Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 3–16
www.elsevier.com/locate/entcs

Tickling Java with a Feather

Tristan O.R. Allwood[1] Susan Eisenbach[2]

Department of Computing
Imperial College
London
United Kingdom

Abstract

Fragments of mainstream programming languages are formalised in order to show desirable properties of their static semantics. We ask if said formalisms could also be used to define a test suite for the mainstream programming language in question, and what the utility of such a suite would be.
In this work, we present our findings from testing Java with Featherweight Java (FJ). We take the syntax and binding structure of FJ to define an instance space of non-isomorphic test programs and implementations of FJ type checkers to provide oracles for our tests, to ensure the mainstream implementation conforms with the expectations of FJ. Using these, we evaluate (using code coverage techniques) how much of the Sun OpenJDK `javac` can be tested by FJ.

Keywords: Featherweight Java, Tests, Semantics, Oracles

1 Introduction

Writing compilers and type checkers is hard. In addition to the sheer quantity of code, the implementation may be complicated by the desire to produce code that is efficient and fast. The languages compilers are processing are ever-increasingly complicated, with sophisticated type rules, and many possible obscure corner cases. There are also complications with possibly several intermediate representations being used inside the compiler, each with their own invariants and properties.

[1] Email: tristan.allwood@imperial.ac.uk
[2] Email: s.eisenbach@imperial.ac.uk

doi:10.1016/j.entcs.2009.09.037

Creating tests for a new language compiler is time consuming. The test cases need generating, and an oracle consulted to determine if it should be a passing or failing test. If it is a human oracle, it is possible that they could be wrong. Test cases are then also limited by human imagination, obscure corner cases could be overlooked. If the test cases are being generated by the compiler writer; it is possible that they could be biased or influenced by their own assumptions about the implementation or their interpretation of what the language should do.

There is an issue of maintaining or upgrading the tests if the language evolves. The validity of some old tests may change if the language semantics are altered during development, and a human oracle has to find and check all possible tests that are affected. This is as-well as finding new tests for interactions with existing language features and any new ones added.

Many programming languages have been given a formal presentation; either in their entirety, or for a semantically meaningful core subset. This formalism is used to prove desirable properties of the semantics of the language, both static and dynamic. However, the language is only proved safe in theory - we still rely on a correct implementation of the compiler of the language.

This work asks whether the formal presentation of the theory could also be used both as input to generate test programs and to be an oracle for them. They can then be executed by the implementation of the full language to see if it conforms with the theory. We want to evaluate how useful these test programs would be in practice.

To start answering the question, we present here an investigation into using the theory of Featherweight Java [10] to create tests for the type checker component of the OpenJDK Java compiler [14]. We proceed as follows;

In Section 2 we briefly summarise Featherweight Java, and discuss features and omissions that become interesting later in the paper. We then describe how we take the grammar component of FJ and use it to generate test programs in Section 3. We also describe how we use knowledge of name binding to prune some isomorphic programs from the search space of those generated. In Section 4 we describe how we use implementations of FJ type checkers as our oracles to determine whether the test program is one that `javac` should accept or reject. We also discuss the presence of FJ programs that FJ rejects but that `javac` will accept. With the test programs generated and their expectation provided, we describe the set up used to test the OpenJDK Java compiler in Section 5. The results from the experiment are presented in Section 6. Finally we put this work in context in Section 7 before concluding and looking to the future in Section 8.

2 Featherweight Java

`CL ::=`	*class declarations:*
`class C extends C {`$\bar{\texttt{C}}$ $\bar{\texttt{f}}$`; K` $\bar{\texttt{M}}$`}`	
`K ::=`	*constructor declarations:*
`C(`$\bar{\texttt{C}}$ $\bar{\texttt{f}}$`) {super(`$\bar{\texttt{f}}$`); this.`$\bar{\texttt{f}}$`=`$\bar{\texttt{f}}$`;}`	
`M ::=`	*method declarations:*
`C m(`$\bar{\texttt{C}}$ $\bar{\texttt{x}}$`){return t;}`	
`t ::=`	*terms:*
`x`	*variable*
`t.f`	*field access*
`t.m(`$\bar{\texttt{t}}$`)`	*method invocation*
`new C(`$\bar{\texttt{t}}$`)`	*object creation*
`(C) t`	*cast*

Fig. 1. Syntax of Featherweight Java

For this experiment we have chosen to use Featherweight Java [10] as our formalised fragment, and Java as our target testing platform. FJ is one of the most studied formalisms of Java and has been used as a starting point for research into many extensions of Java ([12], [3], [5], [17]) including Generic Java in the original paper.

FJ is designed to be a minimal calculus for Java. The authors omitted as many features as possible while still retaining the core Java typing rules. This included omitting assignment, making FJ a purely functional calculus.

FJ has its own syntax (Fig. 1) which is a restricted subset of Java - all syntactically valid FJ programs are syntactically valid Java programs. An FJ program as presented in [10] consists of a list of FJ class declarations followed by an expression that is taken as the code for the `main` method. Here we omit that expression and treat a program simply as a list of class declarations.

Classes declare a superclass, a number (possibly zero) fields, a constructor, and zero or more methods.

Constructors accept arguments to initialize their fields with. FJ requires the arguments to both have the same names as their respective field names, and also to be in a strict order that matches the field layouts of the classes'

superclass prefixing that of the current class.

Methods accept arguments (arguments are not allowed to be named `this`), and have an expression as their method body. Expressions can be a variable reference, a field lookup, a method invocation, an object creation or a casting operation.

There are many features of Java that are not in the FJ abstraction, for example assignment, field shadowing, covariant return types in overridden methods, method overloading, interfaces, enumerated types, nested class declarations, and many others. As we discuss in Section 4, some of these missing features mean that there are FJ programs that FJ fails to type check that full Java would accept. However, all FJ programs that FJ can type are valid Java programs.

3 Defining Tests

FJ provides a grammar that describes syntactically valid FJ programs. The test programs we generate are all instances of that grammar. We instantiate the grammar by walking it using a bounded, depth-first exploration algorithm. We use structural constraints limiting the maximum number of classes, the number of fields and methods per class, and the complexity (sum of all production rules used) of expressions in each method and the number of variables used in a method, to ensure the depth first exploration does not explore an infinite space.

We use a depth first exploration scheme as it does not suffer the memory explosion problem a breadth first search would suffer. The high branching factor in the grammar means that even at small depths of the tree there would be a lot of programs in the breadth-first 'to visit' queue. This queue would grow exponentially (as opposed to the depth first stack experiencing linear growth) with each production in the grammar that has a choice of values that is visited. However we can emulate a quasi-breadth first walk through the instance space of programs by using iterative deepening. Since we can alter the constraints that bound the depth first walker, we can start it with small constraints, and iteratively grow them until the size of the search space ceases to be tractable for it to explore completely.

The grammar of FJ also makes reference to potentially infinite domains for class names, variable names, etc. For the depth first exploration algorithm to function effectively, it requires a bounded domain for each of these infinite domains. The simple solution to this is to create constraints for the number (and names) of valid class/method/field/arguments, and whenever (for example) a class name is required in a program, n copies of the program are produced,

each using a different substitution from the n available class names.

This approach has the effect of specifying many programs that are isomorphic or α-equivalent to each other. For example:

```
P1:
class C1 extends Object { C2 { super(); } }
class C2 extends Object { C1 { super(); } }
```

and

```
P2:
class C2 extends Object { C1 { super(); } }
class C1 extends Object { C2 { super(); } }
```

If we can assume that the internal representation of names in the Java compiler doesn't try to inspect their values (except to compare them to each other and some built-in values such as `Object` or `this` using library methods), then we can prune away many of the isomorphic programs. To do this, we augment FJ with a notion of *binding*.

We make the non-grammar domains in FJ be populated by some default values (`Object` or `this`), and then other values are bound from some site in the program. For example a new class name is bound globally (across the whole program, before and after the site) when a `class C1 extends C2 { ... }` production is instantiated. The new class name invented becomes the value for `C1`. Field and method declarations both globally bind new field and method names respectively. Method declarations also bind new variables from their argument list locally, so they are only visible to the method's code expression.

Because class, method and field names bind globally, the names can be forward referenced inside earlier definitions. For example in P1 above, the constructor return type `C2` is a forward reference to the next class declaration. To be able to know which globally bound names will be available, the generation algorithm proceeds in two phases.

First a *skeleton* is generated that describes the top level structure of the program and consequently the numbers of global binding sites. The skeleton specifies the number of classes, and for each class, the number of fields and methods that it has. This then specifies the domains of class, method and field names. The skeleton is then instantiated using the remaining binding constraints to specify the number of arguments methods have, and how complex method expressions can be (for example see Fig. 2).

As is expected, the size of the instance space of programs still grows exponentially. The actual sizes for the search space under different constraints is presented in our results in Section 6.

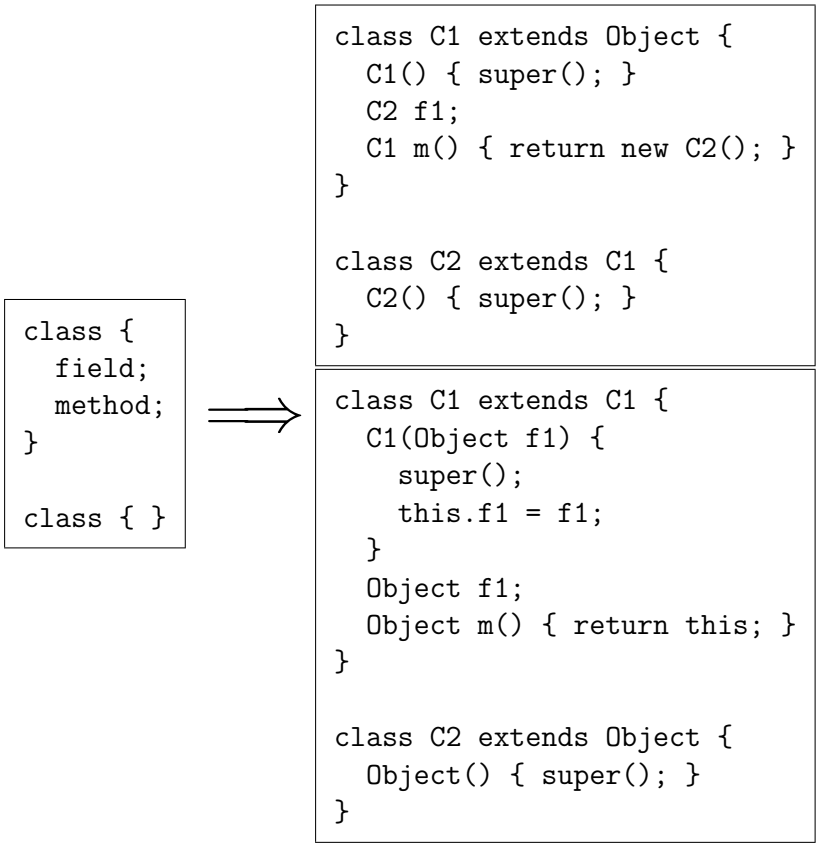

Fig. 2. An example skeleton and two of its possible instantiations

The current generation scheme could be improved in some ways. If we assumed that the relative order of some declarations (e.g. class and method declarations) were of no significance to the compiler we will test, then a further class of isomorphic programs could be pruned.

The current design also precludes generating tests featuring references to unbound names. This means we do not have tests that (for example) check all class names that are used are defined somewhere. We are currently investigating how to do this in a way that does not generate α-equivalent programs, and the trade-offs of allowing just a single undefined name or many distinct undefined names.

4 Classifying Tests

The generated FJ programs by themselves are not very useful, as they are just programs. For them to become tests, they need associating with an expected result for running the test against `javac`. The expected result is provided by

an oracle, in this case we have used an implementation of FJ's type checker [1].

To help ensure our oracle is correct, we have used our generated test programs to check that it gives the same outputs as another implementation of FJ. Given the Java compiler we have chosen to test, we also expect that the implementation of `javac` is actually correct - so the oracle should agree with it in most cases (which it does). However there are some cases where the FJ oracle and `javac` do not agree.

To be as exhaustive as possible, we want to generate both positive and negative test programs for `javac`; i.e. tests that we expect to type check and tests we expect to be rejected. However we have had to be careful. FJ type checking rules on FJ programs are sound w.r.t. Java. If FJ statically accepts a program, we expect Java to accept it. However there are FJ programs that FJ statically rejects that Java will accept. For example, Java supports covariant returns in overridden methods and does not require non-final instance fields to be initialized in constructors, whereas FJ would reject programs that contained these features. There are also some classes of program where the reason FJ rejects the program is strong enough to say Java should reject it too. For example creating cycles detected in the class hierarchy or trying to declare a class named `Object` are always errors in both FJ and Java programs.

When applying the oracle to the test programs, we check whether the test program type checked or not. If it failed to type check we only pass it to `javac` if it was rejected for a reason we would expect `javac` to reject it for (e.g. there was a cycle in the class hierarchy). In this way, we are only testing `javac` (or, in the experiment run here, collecting coverage on) with programs that we can check `javac` agrees with our expectations.

5 Experiment

We have a driver program that generates the FJ programs in a given search space, counts them, applies the oracle to them and counts the number that type checked (for interest). It then counts the number of programs we can actually use as tests to the Java implementation and filters out those we can't.

This driver communicates with a small server Java program we have wrapped around the OpenJDK Java compiler [14]. The server receives programs and invokes the compiler using the Java 6.0 *JavaCompiler* API [13]. This allows the compiler to receive and process the test programs entirely in memory without needing to round-trip to the file system to write out the source files, which gives huge performance gains. Any `javac` compile errors are caught using Java's exception (`try`/`catch`) mechanism. Finally the result of attempting

to compile the program is communicated back to the driver program, which checks the results conform to the expectation from the oracle.

The use of this server program also aids greatly with collecting code coverage results for a test suite (all the tests in a particular search space). We instrumented the Open JDK compiler jar using the code coverage tool EMMA [9], and ensured the server invokes this. As the server is only invoked once and kept running during the testing of an entire suite, we do not need to do a post processing step of combining individual test code coverages together to find out the coverage of a suite.

We also ran all the test suites against a Java implementation of the FJ type checker using a similar server idiom, and collected the corresponding coverage results.

We ran the experiment for various test-space configurations. All the runtimes were less than 30 minutes on a dual core 3.20GHz P4, 2GB ram, Linux 2.6.22-1-amd64, x86_64. Due to the exponential nature of the growth of the instance spaces, configurations that did not complete within this time are unlikely to be completely explorable in a tractable amount of time.

6 Results and Evaluation

In Table 1 we present the code coverage results obtained using our test suites against the OpenJDK `javac` and one of our implementations of Featherweight Java. The full code coverage output is obtainable from [1]. Each column presents the results from using one particular test suite.

The *TestSuite*s are parameterised by the maximum number of classes, the maximum number of methods each class may have, the maximum number of fields each class may have, the maximum number of variables each method may be parameterised by, and the maximum number of expression production rules that may be used in each method. If a test suite is parameterised by values that are all equal or greater than the corresponding values in a second suite, then it will contain all the test programs that the second one has. We refer to test suites by their parameter configurations; e.g. `11104` in the rightmost column is the test suite defined by 0-1 classes, 0-1 methods per class, 0-1 fields per class, 0 parameters to each method and 0-4 grammar productions used in expressions in each method.

The *configuration space size* gives how many test programs are generated under the constraints of that test suite. We also present the number of programs in the test suite that pass the FJ type checker. The number of *usable programs* identifies how many programs in the search space will be used to collect code coverage results. The programs filtered out are ones which we

TestSuite	00000	10000	11001	10100	11011	20000	11012	11111	11112	11103	21001	12111	11113	11104
Configuration space size	1	9	57	201	281	333	1641	7001	47529	62025	54813	238201	360137	542361
Number of type safe programs	1	2	7	4	20	5	57	46	171	186	251	658	633	675
Number of usable programs	1	9	57	157	281	333	1553	6141	41957	50037	54813	217757	317509	437735
UP that TC (%)	100.00	22.22	12.28	2.55	7.12	1.50	3.67	0.75	0.41	0.37	0.46	0.30	0.20	0.15
Javac (%)	12.27	23.36	25.16	25.49	25.81	23.44	27.13	27.07	28.04	27.97	25.25	27.07	28.09	27.97
code (%)	19.04	28.93	31.87	30.81	32.10	29.03	35.96	32.97	36.28	36.33	31.98	32.97	36.33	36.33
comp (%)	6.21	22.28	25.21	26.45	26.37	22.44	28.46	28.77	30.37	30.48	25.36	28.79	30.48	30.49
FJ (%)	6.17	18.38	28.44	25.95	30.46	19.18	35.67	36.56	42.67	43.85	29.25	37.23	44.65	44.43
model (%)	1.64	25.37	54.02	34.84	57.62	25.78	72.95	66.68	86.93	83.89	57.30	66.68	87.46	83.89
passes (%)	3.60	17.77	41.08	44.96	54.53	17.77	65.76	71.29	86.12	84.03	41.08	71.29	87.12	84.03
typecheck (%)	15.61	31.46	62.56	31.46	62.56	35.79	82.13	62.56	87.56	88.17	64.45	64.39	91.22	88.17

Table 1: Code coverage of OpenJDK `javac` and FJ

cannot use as tests because FJ rejects them when Java might accept them, for example they may fail FJ because a method overrides with a covariant return type, which Java supports.

We note that the percentage of usable programs that actually type check (*UP that TC*), and observe that as the space of syntactically correct programs expand, the percentage of those programs that are type correct drops to less than 0.2%.

The final lines gives the percent of executable lines that were executed by the Java and FJ compiler while processing the test suite. We present results for the entire compilers in question, and also local results for the packages within the compiler source that are more focused on type checking. The `javac` implementation is 26163 executable lines of Java, and the FJ implementation is 4990 executable lines of Java.

Test suite `00000` (the first column) gives the results for running the empty program through the type checkers. This establishes a baseline percentage we get "for free" with both compilers. The code executed here is the core path through the compiler, including the amount of work done by any static initializers.

Some trends are apparent in the results. For example, the change in code coverage when moving from test suite `10000` to `20000` shows a relatively low increase in the code coverage of Java, but a slightly larger jump in FJ. This is due in part to the number of new concepts or conditions being tested in the second case (e.g. possibly a user defined cyclic class hierarchy) being low in terms of all the concepts in Java, but less insignificant in the face of FJ.

Comparing the results of test suite `11103` with `11113` and `11104` is also telling. Despite the latter two test suites being over 6 times bigger than the former, the coverage of `javac` is only negligibly larger. An expression complexity of 3 is large enough for all expression productions to be used in some way. Increasing the limit from 3 to 4 with almost no change in the type checking code of both FJ and `javac` show how the recursive nature of the implementations of the type checkers can be tested mainly by lots of small examples. However there is a small code increase outside of the type check packages which is due to the parser now handling more complicated expressions. They also demonstrate that adding method arguments to method definitions is also negligible in the context of `javac`, but a more meaningful to the much simpler FJ, which gains about 3% coverage increase.

It would be nice if our tests could achieve 100% test coverage of FJ. However there are several reasons why this could not be achieved in this experiment. The best coverage of all of the FJ implementation was just below 45%, while the packages relating to type checking where 80% - 90% covered in the

larger tests. The FJ implementation featured a parser written in pure Java, and since all the programs being provided to it are syntactically correct, a large proportion of that code was not exercised. We also collected coverage for the same tests that we provided to `javac`, which means the programs that were not usable and filtered out where also not used to collect FJ code coverage. To further explore this, we re-ran test-suite `11103` against our FJ implementation without filtering out programs and the code coverage was 0.5% higher.

7 Related Work

The idea of generating data structures for testing purposes has been well explored in the literature. In the context of Haskell, the testing frameworks QuickCheck [7], SmallCheck [15] and Lazy SmallCheck [11] all generate instances of data structures as inputs for test predicates. QuickCheck generates the data structures randomly, whereas SmallCheck performs a structural walk of the data structure instance space. Lazy SmallCheck takes advantage of laziness in Haskell to only generate the parts of the test data that get evaluated by the user test predicate. Unforced parts of the data structure represent variations that can be pruned from the exploration space. Although none of these libraries are designed to handle data structures with binding patterns in them (which our FJ programs feature), Lazy SmallCheck's way of pruning the search space does seem like an appealing method of making the instance space more tractable. If we can determine that an FJ program will fail to type check due to a given declaration in the first class, we could prune away all variations of the rest of the code that will fail to type check for the same reason. While this removes a lot of test cases from the suite, it would in theory focus the test generation on more interesting tests, and would allow larger configuration parameter spaces to be explored.

In [6], the authors present their work on looking for counter-examples to predicates, in the context of programming language meta-theory. They use α-Prolog, which is a variant of Prolog extended with the concept of "fresh" (unbound) names. This gives them a tool which allows them to talk about binding and uses a depth-first pruning search. Although in their case they are looking for counter examples (since the test is in the same language as the generation), it could be possible to alter their approach to generate search spaces, which could implicitly have our non-name-isomorphic pruning technique applied.

Model checkers and automated test generators use the *small scope hypothesis*) to justify why testing most (or all) inputs in some small bounded domain can give confidence in the reliability of an implementation. In [2] the authors

attempt to evaluate the accuracy of the small scope hypothesis in the context of Java libraries. They use a tool (Korat [4]) that generates non-isomorphic Java programs of bounded size matching a predicate, and check implementations of several library data structures (LinkedList, HashSet, TreeMap, etc), using standard metrics of statement coverage and *mutant killing* [3] to see how much of the space of the implementation they have tested in the bounded size of inputs. The results of the experiment show that even with low bounds set on the inputs, very high (generally near 100%) mutant killing ratios are achieved, and over 80% statement coverage is achieved. We achieve similar results in the type checking packages of our FJ implementation. While we do not achieve these levels of coverage on the Java compiler (nor would we expect to), we do show that lots of small, simple tests can cover a non-trivial amount of the code base.

In [8] the authors instantiate Java programs and test the refactoring engines of two popular IDEs. Instead of enumerating all possible programs, they provide an API to specify constraints on the programs generated, and perform a bounded search upon that space. Oracles for their tests are provided by heuristics (e.g. the transformed source code compiles, inverting an applied refactoring returns the source code to it's starting point) and by conformance testing the refactoring results between the two IDEs.

Automated test generation in the context of Java has also been attempted in [16] for checking runtime semantics. Here the focus has been on testing conformance of JVM's to the J2SE JVM. The standard Java JVM was used as test oracle, and programs are randomly generated within a bounded size.

8 Conclusion and Future Work

Using large numbers of very small and simple Featherweight Java programs, we can achieve a test coverage of around 80% - 90% of an FJ type checker. Adding the programs that were filtered out because they were possibly correct Java programs but incorrect FJ programs hardly increases the code coverage at all. This may be because the constraints keeping the problem "small scope" and hence tractable are too limiting; for example none of our tests create method calls featuring two arguments. Achieving a near-100% code coverage for FJ type checking is a future goal, that requires a more structured approach to generating tests that doesn't have the explosion in the state space we currently experience.

[3] Mutant killing is where the library code under test is automatically mutated to introduce subtle bugs and then seeing what percentage of the mutants survive - i.e. they go undetected as bugs.

The same tests run on the OpenJDK `javac` correspond to exercising 25% - 30% of the code base of a full, industrial strength Java implementation. The results indicate that the recursive nature of expressions, and the associated recursive implementation in compilers, means that testing using lots of small expressions can be effective.

Currently we use an implementation of Featherweight Java as our oracle. This has the circular problem that we require our oracle to be correct in order to make assertions about the tests that it is classifying. In this work we used two implementations of Featherweight Java and use them to test each other. Some further work would be to look at ways of automatically deriving an oracle for the simpler language based upon it's type rules.

In future we would like to investigate alternative and new ways specifying the search space to explore, perhaps abandoning total search space coverage for test suites which are disjoint but composable. This would enable us to cover more base-cases in the compiler without creating tests for language features which are implemented orthogonally and thus creating every permutation of the interaction doesn't test anything new.

References

[1] Tristan Allwood and Susan Eisenbach. JavaFeather, code coverage output and FJ implementations. http://www.doc.ic.ac.uk/~tora/JavaFeather/.

[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis".

[3] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight Java with multi-methods. In Vasco Amaral, Luis Marcelino, Luís Veiga, and H. Conrad Cunningham, editors, *PPPJ*, volume 272 of *ACM International Conference Proceeding Series*, pages 83–92. ACM, 2007.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 123–133, July 2002.

[5] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an Existential Types Model for Java Wildcards. In *Formal Techniques for Java-like Programs (FTfJP) 2007*, July 2007.

[6] J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2007.

[7] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[8] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[9] Vlad Roubtsov et. al. EMMA: a free Java code coverage tool. http://emma.sourceforge.net/index.html.

[10] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 132–146, N. Y., 1999.

[11] Fredrik Lindblad, Matthew Naylor, and Colin Runciman. Lazy SmallCheck: A library for demand-driven testing of Haskell programs. http://www-users.cs.york.ac.uk/~mfn/lazysmallcheck/index.html.

[12] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007. To appear.

[13] Sun Microsystems. JavaCompiler interface. http://java.sun.com/javase/6/docs/api/javax/tools/JavaCompiler.html.

[14] Sun Microsystems. OpenJDK. openjdk.java.net.

[15] Colin Runciman. SmallCheck: another lightweight testing library in Haskell. http://www.cs.york.ac.uk/fp/darcs/smallcheck/.

[16] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. Random program generator for Java JIT compiler test system. In *QSIC*, page 20. IEEE Computer Society, 2003.

[17] T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for Featherweight Java, 2003.