A Bytecode Logic for JML and Types

Lennart Beringer and Martin Hofmann

Institut für Informatik, Universität München
Oettingenstrasse 67, 80538 München, Germany
{beringer, mhofmann}@tcs.ifi.lmu.de

Abstract. We present a program logic for virtual machine code that may serve as a suitable target for different proof-transforming compilers. Compilation from JML-specified source code is supported by the inclusion of annotations whose interpretation extends to non-terminating computations. Compilation from functional languages, and the communication of results from intermediate level program analysis phases are facilitated by a new judgement format that admits the compositionality of type systems to be reflected in derivations. This makes the logic well suited to serve as a language in which proofs of a PCC architecture are expressed. We substantiate this claim by presenting the compositional encoding of a type system for bounded heap consumption. Both the soundness proof of the logic and the derivation of the type system have been formally verified by an implementation in Isabelle/HOL.

1 Introduction

Modeling languages such as JML [25] allow the software architect to specify functional and non-functional behaviour of code modules. Typically, these languages comprise a variety of specification idioms such as partial-correctness specifications using pre- and post-conditions, termination measures, specification of exceptional behaviour, model fields, ghost variables and fields, invariants at object or class level, lightweight specifications, or the inclusion of pure (i.e. non-side-effecting) code in specification clauses. Although the precise interpretation of some of these features is still a matter of ongoing debate, a number of verification tools have been presented that validate code w.r.t. JML specifications [14]. Although the proposed formalisms mainly target Java source code, they can relatively easily be adapted to bytecode.

The adaptation of specification constructs to low-level code admits a smooth translation of high-level specifications into specifications of mobile code units. However, we do not expect that a similarly direct transfer of validation strategies such as verification condition generators would suffice for their verification, for two reasons. Firstly, bytecode that was obtained by compilation from languages other than Java may not be amenable to the same proof strategies, or may lead to different verification conditions if it has undergone an obfuscation routine. Secondly, a recipient may require transmitted code to be complemented by a proof certifying that the code is safe to execute [28]. Typically, the production of certificates exploits results of program analyses such as type systems. In this case, the validation of certificates by the code consumer is supported if the type system's structuring principles (invariants) are communicated as part

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 389–405, 2006.

of the certificate [4,13]. Again, it is not guaranteed that these abstraction barriers are respected by a verification strategy for source code verification.

In this paper, we therefore propose a program logic for a bytecode language that satisfies requirements motivated by JML specifications and admits different verification strategies to be implemented, including strategies that are suitable for validating high-level type systems. More specifically, we present a formalism where partial-correctness method specifications can be complemented by method invariants and local annotations at intermediate program points whose interpretation applies to terminating *as well as non-terminating* program executions. Non-terminating executions are not covered by traditional (partial or total) Hoare logics, but are required for a faithful interpretation of JML code annotations. They are also desirable for proof-carrying code (PCC) frameworks: the significance of a certificate regarding the safety or the consumption of resources is increased if its validity does not derive from a partial-correctness interpretation - for example, consider a certificate purporting to guarantee an upper bound on the runtime. On the other hand, non-terminating program executions are often implicitly covered by program analysis formalisms such as type systems, but this fact is often not stated (or proven) explicitly, for example if the soundness proof is formulated as a syntactic subject-reduction proof w.r.t. a big-step operational semantics. In order to demonstrate the suitability of our logic for the interpretation of such type systems, we present the syntax-directed encoding of a type system for bounded heap consumption which covers terminating and non-terminating executions.

For presentational reasons, the program logic described in the present paper covers only a small fragment of the JVML. However, in collaboration with partners from the Mobius project [8], a variation of the logic has been produced that covers a more substantial subset of JVML, including virtual method invocations, static fields, arrays, exceptions, and various datatypes. At the same time, work is under way to translate JML specification constructs that are not considered in the present paper into the extended logic, in particular the constructs of JML specification level 0 [25].

Motivation and overview of assertion format. The format of judgements in a program logic is strongly influenced by semantic considerations, i.e. by the conclusions one may draw from a derivable judgement regarding the operational behaviour. Our logic aims to fulfill two sets of requirements. The first requirement concerns JML annotations at intermediate program points. Their common understanding mandates that an assertion A associated to a program point ℓ should be satisfied whenever the control flow reaches ℓ. At first sight, this interpretation motivates a notion of validity like

$$\forall\, s.\ \ell_0, s_0 \to^* \ell, s \Rightarrow A(s) \tag{1}$$

where s_0 denotes the entry state of the program fragment (e.g. method) and ℓ_0 the label of the first instruction. Indeed, this interpretation extends partial-correctness program logics by also applying to non-terminating program executions. Furthermore, the generalisation to binary predicates A, with validity defined by

$$\forall\, s.\ \ell_0, s_0 \to^* \ell, s \Rightarrow A(s_0, s), \tag{2}$$

admits assertions to refer to the initial state, as is required for the translation of idioms such as JML's `old` keyword [22].