Formal semantics and other research around Spark2014

Virginia Aponte, **Pierre Courtieu**, Tristan Crolard (CPR Team - Cnam)
Zhi Zhang (SAnToS, KSU)

November 2014

In this talk

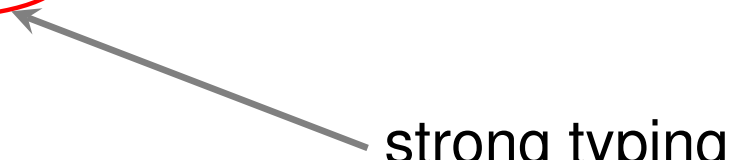
1. Ada and SPARK
2. Motivations & Current State
3. Run-Time Checks
4. Nested and recursive procedures
5. A prototype SPARK frontend for CompCert

Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```

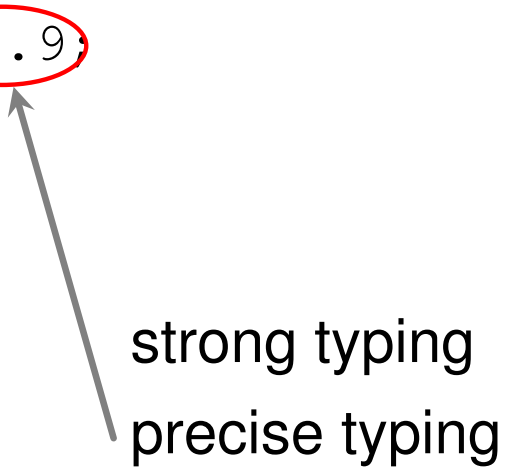
Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```



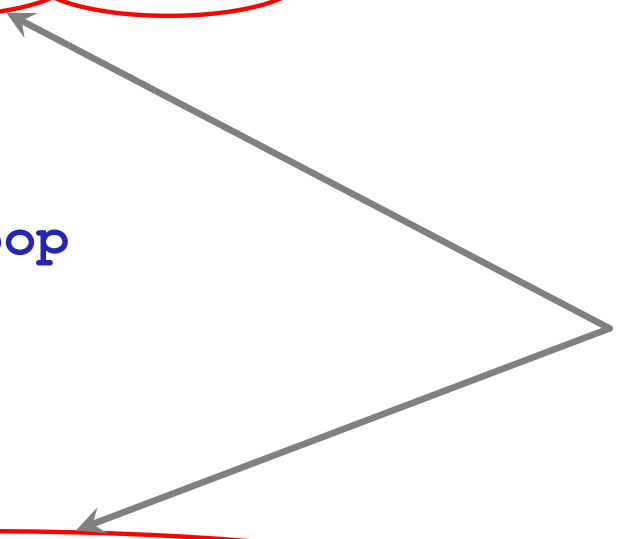
Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```



Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning: overlap
end F;
```



strong typing

precise typing

controlled effects

Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```

Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```

Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```

OK statically

OK statically

dynamic!

Ada is not Pascal

```
type MY_INT is new INTEGER range 0..9;

procedure F(U: in out MY_INT) is
   TMP : MY_INT := 0;
   RES : MY_INT := U;
begin
   while RES > TMP loop
      RES := RES - 1;
      TMP := TMP;
   end loop;
   U := RES;
   G(U,U); - warning:  overlap
end F;
```

OK statically

OK statically

dynamic!

$\rightarrow$ runtime checks ($\mathcal{RC}$)

Runtime checks

Tests at runtime, **exception** raised when failing

- implicit tests: range, overflows, division by zero, array bounds, ...
- **explicit annotations**
  - always executable (`gnat -gnata`)
  - used for testing mostly

```
procedure F(U: in out MY_INT)
  with Pre => U >=0, Post => U<= U'Old ;
```

Annotations

- In the specification (à la Hoare)
- In the code (à la Hoare)
- On types
- For flow analysis (SPARK only)
- etc

```
procedure F(U: in out MY_INT)
  with Pre => U >=0, Post => U<= U'Old ;
```

Annotations

- In the specification (à la Hoare)
- In the code (à la Hoare)
- On types
- For flow analysis (SPARK only)
- etc

```
while RES > TMP loop
  Pragma Loop_Invariant ((RES <= U'Loop_Entry)
   and (for all I in 0..t'range => t[I]>0)) ;
  ...
end loop;
Pragma Assert (RES <= TMP);
```

Annotations

- In the specification (à la Hoare)
- In the code (à la Hoare)
- On types
- For flow analysis (SPARK only)
- etc

```
type Stack is private
  with Type_Invariant => Is_Unduplicated(Stack);
```

Annotations

- In the specification (à la Hoare)
- In the code (à la Hoare)
- On types
- For flow analysis (SPARK only)
- etc

```
procedure F(U: out MY_INT2)
  with
    Global => (Input => (X,Y)),
    Depends => (U => (X,Y));
```

SPARK is not Ada

- Developed by ALTRAN and now AdaCore too
- Dedicated to static analysis
- No dedicated compiler, strict subset of Ada
  - ~~Access types~~
  - ~~Effects in expressions~~
  - Aliasing: ~~warning~~ not allowed (since no pointer)
  - ~~Exception handler~~ ...
- ~~runtime checks~~ static analysis
  - Verification conditions à la Hoare (VCs)

  → Ensures: **No check would fail at runtime**
  neither implicit (type/array bounds, initialization, div by zero, etc)
  nor explicit (assertion, loop invariant etc)

Plan

Motivations

- Qualification of SPARK (DO-178C)
- **AST validator w.r.t. run-time checks ($\mathcal{RC}$)**
- Link with other formalized tools (KSU)
- **CompCert**, certified compilation (Lockheed Martin)
- Semantics for Ada? ((very) long term)
- Proving invariant patterns

Previous works

- Ian O'Neil. The formal semantics of SPARK83. Technical report. (format: Z)
- Eduardo Brito. PhD. A Formal Approach for a Subset of the SPARK Programming Language (format: Natural Semantics style)

Work in progress

- Team work, 2 internships of Zhi Zhang
- Big step semantics on untyped terms (ASTs)
- Typing = Property on (untyped-)terms
- Judgments:
  - $\langle e, \sigma \rangle \rightsquigarrow v$ or $\langle e, \sigma \rangle \rightsquigarrow \bot$
  - $\langle p, \sigma \rangle \rightsquigarrow \sigma'$ or $\langle p, \sigma \rangle \rightsquigarrow \bot$
- $\bot$ = runtime error
- Abnormal (`1+TRUE`, uninitialized variable, etc) = execution blocked

Return states and error propagation

```
Inductive Return (A : Type) : Type :=
  Normal : A → Return A   (* σ, v *)
| Run_Time_Error : error_type → Return A  (* ⊥ (+ msg) *)
```

$$Seq_1 \ \frac{\langle c_1, \sigma \rangle \rightsquigarrow \bot}{\langle c_1 ; c_2, \sigma \rangle \rightsquigarrow \bot}$$

$$Seq_2 \ \frac{\langle c_1, \sigma \rangle \rightsquigarrow \sigma' \neq \bot \qquad \langle c_2, \sigma' \rangle \rightsquigarrow \bot}{\langle c_1 ; c_2, \sigma \rangle \rightsquigarrow \bot}$$

$$Seq_3 \ \frac{\langle c_1, \sigma \rangle \rightsquigarrow \sigma' \neq \bot \qquad \langle c_2, \sigma' \rangle \rightsquigarrow \sigma'' \neq \bot}{\langle c_1 ; c_2, \sigma \rangle \rightsquigarrow \sigma''}$$

Return states and error propagation

```
Inductive eval_stmt:
 symboltable → stack → statement → Return stack → Prop :=
...
| Eval_S_Sequence_RTE: ∀ st s c1 msg c2,
    eval_stmt st s c1 (Run_Time_Error msg) →
    eval_stmt st s (S_Sequence c1 c2) (Run_Time_Error msg)

| Eval_S_Sequence: ∀ st s c1 s1 c2 s2,
    eval_stmt st s c1 (Normal s1) →
    eval_stmt st s1 c2 s2 →
    eval_stmt st s (S_Sequence c1 c2) s2
```

Error Propagation

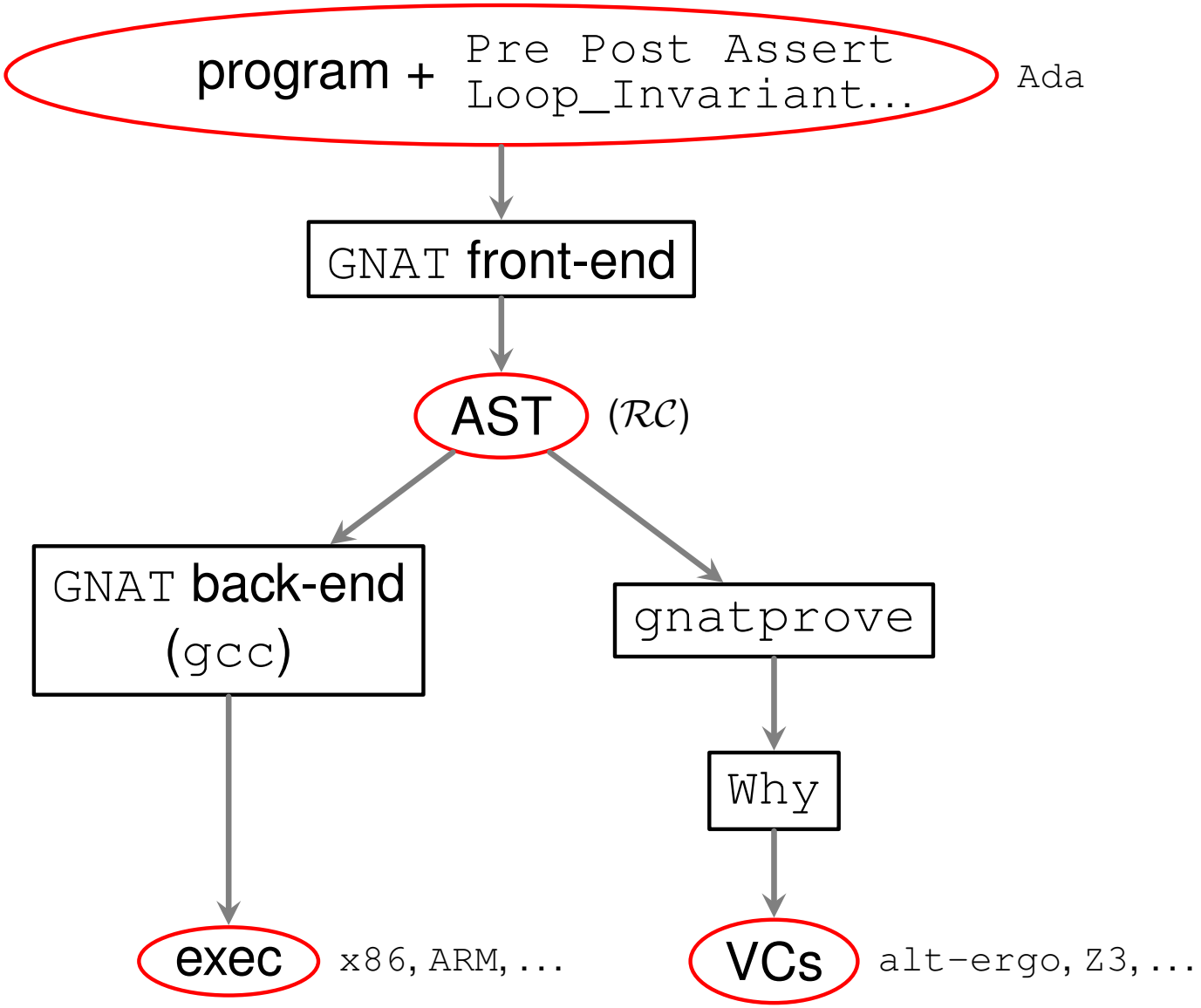
```
| Eval_S_If_RTE: ∀ st s b msg c1 c2,
    eval_expr st s b (Run_Time_Error msg) →
    eval_stmt st s (S_If b c1 c2) (Run_Time_Error msg)

| Eval_S_If_True: ∀ st s b c1 s1 c2,
    eval_expr st s b (Normal (BasicV (Bool true))) →
    eval_stmt st s c1 s1 →
    eval_stmt st s (S_If b c1 c2) s1
...
```

Work in progress

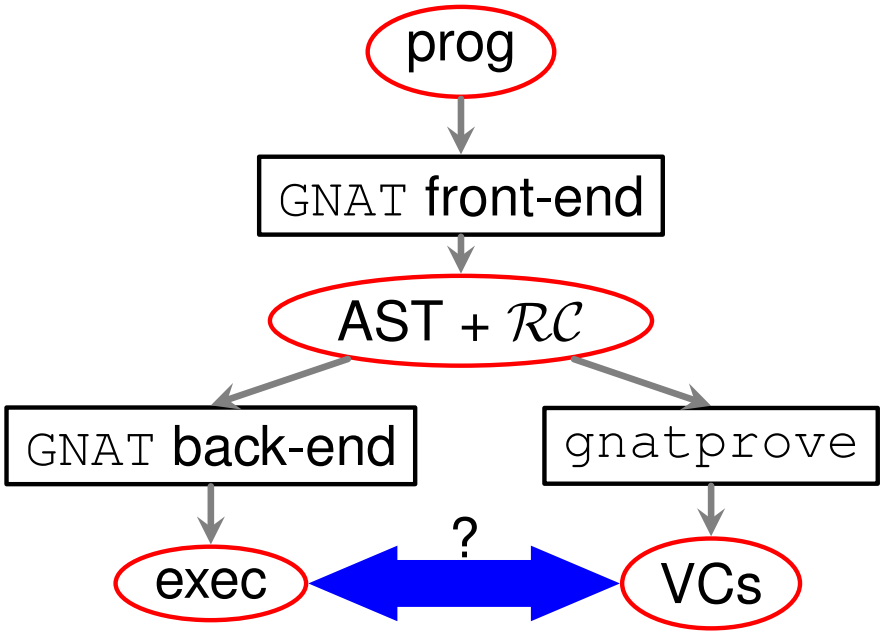
- Subset of SPARK covered:
  - Integers + subtypes + derived types
  - Arrays and records (nested: current work by Z. Zhang)
  - Procedures (recursive & nested), **`in`**, **`out`**, **`in out`**, No function
  - Run-Time checks: a few, no annotation yet
  - Prototype of a compiler to `Cminor` of CompCert (demo!)
    $\rightarrow$ ia32, ppc, Arm
  - No procedure contract yet
- Proved parts:
  - Type checking/initialization (on a smaller subset)
  - Run-Time checks (Z. Zhang)
  - No compilation proof (starting now)

Plan

GNAT/SPARK 2014



Correctness GNAT-gnatprove

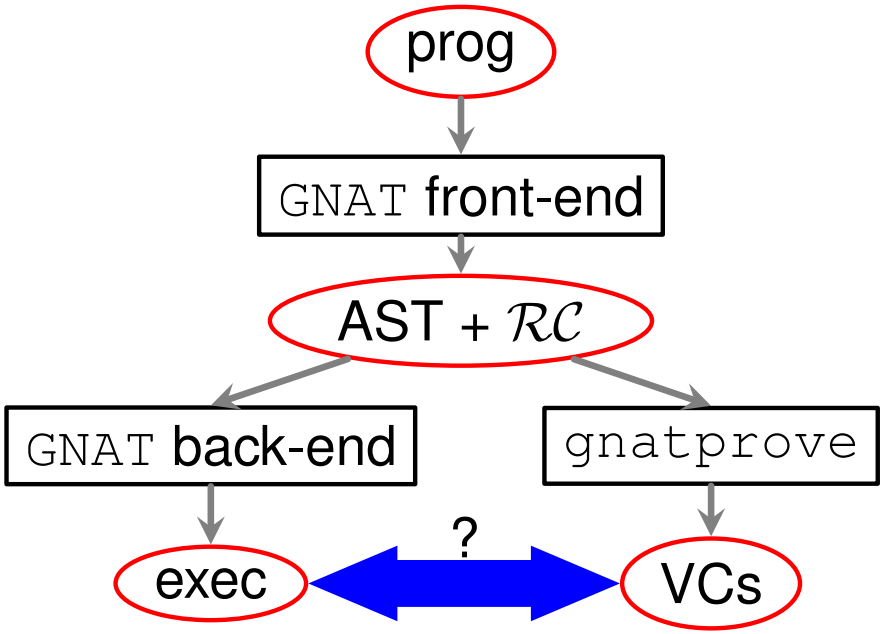


- Usual commutation diagram

$$\forall p : \mathit{AST}{+}\mathcal{RC},\ \texttt{gnatprove}(p) \rightarrow \forall s, \langle \texttt{GNAT}(p), s \rangle \not\rightsquigarrow \bot$$

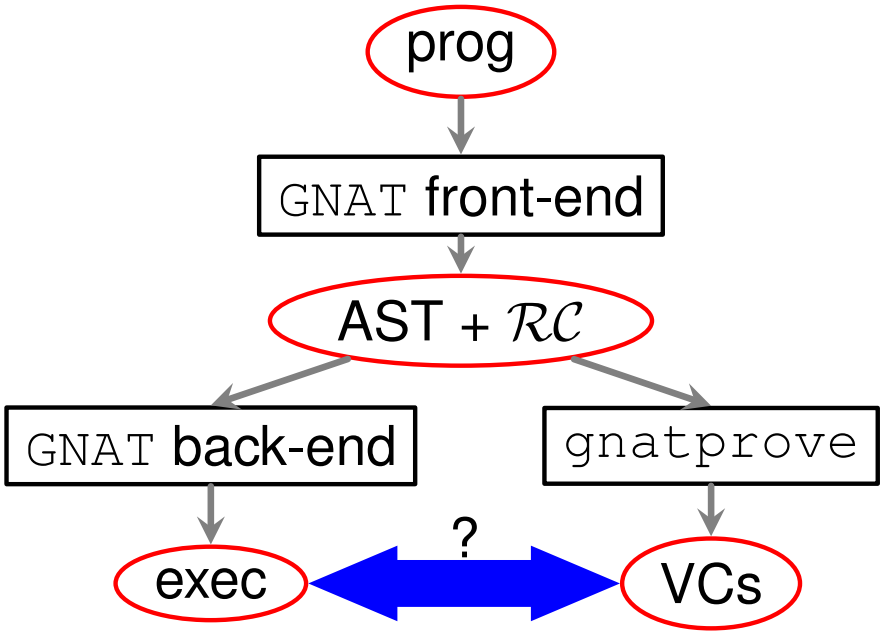
Not sufficient

Correctness GNAT-gnatprove



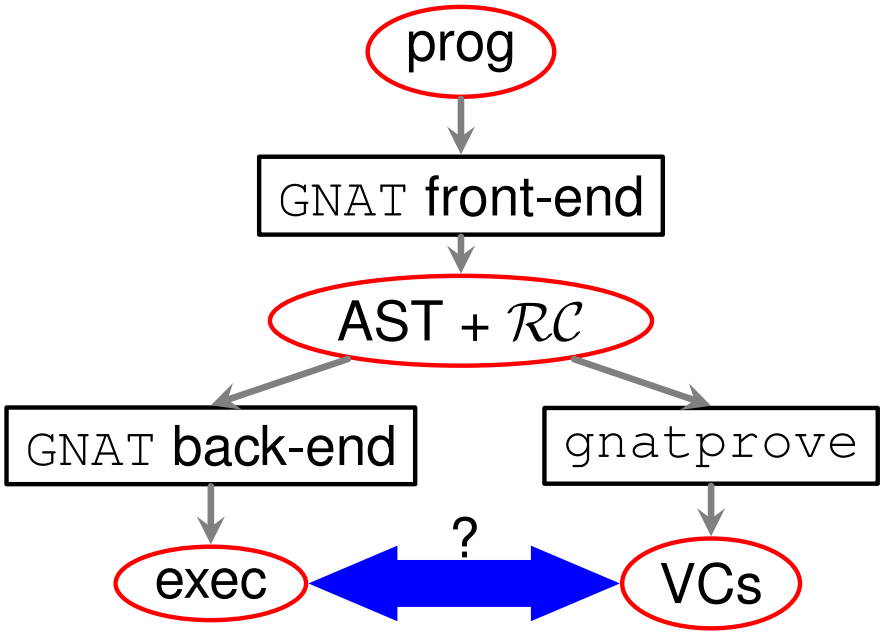
- If $\mathcal{RC}$ incomplete?

Correctness GNAT-gnatprove



- If $\mathcal{RC}$ incomplete?
- Neither detected at runtime nor during anlysis “erroneous program” ou “ bounded errors”

Correctness GNAT-gnatprove



- If $\mathcal{RC}$ incomplete?
- Neither detected at runtime nor during anlysis “erroneous program” ou “ bounded errors”
- Correctness of the (result of the) front-end

Validating Run-time Checks (Z. Zhang)

Two semantics:

- Semantics S_1 with on the fly checks
  - Checks everything (no $\mathcal{RC}$in AST)
  - Reference, validated by experts
- Semantics S_2 with on demand checks
  - Checks only $\mathcal{RC}$
  - “Real”

+ `check_RC: AST+RC -> boolean`
+ proof: $\forall t$ s.t. `check_RC` $t =$ `true`, S_1 and S_2 coincide on t.

S_2 may be optimized

With implicit on the fly run-time checks

$$binop_{RTE1}\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow \bot}{\langle e_1 \bullet e_2, \sigma\rangle \rightsquigarrow \bot}$$

$$binop_{RTE2}\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow v_1 \qquad \langle e_2, \sigma\rangle \rightsquigarrow \bot}{\langle e_1 \bullet e_2, \sigma\rangle \rightsquigarrow \bot}$$

$$binop_2\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow v_1 \qquad \langle e_2, \sigma\rangle \rightsquigarrow v_2 \qquad \neg\mathcal{RC}(v_1, v_2, \bullet)}{\langle e_1 \bullet e_2, \sigma\rangle \rightsquigarrow \bot}$$

$$binop_1\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow v_1 \qquad \langle e_2, \sigma\rangle \rightsquigarrow v_2 \qquad \mathcal{RC}(v_1, v_2, \bullet)}{\langle e_1 \bullet e_2, \sigma\rangle \rightsquigarrow v_1 \bullet v_2}$$

- $\mathcal{RC}$: predicate validated by ADA experts (division by zero, etc)

With explicit checks

$$binop_{RTE1}\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow \bot}{\langle e_1 \bullet^{\mathcal{C}} e_2, \sigma\rangle \rightsquigarrow \bot}$$

$$binop_{RTE2}\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow v_1 \qquad \langle e_2, \sigma\rangle \rightsquigarrow \bot}{\langle e_1 \bullet^{\mathcal{C}} e_2, \sigma\rangle \rightsquigarrow \bot}$$

$$binop_1\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow v_1 \qquad \langle e_2, \sigma\rangle \rightsquigarrow v_2 \qquad \mathcal{C}(v_1, v_2)}{\langle e_1 \bullet^{\mathcal{C}} e_2, \sigma\rangle \rightsquigarrow v_1 \bullet v_2}$$

$$binop_2\ \frac{\langle e_1, \sigma\rangle \rightsquigarrow v_1 \qquad \langle e_2, \sigma\rangle \rightsquigarrow v_2 \qquad \neg\mathcal{C}(v_1, v_2)}{\langle e_1 \bullet^{\mathcal{C}} e_2, \sigma\rangle \rightsquigarrow \bot}$$

- $\mathcal{C}$: tags in the AST, generated by `GNAT` (front-end)
- $\mathcal{C} \equiv \mathcal{RC} \setminus \{\text{optimization}\}$

On the fly vs On demand

```
Inductive eval_stmt:
 symboltable → stack → statement → Return stack → Prop :=
...
```

```
Inductive eval_stmt_x:
 symboltable → stack → statement_x → Return stack → Prop :=
...
```

On the fly vs On demand: binary operations

```
| Eval_E_Binary_Operation_RTE_Bin:
   ∀ st s e1 v1 e2 v2 msg op,
     eval_expr st s e1 (Normal v1) →
     eval_expr st s e2 (Normal v2) →
     do_run_time_check_on_binop op v1 v2 (Exception msg) →
     eval_expr st s (E_Binary_Operation op e1 e2)
                    (Run_Time_Error msg)
```

```
| Eval_E_Binary_Operation_RTE_Bin_X:
   ∀ st s e1 v1 e2 v2 checkflags op msg,
     eval_expr_x st s e1 (Normal v1) →
     eval_expr_x st s e2 (Normal v2) →
     do_flagged_checks_on_binop checkflags op v1 v2
                                (Exception msg) →
     eval_expr_x st s
       (E_Binary_Operation_X op e1 e2 checkflags)
       (Run_Time_Error msg)
```

On the fly vs On demand: binary operations

```
|Eval_E_Binary_Operation:
  ∀ st s e1 v1 e2 v2 op v,
   eval_expr st s e1 (Normal v1) →
   eval_expr st s e2 (Normal v2) →
   do_run_time_check_on_binop op v1 v2 Success →
   Val.binary_operation op v1 v2 = Some v →
   eval_expr st s (E_Binary_Operation op e1 e2) (Normal v)
```

```
|Eval_E_Binary_Operation_X:
  ∀ st s e1 v1 e2 v2 checkflags op v,
   eval_expr_x st s e1 (Normal v1) →
   eval_expr_x st s e2 (Normal v2) →
   do_flagged_checks_on_binop checkflags op v1 v2 Success →
   Val.binary_operation op v1 v2 = Some v →
   eval_expr_x st s (E_Binary_Operation_X op e1 e2 checkfla
                     (Normal v)
```

On the fly vs On demand: Range check for assignment

```
| Eval_S_Assignment_Range_RTE:
   ∀ st s e v x t l u,
     eval_expr st s e (Normal (BasicV (Int v))) →
     extract_subtype_range st x (Range l u) →
     do_range_check v l u (Exception RTE_Range) →
     eval_stmt st s (S_Assignment x e)
                    (Run_Time_Error RTE_Range)
```

```
| Eval_S_Assignment_RTE_X:
   ∀ e st s msg x,
     ¬(List.In Do_Range_Check (exp_check_flags e)) →
     eval_expr_x st s e ...

| Eval_S_Assignment_Range_RTE_X:
   ∀ e flgs st s v x t l u,
     exp_check_flags e = Do_Range_Check ⊎ flgs →
     eval_expr_x st s (update_exp_check_flags e flgs)
                      (Normal (BasicV (Int v))) →
     extract_subtype_range_x st x (Range_X l u) → ...
```

On the fly vs On demand: Range check for assignment

```
| Eval_S_Assignment_Range:
   ∀ st s e v x t l u s1,
     eval_expr st s e (Normal (BasicV (Int v))) →
     fetch_exp_type x st = Some t →
     extract_subtype_range st t (Range l u) →
     do_range_check v l u Success →
     storeUpdate st s x (BasicV (Int v)) s1 →
     eval_stmt st s (S_Assignment x e) s1
```

```
| Eval_S_Assignment_Range_X:
   ∀ e cks1 cks2 st s v x t l u s1,
    exp_check_flags e = Do_Range_Check ⊎ flgs →
    eval_expr_x st s (update_exp_check_flags e flgs)
                     (Normal (BasicV (Int v))) →
    extract_subtype_range_x st x (Range_X l u) →
    do_range_check v l u Success →
    storeUpdate_x st s x (BasicV (Int v)) s1 →
    eval_stmt_x st s (S_Assignment_X x e) s1
```

Properties of the ast validator

```
Definition check_generator: statement → statement_x.

Lemma statement_checks_soundness:
 ∀ st s stmt s' st',
   eval_stmt st s stmt s' →
   check_generator stmt = stmt' →
   eval_stmt_x st s stmt' s'.

Definition check_RC stmt_x =
  check_generator(noflags stmt_x) = stmt_x.
```

Run-Time Checks covered

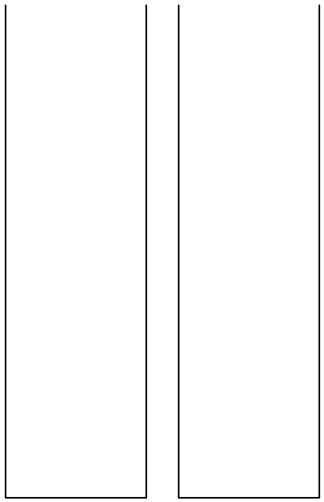
- Very few, proof of concept
- Division by zero (integers)
- Overflows (integers,arrays)

Table des matieres

Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

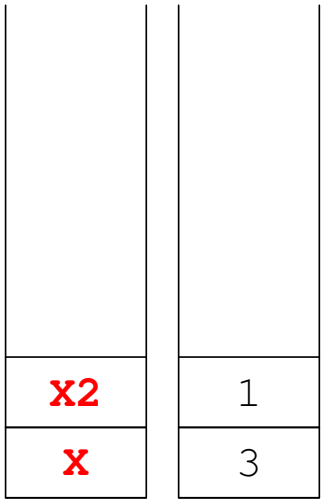
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

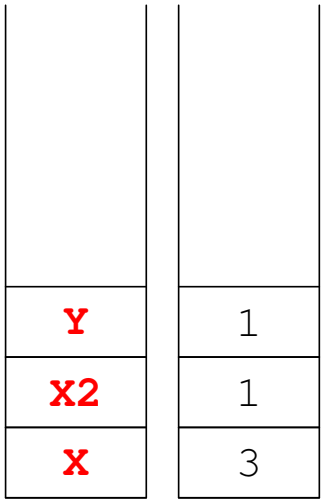
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
    X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

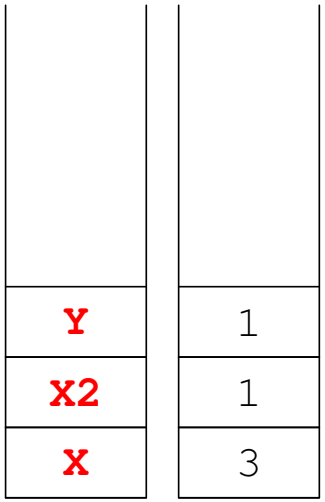
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
    | if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

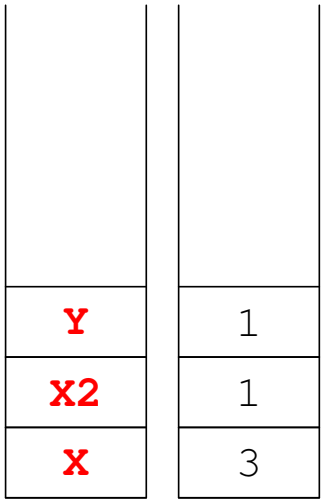
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

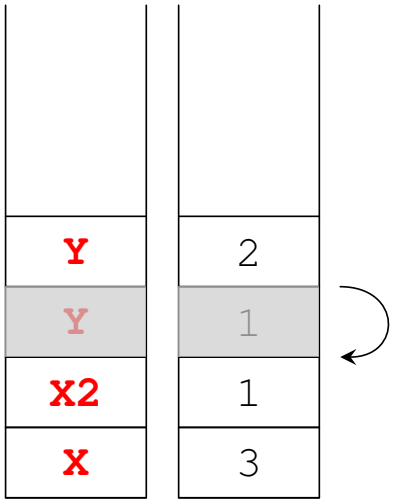
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
    X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

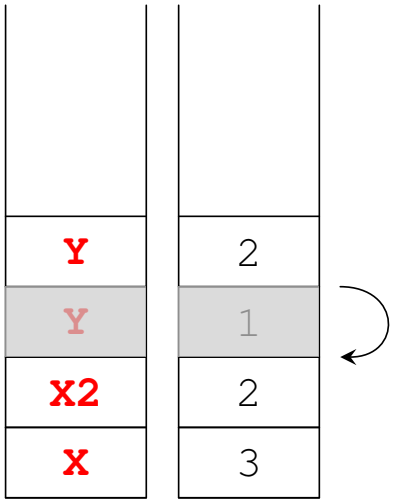
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
    | if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

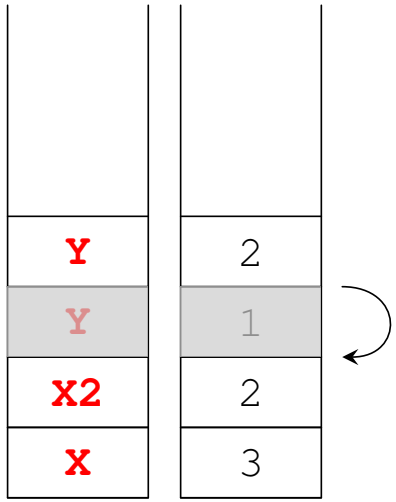
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

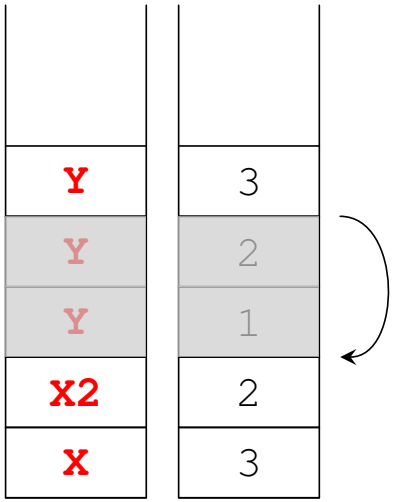
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
    X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

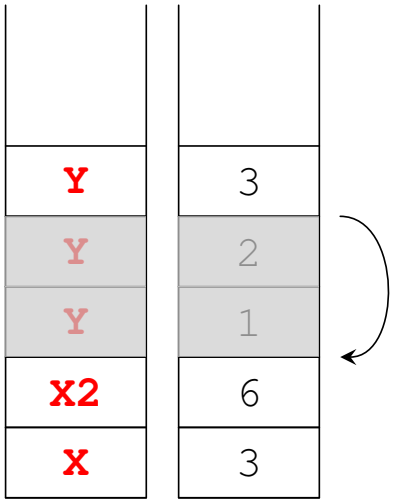
Dynamic context



Nested recursive procedures: P-machine

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
    | if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

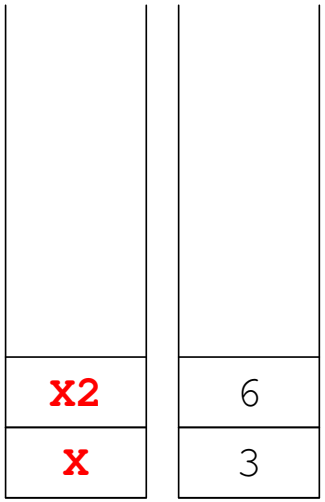
Dynamic context



Nested recursive procedures: P-machine

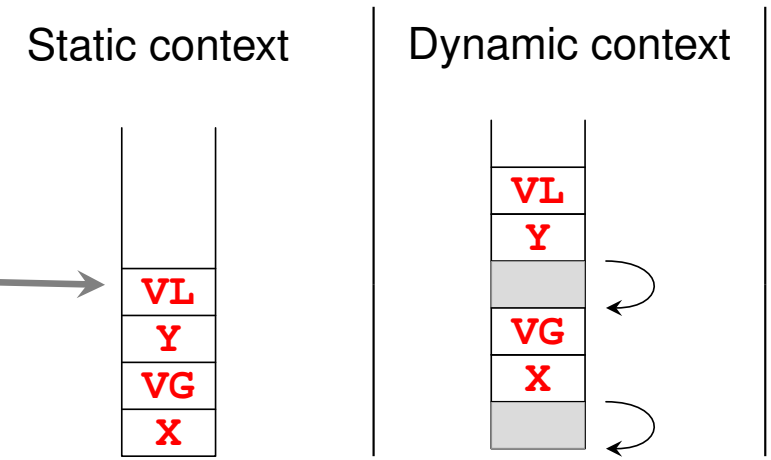
```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

Dynamic context



Nested recursive procedures

```
procedure G(X:INTEGER) is

  VG: INTEGER;

  procedure L(Y:INTEGER) is
    VL: INTEGER
  begin
    ..L(e)..
  end L;

begin
  ...
end G;
```

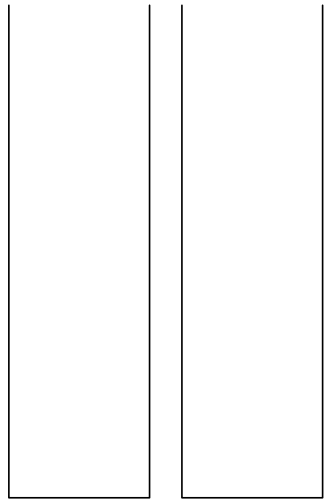


- Dynamic environment $\not\simeq$ static one
- But Dynamic environment $\simeq$ static "visible" one
- How to present a **reference** semantics?

Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

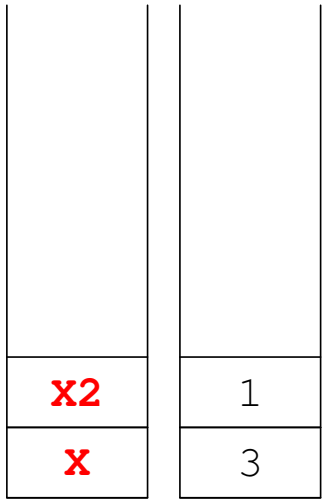
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

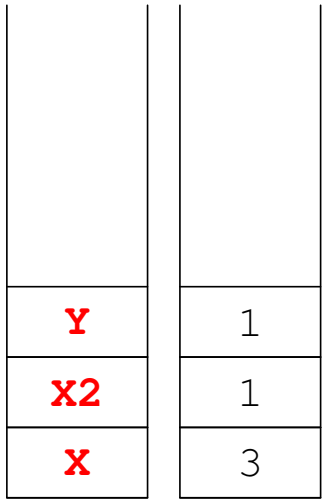
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
    X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

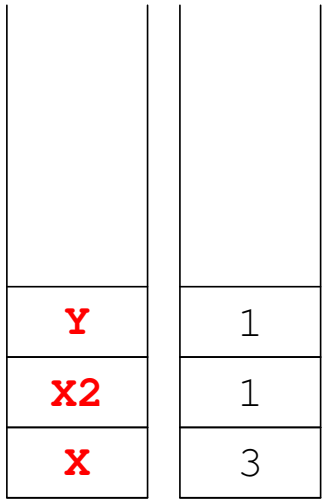
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
    | if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

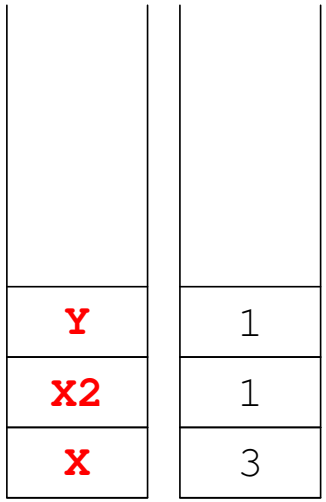
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

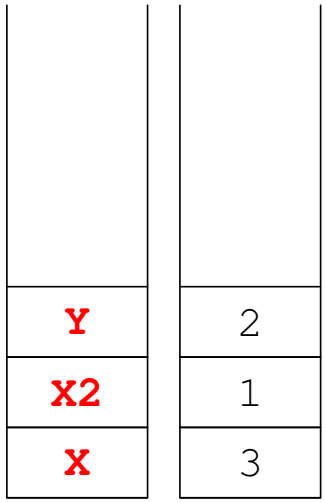
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
    X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

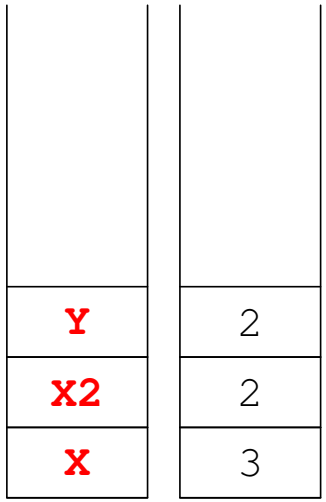
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
    | if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

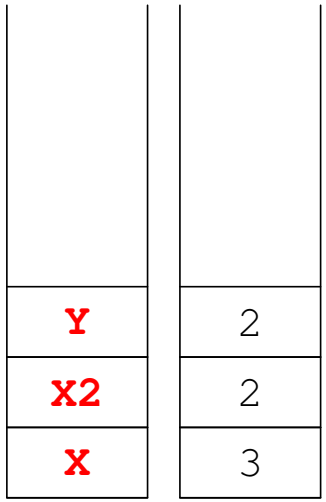
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

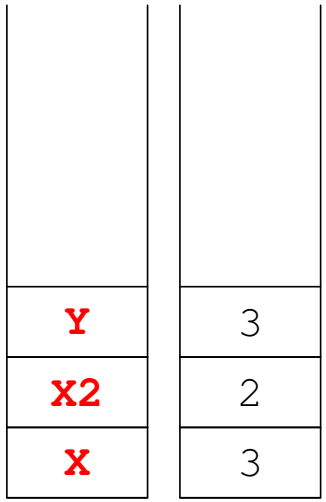
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
    X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

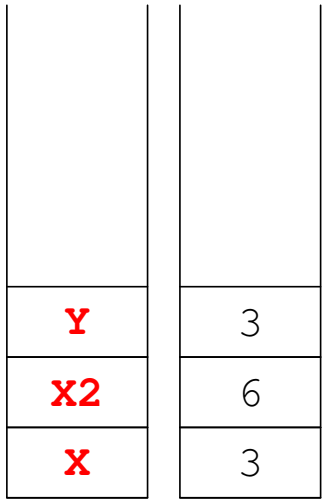
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
    | if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

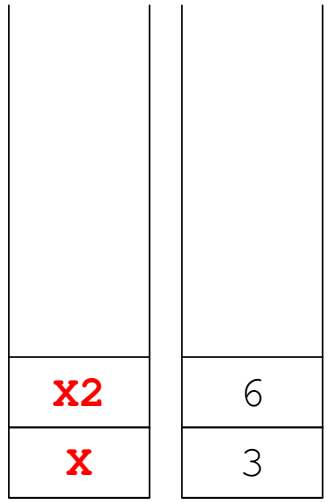
Dynamic context



Nested recursive procedures: Reference

```
procedure G (X: in Integer) is
  X2:  Integer := 1;

  procedure L (Y: in Integer) is
  begin
     X2 := X2 * Y;
     if Y < X then L(Y+1); end if;
  end;

begin
  L(X2);
  Put(X2);
end;

G(3);
```

Dynamic context

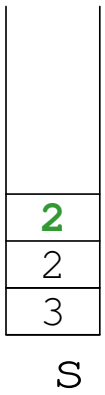


Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
```

```
eval_stmt st s (S_Procedure_Call p args) s4
```

```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```

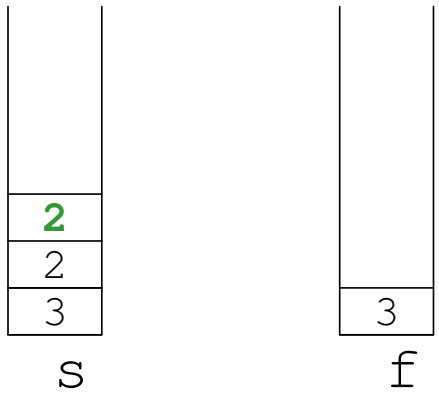


Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
```

```
eval_stmt st s (S_Procedure_Call p args) s4
```

```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```

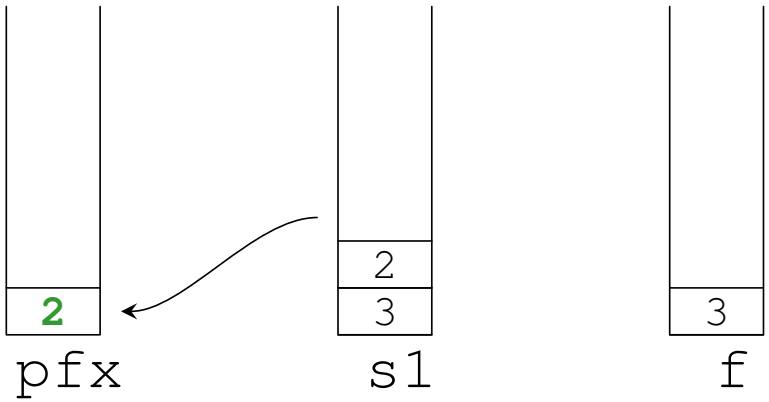


Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
```

```
eval_stmt st s (S_Procedure_Call p args) s4
```

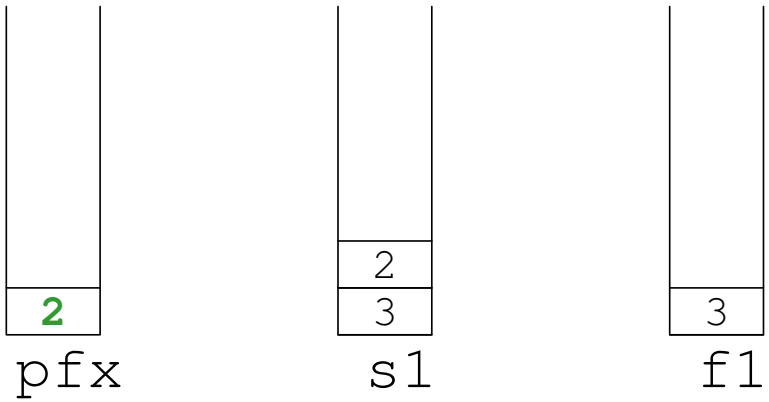
```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```



Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
eval_decl st s1 f (declarative_part pb) (Normal f1) →


eval_stmt st s (S_Procedure_Call p args) s4
```

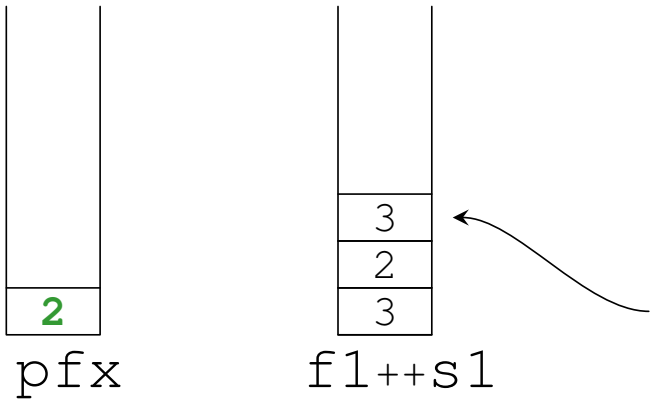
```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```



Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
eval_decl st s1 f (declarative_part pb) (Normal f1) →
eval_stmt st (f1 ++ s1) (statements pb) (Normal s2) →


eval_stmt st s (S_Procedure_Call p args) s4
```

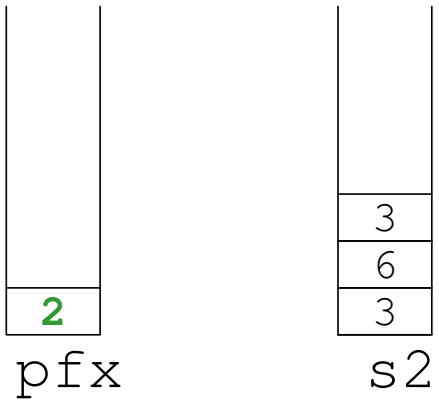
```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
 | X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```



Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
eval_decl st s1 f (declarative_part pb) (Normal f1) →
eval_stmt st (f1 ++ s1) (statements pb) (Normal s2) →


eval_stmt st s (S_Procedure_Call p args) s4
```

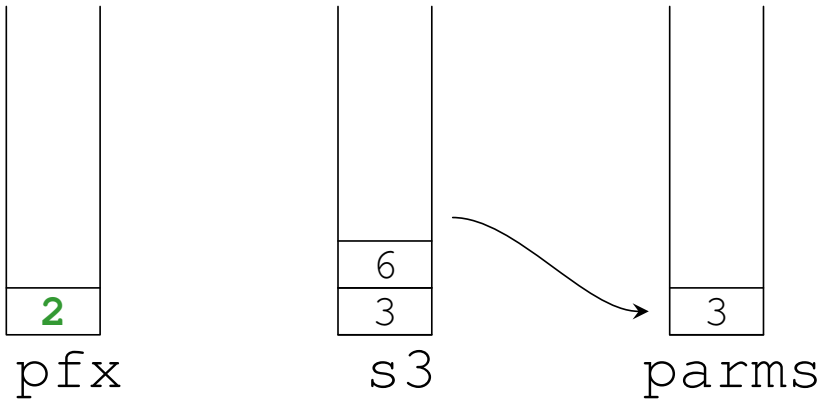
```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```



Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
eval_decl st s1 f (declarative_part pb) (Normal f1) →
eval_stmt st (f1 ++ s1) (statements pb) (Normal s2) →
s2 = locls ++ parms ++ s3 →

eval_stmt st s (S_Procedure_Call p args) s4
```

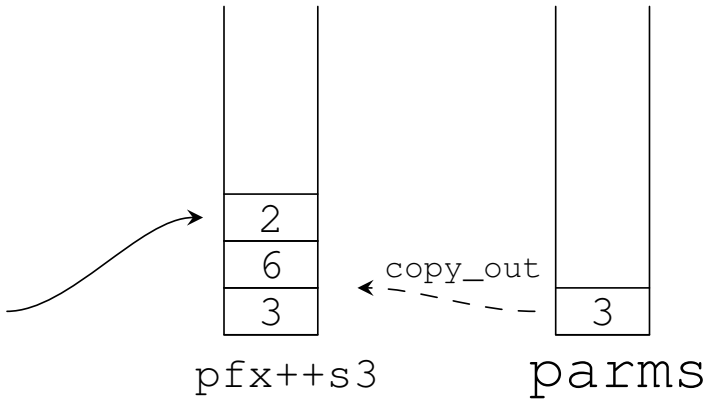
```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```



Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
eval_decl st s1 f (declarative_part pb) (Normal f1) →
eval_stmt st (f1 ++ s1) (statements pb) (Normal s2) →
s2 = locls ++ parms ++ s3 →
copy_out st (pfx ++ s3) parms (parameters pb) args s4 →
eval_stmt st s (S_Procedure_Call p args) s4
```

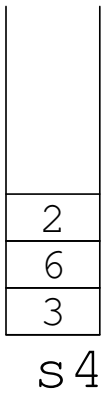
```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```



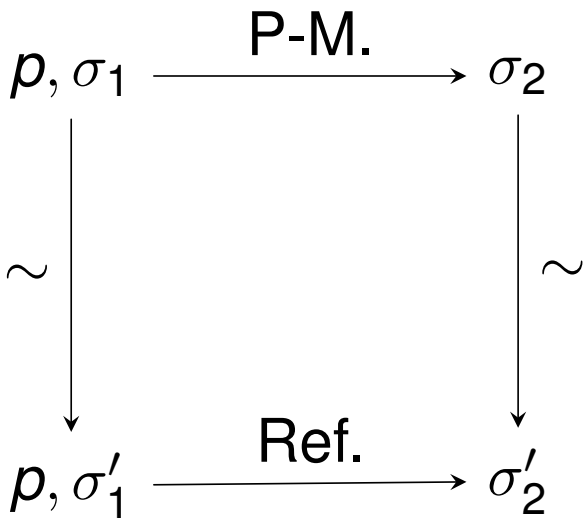
Nested recursive procedure

```
fetch_proc p st = (proc_lvl,pb) →
copy_in st s (parameters pb) args (Normal f) →
cut_until s proc_lvl pfx s1 →
eval_decl st s1 f (declarative_part pb) (Normal f1) →
eval_stmt st (f1 ++ s1) (statements pb) (Normal s2) →
s2 = locls ++ parms ++ s3 →
copy_out st (pfx ++ s3) parms (parameters pb) args s4 →
eval_stmt st s (S_Procedure_Call p args) s4
```

```
procedure G (X: in Integer) is
 X2:  Integer := 1;
 procedure L (Y: in Integer) is
 begin
  X2 := X2 * Y;
  if Y < X then L(Y+1);
  end if;
 end;
...
```

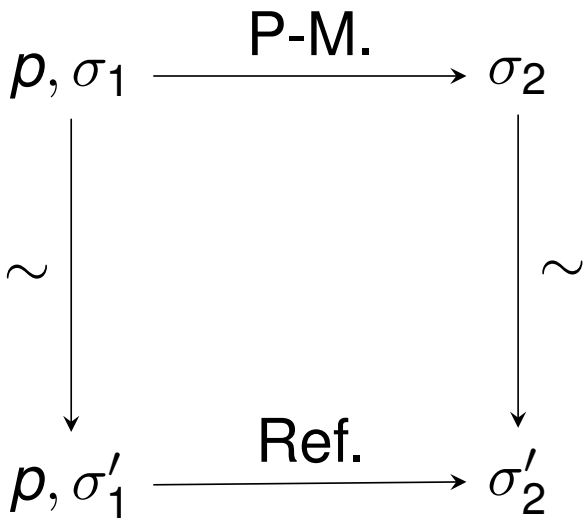


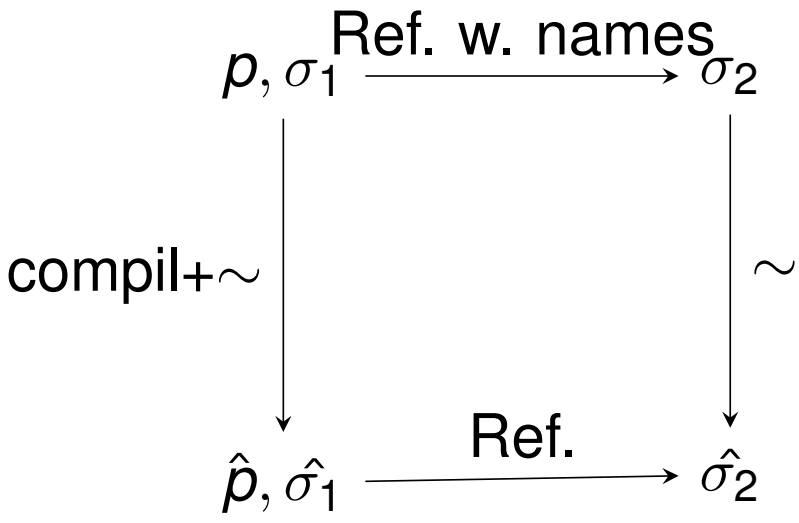
Equivalence with P-machine (T. Crolard)



- Coq development
- Small toy language
- Correctness P-Machine vs Ref. Semantics

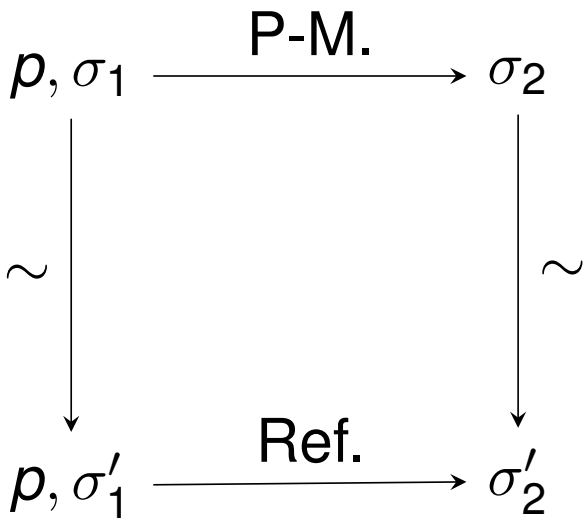
Equivalence with P-machine (T. Crolard)

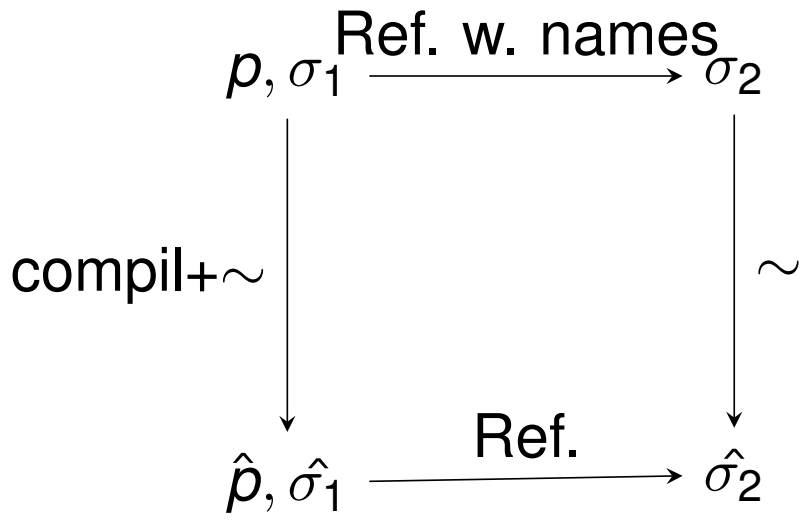




- Coq development
- Small toy language
- Correctness P-Machine vs Ref. Semantics
- + compilation P-machine with var. names $\rightsquigarrow$ P-machine (offsets)

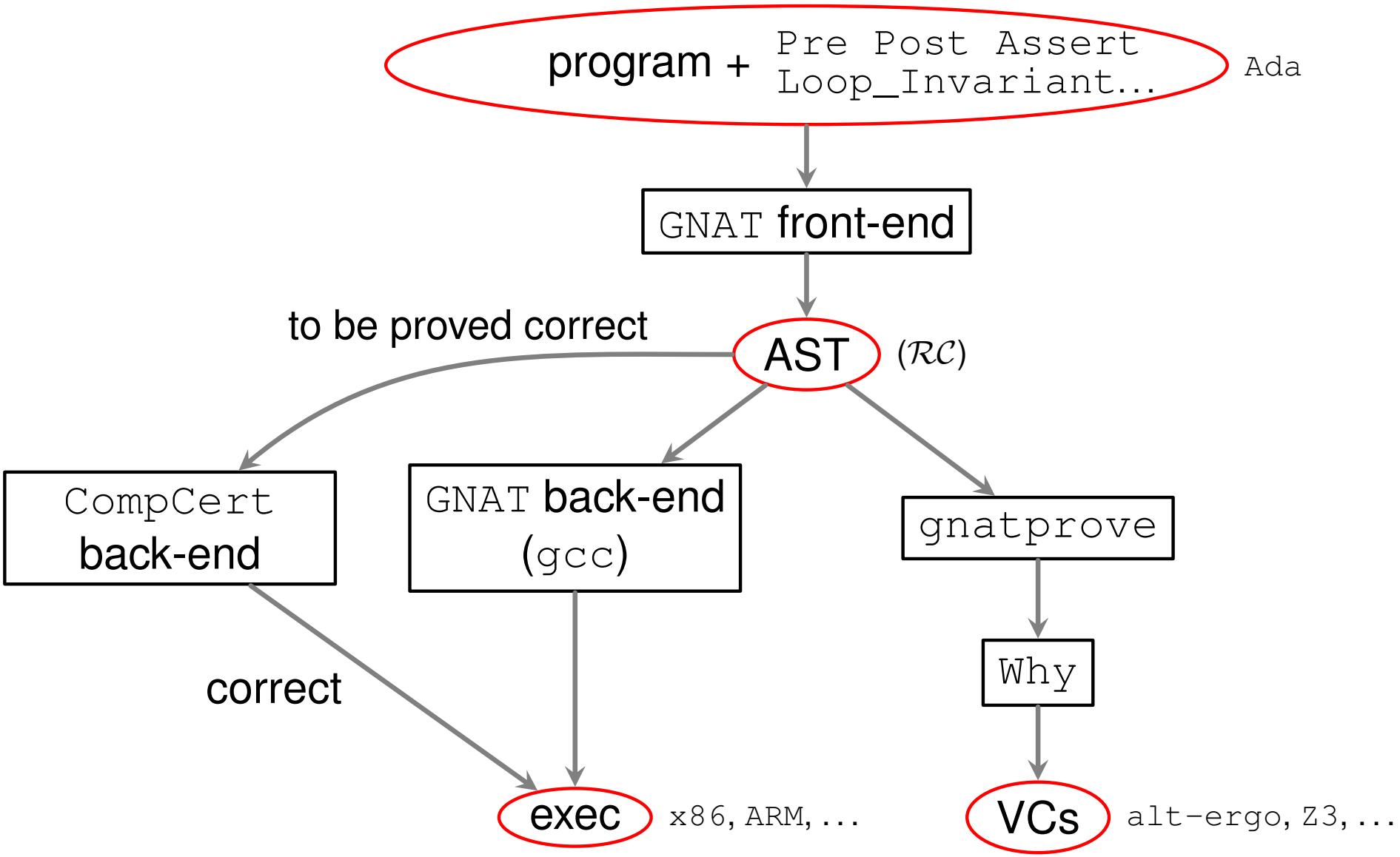
Equivalence with P-machine (T. Crolard)





- One functorized semantics, three stack modules
- 1376 LOC statements, 1777 LOC proofs
- Completeness under way (T.Crolard, M.V. Aponte, J.Lawall)
- Extend it to the whole SPARK semantics?

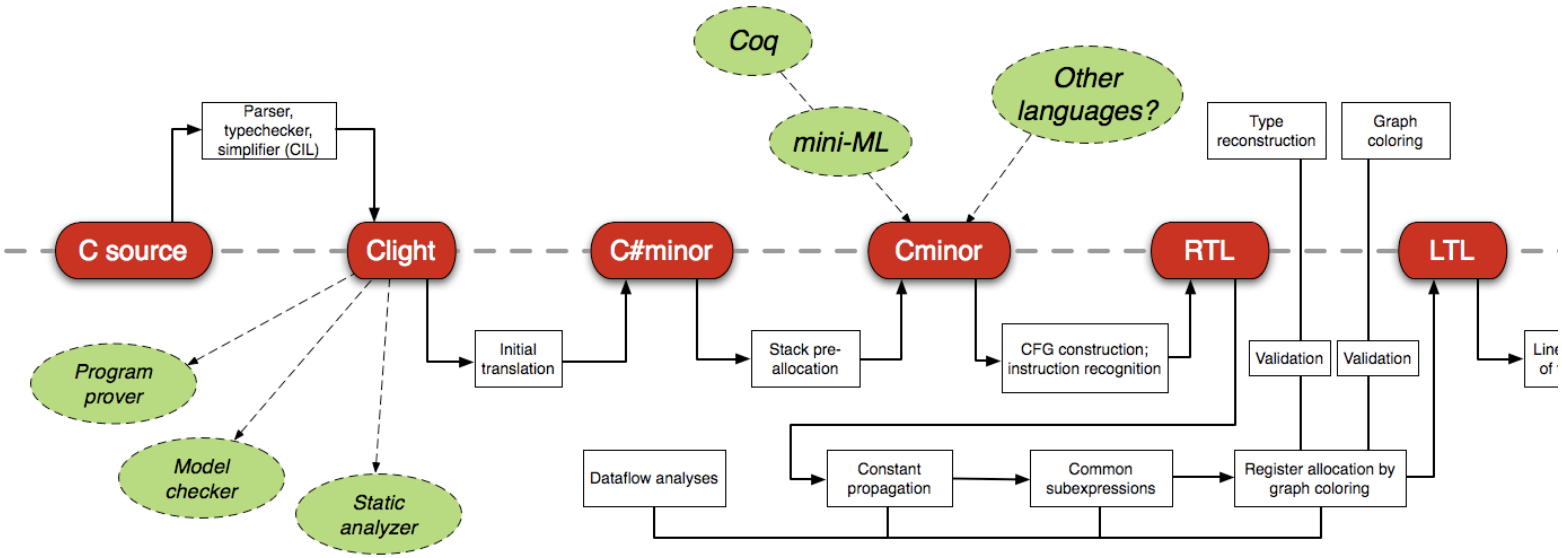
CompCert frontend



Why CompCert?

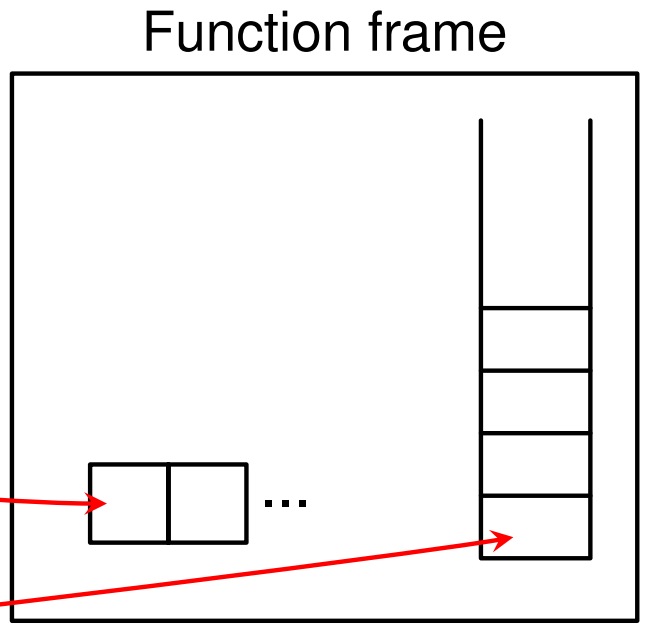
- Having a (partially) proved compilation chain
- Testing semantics for pure SPARK?
- How flexible is CompCert?
- Nested (recursive) procedures are fun ($\longleftarrow$ see later), despite easier than in Ada

Target language: `Cminor`



- `C#minor` too loose memory model
- `RTL` to low level
- `Cminor` all what we need

Cminor

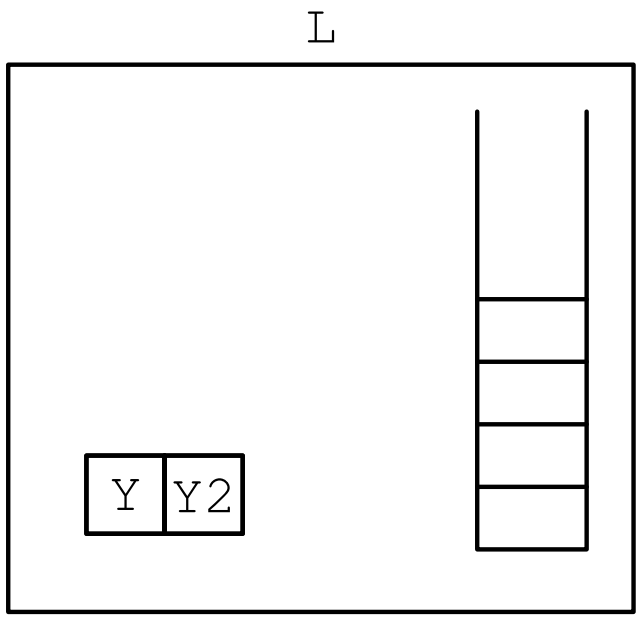


Each function leaves in a separate frame

- Local temporary variables (parameters + some local variables)
- Local stack (other variables)
- Initially: empty stack, all arguments in temporary variables
- C frontend: prelude to copy some variables in the stack

Nested procedures in Cminor

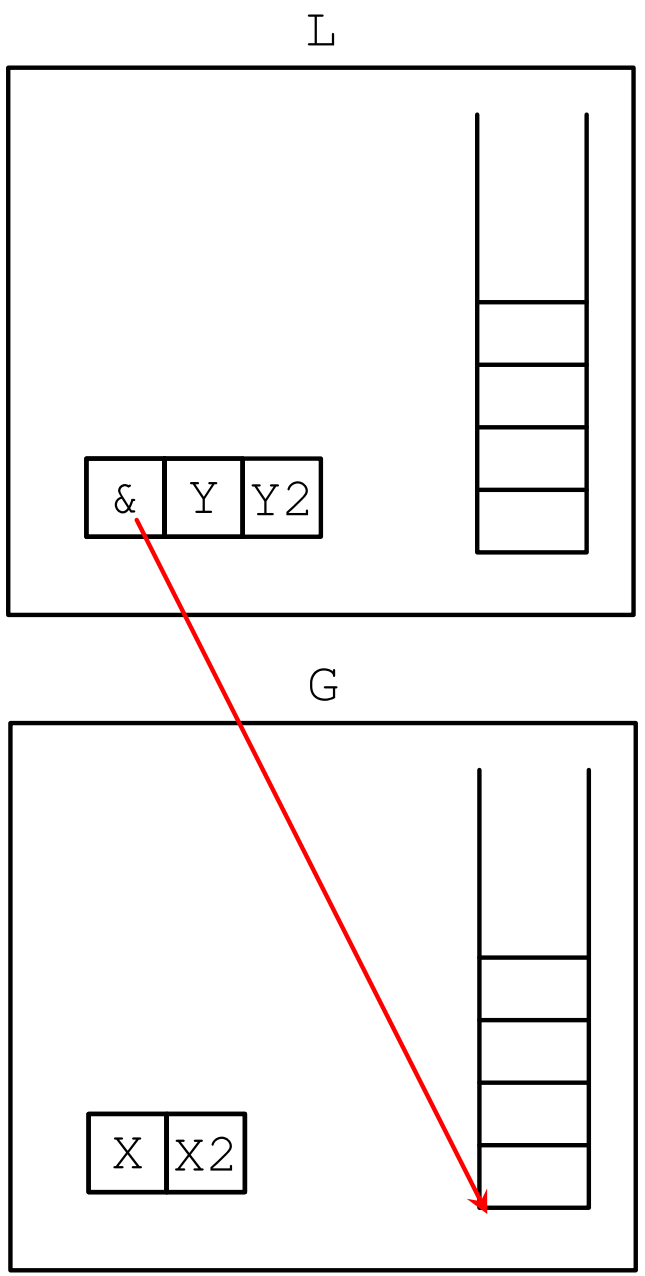
```
procedure G (X: in out Integer) is
   X2:  Integer := 1;
   procedure L (Y: in Integer) is
      Y2: Integer := 1;
   begin
      X2 := X2 * Y;
      if Y < X then L(Y+1); end if;
   end;
begin
   L(X2);
   X := X2;
end;
```



Nested procedures in Cminor

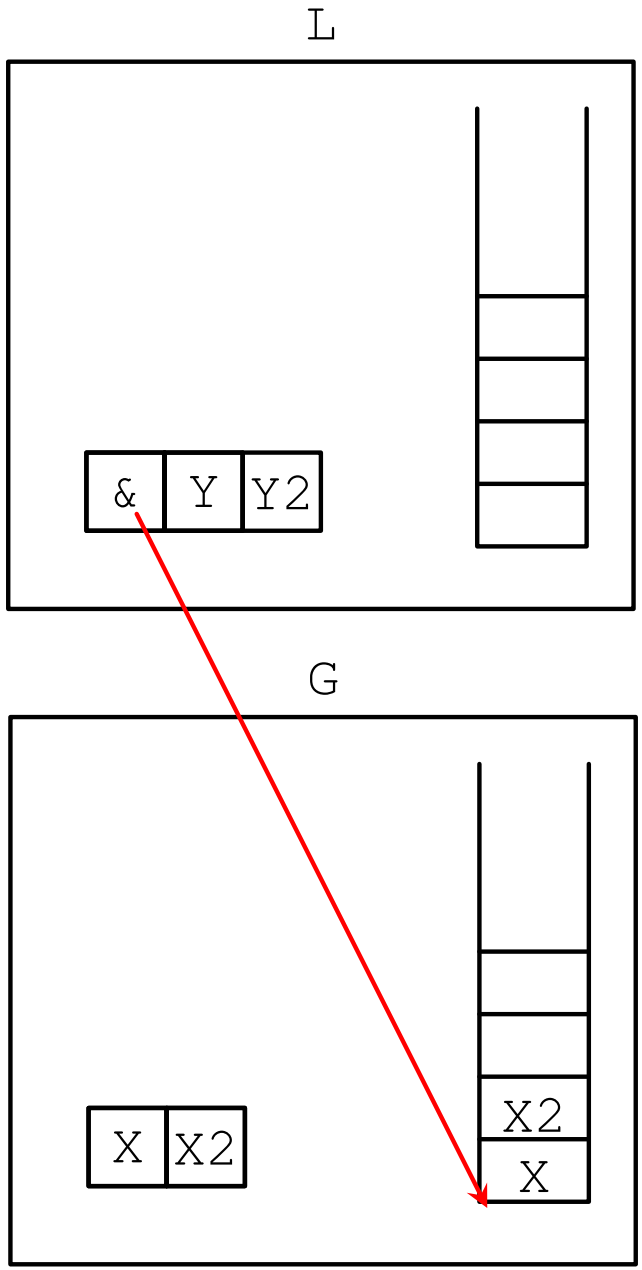
```
procedure G (X: in out Integer) is
   X2:  Integer := 1;
   procedure L (Y: in Integer) is
      Y2: Integer := 1;
   begin
      X2 := X2 * Y;
      if Y < X then L(Y+1); end if;
   end;
begin
   L(X2);
   X := X2;
end;
```

- `&` = ptr to `G`'s stack



Nested procedures in Cminor

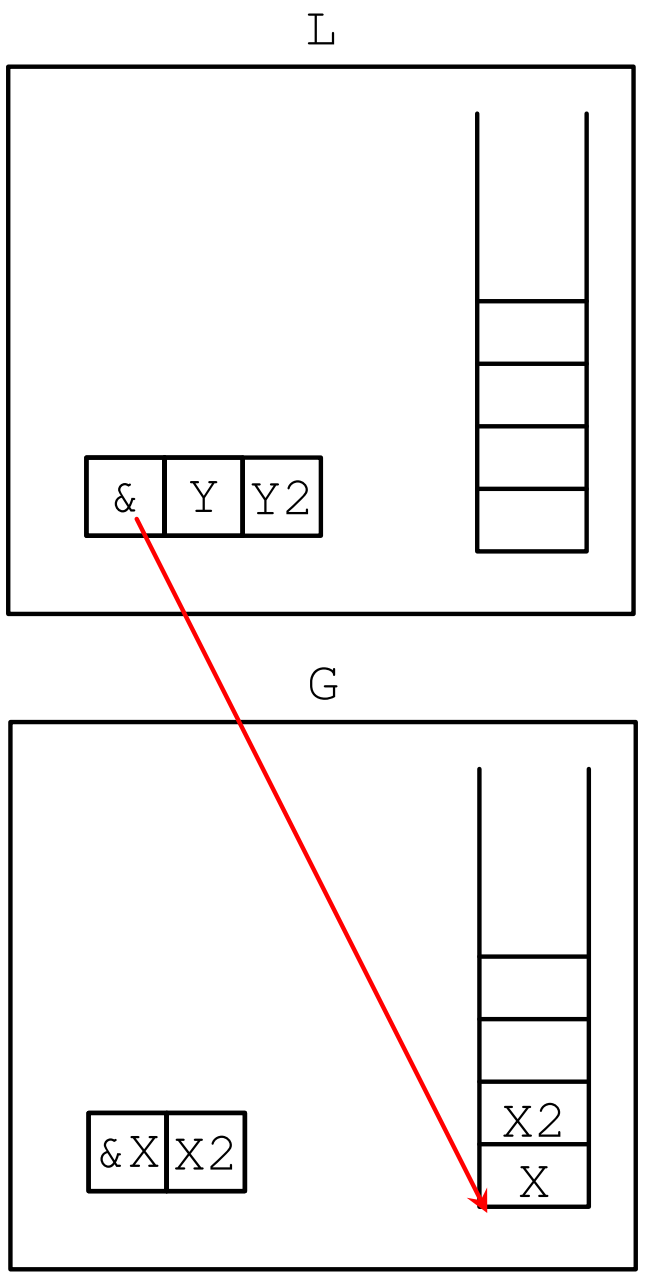
```
procedure G (X: in out Integer) is
   X2:  Integer := 1;
   procedure L (Y: in Integer) is
      Y2: Integer := 1;
   begin
      X2 := X2 * Y;
      if Y < X then L(Y+1); end if;
   end;
begin
   L(X2);
   X := X2;
end;
```



- `&` = ptr to `G`'s stack
- Needed copies on stack: prelude

Nested procedures in Cminor

```
procedure G (X: in out Integer) is
   X2:  Integer := 1;
   procedure L (Y: in Integer) is
      Y2: Integer := 1;
   begin
      X2 := X2 * Y;
      if Y < X then L(Y+1); end if;
   end;
begin
   L(X2);
   X := X2;
end;
```



- `&` = ptr to `G`'s stack
- Needed copies on stack: prelude
- **`out`** parameters: postlude

Example

```
"G"('X') : int -> void
{
  stack 8;
  var 'X2';
  int32[&0 + 0] = int32['X'];
  int32[&0 + 4] = 1;
  "L"(&0, int32[&0 + 4]) : int -> int -> void;
  int32[&0 + 0] = int32[&0 + 4];
  int32['X'] = int32[&0 + 0];
}
"L"('CHAIN', 'Y') : int -> int -> void
{
  stack 12;
  var 'Y2';
  int32[&0] = 'CHAIN';
  int32[&0 + 4] = 'Y';
  int32[&0 + 8] = 1;
  int32[int32[&0] + 4] = int32[int32[&0] + 4] * int32[&0 + 4];
  if (int32[&0 + 4] < int32[int32[&0] + 0] != 0) {
    "L"(int32[&0], int32[&0 + 4] + 1) : int -> int -> void;
  }
}
```

Demo CompCert

Future Works

- Annotations, contracts
- Prove CompCert compilation
- Prove loop invariants patterns [FM2012] (M-V. Aponte)
- Test and proof, a.k.a. mixing memory models of SPARK and Ada
- etc
- Thanks!