Partial Support for Access Types in SPARK

Access Types in SPARK

- Pointers are called access types in Ada
- Dereferences are done using .all

```
type Int_Access is access Integer;
X : Int_Access := new Integer'(10);
pragma Assert (X.all = 10);
```

- We only support pointing to the heap in SPARK

```
❌ type Int_Access is access all Integer;
   Y : Integer := 10;
❌ X : Int_Access := Y'Address;
```

Access Types in SPARK – Ownership Rules

- Enforce single writer / multiple readers principle
- Rules enforced in the compiler frontend part of GNATprove

```
X : Int_Access := new Integer'(10);
Y : Int_Access := X;
V : Integer := X.all;
```

- The rules still allow interesting programs

```
procedure Swap (X, Y : in out Int_Access) is
   T : Int_Access := X; -- ownership of X transferred to T
begin
   X := Y;              -- ownership of Y transferred to X
   Y := T;              -- ownership of T transferred to Y
end Swap;
```

Access Types in SPARK – Translation to Why

- Access types are translated as regular types (copied on assignment)
- Normal VC-gen is only valid because of ownership rules

```
Y := X;
Y.all := 11;
pragma Assert (X.all = 10);
```

- Allows to verify simple programs using pointers

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

Future Enhancements

- Already in the SPARK language manual
- But not supported by the toolset yet
- Some are challenges

1. Check absence of memory leaks
2. Support recursive data structures
3. Support statically known aliases (aka. Local borrowers)
4. Quantification over recursive data structures

1. Check Absence of Memory Leaks

- Can take advantage of single ownership to check for memory leaks
- Access objects need to be moved or freed before being overritten / going out of scope

```
declare
  X : Int_Access := new Integer’(10);
  Y : Int_Access := new Integer’(10);
begin
  Y := X;      --  Memory leak, Y’s content is lost
end;           --  Memory leak, X’s content is lost
```

1. Check Absence of Memory Leaks

- Need to know when something is *erased* (goes out of scope/ is overridden)

```
X : My_Rec := (F => new Integer'(10));
declare
  Y : constant My_Rec := X;   -- Observe X.F
begin
  ...
end;                          -- Y.F does not go out of scope
```

- Checks can be done in flow analysis / frontend when easy
- Have to use proof on more complex cases

1. Check Absence of Memory Leaks

Check for memory leaks in proof:

- Set accesses to null when moved in Why

```
X : R := (F1 => 1,
          F2 => new Integer'(10),
          F3 => (G => new Integer'(10)));
Y : R := X;        -- nullify X.F2 and X.F3.G
```

- Check for nullity when values are erased

```
X.F3 := Y.F3;      -- check that X.F3.G is null
```

- Nullification not visible from regular semantics

2. Support Recursive Data Structures

- Ada record types cannot be directly recursive
- Access types allow to construct recursive types

```
type List_Cell;
type List is access List_Cell;
type List_Cell is record
  Next : List;
end record;
```

- Could be traversed using recursive calls

```
function Length (L : List) return Natural is
  (if L = null then 0 else 1 + Length (L.Next));
```

2. Support Recursive Data Structures

- Can be supported in Why using an abstract type

```
type closed_list
type list_cell = { next : closed_list }
type list =
    { is_null : bool; value : list_cell; address : int }
```

- Along with conversion functions

```
function open (l : closed_list) : list
function close (l : list) : closed_list
axiom open_close:
    forall l : list. open (close l) = l
```

3. Statically known aliases (aka. Local borrowers)

- SPARK RM allows local borrowers of (recursive) data structures

```
X : List := new List_Cell'(Next => ...);
declare
   Y : access List_Cell := X;   --  value of X is not moved
begin
   ...                          --  modify Y
end;                            --  ownership goes back to X
```

- Aliases are known statically

```
Z : access T := (if Use_X then X else Y);
```

3. Statically known aliases (aka. Local borrowers)

- Borrowers can reference arbitrarily deep parts of the object

```
  Y : access List_Cell := X;
begin
  if Y.Val /= 0 then
    Y := Y.Next;   --  The exact position of Y in X is not
                   --  known statically.
  end if;
```

- Can also call (traversal) functions to initialize a borrower

```
function Find (X : List; V : Integer) return access List_Cell;
Y : access List_Cell := Find (X, 0);
```

3. Statically known aliases (aka. Local borrowers)

- Local borrowers can be used to modify the underlying object

```
declare
  Y : access List_Cell := Find (X, 0);
begin
  Y.Val := 1;
end;
--  X has been modified
```

- Idea: translate local borrowers as a path in the underlying object
- Modify the underlying object instead of modifying the borrower

3. Statically known aliases (aka. Local borrowers)

First attempt: Use sequences of directions for paths

```
X : Tree := ...;
Y : access Tree_Cell := X.all.Left.all.Left.all.Right;
--  Y is statically known to refer to a part of X
--  Y is translated as the path (Left, Left, Right)
```

But proof will be difficult:

- Inductive reasoning over paths
- Quantification over complex types

3. Statically known aliases (aka. Local borrowers)

Second attempt: Use the address of the local borrower

- Reachability is axiomatized
- Get queries the structure at an arbitrary position
- Set modifies the structure at arbitrary position

```
let all_to_zero (l : ref list) =
    (if !l.address = 0 then return);
    let w = ref !l.address in
    while !w <> 0 do
     invariant { !w = 0 \/ valid !l !w }
     let new_val = {(get !l !w) with value = {(get !l !w).value with content = 0 }} in
       l := set !l !w new_val;
       w := (open (get !l !w).value.next).address
    done
```

4. Quantification over Recursive Data Structures

- Quantification in Ada is bounded

```
pragma Assert (for all Y : ???. (if Reach (X, Y) then ...));
```

- Can be defined through iteration primitives

```
type List is access List_Cell with
  Iterable => (First => First,
               Next  => Next,
               ...);
function First (L : List) return Cursor;
function Next (L : List; C : Cursor) return Cursor;
```

4. Quantification over Recursive Data Structures

- Define a generic package providing this aspect for any simply recursive data type

```
generic
   type Cell is private;
   type Base_Cont is access Cell;
   type Succ is (<>);
   with function Next (L : access Cell; S: Succ) return access Cell;
package Iterator with SPARK_Mode is
   type Container is new Base_Cont with
        Iterable => ...;         -- allow quantification
   type Address is private;      -- direct access to Why3 representation
   function Reach (L : Container; A1, A2 : Address) return Boolean;
```

4. Quantification over Recursive Data Structures

- 4 examples on lists and 3 on binary trees manually translated to Why

```
procedure All_To_Zero (L : in out List) with
  Post => (for all Y of L => Y.Content = 0);
```

- Proof requires a complex axiomatization of reachability which should be generated depending on the data structure

Lists	Binary Trees
150 lines	230 lines
26 axioms	39 axioms
15 are redundant	23 are redundant