Footprint Logic for Object-Oriented Components

Frank S. de Boer[1,2], Stijn de Gouw[3,1], Hans-Dieter A. Hiep[1,2][0000−0001−9677−6644], and Jinting Bian[1,2][0000−0001−5003−598X]

[1] Centrum Wiskunde & Informatica, the Netherlands {frb,hdh,j.bian}@cwi.nl
[2] Leiden Institute of Advanced Computer Science (LIACS), the Netherlands
[3] Open Universiteit, the Netherlands sdg@ou.nl

Abstract. 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.

Keywords: Hoare logic, invariance, strong partial correctness

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 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*. 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. Hoare introduced in [10] one rule, *the* adaptation rule, which generalizes all adaptation rules. In his seminal paper [13], 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. The adaptation rules, including Hoare's adaptation rule [10], 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.

In this paper we introduce a novel Hoare logic for reasoning about invariant properties using footprints as they arise in object-oriented components, viz. components of object-oriented programs. 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.

We therefore generalize the weakest precondition calculus underlying the Hoare logic for object-orientation introduced in [7], 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*. 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 [15] and dynamic frames [12], showing the differences and commonalities between the three approaches by proving a main invariant property of the push operation on a stack data structure.

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

2 The Programming Language

To focus on the main features of our approach, we consider a basic object-oriented programming language. This model 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 model, 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 field are implicitly typed. Expressions $t, t_0, \ldots$ are constructed from program variables, fields, and built-in operations. 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).

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

In the assignment $u := t$, the left-hand side u denotes a variable or a field and t is an expression of the same type. For technical convenience (and without loss of generality), we restrict to object creation statements $x :=$ **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 ($\bar{t}$ 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 $x_1, \ldots, x_n \in \mathit{Var}$ (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.

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 $\textbf{null} \in O$. Each object has its own local state which assigns values to its fields. A *heap* $\sigma \in \Sigma$ is a *partial* function, i.e. $\Sigma = O \rightharpoonup (\mathit{Field} \to V)$, 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 $dom(\sigma)$ we denote the set of existing objects, that is, the domain of σ consists of the objects o for which $\sigma(o)$ is defined. A *store* is a function $\tau \in T$ that assigns values to variables, i.e. $T = Var \to V$. 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 variables of type **object** only to 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 it does not terminate properly, or it denotes a single final configuration of a properly terminating computation.

Following [2], we have the following partial correctness 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. The *parallel assignment* $\mathbf{this}, \bar{x} := y, \bar{t}$ initializes the variables $\mathbf{this}, \bar{x}$ of S, and the parallel assignment $\mathbf{this}, \bar{x} := \tau(\mathbf{this}), \tau(\bar{x})$ restores the initial values of the variables $\mathbf{this}, \bar{x}$. For further details of the program semantics we refer to [2, p.206].

To observe *failures* due to calling a method on **null** (which generates a 'null-pointer exception'), we introduce a so-called strong partial correctness 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 $\tau(y)$ 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 (which thus means failures arise from calling a method on **null**). Another crucial characteristic of our model is that newly created objects are naturally considered callable, and therefore new objects are added to the footprint ϕ, 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, i.e. $\sigma(o) = \bot$ and $o \neq \mathbf{null}$, and σ' results from σ by initializing the local state of the newly created object o.

3 The Specification Language

Given a set of method definitions, we specify the correctness of a statement by a pre- and postcondition 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 of our programming language with the *dereferencing* operator: $x.f$ denotes the value of field f of the object denoted by the variable x. As a special case, we abbreviate **this**.f by 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 variable x or a *monadic predicate* X. Second-order quantification of such predicates allow for the specification of properties like *reachability*: $\forall Y \Big(\big(Y(e) \wedge \forall x (Y(x) \rightarrow Y(x.next)) \big) \rightarrow Y(e') \Big)$ 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.

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). 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 $\mathcal{M}_{\mathbf{F}}$. 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 the interpretations, we introduce the substitution $p[f := t]$ which caters for *aliasing*: it replaces every expression of the form $x.f$ by the *conditional expression* $\mathbf{if}\ x = \mathbf{this}\ \mathbf{then}\ t\ \mathbf{else}\ x.f\ \mathbf{fi}$.

Axiom 1 (FIELD ASSIGNMENT). $\quad \vdash \{p[f := t]\}\ f := t\ \{p\}$

As a simple example, we have the following instantiation

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

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 [7], 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 1 lists a few 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*. 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 [7], 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.

Definition 1. *We have the following main clauses.*

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

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.

We have the following axiomatization of object creation.

Axiom 2 (OBJECT CREATION)**.**

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

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

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

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

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 1. *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 $\mathcal{M}_{\mathbf{F}}(S)(\sigma, \tau, \phi) = (\sigma', \tau', \phi')$. 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. 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 $\exists x(p)$.

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

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 following rule for reasoning about invariance.

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 1. 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 [15] and dynamic frames [12]. 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 1.1. 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.

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

Listing 1.1. Stack implementation for pushing items

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 proving that all nodes in the stack that were reachable (starting from the *first* node, by repeatedly following *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

$$[Y(e) \wedge \forall x(Y(x) \rightarrow Y(x.next))] \rightarrow Y(e')$$

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

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

and, consequently, we prove the correctness specification 2. Recall that $\mathbf{this}.\mathit{first}$ 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 $F_L(\mathbf{this}) \wedge \forall x(F_L(x) \rightarrow x = \mathbf{this})$. 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

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

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

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

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

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

As above, Y^c denotes the complement of Y: $Y^c(\mathit{first})$ if and only if $\neg Y(\mathit{first})$. Let Y be such that $Y(z) \wedge \forall x(Y(x) \rightarrow Y(x.\mathit{next}))$. We show that $Y(y)$. Let $Y' = Y \setminus \{\mathit{first}\}$. Since $\mathit{first} \neq z \wedge \forall x(x.\mathit{next} \neq \mathit{first})$, it follows that $Y'(z) \wedge \forall x(Y'(x) \rightarrow Y'(x.\mathit{next}))$. 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

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

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

Thus, we obtain

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

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 have to give a proof of the local correctness specification (2), and let p denote the precondition

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

and q denote the postcondition

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

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

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

Next we observe that

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

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 have 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:

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

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 [12] as it is implemented in the KeY tool [16]. The KeY proof of our case study can be found in the artifact [4] 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 [3] shows the steps for reproducing the proof of invariance of reachability over the push method using KeY. This section describes on 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 with an assignable clause. For example, the clause **assignable** {**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 **this**.$\mathit{push}(v)$) differ at most with respect 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 $init(h)$ as $\forall x(x \in dom(h) \wedge \bigwedge_f x.f = h(x.f))$ which expresses that heap variable h stores the current heap, that is, $\sigma, \tau, \phi, \omega \models init(h)$ 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

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

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

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

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 method with an assignable clause {**this**} and let *heap* be the heap before the method executes. In KeY's logic, the heap resulting after the method call is expressed as $heap[anon(self.*, anon\_heap)]$. When evaluating a field access $x.f$ in the heap $heap[anon(self.*, anon\_heap)]$, the evaluation rules for heaps in KeY require to perform what is essentially an aliasing analysis, e.g. checking whether $x =$ **this**. As a program (fragment) is symbolically executed in KeY, updates to the heap are accumulated, resulting in new, larger heap terms. For real-world programs, the heap terms tend to get large (in our case study, see e.g. [3, 0:41]). Consequently, it may require many proof steps to reason about the value of fields in such heaps.

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 [14]. It contains a basic programming language with the core rules of separation logic that are shared by many of the separation logic 'versions' that have extended language features, e.g. separation logic suitable for Java [9], and tools for verifying (Java) programs using separation logic such as VeriFast [11] and VerCors [5]. For example, VerCors can be used to prove the correctness of a (concurrent wait-free) push explicitly using permissions[4]. However, for the purposes of this case study we keep the discussion on a more abstract level to focus on the basic ideas underlying separation logic which allows for a better comparison with our footprints approach.

Instead of object orientation, Reynolds' uses a fairly low-level programming language with allocation, de-allocation and several pointer operations, such as reading or writing 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 corresponds to accessing a location referenced by a pointer.

Before we perform 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 1.2. 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

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

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. 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

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

Furthermore, we have the following local specification in separation logic

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

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

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:
$\{\mathit{first} = z \wedge \mathbf{emp}\}\ \mathit{first} := cons(v, \mathit{first})\ \{\mathit{first} \mapsto (v, z)\}$.

6 Discussion

All 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 [15] 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 [6]).

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 [12] 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.

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, 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.

Using results from the paper [8], 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.

References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.
2. Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009.
3. Jinting Bian and Hans-Dieter A. Hiep. *Reasoning About Invariant Properties of Object-oriented Programs - dynamic frames*. FigShare, 2021. Available at: https://doi.org/10.6084/m9.figshare.16782667.v1. doi:10.6084/m9.figshare.16782667.v1.
4. Jinting Bian and Hans-Dieter A. Hiep. *Reasoning About Invariant Properties of Object-oriented Programs-dynamic frames: Proof files*. Zenodo, 2021. Available at: https://doi.org/10.5281/zenodo.6044345. doi:10.5281/zenodo.6044345.
5. Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve A. Schneider, editors, *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017. doi:10.1007/978-3-319-66845-1_7.
6. Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2001.
7. Frank S. de Boer. 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*, pages 135–149, 1999.
8. Stijn de Gouw, Frank S. de Boer, Wolfgang Ahrendt, and Richard Bubel. Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic. *Software and Systems Modeling*, 15(4):1117–1140, 2016.
9. Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In Gail E. Harris, editor, *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*, pages 213–226. ACM, 2008. doi:10.1145/1449764.1449782.
10. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. 1971.
11. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

12. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006. doi:10.1007/11813040_19.
13. Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaption. *Theor. Comput. Sci.*, 24:337–347, 1983.
14. John C. Reynolds. 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*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.
15. John C. Reynolds. 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*, pages 460–469, 2005.
16. Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.