler on standard Java code

The original plan of testing demanded that any modifications made to `javac` do not modify its operation upon Java which does not utilise the behavioural typing extension. The modified compiler was used to compile:

- Large parts of the unmodified `javac` source: this virgin code compiled without errors or warnings just as it did when compiled with the normal `javac` installed on my system, and could still compile Java
- One of my Java ticks from the 1A Computer Science course: Tick 5, the polynomial arithmetic program was chosen because it was moderately complex and had a demo method built into it which would output verifiably correct arithmetic if compiled correctly, which it was
- A "Hello World" program: this compiled and did indeed print out "Hello World" to console

The modified `javac` passed this test with no issues whatsoever in evidence.

## 4.2 Testing for intended use

As planned, the modified `javac` was tested in realistic usage; that is, on non-contrived code that had `function` and `mutator` methods used correctly and to good effect.

### 4.2.1 ML style lists

Considering that this project was inspired by pure functional programming, it seems appropriate to test its suitability against the implementation of such a data structure. An "ML style list" is just a collection of singly-linked nodes, each of which contains the data for that node and a reference to the next node, or some variety of null pointer for the node at the end of the list. All operations are functional, not modifying the list being invoked upon, but instead returning modified copies. Such a list was implemented in Java with the behavioural types for the purposes of this test, and can be found in Appendix C.

"MLList.java" compiled with no errors, about the behavioural typing or otherwise. It also behaved functionally, as expected:

```
MLList l=new MLList("One",null);
l=l.addToHead("Two").addToHead("Three").addToHead("Four").
addToHead("Five");
MLList m=l;
System.out.println("l = "+l);
System.out.println("m = "+m);
l=l.addToHead("Six");
System.out.println("Added \"Six\" to l");
System.out.println("l = "+l);
System.out.println("m = "+m);
System.out.println("Calling l.length() = "+l.length());
System.out.println("l = "+l);
System.out.println("m = "+m);
```

The above was a test suite used upon MLList, which produced the following output:

```
l = Five::[Four::[Three::[Two::[One::[null]]]]]
m = Five::[Four::[Three::[Two::[One::[null]]]]]
Added "Six" to l
l = Six::[Five::[Four::[Three::[Two::[One::[null]]]]]]
m = Five::[Four::[Three::[Two::[One::[null]]]]]
Calling l.length() = 6
l = Six::[Five::[Four::[Three::[Two::[One::[null]]]]]]
m = Five::[Four::[Three::[Two::[One::[null]]]]]
```

So, the `addToHead(...)` method was indeed behaving as a function, leaving `m` invariant in its operation on `l` which was aliased to `m`. `length()` also displayed this property.

The compiler is here demonstrated to be working correctly; a method being accepted as a `function` implies that that method behaves like a `function`. This is the desired soundness property of the type system.

### 4.2.2 A modification to ML style lists

A small modification was made to the above list implementation to improve efficiency by a small amount. It was observed that the length method would run in $O(n)$ for length of list $n$, and this situation could be improved by having this method cache its return value; recall that with functional lists, the length of a particular list will never actually change.

The solution was to override MLList, and the code is shown here:

```
/**
 * This is an optimised version of MLList, specifically with a
 * cached length result to speed up calls to length.
 */
public class MLListOpt extends MLList
{
    public mutator MLListOpt(Object head,MLList tail)
    {
        super(head,tail);
    }

    public function int length()
    {
        if(cachedLength==-1)
        {
            cachedLength=super.length();
        }
        return cachedLength;
     }
    private int cachedLength=-1;
}
```

This method would obviously work, however it does not compile:

```
MLListOpt.java:16: Could not prove that length() was a function:
Program state tainted
                        cachedLength=super.length();
                        ^
1 error
```

Again, as in the testing upon the Polynomial program, this is an issue where the compiler is being too harsh. While state is indeed tainted and `length()` fails as a `function`, it is fairly apparent that this modification is effectively invisible to the outside world. Its behaviour is indistinguishable from the `length()` method being overridden, so arguably the method should in fact be a `function`.

### 4.2.3 Polynomial arithmetic

For this purpose Java Tick 5 from 1A, the polynomial arithmetic program, was appropriately modified to have its various methods claim a behavioural type. The full modified source is shown in Appendix C. In this case, the compilation failed:

```
Poly.java:38: Could not prove that multInt(long) was a mutator:
Program state tainted
cof[i]*=a;
   ^
Poly.java:193: Could not prove that cancelDown() was a mutator:
Program state tainted
cof[i]/=cgcd;
   ^
2 errors
```

Here, the compiler is pedantically correct; the object pointed to by the member field `cof`, that is, an array, is being modified, and although `cof` is part of the object and would be allowed to be modified to point to something else, the array being pointed to is not considered part of the object. However, the array pointed to is private within the object, and is never shared with any other; it is effectively part of the object. The spirit of the `mutator` rules is being obeyed here, just not the formalities of it.

On the other hand, the checker passed the remainder of the `functions` and `mutators` in `Poly.java`. These methods were confirmed to indeed behave as they were declared to, by inspection, hence this test was a partial success.

## 4.3 Rule oriented testing

This kind of testing was not planned, because it arose as a result of implementation decisions. That is, the use of individual, independent rules. While testing cannot hope to cover every realistic programming example utilising `functions` to verify correctness, the implementation of the individual rules can be verified for correctness. If one is to assume that the use of rules jointly within the checker is correct, then verifying that the rules have been implemented correctly effectively verifies that the entire checker is correct.

For this purpose, debugging mode was enabled in the modified `javac`, and small snippets of code chosen to typify the use of each rule were fed through the compiler. The debugging output enables one to verify that the rule has been implemented correctly, as the outputted pre and postconditions should match the rule. Many of these kinds of tests were made, both during and after implementation, and *none were discovered to be incorrect*. A reasonable subset of these tests are displayed in Appendix C.

## 4.4 Summary

To summarise this evaluation, the system worked without noticeable error, for a rigid interpretation of the behavioural type definitions. The type system as implemented was *sound*, in that all methods judged to be `mutators` or `functions` did in fact behave as such according to the definitions. When one takes a non-strict interpretation of the definitions, the system is far from perfect. Taking the failure to compile of the modified polynomial arithmetic program, it is apparent that the failed methods do effectively behave as `mutators`, while the compiler does not acknowledge this. Similarly with the optimised ML style list, the compiler is being too harsh on a method which is totally indistinguishable from an actual `function` which has the same input/output behaviour.

# Chapter 5

# Conclusion

This project has succeeded, in that it allows a programmer to denote a method as having a particular behavioural type, and the modified compiler will accept this if the method does behave as specified in most cases (and in all cases, only if the method behaves as specified). As discussed in the introduction, the sort of guarantee a `function` provides can be immensely useful for certain kinds of project, and this system provides that sort of guarantee with little room for doubt; it has been shown to be *sound* to a reasonable degree of confidence.

The issues highlighted in the evaluation do not necessarily illustrate a mistake in the rules or in their implementation, but in the definition. While the failed `function` in the optimised ML style list behaves identically to a function, it is still not accepted, because at the core of it the definition of a `function` as used here is based upon data as opposed to actual behaviour. The same criticism can also be made with regard to the failed `mutator` methods in the polynomial arithmetic code; the behaviour is effectively that of a `mutator`.

With the benefit of hindsight with respect to these issues, I would have chosen different definitions for the behavioural types that were genuinely based upon behaviour and not just data. Were I to undertake future work aimed at improving the project in this way, I/O behaviour of methods could have been reasoned about, and a `function` could have denoted a method with invariant I/O behaviour. One could reason about method return values as a function of program state, and devise or use a different variety of theorem prover that showed the return value as being invariant where argument values were fixed.

The issue raised by the polynomial arithmetic program was slightly more complex; to be judged as a `mutator`, an improved checker would have to be convinced that the array being modified was effectively part of the object. Informally, this would be having no other object having a direct reference to this array; there is already ongoing research into the concept of data "sharing" and what is considered to be "ownership", so this is no trivial extension.

In summary, this project has been a success, but there is definitely room for improvement if we wish programmers to be as unconstrained as possible while still working within the bounds of the behavioural type being used.

# Bibliography


[1] Alan Lawrence. Addition of Generics to Java

[2] David Flanagan. Java in a Nutshell, 3rd Edition

[3] Mike Gordon, Specification and Verification I course notes, 2002-2003

[4] Al Stevens and Clayton Walnum. Standard C++ Bible, 2000

[5] Tim Lindholm and Frank Yellin. The Java™ Virtual Machine Specification, Second Edition

[6] Dr Alan Mycroft. Course Notes for Optimising Compilers course, 2003

[7] Bertrand Meyer/ISE. An Invitation to Eiffel, 1985-1996.

[8] Michael Norrish. C formalised in HOL, December 1998

# Appendix A

# More proof rules

## A.1 A Block

$$\frac{\begin{array}{ll} & \{P\} \\ \vdash & \texttt{C} \\ & \{Q\}\{E\} \end{array}}{\begin{array}{ll} & \{P, d = \texttt{dom}(\texttt{Local})\} \\ \vdash & \{\ \texttt{C}\ \} \\ & \{Q[\texttt{Local}\prime/\texttt{Local}], \texttt{Local} \subseteq \texttt{Local}\prime | \texttt{dom}(\texttt{Local}) = d\}\{E[\texttt{Local}\prime/\texttt{Local}], \\ & \texttt{Local} \subseteq \texttt{Local}\prime | \texttt{dom}(\texttt{Local}) = d\} \end{array}}$$

This rule enforces the nicety of local variable scope. When any statement is enclosed within a scoping block, its pre and postconditions still hold except that the domain of `Local` in any postcondition (and exception condition) statement is clipped to remove any local variables declared inside the block from `Local`.

## A.2 New array

$$\frac{\begin{array}{ll} & \{P\} \\ \vdash & \texttt{C} \\ & \{Q\}\{E\} \end{array}}{\begin{array}{ll} & \{P\} \\ \vdash & \texttt{new}\ \mathit{Class}[\texttt{C}] \\ & \{Q[\texttt{State}\prime/\texttt{State}], \texttt{State} = \texttt{State}\prime \cup n\}\{E\} \end{array}}$$

The index `C` is first evaluated, followed by modifying `State`, reflecting the creation of the array. Creating a new array adds to `State`. Specifically, $n$ is the extra component of `State` created, and hence a reference to this is the result of evaluation. Content of $n$ is left as unknown but unique in `State`.

## A.3 Return

$$\frac{\begin{array}{l}\{P\}\\ \vdash \texttt{C}\\ \{Q\}\{\texttt{RETURN}:E\prime, E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{return C}\\ \{\text{f}\}\{\texttt{RETURN}:E\prime \vee Q, E\}\end{array}}$$

The reasoning behind this is identical to that behind the Throw rule(3.1.6), with `return` and `RETURN:` in place of `throw` and `EX:`.

## A.4 Break

$$\begin{array}{l}\{P\}\\ \vdash \texttt{break}\\ \{\text{f}\}\{\texttt{BREAK}:P\}\end{array}$$

As `break` evaluates nothing, just alters control flow, any precondition will simply provide the condition that control flow leaves this statement with.

## A.5 Continue

$$\begin{array}{l}\{P\}\\ \vdash \texttt{continue}\\ \{\text{f}\}\{\texttt{CONT}:P\}\end{array}$$

Similar to `break` (A.4).

## A.6 Switch/Case

$$\frac{\begin{array}{l}\{Q \vee R_{n-1}\}\\ \vdash \texttt{Cn}\\ \{R_n\}\{\texttt{BREAK}:B_n, E_n\}\end{array} \quad , \quad \begin{array}{l}\{Q \vee R_1\}\\ \vdash \texttt{C2}\\ \{R_2\}\{\texttt{BREAK}:B_2, E_2\}\end{array} \quad , \quad \begin{array}{l}\{Q\}\\ \vdash \texttt{C1}\\ \{R_1\}\{\texttt{BREAK}:B_1, E_1\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{switch (A) \{ case } 1\texttt{: C1} \ldots \texttt{case } n\texttt{: Cn \}}\\ \{R_n \vee B_1 \vee \ldots \vee B_n\}\{E \vee E_1 \vee \ldots \vee E_n\}\end{array}}$$

$$,$$

$$\begin{array}{l}\{P\}\\ \vdash \texttt{A}\\ \{Q\}\{E\}\end{array}$$

Like sequencing, a switch statement could have the exception conditions of any of its component statements/expressions when control flow leaves exceptionally(except for `break` which is handled within for the cases statements), hence the disjunction in the exception condition. When control flow leaves normally, the condition is as it is because control flow could leave by any one of the case statements invoking break, hence the disjunction of all their `BREAK:s`. Control flow could also exit by the final case statement falling through without a `break` , so the postcondition for that statement is added to the disjunction. Moving on to the conditions demanded of the case statements; the conditions are such that `A` can sequence with any of the case statements after it ($Q$ implies all their pre-conditions), as control flow can move to any of them after evaluation of the switch expression. Also, each statement must be able to sequence with its immediate successor, and the conditions have been setup to allow this.

## A.7 Conditional

$$\frac{\begin{array}{c}\{Q\}\\ \vdash \texttt{C}\\ \{R\}\{E_c\}\end{array}\quad,\quad\begin{array}{c}\{Q\}\\ \vdash \texttt{B}\\ \{S\}\{E_b\}\end{array}\quad,\quad\begin{array}{c}\{P\}\\ \vdash \texttt{A}\\ \{Q\}\{E_a\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{A ? B : C}\\ \{R \vee S\}\{E_a \vee E_b \vee Ec\}\end{array}}$$

Normal control flow can be sequenced, `AB` or `AC`, depending on the evaluation of `A`. The conditions demanded allow this, and the split nature of possible control flows give the disjuncted postcondition. Any exceptional control flow generated out of `A`,`B` or `C` will pass straight out of this expression without being affected, giving the exception condition you see above.

## A.8 If-Then-Else

$$\frac{\begin{array}{c}\{Q\}\\ \vdash \texttt{C}\\ \{R\}\{E_c\}\end{array}\quad,\quad\begin{array}{c}\{Q\}\\ \vdash \texttt{B}\\ \{S\}\{E_b\}\end{array}\quad,\quad\begin{array}{c}\{P\}\\ \vdash \texttt{A}\\ \{Q\}\{E_a\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{if(A)then B else C}\\ \{R \vee S\}\{E_a \vee E_b \vee Ec\}\end{array}}$$

See the Conditional rule(A.7), to which this is equivalent.

## A.9 While

$$\dfrac{\begin{array}{l}\{S\}\\ \vdash \mathsf{B}\\ \{T\}\{F\prime, \mathsf{CONT}:G\prime, \mathsf{BREAK}:H\prime\}\end{array}\quad\begin{array}{l}\{T \vee G\prime\}\\ \vdash \mathsf{A}\\ \{U\}\{E\prime\prime\}\end{array}\quad\begin{array}{l}\{Q\}\\ \vdash \mathsf{B}\\ \{R\}\{F, \mathsf{CONT}:G, \mathsf{BREAK}:H\}\end{array}\quad\begin{array}{l}\{R \vee G\}\\ \vdash \mathsf{A}\\ \{S\}\{E\prime\}\end{array}\quad\begin{array}{l}\{P\}\\ \vdash \mathsf{A}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{while}(\mathsf{A})\ \mathsf{B}\\ \{Q \vee S \vee U \vee H \vee H\prime\}\{E \vee E\prime \vee E\prime\prime \vee F \vee F\prime\}\end{array}}$$

This is equivalent to the sequencing ABABA, which does reflect the behaviour of a `while`() loop. Normal exit can occur after any A (condition evaluating to false), or from a `break` inside the body, B. `continue` in the body moves control flow to the next evaluation of the condition.

This only considers two whole iterations at most because no more information regarding `State` and `Local` can be gained by unrolling the loop further with the limited ruleset used here.

## A.10 Do-While

$$\frac{\begin{array}{l}\quad\{R\}\\ \vdash\ \mathtt{A}\\ \quad\{S\}\{F\prime, \mathtt{CONT}:G\prime, \mathtt{BREAK}:H\prime\}\end{array}\qquad\begin{array}{l}\quad\{S \vee G\prime\}\\ \vdash\ \mathtt{B}\\ \quad\{T\}\{E\prime\}\end{array}\qquad\begin{array}{l}\quad\{P\}\\ \vdash\ \mathtt{A}\\ \quad\{Q\}\{F, \mathtt{CONT}:G, \mathtt{BREAK}:H\}\end{array}\qquad\begin{array}{l}\quad\{Q \vee G\}\\ \vdash\ \mathtt{B}\\ \quad\{R\}\{E\}\end{array}}{\begin{array}{l}\quad\{P\}\\ \vdash\ \mathtt{do\ A\ while(B)}\\ \quad\{R \vee T \vee H \vee H\prime\}\{E \vee E\prime \vee F \vee F\prime\}\end{array}}$$

Similar to `while()` (A.9), the conditions here are chosen to allow circular sequencing of AB as ABAB, but in this case terminating on B(when it evaluates as false).

## A.11 For

Rewrite

$$\mathtt{for(A;B;C)\ D}$$

as

$$\{\mathtt{A};\mathtt{while(B)}\ \{\mathtt{D};\mathtt{C}\}\}$$

and reason about it, as the two are entirely equivalent.

## A.12 Global assignment

$$\frac{\begin{array}{l}\quad\{R\}\{F\}\\ \vdash\ \mathtt{C}\\ \quad\{Q\}\{E\}\end{array}\qquad\begin{array}{l}\quad\{Q\}\\ \vdash\ \mathtt{D}\\ \quad\{R\}\{F\}\end{array}}{\begin{array}{l}\quad\{P\}\\ \vdash\ \mathtt{C} \cdot \mathit{Field} = \mathtt{D}\\ \quad\{R[\mathtt{State}\prime/\mathtt{State}], \mathtt{State} = \mathtt{State}\prime[n\prime/\mathtt{!(eval(C))}] | \mathtt{dom}(n\prime) = \mathtt{dom}(\mathtt{!(eval(C))})\}\{E \vee F\}\end{array}}$$

This is, as usual, derived from evaluating C followed by evaluation of D and sequencing these. Also, after the assignment, the object being assigned to is totally "tainted" there is no need to be more specific for the uses of this system, and the whole object is just replaced with $n\prime$, an anonymous variable with nothing said about its values.

## A.13 Try-Catch-Finally

$$\frac{\begin{array}{l}\{E \vee E\prime \vee R \vee F \vee Q\} \\ \vdash \texttt{C} \\ \{S\}\{G\} \\ \{E[\texttt{Local}\prime/\texttt{Local}][\texttt{State}\prime/\texttt{State}], \\ \texttt{Local} = \texttt{Local}\prime \cup (e, \texttt{reference to } n), \texttt{State} = \texttt{State}\prime \cup n\} \\ \vdash \texttt{B} \\ \{R\}\{F\} \\ \{P\} \\ \vdash \texttt{A} \\ \{Q\}\{\texttt{EX}: E, E\prime\}\end{array}}{\begin{array}{l}\{P\} \\ \vdash \texttt{try A catch}(\mathit{ExceptionType}\ e)\ \texttt{B finally C} \\ \{S\}\{G \vee S\}\end{array}}$$

This rule does not go into the detail of matching up the type of the thrown exception with the catch clause, instead allowing that it is possible that a thrown exception in the body code is caught. Again, this is all just a matter of sequencing; the catch block code B can only enter from an exception out of A, hence that is how it is sequenced (using $E$), except that a local variable $e$ is added, being the handle to the thrown exception (which is an addition to State).

The whole Try-Catch-Finally block, when exiting normally, will always exit out of the finally block, hence the postcondition is the postcondition of the finally block, $S$. The finally block can enter from one of A exiting normally, A exiting with an exception or other exceptional control flow, from any exceptional control flow out of the catchblock, from normal termination of the catchblock, hence the massive disjunction of conditions in the precondition ($E \vee E\prime \vee R \vee F \vee Q$).

Control flow can leave the entire block exceptionally by either an exception being thrown out of the finally block (condition $G$), or the finally block terminating normally $S$ and throwing the exception/break/etc. it has "hung onto"

## A.14 Evaluation of local variables

$$\frac{V \in \texttt{dom}(\texttt{Local})}{\begin{array}{l}\{P\} \\ \vdash V \\ \{P\}\{\text{f}\}\end{array}}$$

## A.15 Evaluation of object fields

$$\frac{\begin{array}{l}\{P\}\\ \vdash \texttt{C}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{C} \cdot \mathit{Field}\\ \{Q\}\{E\}\end{array}}$$

The evaluation will always be `Unknown`, as there is little point in anything else with a prover of such limited goals.

## A.16 Evaluation of static fields

$$\begin{array}{l}\{P\}\\ \vdash \mathit{Class} \cdot \mathit{Field}\\ \{P\}\{\text{f}\}\end{array}$$

## A.17 Evaluation of literals

$$\begin{array}{l}\{P\}\\ \vdash \text{“String”/}\texttt{Number}\\ \{P\}\{\text{f}\}\end{array}$$

## A.18 Evaluation of unary operators

(excepting prefix/postfix incrementers which have side effects) Treat incrementers as the equivalent assignment operations.

$$\frac{\begin{array}{l}\{P\}\\ \vdash \texttt{C}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{Op C}\\ \{Q\}\{E\}\end{array}}$$

## A.19 Binary operators

(excepting those with conditional rhs evaluation)

$$\frac{\begin{array}{l}\{Q\}\\ \vdash \texttt{D}\\ \{R\}\{F\}\end{array} \quad , \quad \begin{array}{l}\{P\}\\ \vdash \texttt{C}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{C Op D}\\ \{R\}\{E \vee F\}\end{array}}$$

This is sequencing `C` and `D` as Java will evaluate them in left to right order.

## A.20 Bracketed expressions

$$\frac{\begin{array}{l}\{P\}\\ \vdash \texttt{C}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash (\texttt{C})\\ \{Q\}\{E\}\end{array}}$$

## A.21 Logical operators

$$\frac{\begin{array}{l}\{Q\}\\ \vdash \texttt{B}\\ \{R\}\{F\}\end{array} \quad , \quad \begin{array}{l}\{P\}\\ \vdash \texttt{A}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{A} \,||\, \texttt{B}\\ \{Q \vee R\}\{E \vee F\}\end{array}}$$

$$\frac{\begin{array}{l}\{Q\}\\ \vdash \texttt{B}\\ \{R\}\{F\}\end{array} \quad , \quad \begin{array}{l}\{P\}\\ \vdash \texttt{A}\\ \{Q\}\{E\}\end{array}}{\begin{array}{l}\{P\}\\ \vdash \texttt{A} \,\&\&\, \texttt{B}\\ \{Q \vee R\}\{E \vee F\}\end{array}}$$

# Appendix B

# Sample code

This appendix exhibits a few representative samples of code that was written in the implementation of this project.

`LocalValue.java`

```
package sefprover;

/**
 * A value that a local variable in a Local can take.
 */
class LocalValue
{
    /**
     * Creates a Value that represents a reference to a subset of State (a Java object).
     */                                                                          10
    public LocalValue(StateChunk ref)                                            LocalValu
    {
        this.ref=ref;
    }

    /**
     * Get the subset of State that this value is referring to, if it is a
     * reference.
     */
    public StateChunk getRef()                                                   20 getRef
    {
        if(ref!=null)
        {
            return ref;
        }
        else
        {
            throw new IllegalStateException
                    ("Attempted to call getRef() on a Value that isn't a reference");
        }                                                                        30
    }

    /**
```

```
 * Returns a human readable description of this LocalValue.
 */
public String toString()                                                    toString
{
    if(this==UNKNOWN)
    {
        return "Unknown";                                                   40
    }
    else if(this==UNDEF_NULL)
    {
        return "UNDEF/NULL";
    }
    else
    {
        return "ref("+ref+")";
    }
}                                                                           50

/**
 * A LocalValue that is unknown by the prover.
 */
public static final LocalValue UNKNOWN=new LocalValue();

/**
 * A LocalValue that is know to be null or undefined by the prover.
 */
public static final LocalValue UNDEF_NULL=new LocalValue();                 60

/**
 * Honour the equality relation, for using this in Sets.
 */
public boolean equals(Object other)                                         equals
{
    if(other instanceof LocalValue)
    {
        LocalValue other_l=(LocalValue)other;
        if(this==other_l)                                                   70
        {
            return true;
        }
        else if(this.ref==other_l.ref && this.ref!=null)
        {
            return true;
        }
        else
        {
            return false;                                                   80
        }
    }
    else
    {
        return false;
    }
```

```
    }

    /**
     * For use in HashSets.                                          90
     */
    public int hashCode()                                                hashCode
    {
        if(ref!=null)
        {
            return ref.hashCode();
        }
        else
        {
            return super.hashCode();                                  100
        }
    }

    /**
     * Purely for internal use, in creating the 'enums' of UNKNOWN and
     * UNDEF_NULL.
     */
    private LocalValue()                                                 LocalValu
    {
    }                                                                 110

    /**
     * The StateChunk this Value refers to (if it is a reference)
     */
    private StateChunk ref;
}
```

## Condition.java

```
package sefprover;

import com.sun.tools.javac.v8.tree.Tree;

/**
 * This is an assertion about program state, specifically about the values in
 * State and Local.
 */
class Condition
{                                                                     10
    /**
     * Creates a Condition with values of State and Local asserted.
     */
    public Condition(GlobalState state,LocalState local)                 Condition
    {
        this.state=state;
        this.local=local;
    }

    /**                                                               20
```

```
 * Creates a new Condition that's an ORing of 'this' and 'with'. Must be
 * given the code causing the disjunction.
 */
public Condition disjunct(Condition with,Tree where)                    disjunct
{
    if(this==Condition.FALSE)
    {
        return with;
    }
    else if(with==Condition.FALSE)                                          30
    {
        return this;
    }
    else
    {
        return new Condition(this.state.disjunct(with.state,where),this.local.disjunct(with.local));
    }
}

/**                                                                         40
 * Returns a readable representation of this assertion about state.
 */
public String toString()                                                toString
{
    if(this==Condition.FALSE)
    {
        return "false";
    }
    else
    {                                                                       50
        return state.toString()+'\n'+local.toString();
    }
}

/**
 * The possible values of program state asserted by this condition
 */
public final GlobalState state;

/**                                                                         60
 * The state of the local variables, as associated by this condition
 */
public final LocalState local;

/**
 * A condition logically equivalent to 'false' as opposed to an assertion
 * about values of global and local state.
 */
public static final Condition FALSE=new Condition();
                                                                            70
/**
 * Constructor to support singleton state of FALSE.
 */
```

```
    private Condition()                                                    Condition
    {
        state=null;
        local=null;
    }
}
```

# Appendix C

# Testing

## C.1 Modified Poly.java

```
class Poly //Class representing an arbitrary sized polynomial
{
    //Name of the variable, traditionally 'x'
    private String name;
    //Co-efficients for each polynomial, the degree represented by the array index
    private long[] cof;
    //Denominator of the entire polynomial; allows fractions
    private long den;
    //Debug printing mode?
    private boolean debug;                                                    10

    //Constructs a new polynomial with the given data
    public mutator Poly(String name,long[] cof,long den,boolean debug)        Poly
    {
        subPoly(name,cof,den,debug);
    }

    //Alternative for lazy people who don't want fractions or debugging
    public mutator Poly(String name,long[] cof)                               Poly
    {                                                                         20
        subPoly(name,cof,1,false);
    }

    public mutator void subPoly(String name,long[] cof,long den,boolean debug)   subPoly
    {
        this.name=name;
        this.cof=cof;
        this.den=den;
        this.debug=debug;
        cancelDown();                                                         30
    }

    //Multiplies the entire polynomial by an integer
    public mutator void multInt(long a)                                       multInt
```

```
{
    for(int i=0; i<cof.length; i++)
    {
            cof(i)*=a;
    }
    cancelDown();                                                    40
}

//Divides the entire polynomial by an integer
public mutator void divInt(long a)                                   divInt
{
    den*=a;
    cancelDown();
}

//Multiplies the entire polynomial by the variable involved        50
public mutator void multVar()                                        multVar
{
    long() cof2=new long(cof.length+1);
    cof2(0)=0;
    for(int i=0; i<cof.length; i++)
    {
        cof2(i+1)=cof(i);
    }
    cof=cof2;
}                                                                    60

//Set the debug printing mode
public void setDebug(boolean debug)                                  setDebug
{
    this.debug=debug;
    cancelDown();
}

//Add two polynomials
public static function Poly addPolys(Poly p1,Poly p2)            70 addPol
{
    return addsubPolys(p1,p2,false);
}

//Subtract two polynomials
public static function Poly subPolys(Poly p1,Poly p2)                subPolys
{
    return addsubPolys(p1,p2,true);
}
                                                                     80
//Add/subtract two polynomials
private static function Poly addsubPolys(Poly p1,Poly p2,boolean sub)  addsubPo
{
     long() rcof; //Co-efficients of result
     long rden; //Denominator of result

     if(p1.cof.length>p2.cof.length)
```

```
        {
            rcof=new long(p1.cof.length);
        }
        else
        {
            rcof=new long(p2.cof.length);
        }
        rden=p1.den*p2.den;
        for(int i=0; i<rcof.length; i++)
        {
            if(i<p1.cof.length)
            {
                rcof(i)+=p1.cof(i)*p2.den;
            }
            if(i<p2.cof.length)
            {
                if(sub)
                {
                    rcof(i)-=p2.cof(i)*p1.den;
                }
                else
                {
                    rcof(i)+=p2.cof(i)*p1.den;
                }
            }
        }
        return (new Poly(p1.name,rcof,rden,false));
    }

    //Multiplies together p1 and p2, returning the result
    public static function Poly multPolys(Poly p1,Poly p2)                    multPolys
    {
        long() rcof; //Co-efficients of result
        long rden; //Denominator of result

        rcof=new long(p1.cof.length+p2.cof.length);
        rden=p1.den*p2.den;
        for(int i=0; i<p1.cof.length; i++)
        {
            for(int j=0; j<p2.cof.length; j++)
            {
                rcof(i+j)+=p1.cof(i)*p2.cof(j);
            }
        }
        return (new Poly(p1.name,rcof,rden,false));
    }

    //Differentiates the polynomial
    public mutator void diffPoly()                                            diffPoly
    {
        long() rcof=new long(cof.length-1); //Co-efficients of result

        for(int i=1; i<cof.length; i++)
```

```
        {
            rcof(i-1)=i*cof(i);
        }
        cof=rcof;
    }


    //Simplifies the fraction as much as possible
    public mutator void cancelDown()                                    cancelDo
    {                                                                   150
        long divs()=new long(cof.length+1); //Things to try dividing through
        divs(0)=den;
        long cgcd=divs(0); //Current gcd
        long r; //Remainder

        if(debug | den==0)
        {
            return;
        }
        for(int i=0; i<cof.length; i++)                                 160
        {
            if(cof(i)!=0)
            {
                divs(i+1)=cof(i);
            }
            else
            {
                divs(i+1)=divs(i);
            }
        }                                                               170
        for(int i=1; i<divs.length; i++)
        {
            long a=cgcd;
            long b=divs(i);
            do
            {
                r=a%b;
                if(r!=0)
                {
                    a=b;                                                180
                    b=r;
                }
            }while(r!=0);
            cgcd=b;
        }
        if(cgcd<0) //Having a negative denominator is bad form; this sorts it.
        {
            cgcd*=-1;
        }
        den/=cgcd;                                                      190
        for(int i=0; i<cof.length; i++)
        {
            cof(i)/=cgcd;
```

```
    }
}

//Prints out the polynomial
public String toString()                                                    toString
{
    boolean done=false; //Done one already? (used for the purposes of putting in a 'plus')   200
    String ret=""; //String to be constructed and returned
    String cofs; //Just a scratchpad

    cancelDown();
    for(int i=0; i<cof.length; i++)
    {
        if(cof(i)!=0 && !debug)
        {
            if(done)
            {                                                                                  210
                if(cof(i)>1)
                {
                    ret=ret+" + ";
                }
                else
                {
                    ret=ret+" - ";
                }
            }
            if(cof(i)==1)                                                                      220
            {
                cofs="";
            }
            else
            {
                cofs=Math.abs(cof(i))+"*";
            }
            if(i>1)
            {
                ret=ret+cofs+name+"^"+i;                                                       230
            }
            if(i==1)
            {
                ret=ret+cofs+name;
            }
            if(i==0)
            {
                ret=ret+cof(i);
            }
            done=true;                                                                         240
        }
        if(debug)
        {
            ret=ret+cof(i)+"*"+name+"^"+i+" ";
            done=true;
        }
```

```
        }
        if(!(den==1 && !debug))
        {
            ret="("+ret+")/"+den;
        }
        return ret;
    }
}
```

## C.2 MLList.java

```
/**
 * This implements an ML style list, utilising the language extensions of
 * Side-effect free Java
 */
public class MLList
{
    /**
     * Creates a list from an item to go on the head, and a list to go on the
     * tail.
     */
    public mutator MLList(Object head,MLList tail)
    {
        System.out.println("Debug: created MLList");
        if(head==null && tail!=null)
        {
            throw new IllegalArgumentException("Trying to create empty MLList with a tail");
        }
        if(head!=null && tail==null)
        {
            throw new IllegalArgumentException("Trying to create an MLList with a null tail");
        }
        listHead=head;
        listTail=tail;
    }

    /**
     * Returns whether or not this list is empty.
     */
    public function boolean isEmpty()
    {
        if(listHead==null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

```
                                                                    40
/**
 * Returns the length of this list
 */
public function int length()                                        length
{
    if(this.isEmpty())
    {
        return 0;
    }
    else                                                            50
    {
        return 1+listTail.length();
    }
}

/**
 * Converts this list to an array.
 */
public function Object() toArray()                                  toArray
{                                                                   60
    Object() out=new Object(this.length());
    MLList l=this;
    for(int i=0;i<out.length;i++)
    {
        out(i)=l.listHead;
        l=l.listTail;
    }

    return out;
}                                                                   70

/**
 * Returns a list generated from the given array.
 */
public static function MLList fromArray(Object() from)              fromArray
{
    return fromArray(from,0);
}

private static function MLList fromArray(Object() from,int startPos)  80 fromArr
{
    if(startPos==from.length)
    {
        return new MLList(null,null);
    }
    else
    {
        return new MLList(from(startPos),fromArray(from,startPos+1));
    }
}                                                                   90

public String toString()                                            toString
```

```
    {
        return listHead+"::["+listTail+"]";
    }

    /**
     * The head of the list. Null if this is an empty list.
     */
    public final Object listHead;

    /**
     * The tail of the list
     */
    public final MLList listTail;
}
```

## C.3 Testing local variable rules

```
function public void testfunction(int x)
{
    int a=x;
}
```

Compiled succesfully with debugging output:

```
Now reasoning about testfunction...
Precondition:
State = a  U  b
Local(this) = ref(b)   Local(x) = Unknown

Postconditions:
State = a  U  b
Local(this) = ref(b)   Local(x) = Unknown   Local(a) = Unknown
EX: false
BRK: false
CONT: false
RET: false
eval: [Unknown]
```

Behaved as expected, tainting no `State` and adding an entry to `Local`.

## C.4 Breaching function constraints

```
public class AClass
{
    public function void violator(double x)
    {
        field=(int)x;
        field=field*2;
        field=field+2;
    }

    private int field=0;
}
```

Fails to compile with debugging output:

```
Now reasoning about violator...
Precondition:
State = a  U  b
Local(this) = ref(b)   Local(x) = Unknown

Postconditions:
State = a  U  b'''
Local(this) = ref(b)   Local(x) = Unknown
EX: false
BRK: false
CONT: false
RET: false
eval: [Unknown]
```

And compiler output:

```
AClass.java:5: Could not prove that violator(double) was a
function: Program state tainted
field=(int)x;
^
1 error
```

This behaviour is correct; setting a field within an object should indeed taint the subset of `State` point to by `this`. As the `function` constraints don't allow that part of `State` to be tainted, it fails correctly, pointing out the exact line where the tainting occurred for the first time.

## C.5 Choices with `if()then else`

```
function public void testfunction(int x)
{
    Object a;
    if(x>0)
    {
        a=null;
    }
    else
    {
        a=this;
    }
}
```

Compiled succesfully with debugging output:

```
Now reasoning about testfunction...
Precondition:
State = a  U  b
Local(this) = ref(b)   Local(x) = Unknown

Postconditions:
State = a  U  b
Local(this) = ref(b)   Local(x) = Unknown
Local(a) = ref(b) \/ UNDEF/NULL
EX: false
BRK: false
CONT: false
RET: false
eval: [Unknown]
```

As specified, an if structure delivers an OR-ing of its two branches; either `a` set to `null` or set to point to `this`.

## C.6 Passing mutators

```
public class Incrementor
{
    mutator public void plusMe()
    {
        this.val++;
    }

    private int val;
}
```

Compiled succesfully with debugging output:

```
Now reasoning about plusMe...
Precondition:
State = a  U  b
Local(this) = ref(b)

Postconditions:
State = a  U  b‘
Local(this) = ref(b)
EX: false
BRK: false
CONT: false
RET: false
eval: [Unknown]
```

This output shows that `this` is modified as one would expect, and the compiler accepting it as a `mutator` is correct; tainting of `this` is acceptable in that case.

## C.7 Tricks of control flow

```
public class Incrementor
{
    mutator public Incrementor()
    {
        val=0;
    }

    function public Incrementor plusOne(int x)
    {
        Incrementor target=this;
        Incrementor brandNew=new Incrementor();
        while(x>0)
        {
            if(x>10)
                break;
            target=brandNew;
        }
        target.val++;
        return target;
    }

    public int val;
}
```

Fails to compile with debugging output:

```
Now reasoning about plusOne...
Precondition:
State = c  U  d
Local(this) = ref(d)   Local(x) = Unknown

Postconditions:
false
EX: State = c  U  d  U  e‘
Local(this) = ref(d)   Local(x) = Unknown
Local(target) = ref(d)
BRK: false
CONT: false
RET: State = c  U  d‘  U  e‘‘
Local(this) = ref(d)   Local(brandNew) = ref(e)
Local(x) = Unknown   Local(target) = ref(d) \/ ref(e)
eval: [Unknown]
```

And compiler output:

```
Incrementor.java:19: Could not prove that plusOne(int) was a
function: Program
state tainted
target.val++;
      ^
1 error
```

The interaction of `break` and `while(...)` is working correctly; as the `RETURN:` condition hints at, due to that loop, `target` could either be `this`, which a function should not be allowed to modify, or the newly instantiated `Incrementor` which may be modified. The compiler is correct in disqualifying `plusOne` as a `function` as the possibility exists that `this` could be tainted at the line indicated due to the duality of `target`.

## C.8 Overriding functions

```
public class AClass
{
    public function void myMethod()
    {
    }
}

class BClass extends AClass
{
    public mutator void myMethod()
    {
    }
}
```

Fails to compile with error output:

```
AClass.java:10: myMethod() in BClass cannot override myMethod()
in AClass; overriding method has incompatible behavioural type
        public mutator void myMethod()
                            ^
1 error
```

This is entirely correct. A `mutator` may not override a `function`, as specified in the proof rules(Section 3.1.6).

# Project Proposal

Richard Thrippleton
ret28
Churchill College

Part II Computer Science Tripos Project Proposal
Functional Java
23/10/2002

| | |
|---|---|
| Project Originator: | Richard Thrippleton |
| Special Resources: | None |
| Project Supervisor: | Tim Harris <Tim.Harris@cl.cam.ac.uk> |
| Director of Studies: | Christine Northeast <Christine.Northeast@cl.cam.ac.uk> |

# Introduction

In short, this project aims to extend the Java language to allow methods to be marked as side-effect free, in the style of a functional programming language such as ML. The programmer will be given a keyword to mark such methods, and the compiler will prove the claims or throw a compile error trying.

## Detail

This is an example of programming by contract, specifically aimed at putting the functional contract into the Java language. Programming by contract refers to procedures/functions being annotated with formal pre and post-conditions of invocation, which the implementer must stick to. In most languages, this is enforced informally, with the conditions being documented in plain-text and the onus being on the developers to make their code stick to this. Some languages enforce this more formally, such as Eiffel which can enforce certain kinds of contract with language integrated assertions that are tested before and after invocation of a method at runtime. However, this scheme isn't powerful enough for feasibly implementing the functional contract.

The functional contract specifies that the entire program state does not change between immediately pre-invocation and immediately post-invocation of a method. A functionally contracted method does this and only this; it takes arguments and returns a new value/object that may or may not be based on these arguments. Changes to local state within the method does not count as a change to program state, as that state will no longer exist after return. The functional contract can make use of the method by a fellow developer a considerably easier task; he/she does not have to walk through the implementation code to know for certain that calling it has NO hidden surprises. The definition of state change will certainly not include allocation of new objects on the heap/stack, for the sake of practicality in functional programming.

Implementing this within Java entails that a method signature modifier (called 'function') be added to the language, as well as implementing a contract prover within the compiler. Any method tagged with this will come under the scrutiny of the modified compiler, at a late stage of compilation, which will attempt to prove that the method does indeed conform to the functional contract as the programmer has claimed. In answering the question "Is this side-effect free?" the prover will not be able to answer "Yes" or "No" but rather "Yes" or "Don't know" The former is an impossible problem, for example considering all the obfuscation that can be done along the lines of changing a program's state within a method but changing it back before returning. Should the contract prover answer "Don't know" on a 'function' method, the compilation will fail in the style of a conventional compile time error.

## Plan of work

Using arbitrary units of 10 work packages lasting roughly a fortnight each, on top of the date constraints.

Package 1

28/10/2002 - 10/11/2002

Theoretical foundation: devising a formal scheme for proving that a particular method is functional; the functional prover. The prover will not be tremendously complex, and will definitely fail to pass a significant number of functions that are actually side effect free. Previous work in the area of effects will be researched, certainly including relevant sections of the Optimising Compilers course.

I will complete this stage with a working, proven scheme, suitable for implementation in the first implementation stage.

Package 2

11/11/2002 - 24/11/2002

Existing code-base research: I will obtain and examine the source code to the Sun Java Compiler, with a view to modifying it to implement the functional prover. The same also for the Byte Code Engineering Library, which I believe to be necessary to make the task feasible.

I will have obtained all necessary libraries and third party code, understanding them in sufficient detail to use them in the next stage

Packages 3,4

25/11/2002 - 20/12/2002

Implementation: programming the functional prover and integrating it into the compiler.

First package; coding. Second package, debug/fix cycle.

The deliverable is the working realisation of my initial theoretical scheme, integrated into Sun's javac.

Package 5 13/01/2003 - 31/01/2003

Assessment for extension: I will assess my theoretical work with a view to finding extensions to it that decrease the number of genuinely side effect free functions that aren't passed by the compiler as being so. At the end of this, I will have the knowledge required to implement said extensions into the already modified compiler. These will go into the progress report that will be complete at the end of this stage.

Packages 6,7 03/02/2003 - 02/03/2003

Extensions: The above extensions to the functional prover will be implemented in the modified compiler.

Subdividing this further is near impossible, as I do not yet know what extensions or how many I will create.

Packages 8,9,10

03/03/2003 - 25/04/2003

Dissertation write-up. By the end of this period, it will be in a suitable state for submission.

Any time after this up until the stricter deadline of 16/5/2003 will be overflow time, to allow for any of the above stages to overrun.