Program Verification in SPARK and ACSL: A Comparative Case Study

Eduardo Brito and Jorge Sousa Pinto
{edbrito,jsp}@di.uminho.pt

- Introduction
- Background
    - Software Contracts & Verification Process
    - SPARK
    - ACSL
    - Running Example (Stack)
- Bounded Stack Specification
    - Reasoning With Specifications
- Implementation, Refinement & Program Verification
- Conclusion
- Future Work

- Why compare SPARK and ACSL?
  - C and Ada are the most used languages in critical systems;
  - SPARK enables program verification (among other things) for a subset of Ada;
  - ACSL is “ANSI/ISO C Specification Language”;
  - To show the similarities and differences between the two specification languages and approaches.
- Promoting the use of verification tools for both languages!
  - As a short tutorial using a simple example.

- Software Contracts
  - We write the contracts in a Behavioral Interface Specification Language;
  - The contracts are the specification of properties;
  - The contracts state:
    - what a subprogram is expecting; (pre-condition)
      - Established by the caller.
    - what is expected from the suprogram. (post-condition)
      - Established by the callee.
    - There are usually contracts describing the state of the program (e.g. data and class invariants).

- Program Verification
  - After the program is annotated with its annotations...
  - … the Verification Condition (VC) generator (VCGen) generates the VCs/Proof Obligations;
  - The VCs are fed to theorem provers;
  - They may be discharged (proved to be valid) automatically (if possible) or manually.
    - Or we may be able to find counter-examples that show the VC is not valid.

- SPARK
  - The language is a strict/true subset of Ada;
  - Uses its own BISL for the contracts;
  - Uses a toolset to enforce its subset of Ada and to generate and discharge Vcs;
  - Depends on Ada compilers;
  - Used in several large safety-critical projects and is the focus of on-going academic research.

- ACSL
  - It is used for ANSI/ISO C code;
  - The language is a separate entity from the annotations;
  - The BISL has to deal with more problems (e.g. pointers, dynamic memory...);
  - It provides several ways to specify mathematical properties (axioms, lemmas, predicates, behaviours...).

Running example (general stack specification)

```
nat count()
nat capacity()
boolean isEmpty()
Postcond: Result = (count() = 0)
boolean isFull()
Postcond: Result = (count() = capacity())
int top()
Precond: not isEmpty()
void pop()
Precond: not isEmpty(); Postcond: count() = old_count() - 1
void push(int n)
Precond: not isFull(); Postcond: count() = old_count() + 1 and top() = n
```

What is missing?

```
package Stack
--# own State: StackType;
is
   --# type StackType is abstract;
   --# function Count_of(S: StackType) return Natural;
   --# function Cap_of(S: StackType) return Natural;
   --# function Substack(S1: StackType; S2: StackType) return Boolean;

   MaxStackSize: constant := 100;

   procedure Init;
   --# global out State;
   --# derives State from;
   --# post Cap_of(State) = MaxStackSize and Count_of(State) = 0;

   function isEmpty return Boolean;
   --# global State;
   --# return Count_of(State) = 0;

   function isFull return Boolean;
   --# global State;
   --# return Count_of(State) = Cap_of(State);

   function Top return Integer;
   --# global State;
   --# pre Count_of(State) > 0;

   procedure Pop;
   --# global in out State;
   --# derives State from State;
   --# pre 0 < Count_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)-1 and
   --#      Substack(State, State~);

   procedure Push(X: in Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre Count_of(State) < Cap_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)+1 and
   --#      Top(State) = X and Substack(State~, State);
end Stack;
```

```
package Stack
--# own State: StackType;
is
    --# type StackType is abstract;
    --# function Count_of(S: StackType) return Natural;
    --# function Cap_of(S: StackType) return Natural;
    --# function Substack(S1: StackType; S2: StackType) return Boolean;

    MaxStackSize: constant := 100;

    procedure Init;
    --# global out State;
    --# derives State from;
    --# post Cap_of(State) = MaxStackSize and Count_of(State) = 0;

    function isEmpty return Boolean;
    --# global State;
    --# return Count_of(State) = 0;

    function isFull return Boolean;
    --# global State;
    --# return Count_of(State) = Cap_of(State);

    function Top return Integer;
    --# global State;
    --# pre Count_of(State) > 0;

    procedure Pop;
    --# global in out State;
    --# derives State from State;
    --# pre 0 < Count_of(State);
    --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)-1 and
    --#      Substack(State, State~);

    procedure Push(X: in Integer);
    --# global in out State;
    --# derives State from State, X;
    --# pre Count_of(State) < Cap_of(State);
    --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)+1 and
    --#      Top(State) = X and Substack(State~, State);
end Stack;
```

```
package Stack
--# own State: StackType;
is
    --# type StackType is abstract;
    --# function Count_of(S: StackType) return Natural;
    --# function Cap_of(S: StackType) return Natural;
    --# function Substack(S1: StackType; S2: StackType) return Boolean;

    MaxStackSize: constant := 100;

    procedure Init;
    --# global out State;
    --# derives State from;
    --# post Cap_of(State) = MaxStackSize and Count_of(State) = 0;

    function isEmpty return Boolean;
    --# global State;
    --# return Count_of(State) = 0;

    function isFull return Boolean;
    --# global State;
    --# return Count_of(State) = Cap_of(State);

    function Top return Integer;
    --# global State;
    --# pre Count_of(State) > 0;

    procedure Pop;
    --# global in out State;
    --# derives State from State;
    --# pre 0 < Count_of(State);
    --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)-1 and
    --#       Substack(State, State~);

    procedure Push(X: in Integer);
    --# global in out State;
    --# derives State from State, X;
    --# pre Count_of(State) < Cap_of(State);
    --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)+1 and
    --#       Top(State) = X and Substack(State~, State);
end Stack;
```

```
package Stack
--# own State: StackType;
is
   --# type StackType is abstract;
   --# function Count_of(S: StackType) return Natural;
   --# function Cap_of(S: StackType) return Natural;
   --# function Substack(S1: StackType; S2: StackType) return Boolean;

   MaxStackSize: constant := 100;

   procedure Init;
   --# global out State;
   --# derives State from;
   --# post Cap_of(State) = MaxStackSize and Count_of(State) = 0;

   function isEmpty return Boolean;
   --# global State;
   --# return Count_of(State) = 0;

   function isFull return Boolean;
   --# global State;
   --# return Count_of(State) = Cap_of(State);

   function Top return Integer;
   --# global State;
   --# pre Count_of(State) > 0;

   procedure Pop;
   --# global in out State;
   --# derives State from State;
   --# pre 0 < Count_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)-1 and
   --#      Substack(State, State~);

   procedure Push(X: in Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre Count_of(State) < Cap_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)+1 and
   --#      Top(State) = X and Substack(State~, State);
end Stack;
```

Now in C/ACSL

Bounded Stack Specification

```
typedef ... Stack;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = ...
  @ logic integer top_of{L} (Stack st) = ...
  @ logic integer count_of{L} (Stack st) = ...
  @ predicate substack{L1,L2} (Stack st) = ...
  @ } */

/*@ requires cap >= 0;
  @ ensures cap_of{Here}(st) == cap  && count_of{Here}(st) == 0;
  @*/
void init (int cap);

/*@ assigns \nothing;
  @ behavior empty:
  @    assumes count_of{Here}(st) == 0;
  @    ensures \result == 1;
  @ behavior not_empty:
  @    assumes count_of{Here}(st) != 0;
  @    ensures \result == 0;
  @*/
int isEmpty (void);

/*@ assigns \nothing;
  @ behavior full:
  @    assumes count_of{Here}(st) == cap_of{Here}(st);
  @    ensures \result == 1;
  @ behavior not_full:
  @    assumes count_of{Here}(st) != cap_of{Here}(st);
  @    ensures \result == 0;
  @*/
int isFull (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures \result == top_of{Here}(st);
  @ assigns \nothing;
  @*/
int top (void);
```

```
/*@ requires 0 < count_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) - 1 &&
  @         substack{Here,Old}(st);
  @*/
void pop(void);

/*@ requires count_of{Here}(st) < cap_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) + 1 &&
  @         top_of{Here}(st) == x && substack{Old,Here}(st);
  @*/
void push (int x);
```

```
typedef ... Stack;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = ...
  @ logic integer top_of{L} (Stack st) = ...
  @ logic integer count_of{L} (Stack st) = ...
  @ predicate substack{L1,L2} (Stack st) = ...
  @ } */
```

```
/*@ requires cap >= 0;
  @ ensures cap_of{Here}(st) == cap  && count_of{Here}(st) == 0;
  @*/
void init (int cap);

/*@ assigns \nothing;
  @ behavior empty:
  @    assumes count_of{Here}(st) == 0;
  @    ensures \result == 1;
  @ behavior not_empty:
  @    assumes count_of{Here}(st) != 0;
  @    ensures \result == 0;
  @*/
int isEmpty (void);

/*@ assigns \nothing;
  @ behavior full:
  @    assumes count_of{Here}(st) == cap_of{Here}(st);
  @    ensures \result == 1;
  @ behavior not_full:
  @    assumes count_of{Here}(st) != cap_of{Here}(st);
  @    ensures \result == 0;
  @*/
int isFull (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures \result == top_of{Here}(st);
  @ assigns \nothing;
  @*/
int top (void);
```

```
typedef ... Stack;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = ...
  @ logic integer top_of{L} (Stack st) = ...
  @ logic integer count_of{L} (Stack st) = ...
  @ predicate substack{L1,L2} (Stack st) = ...
  @ } */

/*@ requires cap >= 0;
  @ ensures cap_of{Here}(st) == cap  && count_of{Here}(st) == 0;
  @*/
void init (int cap);

/*@ assigns \nothing;
  @ behavior empty:
  @    assumes count_of{Here}(st) == 0;
  @    ensures \result == 1;
  @ behavior not_empty:
  @    assumes count_of{Here}(st) != 0;
  @    ensures \result == 0;
  @*/
int isEmpty (void);

/*@ assigns \nothing;
  @ behavior full:
  @    assumes count_of{Here}(st) == cap_of{Here}(st);
  @    ensures \result == 1;
  @ behavior not_full:
  @    assumes count_of{Here}(st) != cap_of{Here}(st);
  @    ensures \result == 0;
  @*/
int isFull (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures \result == top_of{Here}(st);
  @ assigns \nothing;
  @*/
int top (void);
```

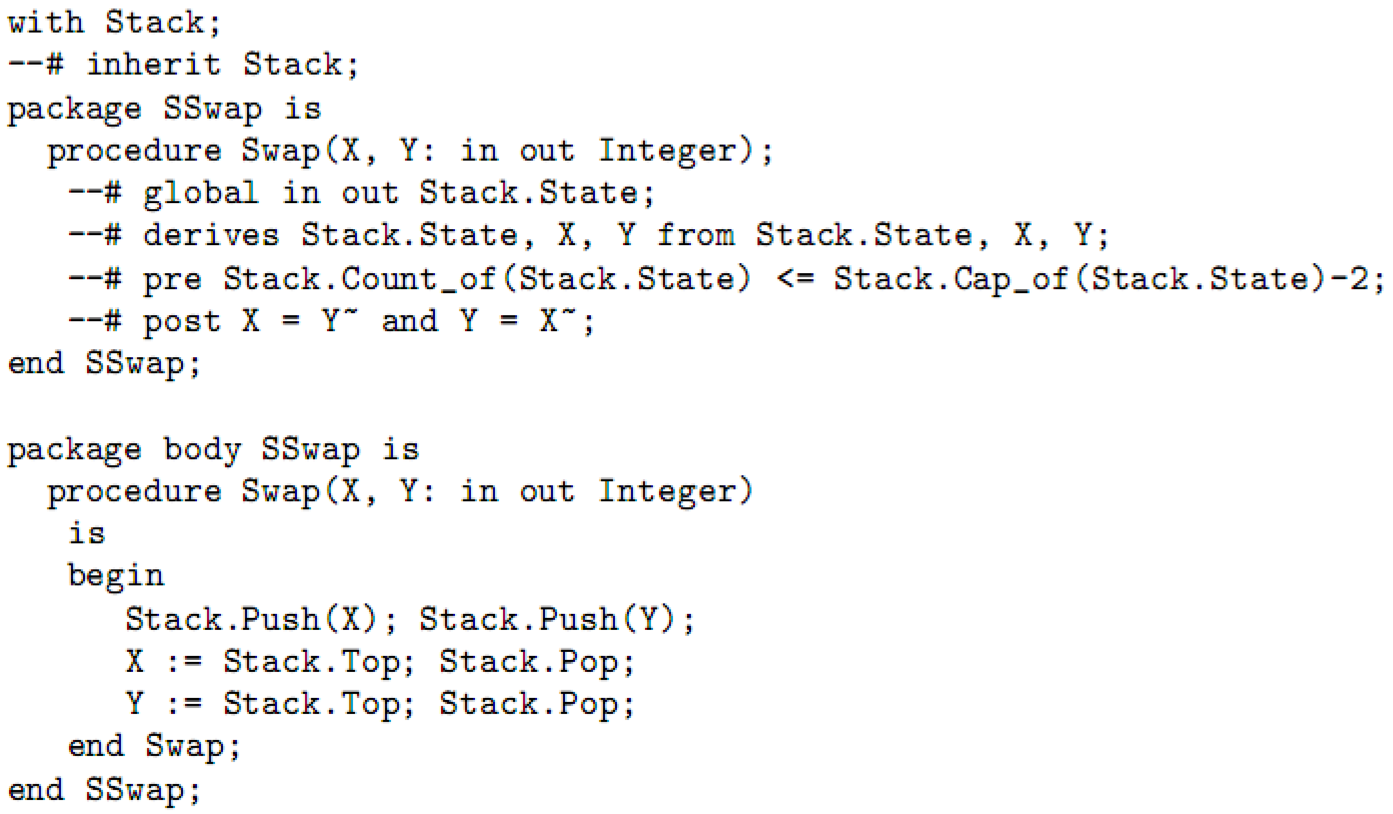
Bounded Stack Specification

```
/*@ requires 0 < count_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) - 1 &&
  @         substack{Here,Old}(st);
  @*/
void pop(void);
```

```
/*@ requires count_of{Here}(st) < cap_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) + 1 &&
  @         top_of{Here}(st) == x && substack{Old,Here}(st);
  @*/
void push (int x);
```

State labels

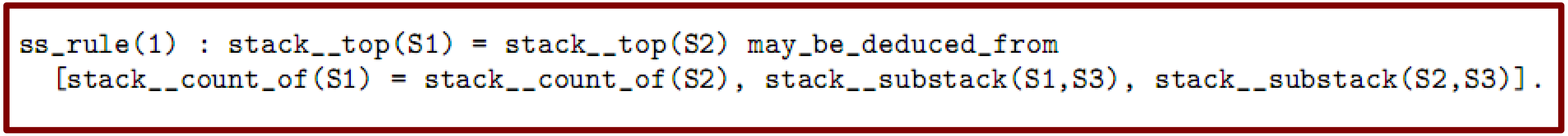
- Reasoning With Specifications



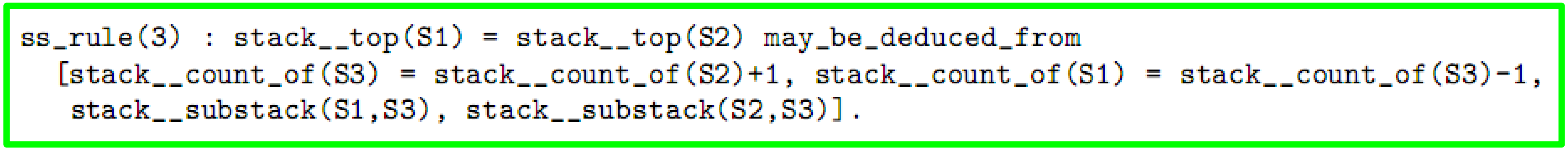
```
with Stack;
--# inherit Stack;
package SSwap is
  procedure Swap(X, Y: in out Integer);
   --# global in out Stack.State;
   --# derives Stack.State, X, Y from Stack.State, X, Y;
   --# pre Stack.Count_of(Stack.State) <= Stack.Cap_of(Stack.State)-2;
   --# post X = Y~ and Y = X~;
end SSwap;

package body SSwap is
  procedure Swap(X, Y: in out Integer)
   is
   begin
      Stack.Push(X); Stack.Push(Y);
      X := Stack.Top; Stack.Pop;
      Y := Stack.Top; Stack.Pop;
   end Swap;
end SSwap;
```

Is it wrong not to ensure that the stack stays the same? We say it depends(?)

- Automatic proof with Simplifier



```
ss_rule(1) : stack__top(S1) = stack__top(S2) may_be_deduced_from
   [stack__count_of(S1) = stack__count_of(S2), stack__substack(S1,S3), stack__substack(S2,S3)].
```



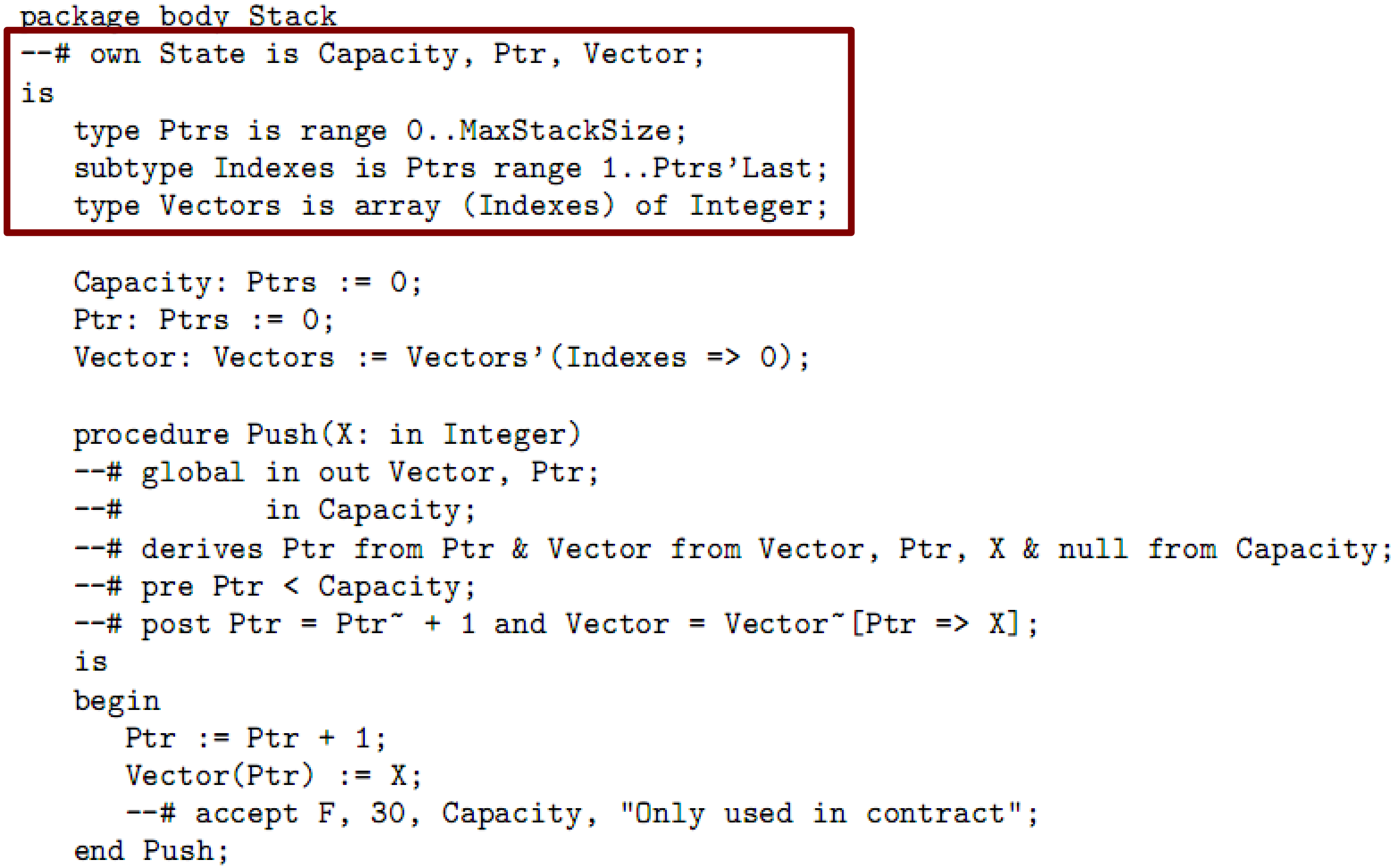
```
ss_rule(3) : stack__top(S1) = stack__top(S2) may_be_deduced_from
   [stack__count_of(S3) = stack__count_of(S2)+1, stack__count_of(S1) = stack__count_of(S3)-1,
    stack__substack(S1,S3), stack__substack(S2,S3)].
```

Equivalent rules but only the second is able to discharge the VC.

Bounded Stack Implementation

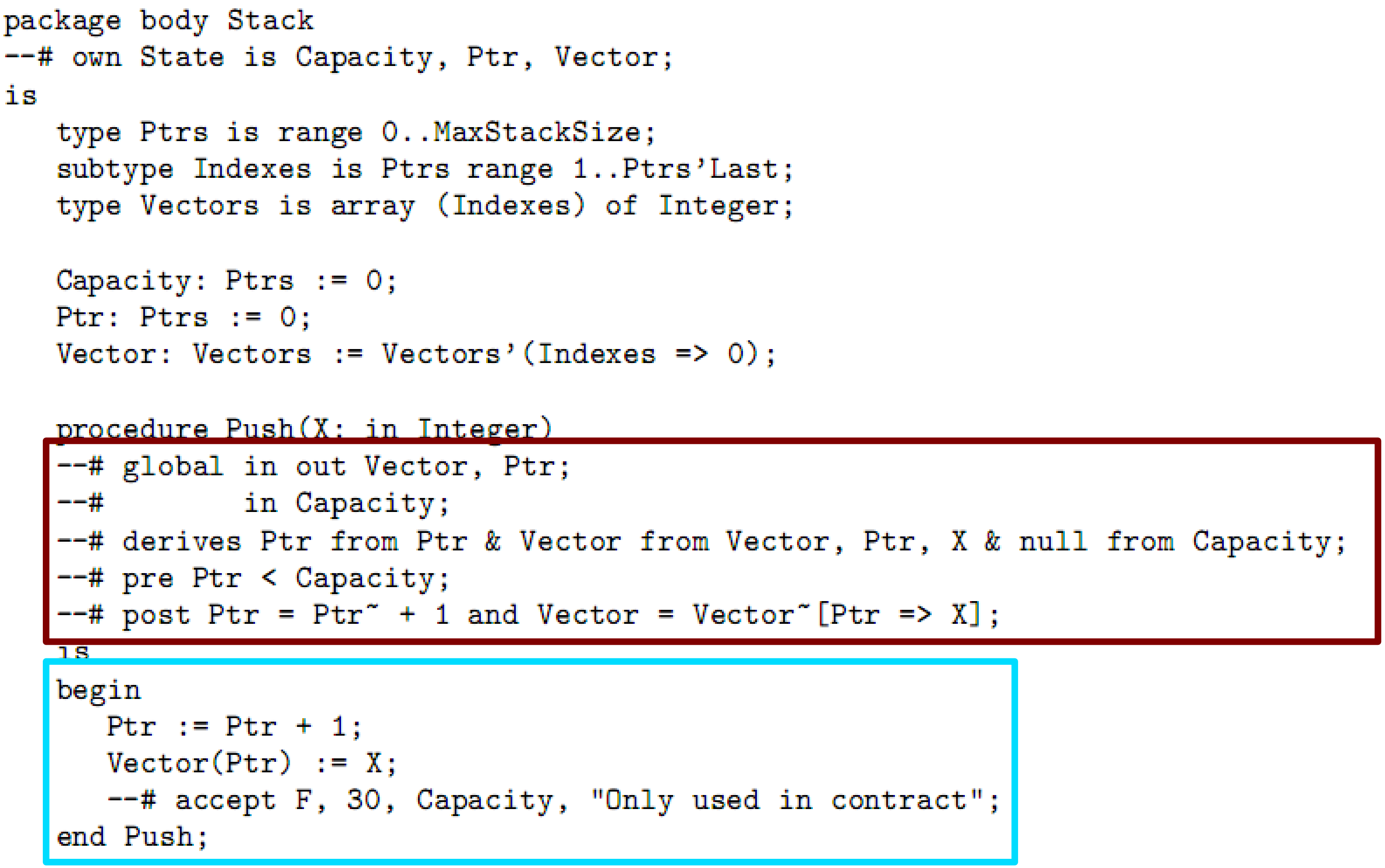
```
package body Stack
--# own State is Capacity, Ptr, Vector;
is
   type Ptrs is range 0..MaxStackSize;
   subtype Indexes is Ptrs range 1..Ptrs'Last;
   type Vectors is array (Indexes) of Integer;

   Capacity: Ptrs := 0;
   Ptr: Ptrs := 0;
   Vector: Vectors := Vectors'(Indexes => 0);

   procedure Push(X: in Integer)
   --# global in out Vector, Ptr;
   --#        in Capacity;
   --# derives Ptr from Ptr & Vector from Vector, Ptr, X & null from Capacity;
   --# pre Ptr < Capacity;
   --# post Ptr = Ptr~ + 1 and Vector = Vector~[Ptr => X];
   is
   begin
      Ptr := Ptr + 1;
      Vector(Ptr) := X;
      --# accept F, 30, Capacity, "Only used in contract";
   end Push;
```

Bounded Stack Implementation



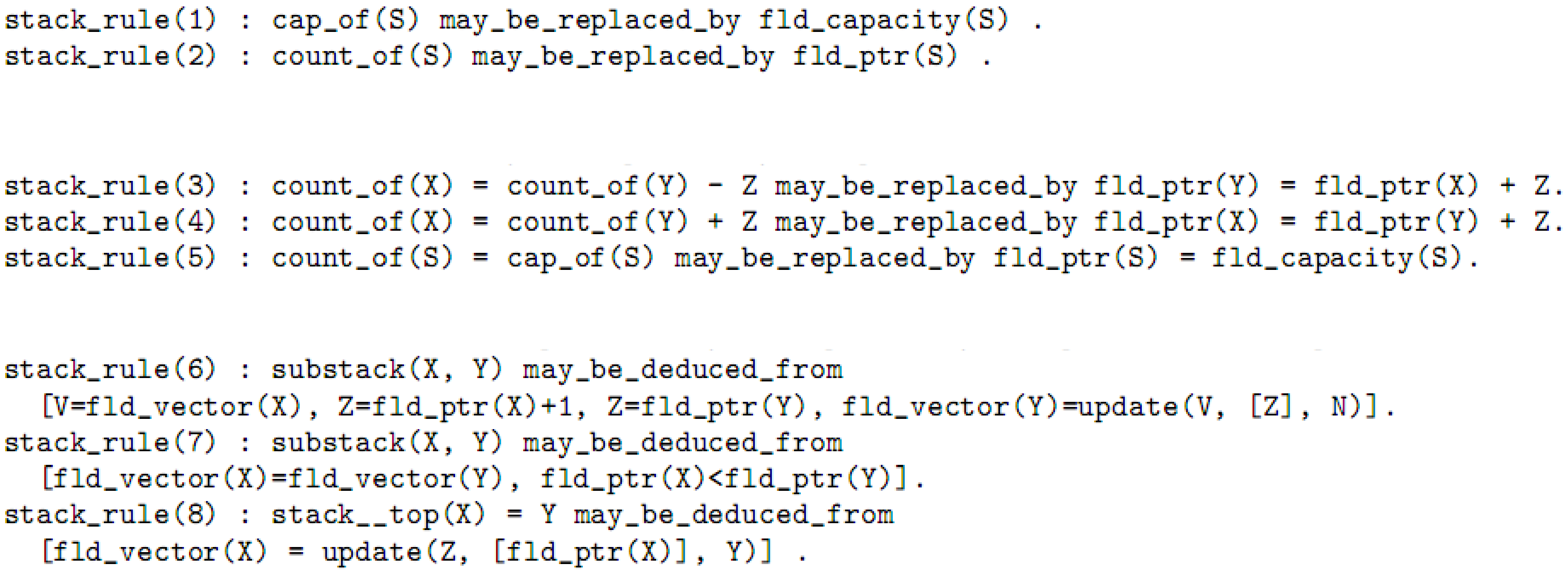
```
package body Stack
--# own State is Capacity, Ptr, Vector;
is
   type Ptrs is range 0..MaxStackSize;
   subtype Indexes is Ptrs range 1..Ptrs'Last;
   type Vectors is array (Indexes) of Integer;

   Capacity: Ptrs := 0;
   Ptr: Ptrs := 0;
   Vector: Vectors := Vectors'(Indexes => 0);

   procedure Push(X: in Integer)
   --# global in out Vector, Ptr;
   --#        in Capacity;
   --# derives Ptr from Ptr & Vector from Vector, Ptr, X & null from Capacity;
   --# pre Ptr < Capacity;
   --# post Ptr = Ptr~ + 1 and Vector = Vector~[Ptr => X];
   is
   begin
      Ptr := Ptr + 1;
      Vector(Ptr) := X;
      --# accept F, 30, Capacity, "Only used in contract";
   end Push;
```

Bounded Stack Implementation



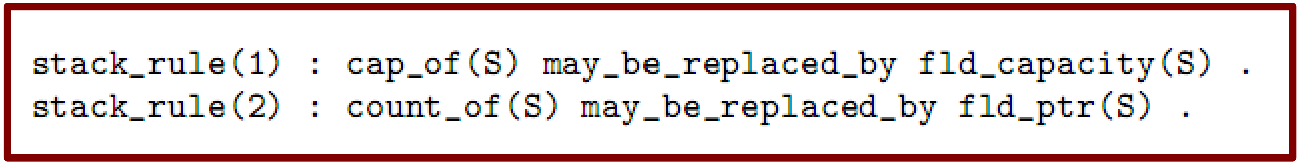
```
package body Stack
--# own State is Capacity, Ptr, Vector;
is
   type Ptrs is range 0..MaxStackSize;
   subtype Indexes is Ptrs range 1..Ptrs'Last;
   type Vectors is array (Indexes) of Integer;

   Capacity: Ptrs := 0;
   Ptr: Ptrs := 0;
   Vector: Vectors := Vectors'(Indexes => 0);

   procedure Push(X: in Integer)
   --# global in out Vector, Ptr;
   --#        in Capacity;
   --# derives Ptr from Ptr & Vector from Vector, Ptr, X & null from Capacity;
   --# pre Ptr < Capacity;
   --# post Ptr = Ptr~ + 1 and Vector = Vector~[Ptr => X];
   is
   begin
      Ptr := Ptr + 1;
      Vector(Ptr) := X;
      --# accept F, 30, Capacity, "Only used in contract";
   end Push;
```

- Proof rules in SPARK

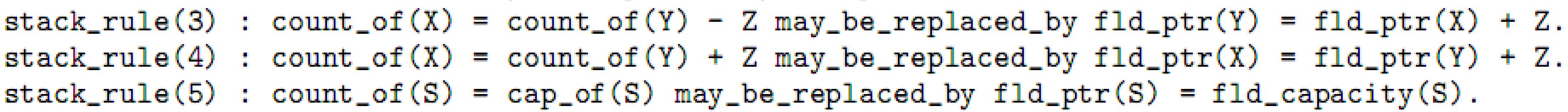


```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .


stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z.
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).


stack_rule(6) : substack(X, Y) may_be_deduced_from
  [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
  [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
  [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

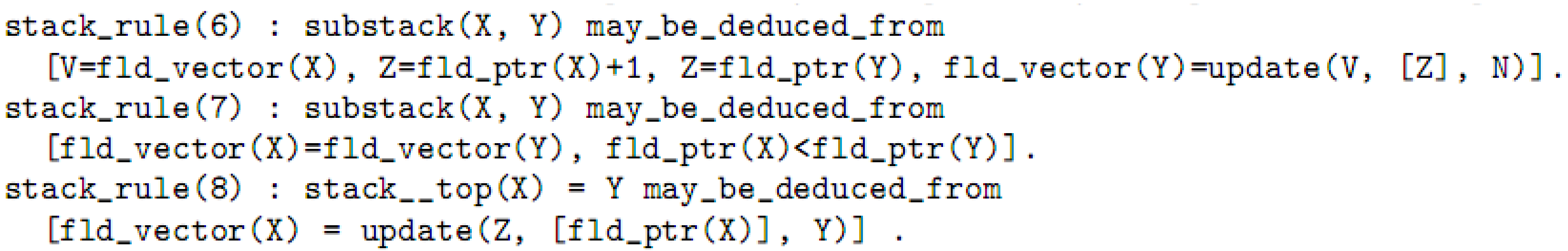
- Proof rules in SPARK



```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .
```

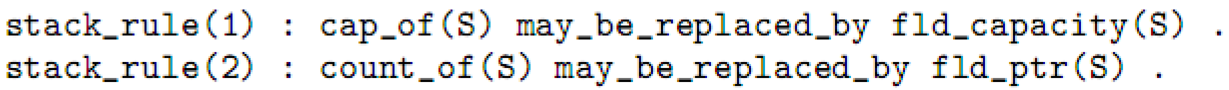


```
stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z.
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).
```

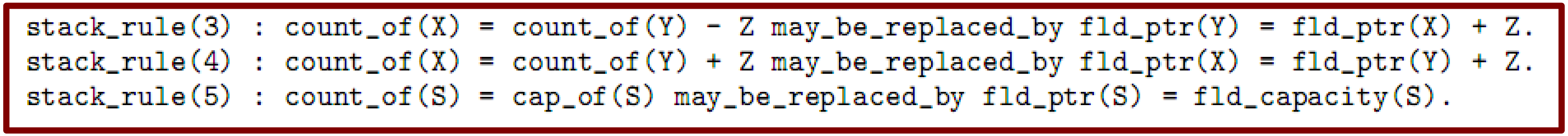


```
stack_rule(6) : substack(X, Y) may_be_deduced_from
   [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
   [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
   [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

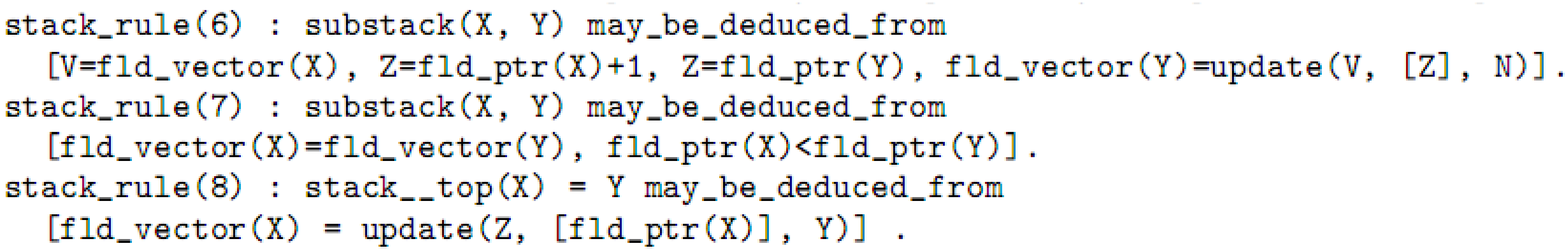
- Proof rules in SPARK



```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .
```

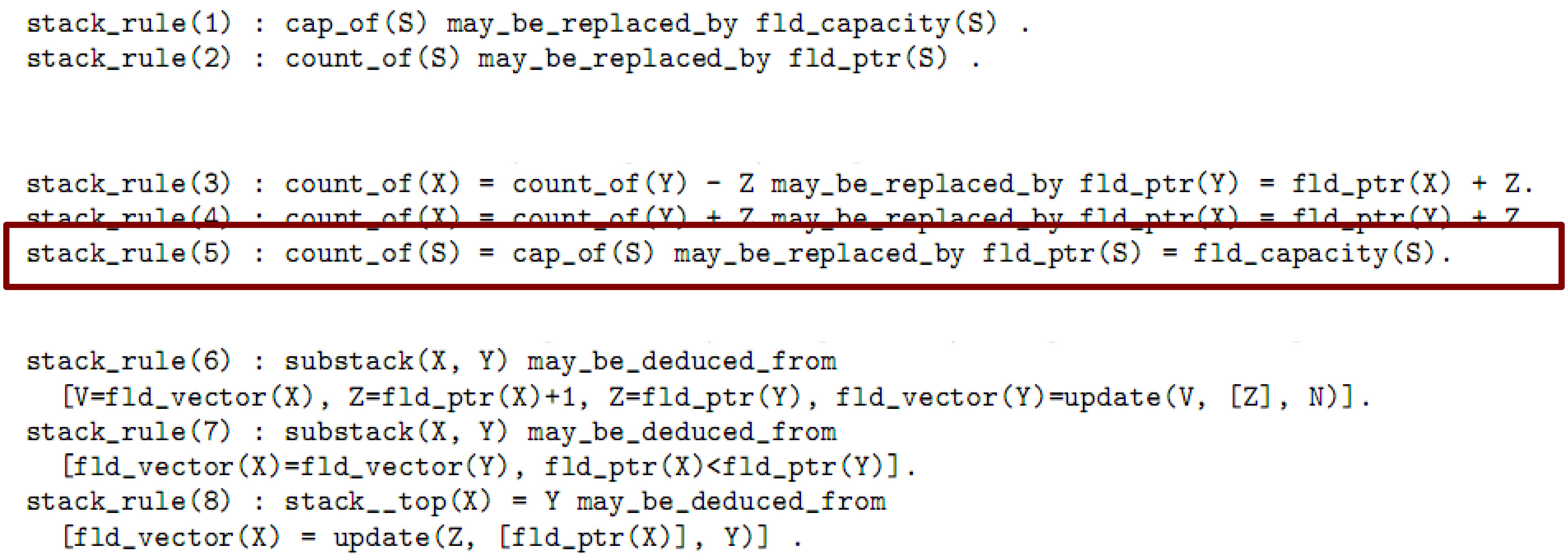


```
stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z.
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).
```



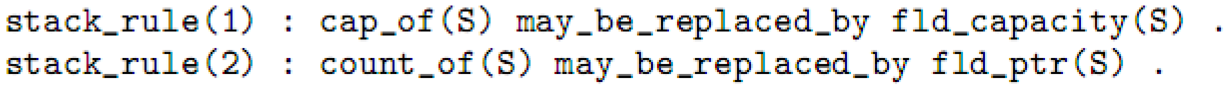
```
stack_rule(6) : substack(X, Y) may_be_deduced_from
   [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
   [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
   [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

- Proof rules in SPARK

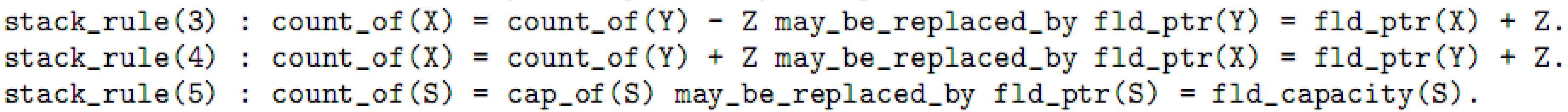


```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .


stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).


stack_rule(6) : substack(X, Y) may_be_deduced_from
   [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
   [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
   [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

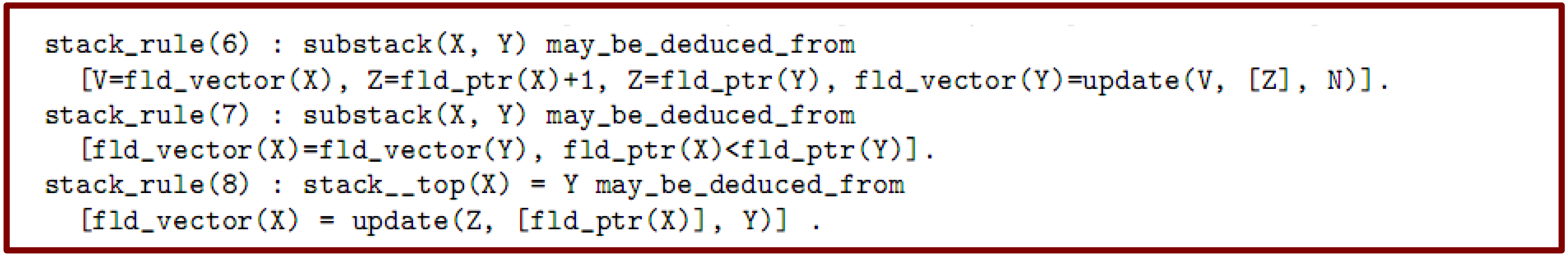
- Proof rules in SPARK



```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .
```



```
stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z.
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).
```



```
stack_rule(6) : substack(X, Y) may_be_deduced_from
   [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
   [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
   [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

Now for C/ACSL

Bounded Stack Implementation

```
typedef struct stack {
  int capacity;
  int size;
  int *elems;
} Stack;

int x, y;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = st.capacity;
  @ logic integer top_of{L} (Stack st) = st.elems[st.size-1];
  @ logic integer count_of{L} (Stack st) = st.size;
  @ predicate substack{L1,L2} (Stack st) = \at(st.size,L1) <= \at(st.size,L2) &&
  @  \forall integer i; 0<=i<\at(st.size,L1) ==> \at(st.elems[i],L1) == \at(st.elems[i],L2);
  @ predicate stinv{L}(Stack st) =
  @  \valid_range(st.elems,0,st.capacity-1) && 0 <= count_of{L}(st) <= cap_of{L}(st);
  @ } */

/*@ requires count_of{Here}(st) < cap_of{Here}(st) && stinv{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) && count_of{Here}(st) == count_of{Old}(st)+1
  @  && top_of{Here}(st) == x && substack{Old,Here}(st) && stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}
```

Bounded Stack Implementation

```
typedef struct stack {
  int capacity;
  int size;
  int *elems;
} Stack;

int x, y;
Stack st;
```

```
/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = st.capacity;
  @ logic integer top_of{L} (Stack st) = st.elems[st.size-1];
  @ logic integer count_of{L} (Stack st) = st.size;
  @ predicate substack{L1,L2} (Stack st) = \at(st.size,L1) <= \at(st.size,L2) &&
  @  \forall integer i; 0<=i<\at(st.size,L1) ==> \at(st.elems[i],L1) == \at(st.elems[i],L2);
  @ predicate stinv{L}(Stack st) =
  @  \valid_range(st.elems,0,st.capacity-1) && 0 <= count_of{L}(st) <= cap_of{L}(st);
  @ } */

/*@ requires count_of{Here}(st) < cap_of{Here}(st) && stinv{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) && count_of{Here}(st) == count_of{Old}(st)+1
  @  && top_of{Here}(st) == x && substack{Old,Here}(st) && stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}
```

Bounded Stack Implementation

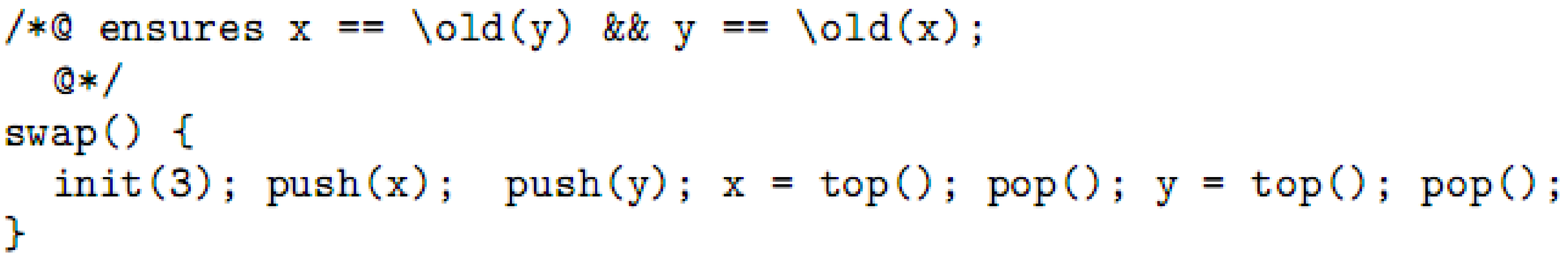
```
typedef struct stack {
  int capacity;
  int size;
  int *elems;
} Stack;

int x, y;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = st.capacity;
  @ logic integer top_of{L} (Stack st) = st.elems[st.size-1];
  @ logic integer count_of{L} (Stack st) = st.size;
  @ predicate substack{L1,L2} (Stack st) = \at(st.size,L1) <= \at(st.size,L2) &&
  @  \forall integer i; 0<=i<\at(st.size,L1) ==> \at(st.elems[i],L1) == \at(st.elems[i],L2);
  @ predicate stinv{L}(Stack st) =
  @  \valid_range(st.elems,0,st.capacity-1) && 0 <= count_of{L}(st) <= cap_of{L}(st);
  @ } */

/*@ requires count_of{Here}(st) < cap_of{Here}(st) && stinv{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) && count_of{Here}(st) == count_of{Old}(st)+1
  @  && top_of{Here}(st) == x && substack{Old,Here}(st) && stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}
```

Bounded Stack Implementation

```
typedef struct stack {
  int capacity;
  int size;
  int *elems;
} Stack;

int x, y;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = st.capacity;
  @ logic integer top_of{L} (Stack st) = st.elems[st.size-1];
  @ logic integer count_of{L} (Stack st) = st.size;
  @ predicate substack{L1,L2} (Stack st) = \at(st.size,L1) <= \at(st.size,L2) &&
  @  \forall integer i; 0<=i<\at(st.size,L1) ==> \at(st.elems[i],L1) == \at(st.elems[i],L2);
  @ predicate stinv{L}(Stack st) =
  @  \valid_range(st.elems,0,st.capacity-1) && 0 <= count_of{L}(st) <= cap_of{L}(st);
  @ } */

/*@ requires count_of{Here}(st) < cap_of{Here}(st) && stinv{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) && count_of{Here}(st) == count_of{Old}(st)+1
  @  && top_of{Here}(st) == x && substack{Old,Here}(st) && stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}
```

Array bounds safety condition

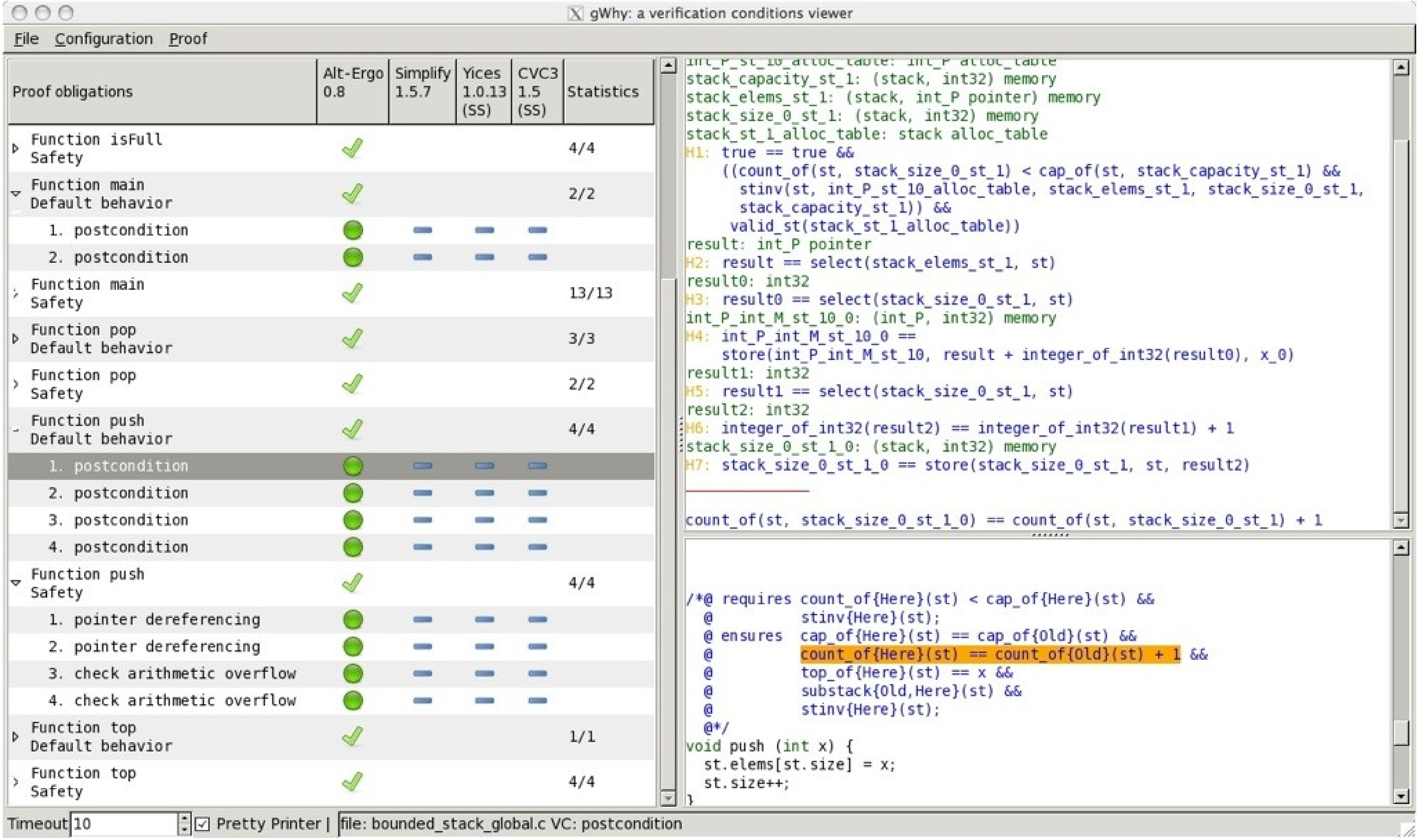
Bounded Stack Implementation



```
/*@ ensures x == \old(y) && y == \old(x);
  @*/
swap() {
  init(3); push(x);  push(y); x = top(); pop(); y = top(); pop();
}
```

- Same “swap with stack” as in the SPARK example.
  - Discharges all proof obligations without needing additional rules;
  - Requires an implementation.

Bounded Stack Implementation



Proof obligations	Alt-Ergo 0.8	Simplify 1.5.7	Yices 1.0.13 (SS)	CVC3 1.5 (SS)	Statistics
Function isFull Safety	✓				4/4
Function main Default behavior	✓				2/2
1. postcondition	●	▬	▬	▬	
2. postcondition	●	▬	▬	▬	
Function main Safety	✓				13/13
Function pop Default behavior	✓				3/3
Function pop Safety	✓				2/2
Function push Default behavior	✓				4/4
1. postcondition	●	▬	▬	▬	
2. postcondition	●	▬	▬	▬	
3. postcondition	●	▬	▬	▬	
4. postcondition	●	▬	▬	▬	
Function push Safety	✓				4/4
1. pointer dereferencing	●	▬	▬	▬	
2. pointer dereferencing	●	▬	▬	▬	
3. check arithmetic overflow	●	▬	▬	▬	
4. check arithmetic overflow	●	▬	▬	▬	
Function top Default behavior	✓				1/1
Function top Safety	✓				4/4

```
int_P_st_10_alloc_table: int_P alloc_table
stack_capacity_st_1: (stack, int32) memory
stack_elems_st_1: (stack, int_P pointer) memory
stack_size_0_st_1: (stack, int32) memory
stack_st_1_alloc_table: stack alloc_table
H1: true == true &&
    ((count_of(st, stack_size_0_st_1) < cap_of(st, stack_capacity_st_1) &&
      stinv(st, int_P_st_10_alloc_table, stack_elems_st_1, stack_size_0_st_1,
      stack_capacity_st_1)) &&
     valid_st(stack_st_1_alloc_table))
result: int_P pointer
H2: result == select(stack_elems_st_1, st)
result0: int32
H3: result0 == select(stack_size_0_st_1, st)
int_P_int_M_st_10_0: (int_P, int32) memory
H4: int_P_int_M_st_10_0 ==
    store(int_P_int_M_st_10, result + integer_of_int32(result0), x_0)
result1: int32
H5: result1 == select(stack_size_0_st_1, st)
result2: int32
H6: integer_of_int32(result2) == integer_of_int32(result1) + 1
stack_size_0_st_1_0: (stack, int32) memory
H7: stack_size_0_st_1_0 == store(stack_size_0_st_1, st, result2)

count_of(st, stack_size_0_st_1_0) == count_of(st, stack_size_0_st_1) + 1
```

```
/*@ requires count_of{Here}(st) < cap_of{Here}(st) &&
  @          stinv{Here}(st);
  @ ensures  cap_of{Here}(st) == cap_of{Old}(st) &&
  @          count_of{Here}(st) == count_of{Old}(st) + 1 &&
  @          top_of{Here}(st) == x &&
  @          substack{Old,Here}(st) &&
  @          stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}
```

Timeout 10 ☑ Pretty Printer | file: bounded_stack_global.c VC: postcondition

- Safety in SPARK is easier to prove;
- SPARK is better for software contracts;
    - Because of separate specification, mainly.
- SPARK has better support for abstraction;
- ACSL is more expressive;
- Functional correctness with ACSL is easier to prove;
- ACSL has better proof tool support;
- Hi-Lite, which has been announced last month, addresses the strengths of both approaches.

- This work is part of an effort aiming at formal verification of Ada;
- A MSc thesis related to the formalization of a subset of SPARK will be finished this year (hopefully!);
- 2 PhD projects starting now.