Formal Semantics of Model Fields Inspired by a Generalization of Hilbert's ε Terms

Bernhard Beckert Daniel Bruns

It is widely recognized that abstraction and modularization are indispensable for specification of real-world programs. In source-code level program specification languages, such as the Java Modeling Language (JML) [3], *model fields* [4] are a common means for achieving abstraction and information hiding. However, there is yet no well-defined formal semantics for the general case in which the abstraction relation defining a model field is non-functional and may contain references to other model fields. In this contribution, we discuss and compare several possibilities for defining model field semantics, and we give a complete formal semantics for the general case. Our analysis and the proposed semantics is inspired by a generalization of Hilbert's ε terms.

Model fields. Model fields are abstractions of the program's memory state given in a syntactically convenient form (as fields in a class). For example, one may define a model field of type `Set` containing all elements of some linked list, thus abstracting from and hiding implementation detail. Introducing such a model field in the specification is possible even if the type `Set` is not defined in the programming language but only in the specification language. The relation between the concrete state and model fields, i.e., the abstraction relation, is specified by so-called *represents clauses*. In general, abstraction relations may be non-functional, and they may refer to entities which are not present in the concrete program (e.g., other model fields).

Unfortunately, there is no common understanding of what the semantics of model fields is in the general case. The semantics used by verification and runtime checking tools as well as the semantics defined in the literature is restricted to functional represents clauses, to model fields of a primitive type, or by restricting the syntax of represents clauses. The general case, however, raises several questions:

- On which memory locations does the value of a model field depend?
- What value is chosen if the represents clause is non-functional?
- At what points in time does the value of a memory field change?
- What does an unsatisfiable represents clause mean?
- In which cases are represents clauses well-defined? What about recursive represents clauses?

Hilbert's ε terms. The concept of ε terms was first introduced by Hilbert in 1939 as an extension to classical first-order predicate logic [2] with only informal semantics given. An ε term $\varepsilon x.\varphi(x)$, where x is a variable and φ is a formula, stands for 'some domain element u such that φ holds (if such exists)'. These terms can, for example, be used to represent instantiations of existentially quantified variables without assigning a concrete value (i.e., skolemization). The quantifier in $\exists x.\varphi(x)$ can be eliminated by replacing the formula $\exists x.\varphi(x)$ by $\varphi(\varepsilon x.\varphi(x))$. The most interesting semantics of ε terms are called *extensional* semantics, in which the value is given by a deterministic, yet unknown, choice from the set $\{u \mid \mathcal{S}, \beta\{x \mapsto u\} \models \varphi\}$ of objects satisfying φ.[1]

The classical definition of ε terms is concerned with the value of one particular variable. In contrast, model fields are location-dependent symbols and as such they are highly non-modular. This has lead us to a generalization of ε terms whose semantics are defined in terms of a choice of n-tuples, rather than of single elements: $\varepsilon\langle x_0, \ldots x_{n-1}\rangle_i.\varphi$ is a *generalized ε term*, where $x_0, \ldots, x_{n-1}$ are pairwise distinct

[1] The notion $\beta\{x \mapsto u\}$ denotes a (partial) function which maps x to u and equals β on all other domain elements.

variables, $i \in \mathbb{N}$, and φ is a formula. The value is then given as the i-th position of the tuple chosen from the set $\{\langle u_0, \ldots u_{n-1}\rangle \mid \mathfrak{S}, \beta\{x_j \mapsto u_j\} \models \varphi\}$.

Formal semantics of JML with model fields. For semantics of JML in general, we build on the definitions given in [1]. In particular, a *system state* is modeled as a pair (η, σ) where η and σ are partial functions representing heap and stack memory, respectively.[2] η maps a location (i.e., a pair of a semantical object and an identifier) to a semantical object. The semantical value $val(s, \mathtt{o.f})$ of a (concrete) field reference `o.f` in state $s = (\eta, \sigma)$ is then defined as $\eta(val(s, \mathtt{o}), \mathtt{f})$. We extend this notion by introducing a third function ε to states which handles model references.

The first approach to a definition of ε is in the spirit of the valuation of ε terms introduced above: The value of model reference `o.m` constrained by represents clause φ in state $s = (\eta, \sigma, \varepsilon)$ is defined as the choice (given by ε) from the following set:

$$\{u \mid val(s\{\mathtt{this} \mapsto val(s, \mathtt{o}), (val(s, \mathtt{o}), \mathtt{m}) \mapsto u\}, \varphi) = tt\}$$

This definition possesses some well-regarded properties: (i) It is well-defined even in case φ contains references to other model fields (which could constitute a cycle). (ii) It takes care of aliasing and allows framing the model field implicitly since all dependencies are determined dynamically. (iii) It is independent of other semantical issues, in particular, of the handling of undefinedness in the case the above set is empty. (iv) It is not fixed to any particular application, e.g., runtime-checking. Our approach works well in most cases—even when a represents clause contains references to other model fields. However, it fails to preserve our intuition in the case where these references are cyclic. This is due to the fact that for each model field, evaluation is done independently. Consider two represents clauses such that `x` is constrained to $x \geq y$ and `y` to $y \geq x$. Both are clearly satisfiable, but when evaluated separately, it is not implied that `x` and `y` are assigned the same value. This leads us to a second approach in which the model field value is defined as a projection from a tuple which contains appropriate values for *all* model fields attached to created objects such that *all* represents clauses are satisfied simultaneously. Here, it is crucial that the set of 'interesting' locations is always finite.

Conclusion. Model fields are a mighty instrument for specification. However, apart from the KeY tool, there is yet no other tool implementation (neither for runtime checking nor for program verification) that fully supports JML's model fields. Even the most recent runtime checkers possess no or very limited support for model fields. In addition, preliminary results of empirical analyses suggest that non-functional represents clauses are virtually never used in specifications. This finding may be rooted in the non-trivial semantics which hides behind the familiar syntactical guise. In particular, allowing model fields of a reference type or non-functional represents clauses make it significantly harder to give sound semantics while the need for those remains unclear.

References

[1] D. Bruns. *Formal Semantics for the Java Modeling Language*. Diploma thesis, Universität Karlsruhe, 2009.

[2] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, vol. II. Springer, Berlin, 2nd edn., 1939.

[3] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual. Draft Revision 2.344*, Feb 2011.

[4] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, *24*(5), 2002, pp. 491–553.

[2] σ is defined for every local variable and `this`.