Dynamic Logic with Non-rigid Functions
A Basis for Object-oriented Program Verification

Bernhard Beckert[1] and André Platzer[2]

[1] University of Koblenz-Landau, Department of Computer Science
`beckert@uni-koblenz.de`
[2] University of Oldenburg, Department of Computing Science
`platzer@informatik.uni-oldenburg.de`

Abstract. We introduce a dynamic logic that is enriched by non-rigid functions, i.e., functions that may change their value from state to state (during program execution), and we present a (relatively) complete sequent calculus for this logic. In conjunction with dynamically typed object enumerators, non-rigid functions allow to embed notions of object-orientation in dynamic logic, thereby forming a basis for verification of object-oriented programs. A semantical generalisation of substitutions, called state update, which we add to the logic, constitutes the central technical device for dealing with object aliasing during function modification. With these few extensions, our dynamic logic captures the essential aspects of the complex verification system KeY and, hence, constitutes a foundation for object-oriented verification with the principles of reasoning that underly the successful KeY case studies.

Keywords: dynamic logic, sequent calculus, program logic, software verification, logical foundations of programming languages, object-orientation

1 Introduction

Overview. Dynamic logic serves two purposes: (a) theoretical investigations of programs, programming languages, and verification calculi; and (b) formal verification of particular programs. Deductive verification of real-world object-oriented programs requires the use of a program logic that is suitable for object-orientation instead of a logic for a simple While language (e.g. [11]). In this paper, we add a succinct set of features to a dynamic logic for While, which forms a basis for handling object-oriented programming languages; and we present a sound and (relatively) complete sequent calculus for the extended logic. The logic that we introduce, called ODL, is a minimal extension of dynamic logic [11], i.e., only very few essential notions of object-orientation are directly included.

For inclusion in ODL, we have identified the following essentials: (1) an object type system; (2) object creation; and, most importantly, (3) non-rigid functions that can be used to represent object attributes. Using such a minimal extension that is not cluttered with too many constructs is necessary for theoretical investigations (a). A case in point are the soundness and completeness proofs for the ODL calculus, which are—though not trivial—still readable, understandable and, hence, accessible to investigation. Furthermore, ODL is sufficient for verifying programs written in real-world programming languages (b), because they can be transformed into ODL programs uniformly (as practical experience with the KeY prover implementation shows, see below). ODL thus serves both purposes of a dynamic logic. In this paper, JAVA-like languages are considered for transformation into ODL programs.

In addition to providing a sound and complete calculus for ODL, a prime contribution of this paper is the logic ODL itself, which forms a coherent basis for object-oriented verification.

The KeY Project and ODL. The work reported in this paper has been carried out as part of the KeY project [2], the goal of which is to develop a comprehensive tool supporting formal specification and verification of JAVA CARD programs within a commercial platform for UML/JML-based software development. This approach is based on the design-by-contract paradigm. In KeY, contracts are verified statically using a semi-automatic, interactive theorem prover on the basis of a dynamic logic for 100% JAVA CARD [5].

ODL captures the essence of reasoning underlying the KeY approach. Here, we consolidate the foundational principles of KeY into this concise logic, which is not only (relatively) complete in theory but also provides sufficient means for *practical* object-oriented verification. Practical applicability has been demonstrated in successful case studies (e.g. [15]) with the KeY prover. Now, using ODL, we focus on more theoretical aspects in this paper.

Dynamic Logic. The principle of dynamic logic (DL) is to facilitate the formulation of statements about program behaviour by integrating programs and formulas within a single specification language (see e.g. [11] for a general exposition of DL). By permitting arbitrary programs α as actions of a labelled multi-modal logic, dynamic logic provides formulas of the form $[\alpha]\phi$ and $\langle\alpha\rangle\phi$, where $[\alpha]\phi$ expresses that all (terminating) executions of program α lead to states in which ϕ holds, whereas $\langle\alpha\rangle\phi$ expresses that there is at least one terminating execution of α after which ϕ holds. A Hoare-style specification $\{\phi\}\alpha\{\psi\}$ can be expressed as $\phi \rightarrow [\alpha]\psi$. In contrast to Hoare logic and temporal logic approaches, dynamic logic further permits to express structural relationships between different programs, for example, $\langle\alpha\rangle\phi \rightarrow \langle\alpha'\rangle\phi$ and $[\alpha](c \geq 0 \rightarrow \langle\alpha'\rangle c \leq d \cdot d)$.

Object-orientation. Typical features of object-oriented programming languages include structured object data types with inheritance and subtyping, resolving method invocation by dynamic dispatch, overloading, hiding of fields, object creation, exception handling (as well as other means of abrupt completion) and side-effects during expression evaluation. There is no general consensus on the question which of these features constitute the heart of object-orientation and which are just contingent features of object-oriented languages (see, e.g., [14] for a discussion why exception handling is orthogonal to object-orientation). We are *not* trying to answer this question by including some features into ODL and others not. Instead, our choice was to include those features that are (a) frequently used in object-oriented languages and (b) cannot be removed easily by program transformation. We put more emphasis on the latter criterion (b) than on a general philosophy of what should be considered object-oriented.

Related Work. Stärk and Nanchen [20] define a dynamic logic for single steps of abstract state machines and develop a calculus. Their dynamic logic has some features in common with ODL; it uses a related but distinct notion of parallel updates. Their calculus, however, is unwieldy as it uses a multitude of axioms and necessitates several successive translations with complicated reasoning on termination conditions and the absence of clashes. Due to the limitation to single steps, their logic is not suitable for verification of proper algorithms.

Von Oheimb and Nipkow [22] describe a Hoare calculus for NanoJava, which has many more native language features than ODL. Their calculus, that is accordingly more complicated and harder to use than ours, has been formulated in Isabelle/HOL and proven sound and complete relative to a semantics of NanoJava specified in Isabelle. In [16], Nipkow defines a programming language to capture the essentials of object-orientation (without giving a calculus). Yet, this language keeps more built-in features than ODL, like exceptions and casts.

Pierik and de Boer [17] present a wp-calculus for a moderate abstraction of an object-oriented programming language with a fairly rich set of features (without exceptions) and a focus on method invocation, using an assertion language with quantification over sequences of objects. Their calculus uses a complicated treatment of object creation and is proven complete only relative to the strongest postconditions of programs, which is a comparably weak notion of completeness.

Abadi and Leino [1] present a logic for reasoning about a programming language with prototype-based object inheritance. Their logic resembles a formal type system enriched with pre- and postconditions.

Igarashi *et al.* [12] define a λ-calculus for functional Java (without assignments) and use it to investigate type-safety as well as parametric type genericity.

Other approaches [2, 9, 13, 19, 3], which aim to define and use calculi for verifying full (or large fragments of) Java or C# are too complex for our second goal (besides verification) of a small and easy to understand basis that allows theoretical investigations of programs, program languages, and calculi.

The strength of the ODL approach compared to others lies primarily in an (even) smaller amount of language features and a simple language semantics building on classical first-order dynamic logic. With this basis, the ODL calculus is straightforward and behaves reasonably in practical application scenarios. On a proof-theoretical level, a noteworthy difference is that ODL completeness is even proven relative to first-order arithmetic.

Structure of this Paper. After introducing syntax and semantics of the logic ODL in Section 2, the transformation from existing object-oriented languages into ODL is surveyed in Section 3. As the central contribution of this paper, Section 4 introduces a sound and relatively complete calculus for ODL. Finally, in Section 5 we draw conclusions and discuss future work.

2 Syntax and Semantics of ODL

Overview. In addition to dynamic logic for a standard WHILE programming language [11], we use three important concepts.

Type System. The ODL type system needs to represent types of existing object-oriented programming languages. Since classes are a central concept of object-orientation, ODL uses a proper type system rather than an indirect encoding of types as formulas or numbers. Program constructs whose behaviour depends on dynamic typing (like method invocation) can be translated into ODL code with `instanceof` formulas (see Section 4) to access the dynamic type of expressions.

Dynamic Object Creation. ODL needs to have a way to represent object creation and dynamic types. We introduce object enumerators: for each natural number n there is one distinct object, denoted by the term $\mathtt{obj}_{\mathtt{C}}(n)$, of each object-type $\mathtt{C}$. Then, dynamic type-checks simply amount to checking from which of these free type generators an object originates. As opposed to memory models [21], each type has a disjoint set of created objects. Hence, objects of different types are never aliased. The prover can profit from this higher level of abstraction and the resulting simplicity. This design prohibits arbitrary pointer arithmetic, though.

State Updates. Object-oriented programming languages allow to modify object attributes. ODL represents attributes as *non-rigid* function symbols, i.e., functions that may change their value during program execution. Changes to such non-rigid functions are promoted throughout a formula by means of *state updates*, which can be seen as a "semantical" generalisation of syntactic substitutions. The update mechanism of ODL provides a means for handling symbol aliasing and for applying state updates to modalities. Moreover, bundling changes of multiple locations to one *parallel* update of simultaneous effect, accelerates the prover considerably.

Modelling object attributes as non-rigid function symbols emphasises the logical properties of object states. This avoids encoding objects states in memory structures and improves readability (as compared to memory-model-based approaches). For ODL, the usual object access $o.x$ is a notational variant of $x(o)$.

Syntax of ODL. A (nearly) arbitrary type system can be plugged into ODL. For simplicity, it is assumed to form a lattice (which is no real restriction as any type structure can be embedded into a lattice) satisfying additional conditions.

Definition 1. *The type system* TYP *is a (decidable) lattice with sub-type relation $\leq$. Within* TYP*, there is a designated subset of* object-types *(which are subject to object creation). The type lattice conforms to the following restrictions: (a) the type* `nat` *of natural numbers is an element of* TYP*; (b) object-types have only finitely many subtypes; (c) the bottom type $\perp$ is not an object-type; (d) all subtypes of an object type (except $\perp$) are object-types; (e) there is an object-type* `Null`*, which is a subtype of all object-types.*

Note that function and tuple types are *not* part of the object-level type system TYP. Instead, the typing of a function symbol with n parameters of types $\sigma_1, \ldots, \sigma_n \in$ TYP and result type $\tau \in$ TYP is $\sigma_1 \times \cdots \times \sigma_n \to \tau$. Despite (b), TYP may contain infinitely many object-types (that are not subtypes of each other). Assumptions (*c*–*e*) are not essential but simplify notation in the sequel.

Figure 1 shows part of an ODL type system embedding JAVA types. Of the types shown, `Object`, `Date`, and `String`, are object-types. Unlike the JAVA-type `int`, they permit object creation during program execution. The special object-type `Null` represents the type of the single JAVA null-pointer, which is a possible value for expressions of any object-type but not for those of integer types. In the case of JAVA, `nat` will not occur in the original programs but emerge during the transformation in Section 3.

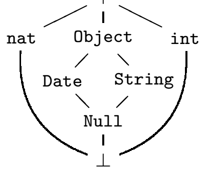

Fig. 1. Lattice for part of JAVA.

Terms and Formulas. The formulas of ODL are built over a set V of variables and a signature Σ of function and predicate symbols, which have a fixed static type. Function symbols are either *rigid* or *non-rigid*, with only non-rigid symbols being subject to assignment during program execution (program variables are represented by non-rigid constants, object attributes by non-rigid functions). Our calculus assumes the presence of sufficiently many symbols of each kind.

The signature Σ is assumed to contain the traditional rigid function and predicate symbols for type `nat`, such as $0, 1, +, \cdot, \leq, \geq$, as well as the rigid symbol `null` of type `Null`. For object-types $\mathtt{C} \in$ TYP, in addition to a non-rigid symbol $\mathtt{next}_\mathtt{C}$ of type `nat` (the number of the next object to be created), Σ contains a rigid function symbol $\mathtt{obj}_\mathtt{C}$ of the typing $\mathtt{nat} \to \mathtt{C}$. The intended semantics of such an *object enumerator* $\mathtt{obj}_\mathtt{C}$ is to enumerate all objects of type `C`.

The set $\mathrm{Trm}(\Sigma \cup V)_\tau$ of *terms* of type τ (or subtypes thereof) is defined as in classical many-sorted first-order logic. Additionally, we use *conditional terms* of the form $(\mathit{if}\, \phi\, \mathit{then}\, t\, \mathit{else}\, t'\, \mathit{fi})$. They evaluate to the value of t if ϕ is true and to the value of t' otherwise (conditional terms are no essential ingredient of ODL, primarily used to simplify concepts and notation).

The *formulas* of ODL are defined as common in first-order dynamic logic. That is, they are built using the connectives $\wedge, \vee, \to, \neg$, equality $\doteq$ and the quantifiers $\forall, \exists$ (first-order part). In addition, if ϕ is a formula and α a program, then $[\alpha]\phi$, $\langle\alpha\rangle\phi$ are formulas (dynamic part). Refer to [18] for a detailed definition of the syntax and semantics of ODL. For enhanced readability, we sometimes use the notation $\forall x\,{:}\,\tau\ \phi$ for quantification when x is of type τ.

Programs. The control structures of ODL are those commonly found in a WHILE programming language: ODL *programs* are constructed using (a) sequential composition $\alpha; \gamma$, (b) conditional execution $\mathtt{if}(\phi)\, \alpha\, \mathtt{else}\, \gamma$, and (c) loops $\mathtt{while}(\phi)\, \alpha$, with quantifier-free first-order formulas ϕ as conditions. The *atomic* programs of ODL are *state updates*:

Definition 2 (State updates). *Let $n \in \mathbb{N}$ and, for $1 \leq i \leq n$, let f_i a non-rigid function symbol of type $\sigma_i^1 \times \cdots \times \sigma_i^{k_i} \to \tau_i$, $f_i(t_i^1, \ldots, t_i^{k_i}) \in \mathrm{Trm}(\Sigma \cup V)_{\tau_i}$,* *and* $t_i \in \mathrm{Trm}(\Sigma \cup V)_{\tau_i}$ *with types* σ_i^j, τ_i*. Then, a* (state) update *has the form* $f_1(t_1^1, \dots, t_1^{k_1}) := t_1, \dots, f_n(t_n^1, \dots, t_n^{k_n}) := t_n$*.*

The intended effect of $f(t) := t'$ is to change the interpretation of f at location t to t', with multiple modifications ($n > 1$) working in parallel, i.e., the t_i^j and t_i are all evaluated prior to the (parallel) modifications.

Method calls can be added to ODL by permitting $\mathtt{c} := \mathtt{m}(t_0, \dots, t_n)$ as an atomic program that represents an invocation of the method `m` on parameters $t_i \in \mathrm{Trm}(\Sigma \cup V)$ and assignment of the result (if any) to `c`. For most programming languages, t_0 is the object on which `m` is invoked. Fixed-point semantics then defines the effect of a method invocation. For simplicity, the particularities of method calls are not formally investigated further here.

Semantics. The interpretations of ODL consist of worlds (states) that are first-order structures, associating total functions and relations of appropriate type with function and predicate symbols.

Definition 3 (Interpretation). *An* interpretation I *is a non-empty set of (typed) first-order structures, called* states*, over a signature* Σ *such that: (1) all states have the same interpretation of rigid symbols; (2) the set of states of* I *is closed under the modification (see below) of finitely many non-rigid symbols at finitely many locations; (3) for each type* τ*, all states share the same set* $I(\tau)$ *as the set of objects of type* τ*; the universe of all states is the union of the* $I(\tau)$*; (4) for all types* σ, τ*: if* $\sigma \leq \tau$ *then* $I(\sigma) \subseteq I(\tau)$*; (5)* $\mathtt{obj}_\mathtt{C}$ *is interpreted as a bijection from* $\mathbb{N}$ *into the set of objects having* `C` *as their most-specific type, i.e., that are of type* `C` *but not of any subtype of* `C`*; (6) the interpretations of the* $\mathtt{obj}_\mathtt{C}$ *symbols have disjoint ranges; (7)* `nat` *is interpreted as the set of natural numbers, with operators as usual; (8) the interpretation of the* `Null` *type is a singleton; that of* $\perp$ *is empty.*

In the following, s denotes a state of I and β an assignment of variables, i.e., a mapping from V to the universe of I that respects types. Non-rigid symbols, like program variables or attributes, are allowed to assume different interpretations in different states. Logical variable symbols, however, are rigid in the sense that their value is determined by β alone and does not depend on the state. We use $s[f(e) \mapsto d]$ to denote the *semantic modification* of state s that is identical to s except for the interpretation of the non-rigid symbol f at position e, which is d.

Definition 4 (Valuation of terms and formulas). *For terms and formulas, the* valuation $val_{I,\beta}(s, \cdot)$ *with respect to* I, β, s *is defined as usual for first-order modal logic [10], i.e., using the following definitions for the modal operators:* $val_{I,\beta}(s, [\alpha]\phi) = true$ *iff* $val_{I,\beta}(s', \phi) = true$ *for all* s' *with* $(s, s') \in \rho_{I,\beta}(\alpha)$ *and* $val_{I,\beta}(s, \langle\alpha\rangle\phi) = true$ *iff* $val_{I,\beta}(s', \phi) = true$ *for some* s' *s.t.* $(s, s') \in \rho_{I,\beta}(\alpha)$*.*

With the exception of state updates, the semantics—$\rho_{I,\beta}(\alpha)$—of programs is as customary. In order to demonstrate how concise and simple the ODL language semantics is devised, the full formal definition is provided.

Definition 5 (Semantics of programs). *The* valuation $\rho_{I,\beta}(\alpha)$ of a program α *is a relation on the states of* I. *It specifies which state* s' *(if any) is reachable from a state* s *by executing program* α *and is defined as follows:*

1. $(s, s') \in \rho_{I,\beta}(f_1(t_1^1, \ldots, t_1^{k_1}) := t_1, \ldots, f_n(t_n^1, \ldots, t_n^{k_n}) := t_n)$ *iff* $s = s_0, s' = s_n$, *and* $s_i = s_{i-1}[f_i(val_{I,\beta}(s, t_i^1), \ldots, val_{I,\beta}(s, t_i^{k_i})) \mapsto val_{I,\beta}(s, t_i)]$ *(*$1 \leq i \leq n$*).*
2. $(s, s') \in \rho_{I,\beta}(\alpha; \gamma)$ *iff* $(s, u) \in \rho_{I,\beta}(\alpha)$ *and* $(u, s') \in \rho_{I,\beta}(\gamma)$ *for some state* u.
3. $(s, s') \in \rho_{I,\beta}(\mathtt{if}(\phi)\, \alpha\, \mathtt{else}\, \gamma)$ *iff (1)* $val_{I,\beta}(s, \phi) = true$ *and* $(s, s') \in \rho_{I,\beta}(\alpha)$, *or (2)* $val_{I,\beta}(s, \phi) = false$ *and* $(s, s') \in \rho_{I,\beta}(\gamma)$.
4. $(s, s') \in \rho_{I,\beta}(\mathtt{while}(\phi)\, \alpha)$ *iff there are* $n \in \mathbb{N}$ *and* $s{=}s_0, \ldots, s_n{=}s'$ *such that (1) for* $0 \leq i < n$, $val_{I,\beta}(s_i, \phi) = true$ *and* $(s_i, s_{i+1}) \in \rho_{I,\beta}(\alpha)$, *and (2)* $val_{I,\beta}(s_n, \phi) = false$.

Note that according to this definition, the modifications of a state update are executed simultaneously in the sense that the terms t_i^j, t_i are evaluated in the initial state $s = s_0$. However, if there is a clash, i.e., if two modifications assign different values to the same location, then the rightmost modification wins, which turns out to be more natural for sequential programs than alternative approaches to clash semantics [18]. Like classical dynamic logic, ODL focuses on the input/output behaviour of programs and program parts. Hence it cannot be used to express properties of programs during an infinite run, which would require an extension of ODL to trace semantics (versions of DL with trace semantics are described in [7] for classical DL and in [6] for Java Card).

3 ODL as a Basis for Handling Real-world Languages

In this section, we survey the transformation from real-world object-oriented programs into ODL as a basis for their verification. The particular transformation that we consider here is implemented by schematic inference rules in the KeY deduction engine. It transforms Java Card programs into a sublanguage of Java that corresponds to ODL, except for notation. Experience with KeY in practice shows that the transformation leads to a linear increase in the size of the program and that the time complexity of the transformation is linear in the size of the program. The resulting program retains the structure of the original, as the transformation only locally replaces language features that are not part of ODL. Hence, the relation between Java and ODL programs is easy to grasp for users. Due to space limitations, we have to restrict this presentation to the key ideas enriched with illustrative examples; see [5, 18] for more details.

Type Transformations. As the subtype relation of the class hierarchy is integrated directly, fields and methods undergo a simple translation. An attribute $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ of class ζ is represented as a non-rigid function symbol $f : \zeta \times \sigma_1 \times \ldots \times \sigma_n \to \tau$, which stores at position $(o, a_1, \ldots, a_n)$ the value that field f of object o has at position $(a_1, \ldots, a_n)$ (for array types $n > 0$).

Code Transformations. Most features of current programming languages have a simple uniform transformation into ODL, which accomplishes their effects with more elementary means and without introducing memory or machine models.

Object Creation. Object creation has to support dynamic type-checks, establish object identity and maintain the current type extension. Because of the properties of ODL object enumerators, these demands are fulfilled by translating occurrences of $c := \mathtt{new}\,\mathtt{C}()$ into the state update $c := \mathtt{obj}_{\mathtt{C}}(\mathtt{next}_{\mathtt{C}}),\ \mathtt{next}_{\mathtt{C}} := \mathtt{next}_{\mathtt{C}} + 1$.

Two objects created by distinct invocations of `new` are always different, which is achieved by means of the disjoint bijection constraints on $\mathtt{obj}_{\mathtt{C}}$ and the increment of $\mathtt{next}_{\mathtt{C}}$. Maintaining the extension, i.e., a set of all objects created by program execution so far, is needed in order to express properties ϕ of all objects that have already been created with an invocation of `new`. As $\mathtt{next}_{\mathtt{C}}$ counts the number of objects created of type `C`, this corresponds to: $\forall n\, (n < \mathtt{next}_{\mathtt{C}} \rightarrow \phi(\mathtt{obj}_{\mathtt{C}}(n)))$. Using object enumerators, it is further possible to express dynamic type-checks. For a term t and object-type `C`, we define the type-check formula $t\,\mathtt{instanceof}\,\mathtt{C}$ to be an abbreviation of $\exists n : \mathtt{nat}\ \bigvee_{\mathtt{Null} < \tau \leq \mathtt{C}} t \doteq \mathtt{obj}_{\tau}(n)$.

Despite the static typing of symbols in Σ, ODL needs dynamic type-checks because the *interpretation* of a constant symbol c of (static) type τ in Σ can have any subtype $\sigma \leq \tau$ depending on the current state.

The ODL treatment of object creation is still safe in the presence of garbage collection due to the absence of pointer arithmetics and resource limitations [18]. A further advantage of object enumerators is the simplicity of the contribution of natural numbers—which are already part of ODL for completeness reasons—to object identity without the need to use Skolem symbols for object creation.

Side-effects. Expressions with side-effects can be replaced by a sequence of state updates to temporary program variables, each of which encapsulates one effect of the original expression. Therefore, the order of assignments has to respect the evaluation order constraints of the investigated real-world language. For example, the Java fragment `a[i++] = b-- + b` can be *schematically* translated into an ODL program $vi := i;\ i := i + 1;\ vb := b;\ b := b - 1;\ a(vi) := vb + b$ that does not have side-effects. This ODL program can be condensed to a single parallel update using our simplification rules (see Section 4), which results in $i := i + 1, b := b - 1, a(i) := b + (b - 1)$. ODL updates can be more verbose than side-effecting Java expressions, but they are also more explicit. For the purpose of verification, it is beneficial to have the actual effects readily identifiable.

Exception Handling. Exceptions are not built into ODL, but have to be emulated by preprocessing program transformations. Exception raising can be simulated by introducing appropriate conditions on a (local) program variable that stores the raised exception (which is passed up the call trace when it is not caught). Consider the following example with exception raising and handling:

```
try { while (d != 0)
          {if (d < 0) {throw new RangeEx(d);} else {d=d-1;}}
      /* do something */
```

```
    } catch (RangeEx r) {/* handle range */}
```

It can be transformed into a program that uses exception polling instead:

```
Exception r = null;
while (r == null && d != 0)
  {if (d < 0) {r = new RangeEx(d);} else {d=d-1;}}
if (r == null) {/* do something */}
else if (r instanceof RangeEx) {/* handle range */}
else {return r; /* pass up the call trace */}
```

In favour of a simple logic and calculus, ODL compromises on readability when handling exceptions by program transformation. This is non-crucial in the sense that exceptions are not an inherently object-oriented feature.

The main advantage of banning exceptions and undefinedness from ODL is that no special features like, e.g., a third truth value, have to be introduced to handle partiality. For example, with built-in exception handling, a logic would have to promote the exceptional case of values being `null` throughout the inductive valuation, which clutters both semantics and inference rules. In contrast, ODL just considers `null` as an ordinary—though designated—object. Further, the truth-value of an expression like $c.a \doteq c.a + 2$ is always consistently false, even in the case of $c \doteq \texttt{null}$, whereas $c.a \doteq c.a$ is consistently true.

Dynamic Dispatch. Dynamic dispatch of method calls can be reduced to static method calls by dynamic type-check cascades with `instanceof` along the *reverse* topological order of the type lattice (which also works for multiple inheritance). An important advantage of the ODL way of dynamic dispatch is its simplicity: the basic idea is to implement dispatch “tables” from classical compiler construction technology with ODL primitives. Dynamic dispatch occurs in situations like the one sketched in the following JAVA fragment:

```
class Car { int follow(Car d) {...} }
class Van extends Car { int follow(Car d) {...} }
... return b.follow(d);
```

Having renamed the methods follow that are subject to overriding to Car_follow or Van_follow, respectively, this code snippet is transformed as follows (type casts are expressible in ODL using existential quantification: $\exists v : \mathrm{Van}\ v \doteq b$):

```
class Car { int Car_follow(Car d) {...} }
class Van extends Car { int Van_follow(Car d) {...} }
...  if (b instanceof Van) {return ((Van)b).Van_follow(d);}
else if (b instanceof Car) {return ((Car)b).Car_follow(d);}
else {/* cannot happen when all types are known */}
```

Built-in Operators. From a theoretical perspective, extending ODL by built-in operators is straightforward when assuming a suitable axiomatisation of the operator semantics. For example, modular arithmetic can be axiomatised as [8]: $r \doteq a\ mod\ n \quad \leftrightarrow \quad \exists z : \texttt{nat}\ a \doteq z \cdot n + r \ \land\ r < n.$

Running Example. Consider the following JAVA fragment that illustrates sequence number generation in object database applications and also is a typical part of the implementation of enumeration types in JAVA (sequence numbers are assumed to be multiples of 5, for example):

```
class E { static int g; int id;
          E create() {E r=new E(); r.id=g;g=g+5; return r;}}
```

With return-value r, the method body of create() has the ODL representation $\alpha \;=\; r := \mathtt{obj}_{\mathtt{E}}(\mathtt{next}_{\mathtt{E}}), \mathtt{next}_{\mathtt{E}} := \mathtt{next}_{\mathtt{E}}{+}1;\; r.\mathrm{id} := g;\; g := g+5$ (using JAVA notation for field access). An important property of class E is that sequence numbers in the field id are unique identifiers for E-objects, which is expressed by the global state invariant $\forall x : \mathrm{E}\ \forall y : \mathrm{E}\ (x.\mathrm{id} \doteq y.\mathrm{id} \rightarrow x \doteq y)$. In this context, a typical conjecture is that two objects generated with successive invocations of α have distinct identifiers, which is represented by the ODL formula: $\forall x\, [\alpha](x \doteq r \rightarrow [\alpha]\ (x.\mathrm{id} < r.\mathrm{id}))$.

Discussion. Assignment to non-rigid function symbols cannot be removed from ODL without losing the operational basis for object-oriented programming that permits the change of structured and dynamically typed data or terms.

Likewise, object creation constitutes an essential ingredient to the dynamics of object-oriented systems. Allocating objects at run-time is characteristic of object-oriented programming. With the presence of object enumerator symbols, ODL does not need a native allocation operator. Both the axiomatisation and the translation are convincing and the practical performance achieved with object enumerators is appropriate [18] (similar reasons apply for dynamic dispatch).

4 A Sequent Calculus for ODL

Overview. In this section, we present a sound and (relatively) complete sequent calculus for ODL. The basic idea of the ODL calculus is to perform a symbolic program execution, thereby successively analysing programs and transforming them into logical formulas describing their effects. Yet, rule applications for first-order reasoning and program reasoning are not separated but intertwined.

For first-order and propositional logic standard rule schemata are listed in Table 1, including an integer induction scheme. Within the rules for the program logic part (Table 2), state update rules R29–R30 constitute a peculiarity of ODL and will be discussed after defining rule applications. Essentially, the ODL inference rules have the effect of reducing more complex formulas to simpler ones. Prior to handling loops by R27 or R22, they transform formulas to the normal form $\langle \mathcal{U} \rangle \langle \mathtt{while}(e)\, \alpha \rangle \phi$ or $[\mathcal{U}][\mathtt{while}(e)\, \alpha]\phi$ with some update $\mathcal{U}$. The rules for treating control structures work similar to the case of the WHILE programming language.

Rules of the Calculus. A *sequent* is of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. Its informal semantics is the same as that of $\bigwedge_{\phi \in \Gamma} \phi \;\rightarrow\; \bigvee_{\psi \in \Delta} \psi$.

Table 1. First-order logic part of the ODL calculus.

(R1) $\dfrac{\vdash A}{\neg A \vdash}$	(R4) $\dfrac{A, B \vdash}{A \wedge B \vdash}$	(R7) $\dfrac{A \vdash \quad B \vdash}{A \vee B \vdash}$	(R10) $\dfrac{\vdash A \quad B \vdash}{A \rightarrow B \vdash}$
(R2) $\dfrac{A \vdash}{\vdash \neg A}$	(R5) $\dfrac{\vdash A \quad \vdash B}{\vdash A \wedge B}$	(R8) $\dfrac{\vdash A, B}{\vdash A \vee B}$	(R11) $\dfrac{A \vdash B}{\vdash A \rightarrow B}$
(R3) $\dfrac{\vdash A_x^X}{\vdash \forall x\, A}$	(R6) $\dfrac{A_x^t, \forall x\, A \vdash}{\forall x\, A \vdash}$	(R9) $\dfrac{A_x^X \vdash}{\exists x\, A \vdash}$	(R12) $\dfrac{\vdash A_x^t, \exists x\, A}{\vdash \exists x\, A}$
(R13) $\dfrac{}{A \vdash A}$	(R15) $\dfrac{\Gamma_t^{t'}, t \doteq t' \vdash \Delta_t^{t'}}{\Gamma, t \doteq t' \vdash \Delta}$	(R17) $\dfrac{}{\vdash t \doteq t}$	
(R14) $\dfrac{A \vdash \quad \vdash A}{\vdash}$	(R16) $\dfrac{\Gamma_t^{t'}, t' \doteq t \vdash \Delta_t^{t'}}{\Gamma, t' \doteq t \vdash \Delta}$	(R18) $\dfrac{\vdash \phi(0) \quad \phi(X) \vdash \phi(X+1)}{\vdash \forall n\, \phi(n)}$	

ODL inference rules use substitutions that replace terms (not only variables) by terms and take effect within formulas *and* programs. The result of applying to ϕ the substitution that replaces s by t is defined as usual; it is denoted by ϕ_s^t. Yet, only admissible substitutions are applicable, which is crucial for soundness:

Definition 6 (Admissible substitution). *An application of a substitution θ is* admissible *if no replaced term s occurs (a) in the scope of a quantifier binding a variable of $\theta(s)$ or s, nor (b) in the scope of a modality in which an update to a non-rigid function symbol of $\theta(s)$ or s occurs.*

As common in sequent calculus, although the direction of entailment is from premisses (sequents above bar) to conclusion (sequent below), the order of reasoning is converse in practice. Rules are applied analytically, starting with the proof obligation at the bottom. To highlight the logical essence of inference rules, the ODL calculus provides the rule *schemata* R1–R30 to which the following definition associates the inference rules that are applicable during an ODL proof.

Definition 7 (Rules). *The rule schemata in Tables 1 and 2 induce rules by:*

1. *If $\Phi_1 \vdash \Psi_1 \ldots \Phi_n \vdash \Psi_n \;/\; \Phi \vdash \Psi$ is an instance of one of the rule schemata R1–R26, then*
$$\frac{\Gamma, \langle\mathcal{U}\rangle\Phi_1 \vdash \langle\mathcal{U}\rangle\Psi_1, \Delta \quad \ldots \quad \Gamma, \langle\mathcal{U}\rangle\Phi_n \vdash \langle\mathcal{U}\rangle\Psi_n, \Delta}{\Gamma, \langle\mathcal{U}\rangle\Phi \vdash \langle\mathcal{U}\rangle\Psi, \Delta}$$
is an inference rule of the ODL *calculus, where $\mathcal{U}$ is an arbitrary (or empty) state update, and Γ, Δ are finite sets of context formulas. The formulas within the schemata R19–R22 can occur on either side of the sequent.*
2. *Instances of the rule schemata R27 and R28 can be applied as an inference rule of the* ODL *calculus.*
3. *If (a) $s \rightsquigarrow t$ is an instance of term rewrite rule R29 or R30, (b) $\Phi' \vdash \Psi'$ results from a sequent $\Phi \vdash \Psi$ by substituting t for s, and (c) that substitution is admissible, then the* ODL *calculus contains the rule $\Phi' \vdash \Psi' \;/\; \Phi \vdash \Psi$.*

Table 2. Program logic part of the ODL sequent calculus.

$$\text{(R19)}\ \frac{\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\gamma\rangle\!\rangle\phi}{\langle\!\langle\alpha;\gamma\rangle\!\rangle\phi}$$

$$\text{(R20)}\ \frac{(e\rightarrow\langle\!\langle\alpha\rangle\!\rangle\phi)\wedge(\neg e\rightarrow\langle\!\langle\gamma\rangle\!\rangle\phi)}{\langle\!\langle\mathtt{if}(e)\,\alpha\,\mathtt{else}\,\gamma\rangle\!\rangle\phi}$$

$$\text{(R21)}\ \frac{(e\rightarrow\phi(t))\wedge(\neg e\rightarrow\phi(t'))}{\phi(\mathit{if}\,e\,\mathit{then}\,t\,\mathit{else}\,t'\,\mathit{fi})}$$

$$\text{(R22)}\ \frac{\langle\!\langle\mathtt{if}(e)\,\{\alpha;\mathtt{while}(e)\,\alpha\}\rangle\!\rangle\phi}{\langle\!\langle\mathtt{while}(e)\,\alpha\rangle\!\rangle\phi}$$

$$\text{(R23)}\ \frac{A\vdash B}{\exists x\,A\vdash\exists x\,B}$$

$$\text{(R24)}\ \frac{}{\vdash\mathtt{obj}_{\mathtt{C}}(i)\doteq\mathtt{obj}_{\mathtt{C}}(j)\rightarrow i\doteq j}$$

$$\text{(R25)}\ \frac{}{\vdash\neg(\mathtt{obj}_{\mathtt{C}}(i)\doteq\mathtt{obj}_{\mathtt{D}}(j))}$$

$$\text{(R26)}\ \frac{}{\vdash\forall o\!:\!\mathtt{C}\ (o\,\mathtt{instanceof}\,\mathtt{C}\vee o\doteq\mathtt{null})}$$

$$\text{(R27)}\ \frac{\Gamma\vdash\langle\mathcal{U}\rangle p,\Delta\quad p,e\vdash[\alpha]p\quad p,\neg e\vdash\phi}{\Gamma\vdash\langle\mathcal{U}\rangle[\mathtt{while}(e)\,\alpha]\phi,\Delta}$$

$$\text{(R28)}\ \frac{A\vdash B}{\langle\!\langle\alpha\rangle\!\rangle A\vdash\langle\!\langle\alpha\rangle\!\rangle B}$$

$$\text{(R29)}\ \langle\!\langle\mathcal{U}\rangle\!\rangle f(u)\ \rightsquigarrow$$
$$\mathit{if}\,s_{i_r}\doteq\langle\!\langle\mathcal{U}\rangle\!\rangle u\,\mathit{then}\,t_{i_r}\,\mathit{else}\ldots\mathit{if}\,s_{i_1}\doteq\langle\!\langle\mathcal{U}\rangle\!\rangle u\,\mathit{then}\,t_{i_1}\,\mathit{else}\,f(\langle\!\langle\mathcal{U}\rangle\!\rangle u)\,\mathit{fi}\ldots\mathit{fi}$$
where $i_1<\cdots<i_r$ are all those indices with $f_{i_j}=f$, for some $r\geq 0$

$$\text{(R30)}\ \langle\!\langle\tilde{\mathcal{U}}\rangle\!\rangle\langle\!\langle\mathcal{U}\rangle\!\rangle\phi\ \rightsquigarrow\ \langle\!\langle\tilde{\mathcal{U}},f_1(\langle\tilde{\mathcal{U}}\rangle s_1):=\langle\tilde{\mathcal{U}}\rangle t_1,\ldots,f_n(\langle\tilde{\mathcal{U}}\rangle s_n):=\langle\tilde{\mathcal{U}}\rangle t_n\rangle\!\rangle\phi$$

In the rule schemata, t, t' are terms, X is a new logical variable, $\mathtt{C}\neq\mathtt{D}$ are object-types and $\langle\mathcal{U}\rangle, \langle\tilde{\mathcal{U}}\rangle$ are updates. All substitutions are admissible, in particular the (implicit) substitution that inserts t into $\phi(t)$ must be admissible. In R29 and R30, $\langle\mathcal{U}\rangle$ is of the form $\langle f_1(s_1):=t_1,\ldots,f_n(s_n):=t_n\rangle$, working accordingly for other arities of f. Moreover, in all rule schemata, the schematic modality $\langle\!\langle\cdot\rangle\!\rangle$ can be instantiated with both $[\cdot]$ and $\langle\cdot\rangle$. The same modality instance has to be chosen within a single schema instantiation, though.

It is of utmost importance for soundness that only the rule schemata R1–R26 allow to add an update prefix $\mathcal{U}$ and a sequent context Γ, Δ (case 1 in the above definition), while that is not possible for rule schemata R27 and R28 (case 2) because of their global form of reasoning.

Rule R26 expresses that all objects, except `null`, that will ever exist are generated by object creation expressions. In addition to the standard treatment of equalities, it can be used to discharge proof obligations depending on dynamic types, which typically occur during object-oriented verification. Similarly, R24 and R25 directly express the disjoint bijection restrictions on object enumerators (see Section 3) that are needed to reflect the impact of the type system.

The rewrite schema R29 symbolically executes a state update. Besides promoting the effect of updates to the arguments inductively, R29 basically unfolds changes to the top-level symbol in the order appearing within update $\mathcal{U}$. Thereby, it respects the last-win semantics that ODL uses for clashing updates. In case of a singleton state update $\mathcal{U}$ of the form $f(s) := t$, the rewrite simplifies to $\langle\mathcal{U}\rangle f(u)\ \rightsquigarrow\ \mathit{if}\,s\doteq\langle\mathcal{U}\rangle u\,\mathit{then}\,t\,\mathit{else}\,f(\langle\mathcal{U}\rangle u)\,\mathit{fi}$. The conditional terms introduced

Table 3. Proof of sequence number generation (with $o \equiv \mathtt{obj_E}$ and $n \equiv \mathtt{next_E}$).

$$
\begin{array}{rl}
 & {\scriptstyle R17}\dfrac{*}{X.\mathrm{id} < g, \neg o(n) \doteq X \vdash X.\mathrm{id} < g} \qquad \dfrac{\dots}{X.\mathrm{id} < g, o(n) \doteq X \vdash g < g} \\ \hline
{\scriptstyle R5} & X.\mathrm{id} < g \vdash (\neg o(n) \doteq X \ \rightarrow\ X.\mathrm{id} < g) \ \wedge\ (o(n) \doteq X \ \rightarrow\ g < g) \\ \hline
{\scriptstyle R21} & X.\mathrm{id} < g \vdash (\mathit{if}\, o(n) \doteq X \ \mathit{then}\, g \ \mathit{else}\, X.\mathrm{id}\, \mathit{fi}) < g \\ \hline
{\scriptstyle R29} & X.\mathrm{id} < g \vdash [r := o(n), n := n{+}1, o(n).\mathrm{id} := g, g := g+5]\ (X.\mathrm{id} < r.\mathrm{id}) \\ \hline
{\scriptstyle R30} & X.\mathrm{id} < g \vdash [r := o(n), n := n{+}1, o(n).\mathrm{id} := g][g := g+5]\ (X.\mathrm{id} < r.\mathrm{id}) \\ \hline
{\scriptstyle R30} & X.\mathrm{id} < g \vdash [r := o(n), n := n{+}1][r.\mathrm{id} := g][g := g+5]\ (X.\mathrm{id} < r.\mathrm{id}) \\ \hline
{\scriptstyle R19} & X.\mathrm{id} < g \vdash [r := o(n), n := n{+}1][r.\mathrm{id} := g; g := g+5]\ (X.\mathrm{id} < r.\mathrm{id}) \\ \hline
{\scriptstyle R19} & X.\mathrm{id} < g \vdash [\alpha]\ (X.\mathrm{id} < r.\mathrm{id}) \\ \hline
{\scriptstyle R11} & \qquad\quad \vdash X.\mathrm{id} < g \rightarrow [\alpha]\ (X.\mathrm{id} < r.\mathrm{id}) \\ \hline
{\scriptstyle R3} & \qquad\quad \vdash \forall x{:}\mathrm{E}\ (x.\mathrm{id} < g \rightarrow [\alpha]\ (x.\mathrm{id} < r.\mathrm{id}))
\end{array}
$$

herewith can, in turn, vanish according to schema R21 once the substitution is admissible. Deferring R21 avoids branching until necessary for progress.

The rules R23 and R28, which are required for completeness but are rarely used in practice, characterise a global consequence relation.

Definition 8 (Provability, derivability). *A formula ψ is* provable *from a set Φ of formulas, denoted by $\Phi \vdash_{\mathrm{ODL}} \psi$ iff there is a finite subset $\Phi_0 \subseteq \Phi$ for which the sequent $\Phi_0 \vdash \psi$ is derivable. In turn, a sequent $\Phi \vdash \Psi$ is* derivable *iff there is an inference rule of the* ODL *calculus (Def. 7) with conclusion $\Phi \vdash \Psi$ such that all premisses of the rule are derivable.*

Verification Example. Continuing the example of Section 3, we consider a specification of the body α (with return value r) of the create() method: $\forall x{:}\mathrm{E}\ (x.\mathrm{id} < g \rightarrow [\alpha]\ (x.\mathrm{id} < r.\mathrm{id}))$. On this basis, uniqueness of E-identifiers is due to the fact that create() is the only source for E-objects and that identifiers do not change after object creation (which needs to be proven separately).

Table 3 shows (part of) the proof for the above formula (the right branch remains open). Apart from reducing object creation to object enumerators, the proof essentially consists in update merging and applying the final update $\mathcal{U} = [r := o(n), n := n{+}1, o(n).\mathrm{id} := g, g := g+5]$, which involves rewriting: $[\mathcal{U}]X.\mathrm{id} \rightsquigarrow \mathit{if}\, o(n) \doteq [\mathcal{U}]X \ \mathit{then}\, g \ \mathit{else}\, ([\mathcal{U}]X).\mathrm{id}\, \mathit{fi} \rightsquigarrow \mathit{if}\, o(n) \doteq X \ \mathit{then}\, g \ \mathit{else}\, X.\mathrm{id}\, \mathit{fi}$.

With results about reasoning with created objects [18], the proof can be extended such that the right branch closes as well. That makes use of the fact that X—when it is restricted to objects that have already been created—must differ from the newly created $r{=}o(n)$. This manifests as an additional antecedent $\exists k\, (X{\doteq}o(k) \wedge k{<}n)$, which contradicts $o(n){\doteq}X$ in the right branch using R24.

Soundness and Completeness. With the usual notions of soundness and relative completeness, the ODL calculus is proven sound and a complete extension of first-order arithmetic [18]. Using the proof technique from [11], a central lemma is that all ODL formulas have an equivalent first-order arithmetic formula. This requires Gödelisation of sequences, which is more complicated in the presence of non-rigid functions of finite but unbounded change.

Theorem 1 (Soundness and relative completeness). *(1) The* ODL *calculus (Def. 8) is* sound*, i.e., derivable formulas are valid (true in all states of all interpretations).*

(2) The ODL *calculus is* complete *with respect to first-order arithmetic, i.e., if an* ODL *formula is valid, then it can be derived from a set of tautologies of first-order arithmetic.*

Moreover, we have shown that relative completeness is preserved for conservative extensions of ODL with language features that so-called locally equivalent inference rules can reduce to original ODL [18].

Example 1 (Relatively complete coverage of `for` *loops).* Adding to ODL the rule "$\vdash \langle \mathcal{U}; \texttt{while}(\chi)\,\{\alpha;\gamma\}\rangle\phi \;/\; \vdash \langle \texttt{for}(\mathcal{U};\chi;\gamma)\alpha\rangle\phi$" yields a calculus for ODL extended with `for` loops that is complete w.r.t. first-order arithmetic. Similarly, constructor calls and side-effecting expression evaluation can be added to ODL without loss of relative completeness.

5 Conclusions and Future Work

We have introduced a dynamic logic, ODL, with non-rigid functions, and presented a sound and relatively complete calculus. The conceptual design of the logic ODL is guided by the ambition to capture the essence of reasoning for a coherent basis of object-oriented verification at an adequate level of abstraction.

ODL provides dynamically typed object enumerators and state updates, i.e., operations to change the interpretation of non-rigid function symbols. State updates work in parallel for multiple pointwise changes at once. With these extensions, notions of object-orientation can be embedded in ODL.

The ODL calculus is based on a classical sequent calculus for the While programming language [11]. In order to deal with function modification, rewrite rules have been introduced that promote the effect of a state update throughout the affected formula, with case distinctions to resolve potential aliasing. State update applications can be delayed to defer branching of the proof.

The completeness proof for our ODL calculus in [18] has revealed and fixed a flaw in the classical completeness proofs for dynamic logic (for While) [11, 7] concerning the treatment of multiple variables.

Future work includes a closer investigation of the pragmatic effects of the ODL approach to software verification. It is useful to build a modular set of verification components for object-oriented calculi by providing add-on inference rules for additional language features on the basis of the extension theorem in [18]. An investigation of the impact of parametric genericity for the type system seems worthwhile to a similar degree.

To sum up, the feasibility of defining an insightful essentials-only verification calculus for object-oriented programming, which is sound and complete relative to classical first-order arithmetic, has been demonstrated.

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97*, volume 1214. Springer-Verlag, 1997.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [4].
4. G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *CASSIS 2004, Revised Selected Papers*, volume 3362 of *LNCS*. Springer, 2005.
5. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of *LNCS*, pages 6–24, 2001.
6. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In *FASE'03*, LNCS. Springer, 2003.
7. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 626–641. Springer, 2001.
8. B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In E. A. Boiten, J. Derrick, and G. Smith, editors, *IFM*, volume 2999 of *LNCS*, pages 207–226. Springer, 2004.
9. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Barthe et al. [4], pages 108–128.
10. M. Fitting and R. L. Mendelsohn. *First-Order Modal Logic*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.
11. D. Harel. *First-Order Dynamic Logic*. Springer-Verlag, New York, 1979.
12. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
13. B. Jacobs and E. Poll. A logic for the Java modeling language JML. In *FASE '01*, pages 284–299, London, UK, 2001. Springer-Verlag.
14. R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *ECOOP*, pages 85–103, 1997.
15. W. Mostowski. *Formal Development of Safe and Secure Java Card Applets*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, February 2005.
16. T. Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In *Proc. Marktoberdorf Summer School*, 2003.
17. C. Pierik and F. S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS*, volume 2884 of *LNCS*, pages 64–78. Springer, 2003.
18. A. Platzer. An object-oriented dynamic logic with updates. Master's thesis, University of Karlsruhe, September 2004. Available at `www.key-project.org`.
19. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, volume 1576 of *LNCS*. Springer, 1999.
20. R. Stärk and S. Nanchen. A logic for abstract state machines. *J. UCS*, 7(11), 2001.
21. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. D. Mosses, editors, *WADT*, volume 1827 of *LNCS*, pages 1–21. Springer, 1999.
22. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *FME*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.