Enforcing Structural Invariants using Dynamic Frames

Diego Garbervetsky, Daniel Gorín, and Ariel Neisen

Departamento de Computación, FCEyN, Universidad de Buenos Aires
{diegog,dgorin,aneisen}@dc.uba.ar

Abstract. The theory of dynamic frames is a promising approach to handle the so-called *framing problem*, that is, giving a precise characterizations of the locations in the heap that a procedure may modify.
In this paper, we show that the machinery used for dynamic frames may be exploited even further. In particular, we use it to check that implementations of abstract data types maintain certain structural invariants that are very hard to express with usual means, including being acyclic (like non-circular linked lists and trees) and having a unique path between nodes (like in a tree).
The idea is that regions in this formalism over-approximate the set of reachable objects. We can then maintain this structural invariants by including special preconditions in assignments, of the kind that can be verified by state-of-the-art SMT-based tools. To test this approach we modified the verifier for the Dafny programming language in a suitable way and were able to enforce these invariants in non-trivial examples.

1 Introduction

A typical procedure specification describes the effects of the procedure over its arguments. However, in a context involving pointers, this information is not enough to enable the verification (neither manual nor automatic) of conformance of an implementation to its specification –at least not in a modular way [9].

What is missing in typical specifications is a precise characterization of the locations in the program heap that can be safely assumed to be left untouched by an operation. The problem of formally describing such locations is known as the *frame problem*. The theory of *dynamic frames* [5] is, perhaps, one of the most promising proposals on how to address this problem in a practical way. Recent implementations have shown that verification of programs containing dynamic frame specifications is feasible using state-of-the-art tools [7,4,8,14].

To use dynamic frames basically means to equip each value o in the heap with a *specification variable* or *ghost field* (i.e., a field that can be used in the program specification but not in the program text) that represents the collection of heap locations that must be affected in order to make the observable value of o change.[1] This attribute is usually called *representation region*. Intuitively, if o_1

[1] More precisely, a value can be equipped with more than one such attributes, which allows for more precise specifications, but for the general case, one is enough.

and o_2 have disjoint representation regions, one can guarantee that modifying o_1 will not inadvertently alter o_2 too.

Now, this article is *not* about framing the scope of the effects of a procedure call. Instead, our starting point is the simple observation that, in practice, the representation region of o roughly corresponds to the set of locations that are *reachable* from o by chasing pointers. This means that in a setting with dynamic frames the developer is required to provide, for the sake of framing, information about the heap that might be extremely useful for the verification of heap-related properties such as reachability, shape-analysis, etc. We are therefore interested in the question of how this information can be effectively exploited for such tasks.

In this paper we give a first step in that direction: we use the machinery of dynamic frames to verify that (the graph induced by the points-to relation of) the internal representation of an *abstract data type* satisfies certain structural properties, such as *acyclicity* or *tree-shapedness*. This kind of structural properties constitute the invariant of countless data-structures and at the same time they tend to be tricky to maintain properly, leading to very subtle bugs. In addition, these are properties that cannot even be expressed using a formula of plain first-order logic and, therefore, are not very amenable to the automated techniques employed in contract-based verification.

The paper is structured as follows. In §2 we briefly introduce the dynamic frames methodology and define a simple language with region inference that serves as a basis for the developments in later sections. In §3, for instance, we show how to extend it with a class qualifier for acyclity and in §4 with another one for tree-shapedness. In §5 we focus on improving the precision of the techniques and in §6 we present some preliminary results using a prototype implementation. We conclude in §7 and §8 discussing related work and future directions.

2 Dynamic frames

We begin by fixing notations and terminology. We assume an infinite set *Ref* of *references* (or *locations*) and a set *Val* of *values*, which we assume *indexed tuples* of references (we shall call them *records* or *objects*). We also assume an infinite set *Var* of *program variables*. As usual, all these sets are mutually disjoint.

A *store* (or *heap*) is a *finite* and *injective* mapping σ_h from *Ref* to *Val*,[2] while an *environment* is a *finite* mapping σ_e from *Var* to *Ref*. A *state* is then a pair $\sigma = \langle \sigma_h, \sigma_e \rangle$ and we will typically write both $\sigma(\iota)$ and $\sigma(v)$ for $\iota \in \mathit{Ref}$ and $v \in \mathit{Var}$ meaning $\sigma_h(\iota)$ and $\sigma_h(\sigma_e(v))$ respectively. That is, $\sigma(x)$ denotes the value of x, where x can be either a program variable or a reference. In the same vein, for f a field of an indexed tuple, we will write $\sigma(\iota.f)$ and $\sigma(v.f)$ for $\sigma(\sigma(\iota).f)$ and $\sigma(\sigma(v).f)$. We refer to the domain of σ_h as the references *used in* σ. Finally, we say that an object o_i *reaches* an object o_j in a state σ if either $i = j$ or for some field f, $\sigma(o_i.f)$ reaches o_j in σ (if the state σ is clear from context, we say simply "o_i reaches o_j").

[2] In very concrete terms, σ_h maps *memory locations* to *objects* and the injectiveness condition witnessess the fact that different locations refer to different objects.

The idea behind the dynamic frames theory is to use finite sets of references, called *regions*, to *frame* a value. Intuitively, a region ρ frames a reference ι in state σ if in any other state τ that coincides with σ in the value of the objects denoted in ρ, $\sigma(\iota)$ and $\tau(\iota)$ must coincide too. It is easy to see that if ρ frames ι then it must be the case that $\iota \in \rho$.[3]

Now, if we know that ρ frames ι, and we also have that region π is the set of references touched by a procedure invocation and we can show that ρ and π are disjoint (denoted $\rho \parallel \pi$), then we can assert that the invocation did not change the value of ι (this is the Value Preservation Theorem in [5]).

The dynamic frames methodology can therefore be summed up as the requirement to assign to each object in the store a region that frames it, called its *representation region*. Representation regions can then appear in procedure specifications, e.g., a precondition may require that two objects have disjoint representation regions –which means they are not aliased; a postcondition may indicate that an object's representation region grew only by the addition of fresh references (cf. *swinging pivots* [9]), which preserves disjointness of regions, etc.

2.1 Inferring regions automatically

In order to illustrate dynamic frames with an example and provide a basis for the developments in the following sections, we introduce next a very small programming language. Its syntax, shown in Fig. 1, is loosely based on Dafny's [7]. We assume the language to be statically typed, but we will not give the formal typing rules since this is standard.

$$
\begin{array}{rcl}
\mathit{Member} & ::= & \mathit{Field} \mid \mathit{Method} \\
\mathit{Field} & ::= & \textbf{var } \mathsf{f} : \mathsf{C} \\
\mathit{Method} & ::= & \textbf{method } \mathsf{m}(x_1 : \mathsf{C}_1, \ldots, x_n : \mathsf{C}_n) \textbf{ returns } (y : \mathsf{C}) \\
 & & \quad \textbf{requires } \alpha(\mathit{this}, x_1, \ldots, x_n, \sigma); \\
 & & \quad \textbf{ensures } \beta(\mathit{this}, x_1, \ldots, x_n, y, \sigma, \tau); \\
 & & \quad \textbf{modifies } \rho(\mathit{this}, x_1, \ldots, x_n, \sigma); \\
 & & \{\ \mathit{Stmt}\ \} \\
\mathit{Stmt} & ::= & \mathsf{v} := \mathit{Expr} \mid \mathsf{v} := \textbf{new } \mathsf{C} \mid \mathsf{v}_t.\mathsf{f} := \mathsf{v}_s \mid \mathsf{v}_t := \mathsf{v}_s.\mathsf{m}(\mathsf{v}_1, \ldots, \mathsf{v}_n) \\
 & \mid & \textbf{if } (\mathit{Expr} == \textbf{null})\{\mathit{Stmt}\} \textbf{ else } \{\mathit{Stmt}\} \mid \mathit{Stmt}; \mathit{Stmt} \\
\mathit{Ref} & ::= & \mathsf{v} \mid \textbf{this} \mid \textbf{null} \\
\mathit{Expr} & ::= & \mathit{Ref} \mid \mathit{Ref}.\mathsf{f}
\end{array}
$$

Fig. 1: Syntax of a language with support for dynamic frame annotations.

Similarly, we leave the language for method contracts unspecified and simply take it to be a form of many-sorted first-order language. In an *ensures* clause, the free variables σ and τ in $\beta(\mathit{this}, x_1, \ldots, x_n, y, \sigma, \tau)$ both have sort *state* and correspond to the states before and after the execution of the method, respectively. In a *requires* clause, only the state variable σ may occur free. That said, we will sometimes write specifications in a sugared form, as in Fig. 2, and leave the unsugaring to the reader. Notice that in sugared *ensure* clauses, variable x corresponds to the term $\tau_e(x)$, while $\mathbf{old}(x)$ denotes $\sigma_e(x)$.

[3] We are excluding the degenerate case where there is only one possible value for ι.

```
method returnCopy(x,y) returns (z)
  modifies x;                          // ⇝ {σ_e(x)}
  requires x ≠ null ∧ y ≠ null;        // ⇝ σ_e(x) ≠ null ∧ σ_e(y) ≠ null
  ensures x.f = old(y).f ∧ z = old(x); // ⇝ τ(x.f) = σ(y.f) ∧ τ(z) = σ(x)
```

Fig. 2: A typical procedure specification and the unsugared version (commented).

The *modifies clause* is an expression of sort *set of references*, where only a state variable σ occurs free. An example of such an expression would be:

$$\{\sigma_e(\mathbf{this})\} \cup reg(\sigma(x.\mathsf{f})) \tag{1}$$

which we will normally write in its sugared form $\{\mathbf{this}\} \cup reg(x.\mathsf{f})$. Notice that *reg* is used to denote the representation region of an object, which every object has. The representation region is an attribute of an object, therefore $o_1 = o_2$ implies $reg(o_1) = reg(o_2)$. It is not a field accessible from the program text, though; in fact two objects may have $reg(o_1) \neq reg(o_2)$ while $o_1.\mathsf{f} = o_2.\mathsf{f}$ for every field f. We use notation $o_1 \approx o_2$ for this form of *structural* equality that takes into account only the value of these fields (of course, $o_1 = o_2$ implies $o_1 \approx o_2$). As we will see next, it is possible to have an execution from σ to τ such that for some reference ι, $\sigma_h(\iota) \approx \tau_h(\iota)$ (i.e., ι is not "touched") while $\sigma_h(\iota) \neq \tau_h(\iota)$ (e.g., $reg(\sigma_h(\iota)) \neq reg(\tau_h(\iota))$ because some reference reachable from ι in σ was modified in τ). For conciseness, we may write $reg_\sigma(x)$ for $reg(\sigma(x))$.

We will impose some sanity conditions on valid states. For instance, to simplify definitions, we want to assume that the null reference $\emptyset$ denotes a special *null object* with an empty representation region. More importantly, the representation region of an object o must include the set of objects reachable from o. We can express these conditions as a *state invariant* $I_s(\sigma)$ using the following:

$$\operatorname{im}\sigma_e \subseteq \operatorname{dm}\sigma_h \wedge \emptyset \in \operatorname{dm}\sigma_h \wedge reg_\sigma(\emptyset) = \emptyset \wedge \forall f \cdot \sigma(\emptyset.f) = \emptyset \tag{2}$$

$$\forall \iota \in \operatorname{dm}\sigma_h \cdot (\iota \neq \emptyset \Rightarrow \iota \in reg_\sigma(\iota) \wedge \forall f \cdot reg_\sigma(\iota.f) \subseteq reg_\sigma(\iota) \subseteq \operatorname{dm}\sigma_h) \tag{3}$$

Fig. 3 presents the interesting cases of the semantics of this language, in an axiomatic form. To minimize boilerplate we will consistently use the constructions $Pre^{\mathbf{P}}(\alpha_1(\sigma), \ldots, \alpha_n(\sigma))$ and $Pst^{\mathbf{P}}_{\mathbf{Q}}(\beta_1(\sigma,\tau), \ldots, \beta_m(\sigma,\tau))$, where the α_i and β_i are the relevant parts of the pre and post-conditions. They correspond, respectively to $\forall\sigma \cdot (\mathbf{P}(\sigma) \Rightarrow (I_s(\sigma) \wedge \alpha_1(\sigma) \wedge \cdots \wedge \alpha_n(\sigma)))$ and $\forall\sigma,\tau \cdot ((\mathbf{P}(\sigma) \wedge \beta_1(\sigma,\tau) \wedge \cdots \wedge \beta_m(\sigma,\tau)) \Rightarrow \mathbf{Q}(\sigma,\tau))$.

Unlike Dafny, the semantics dictate the way in which the representation regions of the objects in the heap are updated. Consider, for example, rule NEW. Firstly, it requires $\mathbf{P}$ to be strong enough to ensure the state invariant holds at the pre-state σ. Next, it requires that $\mathbf{Q}$ must hold whenever $\mathbf{P}$ was satisfied by σ and the post-state τ differs from σ only on the value of variable v (see definition of $\rhd$ below), which corresponds to a *fresh* reference (i.e., not occurring in σ) and refers to an object that *is the only member of its representation region.*

The STORE rule indicates that an assignment may *modify* the representation region of the target v_t: whatever might be reachable from v_s (i.e., its represen-

tation region) is added to what may be reachable from v_t. But this is not the whole story: in the statement $x.f := y;\ y.f := z$, after the second assignment, x's representation region needs to be adjusted too. As we will see next, the state modification operator $\rhd$ takes care of this also.

Let ν be a set of variables and let ρ be a set of locations; the predicate $\sigma \rhd^{\nu}_{\rho} \tau$ is then the conjunction of the following formulas:

$$I_s(\tau) \wedge \mathrm{dm}\,\sigma_e \subseteq \mathrm{dm}\,\tau_e \wedge \mathrm{dm}\,\sigma_h \subseteq \mathrm{dm}\,\tau_h \tag{4}$$

$$\forall v \in (\mathrm{dm}\,\sigma_e \setminus \nu) \cdot \sigma_e(v) = \tau_e(v) \wedge \forall \iota \in (\mathrm{dm}\,\sigma_h \setminus \rho) \cdot \sigma(\iota) \approx \tau(\iota) \tag{5}$$

$$\forall \iota \in \mathrm{dm}\,\sigma_h \cdot reg_\tau(\iota) = reg_\sigma(\iota) \cup regs_\tau(\rho \cap reg_\sigma(\iota)) \tag{6}$$

where $regs_\sigma(\rho) = \bigcup_{\kappa \in \rho} reg_\sigma(\kappa)$. Clearly (4) is just a basic state sanity condition; (5) indicates that ν and ρ characterize the variables and locations that may have changed; while (6) propagates updates in representation regions. Notice that for $\rho \parallel reg_\sigma(\iota)$, (5) and (6) guarantee that $\sigma(\iota) = \tau(\iota)$. We write $\rhd^{\nu}$ and $\rhd_{\rho}$ for $\rhd^{\nu}_{\emptyset}$ and $\rhd^{\emptyset}_{\rho}$ respectively.

$$\textsc{new}\quad \frac{\models Pre^{\mathbf{P}}() \qquad \models Pst^{\mathbf{P}}_{\mathbf{Q}}(\sigma \rhd^{\{\mathsf{v}\}} \tau, \tau_e(\mathsf{v}) \in (\mathrm{dm}\,\tau_h \setminus \mathrm{dm}\,\sigma_h), reg_\tau(\mathsf{v}) = \{\tau_e(\mathsf{v})\})}{\{\mathbf{P}\}\ \mathsf{v} := \mathbf{new}\ \mathsf{C}\ \{\mathbf{Q}\}}$$

$$\textsc{read}\quad \frac{\models Pre^{\mathbf{P}}(\sigma(\mathsf{v}_s) \neq \sigma(\emptyset)) \qquad \models Pst^{\mathbf{P}}_{\mathbf{Q}}(\sigma \rhd^{\{\mathsf{v}_t\}} \tau, \tau(\mathsf{v}_t) = \sigma(\mathsf{v}_s.\mathsf{f}))}{\{\mathbf{P}\}\ \mathsf{v}_t := \mathsf{v}_s.\mathsf{f}\ \{\mathbf{Q}\}}$$

$$\textsc{store}\quad \frac{\models Pre^{\mathbf{P}}(\sigma(\mathsf{v}_t) \neq \sigma(\emptyset)) \qquad \models Pst^{\mathbf{P}}_{\mathbf{Q}}(\sigma \rhd_{\{\sigma_e(\mathsf{v}_t)\}} \tau, \tau(\mathsf{v}_t) \approx \sigma(\mathsf{v}_t)[\mathsf{f} \mapsto \sigma(\mathsf{v}_s)], reg_\tau(\mathsf{v}_t) = reg_\sigma(\mathsf{v}_t) \cup reg_\sigma(\mathsf{v}_s))}{\{\mathbf{P}\}\ \mathsf{v}_t.\mathsf{f} := \mathsf{v}_s\ \{\mathbf{Q}\}}$$

$$\textsc{call}\quad \frac{\models Pre^{\mathbf{P}}(\sigma(\mathsf{v}_s) \neq \sigma(\emptyset), \alpha(\mathsf{v}_s, \mathsf{v}_1, \ldots, \mathsf{v}_n, \sigma)) \qquad \models Pst^{\mathbf{P}}_{\mathbf{Q}}(\sigma \rhd^{\{\mathsf{v}_t\}}_{\rho(\mathsf{v}_s,\mathsf{v}_1,\ldots,\mathsf{v}_n,\sigma)} \tau, \beta(\mathsf{v}_s, \mathsf{v}_0, \ldots, \mathsf{v}_n, \mathsf{v}_t, \sigma, \tau))}{\{\mathbf{P}\}\ \mathsf{v}_t := \mathsf{v}_s.\mathsf{m}(\mathsf{v}_1, ..., \mathsf{v}_n)\ \{\mathbf{Q}\}}$$

Fig. 3: Semantic rules for the language of Fig. 1 (fragment).

In rule CALL, α, β and ρ correspond to the *requires*, *ensures* and *modifies* clauses of method m, respectively. Therefore, the scope of the effects of the method call is *framed* by $\rho(\mathsf{v}_s, \mathsf{v}_1, \ldots, \mathsf{v}_n, \sigma)$, since the post-state τ must satisfy $\sigma \rhd^{\{\mathsf{v}_t\}}_{\rho(\mathsf{v}_s,\mathsf{v}_1,\ldots,\mathsf{v}_n,\sigma)} \tau$.

Given these semantic clauses, one can derive program acceptance rules in a straightforward way: a class C is accepted if all its methods are accepted; and a method declaration of the form

method $m(x_1 : \mathsf{C}_1, \ldots, x_n : \mathsf{C}_n)$ **returns** $(z\colon \mathsf{D})$
  requires $\alpha(this, x_1, \ldots, x_n, \sigma)$;
  ensures $\beta(this, x_1, \ldots, x_n, z, \sigma, \tau)$;
  modifies $\rho(this, x_1, \ldots, x_n, \sigma)$;
  { S }

is accepted if, for $\{v_1, \ldots, v_k\}$ the local variables in S, we have:

$$\{\alpha(this, x_1, \ldots, x_n, \sigma)\}S\{\sigma \rhd^{\{v_1,\ldots,v_k\}}_{\rho(this,x_1,\ldots,x_n,\sigma)} \wedge \beta(this, x_1, \ldots, x_n, z, \sigma, \tau)\} \quad (7)$$

Of course, in order to decide acceptance one needs to resort to some sort of automated reasoner, like is done with the Dafny verifying-compiler. Verification this way can be seen as a form of typing.

It is not hard to prove that programs that are thus accepted behave well with respect to the frame conditions of the **modifies** clauses. For this, one needs to prove a stronger result, namely, that representation regions over-approximate *reachability* (i.e., if o_1 reaches o_2 in σ then $o_2 \in reg(o_1)$), which follows from the fact that this condition is preserved according to the semantics. Formal definitions and proofs can be found in [12]

As a final remark, notice that according to our rules (including condition (6)), representation regions are *monotonic* in the sense that no reference is ever removed –even those that may be no longer reachable are kept. This is suboptimal: one may easily end up in a scenario with two objects o_1 and o_2 such that $reg(o_1) \not\parallel reg(o_2)$ although no location reachable from one is reachable by the other. In Kassios's original formulation [5] this was not the case; but it required higher-order logic and inductive reasoning, which is just too hard for state-of-the-art automated reasoners. Our presentation can be seen as a compromise between precision and automatic verifiability.

Example 1. Consider the declaration of a class *List* in Fig. 4. We are interested only in the framing specification and will ignore the functional part.

```
class List {
  method add(d : Data)
    modifies reg(this);
    ensures
     (reg(old(this)) ∪ reg(d)) ⊆ reg(this);
    ensures
     fresh(reg(this)\(reg(old(this)) ∪ reg(d));
    ensures ...

  method concat(l : List)
    requires reg(this)∥reg(l);
    modifies reg(this);
    ensures
     reg(this) = reg(old(this)) ∪ reg(l);
    ensures ...
}
```

Fig. 4: A simple *List* type interface

Method *add* modifies the list to append a new element and its **modifies** clause states that the set of references reachable from **this** can be affected by this method. Therefore, we need to specify the effect only for those locations. The specification says that after executing *add* every reachable object remains reachable and, in addition, d will be reachable too. It also says that it will not introduce aliasing with other existing objects by declaring that any other object reachable from **this** will be fresh[4].

Similarly, method *concat* declares that after its execution **this** will also reach the objects reachable from list l. The combination of the **requires** and **modifies** clauses also guarantees that l will not be mutated.

[4] A fresh object will most probably correspond to a newly allocated node that will hold the data; but this is an implementation detail and nothing else need to be said about it in the interface.

3 Verifiably acyclic data structures

In this section we will show how the language of §2.1 can be easily adapted to support verifiably correct *acyclic data structures*. These are ubiquitous in computer science, typically implemented using a node type with a recursive reference. Fig. 5 shows two structures that can be built using this type of nodes. The one on the right, though, does not correspond to what one expects from a linked list since it contains a cycle. An incorrect implementation of a linked list that allows such instances to be built may lead to bugs that are very hard to track-down.

Fig. 5: Structures l_1 and l_2 are built with linked list nodes; l_2 is not *acyclic*.

The fact that no node in a linked list should participate in a cycle can be seen as part of the class invariant of the list type. One would be tempted to include this requirement as part of the class invariant of the type and use standard techniques to verify that it holds at the end of every procedure call [1,3]. Regrettably, no first-order logic formula can express this condition (this is a straightforward consequence of the compactness theorem, see, e.g., [12] for more details), which makes this approach currently unfeasible.

The idea we will explore here is to treat this requirement as a *strong* form of class invariant, which must hold at every point of the method execution. We will see that exploiting representation regions makes it feasible to guarantee that this condition is preserved.

3.1 A characterization of *acyclicity*

Suppose we implement the class *List* of Fig. 4 using a linked list and want to enforce its acyclicity. We propose to extend the syntax of the language of §2.1 with a special *class qualifier* "**acyclic**" that allows us to write declarations as the one in Fig. 6. The intended meaning is that any object of a class qualified as **acyclic** satisfies a strong class invariant. The exact invariant deserves some considerations, though. We shall say that o_i *occurs in a cycle* in σ whenever for

```
acyclic class List{
    var first : LLNode;
    ...
}
acyclic class LLNode {
    var next : LLNode
    var payload : T
}
```

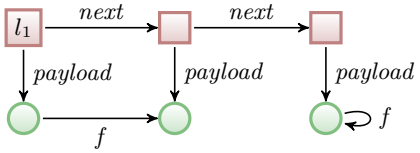

Fig. 6: Declaration of an *acyclic* linked list node.

some f, $\sigma(o_i.\mathsf{f})$ reaches o_i in σ. The most intuitive definition would be, perhaps,

to stipulate that an object o of a class tagged as **acyclic** satisfies the invariant "o does not reach an object that occurs in a cycle". This would be too strong in practice. Consider for example an object n of class *LLNode* (Fig. 6); to fulfill this invariant we might as well demand T to be qualified with **acyclic** too. But this would impose a big restriction on the type of the payload. In other words, a linked list node should not put demands on the internal representation of the payload.

The invariant we will use, instead, is that if o is of a class qualified as **acyclic** then "o cannot occur in a cycle made of objects of *acyclic classes*". That is, o may occur in a cycle as long as some object in the cycle is not tagged as acyclic. To avoid confusion, we will term the cycles that this invariant forbids "invalid cycles". We believe this weaker notion of acyclicity constitutes a good compromise in practice. For instance, observe that if one is given a while-loop where every inductive variable is a reference to an object qualified as **acyclic** that is not mutated during the cycle (as it would typically be the case in algorithms traversing the internal representation of an abstract datatype implemented using an acyclic structure), then if every iteration of the loop can be shown to terminate, the loop cannot hang.

3.2 Preserving the acyclicity invariant

We want to guarantee that an acyclic object remains acyclic after the execution of any statement (that is, assuming the pre-condition of the statement holds). What we will exploit is the fact that $reg(o)$ over-approximates the set of objects that o may reach.

The first thing to observe is that only assignments and method invocations can introduce cycles; and among assignments, only *store* instructions $\mathsf{v}_t.\mathsf{f} = \mathsf{v}_s$ do. Notice, furthermore that if v_t or v_s is of a non-acyclic type, then no invalid cycle can be formed (recall that an invalid cycle involves acyclic objects).

Fig. 7 shows an example of a store instruction that introduces a cycle. The important thing to observe is that this can happen if and only if o_2 reaches o_1. This motivates rule STORE^a shown in Fig. 8, which replaces rule STORE when both the target and source of the store are references to acyclic classes. The only difference with rule STORE is the additional pre-condition $\sigma_e(\mathsf{v}_t) \notin reg_\sigma(\mathsf{v}_s)$.

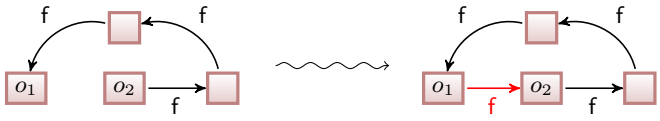

Fig. 7: Execution of $o_1.\mathsf{f} = o.{}_2$ introduces an invalid cycle.

Interestingly, the introduction of the STORE^a rule is the only modification we need to do to the semantics. To see this, consider a method invocation $\mathsf{v}_t = \mathsf{v}_s.\mathsf{m}(\mathsf{v}_1, \ldots, \mathsf{v}_n)$. According to rule CALL, it must be the case that the pre-condition of method m, $\alpha(\mathsf{v}_s, \mathsf{v}_1, \ldots, \mathsf{v}_n, \sigma)$, holds before the invocation. But this precondition must have been strong enough to verify that the body of method m introduces no invalid cycles (cf. (7) on p.6).

$$\textsc{store}^a\ \frac{\begin{array}{l}\models Pre^{\mathbf{P}}(\sigma(\mathsf{v}_t) \neq \sigma(\emptyset), \sigma_e(\mathsf{v}_t) \notin reg_\sigma(\mathsf{v}_s)) \\ \models Pst^{\mathbf{P}}_{\mathbf{Q}}(\sigma \rhd_{\{\sigma_e(\mathsf{v}_t)\}} \tau, \tau(\mathsf{v}_t) \approx \sigma(\mathsf{v}_t)[\mathsf{f} \mapsto \sigma(\mathsf{v}_s)], reg_\tau(\mathsf{v}_t) = reg_\sigma(\mathsf{v}_t) \cup reg_\sigma(\mathsf{v}_s))\end{array}}{\{\mathbf{P}\}\ \mathsf{v}_t.\mathsf{f} := \mathsf{v}_s\ \{\mathbf{Q}\}}$$

Fig. 8: $\textsc{store}^a$ rule applies whenever v_t and v_s are references of acyclic classes.

4 Trees and similar data structures

Acyclicity is not the only common invariant that is impossible to express using classical logic. In this section we will discuss that of *being a tree*, which requires not only having no cycles but also having a *unique path* to every reachable node. We will show that by incorporating a second region to objects, we can handle an additional class qualifier "**tree**", whose precise meaning we will give below.

Before going into the details, it is worth observing that the dynamic frames methodology does not preclude the inclusion of more than one region per object. We will be exploiting this possibility also on the following sections.

For succinctness, we will say that an object o *is a tree* if it is an instance of a class tagged as **tree**. We will also assume that the **tree** qualifier implies the **acyclic** qualifier.

Just like in the previous section, we will consider a notion of "being a tree" that constitutes a compromise between what can be expressed and what can be enforced. Therefore, we want the following invariant for a *tree* o: i) o satisfies the invariant for acyclic objects (see §3), ii) if o reaches an object o' that is a tree, then there is only one path between o and o' where every object in the path is also a tree.

Again, in order to see how this invariant can be preserved, we need to look only at store instructions $\mathsf{v}_t.\mathsf{f} := \mathsf{v}_s$, where both v_t and v_s are trees. One can then see that there are essentially two ways in which the invariant can get broken. The first one, illustrated in Fig. 9 corresponds to the case when both v_t and v_s can reach a common (tree) object o: there is already a unique path from each to o, but executing the store would introduce an additional path from the target to o. This can be avoided by adding the following additional precondition:

$$reg_\sigma(\mathsf{v}_t) \parallel reg_\sigma(\mathsf{v}_s) \tag{8}$$

This clause implies the condition $\sigma_e(\mathsf{v}_t) \notin reg_\sigma(\mathsf{v}_s)$ required to enforce acyclicity.

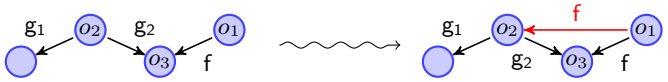

Fig. 9: Execution of $o_1.\mathsf{f} = o_{.2}$ introduces an extra path from o_1 to o_3.

It is not hard to verify that if v_t and v_s satisfy (8) then the tree invariant must hold on every node reachable from either v_t or v_s. But what can we say about the invariant at tree nodes that reach either v_t or v_s? This question leads us to

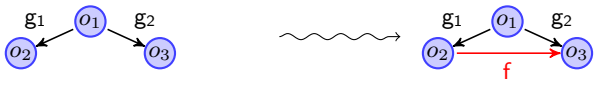

Fig. 10: Execution of $o_2.\mathsf{f} = o.3$ introduces an extra path from o_1 to o_3.

the second case, illustrated in Fig. 10. If v_t and v_s share a common ancestor, then the store will necessarily break the invariant of this ancestor.

Of course, it is not possible to express this condition using a first-order formula and the *reg* predicate. But using analogous ideas and techniques, one can associate to each object o an additional region $ger(o)$ that over-approximates the set of references that reach o and specify its evolution in semantic rules and method contracts. Using this, we can express the missing pre-condition for the store (when v_t and v_s are trees) as:

$$ger_\sigma(\mathsf{v}_t) \parallel ger_\sigma(\mathsf{v}_s) \tag{9}$$

To make everything fit well, we add to invariant $I_s(\sigma)$ (cf. p.4) the requirements:

$$ger_\sigma(\emptyset) = \emptyset \tag{10}$$

$$\forall \iota \in \mathrm{dm}\,\sigma_h \cdot \iota \neq \emptyset \Rightarrow \iota \in ger_\sigma(\iota) \wedge \forall f \cdot ger_\sigma(\iota) \subseteq ger_\sigma(\iota.f) \subseteq \mathrm{dm}\,\sigma_h \tag{11}$$

Similarly, the $\rhd^{\nu}_{\rho}$ predicate must be extended with the following clause:

$$\forall \iota \in \mathrm{dm}\,\sigma_h \cdot ger_\tau(\iota) = ger_\sigma(\iota) \cup gers_\tau(\rho \cap ger_\sigma(\iota)) \tag{12}$$

where $gers_\sigma(\rho) = \bigcup_{\kappa \in \rho} ger_\sigma(\kappa)$. Fig. 11 finally shows the formal semantics of the store rule for trees. It states that $ger(\mathsf{v}_t)$ remains unchanged but, of course, $ger(\mathsf{v}_s)$ is expanded. The latter implies that v_s is modified by the operation and therefore it must be included in the argument of the $\rhd$ predicate and the fact that every other field remains unchanged must be explicitly stated.

The rules in Fig. 3 and 8 need to be modified to accommodate predicate *ger* but this is straightforward so we leave the details for the reader. It is not difficult to see that, in the resulting system, $ger(o)$ represents the set of objects that reach o and that every object of a class tagged as **tree** verifies its invariant.

$$\textsc{store}^t\ \frac{\begin{array}{l} \models Pre^{\mathbf{P}}(\sigma(\mathsf{v}_t) \neq \sigma(\emptyset), reg_\sigma(\mathsf{v}_t) \parallel reg_\sigma(\mathsf{v}_s), ger_\sigma(\mathsf{v}_t) \parallel ger_\sigma(\mathsf{v}_s)) \\ \models Pst^{\mathbf{P}}_{\mathbf{Q}} \left(\begin{array}{l} \sigma \rhd_{\{\sigma_e(\mathsf{v}_t), \sigma_e(\mathsf{v}_s)\}} \tau, \tau(\mathsf{v}_s) \approx \sigma(\mathsf{v}_s), \tau(\mathsf{v}_t) \approx \sigma(\mathsf{v}_t)[\mathsf{f} \mapsto \sigma(\mathsf{v}_s)], \\ reg_\tau(\mathsf{v}_t) = reg_\sigma(\mathsf{v}_t) \cup reg_\sigma(\mathsf{v}_s), ger_\tau(\mathsf{v}_t) = ger_\sigma(\mathsf{v}_t) \\ ger_\tau(\mathsf{v}_s) = ger_\sigma(\mathsf{v}_s) \cup ger_\sigma(\mathsf{v}_t), reg_\tau(\mathsf{v}_s) = reg_\sigma(\mathsf{v}_s) \end{array} \right) \end{array}}{\{\mathbf{P}\}\ \mathsf{v}_t.\mathsf{f} := \mathsf{v}_s\ \{\mathbf{Q}\}}$$

Fig. 11: $\textsc{store}^t$ rule applies whenever v_t and v_s are references to trees.

5 Improving precision

We have shown thus far that representation regions, used in principle for framing, can be also used to enforce complex structural invariants (e.g., acyclicity, etc.). In

this section we will see examples of code that would be rejected by our proposed rules, although the invariants are clearly preserved.

Let us start considering the example in Fig. 12, where *LLNode* is the acyclic class defined in Fig. 6 and T is not tagged as acyclic. Call σ the state before $b.next{:=}a$; then $reg_\sigma(a) = \{\sigma_e(a), \sigma_e(c), \sigma_e(b)\}$ which means that the pre-condition of the STORE^a rule does not hold. That is, it is detected that executing this instruction may lead to an invalid cycle involving a and b and the code is therefore rejected. But as Fig. 12 shows graphically, this would be indeed a valid cycle, since it passes through c that is not an instance of an acyclic class.

```
method rejected1() returns a {
  a := new LLNode;
  b := new LLNode;
  c := new T;
  a.payload := c;
  c.f := b;
  b.next := a;
}
```

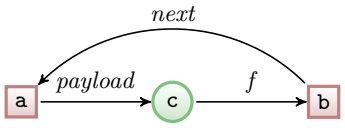

Fig. 12: Rejected code snippet and the shape of a after $b.next := a$.

The technique is imprecise in this case, and the imprecision comes from the fact that $reg(o)$ contains the location of every reachable object, and not just of those that are reachable using only acyclic objects.

In §4 we already explored the possibility of incorporating additional regions in the context of handling the **tree** qualifier; we can take a similar approach here to improve the precision of the methodology. We propose, therefore, adding to each o a region $reg^a(o)$ that contains every acyclic object reachable from o passing only through acyclic objects. Again, this requires adding additional clauses to the state invariant $I_s(\sigma)$ and the $\rhd$ predicate and extending the semantic rules. This changes are straightforward and were already mentioned in more detail in §4, so we will skip the details. The important part is replacing reg by reg^a in the pre-condition of rule STORE^a.

Of course, the same approach can be used to improve the precision on code handling trees: we can add regions reg^t and ger^t that only hold references to tree objects based on reachability via paths that contain only trees.

Let us assume this was indeed done and consider now the scenario in Fig. 13, where *BTNode* is a class qualified as **tree**. The code in question "moves to the root" a node in a binary tree and it is not hard to see that the resulting structure would satisfy the required invariant. The problem is that before the method execution we have $r.\ left \in reg^t(r)$ and, therefore, $reg^t(s) \not\parallel reg^t(r)$ holds after executing $s := t.\ left$. Since regions are monotonic (cf. §2), this is also true before executing $s.\ right := t$ and therefore, the requirements of STORE^t are not satisfied and this program is rejected.

More precisely, the problem originates after the execution of instruction $t.\ left$ **:=null**: s is no longer reachable from t although this is not reflected in $reg^t(t)$ nor in $ger^t(s)$. But since s and t are trees, there is only one path from s to any tree reachable from t; hence, if σ and τ are the pre and post-states of this instruction, it is safe to assume $reg^t_\tau(t) = reg^t_\sigma(t) \setminus reg^t_\sigma(s)$ and

```
method rejected2(BTNode r)
  returns s
  requires t.left ≠ null;
  modifies reg(r);
{
  s := t.left;
  t.left := null;
  s.left := t;
}
```

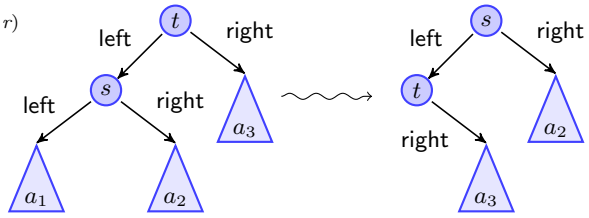

Fig. 13: Rejected code snippet, it is due to the *monotonic* nature of regions.

$ger^t_\tau(s) = ger^t_\sigma(s) \setminus ger^t_\sigma(t)$ (although it is not necessarily the case, for instance, that $reg_\tau(t) = reg_\sigma(t) \setminus reg_\sigma(s)$). Hence, we can improve rule STORE^t even further, by including in $Pst^{\mathbf{P}}_{\mathbf{Q}}$ the following clauses:

$$reg^t_\tau(\mathsf{v}_t) = (reg^t_\sigma(\mathsf{v}_t) \setminus reg^t_\sigma(\mathsf{v}_t.\mathsf{f})) \cup reg^t_\sigma(\mathsf{v}_s) \tag{13}$$

$$ger^t_\tau(\sigma_e(\mathsf{v}_t.\mathsf{f}))) = ger^t_\sigma(\mathsf{v}_t.\mathsf{f}) \setminus ger^t_\sigma(\mathsf{v}_t) \tag{14}$$

Of course, we need also add to the definition of $\rhd^\nu_\rho$ clauses:

$$\forall \iota \in \mathrm{dm}\,\sigma_h \cdot reg^t_\tau(\iota) = (reg^t_\sigma(\iota) \setminus regs^t_\sigma(\rho \cap reg^t_\sigma(\iota)) \cup regs^t_\tau(\rho \cap reg^t_\sigma(\iota)) \tag{15}$$

$$\forall \iota \in \mathrm{dm}\,\sigma_h \cdot ger^t_\tau(\iota) = (ger^t_\sigma(\iota) \setminus gers^t_\sigma(\rho \cap ger^t_\sigma(\iota)) \cup gers^t_\tau(\rho \cap ger^t_\sigma(\iota)) \tag{16}$$

where $regs^t_\sigma(\rho) = \bigcup_{\kappa \in \rho} reg^t_\sigma(\kappa)$ and $gers^t_\sigma(\rho) = \bigcup_{\kappa \in \rho} ger^t_\sigma(\kappa)$.

Using these new rules, it is not hard to see that the example in Fig. 13 is not rejected. Now, we must insist that the soundness of these rules depends on the fact that there is at most one path (passing only through tree nodes) that connects any two tree nodes. To illustrate this point, assume a class *ANode* is tagged as **acyclic** and consider Fig. 14. If after instruction *a.f1* :=**null**, *b* is removed from $reg^a(a)$, then the precondition of the last instruction will hold, which would permit the introduction of a cycle.

In a way, the problem in this case is that the set of reachable regions does not give us enough information to decide if we no longer reach an object. It might be interesting to consider, as future work, the possibility of turning $reg^a(o)$ into a *multiset* of locations, i.e., a total function from *Ref* to $\mathbb{N}$. The intended semantics for $reg^a(o)$ would be that it counts the total *number of paths* going only through acyclic nodes between o and any location ι and the tricky part would be to actually maintain this invariant.

```
method invalid() {
  a := new ANode;
  b := new ANode;
  a.f1 := b;
  b.f2 := b;
  a.f1 := null;
  b.f1 := a;
}
```

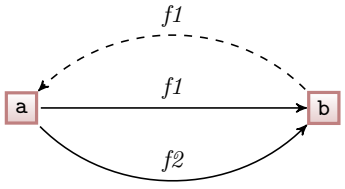

Fig. 14: If *ANode* is tagged **acyclic** this method should be rejected.

6 Evaluation

We developed a prototype implementation[5] based on Dafny's language tool chain. Dafny is an experimental language that explores the use of dynamic frames in object-based sequential programs by enabling the use of ghost fields in contracts and statements [7]. We extended the language with automatic region inference and support for acyclic class qualification. Details can be found in [12].

We then implemented various basic abstract data types using acyclic structures. Stack, Sequence and Queue use a linked list (the last two keep references to both head and tail), Dictionary uses a non-balanced binary search tree (BST). We evaluated the cost of enforcing acyclicity in terms of the number of atoms in region-related annotations, both in specifications (S.Ann) and inside method bodies such as loop invariants (B.Ann), and the verification time (in seconds). The following table summarizes the results obtained. [6]

Module	LOC	#classes	#methods	S.Ann	B.Ann	Verif. Time
Sequence	136	2	4	6	2	2.7s
Stack	48	2	4	5	0	2.3s
Queue	75	2	4	7	1	2.4s
Dictionary	140	2	5	11	4	4.0s

For most cases we were able to automatically check that a method preserves acyclicity. We could not do it, for instance, in the *remove* method of Dictionary because of monotonicity of the regions. A variation of the standard deletion algorithm for BSTs in which values are swapped instead of nodes should be more amenable to verification. It is worth observing that we did not use manual update of ghost fields in method bodies, only loop invariants were provided.

As a second experiment, we tried to assess if the inferred regions together with the **acyclic** qualifier could be used to significantly reduce the overall number of annotations on *idiomatic* Dafny programs. For this, we took some examples from the Dafny distribution (a linked list, an unbounded stack, a queue, and BST based dictionary) and tried to simplify them by tagging the relevant classes as **acyclic**, removing the ghost field used for the representation region (together with their manual updates) and pruning the contracts and invariants accordingly.

We were able to remove about 25%-30% of the annotations in contracts (plus the manual updates of ghost fields that were also removed). The only method we failed to verify was list reversal, which relies on temporarily breaking acyclicity.

It is worth noticing that our implementation cannot currently handle some constructs available in the latest version of the Dafny language; in particular, universal quantification restricted to objects of a given type. Once this is corrected, we might be able to reduce even further the number of annotations in some examples (most notably, in linked lists, which require a ghost field which represents the *spine* of the list, that is, the set of nodes that form the list).

[5] Tool and experiments available at `http://lafhis.dc.uba.ar/dynframes`.

[6] Times measured on a 3GHz Intel® Core™2 Duo based desktop with 4GB of RAM, running Microsoft Windows Vista (32 bits), under regular load.

7 Related work

Dafny is a programming language with a verifying compiler and support for dynamic frames. It is possible to verify the correctness of Dafny programs (that is, without our extensions) that contain class invariants that entail acyclicity or tree-shapedness. It is therefore important to discuss the differences with our approach.

In Dafny, regions are explicitly declared as ghost fields of a class and the programmer is responsible for maintaining them with explicit update instructions. These are, arguably, a form of annotation. With our extensions there are also pre-defined regions handled by the tool. While this scheme is perhaps less flexible, it demands no region update annotations in program text.

Acyclicity can be enforced by way of user-provided class invariants that rely on the fact that certain ghost fields over-approximate reachability (even without manually maintaining regions, a similar effect could be achieved with the incomplete encoding of transitive closure in first-order logic given in [10]). These invariants can be temporally violated and must be provably restored at a later stage. This differs from our approach in two important ways. Firstly, it is the user, but not the compiler, who is aware of these properties. Therefore this knowledge cannot be transparently exploited by the compiler, for instance, for better memory-management (e.g., switching to a reference counting scheme for certain types) or loop-termination analysis. The second difference is that in our approach the properties cannot be temporarily broken and later restored but are preserved throughout the execution of the methods. This has the advantage of being enforceable by relatively simple checks. The price to pay, in the case of trees, is the introduction of a second region (cf. Fig. 10).

One can argue that class invariants expressing acyclicity constraints on complex types such as abstract syntax trees with subformula sharing could easily become unwiedly; on the other hand, they would be trivial to express with our approach. However, a systematic comparison of the annotation burden, both qualitative and quantitative, in Dafny with and without our extensions is needed.

Instead of using the representation regions of the dynamic frames methodology, one could instead enforce acyclicity using the *representation containment* concept of Ownership Types (see, e.g. [2]). In fact, the object graph structure obtained using this approach is typically a tree. However, since every object is required to have at most one owner, it is very non-trivial (if at all possible, depending the setting) to enforce a DAG, like it would be the case, for instance, in a Queue implemented with pointers to the first and last nodes of a linked list.

The most common approach to this problem is via *shape analysis* techniques (e.g., [13]). These are used to determine shape invariants for programs that perform destructive updates on dynamically allocated storage. In [6] a method is presented that automatically verifies acyclic linked lists, by carefully defining axioms for modeling the standard list operations. In [11] the authors introduce a type system which controls acyclicity by defining the concept of regions in which cycles are only allowed inside a region and forcing a partial order within regions.

8 Conclusions

In this work we tried to demonstrate that one can take advantage of the machinery required for dynamic frames to enforce non-trivial structural invariants in abstract data types. In particular, it is possible to guarantee acyclicity practically without performing changes to code. We developed similar techniques to enforce tree-shapedness and, finally, we discussed the use of additional representation regions as means to reject fewer valid programs. We also reported on some preliminary results using a prototypic implementation of some of these ideas.

We believe that besides providing correctness guarantees, these kind of structural invariants can be exploited by the compiler; examples include sound heuristics for analyzing program termination, and enabling cheaper memory management schemes (e.g., reference counting).

As future work, we would like to look at the interplay between the introduced techniques and classical features of object-oriented languages such us inheritance and dynamic binding. Moreover, while acyclic data structures are ubiquitous, so are cyclic ones (e.g. doubly-linked lists, circular buffers, etc.). We believe that common *cyclicity patterns* exist that can be enforced using similar techniques.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'04*, 2004.
2. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Notes*, 1998.
3. K. Huizing and R. Kuiper. Verification of object oriented programs using class invariants. In *FASE '00*, 2000.
4. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative Programs as Proofs. In *VSTTE Workshop on Tools & Experiments*, 2010.
5. I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 2010.
6. S. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL '06*, 2006.
7. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, 2010.
8. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP '09*, 2009.
9. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. In *TOPLAS '02*, 2002.
10. T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE'05*, 2005.
11. Y. Lu and J. Potter. A type system for reachability and acyclicity. In *ECOOP'05*, 2005.
12. A. Neisen. Automatic verification of acyclic data structures using theorem provers. Master's thesis, Universidad de Buenos Aires, 2010.
13. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *TOPLAS '02*, 2002.
14. J. Smans, B. Jacobs, and F. Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In *FMOODS '08*, 2008.