ABSTRACT OBJECT CREATION IN FIRST-ORDER DYNAMIC LOGIC: STATE OF THE ART IN KeY

STIJN DE GOUW, FRANK S. DE BOER, RICHARD BUBEL, AND WOLFGANG AHRENDT

CWI, Amsterdam and Leiden University, The Netherlands
e-mail address: cdegouw@cwi.nl

CWI, Amsterdam and Leiden University, The Netherlands
e-mail address: frb@cwi.nl

Technische Universität Darmstadt, Germany
e-mail address: bubel@cs.tu-darmstadt.de

Chalmers University, Göteborg, Sweden
e-mail address: ahrendt@chalmers.se

Abstract. We present the state of the art in the KeY Java theorem prover, which is based on a first-order object-oriented dynamic logic. This assertion language allows both specifying and verifying properties of objects at the abstraction level of the programming language, abstracting from a specific implementation of object creation. Objects which are not (yet) created never play any role. Corresponding proof theory is discussed and justified formally by soundness theorems. The usage of the assertion language and the proof rules is illustrated by means of an example of a reachability property of a linked list. All proof rules presented are fully implemented in a special version of KeY.

1. Introduction

In object-oriented programming languages like Java, objects as first-class citizens in the domain of values introduce a general mechanism of indirection. This high-level mechanism of indirection abstracts from the underlying representation of objects and the implementation of object creation. At the abstraction level of the programming language, objects are described as instances of their classes, i.e., the classes provide the only user-defined operations (i.e., methods) which can be performed on objects. The Java language itself provides only the following built-in operations on objects:

1998 ACM Subject Classification: F.3.1 [Logics and Meanings of Programs], D.2.4 [Software/Program Verification].

Key words and phrases: specification, verification, program logic, dynamic logic, object creation.

This author is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models .

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

Object Operators in Java	
"instanceof"	to test if an object is of a specified type
"new"	to create a new instance of a specified type
"."	to obtain the value of the specified field of the specified object
"=="	to test equality between object references
"!="	to test inequality between object references
"?:"	conditional expression
"(Type)"	cast operator to covert the type of the specified object

Moreover, these operations can only be performed on the created objects, the objects not (yet) created do not exist and therefore can also not be referred to by *any* programming construct. This ensures memory safety, relieves the programmer of (error-prone) manual memory management, helps portability and allows compiler optimizations to freely move objects in memory. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language, following Wittgenstein:

Whereof one cannot speak, thereof one must be silent.

In [18] a Hoare logic is presented to verify properties of an object-oriented programming language at the abstraction level of the programming language itself. This Hoare logic is based on a weakest precondition calculus for object creation which abstracts from the implementation of object creation. This abstraction requires new techniques for computing the weakest precondition of object creation statements because in the state *prior* to the creation of the object this new object does *not* exist so that we cannot refer to it. Note that thus we cannot obtain simply the weakest precondition as usual by substitution in the post-condition of the variable to which the new object is assigned because there does not exist a term for it in the state prior to the creation of the object! Moreover, because of the abstraction level in the assertion language, *quantification* over objects only involves *created* objects. Consequently, the *scope* of the quantifiers is also affected by object creation which has therefore to be taken into account by the weakest precondition calculus.

The main contribution of this paper is a new formalization of a weakest precondition calculus for abstract object creation in dynamic logic. This logic allows the specification and verification of object-oriented programs at an abstraction level which coincides with that of the object-oriented programming language Java. That is, besides the above operations on objects the logic only supports quantification over created objects, including array objects. Consequently, objects not (yet) created cannot not be referred to by *any* construct in the logic. We show that the standard *first-order* sequent calculus of this logic which forms the basis of the KeY theorem prover [10] adequately captures the abstract data type of objects. Further, we extend the dynamic logic with auxiliary variables. Since arrays in Java are also objects this indirection allows for the first-order specification of properties of object structures which cannot be expressed directly in first-order logic, like reachability.

In order to extend the initial work in [3] to classes, recursive methods, arrays, dynamic binding and failures, we define in this paper a core object-oriented language with support for recursive methods and arrays, and show how more intricate features such as failures and dynamic binding (which are not part of this core language) can be handled by a completely mechanical transformation. None of the transformations require changes in the proof rules

for the core language, indicating that first, the core language is chosen appropriately, and second, that our approach scales up to modern object oriented languages.

We discuss tool support - all proof rules presented in this paper have been fully implemented in a special version of KeY- by means of an example of a linked list. Finally, we show how to symbolically execute abstract object creation in KeY.

Related work. Specification languages like the Java Modeling Language (JML) [28] and the Object Constraint Language (OCL) [33] also abstract from the underlying representation of objects. To the best of our knowledge in contrast all known tools for the (deductive) verification of object-oriented programs are based on some explicit representation of objects, e.g., objects are represented by natural numbers and counters are used to model object creation. Such an explicit representation can be useful in the context of programming languages which support for example pointer arithmetic but it does not comply with the abstract data type of objects as provided by object-oriented languages like Java. Further, we show in this paper that also the verification engine itself does not require such an explicit (internal) representation.

Pure first-order logic without auxiliary variables is not expressive enough to assert for example reachability properties. For example, in [5] a formalization of abstract object creation is given using inductively defined predicates (so-called *pure* methods). Such predicates are typically used in specifying the footprint of a program, which is crucial in tools based on separation logic (VeriFAST [24], jStar [19], Slayer [14] and Smallfoot [13]) to facilitate local reasoning. Another approach is taken by Why3 [21], PVS [38] and Isabelle [26], which generate verification conditions in standard higher-order logic (without introducing additional logical connectives such as the separating conjunction and points-to operator of separation logic). However in general higher-order logic seriously complicates proof theory (in contrast to first-order logic, the validities are not recursively enumerable) and thus automation.

Chalice [29, 31] is a programming language for the specification and verification of multi-threaded programs. The actual verification is done by translation to Boogie which allows the use of the available Boogie tool support. In a multi-threaded environment it is important to restrict access of threads on heap areas in order to avoid data races and similar. Chalice addresses the problem as follows: If a method wants to access (read or write) to a heap location it must require access to these locations via its precondition and grant access to new locations resp. "release" access via its post-condition. If a method tries to read or write to a location it did not explicitly ask for permission, the verification will fail. Chalice supports fractioned permission, i.e., permissions can have a value from 0 to 1 (the sum of all permissions must be equal to 1. Read access is granted for any permission greater than 0, while write access requires a permission of 1). Framing of a method is now treated by computing a safe approximation of the permissions the method has acquired.

In Chalice reasoning about linked data structures is done via data abstraction, i.e., the internal linked data structure is mapped via model fields/ghost fields to a suitable abstract data type. The primary focus of Chalice focus is on concurrency properties like absence of data-races, dead-locks, etc.

Concerning object creation, Chalice uses underneath object activation, though certain aspects have been abstracted away like the order of activation. In contrast to our approach, their domain stays constant and object creation is modelled as follows: The semantics maintains a partial function Ω which maps object references to environment lists. An environment is a message and an object is called allocated if it is in the domain of Ω. On

the semantics level, they have later on always to quantify over the objects which are in the domain, i.e., $\forall o.(o \in dom(\Omega) \rightarrow \phi)$, to assert that a property ϕ holds for all allocated objects. These guards are not necessary in our approach as unallocated objects do not exist at all and thus cannot be quantified over in the first sense.

In recent years a number of verification competitions have been held where most of the current state-of-the-art tools participated. The general structure of the competitions was to provide a number of assignments with each assignment consisting of

- a description of the algorithm in pseudo code or a general problem description which had to be implemented in a programming language of the teams choice.
- a natural language description of the desired properties to be specified and verified.

The focus of the competitions was slightly different. While the emphasis for VSTTE'11 [25] and FOVEOOS'11 [15] was on verifying program correctness, VSTTE'12 moved the focus more towards algorithm verification. The competitions showed that verifiers using abstract programming languages did best as they could use abstract data types directly in the programming language or at least could map linked data structures easier to abstract data types than systems for languages like Java. KIV which won VSTTE'12 (together with ACL2) is also based on dynamic logic like KeY, but is based on abstract data types. They provide support for several languages including Java and an ASM style programming languages. The Java backend was e.g. used only for the first problem in VSTTE'12 (which was also solved by KeY) for the remaining problems (complex heap structures) the more abstract ASM style language was used. Another tool that did very well was Dafny [30] which uses its own (more abstract) object-oriented language and whose specification and reasoning system implements dynamic frames. KeY was competitive in VSTTE'11 and FOVEOOS'11, but did not so well in VSTTE'12 due to its heavier focus on algorithm verification. KeY was able to prove all problems of the first two competitions after the competition using a developer version for KeY 2.0 with explicit support for dynamic frames with that version also (to the best of our knowledge) most (except one) problems of VSTTE'12 could be solved. The only time a system based on separation logic participated was VeriFAST [24] at VSTTE'11. Their specification of some of the problems was elegant, but automation was problematic. They managed to solve the problems in the aftermath, but indicate that labor-intensive work was necessary.

As the main contribution of our work, we show that KeY extended with an abstract data type of objects and auxiliary variables allows both the first-order specification *and* verification of Java programs at the abstraction level of the programming language. For the verification, dynamic logic is used as a systematic formalization of *abstract* object creation, and generator of first-order verification conditions. KeY itself is one of the state-of-the-art verification tools. To the best of our knowledge our extension of KeY is a first tool which supports abstract object creation as described above. A detailed description of other tools is beyond the scope of this paper, however of interest is the following summary of the results of recent competitions.

Our basic approach to abstract object creation provides a solid basis for the integration and extension of many other important proof-theoretical object-oriented concepts like invariants [9, 23], modularity [27], dynamic frames [36], and behavioral subtyping [4, 32]. Further in [2] our basic approach to abstract object creation has been integrated in a proof theory of multi-threaded programs.

Outline. In Section 2 we introduce a dynamic logic for an object-oriented language with recursive methods and object creation. Other features are not part of this core language but in Section 3 we demonstrate how to handle them using a transformational approach. We present the axiomatization of the language in terms of the sequent calculus given in Section 4. To reason about formulas containing updates (which arise from assignments and object creations in the sequent calculus), we define in Section 5 an inductive rewrite relation which simplifies such formulas to standard first-order logic formulas (without updates). We illustrate our approach on a typical case: inserting an element in a queue in Section 6. With the calculus at hand, symbolic execution of programs is described in Section 7. After a discussion of the expressiveness of our approach in Section 8, we conclude with Section 9.

2. Dynamic Logic

Dynamic logic (DL) is a variant of *modal logic* which extends and generalizes Hoare logic, e.g., it allows the direct expression of program equivalence and weakest preconditions. Different parts of a formula are evaluated in different worlds (models), which vary in the interpretation of, in our case, program variables and fields. DL extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *statement*, whereas the second argument is another DL formula. A formula $\langle p\rangle\phi$ is true in a model M if execution of p terminates when started in M and results in a model where ϕ is true. As for the other operator, a formula $[p]\phi$ is true in a model M if execution of p, when started in M, *either* does not terminate *or* results in a model in which ϕ is true. In other words, the difference between the operators is the one between total and partial correctness.[1] DL is closed under all logical connectives. For instance, the formula $\forall\, l.\,(\langle p\rangle\ (l = u)\ \leftrightarrow\ \langle q\rangle\ (l = u))$ states equivalence of p and q w.r.t. the program variable u.

An example formula involving object creation is $\forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$. It states that every new object indeed is new because the logical variable l ranges over all the objects that exist *before* the object creation $u := \mathsf{new}$. Consequently, after the execution of $u := \mathsf{new}$ we have that the new object is not equal to any object that already existed before, i.e., $\neg(u = l)$, when l refers to an "old" object. Note that the formula $\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$ has a completely different meaning. In fact the formula is false (cf. Section 5.2). These examples also illustrate a further advantage of DL over Hoare logic: in the presence of abstract creation it allows for a direct logical expression of the dynamic scope of the object quantifiers (as illustrated above).

All major program logics (Hoare logic, wp calculus, DL) have in common that the resolving of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having *explicit substitutions* in the logic, called *'updates'*. In this paper, elementary updates have the form $u := \mathsf{new}$, $u := e$, $e_1.x := e_2$, or $e_1[e_2] := e_3$. Updates are brought into the logic via the update modality $\{\,.\,\}\,.\,$, connecting arbitrary updates with arbitrary formulas, like in $0 < v \rightarrow \{u := v\}\, 0 < u$.

A full account of KeY style DL is found in [11].

[1] Just as in standard modal logic, the diamond resp. box operators quantify existentially resp. universally over models (reached by the program). In case of deterministic programs, however, the only difference between the two is whether termination is claimed or not.

2.1. **Language.** We introduce here a core class-based object oriented language with support for recursive methods, unbounded arrays and object creation. A corresponding assertion language is presented which is based on first-order dynamic logic.

2.1.1. *Types and declarations.* The language is strongly typed and contains the primitive types Nat, Boolean. Additionally there are user-defined class types C, predefined class types $T[\,]$ of *unbounded* arrays in which the elements are of type T, and a union type Object. Arrays are dynamically allocated and are indexed by natural numbers. Multi-dimensional arrays are modeled (as in Java) as arrays in which the elements themselves are arrays. For instance, Nat[ ] is the type of one dimensional arrays of natural numbers and Nat[ ][ ] is the type of two-dimensional arrays of natural numbers. We assume a transitive reflexive subtype relation between types. Object is the supertype of any class type. We will not be concerned with type checking, but only note that it can be done statically, and is orthogonal to the semantics of the language.

There are three kinds of declarations: variable declarations, field declarations and method declarations. A variable declaration associates a type to a name. If the type is not an array type, we call the variable a simple variable. Arrays are assumed to be unbounded but we show how bounded arrays can be handled by a transformation in Section 3.

Declarations of fields of a class C associate a type C $\to$ T to a field name. Fields can thus be seen as mappings from objects of C to values.

A method declaration $m(\mathsf{this}, u_1, \ldots, u_n) :: s$ associates a list of variables $\mathsf{this}, u_1, ..., u_n$ (its *parameters*) and a statement s (its *body*) to a method name m. Unlike in Java, methods do not return anything. Return values can be simulated using variables. The first formal parameter of each method is the special variable this. It stores the currently executing object and is special since it is read-only (assignments to this are not allowed).

We do not represent declarations explicitly in the language syntax we are about to introduce. Instead we assume to be given a set Var of variable declarations, and for each class C a set F_{C} of fields and M_{C} of methods. Var is partitioned into a set PVar of program variables and a set LVar of logical variables. Logical variables do not occur in programs. They are used in dynamic logic formulas to express properties of programs and can be quantified over. The set of program variables consist of both local and global variables. For technical convenience we restrict local variables to formal parameters (for a treatment of blocks we refer to [8]).

2.1.2. *Syntax.* Expressions of our language are side-effect free. The following grammar generates the language of expressions:

$$\begin{aligned} e \quad ::= \quad & u \mid e.x \mid \mathsf{null} \mid e_1 = e_2 \mid \mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi} \mid \\ & \mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi} \mid e_1[e_2] \mid f(e_1, ..., e_n) \mid C(e) \end{aligned}$$

Variables are indicated by the typical element u, and x is a typical field. Since u and x can be of an array type, this implies that arrays are also expressions. A dot denotes dereferencing, i.e., $e.x$ is the value of the field x of the object to which e points. This is syntactic sugar for $x(e)$ since fields will be considered to be functions from objects to values (this reflects the fact that field types have the form C $\to$ T). The expression null denotes the undefined reference. The Boolean expression $e_1 = e_2$ denotes the test for equality between the values of the expressions e_1 and e_2. For object expressions this means we use the Java reference semantics: the expressions must denote the same object identity. It

is therefore possible for two expressions of type object to be unequal, even if all fields of the objects they denote have the same value. The expression $\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi}$ has value e if the Boolean expression b is true, otherwise it has an arbitrary value. This expression allows a systematic approach to proving properties about partial functions. A conditional expression is denoted by $\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi}$. The motivation for including it in our core language is that it significantly simplifies treatment of failures (Section 3) and aliasing (Section 5). If e_1 is an expression of type $T[\,]$ and e_2 is an expression of type Nat then $e_1[e_2]$ is an expression of type T. Here T itself can be an array type. For example, if a is an array variable of type Nat[ ][ ] then the expression $a[0]$ denotes an array of type Nat[ ]. The function $f(e_1, ..., e_n)$ denotes an arithmetic or Boolean operation of arity n. For class types C the Boolean expression $C(e)$ is true if and only if the (dynamic) type of e is exactly C. Expressions of a class type can only be compared for equality, dereferenced, accessed as an array if the object is of an array type, or appear as arguments of a class predicate, if-expression, or conditional expression.

The language of statements is generated by the following grammar:

$$\begin{aligned} s \quad ::= \quad & \mathsf{skip} \mid s_1; s_2 \mid \mathsf{if}\, b\, \mathsf{then}\, s_2\, \mathsf{else}\, s_3\, \mathsf{fi} \mid \mathsf{while}\ \ e\ \ \mathsf{do}\ \ s\ \ \mathsf{od} \mid \mathsf{abort} \mid \\ & m(e_1, ..., e_n) \mid u := \mathsf{new} \mid u := e \mid e_1[e_2] := e \mid e_1.x := e \end{aligned}$$

By skip we denote the empty statement. A semicolon denotes sequential composition. Conditional branching is denoted by if–then–else–fi. The statement while denotes the usual looping. The condition for both looping and branching is given by a Boolean expression. The abort statement causes a failure. In a method call $m(e_1, ..., e_n)$ the first actual parameter e_1 denotes the callee. Method calls provide the only way to transfer control from the current object to another (the callee). A statement $u := \mathsf{new}$ assigns to the program variable u a newly created object of the declared type (possibly an array type) of u. Objects are never destroyed. The next three statements denote assignments of an expression e to respectively a program variable u, a *subscripted* variable $e_1[e_2]$ (where e_1 must have an array type and e_2 a natural number) and a field x. For technical convenience only we do not have assignments $e_1[e_2] := \mathsf{new}$ and $e_1.x := \mathsf{new}$. We reason about such assignments in terms of the statement $u := \mathsf{new}; e_1[e_2] := u$, where u is a fresh program variable, to separate object creation from the aliasing problem. Following standard practice, assignments are well-typed when the type of the value is a subtype of the type of the variable to which it is assigned. We assume every statement and expression to be well-typed. A *program* in our language consists of a statement (referred to as the main statement) together with sets for variable declarations, field declarations and method declarations.

Dynamic logic formulas are built from the Boolean expressions and statements defined above. Formally formulas are generated by the following grammar:

$$\phi \quad ::= \quad b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi \rightarrow \phi_2 \mid \exists l : \phi \mid \forall l : \phi \mid \{U\}\phi \mid [s]\phi \mid \langle s \rangle \phi$$

In this grammar, b is a Boolean expression, s is a statement, U is an update and l is a logical variable of any type of our core language. The meaning of such formulas was described informally already in the beginning of this section and will be formalized below.

Conventions. We use $\equiv$ for *syntactic* equality. For expressions e and statements s, $var(e)$ and $var(s)$ denote the sets of variables that appear in them. This set is extended to sets of method declarations in a pointwise manner: for a set of method declarations M_{C}, $var(M_{\mathrm{C}})$ is the set of all variables that appear in some method body of the methods in M_{C}. The set change(s) contains the variables that appear on the left-hand side of an assignment in s. For

a method declaration $m(u_1, \ldots, u_n) :: s$, change(m) is defined as change(s)\{$\mathsf{this}, u_1, ..., u_n$}. This reflects the fact that the formal parameters are not changed by method calls, since they are reset to their initial values. For a set M_{C} of method declarations, change(M_{C}) is the union of all sets change(m) with $m \in M_{\mathrm{C}}$. We abbreviate the set of all method declarations of all classes by D and extend *var* and *change* to D in the obvious way. The set $free(\phi)$ contains all variables that occur free in the formula ϕ.

2.1.3. *Semantics.* Before the semantics of statements and expressions of our programming language can be defined, a meaning must be assigned to the non-logical symbols of our language. For reference we list all non-logical symbols here:

(1) The constants (functions of arity 0): $\mathsf{null}_{\mathrm{C}}$ (of type C), 0 (Nat) and true and false (Boolean).
(2) Arithmetical operations. We assume at least the successor function, addition and multiplication.
(3) Boolean operations (at least conjunction, disjunction and negation).
(4) For each type T the conditional $(\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi})_T$.
(5) For each array type $T[\,]$ an access function $[\,]_{T[\,]}$ of type Nat $\to$ (T[ ] $\to$ T).
(6) For every class C a unary predicate C of type Object $\to$ Boolean.
(7) Variables declared in Var.
(8) For every class C, its fields declared in F_{C}.

The basic notion underlying the semantics of both the programming language and the assertion language is that of a many-sorted structure of the form

$$(dom(\mathrm{Nat}),\, dom(\mathrm{Boolean}),\, dom(\mathrm{C}_1),\, \ldots,\, dom(\mathrm{C}_n),\, I)$$

with disjoint domains $dom(T)$ for each type T and an interpretation I. The interpretation assigns a function (a meaning) to each function symbol, and a relation to each relation symbol. Thus for example for the constant $\mathsf{null}_{\mathrm{C}}$, the interpretation assigns an element of dom(C) to it. We omit explicit type annotations of null and the conditional expression if no confusion occurs. What we call a model is sometimes called a model plus a valuation (i.e. an assignment of the variables).

A *model* M for our language is a structure that satisfies the axioms:

- Arithmetical operations satisfy the theory of Peano arithmetic.
- Boolean operations satisfy the axioms for Boolean algebras.
- Each class predicate C satisfies C(null).
- The conditional $(\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi})_{\mathrm{Boolean}}$ is equivalent with $(b \wedge e_1) \vee (\neg b \wedge e_2)$, and if-then satisfies $\mathsf{if}\, true\, \mathsf{then}\, b\, \mathsf{fi}_{\mathrm{Boolean}} \leftrightarrow b$.
- Let *op* be any n-ary function symbol or relation symbol of our language with resulting type T' (for relation symbols the result has type Boolean). Then the conditional $(\mathsf{if}\, b\, \mathsf{then}\, e_i\, \mathsf{else}\, e_{i'}\, \mathsf{fi})_T$ satisfies
  $op(e_1, \ldots, (\mathsf{if}\, b\, \mathsf{then}\, e_i\, \mathsf{else}\, e_{i'}\, \mathsf{fi})_T, \ldots, e_n) =$
  $(\mathsf{if}\, b\, \mathsf{then}\, op(e_1, \ldots, e_i, \ldots, e_n)\, \mathsf{else}\, op(e_1, \ldots, e'_i, \ldots, e_n)\, \mathsf{fi})_{T'}$.
  Similarly, $\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi}_T$ satisfies
  $op(e_1, \ldots, (\mathsf{if}\, b\, \mathsf{then}\, e_i\, \mathsf{fi})_T, \ldots, e_n) = (\mathsf{if}\, b\, \mathsf{then}\, op(e_1, \ldots, e_i, \ldots, e_n)\, \mathsf{fi}_{T'}$.

The above first-order axioms are not categorical: there are multiple structures in which all of the above axioms are true. We stipulate that *any* structure that satisfies the axioms is a model for our language, including those structures in which for example the arithmetical

operations have a non-standard interpretation. For notational convenience we write $M(s)$ instead of $I(s)$ for the interpretation of the non-logical symbol s in the model M, and $M(T)$ for the domain $dom(T)$ in M. If u is declared in Var as a variable of type T, it is interpreted as an individual of the domain $M(T)$. A field $x \in F_{\mathrm{C}}$ of type $\mathrm{C} \to \mathrm{T}$ is interpreted as a unary function $M(\mathrm{C}) \to \mathrm{M(T)}$. Array access functions $[\ ]_{T[\ ]}$ are interpreted as binary functions $M(\mathrm{Nat}) \to (\mathrm{M(T[\ ]}) \to \mathrm{M(T)})$. Thus array indices can be seen as fields.

Semantics of Expressions and Statements. The meaning of an expression e of type T is a (total) function $[\![e]\!]$ that maps a model M to an individual of $M(T)$. This function is defined by induction on e. Here are the main cases:

- $e \equiv e_1.x$: $[\![e]\!](M) = M(x)([\![e_1]\!](M))$.
- $e \equiv e_1[e_2]$: $[\![e]\!](M) = M([\ ]_{T[\ ]})([\![e_2]\!](M))([\![e_1]\!](M))$
  where e_1 has the array type $T[\ ]$ and e_2 has type Nat.
- $e \equiv \mathrm{C(e_1)}$: $[\![e]\!](M) = \mathsf{true}$ iff $[\![e_1]\!](M) \in M(\mathrm{C})$
- $e \equiv \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$:

$$[\![e]\!](M) = \begin{cases} [\![e_1]\!](M) & \text{if } [\![b]\!](M) = \mathsf{true} \\ [\![e_2]\!](M) & \text{if } [\![b]\!](M) = \mathsf{false} \end{cases}$$

It is easily checked that this semantics satisfies the axioms for models. One feature of our semantics is that since the meaning function is total, some meaning is also assigned to the expression $\mathsf{null}.x$. However, in the execution of programs their meaning is given operationally by executing the abort statement. Section 3 describes a general treatment of failures, including dereferencing null-pointers in programs.

Statements in our language are deterministic and can fail (abort) or diverge. We define the meaning of a statement in terms of a big-step operational semantics, and use the (quite common) notation

$$\langle s, M\rangle \to M'$$

to express that executing s in the model M, terminates in the model M'. Since calls can appear in statements, the definition of the above transition relation depends in general on the method declarations. We do not distinguish failures from divergence semantically, in either case there is no M' such that $\langle s, M\rangle \to M'$. However in Section 3 we show how to deal with failures by program transformation. Throughout execution, assignments to variables and fields change the model in the interpretation of the variables and fields respectively. The interpretation of the array access function changes due to assignments to subscripted variables. Moreover during executing, the domain $dom(\mathrm{C})$ for instances of C is extended with new objects by object creations $u := \mathsf{new}$. Statements do not affect the domains of natural number and Booleans, and the interpretation of the other non-logical symbols. In summary, statements map models to models, potentially changing variables, array access functions, fields and the domains for class types, leaving fixed the interpretation of the other functions and relations.

The meaning of normal assignments, conditional statements and while loops is defined in the standard way. Therefore we focus on the semantics of object creation, field assignments and method calls. Since object creation involves initialization of the fields of the new object we first define for each type a default value: $init_{\mathrm{Nat}} = 0, init_{\mathrm{Boolean}} = \mathsf{false}$ and $init_{\mathrm{C}} = \mathsf{null}$. Furthermore, for the selection of a new object of class C we use a choice function ν on a model M and class C to get a fresh object $\nu(M, \mathrm{C})$ of class C which satisfies $\nu(M, \mathrm{C}) \notin \mathrm{M(T)}$ for any type T (in particular, $\nu(M, \mathrm{C}) \notin \mathrm{M(C)}$). Clearly, without

loss of generality we may assume that $\nu(M, \mathrm{C})$ only depends on $M(\mathrm{C})$ in the sense that $\nu(M, \mathrm{C}) = \nu(\mathrm{M}', \mathrm{C})$ if $M(\mathrm{C}) = \mathrm{M}'(\mathrm{C})$. It is worthwhile to observe that this choice function preserves the deterministic nature of our core language. Non-deterministic (or random) selection of a fresh object would require reasoning semantically up to a notion of isomorphic models which would unnecessarily complicate soundess proofs. The The semantics of object creations can now be defined as follows:

$$\langle u := \mathsf{new}, M\rangle \to M'$$

where u has type C and M' satisfies

(1) $o = \nu(M, \mathrm{C})$.
(2) $M'(\mathrm{C}) = \mathrm{M}(\mathrm{C}) \cup \{\mathrm{o}\}$.
(3) $M'(x)(o) = \mathit{init}_T$ for any field x of type $\mathrm{C} \to \mathrm{T}$.
(4) $M'(u) = o$.
(5) The domains other than C and the interpretations of the other non-logical symbols in M' are the same as in M.

This semantics for object creation corresponds to the intuition that an object creation first adds a new object to the domain for the class C, then sets the values of the fields in the new object to their default value and finally assigns the new object to u. Similarly, the semantics of an array creation is modeled by:

$$\langle u := \mathsf{new}, M\rangle \to M'$$

where u has type $T[\,]$ and M' satisfies:

(1) $o = \nu(M, T[\,])$.
(2) $M'(T[\,]) = M(T[\,]) \cup \{o\}$.
(3) $M'([\,]_{T[\,]})(n)(o) = \mathit{init}_T$ for all $n \in M(\mathrm{Nat})$.
(4) $M'(u) = o$.
(5) The domains other than C and the interpretations of the other non-logical symbols in M' are the same as in M.

The third clause states that all elements in the array are initialized to their default value. Assignments to fields are executed as follows:

$$\langle e.x := e_1, M\rangle \to M'$$

where $M'(x)([\![e]\!](M)) = [\![e_1]\!](M)$ and the domains and the interpretations of the other non-logical symbols in M' are the same as in M. Assignments to subscripted variables are defined analogously and therefore omitted.

For method calls we use a copy rule: whenever a call is encountered, the program proceeds by executing the method body. The values of the actual parameters are evaluated in parallel and assigned to the formal parameters before the method body executes. Directly after the call succeeds, the formal parameters are restored to their initial values. This leads to the following rule:

$$\frac{\langle s, M'\rangle \to M''}{\langle m(e_1, \ldots, e_{n+1}), M\rangle \to M'''}$$

where the method m is declared as $m(\mathsf{this}, u_1, \ldots, u_n) :: s$,
M' differs from M only as follows:
$M'(\mathsf{this}) = [\![e_1]\!](M)$ and $M'(u_i) = [\![e_{i+1}]\!](M)$ for $1 \le i \le n$,

and M''' differs from M'' only as follows:
$M'''(\mathsf{this}) = M(\mathsf{this})$ and $M'''(u_i) = M(u_i)$ for $1 \leq i \leq n$.

Intuitively this corresponds to defining the operational meaning of calls in three steps: in the first step M' is obtained by assigning the actual parameters to the formal parameters. In the second step, the actual method body is executed in M', resulting in M''. In the final step, M''' is obtained by resetting the formal parameters to their initial values.

Semantics of Formulas. Since calls may occur inside the modal operators of a dynamic logic formula, the truth of a dynamic logic formula ϕ depends in general on a set of method declarations to bind calls to the right method body. Given a set of method declarations, we write $M \models \phi$ if the formula ϕ is true in the model M. A formula ϕ is valid if $M \models \phi$ holds for every model M. Interestingly, even though we allow quantification over arrays, all assertions are *first-order* dynamic logic formulas. This is because of a subtle difference in meaning between modeling arrays as sequences (not first-order), or as pointers to sequences (first-order).

In the first case $\exists s : s[0] = 0$ expresses that there exists an array s of natural numbers, of which the first is 0. The array itself is not an element of the domain of a model for our many-sorted dynamic logic language, but rather a sequence of elements of the domain Nat. In this interpretation the above formula is valid.

The second case of modeling arrays as pointers is taken in this paper and by Java. For a logical variable l of type T we have the following semantics of existential quantification

$$M \models \exists l.\phi \text{ iff for some } o \in M(T) : M' \models \phi.$$

where M' differs from M only in $M'(l) = o$. The semantics of universal quantification can be derived using the equivalence $(\forall l : \phi) \leftrightarrow (\neg\exists l : \neg\phi)$. In this interpretation for quantification, if a is a logical variable of type Nat[ ] then $\exists a : a[1] = 0$ asserts the existence of an array object (an individual of the domain for Nat[ ]) in which currently the first element is 0. This formula is not valid, for it is false in all models in which no such array object exists. Note also that the *extensionality* axiom $\forall a, b, n : a[n] = b[n] \rightarrow a = b$ for arrays is also not valid.

As an illustration of the semantics for object creation and quantification, we show that the scope of quantification over objects is dynamic:

$$\begin{array}{l} M \models \forall l.\langle u := \mathsf{new}\rangle\neg(u = l) \\ \text{iff} \\ \text{for all } o' \in M(T) : M' \models \langle u := \mathsf{new}\rangle\neg(u = l) \end{array}$$

where M' differs from M only in $M'(l) = o'$. Let $o = \nu(M[l := o'], \mathrm{C})$. By the semantics of the diamond modality of dynamic logic and the above semantics of object creation we conclude that

$$\begin{array}{l} M' \models \langle u := \mathsf{new}\rangle\neg(u = l) \\ \text{iff (semantics of object creation)} \\ M'' \models \neg(u = l) \\ \text{iff (compute values in the model } M'' \text{ of } u, l) \\ o \neq o' \end{array}$$

where $\langle u := \mathsf{new}, M'\rangle \rightarrow M''$. In the last step of the proof we have used properties of the semantics of object creation to deduce that $M''(l) = o'$ and $M''(u) = o$. This is justified by the following line of reasoning. By definition of $o = \nu(M, \mathrm{C})$ we have $o \notin M(\mathrm{C})$.

Furthermore, since $M(\mathrm{C}) = \mathrm{M}'(\mathrm{C})$ we also have $o \notin M'(\mathrm{C})$. On the other hand, $o' \in M'(\mathrm{C})$, hence we indeed have $o \neq o'$ for all $o' \in M(\mathrm{C})$.

The following two lemmas summarize properties of the semantics of expressions, statements and formulas. The coincidence lemma states that the interpretation of variables that do not appear in expressions (formulas) have no effect on their value (truth).

Lemma 1. *Coincidence lemma*
Let $M_1 = M_2$ be two models such that $M_1(z) = M_2(z)$ for all non-logical symbols $z \notin \mathrm{Var}$ and $M_1(T) = M_2(T)$ for all types T.

(1) For all expressions e: if $M_1(v) = M_2(v)$ for all variables $v \in var(e)$ then $[\![e]\!](M_1) = [\![e]\!](M_2)$.
(2) For all formulas ϕ: if $M_1(v) = M_2(v)$ for all variables $v \in free(\phi)$ then $M_1 \models \phi$ iff $M_2 \models \phi$.

Proof: The proof follows by induction on the structure of expressions (first case), and formulas (second case). We will prove the case for $\phi \equiv \exists l : p$.

$M_1 \models \exists l.p$
iff (semantics of existential quantification)
for some $o \in M_1(\mathrm{C}) : \mathrm{M}'_1 \models \mathrm{p}$
iff (induction hypothesis and $M_1(\mathrm{C}) = \mathrm{M}_2(\mathrm{C})$)
for some $o \in M_2(\mathrm{C}) : \mathrm{M}'_2 \models \mathrm{p}$
iff (semantics of existential quantification)
$M_2 \models \exists l.p$

Here M'_1 and M'_2 are obtained from respectively M_1 and M_2 by setting $M'_1(l) = M'_2(l) = o$.

The first part of the change and access lemma states that the interpretation of the variables that are not assigned to does not change (recall that D is the set of method declarations in all classes). The second part states that the interpretation of the variables that do not appear in a statement do not affect the execution of the statement.

Lemma 2. *Change and access lemma*
Let M_1, M_2 be two models such that $M_1(z) = M_2(z)$ for all non-logical symbols $z \notin \mathrm{Var}$ and $M_1(T) = M_2(T)$ for all types T. Further suppose $\langle s, M_1\rangle \to M'_1$ and $\langle s, M_2\rangle \to M'_2$.

(1) If $v \in (\mathrm{Var} \setminus change(s))$ then $M_1(v) = M'_1(v)$.
(2) If $M_1(v) = M_2(v)$ for all variables $v \in var(s)$ then $M'_1(v) = M'_2(v)$.

Proof: The proof proceeds by induction on the length of the computation of s.

3. Transformations

We justify our core language here by showing that it contains suitable primitive constructs from which more intricate features can be handled by a completely mechanical translation. The features to which this transformational approach applies include *bounded* arrays, inheritance and dynamic binding. The transformational approach (see also [8]) in general clarifies the relation between programs of our core language, and programs written in Java using such 'derived' features. Moreover the transformations allow us to treat object creation orthogonal to such features, and thereby indicates our approach scales up to modern languages.

Failures. A failure occurs in the execution of statements when executing abort, calling a method or accessing a field of the null object and when evaluating an undefined expression (such as division by zero). Therefore in principle one needs to add checks for definedness to all expressions and statements. This seriously obfuscates both proof rules for and semantics of programs. We solve this problem by assuming that only an execution of abort fails.

To this end we define a program transformation which guarantees that when the original program is about to fail, abort is executed in the transformed program. In the definition of the transformation we will make use of a Boolean expression def(e) with the following property:

$[\![\text{def}(e)]\!](M)$ if and only if no failure would occur when evaluating e in M.

In principle any definition of def which satisfies the above property will suffice, but as an example we define here a few representative cases of the definition. A more complete treatment can be found in [17].

- $\text{def}(u) \equiv \mathsf{true}$
- $\text{def}(e.x) \equiv \text{def}(e) \wedge e \neq \mathsf{null}$
- $\text{def}(e_1[e_2]) \equiv \text{def}(e_1) \wedge \text{def}(e_2) \wedge e_1 \neq \mathsf{null}$
- $\text{def}(f(e_1, \ldots, e_n)) \equiv \text{def}(e_1) \wedge \ldots \wedge \text{def}(e_n) \wedge \text{def}(f)(e_1, \ldots, e_n)$
- $\text{def}(\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} \equiv \text{def}(b) \wedge (b \wedge \text{def}(e_1) \vee \neg b \wedge \text{def}(e_2))$

The fourth case covers arithmetical operations which can fail, such as division. For such operations we assume to have a condition $\text{def}(f)(v_1, ..., v_n)$ on the arguments on which f is defined. For division this condition is $\text{def}(div)(x, y) \equiv y \neq 0$.

We are now in the position to define the transformation $\Theta(s)$ by induction on the structure of s:

- $\Theta(u{:=}e) \equiv \mathsf{if}\ \text{def}(e)\ \mathsf{then}\ u{:=}e\ \mathsf{else\ abort\ fi}$
- $\Theta(s_1; s_2) \equiv \Theta(s_1); \Theta(s_2)$
- $\Theta(\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}) \equiv \mathsf{if}\ \text{def}(e)\ \mathsf{then}(\mathsf{if}\ e\ \mathsf{then}\ \Theta(s_1)\ \mathsf{else}\ \Theta(s_2)\ \mathsf{fi})\ \mathsf{else\ abort\ fi}$
- $\Theta(m(e_1, \ldots, e_n)) \equiv$
  $\mathsf{if}(\text{def}(e_1) \wedge \ldots \wedge \text{def}(e_n) \wedge e_1 \neq \mathsf{null})\ \mathsf{then}\ m(e_1, \ldots, e_n)\ \mathsf{else\ abort\ fi}$

The cases not covered here are trivial adaptations of the above cases. The final result is a transformed program where the above transformation is applied to the main statement of the program, and to each method body.

In addition to transformations of expressions and programs, it is also possible to define a transformation on formulae, depending on the intended semantics of assertions. For instance, in the proof system we shall use in this paper, arrays will be treated in the assertion language as unbounded. Consequently if an array of natural numbers is created (more on bounded arrays follows below) and assigned to the variable a, then afterwards $a[a.length + 1] = 0$ will be provable (because integers are initialized to 0), despite the fact that $a.length+1$ lies outside the array bounds. If instead a semantics of assertions is desired in which this formula is not provable, a transformation on assertions could be defined as: $\Theta(a[e]) \equiv \mathsf{if}\ 0 \leq e < a.length\ \mathsf{then}\ a[e]\ \mathsf{fi}$.

Return Values and Constructor Methods. Java constructors are special methods that initialize the fields of a newly created object, do not return anything and are named after their class. In java the statement `new C(e)` initializes the fields of the new object by executing

the constructor C (with actual parameters e). Constructors can be handled in our core language by the following simple transformation (the argument of Θ is a Java code fragment, the result is a code fragment of our core language):

- $\Theta(\texttt{new C(e)}) \equiv u := new;\ u.C'(e)$

where u is a fresh variable of type C. Intuitively this first creates the new object and then invokes the constructor (which is renamed to C') on it, treating it like a normal method. The renaming avoids name clashes with class predicates. Return values of methods can be simulated by introducing a fresh (global) variable *result* and assigning the return value to *result* whenever the `return`-statement would be executed in the original Java program. This results in the following transformation:

- $\Theta(\texttt{return}\ e) \equiv result := e$

Bounded Arrays. An expression $a[n]$ involving an unbounded array a is defined for each natural number n. Bounded arrays are defined only on an initial segment $\{0, ..., b\}$ of the natural numbers. The number b is called the bound of the array. In Java bounded arrays are created with the statement `new T[n]`. We model the bound by adding a read-only field *length* of type Nat to each array type, and defining the following transformation from Java to our core language:

- $\Theta(\texttt{new T[n]}) \equiv a := new;\ a.length := n.$

where a is a fresh array variable of type T. The intention is that $a[k]$ is defined if and only if $0 \leq k < a.length$. If an element outside of the bounds of the array is accessed, a failure occurs. This requires only the following revision in the definition of def(e):

- $\mathrm{def}(e_1[e_2]) \equiv$
  $\mathsf{if}(\mathrm{def}(e_1) \wedge \mathrm{def}(e_2))\ \mathsf{then}(e_1 \neq \mathsf{null} \wedge 0 \leq e_2 < e_1.length)\ \mathsf{else}\ \mathsf{false}\ \mathsf{fi}$

Note that the length of the array is not part of the type of an array variable. This follows the usual practice in Java where the number of dimensions is part of the type of an array variable, but the length is not. Consequently arrays with different lengths can be assigned to the same array variable during a program execution.

Inheritance and Dynamic Binding. An inheritance relation R is a reflexive transitive binary relation between classes. If $R(C, D)$ then class C is a *superclass* of class D, and class D is a *subclass* of class C. In our semantics (described in Section 2.1.3) subclasses do not correspond to subsets: if C is a subclass of D then their sets of instances $dom(C)$ and $dom(D)$ are disjoint. This property will be convenient in modeling dynamic binding below. We will treat *single* inheritance here, i.e. each class has at most one direct superclass. Adding support for inheritance now reduces to adding easy type checks: if C is a superclass of D then the fields of C are a subset of those of D, and whenever an expression of type C is expected, an expression of type D is also allowed. The e instanceof C operator of Java can be defined in as the Boolean expression $n \neq null \wedge (C(e) \vee C_1(e) \vee \ldots \vee C_n(e)$ of all the superclasses $C_1, \ldots, C_n$ of C.

To support dynamic binding we first rename each method uniquely by prefixing its class: if m is a method of class C then its new name is $C.m$. Dynamic binding is then achieved by transforming each method call $m(s, e_1, ..., e_k)$ to the statement S_n, defined inductively as follows:

$$\begin{aligned} S_0 &\equiv \mathsf{skip} \\ S_{i+1} &\equiv \mathsf{if}\ C_i + 1(s)\ \mathsf{then}\ C.m(s, e_1, ..., e_k)\ \mathsf{else}\ S_i\ \mathsf{fi} \end{aligned}$$

where $\{C_1, ..., C_n\}$ is the set of subclasses of the type of s. In this scheme we make essential use of the fact that $C(s)$ returns true only if s is exactly of type C, not if the type of s is merely a subclass of C.

The following theorem summarizes the outcome of the transformations:

Theorem 1. For all statements s, method declarations D and models M:
$\langle s, M\rangle \to M'$ iff $\langle \Theta(s), M\rangle \to M'$,
where s is executed in the context of the method declarations D, and $\Theta(s)$ in the context of the *transformed* method declarations $\Theta(D)$.

Proof: The proof of this theorem proceeds by induction on s, is straightforward and therefore omitted.

4. Sequent Calculus

In this section, we introduce a proof system for dynamic logic with object creation which abstracts from the explicit representation of objects in the semantics defined above. As a consequence the rules of the proof system are purely defined in terms of the logic itself and do not refer to the semantics. It is characteristic for dynamic logic, in contrast to Hoare logic or weakest precondition calculi, that program reasoning is fully interleaved with first-order logic reasoning, because diamond, box or update modalities can appear both outside and inside the logical connectives and quantifiers. It is therefore important to realize that in the following proof rules, ϕ, ψ and alike, match *any* formula of our logic, possibly containing programs or updates.

We follow [12, 10] in presenting the proof system for dynamic logic as a Gentzen-style sequent calculus. A sequent is a pair of sets of formulas (each formula closed for logical variables) written as $\phi_1, ..., \phi_m \vdash \psi_1, ..., \psi_n$. The formulae ϕ_i on the left of the *sequent arrow* $\vdash$ are called the *antecedent*, the formulae ψ_j on the right the *succedent* of the sequent. The meaning of such a sequent is identical to the meaning of the logic formula

$$\bigwedge_{i=1...m} \phi_i \to \bigvee_{i=1...n} \psi_i$$

where an empty antecedent (or succedent) is the neutral element of the conjunction ($true$) resp. disjunction ($false$).

We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by $\Gamma \vdash \phi \to \psi, \Delta$ we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and zero, one or many sequents as premises:

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots \ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Semantically, a rule states that the validity of all n premises implies the validity of the conclusion ("top-down").

We use the sequent calculus analytically, i.e., we start with the sequent to be proven valid. A sequent calculus proof is a tree in which each node is labeled with a sequent. The root node is labeled with the initial sequent to be proven valid. Rules are applied on the leaves of the proof tree in a bottom-up manner. This means, a rule is applied to the sequent seq_n of a leaf node n by matching its conclusion against seq_n, instantiating the premises with the obtained instantiation and adding them as new children of n. This reduces the

$$\textsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta} \qquad \textsf{andRight}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

$$\textsf{allRight}\ \frac{\Gamma \vdash \phi[c/l], \Delta}{\Gamma \vdash \forall l.\phi, \Delta}\ \text{with } c \text{ a new constant} \qquad \textsf{allot}\ \frac{\Gamma, \forall l.\phi, \phi[e/l] \vdash \Delta}{\Gamma, \forall l.\phi \vdash \Delta}\ \text{with } e \text{ an expression}$$

$$\textsf{close}\ \frac{}{\Gamma, \phi \vdash \phi, \Delta} \qquad \textsf{ind}\ \frac{\Gamma \vdash \phi[0/l], \Delta \qquad \Gamma \vdash \forall l.(\phi \to \phi[l+1/l]), \Delta}{\Gamma \vdash \forall l.\phi, \Delta}\ \text{with } l \text{ of type Nat}$$

Figure 1: Some first-order rules

provability of the conclusion to the provability of the premises step-wise until the obtained sequents are trivially valid (same formula on both sides, $false$ in the antecedent or $true$ in the succedent) and can be closed by applying rules with no premise. A proof is closed if and only if all its branches are closed.

In Fig. 1 we present some of the rules dealing with propositional connectives and quantifiers (see [22] for the full set). We omit the rules for the left hand side, the rules to deal with negation and the rule to cover conditional expressions. $\phi[e/l]$ denotes standard substitution of l with e in ϕ.

Several sequent rules resemble more (conditional) rewrite rules which replace a formula ϕ by an equivalent formula ϕ' or a term t by a semantically equal term t'. Most rules dealing with programs are actually of this kind as most of them are not sensitive to the side of the sequent and can moreover be applied to sub-formulas even. For instance, $\langle s_1; s_2 \rangle \phi$ can be split up into $\langle s_1 \rangle \langle s_2 \rangle \phi$ regardless of where it occurs. For ease of presentation, we introduce the following syntax

$$\frac{\lfloor\, \phi' \,\rfloor}{\lfloor\, \phi \,\rfloor}$$

to express these kind of rules where the premise is constructed from the conclusion via replacing an occurrence of ϕ by ϕ'.

Example 1. A simple example for a rewrite rule is for instance

$$\textsf{concreteAnd}\ \frac{\lfloor\, \phi \,\rfloor}{\lfloor\, \phi \wedge true \,\rfloor}$$

which can be applied on all occurrences of $A \wedge true$ in the sequent

$$B \to (A \wedge true), A \wedge true \vdash B \wedge (A \wedge true)$$

reducing it step-wise to the sequent

$$B \to A, A \vdash B \wedge A$$

In Fig. 2 we present the rules dealing with statements. The schematic modality $\langle\!|\cdot|\!\rangle$ can be instantiated with both $[\cdot]$ and $\langle\cdot\rangle$, though consistently within a single rule application.

Total correctness formulas of the form $\langle \textsf{while} \ldots \rangle \phi$ are proved by first applying the induction rule ind (possibly after generalizing the formula) and applying the unwind rule within

$$\mathsf{split}\ \frac{\lfloor\, \langle\!\langle s_1 \rangle\!\rangle \langle\!\langle s_2 \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle s_1; s_2 \rangle\!\rangle \phi \,\rfloor} \qquad \mathsf{if}\ \frac{\lfloor\, (e \rightarrow \langle\!\langle s_1 \rangle\!\rangle \phi) \wedge (\neg e \rightarrow \langle\!\langle s_2 \rangle\!\rangle \phi) \,\rfloor}{\lfloor\, \langle\!\langle \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi} \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{unwind}\ \frac{\lfloor\, \langle\!\langle \mathsf{if}\ e\ \mathsf{then}\ s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi} \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{assignVar}\ \frac{\lfloor\, \{u := e\}\phi \,\rfloor}{\lfloor\, \langle\!\langle u := e \rangle\!\rangle \phi \,\rfloor} \qquad \mathsf{assignField}\ \frac{\lfloor\, \{e_1.x := e_2\}\phi \,\rfloor}{\lfloor\, \langle\!\langle e_1.x := e_2 \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{createObj}\ \frac{\lfloor\, \{u := \mathsf{new}\}\phi \,\rfloor}{\lfloor\, \langle\!\langle u := \mathsf{new} \rangle\!\rangle \phi \,\rfloor}$$

$$\exists - \mathsf{Introduction}\ \frac{\lfloor\, \phi \rightarrow \langle\!\langle s \rangle\!\rangle \phi' \,\rfloor}{\lfloor\, (\exists l.\phi) \rightarrow \langle\!\langle s \rangle\!\rangle \phi' \,\rfloor}$$
where $l \notin free(\phi')$ is a logical variable

$$\mathsf{Invariance}\ \frac{\lfloor\, \phi_1 \rightarrow \langle\!\langle s \rangle\!\rangle \phi_1' \,\rfloor}{\lfloor\, (\phi_1 \wedge \phi_2) \rightarrow \langle\!\langle s \rangle\!\rangle (\phi_1' \wedge \phi_2) \,\rfloor}$$
where $free(\phi_2) \cap change(s) = \emptyset$

Figure 2: Dynamic logic rules

the induction step. For space reasons, we omit the invariant rule dealing with formulas of the form $[\mathsf{while}\ldots]\phi$ (see [12, 11]). The last two auxiliary rules are adaptation rules. This name is derived from the fact that these rules allow adapting a contract to a specific context. For while-programs such rules are unnecessary, but they are essential for a relative-complete proof system (see [6]).

A proof-rule

$$\frac{\phi_1, \ldots, \phi_n}{\phi'}$$

is sound if whenever all of $\phi_1, \ldots, \phi_n$ are true then ϕ' is true.

Theorem 2. *Soundness.*
The proof-rules given in Figure 1 and Figure 2 are sound.

Proof We prove the soundness of the rule $\exists$-Introduction. Let M be an arbitrary model such that $M, D \models \exists l.\phi$. According to the semantics of formulas there exists some value d such that $M[l := d], D \models \phi$. Suppose $\langle s, D, M\rangle \rightarrow M'$ and $\langle s, D, M[l := d]\rangle \rightarrow M''$. The premise of $\exists$-Introduction now guarantees $M'', D \models \phi'$. From lemma 2 we infer that for all non-logical symbols z different from l: $M'(z) = M''(z)$. Therefore it follows from lemma 1 that $M' \models \phi'$, as desired.

Method calls. In the specification of methods we employ auxiliary variables, sometimes also called model or ghost variables. One aspect in which our approach differs from the seminal work of Owicki and Gries [34] is that due to object creation, their rule of deleting assignments to ghost variables does not hold in our approach. For instance, the formula $\{u := \mathsf{new}\}\exists x.x \neq null$ might not be valid if the update is deleted.

From a pure logic point of view ghost variables (or ghost fields) are nothing else than normal local variables (or fields) and their semantics are exactly the same. The syntactic distinction is made to distinguish cleanly between program code and specification. This allows to strip the whole specification code out of the source code without altering its semantics, and, in particular, the production program code can be compiled such that it does not contain variable declarations and other statements which are used for specification purposes only.

To ease specification we restrict the usage of ghost variables as follows: In our approach ghost variables do not occur in method bodies and are used only to express properties of methods. The value of ghost variables will be set in a special block of code that executes directly after the method body. This code block is required to terminate for obvious reasons. In the block, fields and normal variables are *read-only* and only ghost variables may be assigned to. To avoid complicated analyses of the control flow we disallow method calls in the code block: the block is a while-program.

We associate two dynamic logic formulas to each method: a precondition and a post-condition. Following the design by contract paradigm the partial correctness *contract* of a method m with $u_1, \ldots, u_n$ as its formal parameters, a precondition p and a post-condition q is given by the formula $p \rightarrow [m(t_1, \ldots, t_n]\langle s \rangle q$. In contract we require $free(q) \cap \{u_1, \ldots, u_n\} = \emptyset$. The while-program s is the code block for ghost variables (note that the modality ensures termination of this statement), and $t_1, \ldots t_n$ is a sequence of fresh variables representing the actual parameters (which can be used in the post-condition to refer to the formal parameters' value). The freshness of the variables ensures that the contract specifies a property of a *generic* call. The restriction that the formal parameters do not occur in the post-condition is necessary for the soundness of the verification conditions we are about to introduce. To prove for example that the value of the formal parameters before and after the call is the same (they are reset to their initial values according to the semantics of method calls), other rules such as the invariance rule must be used.

Proving correctness of a program where the main statement has the contract $p \rightarrow [s_{\text{main}}]q$ involving (possibly recursive) methods with contracts
$p_i \rightarrow [m_i(t_1, \ldots, t_n)]\langle s_i \rangle q_i$ and method body B_i amounts to proving the following *verification conditions*:

(1) $A \vdash p \rightarrow [s_{\text{main}}]q$
(2) $A \vdash (p_i \wedge u_1 = t_1 \wedge \ldots \wedge u_n = t_n) \rightarrow [B_i]\langle s_i \rangle q_i$

These verification conditions are inspired by the (partial correctness) rules for Hoare logic given in [7].

5. Applications of updates

In this section, we define a rewrite relation on dynamic logic formulas with updates, to standard first-order logic formulas without updates. This relation is necessary to reason about formulas containing updates. Updates are essentially delayed substitutions. [2] They are resolved by application to the succeeding formula, e.g., $\{u := e\}(u > 0)$ leads to $e > 0$. Update application is only allowed on formulas *not* starting with either a diamond, box or update modality. The last restriction is dropped for symbolic execution, see Section 7.

[2]The benefit of delaying substitutions in the context of symbolic execution is illustrated in Section 7.

$$\mathsf{R1}\ \frac{\{\mathcal{U}\}\phi_1 * \{\mathcal{U}\}\phi_2 \rightsquigarrow \phi'}{\{\mathcal{U}\}(\phi_1 * \phi_2) \rightsquigarrow \phi'} \text{ with } * \in \{\wedge, \vee, \rightarrow\} \qquad \mathsf{R2}\ \frac{\neg\{\mathcal{U}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}\}(\neg\phi) \rightsquigarrow \phi'} \qquad \mathsf{R3}\ \frac{Q\,l.\,\{\mathcal{U}_{nc}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}_{nc}\}(Q\,l.\,\phi) \rightsquigarrow \phi'} \text{ with } Q \in \{\forall, \exists\},\ l \text{ not in } \mathcal{U}_{nc}$$

$$\mathsf{R4}\ \frac{\{\mathcal{U}_{nc}\}e_1 = \{\mathcal{U}_{nc}\}e_2 \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}(e_1 = e_2) \rightsquigarrow e'} \qquad \mathsf{R5}\ \frac{f(\{\mathcal{U}\}e_1, ..., \{\mathcal{U}\}e_n) \rightsquigarrow e'}{\{\mathcal{U}\}f(e_1, ..., e_n) \rightsquigarrow e'} \qquad \mathsf{R6}\ \frac{C(\{\mathcal{U}_{nc}\}e) \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}C(e) \rightsquigarrow e'} \text{ where } C \text{ is the class predicate}$$

$$\mathsf{R7}\ \frac{\mathsf{if}(\{\mathcal{U}\}b)\,\mathsf{then}(\{\mathcal{U}\}e_1)\,\mathsf{else}(\{\mathcal{U}\}e_2)\,\mathsf{fi} \rightsquigarrow e'}{\{\mathcal{U}\}(\mathsf{if}\,b\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2\,\mathsf{fi}) \rightsquigarrow e'}$$

Figure 3: Update Application, general cases

We now define update application on formulas in terms of a rewrite relation $\{\mathcal{U}\}\phi \rightsquigarrow \phi'$ on formulas. As a technical vehicle, we extend the update operator to expressions, such that $\{\mathcal{U}\}e$ is an expression, for all updates $\mathcal{U}$ and expressions e. Accordingly, the rewrite relation $\rightsquigarrow$ carries over to such expressions: $\{\mathcal{U}\}e \rightsquigarrow e'$.

5.1. **General Updates.** Fig. 3 defines $\rightsquigarrow$ for all standard cases (see also [35, 10]). The symbol $\mathcal{U}$ matches all updates, whereas $\mathcal{U}_{nc}$ ('*n*on-*c*reating') excludes the form $u := \mathsf{new}$.

Object creation of the form $u := \mathsf{new}$ is only covered in so far as it behaves like any other update. The cases where object creation makes a difference are discussed separately in Section 5.2. The relation $\rightsquigarrow$ is defined in a big-step manner, such that updates are resolved completely in a single $\rightsquigarrow$ step.

Note that $\rightsquigarrow$ is not defined for formulas of the form $\{\mathcal{U}\}\langle s\rangle\phi$, $\{\mathcal{U}\}[s]\phi$ or $\{\mathcal{U}\}\{\mathcal{U}'\}\phi$, i.e., they are not subject to update application. The conditional expressions used in the rules **R11** and **R13** take into account possible *aliases*. For example, in **R11** we have to check whether the object denoted by e_2 *after* the update $\{\mathcal{U}\}$ equals that of e (*before* the update), because in that case its values obviously also changes by the update.

Figure 5 contains an example illustrating the use of the rules. Note that in the derivation, both **R12** and **R13** are applied to subscripted expressions, but that **R12** is applied since $s.a$ and $u[i]$ have a different type, and **R13** is applied since $s.a$ and u have the same array type, and consequently may be the same array.

The following rule links the rewrite relation $\rightsquigarrow$ with the sequent calculus:

$$\mathsf{applyUpd}\ \frac{\lfloor\, \phi' \,\rfloor}{\lfloor\, \{\mathcal{U}\}\phi \,\rfloor} \text{ with } \{\mathcal{U}\}\phi \rightsquigarrow \phi'$$

The soundness of the rewrite relation for non-creating updates is stated in the next substitution lemma:

Lemma 3. *Substitution Lemma*
Suppose $\langle \mathcal{U}_{nc}, M\rangle \rightarrow M'$.

(1) For all expressions e: if $\{\mathcal{U}_{nc}\}e \rightsquigarrow e'$ then $[\![e']\!](M) = [\![e]\!](M')$.
(2) For all formulas ϕ: if $\{\mathcal{U}_{nc}\}\phi \rightsquigarrow \phi'$ then $M \models \phi'$ iff $M' \models \phi$.

$$\mathsf{R8}\ \frac{}{\{u := e\}u \rightsquigarrow e} \qquad \mathsf{R9}\ \frac{}{\{\mathcal{U}\}u \rightsquigarrow u} \text{ where } \mathcal{U} \equiv u' := e \text{ or } \mathcal{U} \equiv e.x := e_1 \text{ or } \mathcal{U} \equiv e_1[e_2] := e_3$$

$$\mathsf{R10}\ \frac{(\{\mathcal{U}\}e_2).x \rightsquigarrow e'}{\{\mathcal{U}\}(e_2.x) \rightsquigarrow e'} \text{ where } \mathcal{U} \equiv u := e_1 \text{ or } \mathcal{U} \equiv e.y := e_1 \text{ or } \mathcal{U} \equiv e_1[e_2] := e_3$$

$$\mathsf{R11}\ \frac{\mathsf{if}(e = \{\mathcal{U}\}e_2)\ \mathsf{then}\ e_1\ \mathsf{else}(\{\mathcal{U}\}e_2).x\ \mathsf{fi}\ \rightsquigarrow\ e'}{\{\mathcal{U}\}(e_2.x) \rightsquigarrow e'} \text{ where } \mathcal{U} \equiv e.x := e_1$$

$$\mathsf{R12}\ \frac{(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]\ \rightsquigarrow\ e'}{\{\mathcal{U}\}(e_4[e_5]) \rightsquigarrow e'} \text{ where } \mathcal{U} \equiv u := e \text{ or } \mathcal{U} \equiv e.x := e_1 \text{ or } \mathcal{U} \equiv e_1[e_2] := e_3 \text{ and } e_1,\ e_4 \text{ have a different type}$$

$$\mathsf{R13}\ \frac{\mathsf{if}(e_1 = \{\mathcal{U}\}e_4 \wedge e_2 = \{\mathcal{U}\}e_5)\ \mathsf{then}\ e_3\ \mathsf{else}(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]\ \mathsf{fi}\ \rightsquigarrow\ e'}{\{\mathcal{U}\}(e_4[e_5]) \rightsquigarrow e'} \text{ where } \mathcal{U} \equiv e_1[e_2] := e_3 \text{ and } e_1,\ e_4 \text{ have the same type}$$

Figure 4: Update Application, special cases

$$\mathsf{R12}\ \frac{\mathsf{R13}\ \dfrac{\mathsf{R9}\ \dfrac{}{\{\mathcal{U}\}u \rightsquigarrow u} \quad \mathsf{R9}\ \dfrac{}{\{\mathcal{U}\}i \rightsquigarrow i}}{\{\mathcal{U}\}(u[i]) \rightsquigarrow \mathsf{if}(s.a = u \wedge 1 = i)\ \mathsf{then}\ v\ \mathsf{else}\ u[i]\ \mathsf{fi}} \quad \mathsf{R9}\ \dfrac{}{\{\mathcal{U}\}j \rightsquigarrow j}}{\{\mathcal{U}\}(u[i][j]) \rightsquigarrow \mathsf{if}(s.a = u \wedge 1 = i)\ \mathsf{then}\ v\ \mathsf{else}\ u[i]\ \mathsf{fi}[j]}$$

Figure 5: Illustration of the array rules. a and u are two dimensional integer arrays, v a one dimensional integer array, i, j are integers and $\mathcal{U} \equiv s.a[1] := b$

Proof: All cases are standard and can be found in [7], except for the subscripted assignment. Let $\mathcal{U} \equiv e_1[e_2] := e_3$ and suppose $\langle \mathcal{U}, M \rangle \to M'$.

$$\begin{array}{l} [\![e_4[e_5]]\!](M') \\ = \text{(semantics of array access)} \\ M'([\,])([\![e_5]\!](M'))([\![e_4]\!](M')) \\ = \text{(induction hypothesis)} \\ M'([\,])([\![\{\mathcal{U}\}e_5]\!](M))([\![\{\mathcal{U}\}e_4]\!](M)) \end{array}$$

$$
\begin{array}{l}
= \text{(definition of } M', \text{ semantics of subscripted assignment)} \\
\begin{cases}
[\![e_3]\!](M) & \text{if } [\![e_1]\!](M) = [\![\{\mathcal{U}\}e_4]\!](M) \wedge \\
 & \quad [\![e_2]\!](M) = [\![\{\mathcal{U}\}e_5]\!](M) \\
M([\,])([\![\{\mathcal{U}\}e_5]\!](M))([\![\{\mathcal{U}\}e_4]\!](M)) & \text{otherwise}
\end{cases} \\
= \text{(semantics of array access)} \\
\begin{cases}
[\![e_3]\!](M) & \text{if } [\![e_1]\!](M) = [\![\{\mathcal{U}\}e_4]\!](M) \wedge \\
 & \quad [\![e_2]\!](M) = [\![\{\mathcal{U}\}e_5]\!](M) \\
[\![(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]]\!](M)) & \text{otherwise}
\end{cases} \\
= \text{(semantics of formulas)} \\
\begin{cases}
[\![e_3]\!](M) & \text{if } [\![e_1 = \{\mathcal{U}\}e_4 \wedge e_2 = \{\mathcal{U}\}e_5]\!](M) \\
[\![(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]]\!](M)) & \text{otherwise}
\end{cases} \\
= \text{(semantics of conditional expression)} \\
[\![\mathsf{if}(e_1 = \{\mathcal{U}\}e_4 \wedge e_2 = \{\mathcal{U}\}e_5)\,\mathsf{then}\,e_3\,\mathsf{else}(\{\mathcal{U}\}e_5)[\{\mathcal{U}\}e_4]\,\mathsf{fi}]\!](M) \\
= \text{(definition of update application on array expressions)} \\
[\![\{\mathcal{U}\}(e_4[e_5])]\!](M)
\end{array}
$$

5.2. **Contextual Application of Object Creation.** To define update application on $\{u := \mathsf{new}\}e$, simple substitution is not sufficient, i.e., replacing u in e by some expression, because we cannot refer to the newly created object in the model prior to its creation. However, object expressions can only be compared for equality, dereferenced, accessed as an array if the object is of an array type, or appear as arguments of a class predicate or conditional expression. Since object expressions do not appear as arguments of any other function, we define update application by a contextual analysis of the occurrences of u in e. Some cases are already covered in Section 5.1, Fig. 3 (the rules dealing with unrestricted $\mathcal{U}$). The other cases are discussed below.

Conditional expressions satisfy the following identity:

$$
\begin{array}{c}
op(e_1, \ldots, \mathsf{if}\,b\,\mathsf{then}\,e_i\,\mathsf{else}\,e_i'\,\mathsf{fi}, \ldots, e_n) = \\
\mathsf{if}\,b\,\mathsf{then}\,op(e_1, \ldots, e_i, \ldots, e_n)\,\mathsf{else}\,op(e_1, \ldots, e_i', \ldots, e_n)\,\mathsf{fi}
\end{array}
$$

where op is any function symbol or relation symbol in our language. Based on this outward shifting of the conditional expression we have the following rule:

$$
\mathsf{R14}\ \frac{\begin{array}{l}\mathsf{if}(\{\mathcal{U}\}b)\,\mathsf{then}(op(\{\mathcal{U}\}e_1, \ldots, \{\mathcal{U}\}e_i, \ldots, \{\mathcal{U}\}e_n)) \\ \mathsf{else}(op(\{\mathcal{U}\}e_1, \ldots, \{\mathcal{U}\}e_i', \ldots, \{\mathcal{U}\}e_n))\,\mathsf{fi} \rightsquigarrow e'\end{array}}{\begin{array}{c}\{\mathcal{U}\}op(e_1, \ldots, \mathsf{if}\,b\,\mathsf{then}\,e_i\,\mathsf{else}\,e_i'\,\mathsf{fi}, \ldots, e_n) \rightsquigarrow e' \\ \text{where } \mathcal{U} \equiv \{u := \mathsf{new}\}\end{array}}
$$

Similarly an expression $\mathsf{if}\,b\,\mathsf{then}\,e\,\mathsf{fi}$ obeys

$$
\mathsf{R15}\ \frac{\mathsf{if}(\{\mathcal{U}\}b)\,\mathsf{then}(op(\{\mathcal{U}\}e_1, \ldots, \{\mathcal{U}\}e_i, \ldots, \{\mathcal{U}\}e_n))\,\mathsf{fi} \rightsquigarrow e'}{\begin{array}{c}\{\mathcal{U}\}op(e_1, \ldots, \mathsf{if}\,b\,\mathsf{then}\,e_i\,\mathsf{fi}, \ldots, e_n) \rightsquigarrow e' \\ \text{where } \mathcal{U} \equiv \{u := \mathsf{new}\}\end{array}}
$$

For program variables we have the rule

$$
\mathsf{C1}\ \frac{}{\begin{array}{c}\{u_1 := \mathsf{new}\}u_2 \rightsquigarrow u_2 \\ \text{where } u_1, u_2 \text{ are different variables}\end{array}}
$$

Note $\{u_1 := \mathsf{new}\}u_1$ can not be simplified further.
Since the fields of a new object are initialized to their default value we have the following rules

$$\mathsf{C2}\ \frac{}{\{u := \mathsf{new}\}u.x \rightsquigarrow init_T}$$
where C → T is the type of x

$$\mathsf{C3}\ \frac{(\{u := \mathsf{new}\}e).x \rightsquigarrow e'}{\{u := \mathsf{new}\}(e.x) \rightsquigarrow e'}$$
where e is neither u nor a conditional expression

The next cases states that all elements in a newly created array are initialized to their default value:

$$\mathsf{C4}\ \frac{}{\{u := \mathsf{new}\}u[e] \rightsquigarrow init_T}$$
where $T[\,]$ is the type of u

$$\mathsf{C5}\ \frac{(\{u := \mathsf{new}\}e)[\{u := \mathsf{new}\}(e_1)]) \rightsquigarrow e'}{\{u := \mathsf{new}\}(e[e_1]) \rightsquigarrow e'}$$
where e is neither u nor a conditional expression

Another possible context in which u can occur is that of an equality $e = e'$. We distinguish the following cases.
If neither e nor e' is u or a conditional expression then they cannot refer to the newly created object and we define[3]

$$\mathsf{C6}\ \frac{(\{u := \mathsf{new}\}e) = (\{u := \mathsf{new}\}e') \rightsquigarrow e''}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow e''}$$
where neither e nor e' is u or a conditional expression

If both the expressions e and e' are u we obviously have

$$\mathsf{C7}\ \frac{}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow \mathsf{true}}$$
where e and e' are u

On the other hand if e is u and e' is neither u nor a conditional expression (or vice versa) then after $u := \mathsf{new}$ the expressions e and e' cannot denote the same object (because one of them refers to the newly created object and the other one refers to an already existing object) and so we define

$$\mathsf{C8}\ \frac{}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow \mathsf{false}}$$
where e is u and e' is neither u nor a conditional expression (or vice versa)

The final context in which u can occur is that of a class predicate, where the following three rules apply:

$$\mathsf{C9}\ \frac{}{\{u := \mathsf{new}\}(\mathrm{C(e)}) \rightsquigarrow \mathsf{true}}$$
where e is u and u is of type C

$$\mathsf{C10}\ \frac{}{\{u := \mathsf{new}\}(\mathrm{C(u)}) \rightsquigarrow \mathsf{false}}$$
where e is u and u is not of type C

[3]To see why the shifting inwards of $\{u := \mathsf{new}\}$ is necessary, consider the case $\{u := \mathsf{new}\}(u.x = 0)$.

$$\mathsf{C11}\ \frac{\mathrm{C}(\{\mathrm{u} := \mathsf{new}\}\mathrm{e}) \rightsquigarrow \mathrm{e}'}{\{u := \mathsf{new}\}(\mathrm{C}(\mathrm{e})) \rightsquigarrow \mathrm{e}'}$$

where e is neither u nor the conditional expression

Now we define the rewriting of $\{u := \mathsf{new}\}\phi$, where ϕ is a first-order formula in predicate logic (which does not contain modalities). The rules for universal quantification are:

$$\mathsf{C12}\ \frac{(\{u := \mathsf{new}\}\phi[u/l]) \wedge \forall l.(\{u := \mathsf{new}\}\phi) \rightsquigarrow \psi}{\{u := \mathsf{new}\}\forall l.\phi \rightsquigarrow \psi}$$

where l is a logical variable of the same type as u

$$\mathsf{C13}\ \frac{\forall l.(\{u := \mathsf{new}\}\phi) \rightsquigarrow \psi}{\{u := \mathsf{new}\}\forall l.\phi \rightsquigarrow \psi}$$

where l is a logical variable of a different type as u

The first rewrite rule takes care of the *changing scope* of the quantified variable l by distinguishing two cases: ϕ holds for the new object is expressed by the first conjunct $\{u := \mathsf{new}\}\phi[u/l]$ (which is obtained by application of the update to $\phi[u/l]$) and ϕ holds for all 'old' objects is expressed by the second conjunct $\forall l.(\{u := \mathsf{new}\}\phi)$. The rules for existential quantification can be derived from the rules for universal quantification and the equivalence $\exists l.\phi$ iff $\neg\forall l.\neg\phi$. For easy reference the object creation rules are summarized in the table 6.

Abstract rewrite systems in general can obey two properties: termination and confluence. Together these two properties ensure the existence of a normal form. In our case, the normal form is an expression or formula without updates and uniqueness of the normal form is clearly only important up to logical equivalence (i.e. a sort of 'semantic version' of confluence). The next lemma characterizes the expressions on which our rewrite relation terminates[4]

Lemma 4. *Termination of the rewrite relation*
Let ϕ' be any pure first-order formula (no modal operators) and let e' be any expression not generated by the grammar

$$e_{\mathrm{nnf}} \quad ::= \quad u \mid \mathsf{if}\, b\, \mathsf{then}\, e_{\mathrm{nnf}}\, \mathsf{fi} \mid \mathsf{if}\, b\, \mathsf{then}\, e_{\mathrm{nnf}}\, \mathsf{else}\, e\, \mathsf{fi} \mid \mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{else}\, e_{\mathrm{nnf}}\, \mathsf{fi}$$

Then

(1) $\{u := \mathsf{new}\}e' \rightsquigarrow e$ where e contains no updates.
(2) $\{u := \mathsf{new}\}\phi' \rightsquigarrow \phi$ where ϕ contains no updates.

Proof: For formulas ϕ with quantifiers it suffices to note that in the derivation of $\{u := \mathsf{new}\}\phi \rightsquigarrow \phi'$, the quantifier rule applies in its premises $\{u := \mathsf{new}\}$ to formulas with one less quantifier. This proves the second case. For the first case, note there is a rewrite rule for each expression $e' \not\equiv u$. In all rules without premises, the resulting expression contains no updates. The lemma now follows from the observation that for rules with premises, but not involving the offending forms of conditional expressions excluded by the grammar above, the update is applied to expressions different from u, whose parse trees are of a lesser height.

The following theorem extends Lemma 3 to object creations. It guarantees that the normal forms obtained from applying the rewrite relation are unique up to logical equivalence.

[4] As a counterexample, the term $\{u := \mathsf{new}\}u$ cannot be simplified further.

$$\mathsf{C1}\ \frac{}{\{\mathcal{U}\}u' \rightsquigarrow u'}$$
where u, u' are different variables

$$\mathsf{C2}\ \frac{}{\{\mathcal{U}\}(u.x) \rightsquigarrow init_T}$$
where x has type C $\to$ T

$$\mathsf{C3}\ \frac{(\{\mathcal{U}\}e).x \rightsquigarrow e'}{\{\mathcal{U}\}(e.x) \rightsquigarrow e'}$$
where e is neither u nor a conditional expression

$$\mathsf{C4}\ \frac{}{\{\mathcal{U}\}u[e] \rightsquigarrow init_T}$$
where u has type $T[\,]$

$$\mathsf{C5}\ \frac{(\{\mathcal{U}\}e)[\{\mathcal{U}\}(e_1)]) \rightsquigarrow e'}{\{\mathcal{U}\}(e[e_1]) \rightsquigarrow e'}$$
where e is neither u nor a conditional expression

$$\mathsf{C6}\ \frac{(\{\mathcal{U}\}e) = (\{\mathcal{U}\}e') \rightsquigarrow e''}{\{\mathcal{U}\}(e = e') \rightsquigarrow e''}$$
where neither e nor e' is u nor a conditional expression

$$\mathsf{C7}\ \frac{}{\{\mathcal{U}\}(e = e') \rightsquigarrow \mathsf{true}}$$
where e and e' are u

$$\mathsf{C8}\ \frac{}{\{\mathcal{U}\}(e = e') \rightsquigarrow \mathsf{false}}$$
where e is u and e' is neither u nor a conditional expression (or vice versa)

$$\mathsf{C9}\ \frac{}{\{\mathcal{U}\}(\mathrm{C(e)}) \rightsquigarrow \mathsf{true}}$$
where e is u and u is of type C

$$\mathsf{C10}\ \frac{}{\{\mathcal{U}\}(\mathrm{C(u)}) \rightsquigarrow \mathsf{false}}$$
where e is u and u is not of type C

$$\mathsf{C11}\ \frac{\mathrm{C}(\{\mathcal{U}\}\mathrm{e}) \rightsquigarrow \mathrm{e}'}{\{\mathcal{U}\}(\mathrm{C(e)}) \rightsquigarrow \mathrm{e}'}$$
where e is neither u nor the conditional expression

$$\mathsf{C12}\ \frac{(\{\mathcal{U}\}\phi[u/l]) \wedge \forall l.(\{\mathcal{U}\}\phi) \rightsquigarrow \psi}{\{\mathcal{U}\}\forall l.\phi \rightsquigarrow \psi}$$
where l is a logical variable of the same type as u

$$\mathsf{C13}\ \frac{\forall l.(\{\mathcal{U}\}\phi) \rightsquigarrow \psi}{\{\mathcal{U}\}\forall l.\phi \rightsquigarrow \psi}$$
where l is a logical variable of a different type as u

Figure 6: Simplification of object creation with $\mathcal{U} \equiv u := \mathsf{new}$

Intuitively the second case of the theorem together with the second case of the previous lemma states that we can compute weakest preconditions of abstract object creation for any pure first-order formula (i.e. not involving modalities).

Theorem 3. *Semantic correctness of the rewrite relation*
Suppose $\langle u := \mathsf{new}, M\rangle \to M'$.

(1) For any expression e, if $\{u := \mathsf{new}\}e \rightsquigarrow e'$ then $[\![e']\!](M) = [\![e]\!](M')$.
(2) For any formula ϕ, if $\{u := \mathsf{new}\}\phi \rightsquigarrow \phi'$ then $M \models \phi'$ iff $M' \models \phi$.

Proof: By induction on the structure of expressions and formulas. We illustrate two representative cases, namely when the rewrite relation for object creation is applied to an

expression $e'.x$, and when it is applied to a formula $\forall l : \phi$. Other cases follow analogously to these.

Suppose $\mathcal{U} \equiv u := \mathsf{new}$, $\langle M, \mathcal{U} \rangle \to M'$, and o is the newly created object.

- If $e \equiv e'.x$, where x is a field of type C $\to$ T. We distinguish three subcases.

  (1) $e \equiv u.x$:

  $[\![\{\mathcal{U}\}(u.x)]\!](M)$
  = (rule C2 and value of $init_T$ in the model M)
  $init_T$
  = (Since fields of new objects are initialized their default value)
  $M(x)(o)$
  = (Operational semantics of object creation)
  $M(x)([\![u]\!](M'))$
  = (Semantics of field access)
  $[\![u.x]\!](M')$

  (2) $e \equiv (\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}).x$:

  $[\![\{\mathcal{U}\}(\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}\ .x)]\!](M)$
  = (rule R14)
  $[\![\mathsf{if}(\{\mathcal{U}\}b)\ \mathsf{then}(\{\mathcal{U}\}e_1)\ \mathsf{else}(\{\mathcal{U}\}e_2)\ \mathsf{fi}]\!](M)$
  = (Semantics of conditional expression)
  $\begin{cases} [\![\{\mathcal{U}\}e_1]\!](M) & \text{if } [\![\{\mathcal{U}\}b]\!](M) \\ [\![\{\mathcal{U}\}e_2]\!](M) & \text{otherwise} \end{cases}$
  = (Induction hypothesis)
  $\begin{cases} [\![e_1]\!](M') & \text{if } [\![b]\!](M') \\ [\![e_2]\!](M') & \text{otherwise} \end{cases}$

  (3) $e \equiv e'.x$ and e' is neither u nor a conditional expression:

  $[\![\{\mathcal{U}\}(e'.x)]\!](M)$
  = (rule C3)
  $[\![(\{\mathcal{U}\}e').x]\!](M)$
  = (Semantics of field access)
  $M(x)([\![\{\mathcal{U}\}e']\!])$
  = (Induction hypothesis)
  $M'(x)([\![\{\mathcal{U}\}e']\!])$
  = (Semantics of field access)
  $[\![e'.x]\!](M')$

- If $\phi \equiv \forall l : \phi$ where l has the same (class) type as u then:

$M \models \{\mathcal{U}\}\forall l.\phi$
iff (rule C12, semantics of formulas)
$M \models \{\mathcal{U}\}\phi[u/l]$ and $M \models \forall l.\{\mathcal{U}\}\phi$
iff (induction hypothesis, definition of M')
$M' \models \phi[u/l]$ and $M \models \forall l.\{\mathcal{U}\}\phi$
iff (semantics of formulas)
$M' \models \phi[u/l]$ and $M[l := o'] \models \{\mathcal{U}\}\phi$ for all $o' \in M(\mathrm{C})$
iff (substitution lemma and $M'(u) = o$)
$M'[l := o] \models \phi$ and $M[l := o'] \models \{\mathcal{U}\}\phi$ for all $o' \in M(\mathrm{C})$
iff ($\langle \mathcal{U}, M[l := o'] \rangle \to M''$ implies $M'' = M'[l := o']$)
$M'[l := o] \models \phi$ and $M'[l := o'] \models \phi$ for all $o' \in M(\mathrm{C})$
iff (since $M'(\mathrm{C}) = \mathrm{M(C)} \cup \{\mathrm{o}\}$)
$M'[l := o'] \models \phi$ for all $o' \in M'(\mathrm{C})$
iff (semantics of formulas)
$M' \models \forall l.\phi$

As an illustration of applying the rewrite relation, we derive $\{u := \mathsf{new}\}\forall l.\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\,\mathsf{false}$:

$$\dfrac{\dfrac{\dfrac{}{\{u := \mathsf{new}\}(u = u) \rightsquigarrow \mathsf{true}} \quad \dfrac{\dfrac{}{\{u := \mathsf{new}\}(u = l) \rightsquigarrow \mathsf{false}}}{\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \neg\,\mathsf{false}}}{\{u := \mathsf{new}\}\neg(u = u) \rightsquigarrow \neg(\mathsf{true})} \quad \dfrac{}{\forall l.\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \forall l.\neg\,\mathsf{false}}}{\{u := \mathsf{new}\}\neg(u = u) \wedge \forall l.\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\,\mathsf{false}}$$
$$\dfrac{}{\{u := \mathsf{new}\}\forall l.\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\,\mathsf{false}}$$

The resulting formula is equivalent to false. We use this to prove the formula $\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$, which states that u is different from all objects existing *after* the update (including u itself), invalid. In fact we have the following derivation for $\neg\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall l.\neg\,\mathsf{false} \vdash \mathsf{true}}\,\mathsf{closeTrue}}{\neg(\mathsf{true}), \forall l.\neg\,\mathsf{false} \vdash}\,\mathsf{notLeft}}{\neg(\mathsf{true}) \wedge \forall l.\neg\,\mathsf{false} \vdash}\,\mathsf{andLeft}}{\{u := \mathsf{new}\}\forall l.\neg(u = l) \vdash}\,\mathsf{applyUpd}}{\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)) \vdash}\,\mathsf{assignVar}}{\vdash \neg\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)}\,\mathsf{notRight}$$

On the other hand, we have the following derivation of

$$\forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$$

which expresses in an abstract and natural way that u indeed is a new object different from objects existing *before* the update.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{false} \vdash}\,\mathsf{closeFalse}}{\vdash \neg\,\mathsf{false}}\,\mathsf{notRight}}{\vdash \{u := \mathsf{new}\}\neg(u = c)}\,\mathsf{applyUpd}}{\vdash \langle u := \mathsf{new}\rangle\neg(u = c)}\,\mathsf{assignVar}}{\vdash \forall l.(\langle u := \mathsf{new}\rangle\neg(u = l))}\,\mathsf{allRight}$$

The second example shows that the standard rules for quantification apply to the quantification over the existing objects.

6. Case Study and Implementation

Consider a queue data structure where items can be added to the beginning of the queue and removed from the end of the queue. The public interface of such a queue contains

- two global variables pointing to its $first$ and $last$ element,
- an $enqueue(v)$ method which adds a Nat v to the beginning of the queue
- and a $dequeue$ method which removes the last item from the queue.

The queue is implemented as a linked list using a $next$ field which points to the next item in the queue. Figure 7 visualizes the result of the method call $first.enqueue(2)$, where $first$ initially (i.e. before executing the call) points to an item with value 3, and $last$ points to an item with value 25. We will verify the following reachability property of the $enqueue$

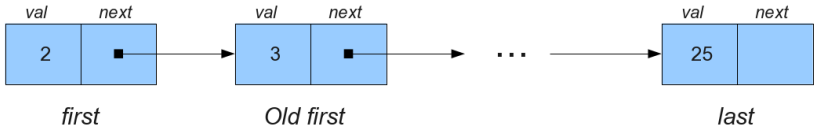

Figure 7: Queue resulting from $first.enqueue(2)$

method, expressed in first-order logic using a ghost array:
$(\exists a : \exists n : n \geq 0 \wedge a[0] = first \wedge a[n] = last \wedge \forall j(0 \leq j < n) : a[j].next = a[j+1])$
$\rightarrow [enqueue(v)] < s >$
$(\exists a : \exists n : n \geq 0 \wedge a[0] = first \wedge a[n] = last \wedge \forall j(0 \leq j < n) : a[j].next = a[j+1]).$
The statement s is the block on the right in Figure 8 and contains updates to the ghost array variable. Intuitively this property means that after executing $enqueue(v)$, $last$ is reachable from $first$ through repeated dereferencing of the $next$ field, provided this was the case initially. Note that by simply adding that $last.next =$ we can rule out cycles. For readability we use the following abbreviations:

- $\text{reach}(f, l, a, n) \equiv n \geq 0 \wedge a[0] = f \wedge a[n] = l \wedge f \neq null \neq l \wedge l.next = null \wedge \forall j(0 \leq j < n) : a[j] \neq null \wedge a[j].next = a[j+1]$
- $p \equiv \text{reach}(first.next, last, a, n) \wedge first \neq null$
- $inv \equiv p \wedge 0 \leq i \wedge b \neq null \wedge b[0] = first \wedge \forall j(0 < j \leq i) : b[j] = a[j-1]$
- B: the method body of $enqueue$ (see the left code block in Figure 8)
- s: updates to the auxiliary variables (right code block in Figure 8)

Note that $\exists a : \exists n : \text{reach}(first, last, a, n)$ is the precondition and post-condition of the contract of $enqueue$, and p is an intermediate assertion which holds directly after the body of $enqueue$ and before the updates to the ghost array variable. The loop invariant of the updates to the ghost array variable is abbreviated to inv.

The proof outline on the right in Figure 8 shows $p \rightarrow \langle s \rangle \text{reach}(first, last, b, n)$. From an application of the consequence rule we infer $p \rightarrow < s > (\exists a : \exists n : \text{reach}(first, last, a, n))$. An application of the rule for sequential composition on this formula and the proof outline for the method body of enqueue yields $\text{reach}(first, last, a, n) \rightarrow [B]\langle s \rangle (\exists a : \exists n : \text{reach}(first, last, a, n))$.

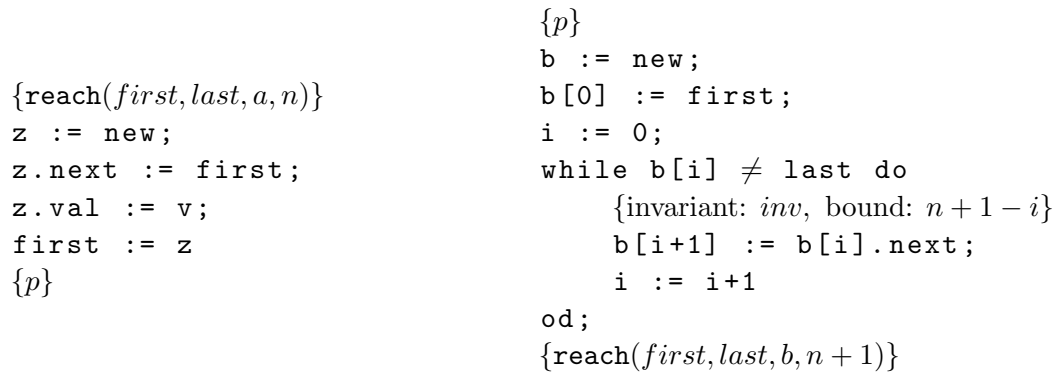

Figure 8: Proof outlines for the *enqueue*(v) method body and ghost variable updates

Finally since a does not occur freely in $(\exists a : \exists n : \text{reach}(first, last, a, n))$, B and s the introduction rule for existential quantification gives the desired result $(\exists a : \exists n : \text{reach}(first, last, a, n)) \rightarrow [B]\langle s\rangle(\exists a : \exists n : \text{reach}(first, last, a, n))$

6.1. **Implementation.** The described dynamic logic has been implemented based on the KeY verification system for Java [10]. In particular, we implemented all dynamic logic rules and update simplification rules as described in sections Section 4 and Section 5.

A notable feature of the implementation is that actual Java programs are supported, not just the core Java-like language introduced in this paper to focus on the relevant issues. In fact, due to building upon KeY we inherit support for a considerably larger subset of Java than discussed in the paper. Besides inheritance, dynamic method binding and bounded arrays, which were discussed in detail, we support classes with constructors, static fields and static methods. The supported subset is basically equal to sequential Java 1.4 without floating points arithmetic and garbage collection. Support for Java 5 features is preliminary. Enhanced for-loops and variable arguments methods are fully supported, generics on the other side need to be transformed away using a provided but external tool.

None of these features required changes in the rules presented in this paper, which strengthens our belief that first, abstract object creation as introduced in Section 5 allows an orthogonal treatment of other features of Java, and second, that the base language in Section 2 is chosen appropriately. We could also reuse the proof search strategies from standard KeY with only minor modifications. This gives us instantaneous support for a highly automated proof search including advanced reasoning about linear and non-linear arithmetic problems. To investigate the degree of automation, we modeled the case study in our KeY-variant. The only required interaction was to provide a suitable loop invariant. Once the loop invariant was given the correctness of the case study program could be proven fully automatically.

Standard KeY already provides support for the Java Modeling Language (JML) [16] as assertion language instead of dynamic logic. Assertions given in JML are first translated into dynamic logic proof obligations before being loaded into the prover. To achieve support for our KeY variant we had to adopt the JML translation to make it aware of abstract object creation. By performing the required changes we were able to support almost all of the JML features that are also supported by standard KeY and can use it as a convenient way to specify our programs.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u<v \vdash u<v}\,\mathsf{close}}{u<v \vdash \{w:=u \,|\, u:=v \,|\, v:=u\}v<u}\,\mathsf{applyUpd}}{u<v \vdash \{w:=u \,|\, u:=v\}\{v:=w\}v<u}\,\mathsf{mergeUpd}}{u<v \vdash \{w:=u \,|\, u:=v\}\langle v:=w\rangle v<u}\,\mathsf{assignVar}}{u<v \vdash \{w:=u\}\{u:=v\}\langle v:=w\rangle v<u}\,\mathsf{mergeUpd}}{u<v \vdash \{w:=u\}\langle u:=v; v:=w\rangle v<u}\,\mathsf{split,\,assignVar}}{u<v \vdash \langle w:=u; u:=v; v:=w\rangle v<u}\,\mathsf{split,\,assignVar}$$

Figure 9: Symbolic execution style proof

7. Symbolic Execution

7.1. **Simultaneous Updates for Symbolic State Representation.** The proof system presented so far allows for classical backwards reasoning, in a weakest precondition manner. We now generalize the notion of updates, to allow for the *accumulation* of substitutions, thereby delaying their application. In particular, this can be done in a *forward manner*, giving the proofs a *symbolic execution* nature. We illustrate this principle by example, in Fig. 9.

The first application of the update rule mergeUpd introduces what is called the simultaneous update $w := u \,|\, u := v$. After applying the second mergeUpd, note that the w from the inner update was turned into a u in the simultaneous update. This is achieved by *applying* the outer update to the inner one:

$$\mathsf{mergeUpd}\ \frac{\lfloor \{\mathcal{U}_1 \,|\, \ldots \,|\, \mathcal{U}_n \,|\, \mathcal{U}'\}\phi \rfloor}{\lfloor \{\mathcal{U}_1 \,|\, \ldots \,|\, \mathcal{U}_n\}\{U\}\phi \rfloor} \quad \text{with } \{\mathcal{U}_1 \,|\, \ldots \,|\, \mathcal{U}_n\}\mathcal{U} \rightsquigarrow \mathcal{U}'$$

For this, we need to extend the rewrite relation $\rightsquigarrow$ towards defining application of updates to updates:

$$\frac{u := \{\mathcal{U}_{nc}\}e \rightsquigarrow \mathcal{U}'}{\{\mathcal{U}_{nc}\}(u := e) \rightsquigarrow \mathcal{U}'} \qquad \frac{(\{\mathcal{U}_{nc}\}e_1).x := \{\mathcal{U}_{nc}\}e_2 \rightsquigarrow \mathcal{U}'}{\{\mathcal{U}_{nc}\}(e_1.x := e_2) \rightsquigarrow \mathcal{U}'}$$

What remains is the definition of the application of simultaneous updates to *expressions*. For space reasons, we will not include the full definition here, but only one interesting special case, where two left-hand sides both write the field x which is accessed in $e.x$.

$$\frac{\mathsf{if}\ ((\mathcal{U}e) = e_2)\ \mathsf{then}\ e_2'\ \mathsf{else}\ \mathsf{if}\ ((\mathcal{U}e) = e_1)\ \mathsf{then}\ e_1'\ \mathsf{else}\ \mathcal{U}(e).x\ \mathsf{fi}\ \mathsf{fi}\ \rightsquigarrow\ e'}{\mathcal{U}(e.x) \rightsquigarrow e'} \quad \text{with } \mathcal{U} = \{e_1.x := e_1' \,|\, e_2.x := e_2'\}$$

This already illustrates two principles: using updates allows us to delay the alias analysis up to the actual application of an update to a field expression. Delaying the analysis allows often to skip the analysis completely because of intermediate simplifications or because the analysis is—for the task at hand—not necessary at all (e.g., imagine one never has to evaluate an expression $e.x$). However, sometimes it is necessary to perform the alias analysis. Here, it means that we have to evaluate e under the update $\mathcal{U}$ and to check if it evaluates to same value as e_2 or e_1.

This brings us to the second principle, namely, what happens in case of a clash, where e_1, e_2 and $\mathcal{U}(e)$ denote the same object. In our semantics the rightmost update will 'win'. This exactly reflects the destructive semantics of imperative programming. Most cases are, however, much simpler. Most of the time, it is sufficient to think of an application of a simultaneous update as an application of a standard substitution (of more than one variable). For a full account on simultaneous updates, see [35].

The idea to use simultaneous updates for symbolic execution was developed in the KeY project [10], and turned out to be a powerful concept for the validation of real world (Java) programs. A simultaneous update forms a representation of the symbolic state which is reached by "executing" the program in the proof up to the current proof node. The program is "executed" in a forward manner, avoiding the backwards execution of (pure) weakest precondition calculi, thereby achieving better readability of proofs. The simultaneous update is only applied to the post-condition as a final, single step. The KeY tool uses these updates not only for verification, but also for test case generation with high code based coverage [20] and for symbolic debugging.

7.2. **Symbolic Execution and Abstract Object Creation.** A motivation to choose the setting of dynamic logic with updates is to allow for abstract object creation in symbolic execution style verification. To do so, we have to answer the question of how symbolic execution and abstract object creation can be combined. The problem is that there is no natural way of merging object creation $\{u := new\}$ with other updates. Consider, for instance, the following formulas, only the first of which is valid.

$$\langle u := \mathsf{new}; v := u\rangle(u = v) \qquad\qquad \langle u := \mathsf{new}; v := \mathsf{new}\rangle(u = v)$$

Symbolic execution generates the following formulas:

$$\{u := \mathsf{new}\}\{v := u\}(u = v) \qquad\qquad \{u := \mathsf{new}\}\{v := \mathsf{new}\}(u = v)$$

Merging the updates naively results in both cases in:

$$\{u := \mathsf{new} \mid v := \mathsf{new}\}(u = v)$$

Whichever semantics one gives to a simultaneous update with two object creations, the formula cannot be both valid and invalid.

The proposed solution is twofold: not to merge an object creation with other updates at all, but to create a second reference to the new object, to be used for merging. For this, we introduce a *fresh* auxiliary variable to store the newly created object, and generate *two* updates according to the following rule:

$$\mathsf{createObj}\ \frac{\lfloor\, \{a := \mathsf{new}\}\{u := a\}\phi \,\rfloor}{\lfloor\, \langle u = \mathsf{new}\rangle\phi \,\rfloor}$$

with a a fresh program variable

The inner update $\{u := v\}$ can be merged with other updates resulting from the analysis of ϕ. The next point to address is the "disruption" of the symbolic state, caused by object creation being unable to merge with their "neighbors", thereby strictly separating state changes happening before and after object creation. The key idea to overcome this is to gradually move all object creations to the very front (as if all objects were allocated up

front) and perform standard symbolic execution on the remaining updates. We achieve this by the following rule:

$$\mathsf{pullCreation}\ \frac{\lfloor\, \{u := \mathsf{new}\}\mathcal{U}_{nc}\phi \,\rfloor}{\lfloor\, \mathcal{U}_{nc}\{u := \mathsf{new}\}\phi \,\rfloor}$$

with u not appearing in $\mathcal{U}_{nc}$

We illustrate symbolic execution with abstract object creation by an example.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg\,\mathsf{false}}\,\mathsf{notRight},\mathsf{closeFalse}}{\vdash \{a := \mathsf{new}\}\neg(v = a)}\,\mathsf{applyUpd}}{\vdash \{a := \mathsf{new}\}\{u := v \mid v := a \mid w := u\}\neg(w = v)}\,\mathsf{applyUpd}}{\vdash \{a := \mathsf{new}\}\{u := v \mid v := a\}\{w := u\}\neg(w = v)}\,\mathsf{mergeUpd}}{\vdash \{a := \mathsf{new}\}\{u := v\}\{v := a\}\langle w := u\rangle\neg(w = v)}\,\mathsf{mergeUpd},\mathsf{assignVar}}{\vdash \{u := v\}\{a := \mathsf{new}\}\{v := a\}\langle w := u\rangle\neg(w = v)}\,\mathsf{pullCreation}}{\vdash \{u := v\}\langle v := \mathsf{new}; w := u\rangle\neg(w = v)}\,\mathsf{split},\mathsf{createObj}}{\vdash \langle u := v; v := \mathsf{new}; w := u\rangle\neg(w = v)}\,\mathsf{split},\mathsf{assignVar}$$

8. Discussion

8.1. **Object Creation vs. Object Activation.** Proof systems for object-oriented languages([1]) usually achieve the uniqueness of objects via an injective mapping, here called obj, from the natural numbers to object identities. Only the object identities $\mathsf{obj}(i)$ up to a maximum index i are considered to stand for actually created objects. In each model, the successor of this maximum index is stored in a ghost variable, here called next. (In case of Java, next would be a `static` field, for each class). Object creation increases the value of next, which conceptually is more an activation than a creation. Quantifiers cover the entire co-domain of obj, including "not yet created" objects. In order to restrict a certain property ϕ to the "created" objects, the following pattern is used: $\forall l.(\psi \to \phi)$, where ψ restricts to the created objects. Formulas of the form $\exists n.\,(n < \mathsf{next} \wedge \mathsf{obj}(n) = l)$ are the approach taken in ODL[12]. To avoid the extra quantifier, ghost instance variable of Boolean type, here called $\mathsf{created}$, can be used to indicate for each object whether or not it has already been "created" [11]. In this case we set the $\mathsf{created}$ status of the "new" object (identified by next) and increase next. The assertion $\forall n.(\mathsf{obj}(n).\mathsf{created} \leftrightarrow n < \mathsf{next})$ retains the relation between the $\mathsf{created}$ status and the object counter next on the level of the proofs. In both case, we need further assertions to state that fields of created objects always refer to created objects.

To state in this setting that a new object indeed is new we need to argument the formula introduced in Section 4, i.e. $\forall l.\,(l.\mathsf{created} \to \langle u := \mathsf{new}\rangle\neg(u = l))$. In fact the formula in Section 4 is not valid in this setting. An object activation style proof of this is given in Fig. 10 (abbreviating $\mathsf{created}$ by cr). Many steps in this proof are caused by the particular details of the explicit representation of objects and the simulation of object creation by object activation.

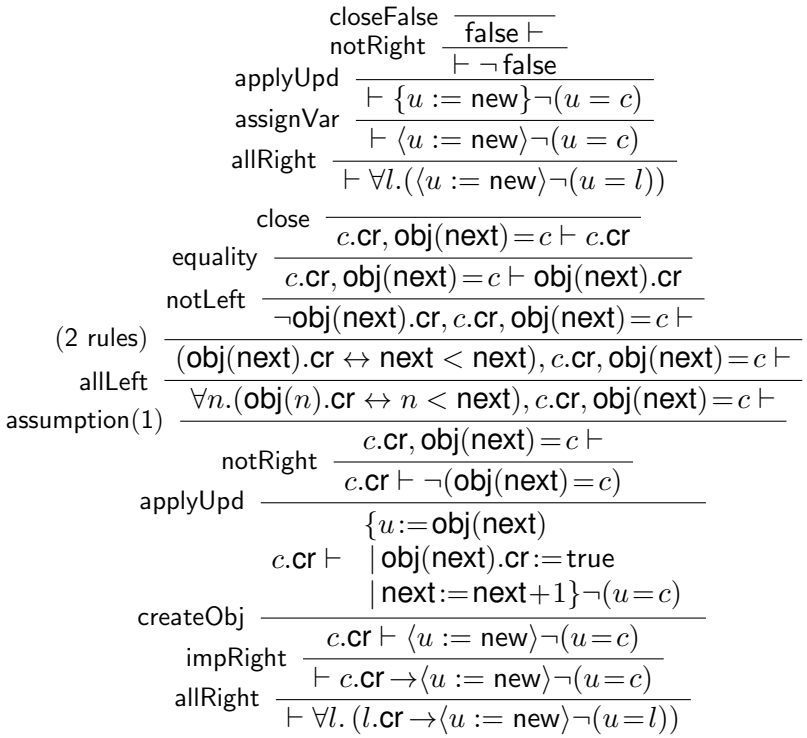

Figure 10: Abstract Object Creation proof (left) vs activation (right)

8.2. **Expressiveness.** Standard first-order logic cannot express reachability properties. We have proposed first-order logic *together with auxiliary variables* to specify properties of the heap. In the case study, an example of a reachability property was expressed and proved subsequently. The question now arises how expressive our approach is in general. In the presence of general abstract data types Tucker and Zucker [37] observe that for expressing, for example, strongest post-conditions standard arithmetic coding techniques do not apply. Therefore Tucker and Zucker prove expressibility of strongest post-conditions in a weak second-order language which contains quantification over finite sequences. It is not difficult though tedious to show that using auxiliary array variables we can express in our first-order language strongest post-conditions for our programming language. We even conjecture that the strongest post-condition of a formula in the language of Presburger arithmetic, and a program instrumented with auxiliary variables in a suitable manner is definable in Presburger arithmetic itself. This is surprising, since the standard approach to show the strongest post-condition is definable is based on the usual Gödel encoding of partial recursive functions, which relies on the presence of multiplication in the assertion language, and multiplication is not available in Presburger arithmetic. The basic idea is that one can instrument any program to store the computation in auxiliary array variables. The computation can then be recovered in an assertion by accessing these auxiliary variables.

9. Conclusion

In this paper we presented the state of the art in the KeY theorem prover. We showed how the assertion language used in KeY can be used conveniently together with auxiliary variables to provide a powerful way to express properties of the heap. Moreover the assertion language supports *abstract* object creation (including dynamically created arrays), abstracting from irrelevant implementation details of object creation, which in general complicate proofs. First-order dynamic logic was used as a systematic way to formalize the inductively defined rewrite relation needed to reason about abstract object creation, and to generate verification conditions. We further showed how more complicated features can be handled by the transformational approach. Tool support is provided by a special version of KeY available on KeY-AOC.

Future Work. A main line of future research concerns the integration of different techniques to further support modularity, i.e., local reasoning as supported by the separating conjunction of separation logic and dynamic frames.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Proc. 7 th Int. Conf. Theory and Practice of Software*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, 1997.

[2] E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. A deductive proof system for multithreaded java with exceptions. *Fundam. Inform.*, 82(4):391–463, 2008.

[3] W. Ahrendt, F. S. de Boer, and I. Grabe. Abstract object creation in dynamic logic. In *FM*, pages 612–627, 2009.

[4] P. America. Designing an object-oriented programming language with behavioural subtyping. In *REX Workshop*, pages 60–90, 1990.

[5] P. America. Formal techniques for parallel object-oriented languages. In *CONCUR*, pages 1–17, 1991.

[6] K. R. Apt. Ten years of hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[7] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs, 3rd Edition*. Texts in Computer Science. Springer-Verlag, 2009. 502 pp, ISBN 978-1-84882-744-8.

[8] K. R. Apt, F. S. de Boer, E.-R. Olderog, and S. de Gouw. Verification of object-oriented programs: A transformational approach. *J. Comput. Syst. Sci.*, 78(3):823–852, 2012.

[9] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[10] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[11] B. Beckert, V. Klebanov, and S. Schlager. Dynamic Logic. In Beckert et al. [10], pages 69–177.

[12] B. Beckert and A. Platzer. Dynamic Logic with Non-rigid Functions. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2006.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[14] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[15] T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J.-C. Filliâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, and M. Ulbrich. The cost ic0701 verification competition 2011. In B. Beckert, F. Damiani, and D. Gurov, editors, *FoVeOOS*, volume 7421 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2011.

[16] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *STTT*, 7(3):212–232, 2005.

[17] Á. Darvas, F. Mehta, and A. Rudich. Efficient well-definedness checking. In *IJCAR*, pages 100–115, 2008.
[18] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *FoSSaCS*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999.
[19] D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.
[20] C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In Y. Gurevich and B. Meyer, editors, *TAP*, volume 4454 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2007.
[21] J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.
[22] M. Giese. First-Order Logic. In Beckert et al. [10], pages 21–68.
[23] K. Huizing and R. Kuiper. Verification of object oriented programs using class invariants. 1783:208–221, 2000.
[24] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.
[25] V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st verified software competition: experience report. In *Proceedings of the 17th international conference on Formal methods*, FM'11, pages 154–168, Berlin, Heidelberg, 2011. Springer-Verlag.
[26] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
[27] G. T. Leavens, J. R. Kiniry, and E. Poll. A jml tutorial: Modular specification and verification of functional behavior for java. In *CAV*, page 37, 2007.
[28] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, B. Jacobs, G. T. Leavens, and C. Ruby. JML: notations and tools supporting detailed design in Java. In *In OOPSLA 2000 Companion*, pages 105–106. ACM, 2000.
[29] K. R. Leino, P. Müller, and J. Smans. Foundations of security analysis and design v. chapter Verification of Concurrent Programs with Chalice, pages 195–222. Springer, Berlin, Heidelberg, 2009.
[30] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
[31] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *Proceedings of the 19th European conference on Programming Languages and Systems*, ESOP'10, pages 407–426, Berlin, Heidelberg, 2010. Springer-Verlag.
[32] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
[33] Object Modeling Group. *Object Constraint Language Specification, version 2.0*, June 2005.
[34] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
[35] P. Rümmer. Sequential, Parallel, and Quantified Updates of First-Order Structures. In M. Hermann and A. Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 2006.
[36] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems*, 34(1), Apr. 2012.
[37] J. Tucker and J. Zucker. *Program correctness over abstract data types, with error-state semantics*. Elsevier Science Inc., 1988.
[38] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.