Safe Dynamic Memory Management in Ada and SPARK

Maroua Maalej, Tucker Taft, Yannick Moy

AdaCore

Ada-Europe
June 19, 2018

Why Try To Verify Use of Pointers?

AdaCore The GNAT Pro Company

- Automatic storage management
- Control “unknown” aliasing of names
- Use pointers in SPARK for formal verification

How?

- Implement a variant of pointer *Ownership*
  - ↦ Inspired from Rust
  - ↦ Concurrent-Read-Exclusive-Write (CREW) policy

Quick Reminder 1/2

- Named types:

```
type Int_Ptr is access Integer;
X : Int_Ptr;
```

- Anonymous types:

```
Y : access Integer;
```

- General access types

```
type Int_Cst_Ptr is access constant Integer;
type Int_Cst_Ptr is access all Integer;
```

- Pool-specific access types

  ⇝ No general_access_modifier appears

Quick Reminder 2/2

- Access types

```
X : Int_Ptr;
```

- Composite types

```
type Rec is record
  Data : Int_Ptr;
end record
```

- By-copy types

  Parameter passed by copy

- By-reference types

  Parameter passed by reference: a view on the actual parameter

Motivating Example: Swap Pointers

AdaCore
The GNAT Pro Company

```
type Int_Ptr is access Integer;

procedure Swap(X_Param, Y_Param : in out Int_Ptr) is
  Tmp : Int_Ptr := X_Param;
begin
  X_Param := Y_Param;
  Y_Param := Tmp;
end Swap;

X : Int_Ptr := new Integer;
Y : Int_Ptr := new Integer;

Swap(X, Y);
```

Dangling refs?

Storage leaks?

Correct result?

Pointer Ownership: Overview

AdaCore The GNAT Pro Company

Idea: No more than one “owning” pointer to a given object

Constraints:

- Composite types are by-reference types
  - ⇝ Always passed by reference
- Access types are pool-specific types
  - ⇝ Cannot point to stack

Goal: Automatic storage management

Pointer Ownership: Overview

AdaCore The GNAT Pro Company

Idea: No more than one “owning” pointer to a given object

Constraints:

- Composite types are by-reference types
  - ⇝ Always passed by reference
- Access types are pool-specific types
  - ⇝ Cannot point to stack

Goal: Automatic storage management

Operations

- Move → complete transfer of the ownership
- Borrow → temporary transfer of the ownership
- Observe → no owning object

Pointer Ownership: Overview

Idea: No more than one "owning" pointer to a given object

Constraints:

- Composite types are by-reference types
  - $\rightsquigarrow$ Always passed by reference
- Access types are pool-specific types
  - $\rightsquigarrow$ Cannot point to stack

Goal: Automatic storage management

Operations

- Move
- Borrow
- Observe

Objects states

- Unrestricted $\Rightarrow$ Read Write
- Observed $\Rightarrow$ Read Only
- Borrowed $\Rightarrow$ No

Contents

Moving Operations

Moving Access Values

When: Assignment to **named**

- variables or return objects
- parameters of mode out/in-out

Example:

```
Y : Int_Ptr;
X : Int_Ptr := Y;
```

Conditions

- X, Y of named type
- X, Y unrestricted

Results

- X unrestricted
- Old storage of X deallocated
- Y unrestricted, **null**

Moving Composite Types

When: Assignment to

- variables or return objects
- ~~parameters of mode out/in-out~~

Example:

```
R : Rec := (...);
S : Rec := (...);
S := R;
```

Conditions

- R, S unrestricted

Results

- S unrestricted
- Old S components deallocated
- R unrestricted; components **null**

Borrowing Operations

Borrowing Access Values

AdaCore
The GNAT Pro Company

When: Initializing

- `in` parameters
- stand-alone **anonymous** objects
- constants

of an access-to-**variable** type

Borrowing Access Values

When: Initializing

- `in` parameters
- stand-alone **anonymous** objects
- constants

of an access-to-**variable** type

Example:

```
procedure f(X_Param : in Int_Ptr);
f(X);
```

Conditions

- `X_Param` of mode `in`; access-to-variable type
- `X` unrestricted

Results

- `X_Param` unrestricted
- `X` borrowed

Borrowing Access Values

When: Initializing

- `in` parameters
- **anonymous** stand-alone objects
- constants

of an access-to-**variable** type

Example:

```
X : access Integer := Y
```

Conditions

- X of an anonymous access-to-variable type
- Y unrestricted

Results

- X unrestricted
- Y borrowed

Borrowing Access Values

When: Initializing

- `in` parameters
- **anonymous** stand-alone objects
- constants

of an access-to-**variable** type

Example:

```
X : access Integer := Y
```

Conditions

- `X` of an anonymous access-to-variable type
- `Y` unrestricted

Results

- `X` unrestricted
- `Y` borrowed

Borrowing Composite Objects

When: Passing parameters of mode out or in-out

Example:

```
procedure f(X_Param : in out Rec);
f(X);
```

Conditions

- X_Param of mode in-out
- X passed by-reference

Results

- X_Param unrestricted
- X borrowed

Observing Operations

Observing Access Values

When: Initializing

- `in` parameters
- **anonymous** stand-alone objects

of an access-to-**constant** type

Example:

```
X : access constant Integer := Y;
```

Conditions

- `X` of an anonymous access-to-constant type
- `Y` unrestricted or observed

Results

- `X` observed → Read Only
- `Y` observed → Read Only

Observing Access Values

When: Initializing

- in parameters
- **anonymous** stand-alone objects

of an access-to-**constant** type

Example:

```
X : access constant Integer := Y;
```

Conditions

- X of an anonymous access-to-constant type
- Y unrestricted or observed

Results

- X observed → Read Only
- Y observed → Read Only

Observing Composite Types

When: Initializing

- **constant** stand-alone objects
- parameters of mode `in`

Example:

```
procedure f(X_Param : in Rec);
f(X);
```

Conditions

- `X_Param` of mode `in`
- `X` passed by-reference

Results

- `X_Param` observed → RO
- `X` observed → RO

Observing Composite Types

AdaCore
The GNAT Pro Company

When: Initializing

- **constant** stand-alone objects
- parameters of mode `in`

Example:

```
procedure f(X_Param : in Rec);
f(X);
```

Conditions

- `X_Param` of mode `in`
- `X` passed by-reference

Results

- `X_Param` observed → RO
- `X` observed → RO

Example Cont'd: Swap Pointers

```
type Int_Ptr is access Integer;

procedure Swap(X_Param, Y_Param : in out Int_Ptr) is
   Tmp : Int_Ptr := X_Param ;
begin
   X_Param := Y_Param ;
   Y_Param := Tmp ;
end Swap;

X : Int_Ptr := new Integer;
Y : Int_Ptr := new Integer;

Swap( X , Y );
```

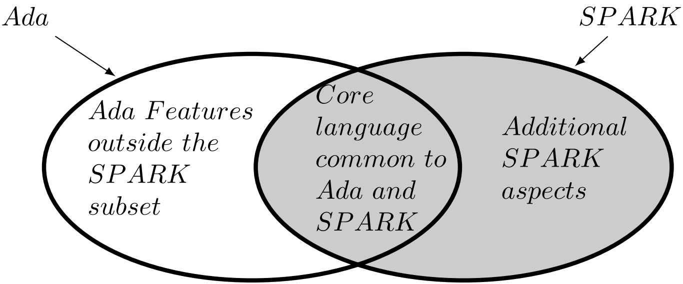
Tmp is the new owning object

⇝ No dangling reference, cannot dereference the old value of X_Param

Formal Verification in SPARK

SPARK - What it is?

- A programming language
  - A subset of Ada, designed for static verification
  - Additional features to enhance program specification



- A set of program verification tools

Why Aliasing Matters in SPARK?

AdaCore
The GNAT Pro Company

```
type Int_Ptr is access Integer;

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

Why Aliasing Matters in SPARK?

```
type Int_Ptr is access Integer;

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

If SPARK ignored aliasing:

```
X : Int_Ptr := new Integer'(3);
 (...)

Add_One (X, X);
pragma Assert (X.all = 4); -- incorrect assertion
```

With Ownership Types: Alias

AdaCore
The GNAT Pro Company

spark_proof.adb

```ada
1  procedure Spark_Proof with SPARK_Mode => On is
2
3     type Int_Ptr is access Integer;
4
5     procedure Add_One(X_Param, Y_Param : Int_Ptr) with
6       Post => X_Param.all = X_Param.all'Old + 1
7       and Y_Param.all = Y_Param.all'Old +1
8     is
9     begin
10       X_Param.all := X_Param.all + 1;
11       Y_Param.all := Y_Param.all + 1;
12    end Add_One;
13
14    X : Int_Ptr := new Integer'(3);
15    Y : Int_Ptr := new Integer'(4);
16 begin
17
18    Add_One(X, X);
19    pragma Assert (Y.all = 5);
20
21 end Spark_Proof;
```

Spark_Proof.Add_One

Messages Locations Breakpoints /bin/bash

```
gnatprove -P/home/mmaalej/tests/spark/pointers/default.gpr -j0 --mode=flow --ide-progress-bar -u spark_proof.adb
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
spark_proof.adb:18:15: insufficient permission for "X"
spark_proof.adb:18:15: expected state "unrestricted" at least, got "borrowed"
gnatprove: error during analysis of data and information flow
[2018-06-08 16:06:19] process exited with status 1, elapsed time: 00.81s
```

With Ownership Types: Alias

```
spark_proof.adb:18:15: insufficient permission for "X"
spark_proof.adb:18:15: expected state "unrestricted" at least, got "borrowed"
```

With Ownership Types: Alias Free

AdaCore The GNAT Pro Company

spark_proof.adb

```
procedure Spark_Proof with SPARK_Mode => On is

   type Int_Ptr is access Integer;

   procedure Add_One(X_Param, Y_Param : Int_Ptr) with
     Post => X_Param.all = X_Param.all'Old + 1
     and Y_Param.all = Y_Param.all'Old +1
   is
   begin
      X_Param.all := X_Param.all + 1;
      Y_Param.all := Y_Param.all + 1;
   end Add_One;

   X : Int_Ptr := new Integer'(3);
   Y : Int_Ptr := new Integer'(4);
begin

   Add_One(X, Y);
   pragma Assert (Y.all = 5);

end Spark_Proof;
```

Spark_Proof.Add_One

Messages Locations Breakpoints

```
Builder results (7 of 18 items in 1 file)
  spark_proof.adb (7 of 18 items)
    6:14   info: postcondition proved
    6:44   info: overflow check proved
    7:40   info: overflow check proved
    18:12  info: initialization of "X" proved
    18:15  info: initialization of "Y" proved
    19:19  info: initialization of "Y" proved
    19:19  info: assertion proved
```

Conclusion

Conclusion

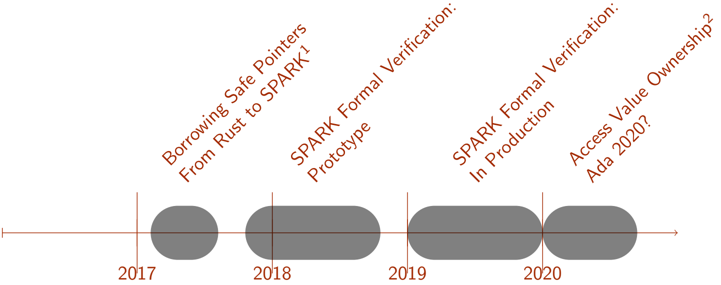
Pointer Ownership Approach

- Inspired from Rust
- Safe pointers w.r.t to CREW policy: full ownership (read/write access); partial ownership (read-only access)

Pointer Ownership Goals

- For Ada
  - No storage leaks
  - No dangling references
- For SPARK
  - No hidden aliasing $\mapsto$ Can verify correctness of algorithms

Supporting Pointers in SPARK: Steps



[1] Georges-Axel Jaloyan, Yannick Moy, and Andrei Paskevich. Borrowing Safe Pointers From Rust in SPARK. 2017. URL: https://arxiv.org/abs/1805.05576.

[2] AdaCore. Access value ownership and parameter aliasing. 2018. URL: http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0240-1.txt.

Questions?