Renaming Global Variables in C
Mechanically Proved Correct

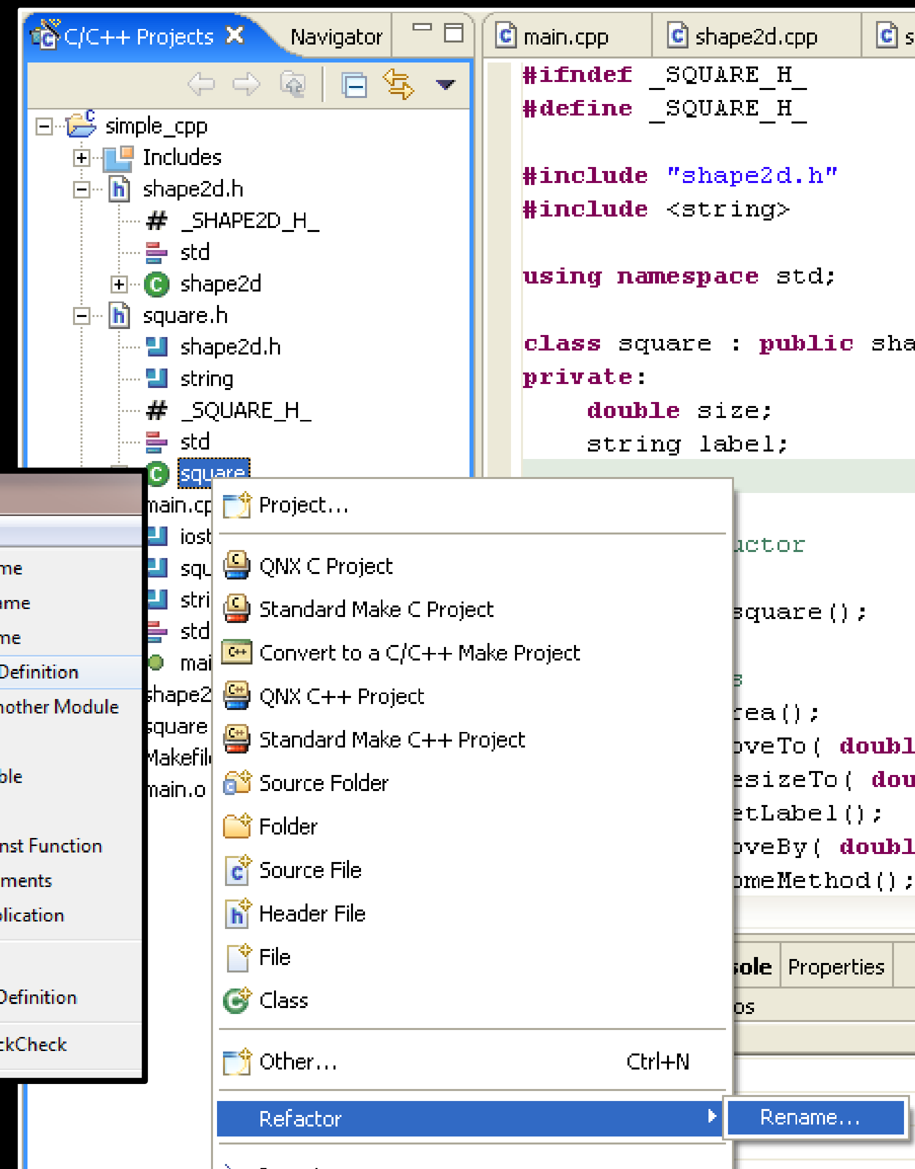
Julien Cohen
Université de Nantes
(Polytech Nantes, LINA, ASCOLA...)

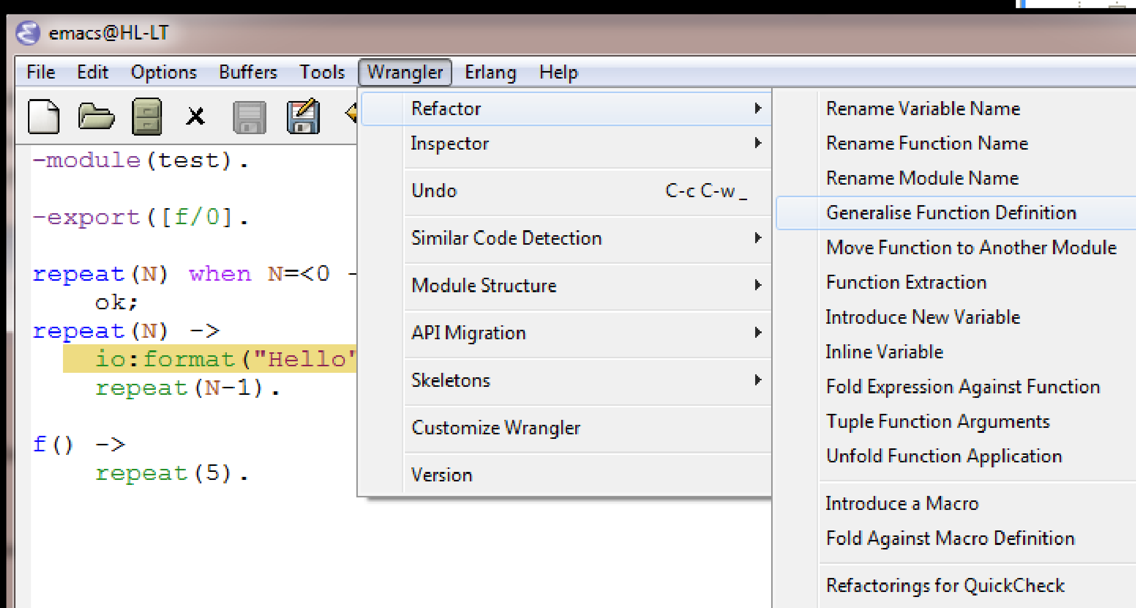
VPT 2016
Fourth International Workshop on
Verification and Program Transformation

Assisted Refactoring

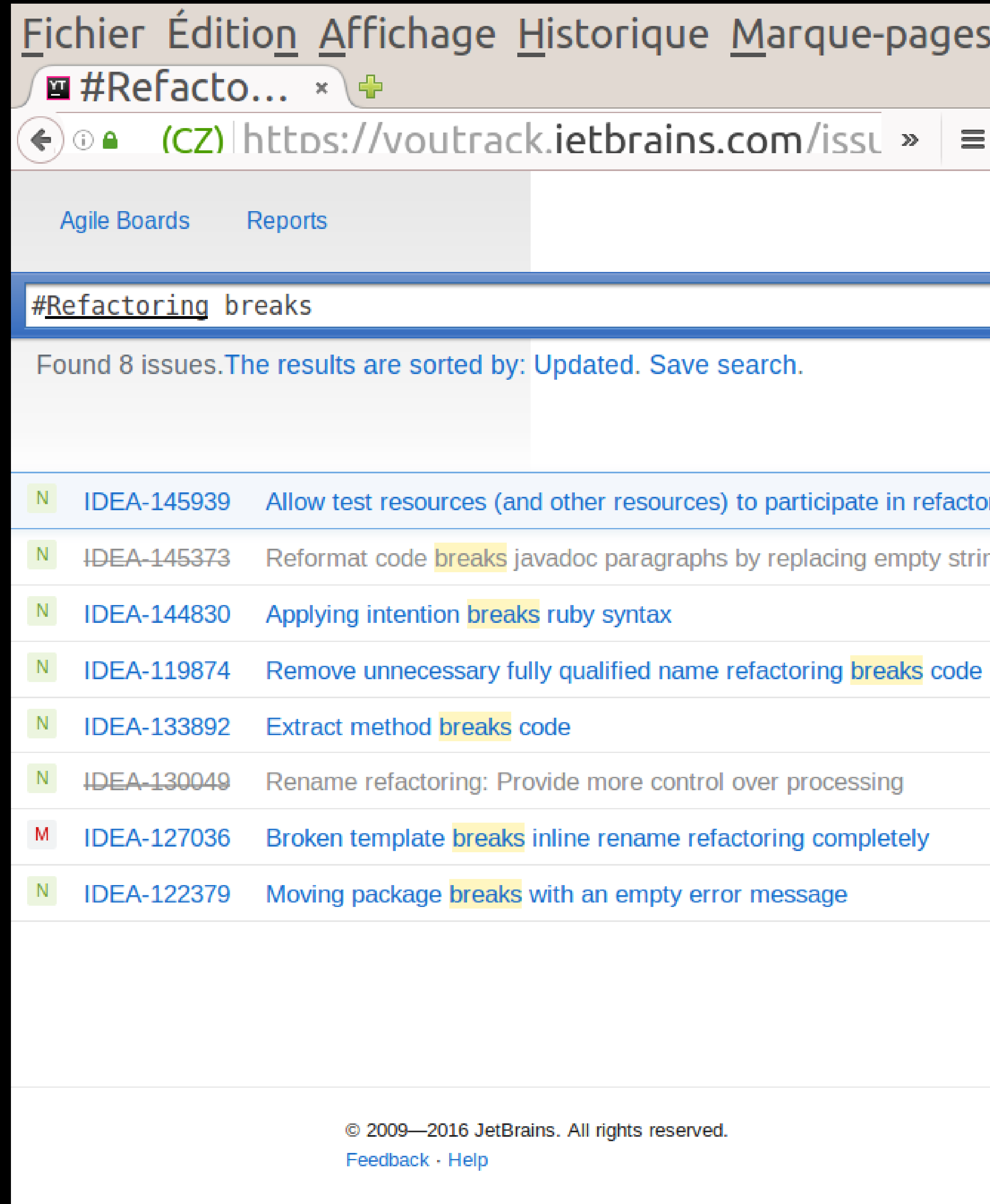
A lot of **tools** :

- Eclipse
- IntelliJ Idea
- Haskell Refactorer / Wrangler / RefactorErl
- ...





A lot of **bugs**



Because of
so **many particular cases** and interactions.

→ difficult to test.

Solution :
a **correct by construction** tool

Difficulties for mechanization :

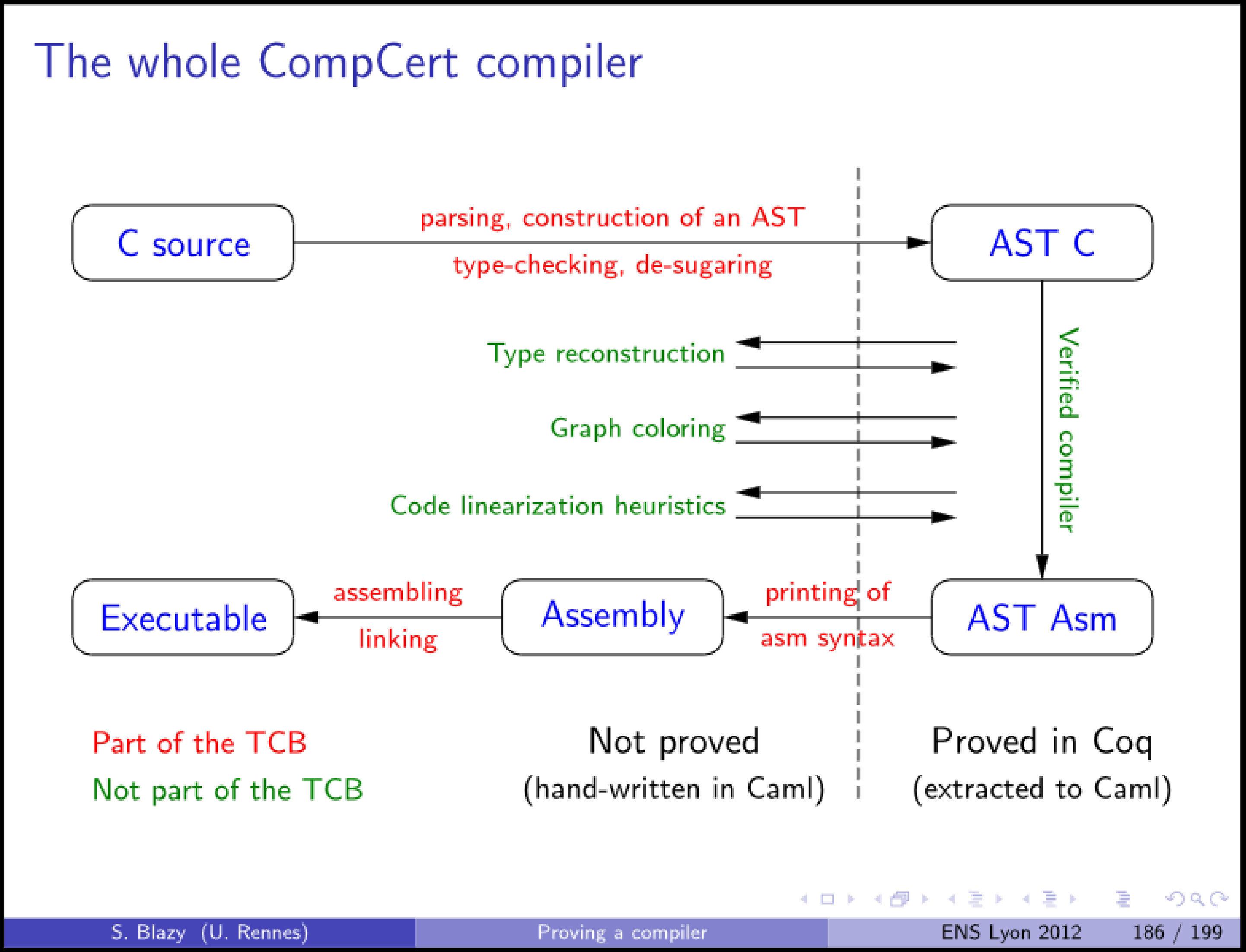
- Must **completely prove correctness**.
- Based on **a complete semantics** :
  - Lambda-calculus ? [Related Work]
  - Featherweight Java ?
  - **CompCert C**

CompCert C

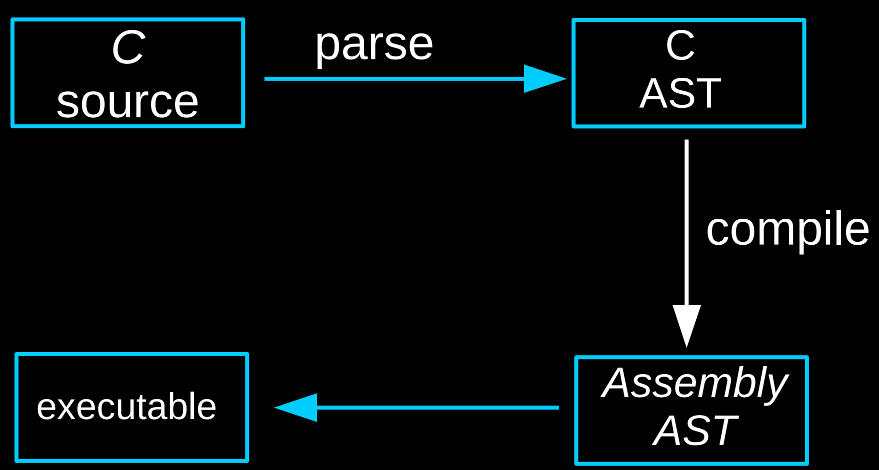
[X. Leroy, S. Blazy...]

[2004 - 2016]

A compiler for C
proved correct
in Coq

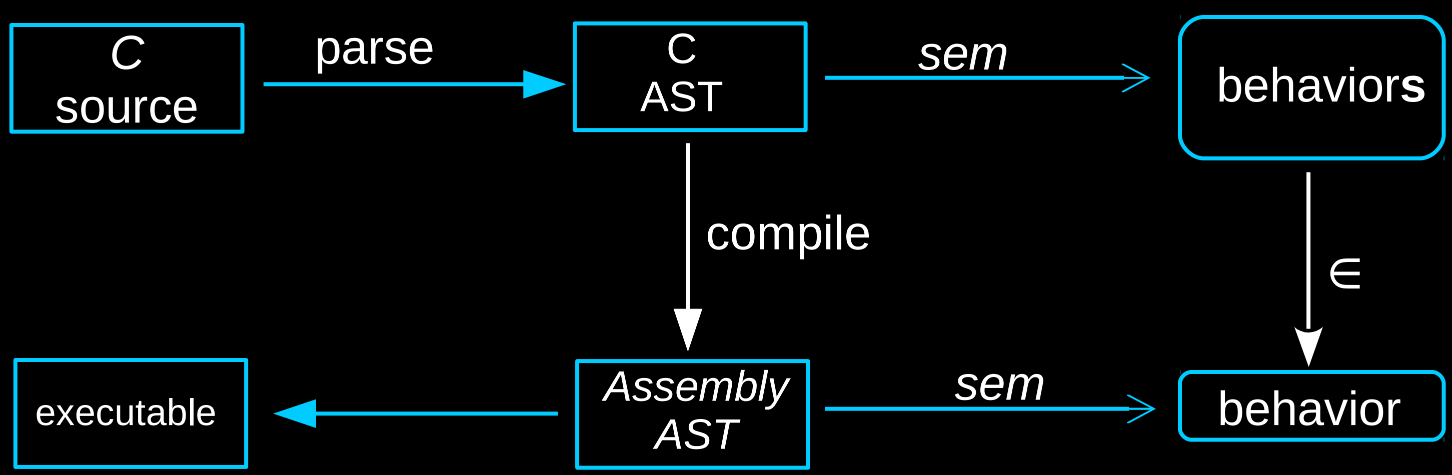


Correct compilation (CompCert C)

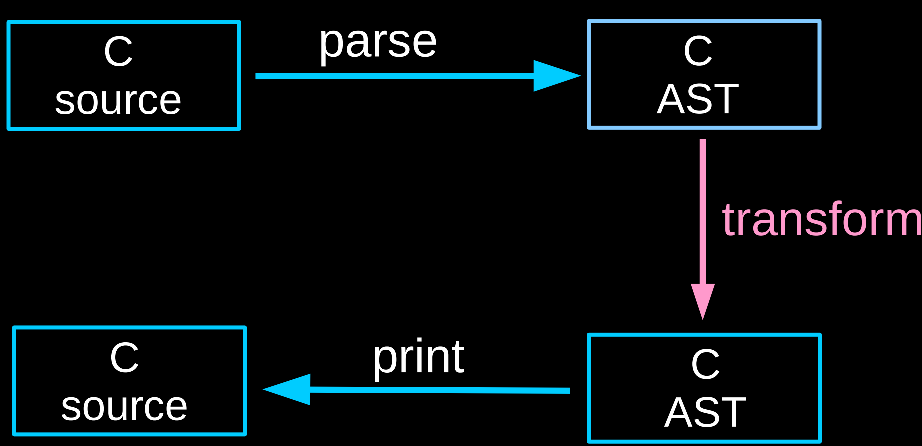


Correct compilation

```
program_behavior ::=
   | Terminates trace int
   | Diverges trace
   | Reacts traceinf
   | Goes_wrong trace
```

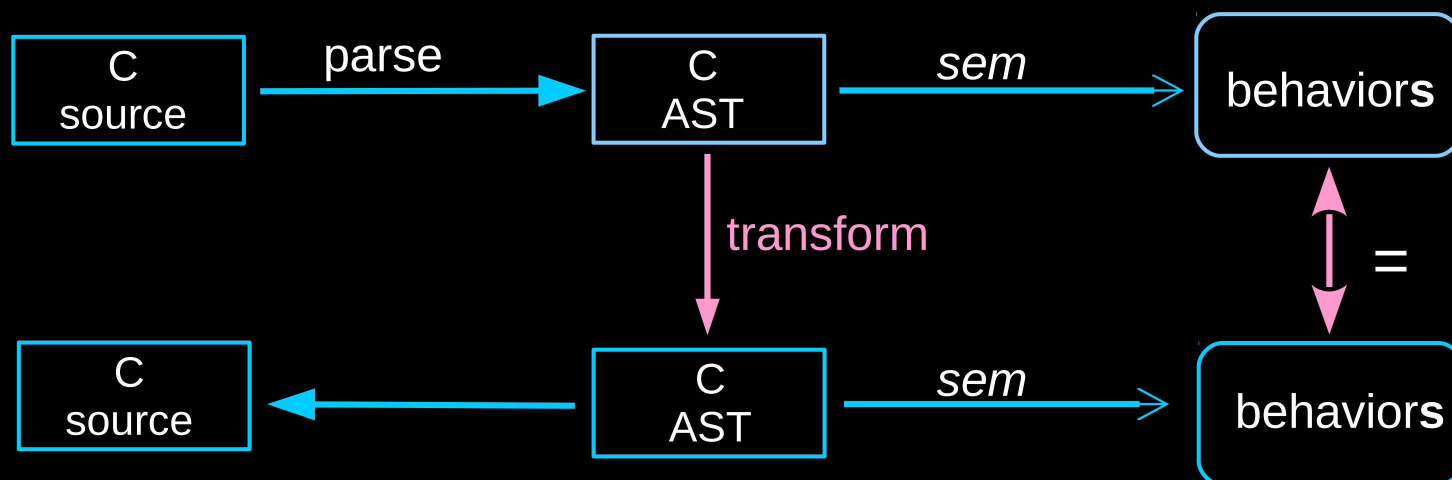


Correct compilation
Correct refactoring



Correct compilation
Correct refactoring

```
program_behavior ::=
   | Terminates trace int
   | Diverges trace
   | Reacts traceinf
   | Goes_wrong trace
```



What you find in Compcert C ?

- **Data structures for syntax trees**
  expr (22), statement (13)…
- **Data structures for semantic constructions**
  continuations (15), states (4), contexts (22), environments, events, memory, behaviors…
- **Relations for small-step semantics**
  reduction (23), transitions (42)…
- **Compilation transformations, proofs...**

What you find in Compcert C ?

```
expr ::=
  | Val val
  | Var ident
  | Field expr ident
  | ValOf expr
  | Deref expr
  | AddrOf expr
  | UnOp unary_op expr
  | BinOp binary_op expr
  | Cast expr
  | SeqAnd expr expr
```

```
Reductions

Inductive
expr -> r
   | red_
        de
        rr
```

```
Contexts

Inductive
(expr ->
   | ctx_t
        con
   | ctx_d
```

```
Environments

Record t
  genv_s
```

```
Transitions

Inductive sstep: sta

   | step_do_1: foral
       sstep (State f
```

```
statement ::=
  | Skip
  | Do expr
  | Sequence s
  | Ifthenelse
  | While expr
  | Dowhile ex
  | For stateme
  | Break
  | Continue
  | Return exp
  | Switch exp
  | Label label
e | Goto label

labeled_stateme
```

```
cont ::=
  | Kstop
  | Kdo cont
  | Kseq statement c
  | Kifthenelse stat
  | Kwhile1 expr sta
  | Kwhile2 expr sta
  | Kdowhile1 expr statement cont
  | Kdowhile2 expr statement cont
  | Kfor2 expr statement statement
  | Kfor3 expr statement statement
  | Kfor4 expr statement statement
  | Kswitch1 labeled_statements co
  | Kswitch2 cont
```

```
state ::=
  | State
  | ExprState
  | Callstate
  | Returnstate
  | Stuckstate
```

```
program_behavior :
  | Terminates tra
  | Diverges trace
  | Reacts tracein
  | Goes_wrong tra
```

```
   | step_seq:  foral
       sstep (State f
         E0 (State f
```

What you find in Compcert C ?

All we need
to prove
behavior preservation
(nearly)

Outline

Part 1 :
Motivation, context
(proving refactoring operations)

→ **Part 2 :**
Consider a particular refactoring operation

Renaming global variables

Renaming global variables ? (in C)

```c
int x = 1 ;

int f(){
   return x + 1 ;
   }
```

rename x→y

```c
int y = 1 ;

int f(){
   return y + 1 ;
   }
```

General case

Renaming global variables ?

(shadowing)

```
int x = 1 ;

int f(int x){
   return x + 1 ;
   }
```

rename x → y

```
int y = 1 ;

int f(int x){
   return x + 1 ;
   }
```

```
int x = 1 ;

int f(int y){
   return y + 1 ;
   }
```

rename x → y

```
int y = 1 ;

int f(int y){
   return y + 1 ;
   }
```

(shadowing)

```
int x = 1 ;

int f(int y){
   return x + y ;
   }
```

rename x → y

capture
(abort)

Particular cases

Renaming global variables ?

```
void f() ;

int x = 1 ;

int main(){
   f() ;
   return x ;
   }
```

rename x → y

```
void f() ;

int y = 1 ;

int main(){
   f() ;
   return y ;
   }
```

lib.a

Pathologic cases

Renaming global variables ?

```c
void f() ;

int x = 1 ;

int main(){
   f() ;
   return x ;
   }
```

rename x → y

```c
void f() ;

int y = 1 ;

int main(){
   f() ;
   return y ;
   }
```

```c
int x ;
void f( ) { x ++ ; }
```

```c
int x ;
void f( ) { x ++ ; }
```

Pathologic cases (breaks the semantics)

Implementation of our refactoring tool

- Program written in Coq,

  (OCaml code generated).
- Transforms abstract syntax trees (AST).
- Fails when problematic case occurs.

```
Definition propagate_change_ident x y (f:function) :=

  if dec_binds x f
  then
    if dec_binds y f
    then OK f
    else
      if dec_appears_statement y (fn_body f)
      then Error (msg "Replacing identifier occurring in function.")
      else OK f

  else (* [x] not bound by [f] *)
    if dec_binds y f
    then
```

Preservation of behaviors (Bisimulation)

- Forward simulation :

program_behaves p b ⟶ program_behaves (Tp) b

- Backward simulation

program_behaves (Tp) b ⟶ program_behaves p b

Preservation of behaviors

```
program_behavior ::=
   | Terminates trace int
   | Diverges trace
   | Reacts traceinf
   | Goes_wrong trace
```

- Forward simulation :

program_behaves p b $\longrightarrow$ program_behaves (Tp) b

program_behaves p b $\longrightarrow$ program_behaves (Tp) (Tb)

- Backward simulation

program_behaves (Tp) b $\longrightarrow$ program_behaves p b

program_behaves (Tp) b $\longrightarrow$

$\exists$ b0,

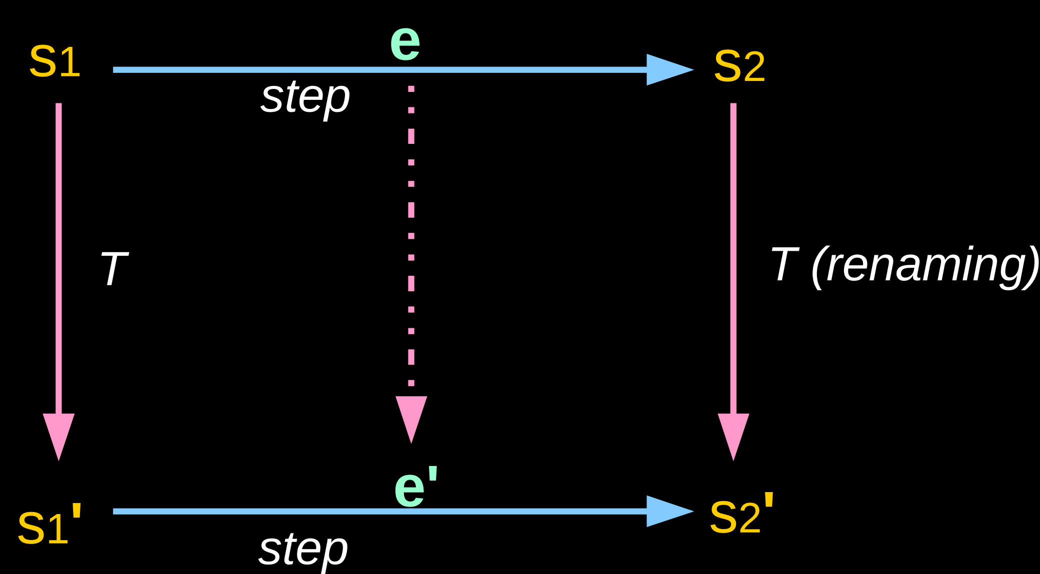
program_behaves p b0 $\wedge$ b = Tb0

Theorem (Coq) : forward simulation

```
Theorem behavior_preserved_1 x y p:
  x <> y →
  ∀ (t_p : program) ,
    rename_globvar x y p = OK t_p →
    ∀ (b : program_behavior),
      program_behaves p b →
      ∃ t_b,
        ( Behaviors.rename_globvar x y b = OK t_b
         ∧ program_behaves t_p t_b ).
```

(+ hypotheses on external functions)

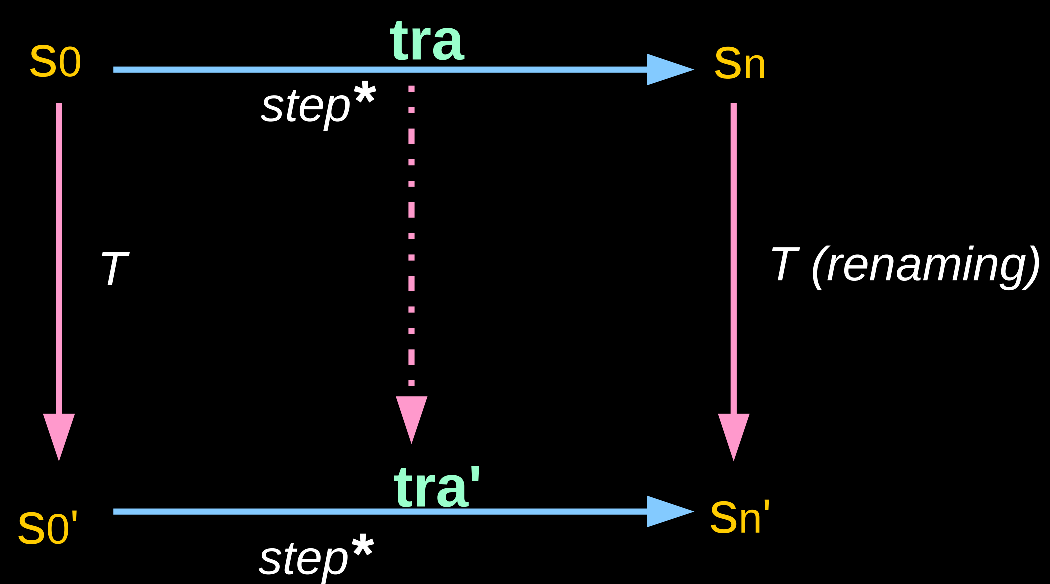
Key result : preservation of transitions (by renaming)



Renaming in events :

`read &x` $\dashrightarrow$ `read &y`

Transitive closure : Preservation of traces




```
program_behavior ::=
  | Terminates trace int
  | Diverges trace
  | Reacts traceinf
  | Goes_wrong trace
```

Difficulty 1 (minor)

Combinatory explosion

```
expr ::=
   | Val val
   | Var ident
   | Field expr ident
   | ValOf expr
   | Deref expr
   | AddrOf expr
   | UnOp unary_op expr
   | BinOp binary_op expr expr
   | Cast expr
   | SeqAnd expr expr
   | SeqOr expr expr
   | Condition expr expr expr
   | SizeOf type
   | AlignOf type
   | Assign expr expr
   | AssignOp binary_op expr expr
   | PostIncr incr_or_decr expr
   | Comma expr expr
   | Call expr exprlist
   | Builtin external_fun exprlist
   | Loc block int
   | Paren expr

exprlist ::=
   | Nil
   | Cons expr exprlist
```

22*22 = 484 cases

```
statement ::=
   | Skip
   | Do expr
   | Sequence statement statement
   | Ifthenelse expr statement statement
   | While expr statement
   | Dowhile expr statement
   | For statement expr statement statement
   | Break
   | Continue
   | Return expr
   | Switch expr labeled_statements
   | Label label statement
   | Goto label

labeled_statements ::=
   | Nil
   | Cons statement labeled_statements
```

Difficulty 2 (minor)

Higher-order Contexts

Small-step semantics generally use **contexts** to specify where reductions are allowed.

Rule for reduction W.R.T contexts :

$$\frac{e \to e'}{C[e] \to C[e']}$$

Examples of contexts :

```
E + .
. + E
. && E
```
~~E && .~~
```
. || E
```
~~E || .~~
```
E(.,E)
E(E,.)
.(E,E)
if (.) { E }
```
~~if (E) { . }~~
```
. ; E
```
~~E ; .~~

Difficulty 2 (minor)

Higher-order Contexts: renaming

In CompCert, contexts are implemented by Coq **functions**.

→ **Difficulty** : we cannot inspect / traverse a Coq function

→ **Solution** : flatten ; rename ; build a function.

(to propagate renamings).

fun e -> (e || Var "x") —*flatten*→ Var "Z" || Var "x"

expr → expr

expr

rename

fun e -> (subst "Z" e (Var "Z" || Var "y")) ←*functify*— Var "Z" || Var "y"

expr → expr

expr

- Painful -

Difficulty 3 (minor)

Continuations

```
cont ::=
  | Kstop
  | Kdo cont
  | Kseq statement cont
  | Kifthenelse statement statement cont
  | Kwhile expr statement cont
  | Kdowhile expr statement cont
  | Kfor expr statement statement cont
  | Kswitch labeled_statements cont
  | Kreturn cont
  | Kcall function env (expr -> expr) cont
```

- Pb 1 : bindings (and shadowing) are implicit.
- Pb 2 : invariants are implicit too.

Example : Kseq (x:=1) (Kcall (func ([x],...) [ ] (\x.x) (Kreturn Kstop))

Difficulty 4 (serious)

Separate compilation

```
void f() ;

int x = 1 ;

int main(){
   f() ;
   return x ;
   }
```

lib.a

In the proofs : some **hypotheses** are needed.

In the prototype : check the libraries in the project (TODO)

Difficulty 5 (limitation)

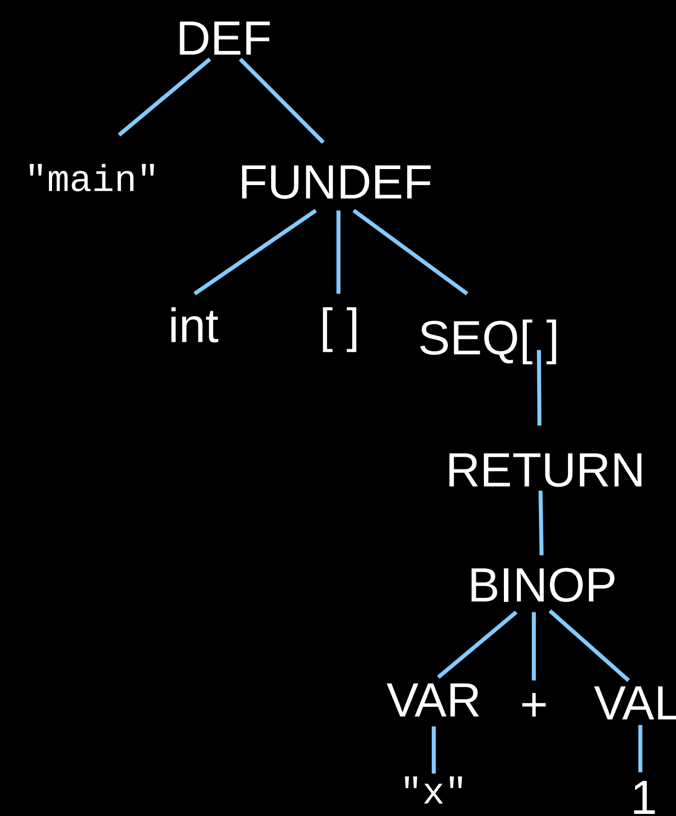
Comments, layout, preprocessing directives

```
int main(){
    /* FIXME */
    return x + 1 ;
    }
```

```
int main(){
    return y + 1 ;
    }
```

Token stream :

```
[
 (1,1,INT) ;
 (1,5, IDENT("main")) ;
 (1,6, LPAR) ;
 (1,7, RPAR) ;
 (1,8, LACC) ;
 (2,4, COMM(" FIXME ")) ;
 (3,4, RET) ;
 (3,11,IDENT("x")) ;
 (3,12,PLUS_OP) ;
 (3,14,NUM(1)) ;
 (3,16,SEMICOLON) ;
 (4,4, RACC) ;
 (5,1, EOF)
]
```



Other properties : Preconditions

When renaming X into Y :

- X is defined as a global variable (**not** as a function)
- X **not volatile**
- X is **not main**
- Y **not** defined globally (global variable or function)
- Y does **not** appear **free**
- Y does **not cover** X (x appears in a function where y is local)

(to avoid captures)

```
Definition no_cover y x f :=
  ~binds y f ∨  binds x f ∨ ~appears_statement x (fn_body f).
```

(+ hypotheses on external functions)

Necessary, sufficient ?

Other Properties

- Invertibility
- Abstract Descriptions.

  Example :

```
Lemma def_gv_z  x  y  z  p  t_p :
 z <> x  →   z <> y  →
 rename_globvar x y p = OK t_p  →
  defines_globvar z p →  defines_globvar z t_p.
```

Limits of the approach

- **Test** and proofs.
- **Pre-processor**, layout, comments.
- **Programs with syntax errors.**
- CompCert C langage limits.

Example :
(wrongly rejected because of a loss of information)

```
int x ;

void f(){
   x++ ;
   {
     int y = 1 ;
     y++ ;
   }
}
```

```
int x ;

void f(){ int y ;
   x++ ;
   {
     y = 1 ;
     y++ ;
   }
}
```

Conclusion

Results

- Prototype tool (to rename global variables).
- Preserves the behavior (bisimulation proved) ***or*** does nothing.

9 000 LOC (transformations) + 26 000 LOC (proofs)

Work in Progress

- Static Composition (static analysis of composites for free)

Future work

- Other refactoring operations.
- Architecture transformations (by composition of operations).
- Collaborate with popular IDEs (Eclipse, Frama-C...).

Take-away messages

- Refactoring operations can be proved correct formally, with some limits.
- Proven properties : behavior preservation, preconditions, abstract descriptions…
- Need a good formalization of the underlying semantics.
- Not easy (1 year, 35 KLOC for this work).