Verified Compilers/Project Cristal

Valerian Wintner

ITP Course

June 20th, 2022

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Verified vs certifying compilers

Verified compiler

- Semantic preservation is proven, i.e. input/output-behavior of code
- All translation-steps in compiler need to be formally verified

Certifying compiler

- Compiler generates proof/"certificate" in addition to compiled program, telling if some specification (e.g. memory safety) holds
- Verified validator checks output
- "Translation validation": Validator checks if semantics are preserved

- Arithmetic Expressions compiler (1967, formalized in LCF in 1972)[1][2]
- Compiler verification: a bibliography[3] (1967-2003)
- **CompCert**[4]: Realistic C compiler, 2005+
- CakeML[5]: Compiler for functional language, 2019
- Lutsig[6]: Verilog compiler, hardware description language, 2021
- CertiCoq[7][8]: Gallina compiler, ongoing, targets Clight

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

- (90%) Formally verified C compiler with optimizations
- 2x faster code than GCC -O0, 10%/15%/20% slower than -O1/-O2/-O3
- Supports most of ISO C99 and few ISO C2011 extensions[9]
- Architectures: PowerPC, ARMv7, AArch64, ia32, AMD64, RISC-V[10]
- Developed since 2005 by Xavier Leroy at INRIA, France
- Won "ACM Software System Award" in 2021
- Commercial license since 2015 (from AbsInt), free for noncommercial use
- 100 000 lines of Coq, extracted to OCaml
- For embedded software (safety critical).
  Alternatives: Manual code-review, extensive testing, static analysis, disable optimizations.

Finding bugs in compilers:

> *Yang et al.[11]: "The striking thing about our* **CompCert** *results is that the* **middleend bugs we found in all other compilers are absent**. *As of early 2011, the under-development version of 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. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."*

However, they found bugs in the unverified parts of the compiler (fixed and verification expanded) and an overflow (caught by assembler).

Usage

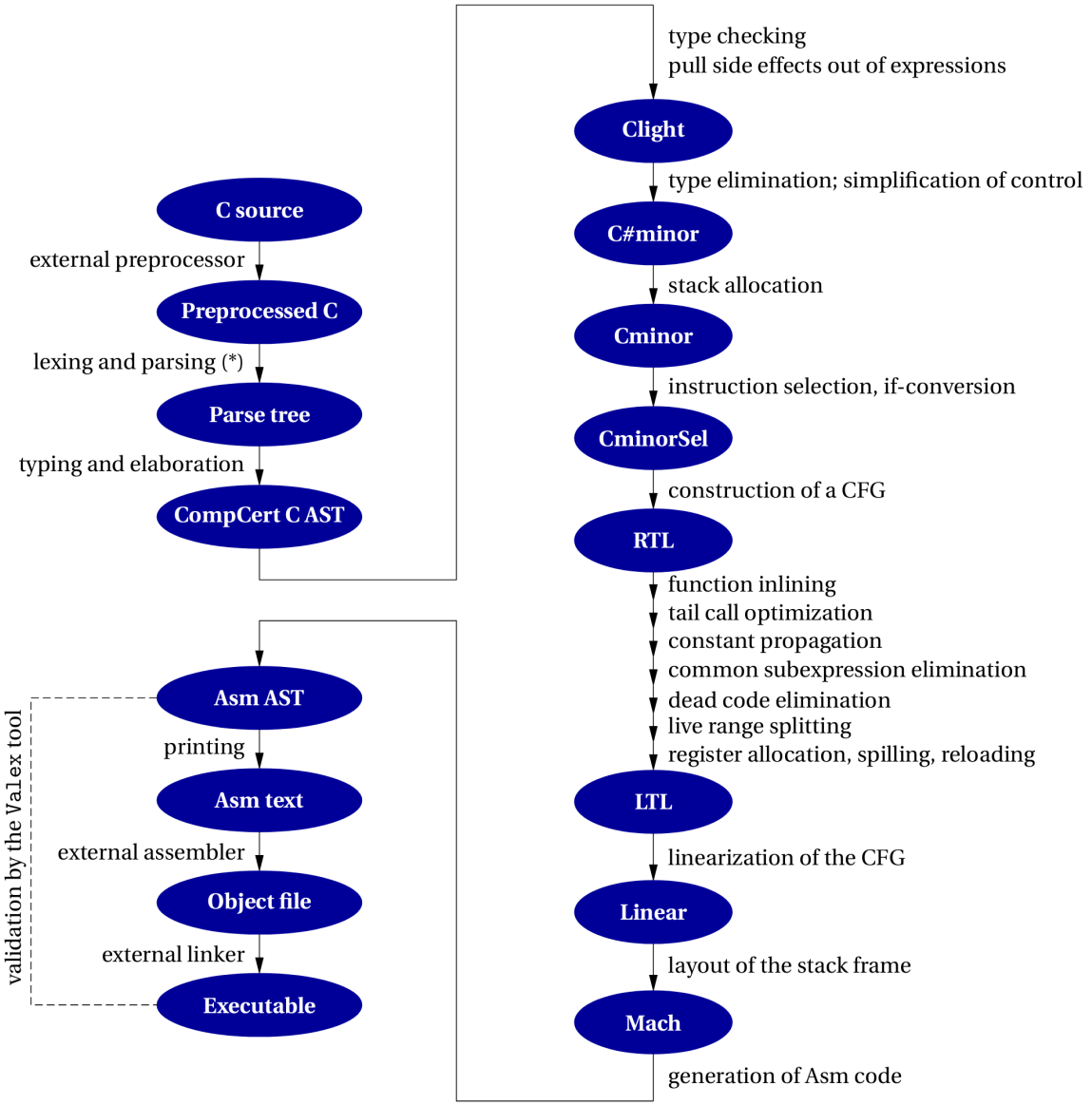
- Verified Software Toolchain[12]
- Airbus[13]
- TU Munich, flight software (real plane testbed)[14]
- CertikOS[15]
- Lustre Compiler (Lustre to Clight)[16]
- MTU Friedrichshafen (nuclear power plant, diesel generator fallback)[17][18]

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Compiler structure



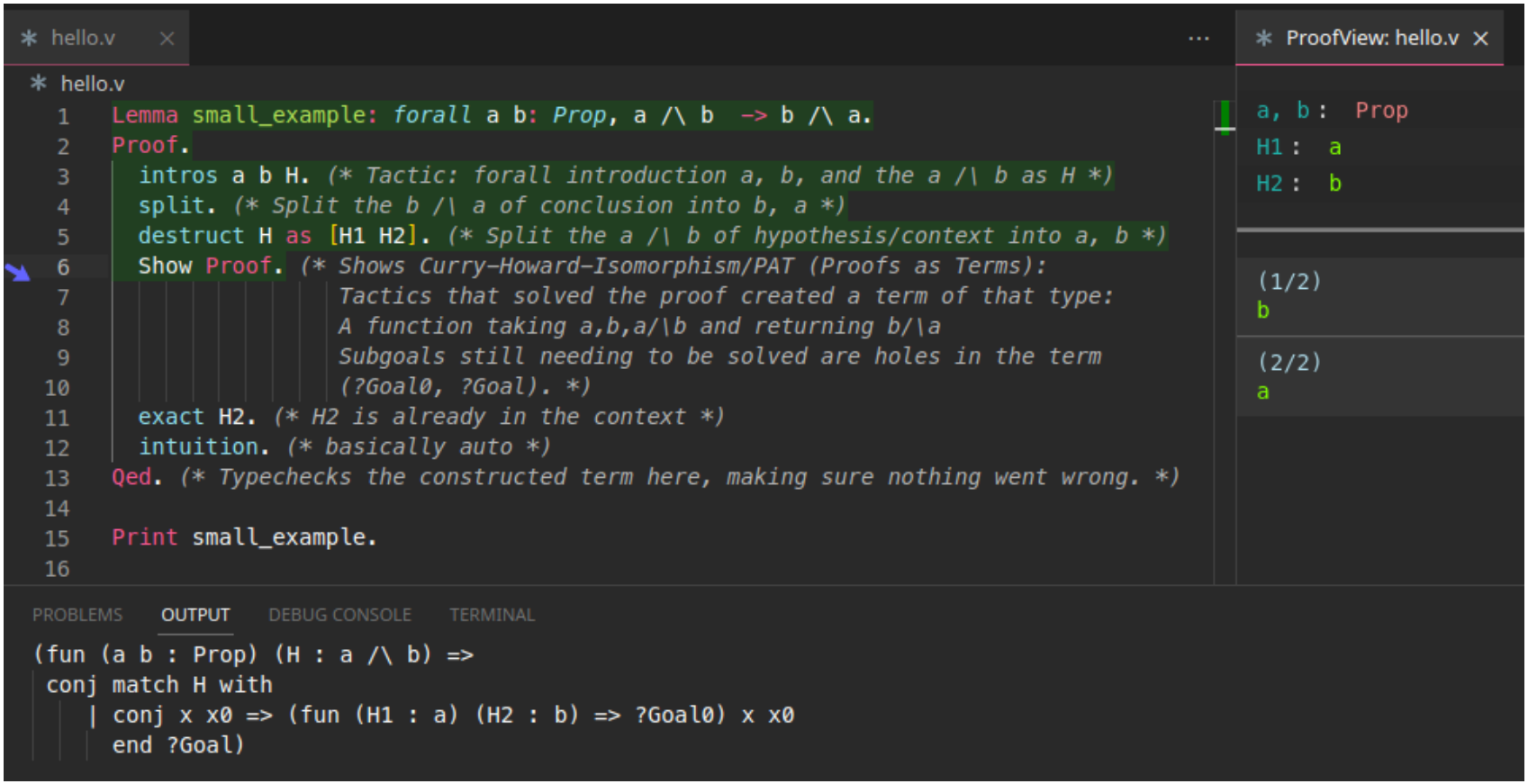
(*) the parser is formally verified

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Proofs


hello.v
hello.v
1 Lemma small_example: forall a b: Prop, a /\ b -> b /\ a.
2 Proof.
3 intros a b H. (* Tactic: forall introduction a, b, and the a /\ b as H *)
4 split. (* Split the b /\ a of conclusion into b, a *)
5 destruct H as [H1 H2]. (* Split the a /\ b of hypothesis/context into a, b *)
6 Show Proof. (* Shows Curry-Howard-Isomorphism/PAT (Proofs as Terms):
7 Tactics that solved the proof created a term of that type:
8 A function taking a,b,a/\b and returning b/\a
9 Subgoals still needing to be solved are holes in the term
10 (?Goal0, ?Goal). *)
11 exact H2. (* H2 is already in the context *)
12 intuition. (* basically auto *)
13 Qed. (* Typechecks the constructed term here, making sure nothing went wrong. *)
14
15 Print small_example.
16
ProofView: hello.v
a, b : Prop
H1 : a
H2 : b
(1/2)
b
(2/2)
a
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
(fun (a b : Prop) (H : a /\ b) =>
conj match H with
| conj x x0 => (fun (H1 : a) (H2 : b) => ?Goal0) x x0
end ?Goal)

Datatypes, functions

```
(* Datatypes *)
Inductive nat' : Type :=
  | zero : nat'
  | succ : nat' -> nat'.

(* Non-recursive functions. *)
Definition add_1 (n:nat') : nat' := succ n.

Compute add_1 (add_1 zero).

(* Fixpoint for recursive functions. *)
```

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Compiler functions (1)

```
Definition transf_cminor_program (p: Cminor.program) : res Asm.program :=
   OK p
   @@ print print_Cminor
  @@@ time "Instruction selection" Selection.sel_program
  @@@ time "RTL generation" RTLgen.transl_program
  @@@ transf_rtl_program.

Definition transf_clight_program (p: Clight.program) : res Asm.program :=
  OK p
   @@ print print_Clight
  @@@ time "Simplification of locals" SimplLocals.transf_program
  @@@ time "C#minor generation" Cshmgen.transl_program
  @@@ time "Cminor generation" Cminorgen.transl_program
  @@@ transf_cminor_program.

Definition transf_c_program (p: Csyntax.program) : res Asm.program :=
  OK p
  @@@ time "Clight generation" SimplExpr.transl_program
  @@@ transf_clight_program.
```

Compiler functions (2)

```
Definition transf_rtl_program (f: RTL.program) : res Asm.program :=
  OK f
  @@ print (print_RTL 0)
  @@ total_if Compopts.optim_tailcalls (time "Tail calls" Tailcall.transf_program)
  @@ print (print_RTL 1)
 @@@ time "Inlining" Inlining.transf_program
  @@ print (print_RTL 2)
  @@ time "Renumbering" Renumber.transf_program
  @@ print (print_RTL 3)
  @@ total_if Compopts.optim_constprop (time "Constant propagation" Constprop.transf_program)
  @@ print (print_RTL 4)
  @@ total_if Compopts.optim_constprop (time "Renumbering" Renumber.transf_program)
  @@ print (print_RTL 5)
 @@@ partial_if Compopts.optim_CSE (time "CSE" CSE.transf_program)
  @@ print (print_RTL 6)
 @@@ partial_if Compopts.optim_redundancy (time "Redundancy elimination" Deadcode.transf_program)
  @@ print (print_RTL 7)
 @@@ time "Unused globals" Unusedglob.transform_program
  @@ print (print_RTL 8)
 @@@ time "Register allocation" Allocation.transf_program
  @@ print print_LTL
  @@ time "Branch tunneling" Tunneling.tunnel_program
 @@@ time "CFG linearization" Linearize.transf_program
  @@ time "Label cleanup" CleanupLabels.transf_program
 @@@ partial_if Compopts.debug (time "Debugging info for local variables" Debugvar.transf_program)
 @@@ time "Mach generation" Stacking.transf_program
  @@ print print_Mach
 @@@ time "Asm generation" Asmgen.transf_program.
```

C syntax excerpt

```
Inductive statement : Type :=
  | Sskip : statement                    (**r do nothing *)
  | Sdo : expr -> statement              (**r evaluate expression for side effects *)
  | Ssequence : statement -> statement -> statement  (**r sequence *)
  | Sifthenelse : expr  -> statement -> statement -> statement (**r conditional *)
  | Swhile : expr -> statement -> statement   (**r [while] loop *)
  | Sdowhile : expr -> statement -> statement (**r [do] loop *)
  | Sfor: statement -> expr -> statement -> statement -> statement (**r [for] loop *)
  | Sbreak : statement                        (**r [break] statement *)
  | Scontinue : statement                     (**r [continue] statement *)
  | Sreturn : option expr -> statement     (**r [return] statement *)
  | Sswitch : expr -> labeled_statements -> statement  (**r [switch] statement *)
  | Slabel : label -> statement -> statement
  | Sgoto : label -> statement

with labeled_statements : Type :=            (**r cases of a [switch] *)
  | LSnil: labeled_statements
  | LScons: option Z -> statement -> labeled_statements -> labeled_statements.
                  (**r [None] is [default], [Some x] is [case x] *)
```

Main Proof (1)

```
(** The [transf_c_program] function, when successful, produces
  assembly code that is in the [match_prog] relation with the source C program. *)
Theorem transf_c_program_match:
  forall p tp,
  transf_c_program p = OK tp ->
  match_prog p tp.
Proof.
  (* [...] *)
Qed.

Theorem c_semantic_preservation:
  forall p tp,
  match_prog p tp ->
  backward_simulation (Csem.semantics p) (Asm.semantics tp).
Proof.
  (* [...] *)
Qed.

(** Combining the results above, we obtain semantic preservation for two
  usage scenarios of CompCert: compilation of a single monolithic program,
  and separate compilation of multiple source files followed by linking.

  In the monolithic case, we have a whole C program [p] that is
  compiled in one run of CompCert to a whole Asm program [tp].
  Then, [tp] preserves the semantics of [p], in the sense that there
  exists a backward simulation of the dynamic semantics of [p]
  by the dynamic semantics of [tp]. *)
Theorem transf_c_program_correct:
  forall p tp,
  transf_c_program p = OK tp ->
  backward_simulation (Csem.semantics p) (Asm.semantics tp).
Proof.
  intros. apply c_semantic_preservation. apply transf_c_program_match; auto.
Qed.
```

Main Proof (2)

```
(** This is the list of compilation passes of CompCert in relational style.
  Each pass is characterized by a [match_prog] relation between its
  input code and its output code.  The [mkpass] and [:::] combinators,
  defined in module [Linking], ensure that the passes are composable
  (the output language of a pass is the input language of the next pass)
  and that they commute with linking (property [TransfLink], inferred
  by the type class mechanism of Coq). *)
  Definition CompCert's_passes :=
    mkpass SimplExprproof.match_prog
::: mkpass SimplLocalsproof.match_prog
::: mkpass Cshmgenproof.match_prog
::: mkpass Cminorgenproof.match_prog
::: mkpass Selectionproof.match_prog
::: mkpass RTLgenproof.match_prog
::: mkpass (match_if Compopts.optim_tailcalls Tailcallproof.match_prog)
::: mkpass Inliningproof.match_prog
::: mkpass Renumberproof.match_prog
::: mkpass (match_if Compopts.optim_constprop Constpropproof.match_prog)
::: mkpass (match_if Compopts.optim_constprop Renumberproof.match_prog)
::: mkpass (match_if Compopts.optim_CSE CSEproof.match_prog)
::: mkpass (match_if Compopts.optim_redundancy Deadcodeproof.match_prog)
::: mkpass Unusedglobproof.match_prog
::: mkpass Allocproof.match_prog
::: mkpass Tunnelingproof.match_prog
::: mkpass Linearizeproof.match_prog
::: mkpass CleanupLabelsproof.match_prog
::: mkpass (match_if Compopts.debug Debugvarproof.match_prog)
::: mkpass Stackingproof.match_prog
::: mkpass Asmgenproof.match_prog
::: pass_nil _.

(** Composing the [match_prog] relations above, we obtain the relation
  between CompCert C sources and Asm code that characterize CompCert's
  compilation. *)
Definition match_prog: Csyntax.program -> Asm.program -> Prop :=
  pass_match (compose_passes CompCert's_passes).
```

Outline

Verified Compilers

Project Cristal/CompCert
- General info
- Compiler Structure
- Coq, excerpt
- Code Examples
- Extensions/Other Works

Extensions

- CompCertTSO (threads and shared memory)[19]
- More concrete memory model (finite)[20]
- SepCompCert (separate compilation)[21]
- CompCertS (memory consumption preserved)[22]
- Cryptography: Preserve constant time (sidechannel attacks)[23]
- CompCertM (linking C & Assembly)[24]
- SSA & Optimizations[25]
- CompCertO (certified C components, preserving interaction)[26]
- CompCert-KVX (VLIW)[27]
- Nominal memory model (reason about sub-regions of memory, for concurrent programs)[28]

Thank you for your attention!

Outline

References

References I

[1] John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. volume 19, pages 33–41. American Mathematical Society.

[2] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. 7(3):51–70.

[3] Maulik A. Dave. Compiler verification: a bibliography. 28(6): 2–2. ISSN 0163-5948. doi: 10.1145/966221.966235. URL https://dl.acm.org/doi/10.1145/966221.966235.

[4] Xavier Leroy. A formally verified compiler back-end. 43(4): 363–446. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-009-9155-4. URL http://link.springer.com/10.1007/s10817-009-9155-4.

References II

[5] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. 29. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796818000229. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/verified-cakeml-compiler-backend/E43ED3EA740D2DF970067F4E2BB9EF7D. Publisher: Cambridge University Press.

[6] Andreas Lööw. Lutsig: a verified verilog compiler for verified circuit development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 46–60. Association for Computing Machinery. ISBN 978-1-4503-8299-1. doi:

References III

10.1145/3437992.3439916. URL https://doi.org/10.1145/3437992.3439916.

[7] The CertiCoq project, . URL https://certicoq.org/.

[8] The CertiCoq pipeline · CertiCoq/certicoq wiki, . URL https://github.com/CertiCoq/certicoq.

[9] CompCert manual. URL https://compcert.org/man/manual.pdf.

[10] AbsInt. Factsheet CompCert c compiler. page 3.

[11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. 46(6): 283–294. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL https://doi.org/10.1145/1993316.1993532.

References IV

[12] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 1–17. Springer. ISBN 978-3-642-19718-5. doi: 10.1007/978-3-642-19718-5_1.

[13] Airbus. URL `https://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf`.

[14] Tu munich, flight system dynamics. URL `https://www.absint.com/tum_fsd.htm`.

[15] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669.

References V

[16] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 586–601. Association for Computing Machinery. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062358. URL https://doi.org/10.1145/3062341.3062358.

[17] Mtu friedrichshafen, nuclear power plant. URL https://www.absint.com/mtu_fh.htm.

References VI

[18] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. page 1. URL https://hal.inria.fr/hal-01643290.

[19] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *ACM*, 60 (3):1–50, 2013. ISSN 0004-5411, 1557-735X. doi: 10.1145/2487241.2487248. URL https://dl.acm.org/doi/10.1145/2487241.2487248.

References VII

[20] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A concrete memory model for CompCert. In Springer, editor, *ITP 2015 : 6th International Conference on Interactive Theorem Proving*, volume Lecture Notes in Computer Science (LNCS) of *Interactive Theorem Proving*, pages 67–83, . doi: 10.1007/978-3-319-22102-1_5. URL https://hal.inria.fr/hal-01194549. Issue: 9236.

[21] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 178–190. Association for Computing Machinery. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837642. URL https://doi.org/10.1145/2837614.2837642.

References VIII

[22] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. CompCertS: A memory-aware verified c compiler using a pointer as integer semantics. 63(2):369, . doi: 10.1007/s10817-018-9496-y. URL https://hal.inria.fr/hal-02401182.

[23] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. 4:7:1–7:30. doi: 10.1145/3371075. URL https://doi.org/10.1145/3371075.

[24] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: CompCert with c-assembly linking and lightweight modular verification. 4:23:1–23:31. doi: 10.1145/3371091. URL https://doi.org/10.1145/3371091.

References IX

[25] Jean-Christophe Léchenet, Sandrine Blazy, and David Pichardie. A fast verified liveness analysis in SSA form. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, Lecture Notes in Computer Science, pages 324–340. Springer International Publishing. ISBN 978-3-030-51054-1. doi: 10.1007/978-3-030-51054-1_19.

[26] Jérémie Koenig and Zhong Shao. CompCertO: compiling certified open c components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1095–1109. ACM. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454097. URL https://dl.acm.org/doi/10.1145/3453483.3454097.

References X

[27] Cyril Six. Optimized and formally-verified compilation for a VLIW processor. URL https://hal.archives-ouvertes.fr/tel-03326923. Issue: 2021GRALM025.

[28] Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. Verified compilation of c programs with a nominal memory model. 6:25:1–25:31. doi: 10.1145/3498686. URL https://doi.org/10.1145/3498686.