Safe Dynamic Memory Management in Ada and SPARK

Maroua Maalej[1](✉), Tucker Taft[2], and Yannick Moy[1]

[1] AdaCore, Paris, France
{maalej,moy}@adacore.com
[2] AdaCore, New York, USA
taft@adacore.com

Abstract. Handling memory in a correct and efficient way is a step toward safer, less complex, and higher performing software-intensive systems. However, languages used for critical software development such as Ada, which supports formal verification with its SPARK subset, face challenges regarding any use of pointers due to potential pointer aliasing. In this work, we introduce an extension to the Ada language, and to its SPARK subset, to provide pointer types ("access types" in Ada) that provide provably safe, automatic storage management without any asynchronous garbage collection, and without explicit deallocation by the user. Because the mechanism for these safe pointers relies on strict control of aliasing, it can be used in the SPARK subset for formal verification, including both information flow analysis and proof of safety and correctness properties. In this paper, we present this proposal (which has been submitted for inclusion in the next version of Ada), and explain how we are able to incorporate these pointers into formal analyses.

Keywords: Compilation · Safe pointers · Formal verification
Memory management

1 Introduction

Standard Ada supports safe use of pointers ("access types" in Ada) via strong type checking, but safety is guaranteed only for programs where there is no explicit deallocation of pointed-to objects – explicit deallocation is considered "unchecked" programming in Ada, meaning that the programmer is responsible for ensuring that the deallocation is not performed prematurely. Ada can provide automatic reclamation of the entire memory pool associated with a particular pointer type when the pointer type goes out of scope, but it does not automatically reclaim storage prior to that point. It is possible for a user to implement abstract data types that do some amount of automatic deallocation at the object level, but this requires additional programming, and typically has certain limitations. As part of its strong type checking, Ada also prevents dangling references to objects on the stack or the heap, by providing automatic compile-time checking of "accessibility" levels, which reflect the lifetimes of stack and heap objects.

A. Casimiro and P. M. Ferreira (Eds.): Ada-Europe 2018, LNCS 10873, pp. 37–52, 2018.
https://doi.org/10.1007/978-3-319-92432-8_3

Conversions between pointer types are restricted to ensure pointers never outlive the objects they designate. Values of a pointer type are by default initialized to null to prevent use of uninitialized pointers, and run-time checks verify that a null pointer is never dereferenced.

SPARK is a subset of the Ada programming language, targeted at the most safety- and security-critical applications. SPARK starts with the basic Ada features oriented toward building reliable and long-lived software, then adds restrictions that ensure that the behavior of a SPARK program is unambiguously defined, and simple enough that formal verification tools can perform an automatic assessment of conformance between a program specification and its implementation. The SPARK language and toolset for formal verification have been applied over many years to on-board aircraft systems, air traffic control systems, cryptographic systems, and rail systems [8,9].

As a consequence of our focus in SPARK on proof automation and usability, we have forbidden the use in SPARK of programming language features that either prevent automatic proof, or require extensive user effort in annotating the program. Pointer types are the main example of this for SPARK. SPARK supports many Ada features that can make up for the lack of pointers: by-reference parameter passing, the ability to specify the address of objects, and the support for arrays as first-class objects. On the other hand, pointers are sometimes desirable, which forces one to exclude from formal SPARK analysis the parts of a program that make use of pointers. While there are idioms that facilitate this isolation of pointers in non-SPARK parts of a program [2], it would be desirable to provide some level of support for pointers in SPARK.

In this work, we propose a restricted form of pointers for Ada that is safe enough to be included in the SPARK subset. As our main contribution, we show how to adapt the ideas underlying the safe pointers from permission-based languages like Rust [3] or ParaSail [13], to safely restrict the use of pointers in more traditional imperative languages like Ada. In Sect. 2, we provide rationale for the rules that we propose to include in the next version of Ada, which takes into account specifics of Ada such as by-copy/by-reference parameter passing and exception handling. In Sect. 3, we outline how these rules make it possible to formally verify SPARK programs using such pointers. Finally, we present related work and conclude.

2 A Proposal for Ownership Types in Ada

Pointers (access types) are essential to many complex Ada data structures, but they also have downsides, and can create various safety and security problems. When attempting to prove properties of a program, particularly those with multiple threads of control, the enemy is often the unknown "aliasing" of names introduced by access types and certain uses of (potentially) by-reference parameters. We say that two names may alias if they have the possibility to refer to overlapping memory regions. By *unknown* aliasing of names, we mean the case where two distinct names might refer to the same object, without the compiler being aware of it. A rename introduces an alias, but not an “unknown alias,” because the compiler is fully aware of such an alias. However, if a global variable is passed by reference as a parameter to a subprogram that also has direct access to the same global, the by-reference parameter and the global are now aliases within the subprogram, and the compiler generating its code has no way of knowing this, hence they are “unknown aliases.” One approach is to always assume the worst, but that makes analyses much harder, and in some cases infeasible. Access types also introduce unknown aliasing, and in most cases, an analysis tool will not be sure whether the aliases exist, and will again have to make worst-case assumptions, which again may make any interesting proof infeasible.

The question that emerges in this context is: can we create a subset of access-type functionality that supports the creation of interesting data structures without bringing along the various problems associated with unknown aliasing? The notion of pointer “ownership” has emerged as one way to “tame” pointer problems, while preserving flexibility [12]. The goal is to allow a pattern of use of pointers that avoids dangling references as well as storage leaks, by providing safe, immediate, automatic reclamation of storage rather than relying on unchecked deallocation, while also not having to fall back on the time and space vagaries of garbage collection. As a side benefit, we can also get safer use of pointers in the context of parallelism. We propose the use of pointer ownership (as well as additional rules, detailed in [1], disallowing “aliasing” involving parameters) to provide safe, automatic, parallelism-friendly heap storage management while allowing the flexible construction of pointer-based data structures, such as trees, linked lists, hash tables, etc.

Although we took inspiration from Rust and ParaSail to produce this proposal, it is also different in many ways, due to the different objectives pursued in Ada and SPARK. Firstly, this proposal is designed to work with existing features of Ada such as by-copy/by-reference parameter passing and exception handling: raising an exception should not lead to memory leaks, and upon handling of a raised exception, objects should not be left in an inconsistent state. Secondly, this proposal relies on Ada’s exiting mechanisms for avoiding uninitialized or dangling pointers: uninitialized and freed pointers should be set to null so that dereferencing such pointers results in a run-time error.

By relying on pointer ownership, we can ensure that pointer-based structures in SPARK can be supported while preserving SPARK’s strict anti-aliasing parameter passing rules, thereby allowing the SPARK proof tools to prove the same range of safety and correctness properties, including freedom from data races, even in programs that use pointer-based structures in conjunction with concurrent and parallel programming constructs (see Sect. 3).

2.1 Ownership Types

In this section, we describe the *Ownership* aspect, a Boolean value that can be specified True for an Ada access type, with the effect that the compiler enforces an additional set of rules to ensure that there can be at most one writable access path to the data *designated* (i.e. pointed to) by an object of such an access type, or alternatively one or more read-only access paths via such access objects, but never both concurrently. This is known as the Concurrent-Read-Exclusive-Write (CREW) access policy [11]. The CREW policy prevents multiple access via different objects to the same memory area whenever one of those access objects is modifying that area.

In addition to access types, Ownership can also be specified True for composite types, allowing for a record, an array, or even a private type with potentially multiple components that are access objects with Ownership True. Any object of a type with Ownership True is called an *ownership* object. In addition, we use the more general term *managed object* to refer to any object that is reachable by following ownership access objects. Here is the overall taxonomy:

- *Ownership* objects: objects of a type with Ownership aspect True, including:
  - *Owning access* objects: access-to-variable objects with Ownership aspect True;
  - *Observing access* objects: access-to-constant objects with Ownership aspect True;
  - *Composite ownership* objects: records and arrays with Ownership aspect True;
- Other *managed* objects: non-ownership objects that are pointed to by an owning or observing access object.

In the remainder, we presume that all objects in the code samples we present are managed objects; we refer the reader to [1] for further details on the Ownership aspect specification, and specific rules that apply to non-ownership managed objects. Note also that some of the Ownership rules will be expressed in terms of "names" rather than "objects," since objects do not really exist at compile-time, when these rules are intended to be enforced.

In the next section, we will focus on owning and observing access objects, which as parameters are passed by copy in Ada, before we consider the situation of composite ownership objects, for which we require pass-by-reference. As it turns out, when worrying about the number of ways one can reach an object, passing a parameter by copy or by reference can make a difference, particularly in the context of propagating and handling exceptions.

2.2 Ownership for Access Objects

To ensure safe memory management, the basic rule is that at most one object that gives update access may be used at one time to refer to a designated object, and while such an updater exists, access via any other object is disallowed. There might be multiple access objects that designate the same object (or some part of it) at certain times, but all of them must provide only read access.

The manipulation of ownership access objects in the program is limited to the following three kinds of operations:

- *Moving*: An assignment operation that leads to moving the value of one access object into another, leaving behind a null;

– *Borrowing*: The declaration of a short-term read-write reference by copying an existing access object, borrowing its value for the lifetime of the *borrower*.
– *Observing*: The declaration of a short-term object that gives read-only access, by copying an existing access object, observing the object(s) reachable from it for the lifetime of the *observer*.

Given an access object, it should have one of three possible states at any point in its scope:

– *Unrestricted*: the object may be dereferenced and used to read or update the designated object;
– *Observed*: the object may be used for read-only access to all or part of the designated object, or part of some object directly or indirectly reachable via a chain of owning access objects from the designated object;
– *Borrowed*: the object's ownership has been temporarily transferred to another object, and while in such a state the original access object is not usable for reading or updating the designated object (nor for any other purpose).

Moving Access Values. The *move* operation involves a complete transfer of the ownership from the right hand side to the left hand side in an assignment operation, where both left- and right-hand-side objects are owning access variables in the unrestricted state. After the assignment, the right-hand side gets set to null, and the newly assigned object becomes the (unrestricted) owner. By setting the original access object to null, any name that starts with a dereference of that original access object is effectively "destroyed"; even if an exception is raised before we explicitly assign a new value to the right-hand side, there is no danger a handler for the exception will be able to dereference the old value of the right-hand side. Being able to use the old value to reach the designated object would, at a minimum, violate our CREW policy, and could cause havoc if the designated object had been deallocated after the move, but prior to the exception being raised.

In addition to setting the right-hand side to null, a move also finalizes and deallocates the object, if any, designated by the left-hand side prior to the assignment. This automatic deallocation means memory is reclaimed as soon as it is no longer accessible, thereby preventing memory leaks, without the need for an asynchronous garbage collector. This is safe to do, because a move requires the left-hand side to be in the *unrestricted* state, meaning that it is the only access object pointing to the object about to be deallocated.

In addition to considering certain assignment statements to be *moves*, we also consider the assignments inherent in passing by copy an `out` or `in out` parameter to be moves, as well as returning a value from a function. Updating a subcomponent of a composite object is also considered a move, but we consider these in the section focused on composite types (see below).

Figure 1 illustrates an example of "move" operations. Objects `X` and `Y` are access-to-variable objects of the named type `Int_Ptr`. At a `Swap` procedure call site, the actual parameter `X`, which is required to be in the unrestricted state, is *copied in* to the formal X_Param. The *ownership* of X.all (X.all is Ada's notation for dereferencing X – X.all denotes the object *designated* by X) is similarly moved from X to X_Param. This requires setting X to null until the subprogram returns (to ensure safety in the presence of an exception), at which point the final value of X_Param is *copied back* to X. Variable X then reasserts its ownership over X.all. Similar state transitions apply for Y and Y_Param. At lines ℓ_4, ℓ_6, and ℓ_7, we have additional move operations, which consist of moving, respectively, the objects X_Param, Y_Param, and Tmp. Thanks to our move-related rules, even such a straightforward implementation of the Swap procedure for access types is nevertheless guaranteed to be alias safe while Swap is executing, both from the *caller* perspective and from *inside* Swap itself, since the ownership is transferred as part of each access-to-variable object assignment.

```
1  type Int_Ptr is access integer;
2
3  procedure Swap(X_Param, Y_Param : in out Int_Ptr) is
4    Tmp : Int_Ptr := X_Param;
5  begin
6    X_Param := Y_Param;
7    Y_Param := Tmp;
8  end Swap;
9
10 X : Int_Ptr := new Integer'(7);
11 Y : Int_Ptr := new Integer'(11);
12
13 Swap(X, Y);
```

Fig. 1. Example of moving the ownership of an object.

Borrowing Access Values. We say that an access value has been "borrowed" if that value has been copied into a short-lived (owning) access-to-variable object. A borrowing operation is a *temporary* transfer of the ownership of the said borrowed object until the end of the scope of the borrower. We want the original access object to still designate the same object until the borrower goes away. As a result, while an access object is in the borrowed state, its value may not be changed; furthermore, to preserve our CREW policy, we disallow using or copying it again until the current borrower goes away; in the borrowed state, the original access object is completely "dead" – it cannot be read nor be the target of an assignment. Furthermore, borrowing applies recursively down the tree rooted at the original access object, meaning that at the point where a name is borrowed, every name with that name as a prefix, is similarly borrowed.

The assignment operations that are considered borrowing are those that *initialize* a stand-alone object of an *anonymous* access-to-variable type, or a constant or an in parameter of a (named or anonymous) access-to-variable type. We also consider as borrowing passing an object of a composite ownership type as a parameter of mode out or in out – see Sect. 2.3 below. The code snippet of Fig. 2 is a simple example of borrowing. X and Y are both access-to-variable objects. We want to swap the objects *designated* by the two pointers (their "contents") using the `Swap_Contents` procedure. To that end, we declare `X_Param` and `Y_Param` as formal parameters of mode `in`. Objects `X` and `Y` become borrowed in the caller, and inside `Swap_Contents` `X_Param` and `Y_Param` are the borrowers, in the unrestricted state. This state allows reading and updating via these formal parameters, which enables swapping the value of their designated objects. Note that we allow an `in` parameter or a constant of an owning access type to provide read/write access to its designated object to accommodate existing Ada practice in the use of such "constant" access-to-variable values to nevertheless update their designated objects.

```ada
type Int_Ptr is access integer;

procedure Swap_Contents (X_Param, Y_Param : in Int_Ptr) is
  Tmp : integer := X_Param.all;
begin
  X_Param.all := Y_Param.all;
  Y_Param.all := Tmp;
end Swap_Contents;

X : Int_Ptr := new Integer'(13);
Y : Int_Ptr := new Integer'(17);

Swap_Contents(X, Y);
```

Fig. 2. Example of borrowing via `in` parameters.

Observing Access Values. We say an access-to-variable object is "observed" when its value has been copied into an "observer," and both the original access object and the copy, starting at that point, can only be used for read access to the designated object. The original object remains in the observed state until the end of the scope of the observer. While being observed, neither the observed object nor the observer is allowed to be moved or borrowed. The original access object cannot be used as the target of an assignment since we need the observed object to continue to designate the same object as long as any observers exist. As with borrowing, observing applies recursively down the tree rooted at the original access object, meaning that at the point where a name is observed, every name with that name as a prefix, is similarly observed.

We consider as observing the assignment operations used to *initialize* standalone objects of an anonymous access-to-*constant* type, as well as `in` parameters of such a type. In the code snippet of Fig. 3, `X_Param` and `Y_Param` are access-to-constant objects of an anonymous type. Since the assignment of the value of `X` to `X_Param` as well as to `Y_Param` are part of the initialization of the target objects, this initiates the observing, and while `X_Param` and `Y_Param` exist they provide read-only access. Note that this allows us to call the function `Sum` using `X` as a first and second parameter – upon the first occurrence of `X` it enters the observed state, but we can still observe it further.

```
type Int_Ptr is access integer;

function Sum (X_Param, Y_Param : access constant Integer) return
    Integer is
begin
   return X_Param.all + Y_Param.all;
end Sum;

X : Int_Ptr := new Integer'(42);

Y : constant Integer := Sum (X, X);
```

Fig. 3. Example of observing via access-to-constant parameters.

Preventing Read-Write Aliasing. We have seen that the observing rules allow multiple access objects to observe the same designated object. In the scope of these objects, the original object is in the observed state; its designated object cannot be written, so there is no read-write aliasing problem here.

We have seen that after borrowing an object, its name allows neither reading nor updating until the borrowing ends. For example, this prevents a call to `Swap_Contents(X, X)`, as borrowing X via parameter `X_Param` makes it illegal to borrow it again via parameter `Y_Param`. The actual order of evaluation does not matter here, as any other order would also be illegal.

We have also seen that after moving an object, its value is set to null, which prevents accessing the designated object again through the original name. This rule by itself does not prevent a call to `Swap(X, X)`, but moving `X` into parameter `X_Param` makes `X` null, so that if it is then moved into parameter `Y_Param`, the value null will be passed, ensuring that a run-time check will prevent read-write aliasing. In fact, in current Ada, passing the same object twice in the same call as an `out` or `in out` parameter is illegal, so this existing Ada rule will catch simple cases such as this at compile time. Furthermore, as part of our proposed extension to Ada, an additional restriction `No_Parameter_Aliasing` is defined, which prevents at compile time the more complex cases as well. We refer the reader to [1] for further details on the `No_Parameter_Aliasing` restriction.

2.3 Extension to Composite Types

The rules presented previously for access objects are extended in natural ways to composite ownership objects (records or arrays with owning access objects as subcomponents) to enforce the Concurrent-Reads-Exclusive-Write principle.

Moving Composite Values. As with access objects, the composite move operation is a complete transfer of the ownership from the right hand side composite object to the left hand side object as part of an assignment operation. And as with access objects, a composite object to be moved must be in the unrestricted state before the assignment. The rules that apply for moving an access object are applied here to each access subcomponent of the composite type: access subcomponents of the moved objects are set to null after being copied, and to avoid memory leaks, if the prior value of the subcomponent in the target composite object is different from the new value, the object designated by this prior value is finalized and its storage deallocated.

As before, we consider as a move each assignment operation for a composite ownership type where the target is a variable (or the "return object" of a function), but this time we do not consider passing of `out` or `in out` parameters to be moves, because for composite ownership objects, parameters are passed by reference and no true copying is occurring. Composite parameter passing is described further below. In the code snippet of Fig. 4, `Rec` is a record with components of an owning access type. The move operation occurs at line ℓ_9 where `R` is moved to `S`, which involves moving `R.X` into `S.X` and moving `R.Y` into `S.Y`. As a result, the objects originally designated by `S.X` and `S.Y` are deallocated and `R.X` and `R.Y` end up null after the assignment.

```
1 type Int_Ptr is access Integer;
2 type Rec is record
3   X, Y : Int_Ptr;
4 end record;
5
6 R : Rec := (...);
7 S : Rec := (...);
8
9 S := R;
```

Fig. 4. Example of moving a composite object.

Borrowing Composite Values. Borrowing composite ownership objects occurs when passing such an object as an `out` or `in out` parameter, consistent with these composite ownership objects being passed by reference. Note how this differs from `out` or `in out` parameters of an access type, which are passed by copy and are thus considered as being *moved* as part of parameter passing. Ada normally allows composite objects to be passed either by copy or by reference, but for ownership composite types, we specify that they must always be passed by reference, to avoid having two different sets of rules for composite objects that would depend on whether the type is passed by copy or by reference.

In the code snippet of Fig. 5, procedure `Swap_Rec` has an `in out` formal parameter `R` of a record type. At the point of call to `Swap_Rec`, the actual parameter name `R1` becomes borrowed until returning from `Swap_Rec`, with the borrower being the formal parameter name `R`. Inside `Swap_Rec`, the formal parameter `R` is initially in the unrestricted state, hence its components `R.X` and `R.Y` can be successively moved in and out through the call to `Swap`, and then borrowed through the call to `Swap_Contents`. Note that subcomponents of a composite type can be individually moved and borrowed, without impacting the state of other non-overlapping subcomponents of the same composite object. We refer the reader to [1] for further details.

```
procedure Swap_Rec (R : in out Rec) is -- R1 is borrowed
begin
   Swap (R.X, R.Y);
   Swap_Contents (R.X, R.Y);
end Swap_Rec;

R1 : Rec := (...);

Swap_Rec (R1);
```

Fig. 5. Example of borrowing via a composite in out parameter.

Observing Composite Values. Observing composite ownership objects occurs when passing such an object as an in parameter, or initializing a stand-alone constant object of such a type.

In the code snippet of Fig. 6, procedure Sum_Rec has an in formal parameter R of a record type. At the point of call to Sum_Rec, the actual parameter name R1 becomes observed, with the formal parameter R as the observer, until returning from Sum_Rec. Inside Sum_Rec, the formal parameter R is initially in an observed state, hence its components R.X and R.Y can only be read (observed) through the call to Sum.

```
function Sum_Rec (R : in Rec) return Integer is
begin
   return Sum (R.X, R.Y);
end Sum_Rec;

R1 : Rec := (...);

Y : Integer := Sum_Rec (R1);
```

Fig. 6. Example of read only access to an object of a composite type.

Traversing Data Structures with Local Variables. In the rules for borrowing access values (Sect. 2.2), initializing a stand-alone object of an anonymous access-to-variable type corresponds to *borrowing* the access object being copied. Similarly, in the rules for observing access values (Sect. 2.2), initializing a stand-alone object of an anonymous access-to-constant type corresponds to *observing* the object being copied. Without these special cases, such initializations might be treated as moves, which would not allow for a non-destructive traversal of a recursive data structure, since every assignment to such a "handle" would deallocate its prior designated object and set to null the object that was moved. Hence, such an object of an anonymous access type acts as a kind of short-term "handle" on the tree of objects rooted at the original access object.

We also allow certain kinds of updates to such "handles," in order to allow traversing the data structure by changing where the handle points. In the borrowing case (for an access-to-*variable* object), we allow the borrower to be updated to point to an object within the tree rooted at the prior value of the borrower;

this is not considered a new borrowing action, as the existing borrower remains the only object providing any read or write access to the subtree rooted at its original value. By limiting the initial borrowing to the initialization of a new stand-alone object, we ensure that borrowing lasts only as long as the lifetime of the "handle." If we allowed any given assignment statement to initiate a new borrowing action, tracking when such borrowing would end might require complex data-flow analysis, potentially across conditional and iterative paths in the program. Somewhat less stringent restrictions are applied when updating an observer – the observer may be updated to point to an already observed object with a compatible scope. Again, doing otherwise might require complex data-flow analysis to determine the extent of the observing action.

In the code snippet of Fig. 7, local variable `Walker` is a stand-alone object of an anonymous access-to-*constant* type, which allows traversing the input binary search tree, so as to find the maximal value (obtained by searching for the rightmost leaf of the tree). After initializing `Walker` with the value of parameter `T`, `T` becomes observed, and `Walker` starts in the observed state (thus preventing updates to `T.all` through `Walker`). The data structure traversal is performed by the instruction of line ℓ_{16}.

```
 1 type Rec;
 2 type Tree is access Rec;
 3 type Rec is record
 4   Data : Natural;
 5   Left, Right : Tree;
 6 end record;
 7
 8 function Max (T : in Tree) return Integer is
 9   Walker : access constant Rec := T; -- Walker observes T
10   Max_Value : Natural := 0;
11 begin
12   while Walker /= null loop
13     if Walker.Data > Max_Value then
14       Max_Value := Walker.Data;
15     end if;
16     Walker := Walker.Right; -- assignment to Walker
17   end loop;
18 return Max_Value;
19 end Max;
```

Fig. 7. Example of traversing a data structure with read-only access: Max on a Binary Search Tree.

In the code snippet of Fig. 8, local variable `Walker` is a stand-alone object of an anonymous access-to-*variable* type, which allows traversing the input binary search tree to insert the input value `V` at the correct leaf position (obtained by searching for the branch where this value would be stored, if it were already present). After initializing `Walker` with a copy of the value of parameter `T`, `T` becomes borrowed, and `Walker` starts its life in the unrestricted state (thus allowing updates via `Walker` to the tree pointed to by `T`). The data structure traversal is performed by the instruction of line ℓ_8 and ℓ_{15}. Insertion in the tree is performed at lines ℓ_{10} and ℓ_{17}.

```
 1 procedure Insert (T : in Tree; V : Natural) is
 2   Walker : access Rec := T;
 3 begin
 4   loop
 5     if V < Walker.Data then
 6       if Walker.Left /= null then
 7         Walker := Walker.Left;
 8       else
 9         Walker.Left := Build_Leaf(V);
10         exit;
11       end if;
12     elsif V > Walker.Data then
13       if Walker.Right /= null then
14         Walker := Walker.Right;
15       else
16         Walker.Right := Build_Leaf(V);
17         exit;
18       end if;
19     end if;
20   end loop;
21 end Insert;
```

Fig. 8. Example of traversing a data structure with read and update access: Insert into a Binary Search Tree. `Build_Leaf(V)` creates a node with `Data` = `V`, and `Left`, and `Right` components both null.

3 Formal Verification with Ownership Types in SPARK

The existing SPARK restrictions imposed on its current subset of Ada ensure that an assignment to one variable cannot change the value of some other visible variable. This property is essential to allow sound modular static analysis, where each subprogram can be analyzed independently while detecting all possible violations of the kinds targeted by the analysis.

This is currently enforced by forbidding all use of access types in SPARK, and by restricting aliasing between parameters and global variables so that only benign aliasing is permitted (i.e. aliasing that does not cause interference). The aliasing restrictions are as follows:

– Two output parameters should never be aliased.
– An input and an output parameter should not be aliased, unless the input parameter is always passed by copy.
– An output parameter should never be aliased with a global variable referenced by the subprogram.
– An input parameter should not be aliased with a global variable updated by the subprogram, unless the input parameter is always passed by copy.

To understand why aliasing matters in SPARK, consider procedure `Add_One` in Fig. 9. If `X_Param` and `Y_Param` are not aliased, then the result of calling `Add_One` on actual parameters `X` and `Y` will increase their contents by one. If `X` and `Y` are aliased, then calling `Add_One` on `X` and `Y` will increment the underlying content by two.

```
1 procedure Add_One(X_Param, Y_Param : in Int_Ptr) is
2 begin
3    X_Param.all := X_Param.all + 1;
4    Y_Param.all := Y_Param.all + 1;
5 end Add_One;
```

Fig. 9. A simple procedure where aliasing would create problems in SPARK.

If SPARK ignored aliasing, it would conclude that procedure `Add_One` always increments by exactly one the content of each of its parameters `X_Param` and `Y_Param`. In particular, it could prove the following postcondition on the procedure.

```
1 procedure Add_One(X_Param, Y_Param : in Int_Ptr) with
2    Post => X_Param.all = X_Param.all'Old + 1
3      and Y_Param.all = Y_Param.all'Old + 1;
```

Indeed, by presuming that the assignment to `X_Param.all` on line ℓ_2 does not influence the value of `Y_Param.all`, proof would be able to derive that the values `Y_Param.all` has been incremented by 1. Similarly, flow analysis could derive wrong data dependencies if possible aliasing is not taken into account.

This wrong postcondition would allow a proof that an incorrect assertion is satisfied in the code snippet of Fig. 10, while in fact it fails at run time. Thus, normal Ada pointers could not be treated like any other component in SPARK, given the possibility for aliasing. But the rules we have described for ownership objects precisely prevent aliasing when one of the objects can be written. This is analogous to the rules in SPARK for preventing aliasing between by-reference parameters, and these rules allow SPARK to treat such pointers like other components.

```
1   X : Int_Ptr := new Integer'(1);
2   (...)
3
4   Add_One (X, X);
5   pragma Assert (Y.all = 2);  --  incorrect assertion
```

Fig. 10. Example of proof of an incorrect assertion due to the presence of aliasing in SPARK.

In the case of `Add_One`, this means that SPARK analysis will be able to conclude that the postcondition above is satisfied by the implementation of `Add_One`.

But unsafe calls such as `Add_One(X, X)` will be rejected both by compilation and analysis.

The SPARK tools also provide detection of potential data races in programs that use concurrent and parallel programming constructs. This detection depends on the strict anti-aliasing conditions on parameters, and provides a sound assurance that no two threads concurrently manipulate the same data, if either has update access. This matches the CREW condition imposed on access objects through the proposed ownership rules, and means that the SPARK tools can handle pointer-based structures that obey these rules, in the same way it already handles record- and array-based structures, enabling provably safe concurrent and parallel programming in SPARK even when enhanced with this more flexible data structuring capability.

4 Related Work

C-like languages are mostly based on pointers and often sacrifice safety for performance purposes. To overcome safety shortcoming and manage the storage of a pointer, C++ introduces the notion of *unique pointers*. An object defined as a `unique_ptr` has the ability to take ownership of an object. It becomes responsible for its deletion at some point. Although these rules help provide greater language safety, the unique pointer concept is limited because it prohibits pointer arithmetic and copy assignments.

Separation logic [10] is an extension of Hoare-Floyd logic that allows reasoning about pointers. In general, it is not well integrated with deductive verification, and, in particular, is not supported by most SMT provers.

Dafny associates each object with its *dynamic frame*, the set of pointers that it owns [7]. 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 [4].

The inspiration for much of our work springs from the systems programming languages Cyclone [5], Rust [3], and ParaSail [13], which achieve absence of harmful aliasing by enforcing an Ownership type system on the memory pointed to by objects. Rust and ParaSail are recent programming languages providing safe systems programming, with a focus on memory safety for concurrent programs. Rust and ParaSail also deal with the lifetime of allocated memory, while preventing dangling pointer references.

The most closely related work to ours springs from Jaloyan *et al.* [6] anti-aliasing rules. In [6], access-to-variable objects and composite objects with access subcomponent objects are considered as *deep* variables and their ownership states are transferred in the same way when used to call subprograms. Actual deep parameters are considered as borrowed and the durations of borrows are only limited to the duration of procedure calls. It turns out that their rules do not allow traversing a linked data structure with read/write permission, or even traversing with read-only permission. In our work, the distinction between stand-alone access objects and composite ones and moving or observing composite objects instead of borrowing them has allowed us to support safely traversing a data structure for read or update.

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. Our approach does not require additional user annotations required in some of the previously mentioned techniques. We instead rely on the existing distinctions in Ada between `in` and `in out` parameters, and between access-to-variables and access-to-constants. We thus achieve high automation and usability, which was one of our goals in supporting pointers in SPARK.

5 Conclusion

We have presented an extension to the Ada language to provide pointer types ("access types" in Ada) that provide provably safe, automatic storage management without any asynchronous garbage collection, and without explicit deallocation by the user. Although we took inspiration from Rust and ParaSail, the extension we propose differs so as to work well with existing features of Ada such as by-copy/by-reference parameter passing and exception handling, and because we rely on the existing mechanisms in Ada for preventing access to uninitialized pointers and freed memory.

This extension relies on the notion of ownership, where only one access object can provide update access to the designated object at any given time. Ownership of a designated object can be *moved* to another object through assignment, which deallocates the object previously designated by the target of the assignment and leaves the source of the assignment null. Ownership can also be *borrowed* giving a short-term borrower read-write access to the designated object. Finally, the value of a designated object can be *observed* by multiple read-only observers, with limited lifetimes. Collectively, these mechanisms enforce a principle of Concurrent-Reads-Exclusive-Write.

Because the mechanism for these safe pointers relies on a strict control of aliasing, they can be used in the SPARK subset for formal verification, which includes both analysis of flows and proof of properties, including in the presence of multiple threads of control.

This proposal has been formalized as Ada Issue [1] for inclusion in a future version of Ada. We have also implemented a prototype of these permission rules in the GNAT/GCC compiler for Ada developed at AdaCore. Our implementation successfully proves the safety of all programs presented in this article.

Acknowledgements. We thank the anonymous reviewers for their remarks, and Georges-Axel Jaloyan for his initial work on the design, formalization and implementation of these ownership rules for Ada and SPARK.

References

1. AdaCore: Access value ownership and parameter aliasing (2018). http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0240-1.txt
2. AdaCore, Thales: Implementation guidance for the adoption of SPARK (2017). https://www.adacore.com/books/implementation-guidance-spark
3. Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamaric, Z., Ryzhyk, L.: System programming in rust: Beyond safety. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, 8–10 May 2017, pp. 156–161 (2017)
4. 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
5. Grossman, D., Morrisett, J.G., Jim, T., Hicks, M.W., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, 17–19 June 2002, pp. 282–293 (2002)
6. Jaloyan, G.A., Moy, Y., Paskevich, A.: Borrowing safe pointers from rust in spark. In: International Conference on Computer-Aided Verification - 29th International Conference, Heidelberg, Germany (2018, in submission)
7. 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
8. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
9. O'Neill, I.: SPARK - a language and tool-set for high-integrity software development. In: Industrial Use of Formal Methods: Formal Verification (2012)
10. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of th 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, pp. 55–74 (2002)
11. Sant, P.M.: Concurrent read, exclusive write. In: Pieterse, V., Black, P.E. (eds.) Dictionary of Algorithms and Darta Structures (2004). https://www.nist.gov/dads/HTML/concurrentReadExcluWrt.html
12. Svoboda, D., Wrage, L.: Pointer ownership model. In: Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS 2014), 6–9 Jan 2014, Waikoloa, Hawaii, pp. 5090–5099 (2014)
13. Taft, S.T.: Multicore programming in ParaSail. In: Romanovsky, A., Vardanega, T. (eds.) Ada-Europe 2011. LNCS, vol. 6652, pp. 196–200. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21338-0_16