Footprint Logic for Object-Oriented Components (extended paper)

F. S. DE BOER, Leiden Institute of Advanced Computer Science, Leiden University, Leiden, Netherlands and Centrum Wiskunde en Informatica, Amsterdam, Netherlands
STIJN DE GOUW, Open University of The Netherlands, Heerlen, Netherlands and Centrum Wiskunde en Informatica, Amsterdam, Netherlands
HANS-DIETER HIEP, Centrum Wiskunde & Informatica, Amsterdam, Netherlands and Leiden Institute of Advanced Computer Science, Leiden University, Leiden, Netherlands
JINTING BIAN, Leiden Institute of Advanced Computer Science, Leiden University, Leiden, Netherlands and Centrum Wiskunde en Informatica, Amsterdam, Netherlands

We introduce a new way of reasoning about invariance in terms of *footprints* in a program logic for object-oriented components. A footprint of an object-oriented component is formalized as a monadic predicate that describes which objects on the heap can be affected by the execution of the component. Assuming encapsulation, this amounts to specifying which objects of the component can be called. Adaptation of local specifications into global specifications amounts to showing invariance of assertions, which is ensured by means of a form of *bounded quantification* which excludes references to a given footprint. The new approach is compared to two existing approaches to reason about invariance: separation logic and dynamic frames.

CCS Concepts: • **Theory of computation** → **Program verification**; **Hoare logic**; • **Software and its engineering** → **Formal software verification;**

Additional Key Words and Phrases: Hoare logic, invariance, strong partial correctness

ACM Reference Format:
F. S. de Boer, Stijn de Gouw, Hans-Dieter Hiep, and Jinting Bian. 2025. Footprint Logic for Object-Oriented Components (extended paper). *Form. Asp. Comput.* 37, 2, Article 11 (February 2025), 23 pages. https://doi.org/10.1145/3703921

1 Introduction

A major and important challenge in theoretical computer science is finding practical and efficient verification techniques for showing functional correctness properties of object-oriented programs. A common approach to the verification task is divide and conquer: to split a program into separate

Hans-Dieter Hiep contributed to this article prior to joining.

Authors' Contact Information: F. S. de Boer, Leiden Institute of Advanced Computer Science, Leiden University, Leiden, Zuid-Holland, Netherlands and Centrum Wiskunde en Informatica, Amsterdam, Netherlands; e-mail: frb@cwi.nl; Stijn de Gouw, Open University of The Netherlands, Heerlen, Limburg, Netherlands and Centrum Wiskunde en Informatica, Amsterdam, Netherlands; e-mail: sdg@ou.nl; Hans-Dieter Hiep, Centrum Wiskunde & Informatica, Amsterdam, Netherlands and Leiden Institute of Advanced Computer Science, Leiden University, Leiden, Netherlands; e-mail: hdh@cwi.nl; Jinting Bian, Leiden Institute of Advanced Computer Science, Leiden University, Leiden, Netherlands and Centrum Wiskunde en Informatica, Amsterdam, Netherlands; e-mail: jinting.bian@cwi.nl.

components, where the behavior of each component is locally formally specified. The benefits of employing local specifications are: simplification of the verification task of each component, reusability of verified components, and distribution of the verification task among verifiers. Verified components can then be composed into a larger program, and the correctness of the resulting composed program with respect to a global specification can be established by adapting the local specifications to the larger context in which the components are used, abstracting from their internal implementation.

But what exactly distinguishes a local specification from a global specification? In this paper, we introduce a novel concept of *footprints*. A component is a fragment of a program. We define the footprint of a component as the set of objects which that component may call: if one assumes encapsulation of objects, as is often the case in component-based software, the state of objects outside of the footprint of a component thus remains unaffected by the execution of that component. Furthermore, we interpret the semantics of local component specifications with respect to such a footprint. Verifying the local correctness of a component then requires to show that the execution of a component is restricted to method calls to objects belonging to its footprint. As such, we can formulate precisely which properties of objects remain invariant within a larger context than that of the component alone.

Traditionally, in program logics, such as Hoare logic, the so-called *adaptation* rules are used to adapt a local correctness specification of a component to the larger context in which the component is used. For example this can be achieved by adding to the pre- and postcondition an invariant. as in the following invariance rule:

$$\frac{\{p\}\ S\ \{q\}}{\{p \wedge r\}\ S\ \{q \wedge r\}}$$

where the statement S does not assign the free variables in formula r. Variables in assertions that do not occur in statements are also known as logical variables, since their value have no influence on the execution of a program. Hoare introduced in [16] one rule, *the* adaptation rule, which generalizes all adaptation rules. In his seminal paper [20], Olderog studied the expressiveness and the completeness of the adaptation rule. These adaptation rules form the basis for reasoning about program components, abstracting from their internal implementation. Proof-theoretically this is formalized by the notion of *modular completeness* which is introduced by Zwiers et al. in [25], and, roughly, requires to show that if $\models \{p\}\ S\ \{q\}$ implies $\models \{p'\}\ S\ \{q'\}$ then $\{p'\}\ S\ \{q'\}$ can be derived from $\{p\}\ S\ \{q\}$, abstracting from the implementation of S. The adaptation rules, including Hoare's adaptation rule [16], however, are of limited use in the presence of *aliasing* in object-oriented programs. Aliasing arises when syntactically different expressions refer to the same memory location. Well-known data structures which give rise to aliasing are arrays and pointer structures. In the presence of aliasing we can no longer syntactically determine general invariant properties, but the standard adaptation rules are based on purely syntactic conditions, e.g., whether a given formula contains free variables which also appear in a given statement.

De Boer and de Gouw present in [10] a Hoare logic for recursive programs with arrays by extending the standard pre-postcondition specification with a *footprint*. Footprints in [10] are specified by predicates which constrain the state changes, e.g., array assignments, that are allowed during the computation. There, footprints are defined as a family of predicates indexed by an array name. The arity of each predicate equals that of the associated array (interpreted as a function). The predicate associated with an array describes a subset of the *domain* of that array which may be changed by the statement. A new invariance rule was presented, that can be formulated by means of a syntactical operation which logically excludes references of locations that may be changed as recorded by the footprints. The soundness of this rule was shown, and the logic applied to the verification of the well-known Quicksort sorting algorithm, which results in a simpler and modular proof compared to previous approaches.

In this paper we introduce a novel Hoare logic for reasoning about invariant properties using footprints as they arise in object-oriented components. An object-oriented component is a program with a pre-postcondition specification. One of the main challenges addressed is a formalization of footprints at an abstraction level that coincides with the programming language. For example, in object-oriented programming languages such as Java, we can only refer to objects that *exist*, i.e., that have been dynamically created. Thus quantifiers only range over objects which actually exist, since otherwise there is a mismatch between what objects can be referred to in the programming language and assertion language. In fact, since the main purpose of the concept of footprints is a generalization of the above invariance rule, in the context of object-orientation it should refer only to the objects that exist initially. We have formalized our notion of footprints by extending the Hoare logic underlying the version of the KeY theorem prover for reasoning about dynamic object creation, as described in [12]. We now define footprints as a family of *monadic* predicates: one predicate associated per class. We show how footprints for reasoning about invariance can be smoothly generalized to dynamic object creation in an assertion language which only talks about existing objects.

We therefore generalize the weakest precondition calculus underlying the Hoare logic for object-orientation introduced in [9], to the specification and verification of footprints as sets of objects. To represent and reason about such sets at an appropriate abstraction level, we formalize the assertion language in a *second-order monadic logic*. Since the basic principle underlying component-based software is encapsulation, the footprint predicate describes which objects may be called, which ensures objects cannot be changed if they fall outside the footprint. To reason about footprints in a modular manner we introduce in this paper a new *hybrid* Hoare logic which combines two different interpretations of correctness specifications. One interpretation requires absence of so-called 'null-pointer exceptions', e.g., calling a method on 'null'. The other interpretation generalizes such absence of failures to footprints, requiring that only objects included in the footprint can be called. That is, semantically, calling an object which does not belong to the footprint generates a failure. In the context of this latter interpretation we introduce a new invariance (or 'frame') rule. This rule enforces invariance by a form of *bounded quantification* which restricts the description to that part of the heap disjoint from the footprint.

We compare our approach to the two main existing approaches to reasoning about invariance: separation logic [23] and dynamic frames [18], showing the differences and commonalities between the three approaches by proving a main invariant property of the push operation on a stack data structure. The stack is implemented as a dynamic list of objects connected by a 'next' field. To implement pushing an item, we create a new object that points to the old top of stack and replaces it as the new top of stack. The main invariant property of the push operation is that it does not affect the list structure starting from the old top of the stack.

Plan of the paper. This paper contains the following contributions. Section 2 introduces a basic object-oriented programming language and its semantics. In Section 3 we introduce the specification language for describing properties of pointer structures and the two different interpretations of correctness specifications considered in this paper. In Section 4 we introduce a *hybrid* Hoare logic which combines reasoning about both interpretations. Section 5 introduces the case study and further illustrates the use of our approach by comparing our proof with the use of separation logic and dynamic frames. In Section 6 we discuss, more generally, how our approach compares with the approaches by separation logic and dynamic frames. Finally, we discuss related and future work in Section 7.

Extended paper. This journal paper is an extended version of the conference paper [11]. In Section 2 we added more details on the programming language and its formal semantics. Section 3 is enriched with the syntax and formal semantics of the assertion language and a more elaborate discussion on the elimination of navigation expressions in formulas. Section 4 addresses in more detail the effect on the footprint predicate of the substitution operator for object creation. In Section 5 provides further details on the comparison to other framing approaches and the conclusion includes a brief discussion on a dual form of our footprint logic and on (relative) completeness.

2 The Programming Language

To focus on the main features of our approach, we consider a basic object-oriented programming language. Our language features the following main characteristics of component-based software: objects *encapsulate* their own local state, that is, objects only interact via *method calls*, and objects can be dynamically *created*. For technical convenience, we restrict the data types of our language to the basic value types of **integer** and **Boolean**, and the reference type **object**, which denotes the set of object identities. It should be emphasized here that only for notational convenience we abstract from classes in the presentation of the footprint logic (it is straightforward to generalize our method to classes, and integrate standard approaches for reasoning about other object-oriented features like inheritance, polymorphism, dynamic dispatch, etc.). In our language, the only built-in operation which involves the type **object** is reference equality. The constant **null** of type **object** represents the *invalid reference*.

The set of program variables with typical elements $x, y, \ldots$ is denoted by Var. These include the formal parameters of methods. The variable **this** is a distinguished variable of type **object**. By $Field$ we denote a finite set of field identifiers, with typical element f. We assume variables and fields are implicitly typed. Expressions t are constructed from program variables, fields, and built-in constants and operations. Note that fields in expressions refer to the fields of the **this** object. For notational convenience we denote by $u, v, \ldots$ a variable x or a field f.

We have the following abstract grammar of statements, that are used for describing component behavior, including the method bodies (we again leave typing information implicit).

$$\begin{gathered} S ::= \bar{u} := \bar{t} \mid x := \mathbf{new} \mid y.m(\bar{t}) \mid S_1;\ S_2 \mid \\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \mid \mathbf{while}\ B\ \mathbf{do}\ S_1\ \mathbf{od} \end{gathered}$$

In the parallel assignment $\bar{u} := \bar{t}$, the left-hand side $\bar{u}$ denotes a sequence of variables or fields and $\bar{t}$ is a sequence of expressions, where both sequences correspond in length and correspond in types. For technical convenience (and without loss of generality), we restrict to object creation statements $x := \mathbf{new}$, where the left-hand side is a variable x (we thus exclude direct assignments of new object references to fields). Furthermore, we restrict to method calls $y.m(\bar{t})$, where the callee is denoted by a variable y (and $\bar{t}$ are the actual parameters). Returning a value can be modeled by the use of a global variable which temporarily stores the return value. A program consists of a main statement S and a set of method definitions $m(x_1, \ldots, x_n) :: S$, with formal parameters $x_1, \ldots, x_n \in Var$ different from **this** (the formal parameters of a method are considered local to the method body) and body S where **this** does not occur on the left-hand side of any assignment. As in most real-world conventional imperative and OO-languages, we shall use static scoping of variables. Every method has an implicit formal parameter **this** which holds a reference to the callee and is not **null**. Further, we assume the main statement S does not contain any field assignments.

The formal semantics assumes an infinite set O of object identities with typical elements $o, o', \ldots$, and a set of values V such that $O \subseteq V$, that is, object identities are values. Furthermore, we assume that $\mathbf{null} \in O$. Each object has its own local state which assigns values to its fields. A *heap* $\sigma \in \Sigma$ is a *partial* function, i.e., $\Sigma = O \rightharpoonup (Field \to V)$, which assigns to an object o its local state. A local state $\sigma(o) \in \mathit{Field} \to V$ assigns values to fields. By $\sigma(o) = \bot$ we denote that $\sigma(o)$ is undefined, in other words, o does not 'exist' in the heap σ. We always have $\sigma(\mathbf{null}) = \bot$. By $\mathit{dom}(\sigma)$ we denote the set of existing objects, that is, the domain of σ consists of the objects o for which $\sigma(o)$ is defined. In order to generate new object identities we restrict to heaps σ such that $O \setminus \mathit{dom}(\sigma)$ is infinite. For any $o \in \mathit{dom}(\sigma)$, we denote by $\sigma(o.f)$ the value of field f of the object o if it exists, and otherwise some fixed default value (**null** in case of fields of type **object**). A *store* is a function $\tau \in T$ that assigns values to variables, i.e., $T = \mathit{Var} \to V$. We restrict to stores and local states that are *type safe*, meaning that variables and fields are assigned to values of their type. A *configuration* $(\sigma, \tau) \in \Sigma \times T$ consists of a heap σ and a store τ. Both in the program semantics, and later in the semantics of assertions, we restrict to configurations that are *consistent*, i.e., fields of type **object** of existing objects only refer to existing objects or to **null**, and the store assigns as values to variables of type **object**, only existing objects or **null**.

Given a set of method definitions, the basic input/output semantics $\mathcal{M}(S)$ of a statement S can be defined as a *partial* function $\Sigma\times T \rightharpoonup \Sigma\times T$. Note that the programming language is *deterministic*, so $\mathcal{M}(S)(\sigma, \tau)$ is either undefined, indicating that S does not terminate or does not terminate properly, or it denotes a single final configuration of a properly terminating computation. Method calls are described by inlining (which involves renaming the local variables of the body by fresh variables).

The input/output semantics $\mathcal{M}(S)$ for parallel assignment, sequential composition, conditionals and while loops are standard. However, the semantics for method calls and object creation are not as well-known, and so described in more detail below.

Following [2], we have the semantics of method calls: if $\tau(y) = \mathbf{null}$ then $\mathcal{M}(y.m(\bar{t}))(\sigma, \tau)$ is undefined, and, otherwise, it is defined in terms of the (run-time) statement

$$\mathbf{this}, \bar{x} := y, \bar{t};\ S;\ \mathbf{this}, \bar{x} := \tau(\mathbf{this}), \tau(\bar{x})$$

where S denotes the body of m with formal parameters $\bar{x}$. The first parallel assignment $\mathbf{this}, \bar{x} := y, \bar{t}$ initializes the variables $\mathbf{this}, \bar{x}$ that are local to S, and the second parallel assignment $\mathbf{this}, \bar{x} := \tau(\mathbf{this}), \tau(\bar{x})$ restores the initial values of the variables $\mathbf{this}, \bar{x}$. It is important to note that the actual parameters are evaluated in the initial configuration (σ, τ), e.g., occurrences of the variable **this** in $\bar{t}$ refer to the caller $\tau(\mathbf{this})$ (and only *after* the initialization it refers to the callee).

The semantics of object creation is as follows: $\mathcal{M}(x := \mathbf{new})(\sigma, \tau)$ is defined to be the configuration (σ', τ') where σ' is obtained from σ by assigning to a fresh[1] object identity o some default initial local state and $\tau' = \tau[x := o]$. The fresh object identity o is fresh in the heap of the current configuration σ, i.e., $\sigma(o) = \bot$ and $o \neq \mathbf{null}$, and we pick such fresh o deterministically by some arbitrary but fixed allocation function. For further details of the program semantics we refer to [2, pp. 195–219].

To observe *failures* due to calling a method on **null** (which generates a 'null-pointer exception'), we introduce a failure-sensitive semantics $\mathcal{M}_{\mathbf{null}}(S)(\sigma, \tau) \in (\Sigma \times T) \cup \{\mathbf{fail}\}$ where $\mathcal{M}_{\mathbf{null}}(y.m(\bar{t}))(\sigma, \tau) = \mathbf{fail}$, if $\tau(y) = \mathbf{null}$.

We extend this semantics to additionally model the execution of a statement restricted by a footprint. We restrict to a *coarse-grained* notion of footprints which specifies the objects that may be called (either directly or indirectly through called methods). Note that this includes all the objects that may be changed, since we assume encapsulation and objects can only modify their own state (field assignments are relative to **this**). Our model then is based on extending configurations with an additional set of objects $\phi \subseteq O$ and the definition of a semantics $\mathcal{M}_{\mathbf{F}}(S)(\sigma, \tau, \phi) \in (\Sigma \times T \times \mathcal{P}(O)) \cup \{\mathbf{fail}\}$ where **fail** indicates failure which arises from a call of a method of an object that is not in the footprint ϕ. More specifically, $\mathcal{M}_{\mathbf{F}}(y.m(\bar{t}))(\sigma, \tau, \phi)$ requires that the object $\tau(y)$

[1]Since the abstract set of object identities is infinite, we can always select a fresh identity.

is callable, that is, it belongs to the set ϕ, otherwise $\mathcal{M}_{\mathbf{F}}(y.m(\bar{t}))(\sigma, \tau, \phi) = \mathbf{fail}$. We restrict to footprints ϕ including only objects that exist in the given σ (e.g., failures arise from calling a method on **null**). Thus new objects are added to the footprint, that is, $\mathcal{M}_{\mathbf{F}}(x := \mathbf{new})(\sigma, \tau, \phi) = (\sigma', \tau[x := o], \phi \cup \{o\})$ where o is a fresh object identity and σ' results from σ by initializing the local state of the newly created object o as before. Note that, as stated above, our course-grained notion of a footprint specifies the objects that *may* be called which clearly includes the newly created object (otherwise one could not call methods on newly created objects).

3 The Specification Language

Given a set of method definitions, we specify the correctness of a statement by a pre- and post-condition which are formally specified by logical formulas $p, q, \ldots$, also called assertions. In order to express global properties of the heap, logical expressions e extend the expressions t of our programming language with the *qualified dereferencing* operator: $e.f$ denotes the value of field f of the object denoted by the expression e. As a special case, we abbreviate **this**.f by f (the denotation of the qualified dereferencing of fields of the **this** object, and unqualified references to fields coincide). Note that an assertion $p(e)$, which contains occurrences of a general *navigation expression* e of the form $x.f_1.\cdots.f_n$, is logically equivalent to the assertion

$$\exists y_1, \ldots, y_n \left(y_1 = x.f_1 \wedge \bigwedge_{i=1}^{n-1} y_{i+1} = y_i.f_{i+1} \wedge p(y_n) \right).$$

As such, we may without loss of generality assume that all occurrences of qualified dereferencing operators are of the form $x.f$ for a variable x and field f.

The assertion language further features *logical* variables (which are assumed not to appear in the programs that describe the component behavior). A logical variable can either be a first-order variable x or a second-order, *monadic predicate* X over objects. Second-order quantification of such predicates allow for the specification of properties like *reachability*:

$$\forall Y \big(\big(Y(e) \wedge \forall x (Y(x) \to Y(x.next)) \big) \to Y(e') \big)$$

states that the object denoted by e' can be reached from the one denoted by e via a chain of *next* fields (see Section 5 for more details). We represent the footprint by the distinguished monadic predicate F. The assertion $F(e)$ then expresses that the object denoted by e belongs to the set of objects denoted by the footprint F.

$$\begin{aligned} e &::= x \mid c \mid f \mid x.f \mid f(e_1, \ldots, e_n) \\ p, q &::= \bot \mid R(e_1, \ldots, e_n) \mid F(e) \mid X(e) \mid (p \to q) \mid (\forall x(p)) \mid (\forall X(p)) \end{aligned}$$

Summarizing, assertions $p, q, \ldots$ denote logical formulas constructed from relations over logical expressions ($R(e_1, \ldots, e_n)$ where R is an n-ary predicate), application of a second-order variable to an expression, the standard propositional connectives, first-order quantification over first-order variables of basic type, and second-order quantification over monadic predicates over objects. We use both round and square parentheses for grouping assertions for readability, and we have abbreviations of other logical connectives defined in terms of the ones given: $\wedge, \vee, \leftrightarrow, \exists$.

We omit the straightforward details of the definition $\mathcal{E}(e)(\sigma, \tau, \omega)$ of the semantics of logical expressions e, and the standard Tarski inductive truth definition for $\sigma, \tau, \phi, \omega \models p$, where ω assigns values to the logical variables (assuming that the sets of logical variables and program variables are disjoint). Also note that it is easy to extend the specification language to many-sorted logic, where we have one footprint per sort of objects. In that case the footprint F can be seen as the union of the family of footprints indexed per sort. We only discuss the single-sorted case in this section, but later on (in Section 5) we will use a many-sorted specification language where each sort corresponds to a class in an object-oriented programming language.

As a particular case, $\sigma, \tau, \phi, \omega \models F(e)$ indicates that $o \in \phi$, where $o = \mathcal{E}(e)(\sigma, \tau, \omega)$. For any other monadic predicate X, $\sigma, \tau, \phi, \omega \models X(e)$ indicates that $o \in \omega(X)$, where, as above, $o = \mathcal{E}(e)(\sigma, \tau, \omega)$. We restrict this truth definition to $\sigma, \tau, \phi, \omega$ which are consistent: $\tau(x)$ and $\omega(x)$ denote an object that exists in σ for every object variable x, and ϕ, and $\omega(X)$ for every monadic predicate X, denote a set of existing objects. Quantification over (sets of) objects ranges over (sets of) *existing* objects. Objects other than **null** that do not exist cannot be referred to—neither in the programming language nor in the assertion language. So $\sigma, \tau, \phi, \omega \models \exists x(p)$, where the variable x is of type **object**, holds precisely when there is an object $o \in dom(\sigma)$ such that $\sigma, \tau, \phi, \omega[x := o] \models p$. Here $\omega[x := o]$ results from ω by assigning the object o to the variable x.

By $\{p\}\ S\ \{q\}$ we denote the usual correctness formula in program logics. We distinguish two different interpretations corresponding to the two semantics $\mathcal{M}_{\mathbf{null}}$ and $\mathcal{M}_{\mathbf{F}}$. Both interpretations are *strong partial correctness* interpretations. By $\models_{\mathbf{null}} \{p\}\ S\ \{q\}$ we denote the interpretation: for any $\sigma, \tau, \phi, \omega$ which are consistent, if $\sigma, \tau, \phi, \omega \models p$ then $\mathcal{M}_{\mathbf{null}}(S)(\sigma, \tau) \neq \mathbf{fail}$ and $\sigma', \tau', \phi, \omega \models q$ in case $\mathcal{M}_{\mathbf{null}}(S)(\sigma, \tau) = (\sigma', \tau')$. Note that in this interpretation the predicate F does not have a special meaning. The interpretation $\models_{\mathbf{F}} \{p\}\ S\ \{q\}$ is defined similarly, but with respect to the semantics $\mathcal{M}_{\mathbf{F}}$, which thus requires reasoning about the footprint (in preconditions, postconditions, and loop invariants) to ensure that the statement does not lead to failure. In these specifications one formalizes facts about the contents of the footprint so as to ensure that components do not fail with respect to the strong partial correctness semantics. Note that both interpretations do *not* require reasoning about termination (i.e., absence of divergence).

4 The Hoare Logic of Footprints

We introduce a *hybrid* proof system which integrates reasoning about the two different interpretations of correctness specifications. We prefix Hoare triples by $\vdash_{\mathbf{null}}$ and $\vdash_{\mathbf{F}}$ to distinguish between the correctness specifications interpreted with respect to the semantics $\mathcal{M}_{\mathbf{null}}$ and $\mathcal{M}_{\mathbf{F}}$, respectively. By $\vdash$ we denote either of these two.

We have the standard axiom for (parallel) assignments to program variables, the usual consequence rule, and the standard rules for sequential composition, if-then-else and while statements, for both interpretations of correctness formulas.

To axiomatize field assignments for both of the interpretations, we introduce the substitution $p[f := t]$ which caters for *aliasing*: it replaces every qualified dereferencing expression of the form $x.f$ by the *conditional expression* **if** $x =$ **this then** t **else** $x.f$ **fi**. Note that an expression **this**.f is replaced by **if this** $=$ **this then** t **else this**.f **fi**, which clearly can be avoided by replacing such an expression directly by t. This substitution operator can be easily extended to parallel assignments involving substitution of both fields and variables.

Axiom 1 (GENERAL ASSIGNMENT).

$$\vdash \{p[\bar{u} := \bar{t}]\}\ \bar{u} := \bar{t}\ \{p\}$$

As a simple example, we have the following instantiation

$$\{\mathbf{if}\ x = \mathbf{this\ then}\ 0\ \mathbf{else}\ x.f\ \mathbf{fi} = 1\}\ f := 0\ \{x.f = 1\}$$

of this axiom, where the precondition is logically equivalent to $x \neq \mathbf{this} \wedge x.f = 1$.

To axiomatize object creation for both interpretations of correctness formulas we introduce a substitution operator $p[x := \mathbf{new}, \Phi]$, where Φ denotes a set of monadic predicates with the intention that X holds for the new object (that is, the set denoted by X includes the new object) if and only if $X \in \Phi$. This substitution involves an extension to second-order quantification of the *contextual* analysis of the occurrences of the variable x, as described in [9], assuming that in assertions an object variable x can only be dereferenced, tested for equality, or appear as argument of a monadic predicate. Without loss of generality we restrict the definition of $p[x := \mathbf{new}, \Phi]$ to assertions p which do not contain conditional expressions which have x as argument, since such conditional expressions (that have been introduced by a prior operation to cater for aliasing) can be systematically removed before applying this substitution.

Definition 4.1. We have the following main clauses, where op is an n-ary operation on arguments that are not of type object.

$$\begin{aligned}
x[x := \mathbf{new}, \Phi] &\text{ is left undefined}\\
(x.f)[x := \mathbf{new}, \Phi] &\text{ has a fixed default value}\\
(x = x)[x := \mathbf{new}, \Phi] &= \mathbf{true}\\
(e = x)[x := \mathbf{new}, \Phi] &= \mathbf{false} \text{ if } x \text{ and } e \text{ are distinct}\\
op(e_1, \ldots, e_n)[x := \mathbf{new}, \Phi] &= op(e_1[x := \mathbf{new}, \Phi], \ldots, e_n[x := \mathbf{new}, \Phi])\\
X(x)[x := \mathbf{new}, \Phi] &= \begin{cases} \mathbf{true} & \text{if } X \in \Phi \\ \mathbf{false} & \text{if } X \notin \Phi \end{cases}\\
X(e)[x := \mathbf{new}, \Phi] &= X(e[x := \mathbf{new}, \Phi]) \text{ if } x \text{ and } e \text{ are distinct}\\
(\exists y(p))[x := \mathbf{new}, \Phi] &= p[y := x][x := \mathbf{new}, \Phi] \ \vee \exists y(p[x := \mathbf{new}, \Phi])\\
(\exists X(p))[x := \mathbf{new}, \Phi] &= \exists X(p[x := \mathbf{new}, \Phi \cup \{X\}]) \ \vee \exists X(p[x := \mathbf{new}, \Phi])
\end{aligned}$$

Definition 4.1 lists the main cases. It should be noted that despite that $x[x := \mathbf{new}, \Phi]$ is undefined, $p[x := \mathbf{new}, \Phi]$ *is* defined for every *assertion*. The substitution operations exploit the context in which x appears to find a suitable equivalent assertion which does not contain x. Furthermore, note that $(x = e)[x := \mathbf{new}, \Phi]$, for any logical expression e (syntactically) different from x, reduces to **false** because the value of e is not affected by the execution of $x := \mathbf{new}$ and as such refers after its execution to an 'old' object.

Because of the restriction that quantification over objects ranges over *existing* objects, we have a *changing scope of quantification* when applying the substitution $[x := \mathbf{new}, \Phi]$ to a formula $\exists y(p)$ or $\exists X(p)$. The resulting assertion namely is evaluated *prior* to the object creation, that is, in the heap where the newly created object does not exist yet. The case $(\exists y(p))[x := \mathbf{new}, \Phi]$ handles this changing scope of quantification by distinguishing two cases, as described initially in [9], namely: p is true for the new object, i.e., $p[y := x][x := \mathbf{new}, \Phi]$ holds; or there exists an 'old' object for which $p[x := \mathbf{new}, \Phi]$ holds, i.e., $\exists y(p[x := \mathbf{new}, \Phi])$ holds.

Similarly, $(\exists X(p))[x := \mathbf{new}, \Phi]$ is handled by distinguishing between whether or not X contains the newly created object. Without loss of generality we may assume that $X \notin \Phi$, since otherwise we apply the substitution to an alphabetic variant of $\exists X(p)$. The assertion $p[x := \mathbf{new}, \Phi \cup \{X\}]$ then describes the case that $X(x)$ is assumed to hold for the newly created object x. On the other hand, $p[x := \mathbf{new}, \Phi]$ describes the case that $X(x)$ is assumed not to hold.

Note that $p[x := \mathbf{new}, \emptyset]$ enforces that for all monadic predicates X which occur *free* in p it is assumed that $X(x)$ does *not* hold for the newly created object x. This is in line with the requirement that the interpretation of such a predicate in the state prior to the assignment $x := \mathbf{new}$ only contains objects that exist in that prior state. On the other hand, the substitution $[x := \mathbf{new}, \{F\}]$ formalizes that the newly created object x is *added* to the footprint F. For example, we have that $\models_{\mathrm{F}} \{\mathbf{true}\}\ x := \mathbf{new}\ \{F(x)\}$, and indeed, $F(x)[x := \mathbf{new}, \{F\}] = \mathbf{true}$. For example, we have that $(\forall y(F(y) \leftrightarrow y = x))[x := \mathbf{new}, \{F\}]$ reduces to $(\mathbf{true} \leftrightarrow x = x) \wedge \forall y(F(y) \leftrightarrow \mathbf{false})$, which is logically equivalent to $\forall y(\neg F(y)))$. Note that $\forall y(\neg F(y)))$, which states that the footprint is empty, is indeed the weakest precondition which establishes that the footprint is a singleton set containing x, expressed by $\forall y(F(y) \leftrightarrow y = x)$, after execution of $x := \mathbf{new}$.

We have the following axiomatization of object creation.

Axiom 2 (OBJECT CREATION).

$$\vdash_{\mathbf{null}} \{q[x := \mathbf{new}, \emptyset]\}\ x := \mathbf{new}\ \{q\}\ \textit{and}\ \vdash_{\mathrm{F}} \{q[x := \mathbf{new}, \{F\}]\}\ x := \mathbf{new}\ \{q\}$$

For reasoning about the footprints of (non-recursive) method calls, we have the following rule which guarantees absence of failure in calling an object that does not belong to the footprint.

Rule 3 (METHOD CALL). *Given the method definition $m(\bar{x}) :: S$, we have*

$$\frac{\vdash_{\mathrm{F}} \{p \wedge F(y)\}\ \mathbf{this}, \bar{x} := y, \bar{t};\ S\ \{q\}}{\vdash_{\mathrm{F}} \{p \wedge F(y)\}\ y.m(\bar{t})\ \{q\}}$$

where, as above, the formal parameters (which include the variable **this***) do not occur free in the postcondition q.*

As already discussed in the introduction, apart from the above axioms and rules, we have the usual adaptation rules for both interpretations of correctness formulas (see again [2]). In order to reason *semantically* about invariance in terms of footprints we introduce the *restriction* $p\downarrow$ of a formula p which restricts the quantification over objects in p to the invariant part of the state, namely that part that is *disjoint* from the footprint as denoted by F. For notational convenience, we introduce the complement F^c of F, e.g., $F^c(x)$ then denotes the assertion $\neg F(x)$. This restriction *bounds* every quantification involving objects to those objects which do *not* belong to the footprint. More specifically, $(\exists x(p))\downarrow$ reduces to $\exists x(F^c(x) \wedge (p\downarrow))$, and similarly $(\forall x(p))\downarrow$ reduces to $\forall x(F^c(x) \rightarrow (p\downarrow))$. Quantification over monadic predicates is treated similarly, i.e., $(\exists X(p))\downarrow$ reduces to $\exists X(\forall x(X(x) \rightarrow F^c(x)) \wedge (p\downarrow))$.

We have the following main semantic invariance property of an assertion $p\downarrow$.

Theorem 4.2. *Let p be an assertion which does not contain free occurrences of dereferenced variables. We have $\sigma, \tau, \phi, \omega \models p\downarrow$ iff $\sigma', \tau', \phi', \omega \models p\downarrow$ where*

- *$dom(\sigma) \subseteq dom(\sigma')$,*
- *$\sigma(o) = \sigma'(o)$, for any $o \in dom(\sigma) \setminus \phi$,*
- *$\phi \subseteq \phi'$, $\phi' \cap dom(\sigma) \subseteq \phi$, and $dom(\sigma') \setminus dom(\sigma) \subseteq \phi'$,*
- *$\tau(x) = \tau'(x)$, for any variable x occurring free in p.*

Note that any assertion $p(x)$ which contains free occurrences of the variable x is logically equivalent to $\exists y(y = x \wedge p(y))$, where $p(y)$ results from replacing the free occurrences of x by y. The first three conditions of this theorem describe general properties of the semantics $\mathcal{M}_{\mathrm{F}}(S)(\sigma, \tau, \phi) = (\sigma', \tau', \phi')$. The last condition is true whenever the variables x that occur free in p do not occur in the statement S nor in any of the method bodies of the (indirectly) called methods.

Proof (of Theorem 4.2). *We prove this theorem for assertions p which may contain free occurrences of dereferenced variables, requiring that $\tau(x) \notin \phi$, for any such variable. We proceed by* *induction on the assertion p. We highlight the main case of $\exists x(p)$.*

$$\begin{array}{l}
\sigma, \tau, \phi, \omega \models (\exists x(p))\downarrow \ \textit{iff}\ (\textit{definition}\ (\exists x(p))\downarrow) \\
\sigma, \tau, \phi, \omega \models \exists x(F^c(x) \wedge (p\downarrow))\ \textit{iff}\ (o \in dom(\sigma) \setminus \phi) \\
\sigma, \tau, \phi, \omega[x := o] \models p\downarrow \ \textit{iff}\ (\textit{induction hypothesis}) \\
\sigma', \tau', \phi', \omega[x := o] \models p\downarrow \ \textit{iff}\ (o \in dom(\sigma') \setminus \phi') \\
\sigma', \tau', \phi', \omega \models \exists x(F^c(x) \wedge (p\downarrow))\ \textit{iff}\ (\textit{definition}\ (\exists x(p))\downarrow) \\
\sigma', \tau', \phi', \omega \models (\exists x(p))\downarrow
\end{array}$$

Note that $o \in dom(\sigma) \setminus \phi$ implies that $o \in dom(\sigma') \setminus \phi'$, since $dom(\sigma) \subseteq dom(\sigma')$ and $\phi' \cap dom(\sigma) \subseteq \phi$. On the other hand, $o \in dom(\sigma') \setminus \phi'$ implies $o \in dom(\sigma) \setminus \phi$, because $dom(\sigma') \setminus dom(\sigma) \subseteq \phi'$ and $\phi \subseteq \phi'$. □

Given the above we can now introduce the main two rules to reason about invariance with footprints.

Rule 4 (SEMANTIC INVARIANCE).

$$\frac{\vdash_{\mathbf{F}} \{p\}\ S\ \{q\}}{\vdash_{\mathbf{F}} \{p \wedge r\downarrow\}\ S\ \{q \wedge r\downarrow\}}$$

where the assertion r does not contain free occurrences of dereferenced variables and its program variables do not occur in the implicitly given set of method definitions and the statement S.

Soundness of the above semantic invariance rule follows directly from Theorem 4.2. We conclude with the following meta-rule for *eliminating* footprints.

Rule 5 (FOOTPRINT ELIMINATION).

$$\frac{\vdash_{\mathbf{F}} \{p\}\ S\ \{q\}}{\vdash_{\mathbf{null}} \{\exists F(p)\}\ S\ \{q\}}$$

where the monadic predicate F does not occur in the postcondition q.

Note that this rule allows a modular reasoning about footprints. That is, it caters for the introduction and elimination of footprints at the level of individual statements. Soundness of this rule follows from the main observation that, since F does not appear in the postcondition q, we have that $\models_{\mathbf{F}} \{p\}\ S\ \{q\}$ implies $\models_{\mathbf{null}} \{p\}\ S\ \{q\}$.

5 A Comparison between Related Approaches

The two other main approaches to reasoning about invariant properties of unbounded heaps are that of separation logic [23] and dynamic frames [18]. In this section we illustrate the main differences between our approach and that of separation logic and dynamic frames by means of respective correctness proofs of a stack data structure where items can be added at the top.

To illustrate our approach, we model the stack structure by the two classes *LinkedList* and *Node*. In our framework, classes can be introduced by partitioning the type **object** into disjoint classes of objects. An instance of the class *LinkedList* holds a pointer, stored in field *first*, to the first node of the linked list. Each instance of class *Node* stores a value in field *val* and a pointer stored in field *next* to the next node. The implementation of the *push* method of the class *LinkedList* is shown in Listing 1. It uses as constructor the method *setAttributes*, which stores the pointer to the next node and the desired value. This refactoring of *first := **new** Node(first,v)* enables the separation of concerns between the creation of an object and calling its constructor in the proof theory.

We will focus on proving a natural global reachability property about the nodes in the stack. In particular, we will apply our approach and that of separation logic and dynamic frames to prove that all nodes in the stack that were reachable (starting from the *first* node, by repeatedly following

```
push(v) ::
    u := new Node;
    u.setAttributes(first, v);
    first := u
setAttributes(node, v) ::
    next := node;
    val := v
```

Listing 1. Stack implementation for pushing items

next links) before a call to *push*, are still reachable after the call to *push*. It should be noted that this classic example serves primarily as an illustration of the different approaches. As such it provides a typical example of reasoning about invariant properties of unbounded heaps, and lends itself well for this purpose.

5.1 Footprints

Since we assume an implicit strongly typed assertion language, we introduce for the *LinkedList* and *Node* classes the distinct footprint predicates F_L and F_N which apply to *LinkedList* objects and *Node* objects, respectively. We omit the straightforward details of refining the definition of the proof system to a class-based programming language and an assertion language which features for each class a corresponding footprint predicate. Note that the restriction operator introduces bounded quantification over F_L and F_N, respectively.

We denote by $r'(Y, e, e')$ the assertion

$$[Y(e) \land \forall x(Y(x) \to Y(x.next))] \to Y(e')$$

and by $r(e, e')$ the assertion $\forall Y(r'(Y, e, e'))$ which formalizes reachability of the *Node* object denoted by e' from the one denoted by e via a chain of *next* fields (the predicate Y thus is assumed to range over *Node* objects).

We first show how to derive the global reachability specification

$$\vdash_{\mathbf{null}} \{r(\mathit{first}, y)\}\ \mathbf{this}.\mathit{push}(v)\ \{r(\mathit{first}, y)\} \tag{1}$$

from the local specification

$$\begin{array}{ll} & \{F_L = \{\mathbf{this}\} \land F_N = \emptyset \land \mathit{first} = z\} \\ \vdash_{\mathbf{F}} & \mathbf{this}.\mathit{push}(v) \\ & \{F_N = \{\mathit{first}\} \land \mathit{first} \neq z \land \mathit{first}.\mathit{next} = z \land \forall x(x.\mathit{next} \neq \mathit{first})\} \end{array} \tag{2}$$

and, consequently, we prove the correctness specification 2. Recall that **this**.*first* and *first* denote the same value, and we use basic set-theoretic notations to abbreviate assertions about monadic predicates, e.g., $F_L = \{\mathbf{this}\}$ abbreviates the assertion $F_L(\mathbf{this}) \land \forall x(F_L(x) \to x = \mathbf{this})$. The variables y and z in the correctness specifications (1) and (2) are assumed to be logical variables which do not appear in the definitions of the 'push' and the 'setAttributes' methods, and as such are not affected by the execution of the 'push' method. The variable z is used to 'freeze' the initial value of the field *first*. Note that the correctness specification (2) only describes the local changes affected by execution of the 'push' method in terms of its footprint.

Since the only variable that is dereferenced in the assertion $r(z, y)$ is the bound variable x, and $r(z, y)$ does not refer to program variables, we can apply the semantic invariance rule (Rule 4) to the correctness specification (2), and

$$\vdash_{\mathbf{F}} \{p \land r(z, y)\downarrow\}\ \mathbf{this}.\mathit{push}(v)\ \{q \land r(z, y)\downarrow\}$$

where p denotes the precondition and q denotes the postcondition of the correctness specification (2). By extending its definition to deal with classes, $r(z,y)\downarrow$ reduces to $\forall Y(Y \subseteq F_N^c \rightarrow r'(Y,z,y)\downarrow)$ where $Y \subseteq F_N^c$ abbreviates $\forall x(Y(x) \rightarrow F_N^c(x))$ and F_N^c denotes the complement of F_N, and $r'(Y,z,y)\downarrow$ reduces to $[Y(z) \wedge \forall x(F_N^c(x) \rightarrow (Y(x) \rightarrow Y(x.next)))] \rightarrow Y(y))$.

First we observe that $F_N = \emptyset$ implies $\forall x(F_N^c(x))$ (all existing objects belong to the complement of the footprint), and so the restriction operator has no effect, that is, $r(z,y)\downarrow$ is equivalent to $r(z,y)$.

Furthermore, we show that the assertion $r(z,y)\downarrow$ implies $r(z,y)$, assuming that $F_N = \{first\}$ and $first \neq z \wedge \forall x(x.next \neq first)$. The argument is a straightforward argument in second-order logic: Under the assumption $F_N = \{first\}$, $r(z,y)\downarrow$ is equivalent to

$$\forall Y(Y^c(first) \rightarrow r'(Y,z,y)\downarrow)$$

where $r'(Y,z,y)\downarrow$ reduces to

$$[Y(z) \wedge \forall x(x \neq first \rightarrow (Y(x) \rightarrow Y(x.next)))] \rightarrow Y(y).$$

As above, Y^c denotes the complement of Y: $Y^c(first)$ if and only if $\neg Y(first)$. Let Y be such that $Y(z) \wedge \forall x(Y(x) \rightarrow Y(x.next))$. We show that $Y(y)$. Let $Y' = Y \setminus \{first\}$. Since $first \neq z \wedge \forall x(x.next \neq first)$, it follows that $Y'(z) \wedge \forall x(Y'(x) \rightarrow Y'(x.next))$. From $r(z,y)\downarrow$ it follows that $r'(Y',z,y)\downarrow$, which is equivalent to $r'(Y',z,y)$ (note that the bounded quantification here has no effect). So we have $Y'(y)$, and by definition of Y', we conclude that $Y(y)$.

By the consequence rule the correctness formula for push then reduces to

$$\vdash_{\mathbf{F}} \{F_L = \{\mathbf{this}\} \wedge F_N = \emptyset \wedge first = z \wedge r(z,y)\}\ \mathbf{this}.push(v)\ \{first.next = z \wedge r(z,y)\}.$$

We next observe that $first.next = z \wedge r(z,y)$ implies $r(first,y)$: Let $Y(first)$ and $\forall x(Y(x) \rightarrow Y(x.next))$. From $first.next = z$ it then follows that $Y(z)$. From $r(z,y)$ we then conclude that $Y(y)$.

Thus, we obtain

$$\vdash_{\mathbf{F}} \{F_L = \{\mathbf{this}\} \wedge F_N = \emptyset \wedge first = z \wedge r(z,y)\}\ \mathbf{this}.push(v)\ \{r(first,y)\}$$

by another application of the consequence rule. Finally, substituting *first* for the variable z in the precondition and subsequently eliminating the footprints predicates F_L and F_N, we obtain by a trivial application of the consequence rule the global correctness specification (1).

We now provide a proof of the local correctness specification (2), and let p denote the precondition

$$F_l = \{\mathbf{this}\} \wedge F_n = \emptyset \wedge first = z$$

and q denote the postcondition

$$F_n = \{first\} \wedge first \neq z \wedge first.next = z \wedge \forall x(x.next \neq first).$$

In this proof we omit the prefix $\vdash_F$. By the standard axiom for assignments to variables we have

$$\begin{gathered}\{F_n = \{u\} \wedge u \neq z \wedge u.next = z \wedge \forall x(x.next \neq u)\}\\ first := u\\ \{q\}.\end{gathered}$$

Next we observe that

$$\begin{gathered}\{F_n = \{u\} \wedge u \neq z \wedge\\ \mathbf{if}\ u = \mathbf{this}\ \mathbf{then}\ node\ \mathbf{else}\ u.next\ \mathbf{fi} = z \wedge\\ \forall x(\mathbf{if}\ x = \mathbf{this}\ \mathbf{then}\ node\ \mathbf{else}\ x.next\ \mathbf{fi} \neq u)\}\\ next := node\\ \{F_n = \{u\} \wedge u \neq z \wedge u.next = z \wedge \forall x(x.next \neq u)\}\end{gathered}$$

by Axiom 1 for field assignments. Towards deriving the correctness formula for the method call, we work with the assignment of actual parameters to formal parameters. By the standard axiom for assignments to variables again, we then derive

$$\begin{gathered}\{F_n = \{u\} \wedge u \neq z \wedge \\ \textbf{if } u = \textbf{this then } \mathit{first} \textbf{ else } u.\mathit{next} \textbf{ fi} = z \wedge \\ \forall x(\textbf{if } x = \textbf{this then } \mathit{first} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u)\} \\ \mathit{node} := \mathit{first} \\ \{F_n = \{u\} \wedge u \neq z \wedge \\ \textbf{if } u = \textbf{this then } \mathit{node} \textbf{ else } u.\mathit{next} \textbf{ fi} = z \wedge \\ \forall x(\textbf{if } x = \textbf{this then } \mathit{node} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u)\}.\end{gathered}$$

Substituting **this** by u and a trivial application of the consequence rule we obtain

$$\begin{gathered}\{F_n = \{u\} \wedge u \neq z \wedge \mathit{first} = z \wedge \\ \forall x(\textbf{if } x = u \textbf{ then } \mathit{first} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u)\} \\ \textbf{this} := u \\ \{F_n = \{u\} \wedge u \neq z \wedge \\ \textbf{if } u = \textbf{this then } \mathit{first} \textbf{ else } u.\mathit{next} \textbf{ fi} = z \wedge \\ \forall x(\textbf{if } x = \textbf{this then } \mathit{first} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u)\}.\end{gathered}$$

Putting the above together using the rule for sequential composition and Rule 3 for method calls, we derive

$$\begin{gathered}\{F_n = \{u\} \wedge u \neq z \wedge \mathit{first} = z \wedge \\ \forall x(\textbf{if } x = u \textbf{ then } \mathit{first} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u)\} \\ u.\mathit{setAttributes}(\mathit{first}, v) \\ \{q\}.\end{gathered}$$

Applying Axiom 2, we next have to prove

$$\begin{gathered}\{F_n = \emptyset \wedge \mathit{first} = z\} \\ u := \textbf{new } \mathit{Node} \\ \{F_n = \{u\} \wedge u \neq z \wedge \mathit{first} = z \wedge \\ \forall x(\textbf{if } x = u \textbf{ then } \mathit{first} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u)\}.\end{gathered}$$

The calculation of $(F_n = \{u\})[x := \textbf{new}, \{F_n\}]$ follows the same steps as that of $(\forall y(F(y) \leftrightarrow y = x))[x := \textbf{new}, \{F\}]$, already shown above as an example. We then calculate

$$(\forall x(\textbf{if } x = u \textbf{ then } \mathit{first} \textbf{ else } x.\mathit{next} \textbf{ fi} \neq u))[u := \textbf{new}, \{F_n\}],$$

and skip the straightforward details of the application of the substitution to the remaining conjuncts. Before applying the substitution we first have to remove the conditional expression from the inequality. Here we go:

$$\begin{gathered}(\forall x(\textbf{if } x = u \textbf{ then } \mathit{first} \neq u \textbf{ else } x.\mathit{next} \neq u \textbf{ fi}))[u := \textbf{new}, \{F_n\}] \\ = \\ \textbf{if } u = u \textbf{ then } \mathit{first} \neq u \textbf{ else } u.\mathit{next} \neq u \textbf{ fi}[u := \textbf{new}, \{F_n\}] \wedge \\ \forall x(\textbf{if } x = u \textbf{ then } \mathit{first} \neq u \textbf{ else } x.\mathit{next} \neq u \textbf{ fi}[u := \textbf{new}, \{F_n\}]) \\ = \\ \textbf{if } (u = u)[u := \textbf{new}, \{F_n\}] \textbf{then } (\mathit{first} \neq u)[u := \textbf{new}, \{F_n\}] \textbf{else } (u.\mathit{next} \neq u)[u := \textbf{new}, \{F_n\}] \textbf{ fi} \wedge \\ \forall x(\textbf{if } (x = u)[u := \textbf{new}, \{F_n\}] \textbf{then } (\mathit{first} \neq u)[u := \textbf{new}, \{F_n\}] \textbf{else } (x.\mathit{next} \neq u)[u := \textbf{new}, \{F_n\}] \textbf{ fi}) \\ = \\ \textbf{if true then true else true fi} \wedge \\ \forall x(\textbf{if false then true else true fi})\end{gathered}$$

The resulting assertion is clearly logically equivalent to **true**.

Putting the above together using the rule for sequential composition and Rule 3 for method calls, we finally derive the above correctness specification.

5.2 Dynamic Frames

In this section we discuss a proof of the push operation on our stack-like data structure using the approach of dynamic frames [18] as it is implemented in the KeY tool [1, 24]. The *KeY system* consists of three main components: a multi-sorted classical first-order sequent calculus, a translator of annotated Java programs into proof obligations, and an interactive GUI to build and manipulate proofs.

In the KeY approach, Java programs are typically specified in the **Java Modeling Language (JML)** [19], a well-known specification language that supports method contracts, class invariants and footprints. JML specifications are translated to *JavaDL*, the language of the formulas used in KeY. JavaDL is a program logic that directly incorporates Java language features. More precisely, it is a multi-modal (dynamic) logic, with diamond and box modalities that contain Java fragments: $\langle P\rangle\varphi$ means that executing program fragment P terminates and φ holds in the final state; and $[P]\varphi$ means that *if* program fragment P terminates, *then* φ holds in the final state. The total correctness of a Hoare triple of the form $\{\phi\}P\{\psi\}$ is expressible in JavaDL simply by the formula $\varphi \rightarrow \langle P\rangle\psi$, which means that if φ holds in the initial state, then execution of P terminates in a state for which ψ holds. The possibility to express Hoare triples as formulas yields a rather expressive formalism. For instance, in contrast to Hoare logic where pre and postconditions are ordinary first-order formulas, in JavaDL this can be any JavaDL formula, hence it is possible to use a Hoare triple as a precondition of another Hoare triple. See Chapter 3 of [1] for a detailed exposition of JavaDL's syntax and semantics. Below we give an introduction to the basic concepts of KeY.

KeY's program logic distinguishes *program variables* from *logical variables*: the value of a program variable can be changed over the course of executing a program, whereas a logical variable always has the same value. We shall write program variables and fragments in a typewriter font and logical variables in italic. For example, `i`, `j` are program variables (of integer sort) and z is a logical variable (also of integer sort). Then the formula

$$i = z \rightarrow \langle j = i; i = 1; i = j; \rangle i = z$$

means that execution of the program fragment terminates and the value of i is the same in the initial and the terminal state. This is expressed by using z as a *freeze variable*: it can never be modified by a program fragment.

The proof system that KeY uses to establish validity of formulas is given as a sequent calculus. A sequent $\varphi_1, \ldots, \varphi_n \Rightarrow \psi_1, \ldots, \psi_m$ consists of n antecendents and m succedents, all formulas of JavaDL. The sequent is true if and only if the conjunction of all antecedents implies the disjunction of the succedents. Next to the rules for classical first-order logic, the proof system also consists of a large number of other rules. The large rule set is given by means of lightweight tactics (called taclets [4]) that perform modifications on the sequent one is proving: e.g., spawn new branches, substitute variables, rewrite terms, or close a branch. There are approximately 1,750 rules that implement Java program semantics to make symbolic execution possible, and implement the theory of the sorts: integers, sequences, heaps, location sets, and others.

Of particular interest to us are rules concerning *updates* and *heaps*. There are rules that transform modalities with program fragments into so-called update modalities. Updates always terminate and they assign program variables to JavaDL terms. As such, updates cannot assign program variables to side-effectful expressions. Given a formula φ, then $\{\mathtt{x}_1 := t_1 || \ldots || \mathtt{x}_n := t_n\}\varphi$ is a formula where in parallel $\mathtt{x}_i$ are updated to t_i. Moreover, as program variables have an integer

$$
\dfrac{\dfrac{\dfrac{\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{i} = z \Rightarrow \mathtt{i} = z}}{\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\mathtt{j} = z}\,\mathsf{applyUpd}}{\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\{\mathtt{i} \coloneqq 1\}\mathtt{j} = z}\,\mathsf{simp}}{\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\{\mathtt{i} \coloneqq 1\}\{\mathtt{i} \coloneqq \mathtt{j}\}\mathtt{i} = z}\,\mathsf{applyUpd}}{\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\{\mathtt{i} \coloneqq 1\}\{\mathtt{i} \coloneqq \mathtt{j}\}\langle\rangle\mathtt{i} = z}\,\mathsf{empty}}{\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\{\mathtt{i} \coloneqq 1\}\{\mathtt{i} \coloneqq \mathtt{j} || \mathtt{j} \coloneqq \mathtt{j} \oplus 1\}\langle\rangle\mathtt{i} = z}\,\mathsf{simp} \\
\vdots \\
\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\{\mathtt{i} \coloneqq 1\}\langle \mathtt{i} = \mathtt{j}; \rangle\mathtt{i} = z
\end{array}}{\mathtt{i} = z \Rightarrow \{\mathtt{j} \coloneqq \mathtt{i}\}\langle \mathtt{i} = 1; \mathtt{i} = \mathtt{j}; \rangle\mathtt{i} = z}\,\mathsf{assign}}{\mathtt{i} = z \Rightarrow \langle \mathtt{j} = \mathtt{i}; \mathtt{i} = 1; \mathtt{i} = \mathtt{j}; \rangle\mathtt{i} = z}\,\mathsf{assign}}{\Rightarrow \mathtt{i} = z \rightarrow \langle \mathtt{j} = \mathtt{i}; \mathtt{i} = 1; \mathtt{i} = \mathtt{j}; \rangle\mathtt{i} = z}\,\mathsf{impRight}
$$

Fig. 1. Example of part of a proof in JavaDL.

sort that is interpreted as the standard infinite set of integers, Java integer expressions must be translated to correctly model integer overflow (Section 5.4.3 of [1]).

To prove the validity of a formula, we create a sequent with only that formula as succedent. If rule applications lead to branches which are all closed, the formula is proven. Part of the proof of our previous example is shown in Figure 1.

Here, the impRight rule comes from standard classical sequent calculus. The assignment rule transforms the program modality into an update modality. The empty rule removes the empty program modality. The function $\oplus$ models Java's overflow behavior on `int`. The simplification rule removes any update that has no effect on the formula on its right. Only the right-most updates can be applied to the formula.

The KeY proof of our case study can be found in the artifact [6] accompanying this paper, which includes user-defined taclets (describing inference rules in the KeY system) that we used to define the reachability predicate. Also, a video recording [5] shows the steps for reproducing the proof of invariance of reachability over the push method using KeY. This section describes at a more abstract level the basic ideas underlying dynamic frames, in an extension of Hoare logic (instead of dynamic logic) which allows for a better comparison with our footprints approach. As such, we restrict to coarse-grained dynamic frames, similar to our footprints (KeY supports dynamic frames on the finer granularity of fields, see Section 7 for a brief discussion on extending the granularity of footprints in our approach).

In the dynamic frames approach, footprints are introduced by extending contracts in JML with an assignable clause. For example, the clause **assignable** $\{\mathbf{this}\}$ states that only the fields of the object **this** can be modified. The corresponding proof obligation then requires to prove that the initial and final heap (of an execution of $\mathbf{this}.push(v)$) differ at most with respect to the values of field $first$. To formally express this proof obligation in our assertion language (introduced in Section 3), we take a snapshot of the entire initial heap and store it in a logical variable: this allows one to refer in the postcondition to the initial heap and express its relation with the final heap.

Thus, we extend the assertion language with heap variables, with typical element h. Given a current heap σ and environment ω, for any heap variable h, $\omega(h)$ denotes a heap (as defined in Section 2) such that its domain is included in that of σ, i.e., $dom(\omega(h)) \subseteq dom(\sigma)$. Given a heap variable h the expression $h(x.f)$ denotes the result of the lookup of field f of object x in heap h, and $h(f)$ abbreviates $h(\mathbf{this}.f)$. Furthermore, we introduce the binary predicate $x \in dom(h)$ which means that the object denoted by x is in the domain of the heap denoted by h. If the object denoted by x is not in the domain of the heap denoted by h, then $h(x.f)$ refers to the value of f in the current heap, denoted by $x.f$.

Given the above, we define the assertion $\mathit{init}(h)$ as $\forall x(x \in \mathit{dom}(h) \wedge \bigwedge_f x.f = h(x.f))$ which expresses that heap variable h stores the current heap, that is, $\sigma, \tau, \phi, \omega \models \mathit{init}(h)$ if and only if $\omega(h) = \sigma$. We note that $\bigwedge_f$ ranges over finitely many field names. We can now present a high-level proof in Hoare logic of our case study, using the above extension of the assertion language with heap variables. We again denote by $r(e, e')$ the reachability assertion defined above. Our goal is

$$\{r(\mathit{first}, y)\}\ \mathbf{this}.\mathit{push}(v)\ \{r(\mathit{first}, y)\}.$$

To derive our goal, we first derive the local specification

$$\begin{array}{c}\{\mathit{init}(h)\}\\ \mathbf{this}.\mathit{push}(v) : \mathbf{assignable}\ \{\mathbf{this}\}\\ \{\mathit{first}.\mathit{next} = h(\mathbf{this}.\mathit{first}) \wedge \mathit{first} \notin \mathrm{dom}(h)\}.\end{array} \tag{3}$$

This specification makes use of an assignable clause which requires a proof of

$$\{\mathit{init}(h)\}\ \mathbf{this}.\mathit{push}(v)\ \left\{q \wedge \forall x\left(x \neq \mathbf{this} \rightarrow \bigwedge_f x.f = h(x.f)\right)\right\}$$

where q denotes the postcondition of the specification (3). The proof of the latter specification is easily derived by inlining the method bodies (in KeY, this proof obligation can even be done automatically).

Next we apply a new invariance rule for heap variables which states the invariance of assertions which are independent of the current heap. Let $r_h(\mathit{first}, y)$ denote the reachability assertion

$$\forall Y\big([Y(h(\mathit{first})) \wedge \forall x(x \in \mathit{dom}(h) \rightarrow (Y(x) \rightarrow Y(h(x.\mathit{next}))))] \rightarrow Y(y)\big)$$

This assertion does not depend on the current heap and thus is invariant. By the conjunction rule, we then obtain

$$\begin{array}{c}\{r_h(\mathit{first}, y) \wedge \mathit{init}(h)\}\\ \mathbf{this}.\mathit{push}(v)\\ \{r_h(\mathit{first}, y) \wedge q \wedge \forall x(x \neq \mathbf{this} \rightarrow \bigwedge_f x.f = h(x.f))\}\end{array} \tag{4}$$

That the postcondition of (4) implies $r(\mathit{first}, y)$ can be established as in the previous proof, and similarly that $r(\mathit{first}, y)$ implies the precondition of (4) with the heap variable h existentially quantified. Thus an application of the consequence rule, and the existential elimination rule of variable h, concludes the proof.

Let us remark one difference with the presentation in this section and the one in the KeY system. Instead of the conjunction over all fields, KeY uses heap updates in the formalization of the assignable clause by so-called 'anonymizing updates' [1, Sect. 9.4.1]. For example, consider a call to a method with an assignable clause $\{\mathbf{this}.*\}$ and let *heap* be the heap just before the method call. Informally, the assignable clause means that all fields of the current object may change, but no fields of other objects (that existed just before the method executed). In KeY's logic, the heap resulting after the method call is expressed as $\mathit{heap}[\mathit{anon}(\mathit{self}.*, \mathit{anon\_heap})]$. When evaluating a field access $x.f$ in the heap $\mathit{heap}[\mathit{anon}(\mathit{self}.*, \mathit{anon\_heap})]$, the evaluation rules for heaps in KeY essentially perform an aliasing analysis, e.g., checking whether $x = \mathbf{this}$. If x is not equal to **this**, the value of $x.f$ is $\mathit{heap}(x.f)$, so any information known about $x.f$ before the method was called is still valid. If x *is* equal to **this**, the value of $x.f$ is $\mathit{anon\_heap}(x.f)$; a fresh anonymous heap, so only facts stated in the postcondition of the method contract about $x.f$ are known and can be used after a call to the method. As a program (fragment) is symbolically executed in KeY, updates to the heap accumulate, resulting in new, larger heap terms. For real-world programs, the heap terms tend to get large. In our case study, we get the heap terms after symbolical execution that are shown below.

$h, s \models b$ iff $s(b) = \mathbf{true}$,
$h, s \models (e \mapsto e')$ iff $dom(h) = s(e)$ and $h(s(e)) = s(e')$,
$h, s \models (e \hookrightarrow e')$ iff $s(e) \in dom(h)$ and $h(s(e)) = s(e')$,
$h, s \models (p \wedge q)$ iff $h, s \models p$ and $h, s \models q$,
$h, s \models (p \vee q)$ iff $h, s \models p$ or $h, s \models q$,
$h, s \models (p \rightarrow q)$ iff $h, s \models p$ implies $h, s \models q$,
$h, s \models \exists x p$ iff $h, s[x := n] \models p$ for some n,
$h, s \models \forall x p$ iff $h, s[x := n] \models p$ for all n,
$h, s \models (p * q)$ iff $h_1, s \models p$ and $h_2, s \models q$ for some h_1, h_2 such that $h = h_1 \uplus h_2$,
$h, s \models (p -\!\!* \, q)$ iff $h', s \models p$ implies $h'', s \models q$ for all h', h'' such that $h'' = h \uplus h'$.

Fig. 2. Semantics of separation logic assertions.

```
self_25.y@heap[anon(self_25.*,anonOut_heap<<anonHeapFunction>>)]=null
|reachable(heap[anon(self_25.*,anonOut_heap<<anonHeapFunction>>)],
self_25.first@heap[anon(self_25.*,anonOut_heap<<anonHeapFunction>>)],
self_25.y@heap[anon(self_25.*,anonOut_heap<<anonHeapFunction>>)])
```

Listing 2. The heap term in the case study.

Consequently, it may require many proof steps to reason about the value of fields in such heaps. For example, in the case study [13], a two-dimensional array was used to implement and verify the radix-sort algorithm and the heap terms grew several pages long in some cases.

5.3 Separation Logic

In this section we present a proof in separation logic of the push operation of our stack data structure. For various classes of programs, there are different corresponding separation logics. We shall use the proof system as presented by Reynolds in [22].

We will first give a brief introduction to separation logic. Separation logic [22] is an extension of Hoare logic with predicates, connectives and proof rules for reasoning about programs with dynamic memory allocation and de-allocation, i.e., programs that manipulate the heap. The heap is a partial function $h : \mathbb{N} \rightharpoonup \mathbb{Z}$ that maps a memory location (locations are represented by natural numbers) to a (integer) value. If in a heap h the location n is not allocated, $h(n)$ is undefined; this is why h is a *partial* function. The domain of a heap h is the set of locations (natural numbers) for which h *is* defined. To specify heap manipulating programs, the assertion language contains several additional predicates and connectives compared to the usual first-order assertions that can be used as pre- and postconditions of ordinary Hoare triples. The assertion language is described by the following grammar:

$$p ::= b \mid (e \mapsto e) \mid (e \hookrightarrow e) \mid (p \wedge p) \mid (p \rightarrow p) \mid \forall x p \mid (p * p) \mid (p -\!\!* \, p)$$

The formal meaning of these assertions is given in Figure 2, where h is a heap and s is a state. States map variable names to values.

Informally the predicate $e \hookrightarrow e'$ is true if and only if the heap at location e has the value e'. The separating conjunction allows to specify properties of a partition of the heap into two disjoint sub-heaps. The separating implication (also called 'the magic wand') allows to express properties of disjoint extensions of the heap. Both separating connectives involve a second-order quantification over heaps (which are represented as particular binary relations).

We consider a basic programming language with the core rules of separation logic. These core rules are shared by many of the separation logic 'versions' that have extended language features, e.g., separation logic suitable for Java [14], and tools for verifying (Java) programs using separation logic such as VeriFast [17] and VerCors [7]. For example, VerCors can be used to prove the correctness of a (concurrent wait-free) push explicitly using permissions.[2] The frame rule shown below is an example of a core proof rule to reason about footprints in separation logic.

Rule 6 (FRAME RULE).

$$\frac{\{p\}\ S\ \{q\}}{\{p * r\}\ S\ \{q * r\}} \quad \textit{where no variable in } r \textit{ is modified by } S$$

The frame rule allows to extend a specification by adding an arbitrary predicate r about variables and parts of the heap that are not modified by S.

Instead of object orientation, Reynolds' logic uses a fairly low-level programming language with allocation ($x := \mathbf{cons}(e_1, e_2)$), de-allocation ($\mathbf{dispose}(e)$) and several pointer operations, such as reading ($x := [e]$) or writing ($[e] := e'$) to a location that a pointer variable references. In this setting, object creation can be modeled by allocation, and accessing a field of an object can be modeled by accessing a location referenced by a pointer. See, e.g., [21] for a treatment of extending separation logic to object-oriented programming, including features such as encapsulation, subclasses, inheritance, and dynamic dispatch.

Before we present the proof, we therefore first write the push operation in Reynolds' language, as follows. Note that in our approach we have separated the concerns for object creation and calling its constructor, but here these two happen at the same time by one assignment.

```
push(v) ::
  first := cons(v, first)
```

Listing 3. Stack implementation for pushing items in separation logic

The cons operation allocates a new cell in memory where two values are stored: the first component is the value of the item to push (denoted by the formal parameter v), and the second component is a pointer to the 'node' storing the next value. Following Reynolds, reachability can then be expressed in separation logic by the following recursively defined predicates:

$$\begin{aligned} reach(begin, other) &\equiv \exists n(n \geq 0 \wedge reach_n(begin, other)) \\ reach_0(begin, other) &\equiv begin = other \\ reach_{n+1}(begin, other) &\equiv \exists v, next[(begin \mapsto (v, next)) * reach_n(next, other)] \end{aligned}$$

The $*$ operation is the so-called separating conjunction: informally it means that the left and right sub-formulas should be satisfied in disjoint heaps (see Figure 2). A heap in this context is a *partial* function from addresses to values. Two heaps are disjoint if their domains are disjoint. Furthermore, the formula $begin \mapsto (v, next)$ asserts that the heap contains precisely one cell (namely, the one at the address stored in the variable $begin$), and that this cell contains the values v and $next$.

The above global correctness specification (1) translates to the following main global correctness property in separation logic

$$\{reach(\mathit{first}, y)\}\ push(v)\ \{reach(\mathit{first}, y)\}. \tag{5}$$

Furthermore, we have the following local specification in separation logic

$$\{\mathit{first} = z \wedge \mathbf{emp}\}\ push(v)\ \{\mathit{first} \mapsto (v, z)\} \tag{6}$$

[2]https://vercors.ewi.utwente.nl/try_online/example/82

of the push method corresponding to the above local specification (2) which uses footprints. Here **emp** denotes the empty heap. In fact, at a deeper level, it can be shown that this information corresponds to the assertion $F_N = \emptyset$ (this is further discussed in Section 7).

Applying the frame rule of separation logic with the 'invariance' formula $reach(z, y)$ to the local specification then yields:

$$\{first = z \wedge \mathbf{emp} * reach(z, y)\}\ push(v)\ \{first \mapsto (v, z) * reach(z, y)\}.$$

Using the rule for Auxiliary Variable Elimination of separation logic, we infer:

$$\{\exists z(first = z \wedge \mathbf{emp} * reach(z, y))\}\ push(v)\ \{\exists z(first \mapsto (v, z) * reach(z, y))\}.$$

We show that the global correctness formula (5) follows from an application of the consequence rule to the following correctness formula:

$$\{\exists z(first = z \wedge \mathbf{emp} * reach(z, y))\}\ push(v)\ \{\exists z(first \mapsto (v, z) * reach(z, y))\}.$$

— First we show that $reach(first, y)$ (the desired precondition) implies the precondition above. In the precondition above, substituting z for $first$ (equals for equals) the existential quantifier in the precondition can be eliminated so the precondition above is equivalent to $\mathbf{emp} * reach(first, y)$. Further, by the equivalence $\mathbf{emp} * p \leftrightarrow p$ and commutativity of the separating conjunction, the clause with the empty heap can be eliminated. Hence, $reach(first, y)$ implies the precondition above (they are equivalent).

— Next we show that the postcondition implies the assertion $reach(first, y)$. Observe that the postcondition and the definition of $reach(z, y)$ imply that for some z_0 and n_0: $(first \mapsto (v, z_0)) * reach_{n_0}(z_0, y)$. So, by the definition of $reach_n$ we have $reach_{n_0+1}(first, y)$, which by definition of the *reach* predicate implies the desired postcondition $reach(first, y)$.

We conclude this case study of separation logic with the observation that the above local specification of the push method (6) follows from the Allocation (local) axiom of separation logic: $\{first = z \wedge \mathbf{emp}\}\ first := cons(v, first)\ \{first \mapsto (v, z)\}$.

6 Discussion

All the above three correctness proofs are based on the application of some form of frame rule to derive the invariance of the global reachability property from a local specification of the push method. In our approach we express such properties by restricting quantification to objects which do not belong to the footprint which includes all objects that can be affected by the execution. In the approach based on dynamic frames such footprints are used in a general semantic definition of a relation between the initial and final heap, where to prove an invariant property one has to show that it is preserved by this relation. Invariance in separation logic is expressed by a separating conjunction which allows to express invariant properties of disjoint heaps.

Summarizing, in both our approach and that of separation logic, invariant properties are so *by definition*: in separation logic by the use of the separating conjunction, in our approach by bounded quantification. The general semantics of the separating conjunction and bounded quantification *ensures* invariance, which thus does not need to be established for each application of the frame rule separately. In contrast, the approach of dynamic frames *requires* for each property an ad-hoc proof of its invariance.

In separation logic [23] invariant properties are specified and verified with the frame rule which uses the separating conjunction to *split* the heap into two disjoint parts. As a consequence, the assertion language of separation logic by its very nature supports reference to locations that are not allocated (or objects that do not exist). Furthermore, because of the restriction to *finite* heaps, the validity of the first-order language restricted to the so-called ‘points to’ predicate is non-compact, and as such not recursively enumerable (see [8]).

In contrast, our approach is based on standard (second-order monadic) predicate logic which allows for the use of established theorem proving techniques (e.g., Isabelle/HOL). Furthermore, as discussed above, our approach allows for reasoning at an abstraction level which coincides with the programming language, i.e., it does not require special predicates like the ‘points to’ predicate or ‘dangling pointers’ which are fundamental in the definition of the separating conjunction and implication.

Dynamic frames [18] were introduced as an extension of Hoare logic where an explicit heap representation as a function in the assertion language is used to describe the parts of the heap that are modified. As footprints in our approach, a dynamic frame is a specification of a set of objects that can be modified. This specification itself may, in general, involve dynamic properties of the heap. Since it is impossible to capture an *entire* heap structure in a fixed, finite number of first-order freeze variables, heap variables are introduced. Such variables allow to store the entire initial heap in a single freeze variable (which does not occur in the program), and thus describe the modifications as a relation between the initial and the final heap. In this bottom-up approach invariant properties then are verified directly in terms of this relation between the initial and the final heap. This low-level relation, which itself is not part of the program specification, in general complicates reasoning. In contrast, in our approach we incorporate footprints in the Hoare logic of *strong partial correctness*: a footprint simply *constraints* the program execution. This allows the formal verification of invariant properties in a compositional, top-down manner by an extension of the standard invariance rule. As such this rule abstracts from the relation between the initial and the final heap as specified by a footprint: it is only used in proving its *soundness*, and *not* in program verification itself.

Currently, the KeY system is based on the dynamic frames approach. What changes are needed to KeY to allow reasoning following our footprint logic approach? There are three main challenges: going from first-order logic to higher-order logic, introducing a new substitution operator, and modeling the different semantics. (1) KeY is based on a first-order logic, but our approach is a second-order logic. As such, we would need to extend KeY to be able to represent second-order quantification. Although we made initial steps towards implementing this feature, we found out that the implementation of KeY is brittle: changing the fundamental structure of logical formulas potentially has far reaching consequences on the stability and correctness of the rest of the system. For example, during our initial steps of implementation we triggered many bugs in the taclet implementation, the implementation of the sequent calculus, and the user interface, all because of the implicit assumption that the underlying logic is first-order. Changing KeY to deal with second-order formulas seems to require a rework of a large part of its implementation. (2) Our approach introduces a new substitution operator (Definition 4.1) which needs to be encoded in KeY either by using taclets or by hard-coding the substitution operator in the system. This work follows much of the prior work in adding a substitution operator for abstract object creation [12]. Different than in [12], our new substitution operator takes as a parameter a set of second-order variables. As such, only after the first challenge has been resolved, can the implementation of the substitution operator commence. (3) Our approach introduces two different semantics: failure-sensitive semantics with respect to a given footprint, and the usual failure semantics. To be able to add the footprint elimination rule, that relates the two semantics, it is necessary to use two different program modalities, each corresponding to the different program semantics. This potentially leads to a duplication of, or the need to change, the existing taclet rule base of KeY.

7 Conclusion

In this paper, we introduced a program logic extended with footprints and a novel invariance rule for reasoning about invariant properties of unbounded heaps. This invariance rule allows to adapt correctness specifications (contracts), abstracting from the internal implementation details.

We kept the programming language small but expressive, and our technique is powerful enough to allow application to general OO-languages (as long as the expressions and formulas to specify footprints are sufficiently expressive). As an example of an operation on objects that is not part of our language (but can be handled): many mainstream OO-languages contain an instanceof operator that evaluates whether a value is part of a given (class) type. Our language restricts to equality as the only operation on objects. However, our substitution operator can be extended in a straightforward manner to support instanceof: $[x := \mathbf{new} D, \{F\}](x \text{ instanceof } C)$ reduces to true if D is a subclass of C (this can be determined statically from the type hierarchy) and false otherwise.

Keeping the language small allowed us to focus on several core challenges: developing a logic that (1) avoids explicit heaps, (2) uses *abstract* object creation (informally this means: objects that do not *yet* exist cannot be referred to in the assertion language), (3) interprets assertions using (the standard) Tarski's truth definition, and (4) features a frame rule to reason about invariance. Our logic satisfies all four of the above features, as opposed to the main existing approaches of separation logic and dynamic frames. Moreover, our hybrid program logic combines two different forms of strong partial correctness specifications. This combination allows for modular/compositional reasoning about footprints, which can be introduced and eliminated on the level of individual statements. We further discussed the differences between our approach and that of separation logic and dynamic frames in terms of the underlying assertion languages.

We further observe that our approach allows for a *dual* form which instead of focusing on objects that may be affected focuses on objects that remain invariant. This dual approach is formalized in terms of a monadic predicate I which includes those objects that cannot be called (and thus do not belong to the footprint). We then simply require that in the rule for reasoning about the strong partial correctness of method calls the precondition of a call $y.m(\bar{t})$ implies that $\neg I(y)$ (instead of $F(y)$). Further in the semantic invariance rule we restrict the quantification to I. Interestingly, this dual approach allows for a *single* axiom for object creation because we no longer need to add the newly created object to the footprint F. By definition it does not belong to I (because I only refers to already existing objects), and therefore we have one object creation axiom for both interpretations of correctness specifications considered in this paper.

Using results from the paper [12], on the verification of object-oriented programs with classes and abstract object creation (but without footprints), the logic in this paper can be further extended in a straightforward manner to cover nearly the entire sequential subset of Java, including other failures and exceptions generated by other language features. That paper also showed how to mechanize proof rules for abstract object creation in the KeY system. Along the same lines, as future work, we aim to extend the KeY system with a mechanized version of the logic in this paper, but this will require a considerable re-implementation of the software as discussed in Section 6.

We also plan to investigate completeness of our extension of Hoare logic for object-oriented programs with footprints. Since it is straightforward to apply the transformational approach of [3] to prove (relative) completeness of the logic *without* the semantic invariance rule, we will develop an appropriate notion of *modular* completeness. The general notion of modular completeness allows for the "adaptation of a given specification to specifications needed in particular circumstances (depending on their application)" [25]. It is worthwhile to observe here that in [15] a completeness proof is given of separation logic for pointer programs with mutual recursive procedures which does not require the use of the frame rule, but also uses the standard invariance rule.

Finally, we also plan on investigating a Hoare logic that supports abstract object deletion and garbage collection, as these features are commonly implemented in object-oriented programming languages. In that setting, we restrict quantification not only to objects that are actually created, but furthermore restrict the domain of quantifiers to range over reachable objects. This may compromise the soundness of the rules as we presented them in this paper, since assignment to fields can cause objects to become unreachable and hence properties which use those objects as witness become invalid.

Acknowledgments

We are grateful for the constructive feedback that was given by the anonymous reviewers.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification: The KeY Book*. LNCS, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

[2] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs*. Springer.

[3] Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, and Stijn de Gouw. 2012. Verification of object-oriented programs: A transformational approach. *J. Comput. Syst. Sci.* 78, 3 (2012), 823–852.

[4] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, and Steffen Schlager. 2004. Taclets: A new paradigm for constructing interactive theorem provers. *RACSAM* 98, 1 (2004), 17–53.

[5] Jinting Bian and Hans-Dieter A. Hiep. 2021. *Reasoning about Invariant Properties of Object-oriented Programs - Dynamic Frames*. FigShare. https://doi.org/10.6084/m9.figshare.16782667.v1 Available at: https://doi.org/10.6084/m9.figshare.16782667.v1

[6] Jinting Bian and Hans-Dieter A. Hiep. 2021. *Reasoning about Invariant Properties of Object-oriented Programs-dynamic Frames: Proof Files*. Zenodo. https://doi.org/10.5281/zenodo.6044345 Available at: https://doi.org/10.5281/zenodo.6044345

[7] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors tool set: Verification of parallel and concurrent software. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10510)*, Nadia Polikarpova and Steve A. Schneider (Eds.). Springer, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7

[8] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. 2001. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, Ramesh Hariharan, V. Vinay, and Madhavan Mukund (Eds.). Lecture Notes in Computer Science, Vol. 2245. Springer Berlin, 108–119.

[9] Frank S. de Boer. 1999. A WP-calculus for OO. In *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. 135–149.

[10] Frank S. de Boer and Stijn de Gouw. 2015. Being and change: Reasoning about invariance. In *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. 191–204.

[11] Frank S. de Boer, Stijn de Gouw, Hans-Dieter A. Hiep, and Jinting Bian. 2022. Footprint logic for object-oriented components. In *International Conference on Formal Aspects of Component Software*. Springer, 141–160.

[12] Stijn de Gouw, Frank S. de Boer, Wolfgang Ahrendt, and Richard Bubel. 2016. Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic. *Software and Systems Modeling* 15, 4 (2016), 1117–1140.

[13] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. 2016. Verification of counting sort and radix sort. In *Deductive Software Verification - The KeY Book - From Theory to Practice*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). Lecture Notes in Computer Science, Vol. 10001. Springer, 609–618. https://doi.org/10.1007/978-3-319-49812-6_19

[14] Dino Distefano and Matthew J. Parkinson. 2008. jStar: Towards practical verification for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 213–226. https://doi.org/10.1145/1449764.1449782

[15] Mahmudul Faisal Al Ameen and Makoto Tatsuta. 2016. Completeness for recursive procedures in separation logic. *Theoretical Computer Science* 631 (2016), 73–96. https://doi.org/10.1016/j.tcs.2016.04.004

[16] C. A. R. Hoare. 1971. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*. 102–116.

[17] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[18] Ioannis T. Kassios. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 268–283. https://doi.org/10.1007/11813040_19

[19] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 1999. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, Haim Kilov, Bernhard Rumpe, and Ian Simmonds (Eds.). The Kluwer International Series in Engineering and Computer Science, Vol. 523. Springer, 175–188. https://doi.org/10.1007/978-1-4615-5229-1_12

[20] Ernst-Rüdiger Olderog. 1983. On the notion of expressiveness and the rule of adaption. *Theor. Comput. Sci.* 24 (1983), 337–347.

[21] Matthew Parkinson and Gavin Bierman. 2013. Separation logic for object-oriented programming. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification* (2013), 366–406.

[22] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[23] John C. Reynolds. 2005. An overview of separation logic. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. 460–469.

[24] Benjamin Weiß. 2011. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. Ph. D. Dissertation. Karlsruhe Institute of Technology.

[25] Job Zwiers, Ulrich Hannemann, Yassine Lakhnech, Willem P. de Roever, and Frank A. Stomp. 1996. Modular completeness: Integrating the reuse of specified software in top-down program development *(Lecture Notes in Computer Science, Vol. 1051)*, Marie-Claude Gaudel and Jim Woodcock (Eds.). Springer, 595–608.

Received 31 May 2023; revised 31 July 2024; accepted 29 October 2024