Verifying Fast and Sparse SSA-based Optimizations in Coq [⋆]

Delphine Demange[1], David Pichardie[2], and Léo Stefanesco[3]

[1] Université Rennes 1 – IRISA – Inria
[2] ENS Rennes – IRISA – Inria
[3] ENS Lyon

Abstract The Static Single Assignment (SSA) form is a predominant technology in modern compilers, enabling powerful and fast program optimizations. Despite its great success in the implementation of production compilers, it is only very recently that this technique has been introduced in *verified* compilers. As of today, few evidence exist on that, in this context, it also allows faster and simpler optimizations. This work builds on the CompCertSSA verified compiler (an SSA branch of the verified CompCert C compiler). We implement and verify two prevailing SSA optimizations: Sparse Conditional Constant Propagation and Global Value Numbering. For both transformations, we mechanically prove their soundness in the Coq proof assistant. Both optimization proofs are embedded in a single sparse optimization framework, factoring out many of the dominance-based reasoning steps required in proofs of SSA-based optimizations. Our experimental evaluations indicate both a better precision, and a significant compilation time speedup compared to previous verified techniques.

1 Introduction

Single Static Assignment (SSA) is an intermediate representation of code in which variables are assigned at most once in the program text, and ϕ-functions are used to merge values at control-flow join points. Introduced in the early 1990's by Cytron et al [7], it has gained over the years a considerable interest in the compilation community. Indeed, although the static single assignment property looks simple, it entails fundamental structural properties in the program control-flow graph. These properties, materialized by e.g. the dominator-tree, or use-def chains, are in turn smartly exploited by program optimizations, whose implementations become simpler than on regular, non-SSA programs. In a way, converting a program into SSA can be seen as a pre-processing that embeds, explicitly in the program syntax, some rich semantic invariants of the program. By the same token, SSA-based optimizations can enjoy precision and efficiency improvement. It is hence not surprising that SSA has constituted, for over a decade

[⋆] This work was supported by Agence Nationale de la Recherche, grant number ANR-??? Discover.

now, the state-of-the-art technique in modern, production compilers, such as GCC or LLVM. For instance, LLVM optimization middle-end includes numerous optimizations (25+ phases), including dead code elimination, loop invariant code motion, sparse conditional constant propagation, and aggressive common sub-expression elimination based on global-value numbering, all of them working on SSA. Moreover, SSA is increasingly used in just-in-time (JIT) compilers, operating on high-level target-independent program representations (e.g. Java byte-code, .NET CLI byte-code, or LLVM bitcode), which gives even stronger evidence of the efficiency of SSA-based optimizations.

Undoubtedly, though these sophisticated optimizations are conceptually simpler, implementing them is far from trivial. Indeed, they exploit the subtle semantic invariants of the SSA form, and rely on highly efficient data structures for better performance. In the literature, it is well-known that the simplicity of SSA has sometimes been over-estimated, and designing bug-free (i.e. semantics-preserving) implementations is not so-easy. The recent work of Yang et al. [19] shows that bugs remain frequent in mainstream compilers. Compiler correctness aims to provide rigorous proofs that compilers preserve the behavior of programs they compile. After 40 years of rich history, the field is entering into a new era, with the advent of realistic and mechanically verified compilers. This new generation of compilers was initiated with CompCert [10], a compiler that is programmed and verified in the Coq proof assistant and generates compact and efficient assembly code for from C. The CompCert project has now reached the maturity to compete with non-verified compilers, such as GCC. However, it does not rely on an SSA-based middle-end.

Recently, the Vellvm [21, 20] and CompCertSSA [2] projects have been conducted, trying to introduce SSA-based compilation techniques in *verified* compilers. Despite the considerable progresses that these work made on the formalization of the semantics of SSA, and of several important of its properties, SSA-based verified compilers still suffer from two main bottlenecks, that clearly limit their application in real world scenarios. First, on the implementation side of things, verified compilers are usually restricted. Indeed, verified compilers must find a balance between efficiency and verifiability, and directly proving the correctness of the transformations they perform often requires to consider less optimized (i.e. less efficient) implementations. To by-pass this problem for the most efficiency-critical parts of the compiler, one can employ the technique of *translation validation*[12]. In this setting, an un-verified tool performs the required computations, and a verified checker ensures, *a posteriori*, the correctness of these results before they are put back in the verified tool chain, thus providing the same formal guaranties as a transformation that would be directly programmed and proved in Coq. This technique is increasingly favored in mechanically verified developments [17, 16]. We argue that this technique allows to achieve good performance in practice: the compilation overhead introduced by the checker does not exceed the performance loss induced by implementations that are easier to verify but less efficient. The second obstruction to the development of SSA-based verified compilers lies in the fact that, when it comes to proving, working on SSA can be quite constraining. In fact, the structural properties provably holding on the input program must be proved to be preserved by each transformation. In addition, compared to pen-and-paper proofs in which some technical arguments can be elided, mechanizing proofs requires to make explicit every single reasoning steps. Previous proof efforts on SSA already provide some general lemmas and proof architectures (e.g. the equation lemma of [2], or the scoping lemma about SSA strictness of [20]). However, they lack a systematic, formalized proof technique that would follow the usual dominance-based reasoning steps one has to take when proving SSA-based optimizations[4].

This present work tries to make some progress in these two directions. More specifically, after recalling in Section 2 some background about the CompCert compiler, our on-going CompCertSSA project, and a brief overview of the two optimizations we consider in this paper, we present the following contributions in verified SSA-based optimizations. We provide realistic implementations, in a verified compiler chain, of leading SSA optimizations, namely Sparse Conditional Constant Propagation (SCCP) and a Common Sub-Expression Elimination based on Global Value Numbering (GVN). Their implementations closely follow the choices made in production compilers, for techniques of intra-procedural and scalar optimizations. Hence, they are realistic in terms of efficiency (compilation time) and precision (number of instructions optimized). We resort on the use of efficient, verified, a-posteriori validators that do not practically penalize compilation time, even in regards of the efficient optimization implementations. On the proof side, we propose a generic proof framework (Section 3) that makes explicit the reasoning on dominated regions, an emblematic reasoning schema of paper-proofs of SSA optimizations. Factoring out many of domination-based reasoning makes the proof effort more lightweight. The proof framework also captures the SSA sparseness *adage*. Indeed, our framework is parameterized by a generic, flow-insensitive, static analysis underlying the optimization. We prove that, at each program point, it is sufficient to establish the correctness of the analysis for the variables that strictly dominates this program point. The correctness proofs of SCCP and GVN (Sections 4 and 5) are done by instantiating the framework on these two optimizations, and their underlying static analysis. All our proofs are done within the Coq proof assistant, extending the CompCertSSA middle-end, an extension of the verified CompCert C compiler. Finally, we conduct an experimental validation of the Ocaml extracted compiler on a benchmark suite (Section 6), demonstrating that our middle-end is able to scale properly to large programs, with improved optimization opportunities. Our full development is available online at http://www.irisa.fr/celtique/ext/ssa_opt.

2 Background

2.1 The verified CompCert compiler

CompCert is a realistic, formally verified compiler that generates PowerPC, ARM or x86 code from source programs written in a large subset of C. Com-

[4] Both work identified the need of such a framework, and the benefits it would permit.

pCert formalizes the operational semantics of dozen intermediate languages, and proves a semantics preservation theorem for each phase.

Preservation theorems are expressed in terms of program behaviors, i.e. finite or infinite traces of external function calls (a.k.a. systems calls producing observable events), that are performed during the execution of the program, and claim that individual compilation phases preserve behaviors.

A consequence of the theorems is that for any C program p that does not go wrong (i.e. it does not reach a non-final state where no execution step is valid), and target program tp output by the successful compilation of p by the compiler compcert_compiler, the set of behaviors of p contains all behaviors of the target program tp. The formal theorem is:

```
Theorem compcert_compiler_correct: forall (p: C.program) (tp: Asm.program),
  (not_wrong_program p /\ compcert_compiler p = OK tp) ->
  (forall beh, exec_asm_program tp beh -> exec_C_program p beh).
```

Each phase of the compiler is formally proved relying on simulation techniques, and the formal development of CompCert provides the general correctness theorems of these simulation diagrams. We will build on these generic lemmas to prove the semantic preservation of GVN and SCCP (see Sections 4 and 5). The main lemmas to prove take the form of a forward lock-step simulation:

```
Variable prog:program. (* initial program *)
Variable tprog:program. (* target program *)
Hypothesis opt_ok: optimization prog = OK tprog. (* optim. succeeded *)

Lemma match_step : forall s1 t s2 s1’,
    (step (genv prog) s1 t s2) /\ (match_states s1 s1’) ->
    exists s2’, step (genv tprog) s1’ t s2’ /\ match_states s2 s2’.
```

where binary relation match_states between semantic states (before and after optimization) carries the invariants needed for proving behavior preservation.

Some parts of the CompCert compiler are not directly proved in Coq. This is the case of the register allocation, which is based on a graph coloring algorithm. The interference graph coloring algorithm is written in OCaml, and then validated a posteriori by a checker written in Coq [13]. The correctness proof of the checker (stating that if a coloring is accepted by the validator, then it is indeed a valid coloring) ensures this compilation phase provides the same guarantees as a transformation written and proved directly in Coq, with the additional benefit of abstracting away complex implementation details and heuristics.

2.2 The verified CompCertSSA compiler

In previous work [2], we have develop CompCertSSA, that builds on top of CompCert, by enriching it with an SSA-based middle-end. It is plugged in at the level of RTL (a non-structured, CFG based, three-address like representation), and generates from it the SSA intermediate form. After optimizing on the SSA

```
Definition reg := ...                         (* type of variables *)
Inductive instr  :=                           (* instructions (excerpt) *)
| Inop (pc: node)
| Iop (op: operation) (args: list reg) (res: reg) (pc: node)
| Iload (chk:chunk) (addr:addressing) (args: list reg) (res: reg) (pc: node)
| Istore (chk:chunk) (addr:addressing) (args:list reg) (src: reg) (pc: node)
| Icall (sig: signature) (fn:ident) (args: list reg) (res: reg) (pc: node)
| Icond (cond: condition) (args: list reg) (ifso ifnot: node)
| Ireturn (src: option reg).

Definition code := PTree.t instr.             (* type of code graph *)

Inductive phiinstr :=
 | Iphi (args: list reg) (res: reg).          (* ϕ-functions *)
Definition phiblock:= list phiinstr.          (* type of ϕ-blocks *)
Definition phicode := PTree.t phiblock.       (* type of ϕ-blocks graph *)

Record function := {
   fn_sig: signature;         (* function signature *)
   fn_params: list reg;       (* parameters *)
   fn_stacksize: Z;           (* activation record size *)
   fn_code: code;             (* code graph *)
   fn_phicode: phicode;       (* ϕ-blocks graph *)
   fn_entrypoint: node        (* entry node *)
}.
```

Figure 1. SSA abstract syntax

form, the middle-end deconstructs it back to RTL, and then leaves the remainder of CompCert's backend generating machine code. In this section, we recall the required material and results achieved in this previous work.

The SSA language The abstract syntax of the SSA form is given in Figure 1. Functions (`function` records), are defined at the bottom of the figure. Their code is organized into two distinct graphs: one for the regular instructions (of type `instr`), and another one for ϕ-blocks (of type `phiinstr`). The idea is to attach a ϕ-block at node `pc` whenever the ϕ-block must be executed before the regular instruction at node `pc`. We will present in more detail the semantics of this language in the next paragraph.

In addition, we equip the notion of SSA programs with a *well-formedness* predicate capturing essential structural properties of SSA forms. First, it requires the single static assignment property of the function (we omit the formal definition). Next, it demands that the function is in strict SSA form: each variable use must be dominated by its (unique) definition point. Formally:

```
Definition strict (f: function) : Prop :=
  forall (x:reg) (u d: node), (use f x u) /\ (def f x d) -> dom f d u.
```

```
Inductive state :=
 | State (stack: list stackframe)  (* call stack *)
         (f: function)              (* current function *)
         (sp: val)                  (* stack pointer *)
         (pc: node)                 (* current program point *)
         (rs: regset)               (* register state *)
         (m: mem)                   (* memory state *)
 | Callstate (stack: list stackframe) (f: fundef) (args: list val) (m: mem)
 | Returnstate (stack: list stackframe) (v: val) (m: mem).

Inductive step: genv -> state -> trace -> state -> Prop :=
 | ex_Inop_njp: forall ge s f sp pc rs m pc',
     fn_code f pc = Some(Inop pc') ->
     ~ join_point pc' f ->
     step ge (State s f sp pc rs m) nil (State s f sp pc' rs m)

 | ex_Inop_jp: forall ge s f sp pc rs m pc' phib k,
     fn_code f pc = Some(Inop pc') ->
     join_point pc' f ->
     fn_phicode f pc' = Some phib ->
     index_pred f pc pc' = Some k ->
     step ge (State s f sp pc rs m) nil (State s f sp pc' (phistore k rs phib) m)

 | ex_Iop: forall ge s f sp pc rs m pc' op args res v,
     fn_code f pc = Some(Iop op args res pc') ->
     eval_operation sp op (rs##args) m = Some v ->
     step ge (State s f sp pc rs m) nil (State s f sp pc' (rs#res <- v) m)
```

Figure 2. Semantics of SSA (excerpt).

Finally, it requires that the instruction code of the function is normalized, in the following sense: the only possible instruction that can lead to a junction point in the CFG of the function is an `Inop`. This design choice can look quite anecdotal, but this greatly simplifies the definition of the semantics (ϕ-blocks can only be executed after an `Inop`), and subsequently the proofs about SSA optimizations. Note that these `Inop` will be easily get rid of by subsequent compilation phases.

SSA semantics The SSA language is provided with a small-step operational semantics, given in Figure 2. Here, we only describe the semantic states, and the main cases in the definition of the transition relation. We refer the reader to the full development for extra details. Depending on the execution phase of the program, there are three possible kinds of execution states: (i) regular, intermediate execution states (constructor `State`), (ii) call states (constructor `Callstate`), reached immediately after executing a function call, indicating the next function to execute and (iii) return states (`Return`), indicating, in addition to the current state of the stackframe and memory, the potential value to return.

Then, the small-step semantic transition relation, `step`, formalizes what it means for each instruction to be executed. For instance, in Figure 2, executing an `Inop`, when no ϕ-block is attached to the successor `pc'` of `pc`, just leaves the semantic state unchanged, except for the program pointer. If `pc'` is a junction point (rule `ex_Inop_jp`), then the ϕ-instructions in the ϕ-block `phib` will be executed on local registers `rs`, through the function `phistore`. This function basically performs the parallel copy of the `k`-th arguments of ϕ-functions to their respective destination registers. All other instructions have the expected, traditional operational semantics. For instance, executing an `Iop` instruction (rule `ex_Iop`) evaluates the operator `op` on the values of its arguments `args` in the current register state `rs`, and updates `rs` by setting the destination register `res` to the result value `v`. For the rules we selected, no observable event is produced, hence the empty trace `nil` is emitted.

Equation lemma The main result we previously achieved is the so-called equation lemma. This semantic lemma establishes a strong, global invariant, that allows to see SSA function as a set of equations relating variables and the right-hand side of their defining instructions. Its formal statement is indicated below. It considers well-formed SSA programs (all of its functions are well-formed), and states that in any reachable execution state, if a variable `x` is defined at point `d` in function `f` (condition `(def f x)`), then the value of `x` in this state evaluates to `(rhs f x i)` (typically an arithmetic instruction `Iop`) in that exact same state, provided that execution state is in a region of the CFG that is strictly dominated by `d` (condition `(sdom f d pc)`).

```
Definition eq_lemma f sp rs pc := forall x d i,
    (def f x d) /\ (rhs f x i) /\ (sdom f d pc) ->
    [f, sp, rs]|= x == i.

Theorem reachable_eq_lemma : forall prog s f sp pc rs m,
    (wf_ssa_program prog) /\ (reachable prog (State s f sp pc rs m)) ->
    eq_lemma f sp rs pc.
```

This lemma makes it clear that syntactic information in SSA functions is rich, thanks to dominance-based structural properties of their CFG. This is what makes SSA so easy to manipulate in program optimizations. In our proof framework, we aim at exploiting the semantic counterparts of these constraints, to simplify our proofs. Indeed, we will make extensive use of the above invariant on SSA program in the proof of GVN (Section 5), and our framework helps to systematize the dominance-based reasoning steps.

2.3 SSA-based optimization

Flow-insensitive analyses are both simpler to implement and less memory expensive as their flow-sensitive counterparts, while giving rise to the same precision in SSA. SSA also provides a simplified notion of def-use chains that can be exploited to speedup fixpoint iteration. Below we briefly overview the two optimizations we consider in this paper.

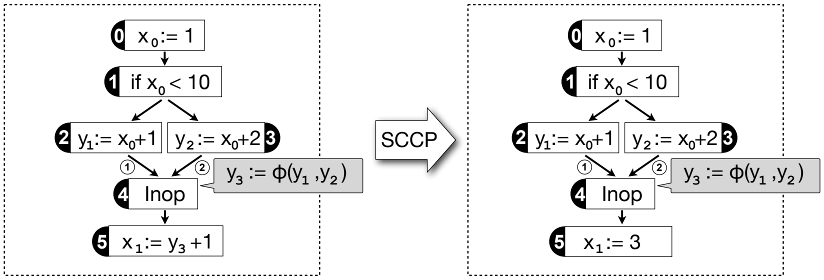

Figure 3. Example of constant propagation (SCCP algorithm)

Sparse Conditional Constant Propagation (SCCP) Constant propagation is a key compiler optimizations. It discovers if variable assignments will be assigned the same constant value on all feasible paths reaching that assignment. When such an assignment is detected, it can be replaced by a simpler instruction, that just assigns that constant value (instead of a more intricate expression) to the variable. Modern compilers like GCC and LVVM implement constant propagation using the Wegman-Zadech algorithm called Sparse Conditional Constant Propagation [18].

SCCP is a very fast constant propagation analysis that is able to perform a program transformation in almost linear time (size of the CFG, plus size of the SSA graph). It non only detects constants but also some unfeasible branches. Dead code and constant analysis are performed simultaneously, so that they benefit one from each-other.

Figure 3 illustrates this mutual benefit. In order to discover the constant 3 at node 5, it is necessary to prove that edge $(1, 3)$ is not feasible. This fact is discovered thanks to the propagation of the constant equality $x_0 = 1$ from node 0 to the conditional statement at node 1. While iterating traditional constant propagation and program simplification could achieve the same result, SCCP is able to generate it in one (fast) run.

SCCP is traditionally implemented with an *ad hoc* iterative workset algorithm. The computation maintains three worksets: $w_\top$ is a set of SSA variables that have been assigned a "I don't know" information ($\top$); w_{var} is a set of SSA variables whose constant information may depend on recently updated variables and must hence be reconsidered in a future iteration of the algorithm; w_{edges} is a workset of feasible edges. Initially, the entry edge is considered as feasible and every function parameter is assigned a $\top$ information. Elements in $w_\top$ are processed in priority during each round, as they may speedup fixpoint convergence. When the abstract information of a variable belonging to either $w_\top$ or w_{var} is updated, the algorithm exploits on a SSA def-use chains data-structure to directly wake up the recomputation of the abstract information associated to the variables which depend on that variable. When an edge in w_{edges} is considered, we only add to the workset the successor edges that are feasible according to the current abstract information given by each variable used in this node.

Global Value Numbering (GVN) Global Value Numbering [1, 4] is a common subexpression elimination optimization that discovers equivalence classes between program variables. Variables belonging to the same class evaluate to the same value. Each class is given a *number* that characterizes it.

Several implementation techniques has been proposed to perform a fast numbering on an SSA program. The technique chosen by the current version of the LLVM compiler is the RPO algorithm [4]. It scans the CFG of the program in reverse-post-order and manages the numbering with a mutable hash-table assigning a number to each symbolic expression encountered in the program syntax. A complete explanation of the algorithm is out of the scope of this paper but two facts are worth mentioning about it. First, efficient implementations require mutable data-structures like hash-tables, which are not currently available when programming in Coq. The use of an external GVN solver, written in OCaml, is thus mandatory to achieve the efficiency of modern compilers. Second, the analysis does not fit the classical monotone framework generally considered in verified static analysis [10]: the computed fixpoint is wrong if not built using the RPO order, which makes a direct proof of this algorithm particularly difficult. Therefore, GVN is a perfect candidate for a posteriori validation.

CompCert includes a common subexpression elimination optimization based on Local Value Numbering (LVN). It does not work on SSA but applies on extended basic blocks only, and does not infer equalities across loop boundaries. Still, it handles intra-procedural redundant load elimination; GVN would require major adaptations. This extension is out of scope of this paper.

3 Generic framework

We present now the general framework, in which we embed the formalization of SCCP and GVN. It is intended to capture a variety of SSA-based optimizations, and to provide the backbone of their correctness proof, by factoring out many of the required dominance-based reasoning steps.

It is made of three parts. The first part consists of the description of a generic optimization, satisfying some basic constraints ensuring the preservation of strict-SSA well-formedness. This optimization relies on the result of a static analysis, whose formalization, the second part of the framework, axiomatizes some of its properties and invariants. The last part of the framework is dedicated to the proof of a dominance-based invariant correctness result of the analysis, under the assumption that the analysis conforms to its specification.

The formalization of the analysis correctness invariant relies on a 3-place predicate `(dsd f x n)`, that holds whenever in function `f`, the definition point of variable `x` strictly dominates the CFG node `n`. In our framework, we provide general lemmas about that predicate, and case-analysis proof schemes, that help structuring proofs. Intuitively, dominance-based reasoning is relatively easy for straight-line code, but conducting proofs in Coq can sometimes add a significant overhead. Reasoning about join points makes the reasoning even more intricate. In our development, we resort on the following two lemmas

```
Lemma dsd_not_joinpoint : forall f n1 n2 x,
    (is_edge f n1 n2) /\ (~join_point n2 f) /\ (dsd f x n2) ->

    (assigned_code f n1 x)
    \/ (ext_params f x /\ n1 = fn_entrypoint f)
    \/ (dsd f x n1 /\ ~ assigned_code f n1 x)

Lemma dsd_joinpoints : forall f n1 n2 x,
    (is_edge f n1 n2) /\ (join_point f n2) /\ (dsd f x n2) ->

    (assigned_phi f n2 x)
    \/ (ext_params f x /\ n1 = fn_entrypoint f)
    \/ (dsd f x n1 /\ ~ assigned_phi f n2 x).
```

which provide helpful case-analysis schemes. When proving lemmas taking the form of a subject-reduction property under the hypothesis that `(dsd f x n1)`, each of the cases provide sufficient information for either knowing exactly the definition point of register `x`, or knowing that definition of `x` strictly dominates one of the predecessors of $n1$, allowing to use the `dsd` hypothesis to conclude.

Generic optimization The generic SSA-based optimization first assumes that the underlying static analysis has the following type:

```
Variable approx : Type.
Definition res := reg -> approx.
Variable analysis : function -> (res * m_exec).
```

It takes an SSA function as input, and returns (i) a flow-insensitive result (of type `res`), mapping to SSA variables an element of type `approx` (typically, an abstract domain formalized as a lattice) and (ii) a map (of type `m_exec`), from control-flow edges to execution flags (booleans) indicating feasibility of edges. In the most general case, the function `analysis` will compute simultaneously these two pieces of information so that the two corresponding static analyses can interact and benefit one from each other. On top of the analysis, we assume that the optimization relies on a per-instruction transformation function `transf_intr`, that is mapped on the whole SSA code. More specifically:

```
Variable transf_instr : res -> node -> instruction -> instruction.
Definition transf_function (f: function) : function :=
  let (res,exec) := analysis f in
  map_code (transf_instr res) f.
```

Result `exec` is not used by `transf_instr`, but improves precision of `res`, and is kept track of for proof purposes. On top of these basic assumptions, we require that for each instruction optimized by `transf_instr`, the changes of variable uses and definitions does not break the strictness of SSA:

```
Hypothesis new_code_same_or_Iop : forall f pc ins,
  (wf_ssa_function f) /\ ((fn_code f)!pc = Some ins) ->
    transf_instr (fst (analysis f)) pc ins = ins
    \/ transf_instr_preserves_strict f ins.
```

```
Record AnalysisProp := {
   exec      : function -> node -> Prop
 ; G         : regset -> approx -> val -> Prop
 ; is_at_Top: result -> reg -> Prop
 ; G_top : forall R r rs,
           is_at_Top R r -> G rs (R r) (rs# r)
 ; is_at_Top_eq : forall R r r',
           (is_at_Top R r) /\ (R r = R r') -> is_at_Top R r'
 ; A_intra : forall f pc r,
           (exec f pc) /\ (assigned_inter_mem_params f pc r) ->
           is_at_Top (A_r f) r }.
```

Figure 4. Axiomatisation of the generic analysis.

where, predicate transf_instr_preserves_strict means that the optimization can change any local variable definition for a simpler statement of the form Iop (e.g. an arithmetic constant or a register move) assigning the same variable, so long as all newly introduced uses keeps dominated by their definition. Other statements are not allowed to be optimized ((un)-conditional branches stay untouched, we decide to focus on optimizations that do not change functions CFG).

Under the hypothesis new_code_same_or_Iop, we can prove that the generic optimization (mapped to all functions of a given program) preserves the well-formedness of the initial SSA program:

```
Theorem transf_program_preserve_wf_ssa : forall prog,
  wf_ssa_program prog -> wf_ssa_program (transf_program prog).
```

While this lemma looks quite anecdotal, it is absolutely necessary to be able to compose several SSA optimizations passes. In addition, it has a high practical impact. Indeed, once the optimization has been defined with the help of this framework, proving that hypothesis is the only thing we need to get the well-formedness preservation. Without this framework, the proof of this result would need to be duplicated for every optimization. It hence allows to focus the proof effort on the more interesting aspects.

Analysis specification We turn now our attention to the axiomatic specification of the analysis function. In the sequel, to lighten the notations, we will assume to work only with well-formed SSA functions, and will write (A_r f) for the first component of (analysis f).

This specification is packed into the Coq record shown in Figure 4. First, we need to formulate the interpretation of the execution flags map returned by the analysis of a function. Hence, we assume a 2-place predicate (exec f pc), characterizing executable CFG nodes. Essentially, it must be proved coherent with the dynamic semantics of the function, i.e. the analysis must not infer a node as not non-executable if its predecessor in the CFG is analyzed as executable, and the function can make as step from the predecessor to that node.

The main part of the axiomatisation consists in specifying a concretisation relation between abstract values associated to SSA variables and concrete, run-time values they can take. This is done by predicate `(G rs a v)`. It is intended to hold whenever, in a context described by register state `rs`, the abstract value `a` is a correct approximation of the concrete value `v`.[5]

The third component we require is predicate `(is_at_Top R r)`, whose intent is to characterize when, in a given result `R`, a register `r` is associated to the static information ”I don’t know”. The type of this predicate alone is not sufficient to express this. We hence include in the specification record a proof obligation (field `G_top`) asking that a register whose analysis result is at $\top$ concretises to any possible value (`rs# r`, where register state `rs` is universally quantified).

Field `is_at_Top_eq` is required for more technical reasons that the others, but is quite natural to have, and can read as sanity check on the definition of `is_at_Top`. This proof obligation asks that, whenever a register `r` is associated to $\top$ for a given result `R`, then any other register `r'` whose static information is equal to the one of `r` is also associates to $\top$ in `R`.

The last field of the specification record, `A_intra`, is a proof obligation saying that the analysis under consideration is intra-procedural, and deal with local variables of the function only. Indeed, we require that for any register `r` of the function, whenever it is a function parameter, or its value depends on the memory or function calls, then the analysis infers a $\top$ information for it. This is only required for registers defined at executable CFG nodes.

Generic analysis correctness proof Assuming that the generic analysis fits in `AnalysisProp`, proving the (instantiated) optimization requires to propagate the correctness of the analysis. We state this as an invariant of its result:

```
Definition gamma (f:function) (pc:node) (rs:regset) :=
 forall x, (dsd f x pc) /\ (exec f pc) ->  G rs (A_r f x) (rs# x).
```

where predicate `(G rs (A_r f x) (rs# x))`, reads as ”the static information computed for register `x` correctly approximates the concrete run-time value of `x` in register state `rs`”. We must stress the fact that, as can be seen in this definition, the correctness of the analysis needs only to hold on variables whose definition dominates the current program point `(dsd f x pc)`, and only when `pc` has been analysed as executable by the analysis `(exec f pc)`.

The final invariance theorem we want to achieve in the framework is the correctness of the analysis (in the sense of `gamma`), for any state reachable during the execution of the program:

```
Theorem analysis_correct : forall prog s f sp pc rs m,
    reachable prog (State s f sp pc rs m) -> gamma f pc rs.
```

To do so, the analysis must satisfy two extra properties (giving rise to two other proof obligations). First, one must show that the analysis must compute a correct abstraction for `Iop` instructions:

[5] Our development also keep track of a global environment and stack pointer to, eventually, deal with symbolic information about read-only globals and offsets values.

```
Hypothesis iop_correct : forall f pc op args res pc’ v rs ge sp m x,
    forall (SINV: eq_lemma f sp rs pc)
           (CODE: (fn_code f) ! pc = Some (Iop op args res pc’))
           (EVAL: eval_operation ge sp op (rs ## args) m = Some v)
    (gamma f pc rs) /\ (exec f pc) /\ (dsd f x pc’) ->
    G (rs # res <- v) (A_r f x) ((rs # res <- v) # x).
```

which can read as follows: if `gamma` holds before executing the `Iop` instruction, it will holds after its execution, in the updated register state. In particular (when `x` and `res` are equal), the static information computed for `res` correctly approximates the concrete value `v` obtained by executing the instruction. Note the `SINV` hypothesis, which makes possible to exploit the equation lemma of the current function. The second proof obligation requires the `gamma` predicate to be preserved by ϕ-blocks execution:

```
Hypothesis gamma_step_phi: forall f pc pc’ phib k rs,
    forall (REACHED: reached f pc) (EXE: exec f pc)
           (PC : (fn_code f) ! pc = Some (Inop pc’))
           (PC’: (fn_phicode f) ! pc’ = Some phib)
           (PRED: index_pred f pc pc’ = Some k)
     gamma f pc rs -> gamma f pc’ (phi_store k phib rs).
```

In the next two sections, we explain how to instantiate the framework on SCCP and GVN. Also, each of the section briefly comments on the correctness proof of the optimization itself (its semantics-preserving theorem). For both of them, we show a lock-step forward simulation lemma, where the matching relation between semantics states carries the invariants about (i) the well-formedness of SSA functions, (ii) the equational lemma and (iii) the correctness of the analysis through a `gamma` predicate on the current state.

4 Verifying SCCP in Coq

4.1 Overview of the implementation

As explained in Section 2.3, SCCP simultaneously detects constants and infeasible paths in the control-flow graph of a function, and replaces arithmetic expressions detected to always evaluate to a constant by that constant. More precisely, our SCCP optimization is built from the following constituents.

The type `approx` of the underlying analysis is instantiated to the elements of the semi-lattice of constants. This lattice is rather standard and was already available in the CompCert compiler distribution. We just recall its definition for the sake of completeness[6]:

```
Inductive approx : Type :=
 | Novalue     (* No value possible, code unreachable. *)
 | Unknown     (* All values possible, no compile-time information *)
 | I (i:int)   (* A known integer value. *)
 | F (f:float) (* A known floating-point value. *)
```

[6] The type `approx` is also equipped with the expected partial order and *join* operator.

We implement a data-flow solver on this constant lattice. The dataflow implementation is new. It iterates on both the CFG (for detecting dead branches) and the SSA graph (also called def-use chains) to propagate constant analysis information, following the algorithms described informally in Section 2.3.

The result of the dataflow solver is of the form `(const,exec)` where `const` maps variables to elements of type `approx`, and `exec` stores the execution flag of CFG edges, indicating whether or not an edge may be taken at run-time. We then send the result of the solver to a formally verified checker ensuring this result is a post-fixpoint of the usual equation system for dataflow constant analysis, augmented with extra equations on execution flags.

The optimization itself consists in propagating the constants detected by the analysis. Every `(Iop op args res _)` instruction, where `args` have been inferred to be constant are replaced by a `(Iop (opconst k) nil res _)` instruction. Note that it does not need to optimize instructions on paths inferred as infeasible.

4.2 Correctness proof

The correctness of SCCP is relatively simple, once the post-fixpoint property is proved. Below, we explain how the analysis fits in our framework and give an intuition on how the proof obligations are discharged. The good news is that the instantiation is straightforward and intuitive for an optimization as simple as SCCP (the framework does not introduce extra overhead in the proof effort).

Analysis To instantiate the specification of Section 3, the relation `G` between abstract and concrete values is standard. We reuse the definition from CompCert's Constant Propagation on RTL, and define predicate `is_at_Top` accordingly:

```
Definition G rs a v :=  match a with
                          | Unknown => True    | Novalue => False
                          | I p => v = Vint p  | F p => v = Vfloat p
                        end.
Definition is_at_Top (R: result) (r: reg) : Prop := (R r = Unknown).
```

The interesting proof obligations of `AnalysisProp` are `iop_correct` and `gamma_step_phi`. The crux of the proof of `iop_correct` is that, as the SSA function is strict and well-formed, we know that all of the arguments `args` of the instruction `(Iop op args res pc’)` have a definition that strictly dominates the program point of the instruction. Hence, by hypothesis, we know that the analysis is correct for these, in the previous register state. In addition, the post-fixpoint checker ensures that the abstract value for `res`, `(A_r f res)` is greater (in the constants lattice) than the static evaluation of the operator `op` on arguments `args`. By correctness of the static evaluation, we get that it matches (in the sense of `G`) the concrete evaluation `v` of the instruction. Hence, `(A_r f res)` will, a fortiori, be a correct approximation of `v`. We show that for other registers, the correctness of the approximation is not altered, using the case-analysis `dsd_not_joinpoint`. The first case is a contradiction thanks to the SSA property, the second case is easily discharged by `G_top`, and in the third case, we use the hypothesis on `gamma` in the previous register state to conclude. The proof of `gamma_step_phi` is similar.

Optimization correctness The analysis and optimization described previously satisfy the various proof obligations of Section 3. First, we remark that it is simple to prove that the strictness condition is preserved, as SCCP only removes variable uses, and does not introduce any new definition.

In proving the optimization correct, the main case is where an `(Iop op args r _)` instruction is optimized into a `(Iop (opconst k) nil r _)`. But this is done only if `(A_r f)` returns a constant for the register being assigned. Thanks to the post-fixpoint property, we hence know that its abstract value matches the concrete value `k` we assigned the register to in the optimized function.

5 Verifying GVN in Coq

5.1 Overview of the implementation

Our implementation of GVN follows LLVM design choices. We rely on a reverse-post-order iteration and a mutable hash table that assigns numbers to symbolic expressions. Each number represents an equivalence class for program variables that hold the same runtime value. For each class we choose a representing variable whose definition must dominate all variables in the same class. Having an efficient dominance test is a keystone of the optimization efficiency. We rely on a fast immediate dominator tree [9] computation and a depth graph traversal numbering that allows constant time dominance test. Our GVN does not handle an execution flags map. We hence use the trivial result saying that all edges may be executable.

Then, we implement and prove correct in Coq a checker for that result. The checker is ensuring three properties. First, that the analysis puts in a singleton class any variable assigned through a memory load or function call. For variables defined by means of an `Iop` instruction, the checker ensures that either it is its own representative, or that the following condition is met: whenever at `pc`, we have the instruction `(Iop op args r _)`, and the representative of `r`, written `(A_r f r)`, is not `r` itself, then `(A_r f r)` is defined at a node `pcr` who strictly dominates `pc`, and `(A_r f r)` and `r` are congruent (i.e. the arguments used in their respective defining instruction have a common representative). A similar check is done by the checker on each ϕ-block of the function: a variable defined by a ϕ-instruction at node `pc` has either itself as a representative, or another ϕ-defined variable in the same block, and their respective ϕ-arguments have the same representative. Finally, the checker ensures that representatives are canonical for all classes.

The optimization itself consists in replacing all instructions of the form `(Iop op args r pc')` at a node `pc`, where the operator `op` does not depend on memory, by a simple register move `(Iop OMove nil (A_r f r) pc')`, under the assumption that `r` and `(A_r f r)` are distinct.

5.2 Proof of correctness

Analysis The case of GVN is a bit more intricate than SCCP. The first difficulty we must overcome is to deal with the intrinsic relational nature of GVN. Indeed, in essence, the GVN external tool computes equivalence classes among variables of a SSA function. Our framework as presented earlier strives for simplicity (so that we can factor out as much as possible proofs), and has a more non-relational flavor, as the analysis is supposed to associate an approximation to each variable.

By looking closer at how the optimization utilizes the result of the analysis, we observe that each time an instruction is optimized, an arithmetic operation is replaced by a variable which represents, symbolically, that arithmetic expression. This naturally leads us to formalizing the analysis as associating, to each variable, another variable (its representative), which concretizes to a single value, its value in the current context:

```
Definition approx : Type := reg.
Definition G rs a v : Prop := (rs# a = v).
```

Now, we must characterize the set of variables for which the analysis does not manage to infer any useful information (or ”I don’t know”). Following the same approach, the $\top$ information is associated to a variable if that variable is alone in its equivalence class. Therefore, we define predicate `is_at_Top` as follows:

```
  Definition is_at_Top (R: result) (a: approx) : Prop :=
    (R a = a) /\ (forall a', R a' = a -> a' = a).
```

In this setup, we can prove that the analysis satisfies the first two conditions of `AnalysisProp`. For proving the last obligation, we resort on the specification provably established by our checker. The proof of `iop_correct` relies on the equation lemma of SSA: we need to prove that the value `v` of variable `x` assigned by an `Iop` instruction is correctly abstracted by `(A_r f x)`. In the interesting case, `x` $\neq$ `(A_r f x)`, and the equation lemma applies, since `(A_r f x)` strictly dominates `x`. Hence, we get that `(A_r f x)` equals the evaluation of its defining right-hand side. By the correctness of the checker, `(A_r f x)` is congruent to `x`, and their respective `Iop` arguments have equal representatives. We conclude by using the `gamma` hypothesis and strictness of SSA. Here again, other cases are tackled using `dsd_not_joinpoint` (which applies by the normalization of SSA code). The preservation proof of `gamma` by the execution of ϕ-blocks follows the same idea, using the representatives specification of ϕ-defined variables, and the case-analysis scheme provided by `dsd_joinpoint`.

Correctness of the optimization Here again, the specification enforced by the checker helps us discharge the obligation `transf_instr_preserves_strict`. The proof of semantic preservation of the optimization goes smoothly with the choice of definition for predicate `G`. Indeed, when an `Iop` instruction is optimized, then the variable `x` that it defines will be, after the optimization, defined by variable move from `(A_r f x)` to `x`. By correctness of the analysis, and the

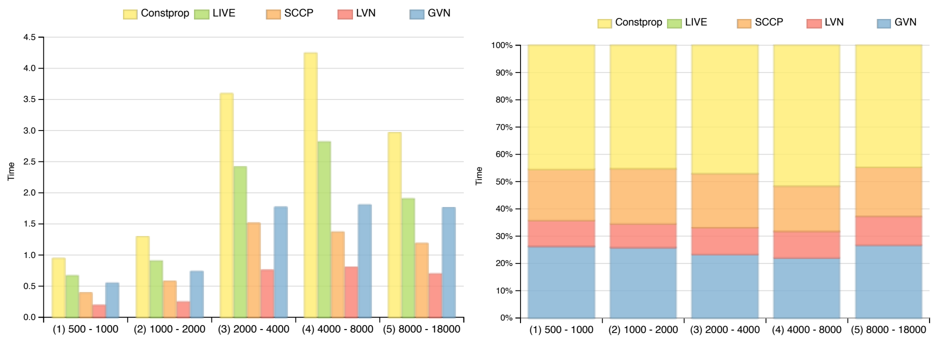

Figure 5. Transformation times. Left: absolute time in seconds. Right: percentage.

definition of G, the arguments of the Iop defining (A_r f x) evaluate to the same values as their representatives. But, by the congruence specification, they also evaluate to the same values as the arguments of the Iop defining x.

6 Experiments

We evaluate the performances of the verified SSA middle-end by extracting its Coq implementation into OCaml code, and running it of some realistic C program benchmarks. These include around 131.000 lines of C code, and fall into the following categories of programs: compression algorithms, a raytracer, the Spass theorem prover, the hmmer and mcf from the SPEC2006 benchmarks and nsichneu and papabench coming from WCET-related reference benchmarks. These programs range from hundreds of lines of C code, to several thousands.

Below, we evaluate the middle-end according to the following criteria: (i) compilation time of SCCP and GVN, compared to the CompCert's corresponding optimizations on the RTL non-SSA form (Constant Propagation and a Common Subexpression Elimination based on a Local Value Numbering), (ii) the efficiency of the SCCP and GVN checkers, relatively to the time required for analysing the code, and optimizing it and (iii) the gain in precision for SCCP and GVN, compared to CompCert's corresponding optimizations.

To evaluate the middle-end scalability in extreme conditions, we force the compiler to always inline functions below 1000 nodes. We classify our results by categories of function size (number of CFG nodes): $[500; 1000[$ (154 functions), $[1000; 2000[$ (73 functions), $[2000; 4000[$ (63 functions), $[4000; 8000[$ (26 functions), $[8000; 18000]$ (8 functions). Experiments are run on a MacBook OSX 10.8.5, 2.9GHz Intel Core i7, 8GB 1600MHz DDR3.

Optimization times Figure 5 shows the cumulative time, in seconds and by category, required to compile all the functions in this category. The left graph measures the absolute time, while the right graph shows the timing distribution among the various optimizations. In the left graph, we additionally show the

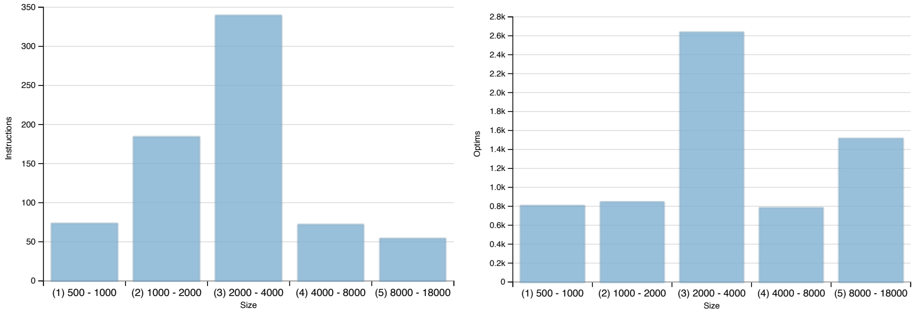

Figure 6. Optimization gain. Left: SCCP. Right: GVN.

time required to perform a liveness analysis on the functions; this liveness analysis is used in CompCert's Constant Propagation to reduce memory usage. As expected, the results show that SCCP, compared to a flow-sensitive analysis like Constant Propagation, scales very well on huge CFG graphs. As for GVN, its computation time is of course higher than the Local Value Numbering of CompCert, but the latter is only block-local, and GVN's computation time keeps reasonable (i.e. below the time-cost of the liveness analysis).

Checkers efficiency On our benchmarks, the SCCP checker represents between 17% and 23% of the whole SCCP optimization, and is amortized as the function CFG grows. The GVN checker represents between 8% and 12% of the whole GVN-based CSE optimization, uniformly on all five categories of function sizes.

Precision For measuring the precision gain brought by SCCP compared to Constant Propagation, we measure, for both optimizations, the number of non-constant `Iop` instructions that are optimized to a numeric, constant `Iop` instruction in the optimized program (and this only for feasible paths, as detected by SCCP). The graph on the left of Figure 6 shows the difference between these two numbers. SCCP detects, on average, around 12% of dead-branches. For measuring the precision of GVN compared to LVN, we count how many arithmetic `Iop` instructions were optimized into register moves, and plot the difference on the right graph of Figure 6. Here again, we observe a substantial gain in precision.

7 Related Work

Most well known achievements in the area of mechanized proof of compilers are the CompCert C compiler [10], Chlipalas's compiler for an impure functional langage [5] and the CakeML compiler [8] that is able to bootstrap itself. All these works are major achievements in verification of semantics preserving transformations but few of them provides advanced program optimizations.

Tristan and Leroy [17, 16] have applied the verified validation approach to instruction scheduling and lazy code motion but their optimizations are more local than GVN, able to infer global loop invariant to perform common subexpression elimination. Leroy has also performed a direct verification of a Local Value Numbering (LVN) optimization [10] without requiring an SSA form but it is limited to extended basic blocks.

The first attempt to formalize SSA semantics was done by Blech et al.[3], using the Isabelle/HOL proof assistant. They verified the generation of machine code from a representation of SSA programs that relies on term graphs. Mansky and Gunter [11] uses Isabelle/HOL to formalize and verify the conversion of CFG programs into SSA. None of these works consider program optimizations.

Zhao et al. [21, 20] formalize the LLVM SSA intermediate form and its generation algorithm in Coq. Their work follow closely the LLVM design and their verified transformation can be run inside the LLVM platform itself. However their extracted verified transformation suffer from strong efficiency limitations. In our previous work, CompCertSSA [2], we also formalise a SSA generation algorithm, using a translation validation technique. To demonstrate the usability of our formal SSA semantics, we prove the soundness of a GVN optimization. The current work provides a major revision of this optimization. We design a new external implementation that follows closely LLVM design and performs an order of magnitude faster than the implementation proposed in [2]. We also redesigned the checker and its soundness proof using our generic framework.

Unverified translation validators have been designed to validate some LLVM optimizations. Stepp et al. [14] uses a technique named Equality Saturation to infer symbolic equalities between source and target. Tristan et al. [15] independently report on a translation validator for LLVM's inter-procedural optimizations, based on Gated-SSA.

8 Conclusion and Perspectives

Our work provides two major verified SSA optimizations. Their implementation closely follows the design choices of realistic compilers (LLVM). We extend the CompCertSSA verified compiler with a new proof framework able to capture the soundness proof of these two optimizations. We also demonstrate the scalability of our optimizations in terms of compiler efficiency and precision.

We foresee two ambitious extensions to this work. First, we would like to extend our optimizations to memory accesses. Modern compilers perform these kinds of memory optimizations but they differ in the way they incorporate alias analysis inside their SSA form. GCC provides a specific program representation with explicit definitions and uses of memory locations. Such a design suffers from compiler memory consumption issues. LLVM proposes a more lightweigth approach with well-chosen queries to alias information. Understanding which approach fits best a verified compiler requires a specific study, taking into account proof engineering and efficiency concerns. A second extension should consider code motion and partial redundancy elimination [6]. GVN provides an important pre-processing for these optimizations.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting Equality of Variables in Programs.” In: *Proc. of POPL’88*. ACM, 1988.

[2] G. Barthe, D. Demange, and D. Pichardie. “Formal Verification of an SSA-Based Middle-End for CompCert.” In: *ACM TOPLAS* 36.1 (2014).

[3] J. Blech et al. “Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL.” In: *COCV’05*. Elsevier, 2005.

[4] P. Briggs, K. D. Cooper, and L. T. Simpson. “Value Numbering.” In: *Software, Practice and Experience* 27.6 (1997).

[5] A. Chlipala. “A verified compiler for an impure functional language.” In: *POPL’10*. ACM, 2010.

[6] F. Chow et al. “A New Algorithm for Partial Redundancy Elimination Based on SSA Form.” In: *Proc. of PLDI ’97*. ACM, 1997.

[7] R. Cytron et al. “Efficiently computing static single assignment form and the control dependence graph.” In: *ACM TOPLAS* 13.4 (4 1991).

[8] R. Kumar et al. “CakeML: a verified implementation of ML.” In: *Proc. of POPL’14*. 2014.

[9] T. Lengauer and R. Tarjan. “A fast algorithm for finding dominators in a flowgraph.” In: *ACM TOPLAS* 1.1 (1 1979).

[10] X. Leroy. “A Formally Verified Compiler Back-end.” In: *JAR* 43.4 (2009).

[11] W. Mansky and E. Gunter. “A Framework for Formal Verification of Compiler Optimizations.” In: *ITP’10*. Springer-Verlag, 2010.

[12] A. Pnueli, M. Siegel, and E. Singerman. “Translation Validation.” In: *TACAS’98*. Springer-Verlag, 1998.

[13] S. Rideau and X. Leroy. “Validating Register Allocation and Spilling.” In: *Proc. of CC’10/ETAPS’10*. Springer-Verlag, 2010.

[14] M. Stepp, R. Tate, and S. Lerner. “Equality-Based Translation Validator for LLVM.” In: *CAV’11*. Springer-Verlag, 2011.

[15] J. Tristan, P. Govereau, and G. Morrisett. “Evaluating value-graph translation validation for LLVM.” In: *PLDI’11*. ACM, 2011.

[16] J. Tristan and X. Leroy. “A simple, verified validator for software pipelining.” In: *POPL’10*. ACM, 2010.

[17] J. Tristan and X. Leroy. “Verified validation of lazy code motion.” In: *PLDI’09*. ACM, 2009.

[18] M. N. Wegman and F. K. Zadeck. “Constant Propagation with Conditional Branches.” In: *ACM Trans. Program. Lang. Syst.* 13.2 (1991).

[19] X. Yang et al. “Finding and Understanding Bugs in C Compilers.” In: *Proc. of PLDI ’11*. ACM, 2011.

[20] J. Zhao et al. “Formal verification of SSA-based optimizations for LLVM.” In: *PLDI’13*. ACM, 2013.

[21] J. Zhao et al. “Formalizing the LLVM Intermediate Representation for Verified Program Transformation.” In: *POPL’12*. ACM, 2012.