Verification of Programs with Pointers in SPARK

Georges-Axel Jaloyan[1,2(✉)], Claire Dross[3], Maroua Maalej[3], Yannick Moy[3], and Andrei Paskevich[4,5]

[1] École normale supérieure, PSL University, Paris, France
georges-axel.jaloyan@ens.fr
[2] CEA, DAM, DIF, 91297 Arpajon, France
[3] AdaCore, Paris, France
[4] LRI, Université Paris-Sud, CNRS, 91405 Orsay, France
[5] Inria Saclay, Université Paris Saclay, 91120 Palaiseau, France

Abstract. In the field of deductive software verification, programs with pointers present a major challenge due to pointer aliasing. In this paper, we introduce pointers to SPARK, a well-defined subset of the Ada language, intended for formal verification of mission-critical software. Our solution uses a permission-based static alias analysis method inspired by Rust's *borrow-checker* and *affine types*. To validate our approach, we have implemented it in the SPARK GNATprove formal verification toolset for Ada. In the paper, we give a formal presentation of the analysis rules for a core version of SPARK and discuss their implementation and scope.

1 Introduction

SPARK [1] is a subset of the Ada programming language targeted at safety- and security-critical applications. SPARK restrictions ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic diagnosis of conformance between a program specification and its implementation. As a consequence, it forbids the use of features that either prevent automatic proof, or make it possible only at the expense of extensive user annotation effort. The lack of support for pointers is the main example of this choice.

Among the various problems related to the use of pointers in the context of formal program verification, the most difficult problem is that two names may refer to overlapping memory locations, a.k.a. aliasing. Formal verification platforms that support pointer aliasing like Frama-C [2] require users to annotate programs to specify when pointers are not aliased. This can take the form of inequalities between pointers when a typed memory model is used, or the form of separation predicates between memory zones when an untyped memory model is

This work is partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007) and project VECOLIB (ANR-14-CE28-0018) of the French National Research Agency (ANR).

S.-W. Lin et al. (Eds.): ICFEM 2020, LNCS 12531, pp. 55–72, 2020.
https://doi.org/10.1007/978-3-030-63406-3_4

used. In both cases, the annotation burden is acceptable for leaf functions which manipulate single-level pointers, and quickly becomes overwhelming for functions that manipulate pointer-rich data structures. In parallel to the increased cost of annotations, the benefits of automation decrease, as automatic provers have difficulties reasoning explicitly with these inequalities and separation predicates.

Programs often rely on non-aliasing in general for correctness, when such aliasing would introduce interferences between two unrelated names. We call aliasing *potentially harmful* when a memory location modified through one name could be read through another name, within the scope of a verification condition. Otherwise, the aliasing is *benign*, when the memory location is only read through both names. A reasonable approach to formal program verification is thus to detect and forbid potentially harmful aliasing of names. Although this restricted language fragment cannot include all pointer-manipulating programs, it still allows us to introduce pointers to SPARK with minimal overhead for its program verification engine.

In this paper, we provide a formal description of the inclusion of pointers in the Ada language subset supported in SPARK, generalizing intuitions that can be found in [3,4] or on Adacore's blog [5,6]. As our main contribution, we show that it is possible to borrow and adapt the ideas underlying the safe support for pointers in permission-based languages like Rust, to safely restrict the use of pointers in usual imperative languages like Ada.

The rest of the paper is organized as follows. In Sect. 2, we give an informal description of our approach. Section 3 introduces a small formal language for which we define the formal alias analysis rules in Sect. 4. In Sect. 5, we describe the implementation of the analysis in GNATProve, a formal verification tool for Ada, and discuss some limitations with the help of various examples. We survey related works in Sect. 6 and future works in Sect. 7.

2 Informal Overview of Alias Analysis in SPARK

In Ada, the access to memory areas is given through *paths* that start with an identifier (a variable name) and follow through record fields, array indices, or through a special field `all`, which corresponds to pointer dereferencing. In what follows, we only consider record and pointer types, and discuss the treatment of arrays in Sect. 5.

As an example, we use the following Ada type, describing singly linked lists where each node carries a Boolean flag and a pointer to a shared integer value.

```
type List is record
  Flag : Boolean;
  Key  : access Integer;
  Next : access List;
end record;
```

Given a variable `A : List`, the paths `A.Flag`, `A.Key.all`, `A.Next.all.Key` are valid and their respective types are `Boolean`, `Integer`, and `access Integer` (a pointer to an `Integer`). The important difference between pointers and

records in Ada is that—similarly to C—assignment of a record copies the values of fields, whereas assignment of a pointer only copies the address and creates an alias.

The alias analysis procedure runs after the type checking. The idea is to associate one of the four permissions—RW, R, W or NO—to each possible path (starting from the available variables) at each sequence point in the program. A set of rules ensures that for any two aliased pointers, at most one has the ownership of the underlying memory area, that means the ability to read and modify it.

The absence of permission is denoted as the NO permission. Any modification or access to the value accessible from the path is forbidden. This typically applies to aliased memory areas that have lost the ownership over their stored values.

The *read-only* permission R allows us to read any value accessible from the path: use it in a computation, or pass it as an `in` parameter in a procedure call. As a consequence, if a given path has the R permission, then each valid extension of this path also has it.

The *write-only* permission W allows us to modify memory occupied by the value: use it on the left-hand side in an assignment or pass it as an `out` parameter in a procedure call. For example, having a write permission for a path of type `List` allows us to modify the `Flag` field or to change the addresses stored in the pointer fields `Key` and `Next`. However, this does not necessarily give us the permission to modify memory accessible from those pointers. Indeed, to dereference a pointer, we must read the address stored in it, which requires the read permission. Thus, the W permission only propagates to path extensions that do not dereference pointers, i.e., do not contain additional `all` fields.

The *read-write* permission RW combines the properties of the R and W permissions and grants the full ownership of the path and every value accessible from it. In particular, the RW permission propagates to all valid path extensions including those that dereference pointers. The RW permission is required to pass a value as an `in out` parameter in a procedure call.

Execution of program statements changes permissions. A simple example of this is procedure call: all `out` parameters must be assigned by the callee and get the RW permission after the call. The assignment statement is more complicated and several cases must be considered. If we assign a value that does not contain pointers (say, an integer or a pointer-free record), the whole value is copied into the left-hand side, and we only need to check that we have the appropriate permissions: W or RW for the left-hand side and R or RW for the right-hand side. However, whenever we copy a pointer, an alias is created. We want to make the left-hand side the new full owner of the value (i.e., give it the RW permission), and therefore, after the permission checks, we must revoke the permissions from the right-hand side, to avoid potentially harmful aliasing. The permission checks are also slightly different in this case, as we require the right-hand side to have the RW permission in order to move it to the left-hand side.

Let us now consider several simple programs and see how the permission checks allow us to detect potentially harmful aliasing.

```
procedure P1
  (A,B: in out List) is
begin
  A := B;
  B.Flag := True;
  B.Key.all := 42;
  -- A.Key.all == 42?
end P1;
```

```
procedure P2
  (A,B: in out access Integer) is
begin
  while B.all > 0 loop
    A.all := A.all + 1;
    B.all := B.all - 1;
    A := B;
  end loop;
  -- loop terminates?
end P2;
```

Fig. 1. Examples of potentially harmful aliasing, with some verification conditions that require tracking aliases throughout the program to be checked.

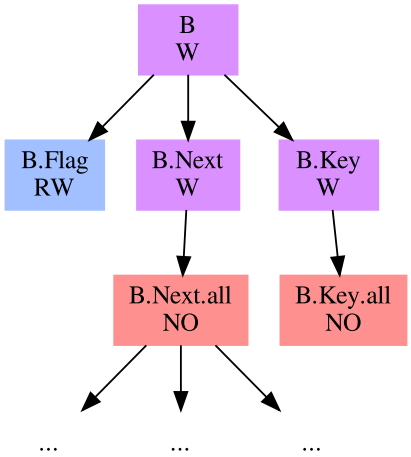

Fig. 2. Graphical representation of the permissions attributed to `B` and its extensions after assignment `A := B;` in `P1`.

Procedure `P1` in Fig. 1 receives two `in out` parameters `A` and `B` of type `List`. At the start of the procedure, all `in out` parameters assume permission RW. In particular, this implies that each `in out` parameter is separated from all other parameters, in the sense that no memory area can be reached from two different parameters. The first assignment copies the structure `B` into `A`. Thus, the paths `A.Flag`, `A.Key`, and `A.Next` are separated, respectively, from `B.Flag`, `B.Key`, and `B.Next`. However, the paths `A.Key.all` and `B.Key.all` are aliased, and `A.Next.all` and `B.Next.all` are aliased as well.

The first assignment does not change the permissions of `A` and its extensions: they retain the RW permission and keep the full ownership of their respective memory areas, even if the areas themselves have changed. The paths under `B`, however, must relinquish (some of) their permissions, as shown in Fig. 2. The paths `B.Key.all` and `B.Next.all` as well as all their extensions get the NO permission, that is, lose both read and write permissions. This is necessary, as

the ownership over their memory areas is transferred to the corresponding paths under `A`. The paths `B`, `B.Key`, and `B.Next` lose the read permission but keep the write-only W permission. Indeed, we forbid reading from memory that can be altered through a concurrent path. However, it is allowed to "redirect" the pointers `B.Key` and `B.Next`, either by assigning those fields directly or by copying some different record into `B`. The field `B.Flag` is not aliased, nor has aliased extensions, and thus retains the initial RW permission. This RW permission allows us to perform the assignment `B.Flag := True` on the next line.

The third assignment, however, is now illegal, since `B.Key.all` does not have the write permission anymore. What is more, at the end of the procedure the `in out` parameters `A` and `B` are not separated. This is forbidden, as the caller assumes that all `out` and `in out` parameters are separated after the call just as they were before.

Procedure `P2` in Fig. 1 receives two pointers `A` and `B`, and manipulates them inside a while loop. Since the permissions are assigned statically, we must ensure that at the end of a single iteration, we did not lose the permissions necessary for the next iteration. This requirement is violated in the example: after the last assignment `A := B`, the path `B` receives permission W and the path `B.all`, permission NO, as `B.all` is now an alias of `A.all`. The new permissions for `B` and `B.all` are thus weaker than the original ones (RW for both), and the procedure is rejected. Should it be accepted, we would have conflicting memory modifications from two aliased paths at the beginning of the next iteration.

3 μSPARK Language

For the purposes of formal presentation, we introduce μSPARK, a small subset of SPARK featuring pointers, records, loops, and procedure calls. We present the syntax of μSPARK, and define the rules of alias safety.

The data types of μSPARK are as follows:

type ::=	`Integer` | `Real` | `Boolean`	scalar type
|	`access` *type*	access type (pointer)
|	*ident*	record type

Every μSPARK program starts with a list of record type declarations:

$$\begin{aligned} \mathit{record} &::= \texttt{type}\ \mathit{ident}\ \texttt{is record}\ \mathit{field}^{\star}\ \texttt{end} \\ \mathit{field} &::= \mathit{ident}\ \texttt{:}\ \mathit{type} \end{aligned}$$

We require all field names to be distinct. The field types must not refer to the record types declared later in the list. Nevertheless, a record type R can be made recursive by adding a field whose type is a pointer to R (written `access` R). We discuss the handling of array types in Sect. 5.

The syntax of μSPARK statements is defined by the following rules:

path	::=	*ident*	variable
	|	*path* `.` *ident*	record field
	|	*path* `.all`	pointer dereference
expr	::=	*path*	l-value
	|	`42` | `3.14` | `True` | `False` | ...	scalar value
	|	*expr* (`+` | `-` | `<` | `=` | ...) *expr*	binary operator
	|	`null`	null pointer
stmt	::=	*path* `:=` *expr*	assignment
	|	*path* `:= new` *type*	allocation
	|	`if` *expr* `then` *stmt** `else` *stmt** `end`	conditional
	|	`while` *expr* `loop` *stmt** `end`	"while" loop
	|	*ident* `(` *expr** `)`	procedure call

Following the record type declarations, a μSPARK program contains a set of mutually recursive procedure declarations:

procedure ::= `procedure` *ident* `(` *param** `)` `is` *local** `begin` *stmt** `end`

param ::= *ident* `:` (`in` | `in out` | `out`) *type*

local ::= *ident* `:` *type*

We require all formal parameters and local variables in a procedure to have distinct names. A procedure call can only pass left-values (i.e., paths) for `in out` and `out` parameters. The execution starts from a procedure named `Main` with the empty parameter list.

The type system for μSPARK is rather standard and we do not show it here in full. We assume that binary operators only operate on scalar types. The null pointer can have any pointer type `access` τ. The dereference operator `.all` converts `access` τ to τ. Allocation p `:= new` τ requires path p to have type `access` τ. In what follows, we only consider well-typed μSPARK programs. A formal semantics for μSPARK statements is given in Appendix A.

On the semantic level, we need to distinguish the units of allocation, such as whole records, from the units of access, such as individual record fields. We use the term *location* to refer to the memory area occupied by an allocated value. We treat locations as elements of an abstract infinite set, and denote them with letter ℓ. We use the term *address* to designate either a location, denoted ℓ, or a specific component inside the location of a record, denoted $\ell.f.g$, where f and g are field names (assuming that at ℓ we have a record whose field f is itself a record with a field g). A *value* is either a scalar, an address, a null pointer or a record, that is, a finite mapping from field names to values.

A μSPARK program is executed in the context defined by a *binding* Υ that maps variable names to addresses and a *store* Σ that maps locations to values. By a slight abuse of notation, we apply Σ to arbitrary addresses, so that $\Sigma(\ell.f)$ is $\Sigma(\ell)(f)$, the value of the field f of the record value stored in Σ at ℓ.

The evaluation of expressions is effect-free and is denoted $[\![e]\!]^{\Upsilon}_{\Sigma}$. We also need to evaluate l-values to the corresponding addresses in the store, written $\langle p \rangle^{\Upsilon}_{\Sigma}$, where p is the evaluated path. Illicit operations, such as dereferencing a null pointer, cannot be evaluated and stall the execution (*blocking semantics*). In the formal rules below, c stands for a scalar constant and $\odot$, for a binary operator:

$$\langle x \rangle^{\Upsilon}_{\Sigma} = \Upsilon(x) \qquad \langle p.f \rangle^{\Upsilon}_{\Sigma} = \langle p \rangle^{\Upsilon}_{\Sigma}.f \qquad \langle p.\texttt{all} \rangle^{\Upsilon}_{\Sigma} = [\![p]\!]^{\Upsilon}_{\Sigma}$$

$$[\![c]\!]^{\Upsilon}_{\Sigma} = c \qquad [\![p]\!]^{\Upsilon}_{\Sigma} = \Sigma(\langle p \rangle^{\Upsilon}_{\Sigma}) \qquad [\![\texttt{null}]\!]^{\Upsilon}_{\Sigma} = \texttt{null}$$

$$[\![e_1 \odot e_2]\!]^{\Upsilon}_{\Sigma} = [\![e_1]\!]^{\Upsilon}_{\Sigma} \odot [\![e_2]\!]^{\Upsilon}_{\Sigma}$$

4 Access Policies, Transformers, and Alias Safety Rules

We denote paths with letters p and q. We write $p \sqsubset q$ to denote that p is a strict *prefix* of q or, equivalently, q is a strict *extension* of p. In what follows, we always mean strict prefixes and extensions, unless explicitly said otherwise.

In the typing context of a given procedure, a well-typed path is said to be *deep* if it has a non-strict extension of an access type, otherwise it is called *shallow*. We extend these notions to types: a type τ is deep (resp. shallow) if and only if a τ-typed path is deep (resp. shallow). In other words, a path or a type is deep if a pointer can be reached from it, and shallow otherwise. For example, the `List` type in Sect. 2 is a deep type, and so is `access Integer`, whereas any scalar type or any record with scalar fields only is shallow.

An extension q of a path p is called a *near extension* if it has as many pointer dereferences as p, otherwise it is a *far extension*. For instance, given a variable `A` of type `List`, the paths `A.Flag`, `A.Key`, and `A.Next` are the near extensions of `A`, whereas `A.Key.all`, `A.Next.all`, and their extensions are far extensions, since they all create an additional pointer dereference by passing through `all`.

We say that *sequence points* are the program points before or after a given statement. For each sequence point in a given μSPARK program, we statically compute an *access policy*: a partial function that maps each well-typed path to one of the four *permissions*: RW, R, W, and NO, which form a diamond lattice: RW > R|W > NO. We denote permissions by π and access policies by Π.

Permission transformers modify policies at a given path, as well as its prefixes and extensions. Symbolically, we write $\Pi \xrightarrow{T}_p \Pi'$ to denote that policy Π' results from application of transformer T to Π at path p. We define a composition operation $\Pi \xrightarrow{T_1}_{p_1} ; \xrightarrow{T_2}_{p_2} \Pi'$ that allows chaining the application of permission transformers T_1 at path p_1 and T_2 at path p_2 to Π resulting in the policy Π'. We write $\Pi \xrightarrow{T_1 ; T_2}_p \Pi'$ as an abbreviation for $\Pi \xrightarrow{T_1}_p ; \xrightarrow{T_2}_p \Pi'$ (that is, for some Π'', $\Pi \xrightarrow{T_1}_p \Pi'' \xrightarrow{T_2}_p \Pi'$). We write $\Pi \xrightarrow{T}_{p,q} \Pi'$ as an abbreviation for $\Pi \xrightarrow{T}_p ; \xrightarrow{T}_q \Pi'$.

Permission transformers can also apply to expressions, which consists in updating the policy for every path in the expression. This only includes paths that occur as sub-expressions: in an expression `X.f.g + Y.h`, only the paths

`X.f.g` and `Y.h` are concerned, whereas `X`, `X.f` and `Y` are not. The order in which the individual paths are treated must not affect the final result.

$$\frac{\Pi \xrightarrow{\text{move}}_e \fatsemi \xrightarrow{\text{check W} \fatsemi \text{fresh RW} \fatsemi \text{lift}}_p \Pi'}{\Pi \cdot p := e \to \Pi'} \quad \text{(P-ASSIGN)}$$

$$\frac{\Pi \xrightarrow{\text{check W} \fatsemi \text{fresh RW} \fatsemi \text{lift}}_p \Pi'}{\Pi \cdot p := \texttt{new}\ \tau \to \Pi'} \quad \text{(P-ALLOC)}$$

$$\frac{\Pi \xrightarrow{\text{check R}}_e \Pi \qquad \Pi \cdot \bar{s}_1 \to \Pi_1 \qquad \Pi \cdot \bar{s}_2 \to \Pi_2 \qquad \forall p.\, \Pi'(p) = \Pi_1(p) \wedge \Pi_2(p)}{\Pi \cdot \texttt{if}\ e\ \texttt{then}\ \bar{s}_1\ \texttt{else}\ \bar{s}_2\ \texttt{end} \to \Pi'} \quad \text{(P-IF)}$$

$$\frac{\Pi \xrightarrow{\text{check R}}_e \Pi \qquad \Pi \cdot \bar{s} \to \Pi' \qquad \forall \pi.\, \Pi'(\pi) \geqslant \Pi(\pi)}{\Pi \cdot \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end} \to \Pi} \quad \text{(P-WHILE)}$$

$$\frac{\begin{array}{c}\texttt{procedure}\ P\ (\ a_1 : \texttt{in}\ \tau_{a_1}\ ; \ldots ; b_1 : \texttt{in out}\ \tau_{b_1}\ ; \ldots ; c_1 : \texttt{out}\ \tau_{c_1}\ ; \ldots) \\ \texttt{is}\ \cdots\ \texttt{begin}\ \bar{s}\ \texttt{end}\ \text{ is declared in the program} \\ \Pi \xrightarrow{\text{check R} \fatsemi \text{observe}}_{e_{a_1},\ldots} \fatsemi \xrightarrow{\text{check RW} \fatsemi \text{borrow}}_{p_{b_1},\ldots} \fatsemi \xrightarrow{\text{check W} \fatsemi \text{borrow}}_{q_{c_1},\ldots} \Pi'' \\ \Pi \xrightarrow{\text{fresh RW} \fatsemi \text{lift}}_{p_{b_1},\ldots} \fatsemi \xrightarrow{\text{fresh RW} \fatsemi \text{lift}}_{q_{c_1},\ldots} \Pi'\end{array}}{\Pi \cdot P(e_{a_1}, \ldots, p_{b_1}, \ldots, q_{c_1}, \ldots) \to \Pi'} \quad \text{(P-CALL)}$$

Fig. 3. Alias safety rules for statements.

We define the rules of alias safety for μSPARK statements in the context of a current access policy. An *alias-safe* statement yields an updated policy which is used to check the subsequent statement. We write $\Pi \cdot s \to \Pi'$ to denote that statement s is safe with respect to policy Π and yields the updated policy Π'. We extend this notation to sequences of statements in an obvious way, as the reflexive-transitive closure of the update relation on Π. The rules for checking the alias safety of statements are given in Fig. 3. These rules use a number of permission transformers such as 'fresh', 'check', 'move', 'observe', and 'borrow', which we define and explain below.

Let us start with the (P-ASSIGN) rule. Assignments grant the full ownership over the copied value to the left-hand side. If we copy a value of a shallow type, we merely have to ensure that the right-hand side has the read permission. Whenever we copy a deep-typed value, aliases may be created, and we must check that the right-hand side is initially the sole owner of the copied value (that is, possesses the RW permission) and revoke the ownership from it.

To define the 'move' transformer that handles permissions for the right-hand side of an assignment, we need to introduce several simpler transformers.

Definition 1. *Permission transformer* check π *does not modify the access policy and only verifies that a given path* p *has permission* π *or greater. In other words,* $\Pi \xrightarrow{\text{check } \pi}_p \Pi'$ *if and only if* $\Pi(p) \geqslant \pi$ *and* $\Pi = \Pi'$. *This transformer also*

applies to expressions: $\Pi \xrightarrow{\text{check } \pi}_e \Pi'$ *states that* $\Pi \xrightarrow{\text{check } \pi}_p \Pi' (= \Pi)$ *for every path* p *occurring in* e.

Definition 2. *Permission transformer* fresh π *assigns permission* π *to a given path* p *and all its extensions.*

Definition 3. *Permission transformer* cut *assigns restricted permissions to a deep path* p *and its extensions: the path* p *and its near deep extensions receive permission* W, *the near shallow extensions keep their current permissions, and the far extensions receive permission* NO.

Going back to the procedure `P1` in Fig. 1, the change of permissions on the right-hand side after the assignment `A := B` corresponds to the definition of 'cut'. In the case where the right-hand side of an assignment is a deep path, we also need to change permissions of the prefixes, to reflect the ownership transfer.

Definition 4. *Permission transformer* block *propagates the loss of the read permission from a given path to all its prefixes. Formally, it is defined by the following rules, where* x *stands for a variable and* f *for a field name:*

$$\frac{}{\Pi \xrightarrow{\text{block}}_x \Pi} \qquad \frac{\Pi[p \mapsto \mathsf{W}] \xrightarrow{\text{block}}_p \Pi'}{\Pi \xrightarrow{\text{block}}_{p.all} \Pi'}$$

$$\frac{\Pi(p) = \mathsf{NO}}{\Pi \xrightarrow{\text{block}}_{p.f} \Pi} \qquad \frac{\Pi(p) \geqslant \mathsf{W} \quad \Pi[p \mapsto \mathsf{W}] \xrightarrow{\text{block}}_p \Pi'}{\Pi \xrightarrow{\text{block}}_{p.f} \Pi'}$$

Definition 5. *Permission transformer* move *applies to expressions:*

- *if* e *has a shallow type, then* $\Pi \xrightarrow{\text{move}}_e \Pi' \Leftrightarrow \Pi \xrightarrow{\text{check R}}_e \Pi'$,
- *if* e *is a deep path* p, *then* $\Pi \xrightarrow{\text{move}}_e \Pi' \Leftrightarrow \Pi \xrightarrow{\text{check RW ; cut ; block}}_p \Pi'$,
- *if* e *is* `null`, *then* $\Pi \xrightarrow{\text{move}}_e \Pi' \Leftrightarrow \Pi' = \Pi$.

To further illustrate the 'move' transformer, let us consider two variables `P` and `Q` of type `access List` and an assignment `P := Q.all.Next`. We assume that `Q` and all its extensions have full ownership (RW) before the assignment. We apply the second case in the definition of 'move' to the deep path `Q.all.Next`. The 'check RW' condition is verified, and the 'cut' transformer sets the permission for `Q.all.Next` to W and the permission for `Q.all.Next.all` and all its extensions to NO. Indeed, `P.all` becomes an alias of `Q.all.Next.all` and steals the full ownership for this memory area. However, we still can reassign `Q.all.Next` to a different address. Moreover, we still can write some new values into `Q.all` or `Q`, without compromising safety. This is enforced by the application of the 'block' transformer at the end. We cannot keep the read permission for `Q` or `Q.all`, since it implies the read access to the data under `Q.all.Next.all`.

Finally, we need to describe the change of permissions on the left-hand side of an assignment, in order to reflect the gain of the full ownership. The idea is that as soon as we have the full ownership for each field of a record, we can assume the full ownership of the whole record, and similarly for pointers.

Definition 6. *Permission transformer* lift *propagates the* RW *permission from a given path to its prefixes, wherever possible:*

$$\frac{}{\Pi \xrightarrow{\text{lift}}_x \Pi} \qquad \frac{\Pi[p \mapsto \mathsf{RW}] \xrightarrow{\text{lift}}_p \Pi'}{\Pi \xrightarrow{\text{lift}}_{p.\mathtt{all}} \Pi'}$$

$$\frac{\forall q \sqsupset p.\, \Pi(q) = \mathsf{RW} \quad \Pi[p \mapsto \mathsf{RW}] \xrightarrow{\text{lift}}_p \Pi'}{\Pi \xrightarrow{\text{lift}}_{p.f} \Pi'} \qquad \frac{\exists q \sqsupset p.\, \Pi(q) \neq \mathsf{RW}}{\Pi \xrightarrow{\text{lift}}_{p.f} \Pi}$$

In the (P-ASSIGN) rule, we revoke the permissions from the right-hand side of an assignment before granting the ownership to the left-hand side. This is done in order to prevent creation of circular data structures. Consider an assignment `A.Next.all := A`, where `A` has type `List`. According to the definition of 'move', all far extensions of the right-hand side, notably `A.Next.all`, receive permission NO. This makes the left-hand side fail the write permission check.

Allocations `p := new` τ are handled by the (P-ALLOC) rule. We grant the full permission on the newly allocated memory, as it cannot possibly be aliased.

In a conditional statement, the policies at the end of the two branches are merged selecting the most restrictive permission for each path. Loops require that no permissions are lost at the end of a loop iteration, compared to the entry, as explained above for procedure `P2` in Fig. 1.

Procedure calls guarantee to the callee that every argument with mode `in`, `in out`, or `out` has at least permission R, RW or W, respectively. To ensure the absence of potentially harmful aliasing, we revoke the necessary permissions using the 'observe' and 'borrow' transformers.

Definition 7. *Permission transformer* borrow *assigns permission* NO *to a given path p and all its prefixes and extensions.*

Definition 8. *Permission transformer* freeze *removes the write permission from a given path p and all its prefixes and extensions. In other words,* freeze *assigns to each path q comparable to p the minimum permission* $\Pi(q) \wedge \mathsf{R}$.

Definition 9. *Permission transformer* observe *applies to expressions:*

- *if e has a shallow type, then* $\Pi \xrightarrow{\text{observe}}_e \Pi' \Leftrightarrow \Pi' = \Pi$,
- *if e is a deep path p, then* $\Pi \xrightarrow{\text{observe}}_e \Pi' \Leftrightarrow \Pi \xrightarrow{\text{freeze}}_p \Pi'$,
- *if e is* `null`*, then* $\Pi \xrightarrow{\text{observe}}_e \Pi' \Leftrightarrow \Pi' = \Pi$.

We remove the write permission from the deep-typed `in` parameters using the 'observe' transformer, in order to allow aliasing between the read-only paths. As for the `in out` and `out` parameters, we transfer the full ownership over them to the callee, which is reflected by dropping every permission on the caller's side using 'borrow'.

In the (P-CALL) rule, we revoke permissions right after checking them for each parameter. In this way, we cannot pass, for example, the same path as an `in` and `in out` parameter in the same call. Indeed, the 'observe' transformer

will remove the write permission, which is required by 'check RW' later in the transformer chain. At the end of the call, the callee transfers to the caller the full ownership over each `in out` and `out` parameter.

We apply our alias safety analysis to each procedure declaration. We start with an empty access policy, denoted $\varnothing$. Then we fill the policy with the permissions for the formal parameters and the local variables and check the procedure body. At the end, we verify that every `in out` and `out` parameter has the RW permission. Formally, this is expressed with the following rule:

$$\frac{\varnothing \xrightarrow{\text{fresh R}}_{a_1,\ldots} \fatsemi \xrightarrow{\text{fresh RW}}_{b_1,\ldots} \fatsemi \xrightarrow{\text{fresh W} \fatsemi \text{cut}}_{c_1,\ldots} \fatsemi \xrightarrow{\text{fresh RW}}_{d_1,\ldots} \Pi' \qquad \Pi' \cdot \bar{s} \to \Pi'' \qquad \Pi''(b_1) = \cdots = \Pi''(c_1) = \cdots = \mathsf{RW}}{\begin{array}{r}\texttt{procedure}\ P\ (\ a_1 : \texttt{in}\ \tau_{a_1}\ ;\ \ldots\ ;\ b_1 : \texttt{in out}\ \tau_{b_1}\ ;\ \ldots\ ;\ c_1 : \texttt{out}\ \tau_{c_1}\ ;\ \ldots\ )\\ \texttt{is}\ d_1 : \tau_{d_1}\ ;\ \ldots\ \texttt{begin}\ \bar{s}\ \texttt{end}\quad \text{is alias-safe}\end{array}}$$

We say that a μSPARK program is *alias-safe* if all its procedures are.

By the end of the analysis, an alias-safe program has an access policy associated to each sequence point in it. We say that an access policy Π is *consistent* whenever it satisfies the following conditions for all valid paths π, $\pi.f$, $\pi.\texttt{all}$:

$$\Pi(\pi) = \mathsf{RW} \implies \Pi(\pi.f) = \mathsf{RW} \qquad \Pi(\pi) = \mathsf{RW} \implies \Pi(\pi.\texttt{all}) = \mathsf{RW} \tag{1}$$

$$\Pi(\pi) = \mathsf{R} \implies \Pi(\pi.f) = \mathsf{R} \qquad \Pi(\pi) = \mathsf{R} \implies \Pi(\pi.\texttt{all}) = \mathsf{R} \tag{2}$$

$$\Pi(\pi) = \mathsf{W} \implies \Pi(\pi.f) \geq \mathsf{W} \tag{3}$$

These invariants correspond to the informal explanations given in Sect. 2. Invariant (1) states that the full ownership over a value propagates to all values reachable from it. Invariant (2) states that the read-only permission must also propagate to all extensions. Indeed, a modification of a reachable component can be observed from any prefix. Invariant (3) states that write permission over a record value implies a write permission over each of its fields. However, the write permission does not necessarily propagate across pointer dereference.

Lemma 1 (Policy Consistency). *The alias safety rules in Fig. 3 preserve policy consistency.*

When, during an execution, we arrive at a given sequence point with the set of variable bindings Υ, store Σ, and statically computed and consistent access policy Π, we say that the state of the execution respects the *Concurrent Read, Exclusive Write* condition (CREW), if and only if for any two distinct valid paths p and q, $\langle p \rangle_\Sigma^\Upsilon = \langle q \rangle_\Sigma^\Upsilon \wedge \Pi(p) \geq \mathsf{W} \implies \Pi(q) = \mathsf{NO}$.

The main result about the soundness of our approach is as follows.

Theorem 1 (Soundness). *A terminating evaluation of a well-typed alias-safe μSPARK program respects the CREW condition at every sequence point.*

The full proof, for a slightly different definition of μSPARK, is given in [7]. The argument proceeds by induction on the evaluation derivation, following the

rules provided in Appendix A. The only difficult cases are assignment, where the required permission withdrawal is ensured by the 'move' transformer, and procedure call, where the chain of 'observe' and 'borrow' transformers, together with the corresponding checks, on the caller's side, ensures that the CREW condition is respected at the beginning of the callee.

For the purposes of verification, an alias-safe program can be treated with no regard for sharing. More precisely, we can safely transform access types into records with a single field that contains either `null` or the referenced value. Since records are copied on assignment, we obtain a program that can be verified using the standard rules of Floyd-Hoare logic or weakest-precondition calculus (as the rules have also ensured the absence of aliasing between procedure parameters).

Indeed, consider an assignment `A := B` where `A` and `B` are pointers. In an alias-safe program, `B` loses its ownership over the referenced value and cannot be used anymore without being reassigned. Then, whenever we modify that value through `A.all`, we do not need to update `B.all` in the verification condition. In other words, we can safely treat `A := B` as a *deep copy* of `B.all` into `A.all`. The only adjustment that needs to be made to the verification condition generator consists in adding checks against the null pointer dereferencement, which is not handled by our rules.

5 Implementation and Evaluation

The alias safety rules presented above have been implemented in the SPARK proof tool, called GNATprove. The real SPARK subset differs from μSPARK in several respects: arrays, functions, additional loop constructs, and global variables. For arrays, permission rules apply to all elements, without taking into account the exact index of an element, which may not be known statically in the general case. Functions return values and cannot perform side effects. They only take `in` parameters and may be called inside expressions. To avoid creating aliases between the function parameters and the returned value, the full RW permission is required on the latter at the end of the callee. The rules for loops have been extended to handle for-loops and plain loops (which have no exit condition), and also the `exit` (break) statements inside loops. Finally, global variables are considered as implicit parameters of subprograms that access them, with mode depending on whether the subprogram reads and/or modifies the variable.

Though our alias safety rules are constraining, we feel that they significantly improve the expressive power of the SPARK subset. To demonstrate it, let us go over some examples.[1] One of the main uses of pointers is to serve as references to avoid copying potentially big data structures. We believe this use case is supported as long as the CREW condition is respected. We demonstrate this on a small procedure that swaps two pointers.

[1] https://github.com/GAJaloyan/SPARKExamples.

```
type Int_Ptr is access Integer;

procedure Swap (X, Y: in out Int_Ptr) is
  T : Int_Ptr := X; -- ownership of X is moved to T, X gets 'W'
begin
  X := Y; -- ownership of Y is moved to X, Y gets 'W', X gets 'RW'
  Y := T; -- ownership of T is moved to Y, T gets 'W', Y gets 'RW'
  return; -- when exiting Swap, X and Y should be 'RW'
end Swap; -- local variable T is not required to have any permission
```

This code is accepted by our alias safety rules. We can provide it with a contract, which can then be verified by the SPARK proof tool.

```
procedure Swap (X, Y: in out Int_Ptr) with
  Pre  => X /= null and Y /= null,
  Post => X.all = Y.all'Old and Y.all = X.all'Old;
```

Another common use case for pointers in Ada is to store indefinite types (that is, the types whose size is not known statically, such as `String`) inside aggregate data structures like arrays or records. The usual workaround consists in storing pointers to indefinite elements instead. This usage is also supported by our alias analysis, as illustrated by an implementation of word sets, which is accepted and fully verified by SPARK.

```
type Red_Black is (Red, Black);
type Tree;
type Tree_Ptr is access Tree;
type Tree is record
  Value : Integer;
  Color : Red_Black;
  Left  : Tree_Ptr;
  Right : Tree_Ptr;
end record;

procedure Rotate_Left
 (T: in out Tree_Ptr)
is
  X: Tree := T.Right;
begin
  T.Right := X.Left;
  X.Left := T;
  T := X;
end Rotate_Left;
```

```
procedure Insert_Rec
   (T: in out Tree_Ptr;
    V: Integer) is
begin
  if T = null then
    T := new Tree'(
      Value => V,
      Color => Red,
      Left  => null,
      Right => null);
  elsif T.Value = V then
    return;
  elsif T.Value > V then
    Insert_Rec (T.Left, V);
  else
    Insert_Rec (T.Right, V);
  end if;
  Balance (T);
end Insert_Rec;
```

The last use case that we want to consider is the implementation of recursive data structures such as lists and trees. While alias safety rules exclude structures whose members do not have a single owner like doubly linked lists or arbitrary graphs, they are permissive enough for many non-trivial tree data structures, for example, red-black trees. To insert a value in a red-black tree, the tree is first traversed top-down to find the correct leaf for the insertion, and then it is

traversed again bottom-up to reestablish balancing. Doing this traversal iteratively requires storing a link to the parent node in children, which is not allowed as it would introduce an alias. Therefore, we went for a recursive implementation, partially shown above. The rotating functions, which are used by the `Balance` procedure (not shown here) can be implemented straightforwardly, since rotation moves pointers around without creating any cycles.

This example passes alias safety analysis successfully (*i.e.* without errors from the tool) and can be verified to be free of runtime exceptions (such as dereferences of null pointers) by the SPARK proof tool.

6 Related Work

The recent adoption of permission-based typing systems by programming languages is the culmination of several decades of research in this field. Going back as early as 1987 for Girard's linear logic [8] and 1983 for Ada's limited types [9], Baker was the first to suggest using linear types in programming languages [10], formalised in 1998 by Clarke et al. [11]. More recent works focus on Java, such as Javari and Uno [12,13].

Separation logic [14] is an extension of Hoare-Floyd logic that allows reasoning about pointers. In general, it is difficult to integrate into automated deductive verification: in particular, it is not directly supported by SMT provers, although some recent attempts try to have it mended [15,16].

Permission-based programming languages generalize the issue of avoiding harmful aliasing to the more general problem of preventing harmful sharing of resources (memory, but also network connections, files, etc.).

Cyclone and Rust achieve absence of harmful aliasing by enforcing an ownership type system on the memory pointed to by objects [17,18]. Furthermore, Rust has many sophisticated lifetime checks, that prevent dangling pointers, double free, and null pointer dereference. In SPARK, those checks are handled by separate analysis passes of the toolset. Even though there is still no formal description of Rust's borrow-checker, we must note a significant recent effort to provide a rigorous formal description of the foundations of Rust [19].

Dafny associates each object with its *dynamic frame*, the set of pointers that it owns [20]. This dynamic version of ownership is enforced by modeling the ownership of pointers in logic, generating verification conditions to detect violations of the single-owner model, and proving them using SMT provers. In Spec#, ownership is similarly enforced by proof, to detect violations of the so-called Boogie methodology [21].

In our work, we use a permission-based mechanism for detecting potentially harmful aliasing, in order to make the presence of pointers transparent for automated provers. In addition, our approach does not require additional user annotations, that are required in some of the previously mentioned techniques. We thus expect to achieve high automation and usability, which was our goal for supporting pointers in SPARK.

7 Future Work

The GNAT+SPARK Community release in 2020 contains support for pointers, as defined in Sect. 3.10 of the SPARK Reference Manual [22], with two important improvement not discussed in this paper: local observe/borrow operations and support for proof of absence of memory leaks.

Both these features require extensive changes to the generation of verification conditions. Support for local borrows requires special mechanisms to report changes on the borrower to the borrowee at the end of the borrow, as shown by recent work on Rust [23]. Support for proof of absence of memory leaks requires special mechanisms to track values that are either null or moved so that we can make sure that all values going out of scope are in this case.

8 Conclusion

In this paper, we have presented the rules for alias safety analysis that allow implementing and verifying in SPARK a wide range of programs using pointers and dynamic allocation. To the best of our knowledge, this is a novel approach to control aliasing introduced by arbitrary pointers in a programming language supported by proof. Our approach does not require additional user annotations or proof of additional verification conditions, which makes it much simpler to adopt. We provided a formalization of our rules for a subset of SPARK in order to mathematically prove the safety of our analysis.

In the future, we plan to extend our formalism and proof to non-terminating executions. For that purpose, we can provide a co-inductive definition of the big-step semantics and perform a similar co-inductive soundness proof, as described by Leroy and Grall [24].

Another long-term goal would be extending our analysis so that it could handle automatic reclamation, parallelism, initialization and lifetime checks, instead of relying on external checks.

A Semantics for μSPARK statements

We use big-step operational semantics and write $\Upsilon \cdot \Sigma \cdot s \Downarrow \Sigma'$ to denote that μSPARK statement s, when evaluated under binding Υ and store Σ, terminates with the state of the store Σ'. We extend this notation to sequences of statements in an obvious way, as the reflexive-transitive closure of the evaluation relation on Σ. Similarly, we write $\Sigma[\ell.f \mapsto v]$ to denote an update of a single field in a record, that is, $\Sigma[\ell \mapsto \Sigma(\ell)[f \mapsto v]]$. In this paper, we do not consider diverging statements.

Allocation adds a fresh address to the store, mapping it to a default value for the corresponding type: 0 for `Integer`, `False` for `Boolean`, `null` for the access types, and for the record types, a record value where each field has the default value. Notice that since pointers are initialised to `null`, there is no deep allocation. We write $\square_\tau$ to denote the default value of type τ.

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = v}{\Upsilon \cdot \Sigma \cdot p := e \Downarrow \Sigma[\langle p \rangle^{\Upsilon}_{\Sigma} \mapsto v]} \quad \text{(E-ASSIGN)}$$

$$\frac{\ell \notin dom\ \Sigma}{\Upsilon \cdot \Sigma \cdot p := \texttt{new}\ \tau \Downarrow \Sigma[\langle p \rangle^{\Upsilon}_{\Sigma} \mapsto \ell, \ell \mapsto \square_{\tau}]} \quad \text{(E-ALLOC)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{True} \qquad \Upsilon \cdot \Sigma \cdot \bar{s}_1 \Downarrow \Sigma'}{\Upsilon \cdot \Sigma \cdot \texttt{if}\ e\ \texttt{then}\ \bar{s}_1\ \texttt{else}\ \bar{s}_2 \Downarrow \Sigma'} \quad \text{(E-IFTRUE)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{False} \qquad \Upsilon \cdot \Sigma \cdot \bar{s}_2 \Downarrow \Sigma'}{\Upsilon \cdot \Sigma \cdot \texttt{if}\ e\ \texttt{then}\ \bar{s}_1\ \texttt{else}\ \bar{s}_2 \Downarrow \Sigma'} \quad \text{(E-IFFALSE)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{True} \qquad \Upsilon \cdot \Sigma \cdot (\bar{s}\ \texttt{;}\ \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end}) \Downarrow \Sigma'}{\Upsilon \cdot \Sigma \cdot \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end} \Downarrow \Sigma'} \quad \text{(E-WHILETRUE)}$$

$$\frac{[\![e]\!]^{\Upsilon}_{\Sigma} = \texttt{False}}{\Upsilon \cdot \Sigma \cdot \texttt{while}\ e\ \texttt{loop}\ \bar{s}\ \texttt{end} \Downarrow \Sigma} \quad \text{(E-WHILEFALSE)}$$

$$\frac{\begin{array}{c}\texttt{procedure}\ P\ (\ a_1 : \texttt{in}\ \tau_{a_1}\ \texttt{;}\ \ldots\ \texttt{;}\ b_1 : \texttt{in out}\ \tau_{b_1}\ \texttt{;}\ \ldots\ \texttt{;}\ c_1 : \texttt{out}\ \tau_{c_1}\ \texttt{;}\ \ldots\ )\\ \texttt{is}\ d_1 : \tau_{d_1}\ \texttt{;}\ \ldots\ \texttt{begin}\ \bar{s}\ \texttt{end}\ \text{ is declared in the program}\\ \ell_{a_1}, \ldots, \ell_{d_1}, \ldots \notin dom\ \Sigma \qquad [\![e_{a_1}]\!]^{\Upsilon}_{\Sigma} = v_{a_1}, \ldots\\ \Upsilon_P = [\,a_1 \mapsto \ell_{a_1}, \ldots, b_1 \mapsto \langle p_{b_1} \rangle^{\Upsilon}_{\Sigma}, \ldots, c_1 \mapsto \langle q_{c_1} \rangle^{\Upsilon}_{\Sigma}, \ldots, d_1 \mapsto \ell_{d_1}, \ldots]\\ \Sigma_P = \Sigma[\ell_{a_1} \mapsto v_{a_1}, \ldots, \ell_{d_1} \mapsto \square_{\tau_{d_1}}, \ldots] \qquad \Upsilon_P \cdot \Sigma_P \cdot \bar{s} \Downarrow \Sigma'\end{array}}{\Upsilon \cdot \Sigma \cdot P(e_{a_1}, \ldots, p_{b_1}, \ldots, q_{c_1}, \ldots) \Downarrow \{\ell_{a_1}, \ldots, \ell_{d_1}, \ldots\} \lhd \Sigma'} \quad \text{(E-CALL)}$$

Fig. 4. Semantics of μSPARK (terminating statements).

The evaluation rules are given in Fig. 4. In the (E-CALL) rule, we evaluate the procedure body in the dedicated context $\Upsilon_P \cdot \Sigma_P$. This context binds the `in` parameters to fresh locations containing the values of the respective expression arguments, binds the `in out` and `out` parameters to the addresses of the respective l-value arguments, and allocates memory for the local variables. At the end of the call, the memory allocated for the `in` parameters and local variables is reclaimed: the operation $\lhd$ stands for domain anti-restriction, meaning that locations $\ell_{a_1}, \ldots, \ell_{d_1}, \ldots$ are removed from Σ'. As there is no possibility to take the address of a local variable, there is no risk of dangling pointers.

References

1. McCormick, J., Chapin, P.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
2. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7

3. Maalej, M., Taft, T., Moy, Y.: Safe dynamic memory management in ada and SPARK. In: Casimiro, A., Ferreira, P.M. (eds.) Ada-Europe 2018. LNCS, vol. 10873, pp. 37–52. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92432-8_3
4. Dross, C., Kanig, J.: Recursive data structures in SPARK. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 178–189. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_11
5. Dross, C.: Using pointers in spark (2019). https://blog.adacore.com/using-pointers-in-spark
6. Dross, C.: Pointer based data-structures in spark (2019). https://blog.adacore.com/pointer-based-data-structures-in-spark
7. Jaloyan, G.A.: Internship report: safe pointers in SPARK 2014 (2017). https://arxiv.org/pdf/1710.07047
8. Girard, J.Y.: Linear logic. Theoret. Comput. Sci. **50**(1), 1–101 (1987)
9. AdaLRM: Reference manual for the Ada(R) programming language. ANSI/MIL-STD-1815A-1983 (1983)
10. Baker, H.: 'Use-once' variables and linear objects: storage management, reflection and multi-threading. SIGPLAN Not. **30**(1), 45–52 (1995)
11. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 48–64. ACM, New York (1998)
12. Tschantz, M., Ernst, M.: Javari: adding reference immutability to Java. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 211–230. ACM, New York (2005)
13. Ma, K.K., Foster, J.: Inferring aliasing and encapsulation properties for Java. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp. 423–440. ACM, New York (2007)
14. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, pp. 55–74. IEEE Computer Society (2002)
15. Distefano, D., Parkinson, M.: jStar: towards practical verification for Java. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, pp. 213–226. ACM, New York (2008)
16. Bakst, A., Jhala, R.: Predicate abstraction for linked data structures. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 65–84. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_3
17. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in Cyclone. SIGPLAN Not. **37**(5), 282–293 (2002)
18. Balasubramanian, A., Baranowski, M., Burtsev, A., Panda, A., Rakamarić, Z., Ruzhyk, L.: System programming in rust: beyond safety. SIGOPS Oper. Syst. Rev. **51**(1), 94–99 (2017)
19. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 66:1–66:34 (2018)
20. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20

21. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
22. AdaCore and Altran UK Ltd: SPARK 2014 Reference Manual (2019)
23. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Technical report, ETH Zurich (2018)
24. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. **207**(2), 284–304 (2009)