Lightweight Java

Rok Strniša Matthew Parkinson

December 14, 2021

Abstract

Lightweight Java (LJ) is an imperative fragment of Java [5]. It is intended to be as simple as possible while still retaining the feel of Java. LJ includes fields, methods, single inheritance, dynamic method dispatch, and method overriding. It does not include support for local variables, field hiding, interfaces, inner classes, or generics. The accompanying Isabelle script proves the type soundness of the Ott-generated LJ definition.

1 Description

When designing or reasoning about a language feature or a language analysis, researchers try to limit the underlying language to avoid dealing with unnecessary details. For example, object-oriented generics were formalised on top of Featherweight Java (FJ) [6], a substantially simplified model of the Java programming language [5].

Many researchers have used FJ as their base language. However, FJ is not always suitable, since it is purely functional — it does not model state; there are only expressions, which are evaluated completely locally. Therefore, FJ is a poor choice for language analyses or language features that rely on state, e.g. separation logic [7] or mixins [3].

In this chapter, we present Lightweight Java (LJ), a minimal *imperative* core of Java. We chose a minimal set of features that still gives a Java-like feel to the language, i.e. fields, methods, single inheritance, dynamic method dispatch, and method overriding. We did not include type casts, local variables, field hiding, interfaces, method overloading, or any of the more advanced language features mainly due to their apparent orthogonality to the Java Module System [11], a research topic at the time; however, we later realised that, by including type casts and static data, we could formally verify properties regarding class cast exceptions (or their lack of) and module state independence — this extension remains future work.

LJ's semantics uses a program heap, and a variable state, but does not model a frame stack — method calls are effectively flattened as they are executed, which simplifies the semantics. In spite of this, LJ is a proper

subset of Java, i.e. every LJ program is a valid Java program, while its observable semantics exactly corresponds to Java's semantics.

LJ is largely a simplification of Middleweight Java (MJ) [2]. In addition to the above, MJ models a stack, type casts, and supports expressions (not just statements).

LJ is defined rigorously. It is designed in Ott [8], a tool for writing definitions of programming languages and calculi. From LJ's Ott code, the tool also generates the language definition in Isabelle/HOL [1], a tool for writing computer-verified maths. Based on this definition, we mechanically prove type soundness in Isabelle/HOL, which gives us high confidence in the correctness of the results.

Initially, we designed LJ as a base language for modelling the Java Module System, Lightweight Java Module System (LJAM) [10], and its improvement, Improved Java Module System (iJAM) [9] — in both, we achieved a high level of reuse in both the definitions and proof scripts. Through this process, LJ has been abstracted to the point where we think it can be used for experimenting with other language features. In fact, LJ has already been used by others to formalise "features" in Lightweight Feature Java [4].

2 Example program

Here are two Lightweight Java class definitions, which show the use of class fields, class methods, class inheritance, method overriding, subtyping, and dynamic method dispatch.

```
class A {                  // class definition
    A f;                   // class field
    A m(B var) { this.f = var; return var; }    // subtyping
}

class B extends A {        // class inheritance
    A m(B var) { this.f = var; return this; }  // overriding
}

// A a, result; B b;
a = new B();               // subtyping
b = new B();
result = a.m(b);           // dynamic method dispatch (calls B::m)
```

Due to method overriding, the method call on the last line calls `B`'s method `m`. Therefore, when the execution stops, both `result` and `a` point to the same heap location.

3 Extending the language

The easiest way to extend the language is to modify its Ott source files. To prove progress and well-formedness preservation of the extension, you can

either:

- modify the existing Isabelle scripts; or,
- prove that any valid program of the extended language can be reduced to a program in LJ.

4 More information

More information about Lightweight Java's operational semantics, type system, type checking, and a detailed walkthrough of the proof of type soundness can be found here:

http://rok.strnisa.com/lj/

References

[1] Isabelle. http://isabelle.in.tum.de/.

[2] G. Bierman, M. J. Parkinson, and A. Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report 563, Computer Laboratory, University of Cambridge, Apr. 2003.

[3] G. Bracha and W. R. Cook. Mixin-based Inheritance. In *Proceedings of OOPSLA*, volume 25(10) of *ACM SIGPLAN Notices*, pages 303–311. ACM Press, Oct. 1990.

[4] B. Delaware, W. R. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. http://www.cs.utexas.edu/~bendy/featurejava.php, Oct. 2008.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. Sun Microsystems, Inc., Third edition, May 2005.

[6] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of OOPSLA*, volume 34(10) of *ACM SIGPLAN Notices*, pages 132–146. ACM Press, Oct. 1999.

[7] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of LICS*, pages 55–74. IEEE Computer Society, July 2002.

[8] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective Tool Support for the Working Semanticist. In *Proceedings of ICFP*, volume 42(9) of *ACM SIGPLAN Notices*, pages 1–12. ACM Press, Oct. 2007.

[9] R. Strniša. Improved Java Module System (iJAM). http://rok.strnisa.com/iJAM/, Nov. 2007.

[10] R. Strniša. Lightweight Java Module System (LJAM). http://rok.strnisa.com/ljam/, Mar. 2007.

[11] Sun Microsystems, Inc. JSR-277: Java™ Module System. http://jcp.org/en/jsr/detail?id=277, Oct. 2006. Early Draft.