CompCert : C compilers you can *formally* trust

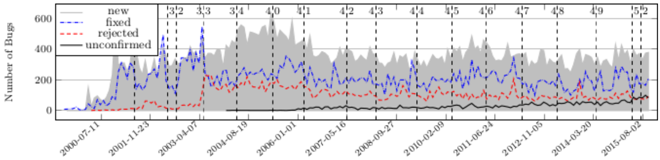
March 2020

`Sylvain.Boulme@univ-grenoble-alpes.fr`

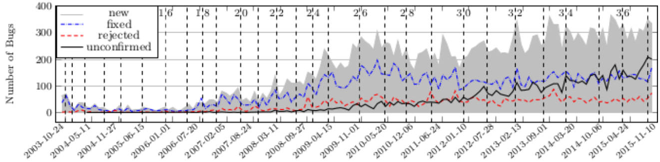
Contents

- Certifying compilers
- The Coq proof assistant for certifying compilers
- Using CompCert
- Overview of CompCert Implementation

Bug trackers of GCC and LLVM (Sun-et-al@PLDI'16)



(a) GCC.



(b) LLVM

The number of attested bugs tends to remain almost constant.
New bugs are introduced when compilers are improved !

Miscompilation bugs in most compilers (GCC, LLVM, etc)

Miscompilation bug $=$ incorrect generated code
$\neq$ "*performance*" bug in an optimization.

Miscompilation bugs in most compilers (GCC, LLVM, etc)

Miscompilation bug $=$ incorrect generated code
$\neq$ "*performance*" bug in an optimization.

Unknown miscompilation bugs **still** remain
as attested by **fuzz (ie randomized) differential testing** :
Eide-Regehr'08, Yang-et-al'11, Lidbury-et-al'15, Sun-et-al'16...

Miscompilation bugs in most compilers (GCC, LLVM, etc)

Miscompilation bug = incorrect generated code
$\neq$ "*performance*" bug in an optimization.

Unknown miscompilation bugs **still** remain
as attested by **fuzz (ie randomized) differential testing** :
Eide-Regehr'08, Yang-et-al'11, Lidbury-et-al'15, Sun-et-al'16...

Why ?

Miscompilation bugs in most compilers (GCC, LLVM, etc)

Miscompilation bug = incorrect generated code
$\neq$ "*performance*" bug in an optimization.

Unknown miscompilation bugs **still** remain
as attested by **fuzz (ie randomized) differential testing** :
Eide-Regehr'08, Yang-et-al'11, Lidbury-et-al'15, Sun-et-al'16...

Why ?

Optimizing compilers are quite large software (in MLoC)
with hundreds of maintainers, e.g :
`https://github.com/gcc-mirror/gcc/blob/master/MAINTAINERS`

Miscompilation bugs in most compilers (GCC, LLVM, etc)

Miscompilation bug = incorrect generated code
$\neq$ "*performance*" bug in an optimization.

Unknown miscompilation bugs **still** remain
as attested by **fuzz (ie randomized) differential testing** :
Eide-Regehr'08, Yang-et-al'11, Lidbury-et-al'15, Sun-et-al'16...

Why ?

Optimizing compilers are quite large software (in MLoC)
with hundreds of maintainers, e.g :
`https://github.com/gcc-mirror/gcc/blob/master/MAINTAINERS`

Another fundamental reason :
Tests of ***optimizing*** *compilers* **cannot cover** all corner cases
because of a **combinatorial explosion**.

Issue : *optimizing* compiler for *safety-critical* software

Strong safety-critical requirements of
DO-178 (Avionics), ISO-26262 (Automotive), IEC-62279 (Railway), IEC-61513 (Nuclear)
often established at the source level...

Issue : *optimizing* compiler for *safety-critical* software

Strong safety-critical requirements of
DO-178 (Avionics), ISO-26262 (Automotive), IEC-62279 (Railway), IEC-61513 (Nuclear)
often established at the source level...

Used solution
human review of the *compiled code*

Issue : *optimizing* compiler for *safety-critical* software

Strong safety-critical requirements of
DO-178 (Avionics), ISO-26262 (Automotive), IEC-62279 (Railway), IEC-61513 (Nuclear)
often established at the source level...

Used solution
human review of the *compiled code* ← intractable if *optimized*

Issue : *optimizing* compiler for *safety-critical* software

Strong safety-critical requirements of
DO-178 (Avionics), ISO-26262 (Automotive), IEC-62279 (Railway), IEC-61513 (Nuclear)
often established at the source level...

Used solution
human review of the *compiled code* ← intractable if *optimized*
+ switch-off compiler optimizations (DO-178B level A).

Issue : *optimizing* compiler for *safety-critical* software

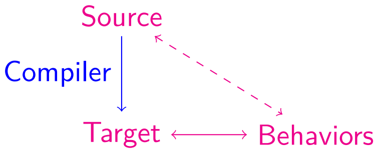
Strong safety-critical requirements of
DO-178 (Avionics), ISO-26262 (Automotive), IEC-62279 (Railway), IEC-61513 (Nuclear)
often established at the source level...

Used solution
human review of the *compiled code* ← intractable if *optimized*
+ switch-off compiler optimizations (DO-178B level A).

Better solution a *formally proved* compiler
for formal tool qualification (DO-178C + DO-333)...

Certified (= *formally proved*) compiler

Diagrammatic view
of the **correctness**



Source

Compiler

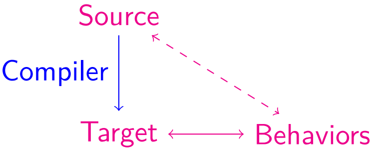
Target

Behaviors

Compiler correctness reduced to that of its formal spec.

Certified (= *formally proved*) compiler

Diagrammatic view
of the **correctness**



$$\begin{tikzcd}
\text{Source} \arrow[d, "\text{Compiler}"'] \arrow[dr, dashed, leftrightarrow] & \\
\text{Target} \arrow[r, leftrightarrow] & \text{Behaviors}
\end{tikzcd}$$

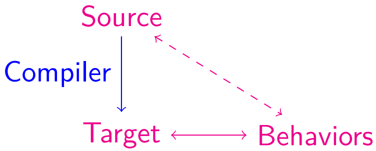
Compiler correctness reduced to that of its formal spec.

Advantages of formal spec over compiler code

- closer to informal spec (e.g. simpler for human reviews)
- more compositional (e.g. simpler for tests)

Certified (= *formally proved*) compiler

Diagrammatic view
of the **correctness**



Compiler correctness reduced to that of its formal spec.

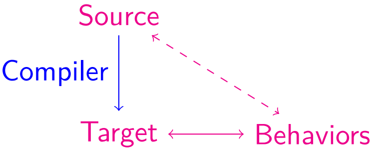
Advantages of formal spec over compiler code

- closer to informal spec (e.g. simpler for human reviews)
- more compositional (e.g. simpler for tests)

Another benefit : traceability
formal proof = computer-aided review of the compiler code w.r.t its spec.

Certified (= *formally proved*) compiler

Diagrammatic view
of the **correctness**



Source

Compiler

Target

Behaviors

Compiler correctness reduced to that of its formal spec.

Advantages of formal spec over compiler code

- closer to informal spec (e.g. simpler for human reviews)
- more compositional (e.g. simpler for tests)

Another benefit : traceability
formal proof = computer-aided review of the compiler code w.r.t its spec.
⇒ up-to-date & very sharp (formal) documentation of the compiler
that may also help "*external developers*"

CompCert : a **certified** compiler

CompCert = a *moderately*-optimizing C compiler
with an *unprecedented* level of trust in its correctness

CompCert : a **certified** compiler

CompCert = a *moderately*-optimizing C compiler
with an *unprecedented* level of trust in its correctness
as noted by Yang-et-al'11 (with randomized differential testing) :

> "CompCert *is the only compiler we have tested for which* Csmith *cannot find wrong-code errors. This is not for lack of trying : we have devoted about six CPU-years to the task.*
> [. . .] *developing compiler optimizations within a proof framework*
> [. . .] *has tangible benefits for compiler users."*

COMPCERT : a **certified** compiler

COMPCERT = a *moderately*-optimizing C compiler
with an *unprecedented* level of trust in its correctness
as noted by Yang-et-al'11 (with randomized differential testing) :

> "COMPCERT *is the only compiler we have tested for which* CSMITH *cannot find wrong-code errors. This is not for lack of trying : we have devoted about six CPU-years to the task.*
> [. . .] *developing compiler optimizations within a proof framework* [. . .] *has tangible benefits for compiler users.*"

Part of an **ongoing effort to certify a whole software chain** in the COQ proof assistant
from the prover (e.g. CertiCoq) to OS kernels (e.g. CertiKOS)
Example `http://deepspec.org` (supported by NSF).

Contents

The COQ proof assistant

A *language* to **formalize mathematical theories** (and their proofs) **with a computer**. Examples :

- Four-color & Odd-order theorems by Gonthier-et-al.
- Univalence theory by Voevodsky (Fields Medalist).

The Coq proof assistant

A *language* to **formalize mathematical theories** (and their proofs) **with a computer**. Examples :

- Four-color & Odd-order theorems by Gonthier-et-al.
- Univalence theory by Voevodsky (Fields Medalist).

With a high-level of confidence :

- Logic reduced to a few powerful constructs ;
  Proofs checked by a small verifiable *kernel*
- Consistency-by-construction of most user theories
  (promotes *definitions* instead of *axioms*)

The Coq proof assistant

A *language* to **formalize mathematical theories** (and their proofs) **with a computer**. Examples :

- Four-color & Odd-order theorems by Gonthier-et-al.
- Univalence theory by Voevodsky (Fields Medalist).

With a high-level of confidence :

- Logic reduced to a few powerful constructs ;
  Proofs checked by a small verifiable *kernel*
- Consistency-by-construction of most user theories
  (promotes *definitions* instead of *axioms*)

ACM Software System Award in 2013

for Coquand, Huet, Paulin-Mohring et al.

The Coq proof assistant

A *language* to **formalize mathematical theories** (and their proofs) **with a computer**. Examples :

- Four-color & Odd-order theorems by Gonthier-et-al.
- Univalence theory by Voevodsky (Fields Medalist).

With a high-level of confidence :

- Logic reduced to a few powerful constructs ;
  Proofs checked by a small verifiable *kernel*
- Consistency-by-construction of most user theories
  (promotes *definitions* instead of *axioms*)

ACM Software System Award in 2013

for Coquand, Huet, Paulin-Mohring et al.

Results from a long history in formalizing mathematical reasonning since Frege, Russel, Hilbert near 1900.

Formally proved programs in the Coq proof assistant

The Coq logic includes a functional programming language with pattern-matching on tree-like data-structures.

Extraction of Coq functions to OCaml
+ OCaml compilation to produce native code.

⇒ **CompCert is programmed in both Coq and OCaml.**

The kernel of Coq in a nutshell (1/2)

A *typed* programming language, *only* handling data of the form

- inductive data (tree-like data)
- (pure) functions (with structural recursion)
- types, where `Type`$_i$ is the type of `Type`$_j$ with $j < i$

The kernel of Coq in a nutshell (1/2)

A *typed* programming language, *only* handling data of the form

- inductive data (tree-like data)
- (pure) functions (with structural recursion)
- types, where `Type`$_i$ is the type of `Type`$_j$ with $j < i$

Example where `Z` in `Type`$_0$ is the type of relative integers

```
Inductive nat: Type := O | S(n:nat).  (* defines natural numbers *)

Fixpoint plus (n m:nat): nat :=       (* defines n+m recursively *)
  match n with O => m | (S n') => (S (plus n' m)) end.

(* Type of tuples containing (S n) values in Z *)
Fixpoint tuple_S (n:nat): Type :=
  match n with O => Z | S n' => Z * (tuple_S n') end.

(* Concatenation operation of such tuples *)
Fixpoint app (n m:nat):(tuple_S n)->((tuple_S m)->(tuple_S (S (plus n m)))) :=
  match n with
    O => fun t1 t2 => (t1, t2)
  | S n' => fun t1 t2 => let (x,t1') := t1 in (x, app n' m t1' t2)
  end.
```

The kernel of COQ in a nutshell (1/2)

A *typed* programming language, *only* handling data of the form

- inductive data (tree-like data)
- (pure) functions (with structural recursion)
- types, where `Type`$_i$ is the type of `Type`$_j$ with $j < i$

Example where `Z` in `Type`$_0$ is the type of relative integers

```
Inductive nat: Type := O | S(n:nat).   (* defines natural numbers *)

Fixpoint plus (n m:nat): nat :=        (* defines n+m recursively *)
  match n with O => m | (S n') => (S (plus n' m)) end.

(* Type of tuples containing (S n) values in Z *)
Fixpoint tuple_S (n:nat): Type :=
  match n with O => Z | S n' => Z * (tuple_S n') end.

(* Concatenation operation of such tuples *)
Fixpoint app (n m:nat):(tuple_S n)->((tuple_S m)->(tuple_S (S (plus n m)))) :=
  match n with
    O => fun t1 t2 => (t1, t2)
  | S n' => fun t1 t2 => let (x,t1') := t1 in (x, app n' m t1' t2)
  end.
```

Decidable typechecking with *computations in types*!

Only *structural* recursion is allowed ⇒ all computations terminates.

The kernel of Coq in a nutshell (2/2)

Type of `app` :

```
forall (n m:nat), tuple_S n -> tuple_S m -> tuple_S(S (plus n m))
```

The kernel of Coq in a nutshell (2/2)

Type of `app` :

```
forall (n m:nat), tuple_S n -> tuple_S m -> tuple_S(S (plus n m))
```

More generally, `forall (x:A),(P x)`
is the type of functions `fun(x:A) => e` where `e:(P x)`.

The kernel of COQ in a nutshell (2/2)

Type of `app` :

```
forall (n m:nat), tuple_S n -> tuple_S m -> tuple_S(S (plus n m))
```

More generally, `forall (x:A),(P x)`
is the type of functions `fun(x:A) => e` where `e:(P x)`.

NB : `A -> B` is `forall (x:A),B` when `x` not occurring in `B`.

The kernel of COQ in a nutshell (2/2)

Type of `app` :

```
forall (n m:nat), tuple_S n -> tuple_S m -> tuple_S(S (plus n m))
```

More generally, `forall (x:A),(P x)`
is the type of functions `fun(x:A) => e` where `e:(P x)`.

NB : `A -> B` is `forall (x:A),B` when `x` not occurring in `B`.

Typing rule : when `A:Type` (with restrictions) and `P: A->Type`$_i$
then `forall (x:A),(P x)` in `Type`$_i$

Propositions as types (Curry-Howard isomorphism)

`Prop` in $\texttt{Type}_1$ represents the type of *logical propositions* :
Coq proofs are *values* in types of `Prop`

Propositions as types (Curry-Howard isomorphism)

`Prop` in $\texttt{Type}_1$ represents the type of *logical propositions* :
Coq proofs are *values* in types of `Prop`

For `A:Prop` and `B:Prop`, `A->B` is read
"*proposition* A *implies proposition* B"
A function in `A->B` is *a proof* of this proposition.

Propositions as types (Curry-Howard isomorphism)

`Prop` in `Type`$_1$ represents the type of *logical propositions* :
Coq proofs are *values* in types of `Prop`

For `A:Prop` and `B:Prop`, `A->B` is read
"*proposition A implies proposition B*"
A function in `A->B` is *a proof* of this proposition.

Similarly, for `A:Type` and `P:A->Prop`,
`forall (x:A),(P x)` is read "*for all x:A, (P x)*"
A function in `forall (x:A),(P x)` is *a proof* of this proposition.

Propositions as types (Curry-Howard isomorphism)

`Prop` in $\texttt{Type}_1$ represents the type of *logical propositions* :
Coq proofs are *values* in types of `Prop`

For `A:Prop` and `B:Prop`, `A->B` is read
"*proposition A implies proposition B*"
A function in `A->B` is *a proof* of this proposition.

Similarly, for `A:Type` and `P:A->Prop`,
`forall (x:A),(P x)` is read "*for all x:A, (P x)*"
A function in `forall (x:A),(P x)` is *a proof* of this proposition.

All logical features (including logical connectors, equality, well-founded induction) are built from the Coq kernel.

Propositions as types (Curry-Howard isomorphism)

`Prop` in $\texttt{Type}_1$ represents the type of *logical propositions* :
COQ proofs are *values* in types of `Prop`

For `A:Prop` and `B:Prop`, `A->B` is read
"*proposition A implies proposition B*"
A function in `A->B` is *a proof* of this proposition.

Similarly, for `A:Type` and `P:A->Prop`,
`forall (x:A),(P x)` is read "*for all x:A, (P x)*"
A function in `forall (x:A),(P x)` is *a proof* of this proposition.

All logical features (including logical connectors, equality, well-founded induction) are built from the COQ kernel.

Gives a *subset* of classical logic called *intuitionistic logic*.
Classical logic recovered with a few additional axioms like

```
Axiom excluded_middle: forall (A:Prop), A \/ (A -> False).
```

A flavour of certifying compilers in Coq

CompCert proof is huge ($>$ 100Kloc of Coq).

Follow this link to have a simpler example :
`http://www-verimag.imag.fr/~boulme/IntroCompCert/DemoCoq/`

Contents

Overview of CompCert

Input most of ISO C99 + a few extensions
Output (32&64 bits) code for PowerPC, ARM, x86, RISC-V, Kalray K1C

Overview of CompCert

Input most of ISO C99 + a few extensions
Output (32&64 bits) code for PowerPC, ARM, x86, RISC-V, Kalray K1C

Developed since 2005 by Leroy-et-al at Inria
- Commercial support since 2015 by AbsInt (German Company)
- Industrial uses in Avionics (Airbus) & Nuclear Plants (MTU)

Overview of CompCert

Input most of ISO C99 + a few extensions
Output (32&64 bits) code for PowerPC, ARM, x86, RISC-V, Kalray K1C

Developed since 2005 by Leroy-et-al at Inria
- Commercial support since 2015 by AbsInt (German Company)
- Industrial uses in Avionics (Airbus) & Nuclear Plants (MTU)

Unequaled level of trust for industrial-scaling compilers
- Correctness proved within the Coq proof assistant

Overview of COMPCERT

Input most of ISO C99 + a few extensions
Output (32&64 bits) code for PowerPC, ARM, x86, RISC-V, Kalray K1C

Developed since 2005 by Leroy-et-al at Inria
 Commercial support since 2015 by AbsInt (German Company)
 Industrial uses in Avionics (Airbus) & Nuclear Plants (MTU)

Unequaled level of trust for industrial-scaling compilers
 Correctness proved within the Coq proof assistant

Performance of generated code (for PowerPC and ARM)

 2× *faster* than `gcc -O0`
 10% *slower* than `gcc -O1` and 20% than `gcc -O3`.

In MTU systems (German provider of Nuclear Power Plants)
28% *smaller* WCET than with a previous *unverified* compiler.

Understanding the formal correctness of COMPCERT

Formally, correctness of compiled code is ensured modulo

- correctness of C formal semantics in COQ
- correctness of assembly formal semantics in COQ
- absence of undefined behavior in the source program

Understanding the formal correctness of CompCert

Formally, correctness of compiled code is ensured modulo

- correctness of C formal semantics in Coq
- correctness of assembly formal semantics in Coq
- absence of undefined behavior in the source program

Formal semantics $\simeq$ relation between “programs” and “behaviors”
i.e. a (possibly non-deterministic) interpretation of programs

for C : formalization of ISO C99 standard
for assembly : formalization/abstraction of ISA

Understanding the formal correctness of CompCert

Formally, correctness of compiled code is ensured modulo

- correctness of C formal semantics in Coq
- correctness of assembly formal semantics in Coq
- absence of undefined behavior in the source program

Formal semantics ≃ relation between “programs” and “behaviors”
i.e. a (possibly non-deterministic) interpretation of programs

for C : formalization of ISO C99 standard
for assembly : formalization/abstraction of ISA

Source program assumed to be without undefined behavior

```
int x, t[10], y;
...
if (...) {
  t[10]=1; // undefined behavior: out of bounds
  // the compiler could write in x or y,
  // or prune the branch as dead-code, ...
```

Informal view of CompCert formal correctness

Observable Value = int or float or address of global variable

Informal view of CompCert formal correctness

Observable Value = int or float or address of global variable

Trace = a sequence of *external function* calls (or *volatile accesses*) each of the form “$f(v_1, \ldots, v_n) \mapsto v$” where f is name

Informal view of CompCert formal correctness

Observable Value = int or float or address of global variable

Trace = a sequence of *external function* calls (or *volatile accesses*)
each of the form “$f(v_1, \ldots, v_n) \mapsto v$” where f is name

Behavior = one of the four possible cases (of an execution) :

- an infinite trace (of a diverging execution)
- a finite trace followed by an infinite “silent” loop
- a finite trace followed by an integer exit code (terminating case)
- a finite trace followed by an error (UNDEFINED-BEHAVIOR)

Informal view of CompCert formal correctness

Observable Value = int or float or address of global variable

Trace = a sequence of *external function* calls (or *volatile accesses*)
each of the form "$f(v_1, \ldots, v_n) \mapsto v$" where f is name

Behavior = one of the four possible cases (of an execution) :

$$\begin{cases} \text{an infinite trace (of a diverging execution)} \\ \text{a finite trace followed by an infinite "silent" loop} \\ \text{a finite trace followed by an integer exit code (terminating case)} \\ \text{a finite trace followed by an error (UNDEFINED-BEHAVIOR)} \end{cases}$$

Semantics = maps each *program* to a set of *behaviors*.

Informal view of CompCert formal correctness

Observable Value = int or float or address of global variable

Trace = a sequence of *external function* calls (or *volatile accesses*)
each of the form "$f(v_1, \ldots, v_n) \mapsto v$" where f is name

Behavior = one of the four possible cases (of an execution) :

- an infinite trace (of a diverging execution)
- a finite trace followed by an infinite "silent" loop
- a finite trace followed by an integer exit code (terminating case)
- a finite trace followed by an error (UNDEFINED-BEHAVIOR)

Semantics = maps each *program* to a set of *behaviors*.

Correctness of the compiler

For any source program S,
if S has no UNDEFINED-BEHAVIOR,
and if the compiler returns some assembly program C,
then any behavior of C is also a behavior of S.

Informal view of CompCert formal correctness

Observable Value = int or float or address of global variable

Trace = a sequence of *external function* calls (or *volatile accesses*)
each of the form "$f(v_1, \ldots, v_n) \mapsto v$" where f is name

Behavior = one of the four possible cases (of an execution) :

- an infinite trace (of a diverging execution)
- a finite trace followed by an infinite "silent" loop
- a finite trace followed by an integer exit code (terminating case)
- a finite trace followed by an error (UNDEFINED-BEHAVIOR)

Semantics = maps each *program* to a set of *behaviors*.

Correctness of the compiler

For any source program S,
if S has no UNDEFINED-BEHAVIOR,
and if the compiler returns some assembly program C,
then any behavior of C is also a behavior of S.

NB : under these conditions, C has no UNDEFINED-BEHAVIOR.

Trust in ELF binaries produced with CompCert

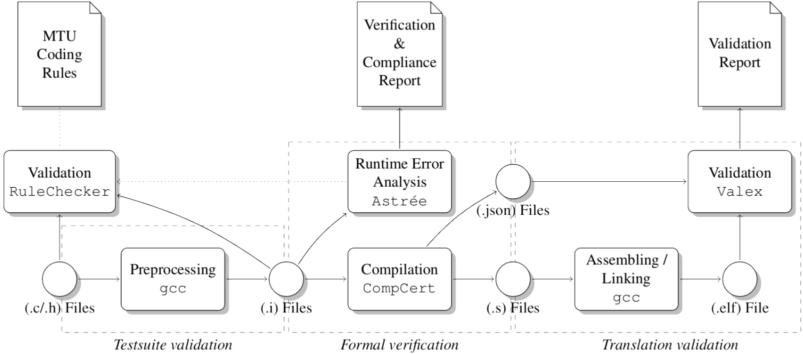
Trust in binaries requires additional verifications, at least :

- absence of undefined behavior in C code (e.g. with Astrée)
- correctness of assembling/linking (e.g. with Valex)

Trust in ELF binaries produced with CompCert

Trust in binaries requires additional verifications, at least :

- absence of undefined behavior in C code (e.g. with Astrée)
- correctness of assembling/linking (e.g. with Valex)



Qualification of MTU *development chain* for Nuclear safety
from Käster, Barrho et al @ERTS'18

Contents

CompCert's model of Intermediate Representations

Definition The transition semantics (of a program) is defined – on a given type of states – by :

- a subset of initial states (i.e. at "`main`" entry-point) ;
- a subset of final states (i.e. at "`returns`" of "`main`") ;
- a step relation written $S \xrightarrow{\mathtt{t}} S'$

with t being either one observable event or ϵ (i.e. "silent" step).

CompCert's model of Intermediate Representations

Definition The transition semantics (of a program) is defined – on a given type of states – by :

- a subset of initial states (i.e. at "`main`" entry-point) ;
- a subset of final states (i.e. at "`returns`" of "`main`") ;
- a step relation written $S \xrightarrow{\mathtt{t}} S'$

  with t being either one observable event or ϵ (i.e. "silent" step).

Behavior = trace produced by a *maximal* sequence of steps from an initial state

CompCert's model of Intermediate Representations

Definition The transition semantics (of a program) is defined – on a given type of states – by :

- a subset of initial states (i.e. at "`main`" entry-point) ;
- a subset of final states (i.e. at "`returns`" of "`main`") ;
- a step relation written $S \xrightarrow{t} S'$

  with t being either one observable event or ϵ (i.e. "silent" step).

Behavior = trace produced by a *maximal* sequence of steps from an initial state

4 kind of behaviors recovered by :

- infinite sequence with a finite or infinite trace
- finite sequence ended on a final state
- finite sequence ended on a non-final state (*stuck*)

  ⇒ UNDEFINED-BEHAVIOR

Certifying compilation passes in CompCert

Theorem : correctness of forward simulations

The *correctness* of a pass between a *source semantics* on S_1 to a *deterministic target semantics* on S_2, can be proved by a simulation relation $S_1 \sim S_2$ that :

- is established on initial states
- preserves final states
- and execution steps with :

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\sim"] \arrow[d, "t"'] & S_2 \arrow[d, dashed, "t"', "+"] \\
S'_1 \arrow[r, dash, dashed, "\sim"] & S'_2
\end{tikzcd}$$

or

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\sim"] \arrow[d, "\epsilon"'] & S_2 \arrow[dl, dash, dashed, "\sim"'] \\
S'_1 &
\end{tikzcd}$$

with $|S'_1| < |S_1|$

NB : condition $|S'_1| < |S_1|$ ensures preservation of infinite silent loops.

Untrusted Oracles in CompCert

Principle : delegate computations to efficient OCaml functions without having to prove them !
⇒ only a checker of the result is verified
i.e. verified defensive programming

Untrusted Oracles in CompCert

Principle : delegate computations to efficient OCaml functions without having to prove them !
⇒ only a checker of the result is verified
i.e. verified defensive programming

Example of *register allocation* – a NP-complete problem (related to a graph-coloring problem)

- finding a *correct* and *efficient* allocation is difficult
- verifying the *correctness* of an allocation is easy

⇒ only "*allocation checking*" is verified in Coq

Untrusted Oracles in COMPCERT

Principle : delegate computations to efficient OCAML functions without having to prove them !
⇒ only a checker of the result is verified
i.e. verified defensive programming

Example of *register allocation* – a NP-complete problem (related to a graph-coloring problem)
- finding a *correct* and *efficient* allocation is difficult
- verifying the *correctness* of an allocation is easy

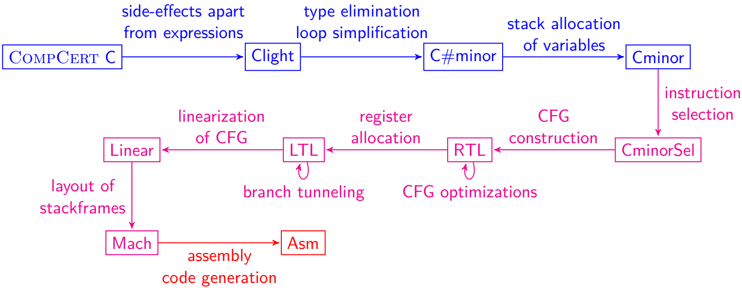
⇒ only "*allocation checking*" is verified in COQ

Benefits of untrusted oracles

simplicity + efficiency + modularity

Modular design of CompCert in Coq

Components independent/parametrized/specific w.r.t. the target



CompCert C → Clight: side-effects apart from expressions

Clight → C#minor: type elimination, loop simplification

C#minor → Cminor: stack allocation of variables

Cminor → CminorSel: instruction selection

CminorSel → RTL: CFG construction

RTL → RTL: CFG optimizations

RTL → LTL: register allocation

LTL → LTL: branch tunneling

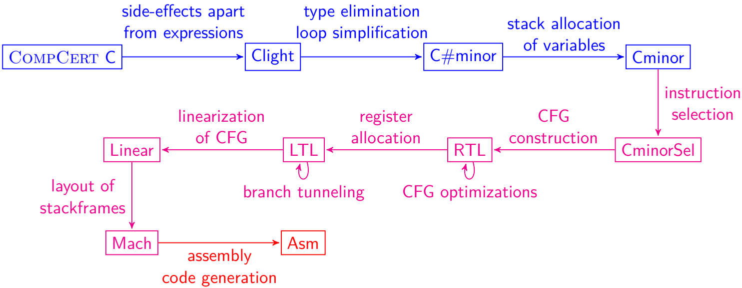
LTL → Linear: linearization of CFG

Linear → Mach: layout of stackframes

Mach → Asm: assembly code generation

Modular design of CompCert in Coq

Components independent/parametrized/specific w.r.t. the target



Demo on a mini example for x86-64 target at this link :
`http://www-verimag.imag.fr/~boulme/IntroCompCert/DemoCompCert/`