A formally verified SSA-based compiler middle-end

Gilles Barthe, Delphine Demange, David Pichardie

A formally verified SSA-based middle-end compiler*

Static Single Assignment meets CompCert

Gilles Barthe[1], Delphine Demange[2], and David Pichardie[3]

[1] IMDEA Software Institute, Madrid, Spain
[2] ENS Cachan Bretagne / IRISA, Rennes, France
[3] INRIA, Centre Rennes - Bretagne Atlantique, Rennes, France

Abstract. CompCert is a formally verified compiler that generates compact and efficient PowerPC, ARM and x86 code for a large and realistic subset of the C language. However, CompCert foregoes using Static Single Assignment (SSA), an intermediate representation that allows for writing simpler and faster optimizers, and is used by many compilers. In fact, it has remained an open problem to verify formally a SSA-based middle-end compiler.
We report on a formally verified, SSA-based, middle-end for CompCert. Our middle-end performs conversion from CompCert intermediate form to SSA form, optimization of SSA programs, including Global Value Numbering, and transforming out of SSA to intermediate form. In addition to provide the first formally verified SSA-based middle-end, our work addresses two problems raised by Leroy [16]: giving a simple and intuitive formal semantics to SSA, and showing how to leverage the global properties given by SSA to reason locally about program optimizations.

1 Introduction

Static single assignment Static single assignment (SSA) form [9] is an intermediate representation where variables are statically assigned exactly once. Thanks to the considerable strength of this property, the SSA form simplifies the definition of many optimizations, and improves their efficiency, as well as the quality of their results. It is therefore not surprising that many modern compilers, including GCC [11] and LLVMC [17], rely heavily on SSA form, and that there is a vast body of work on SSA. However, the simplicity of SSA form is deceptive, and designing a correct SSA-based middle-end compiler is fraught with difficulties. In fact, it has been a significant challenge to design efficient, semantics-preserving, algorithms for converting programs into SSA form, or optimizing programs in SSA form, or even transforming programs out of SSA form.

Verified Compilers Compiler correctness aims at giving a rigorous proof that a compiler preserves the behavior of programs. After 40 years of a rich history, the field is entering into a new dimension, with the advent of realistic and mechanically verified compilers. This new generation of compilers was initiated with CompCert [16], a

* An extended version of the paper and the Coq formalization are available online at `http://www.irisa.fr/celtique/ext/compcertSSA`.

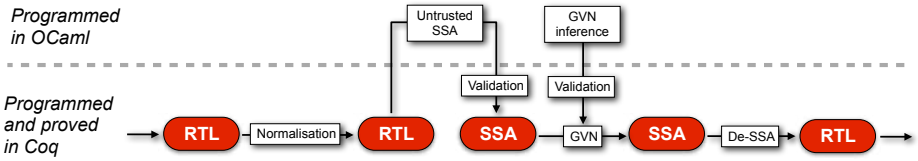


Fig. 1. The SSA Middle-end

compiler that is programmed and verified in the Coq proof assistant [8] and generates compact and efficient assembly code for a large fragment of the C language. Leroy's CompCert has been rightfully acclaimed as a *tour de force*, but CompCert foregoes relying on an SSA-based middle end. In [16], Leroy reports:

> Since the beginning of CompCert we have been considering using SSA-based intermediate languages, but were held off by two difficulties. First, the dynamic semantics for SSA is not obvious to formalize. Second, the SSA property is global to the code of a whole function and not straightforward to exploit locally within proofs.

add adds: "A typical SSA-based optimization that interests us is global value numbering". However verifying GVN is a significant challenge, and its formal verification has remained beyond current state-of-the-art in certified compilers.

Static Single Assignment meets verified compilers The thesis of our work is that a compiler can be realistic, verified and still rely on a SSA form. To support our thesis, we provide the first verified SSA-based middle-end. Rather than programming and proving a verified compiler from scratch, we have programmed and verified a SSA-based middle-end compiler that can be plugged into CompCert at the level of RTL. Figure 1 describes the overall architecture. Our middle-end performs four phases: (i) normalization of RTL program; (ii) transformation from RTL form into SSA form; (iii) optimization of programs in SSA form, including Global Value Numbering (GVN) [1]; (iv) transformation of programs from SSA form to RTL form; and relies on CompCert for the transformation from C to RTL programs prior to SSA conversion, and from RTL programs to assembly code after conversion out of SSA—our point is to program a realistic and verified SSA-based middle-end, rather than to demonstrate that SSA-based optimizations dramatically improve the efficiency of generated code.

We validate our compiler middle-end with a mix of techniques directly inherited from CompCert. We resort to translation validation [22, 21]—increasingly favored by CompCert [28, 29]—for converting programs into SSA form and for GVN. Specifically, we program in Coq verified checkers that can validate *a posteriori* results of untrusted computations of the SSA form and of a GVN numbering, and we implement in OCaml efficient algorithms for these computations; we rely on Cytron *et al* algorithm [9] for computing minimal SSA form, and on Alpern *et al* iteration strategy [1] for computing a numbering in GVN. In contrast, the normalization of the RTL program, and the conversion out of SSA are directly programmed and proved correct in Coq.

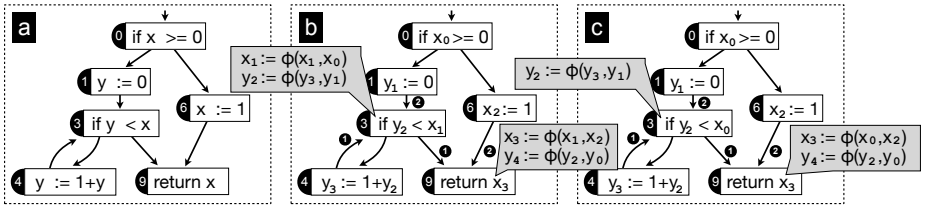


Fig. 2. Example programs. Programs b), c) are SSA forms of program a). Programs b) is in naive form and c) in minimal form.

In addition, our work addresses the two issues raised by Leroy [16]. First, we give a simple and intuitive operational semantics for SSA form; the semantics follows the informal description given in seminal papers [9], and does not require any artificial state instrumentation. Second, we define on SSA programs two global properties (called strictness and equational form) whose combination allows to conclude reasonably directly that the substitutions performed by GVN and other optimizations are sound.

Summarizing, our work provides the first verified SSA-based middle-end, the first formal proof of an SSA-based optimization, as well as an intuitive semantics for SSA. It thus serves as a good starting point for further studies of verified and realistic SSA-based compilers.

Contents The paper is organized as follows: Section 2 provides a brief primer on SSA and CompCert. Section 3 defines the SSA language used by our middle-end. Conversion to and out of SSA forms are presented in Section 4 and 5 respectively. Section 6 presents SSA-based optimizations. We conclude with experimental results in Section 7 and related work in Section 8.

Throughout the paper, we use Coq syntax for our definitions and results. Statements occasionally involve some notions that are not introduced formally. In such cases, names are generally chosen to be self-explanatory (for instance, `not_wrong_program`); in other cases, we forego giving precise definitions as they are not needed to understand the paper (for instance, the types `chunk` and `addressing` are unspecified in the definition of state). Our formalization makes an extensive use of inductive definitions, which are introduced in Coq using the keyword **`Inductive`**. Inductive definitions are used both for introducing new datatypes, e.g. the type of RTL instructions in Figure 4, and for introducing inductive relations, e.g. the operational semantics of RTL instructions in Figure 4. In the latter case the declarations are of the form `Inductive R : A →B →Prop := | Rule1: ∀ a b, ... →R a b | Rule2:....`

2 Background

Static Single Assignment form is an intermediate representation in which program variables are statically assigned exactly once, thus making explicit in the program syntax the link between the point where a variable is defined and where it is read.

Converting into SSA form is easy for straighline code: one simply tags each variable definition with an index, and each variable use with the index corresponding to the last definition of this variable. For example, $[x := 1; y := x + 1; x := y - 1; y := x]$ is transformed into $[x_0 := 1; y_0 := x_0 + 1; x_1 := y_0 - 1; y_1 := x_1]$. The transformation is semantics-preserving, in the sense that the final values of x and y in the first snippet coincide with the final values of x_1 and y_1 in the second snippet. On the other hand, one cannot transform arbitrary programs into semantically equivalent programs in SSA form solely by tagging variables: one must insert *ϕ-functions* to handle branching statements. Figure 2 shows a control-flow graph program a), and a program b) that corresponds to a SSA form of a). In program a), the value of variable x read at node 9 either comes from the definition of x at entry or at point 6. In program b), the two definitions of x at entry and at point 6 are renamed into the unique definition of x_0 and x_2 and merged together by the ϕ-function of x_3 at entry of node 9. The precise meaning of a ϕ-block depends on the numbering convention of the predecessor nodes of each junction point. In Figure 2 b) we make explicit this numbering by labelling the CFG edges. For example, node 3 is the first predecessor of the junction point 9 while node 6 is the second one. The semantics of ϕ-functions is given in the seminal paper by Cytron *et al* [9]: "If control reaches node j from its kth predecessor, then the run-time support remembers k while executing the ϕ-functions in j. The value of $\phi(x_1, x_2, \ldots)$ is just the value of the kth operand. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of control just before entering j. "

There may be several SSA forms for a single control-flow graph program; Figure 2 b) and c) gives alternative SSA forms for program a). As the number of ϕ-functions directly impacts the quality of the subsequent optimizations—as well as the size of the SSA form—it is important that SSA generators for real compilers produce a SSA form with a minimal number of ϕ-functions. Implementations of minimal SSA generally rely on the notion of *dominance frontier* to choose where to insert ϕ-functions. A node i in a CFG dominates another node j if every path from then entry of the CFG to j contains i. The dominance is said to be strict if additionally $i \neq j$. A tree can encode the dominance relation between the nodes of the CFG. For a node i of a CFG, the *dominance frontier* $DF(i)$ of a node i is defined as the set of nodes j such that i dominates at least one predecessor of j in the CFG but does not strictly dominates j itself. The notion is extended to a set of nodes S with $DF(S) = \bigcup_{i \in S} DF(i)$. The *iterated dominance frontier* $DF^+(S)$ of a set of nodes S is the limit $\lim_{i \to \infty} DF^i(S)$. Formally, a program is in *minimal-SSA* form when a ϕ-function of an instance x_i of an original variable x appears in a junction point j iff j belongs to the iterated dominance frontier of the set of definition points of x in the original program. For instance, program c) in Figure 2 is in minimal-SSA form. However, one can achieve more compact SSA forms by observing that, at any junction point, dead variables need not be defined by a ϕ-function. The intuition is captured by the notion of *pruned-SSA* form: a program is in *pruned-SSA* form if it is in minimal-SSA form and for each ϕ-function of an instance x_i of an original variable x at a junction point j, x is live at j in the original program.

SSA-based optimizations The SSA form simplifies the definition of many common optimizations; for instance, copy propagation algorithms can just walk through a SSA program, identify statements of the form $x := y$, and replace every use of x by y.

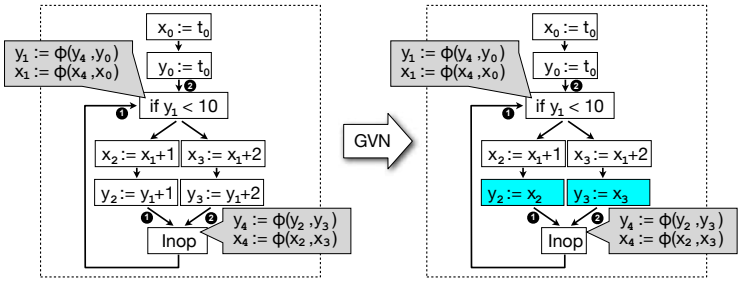


Fig. 3. Common sub-expression elimination using GVN

Furthermore, there are several optimizations that are naturally formulated in the context of SSA form. One typical SSA-based optimization is *Global Value Numbering* (GVN) [1], which assigns to variables an identifying number such that variables with the same number will hold equal values at execution time. The effectiveness of GVN lies in its ability to compute efficiently numberings that identify as many variables as possible. Advanced algorithms [1, 5] allow to compute efficiently such numberings. We briefly explain one such numbering in Section 6.

Figure 3 illustrates how GVN can be used to eliminate redundant computation. The left program is the original code; in this program, the number assigned to x_i and y_i is simply i. Hence, the evaluation of $y_1 + 1$ (resp. $y_1 + 2$) is a redundant computation when assigning y_2 (resp. y_3), and one can transform the program into the semantically equivalent one shown on the right of the figure. The strength of the analysis lies in its ability to reason about ϕ-functions, which allows it to infer the equality $x_2 = y_2$. This is only possible because numbering is global to the whole program; in fact, any block-local analysis would fail to discover the equality $x_2 = y_2$.

CompCert is a realistic formally verified compiler that generates PowerPC, ARM or x86 code from source programs written in a large subset of C. CompCert formalizes the operational semantics of dozen intermediate languages, and proves for each phase a semantics preservation theorem. Preservation theorems are expressed in terms of program behaviors, i.e. finite or infinite traces of external function calls (a.k.a. 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 got wrong, 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: ∀ (p: C.program) (tp: Asm.program),
  (not_wrong_program p ∧ compcert_compiler p = OK tp) →
  (∀ beh, exec_asm_program tp beh → exec_C_program p beh).
```

This paper focuses on the CompCert middle-end where most of the existing optimisations are performed (currently: constant propagation, removal of redundant cast, tail call detection, local value numbering and a register allocation phase that includes copy propagation). Those optimisations operate on a Register Transfer Language (RTL),

```
Inductive instr  :=
  | 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 (or: option reg).

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: state → trace → state → Prop :=
 | ex_Inop: ∀ s f sp pc rs m pc',
     fn_code f pc = Some(Inop pc') →
     step (State s f sp pc rs m) ϵ (State s f sp pc' rs m)
 | ex_Iop: ∀ 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 (State s f sp pc rs m) ϵ (State s f sp pc' (rs#res←v) m)
 | ex_Iload: ∀ s f sp pc rs m pc' chk addr args res a v,
     fn_code f pc = Some(Iload chk addr args res pc') →
     eval_addressing sp addr (rs##args) = Some a →
     Mem.loadv chk m a = Some v →
     step (State s f sp pc rs m) ϵ (State s f sp pc' (rs#res←v) m)
```

Fig. 4. Syntax and semantics of RTL (excerpt)

whose syntax and semantics is given in Figure 4. A RTL program is formalized as a set of global variables, a set of functions, and an entry point. Functions are modelled as records that include a function signature `fn_sig`, a CFG `fn_code` of instructions over pseudo-registers. The CFG is not a basic-block graph: it partially maps each program point to a single instruction, and we stick to this important design choice made in CompCert. As explained by Knoop [13], it allows for simpler implementations of code manipulations and simplifies correctness proofs of analyses and transformations, without impacting too much their efficiency.

The RTL instruction set includes arithmetic operations (`Iop`), memory loads (`Iload`) and stores (`Istore`), function calls (`Icall`), conditional (`Icond`) and unconditional jumps (`Inop`), and a return statement (`Ireturn`)—in this paper we do not discuss jumptables and other kinds of function calls: call to a function pointer stored in a register, tail calls, and built-in functions. All instructions take as last argument a node `pc` denoting the next instruction to be executed; additionally, all instructions but `Inop` take as arguments pseudo-registers of type `reg`, memory chunks, and addressing modes.

The operational behavior of RTL programs is given by a small-step operational semantics in which transitions are labelled with event traces. The type of states is defined

in Figure 4 as the tagged union of regular states, call states and return states. We focus on regular states, as in this paper we will only expose the intra-procedural part of the language. A regular semantic state (`State`) is a tuple that contains a call stack (representing the current pending function calls), the current function description and stack pointer (to the stack data block, a part of the global memory in which reside initial C local variables whose address has been taken), the current program point, the registers state (an assignment of values for the local variables) and the global memory. The semantics also manipulates a global environment that maps function names and global variables to memory addresses; the global environment is never modified during a program execution, and thus ommitted from our presentation.

The operational behavior of programs is modelled by the relation `step` between two semantic states (see Figure 4), and a trace of events; all instructions except function calls do not emit any event, hence the transitions that they induced are tagged by the empty event trace ϵ. We briefly comment on the rules: $(\texttt{Inop}\ \texttt{pc}')$ branches to the next program point $\texttt{pc}'$. $(\texttt{Iop}\ \texttt{op}\ \texttt{args}\ \texttt{res}\ \texttt{pc}')$ performs the arithmetic operation `op` over the values of registers `args` (written `rs##args`), stores the result in `res` (written $\texttt{rs\#res} \leftarrow \texttt{v}$), and branches to $\texttt{pc}'$. The instruction $(\texttt{Iload}\ \texttt{chk}\ \texttt{addr}\ \texttt{args}\ \texttt{res}\ \texttt{pc}')$ loads a `chk` memory quantity from the address determined by the addressing mode `addr` and the values of the `args` registers, stores the quantity just read into `res`, and branches to $\texttt{pc}'$.

3 The SSA language

We describe the syntax and operational semantics of the language SSA that provides the SSA form of RTL programs. We equip the notion of SSA program with a *well-formedness* predicate which captures some essential properties of SSA forms.

SSA programs Our definition of SSA program distinguishes between RTL-like and ϕ-functions; the distinction avoids the need for unwieldy mappings between program points when converting to SSA, and allows for a smooth integration in CompCert. Figure 5 introduces the syntax of SSA. SSA functions operate on indexed registers of type $\texttt{SSA.reg} = \texttt{RTL.reg} * \texttt{idx}$, and include an additional field `fn_phicode` mapping junction points to ϕ-blocks. The latter are modelled as lists of ϕ-functions of the form $(\texttt{Iphi}\ \texttt{args}\ \texttt{res})$, where `res` is an indexed register, and `args` a list of indexed registers.

Next, we define structural constraints that allow giving an intuitive semantics to SSA programs. First, we require that the domain of the function `fn_phicode` is the set of junction points. Second, we require that each ϕ-functions in a ϕ-block has the same number of arguments as the number of predecessors of that block. Third, we require that all predecessors of a junction point are $(\texttt{Inop}\ \texttt{pc})$ instructions. This is a mild constraint, that can be ensured systematically on RTL programs through normalization, and that will carry over to their SSA forms. Figure 6 shows the RTL program from Figure 2 after normalization.

Finally, we consider two properties upon which SSA-based optimizations crucially rely: unique definitions and strictness [6]. The unique definitions property states that

each register is uniquely defined, whereas the strictness property states that each variable use is dominated by the (unique) definition of that variable. While the two properties are closely related, none implies the other; the program $[y_0 := x_0; x_0 := 1]$ satisfies the unique definitions property but is not in strict form whereas the program $[x_0 := 1; x_0 := 2; y_0 := x_0]$ is strict but does not satisfy the unique definitions property. To formalize these properties, one first defines the type of paths in a CFG, and predicates `dom` and `sdom` for dominance and strict dominance. Then, one defines predicates `def, use : SSA.function → SSA.reg → node → Prop` such that proposition `def f x pc` (resp. `use f x pc`) holds iff the register `x` is defined (resp. used) at program point `pc` in the (regular or ϕ-) code of the function `f`. The definition of `use` is complex because variables may be used in ϕ-functions: the widely adopted convention is to view ϕ-functions as lazily evaluated, their ith argument thus being used at the ith predecessor of the instruction. For example, in the SSA program of Figure 6, variable $\mathtt{x}_2$ is defined at point 6 and used at point 8, the 2nd predecessor of the junction point 9 where $\mathtt{x}_2$ appears as 2nd argument of the ϕ-function. A use in the regular code is more straightforward: a variable is used by an instruction if it appears on its right-hand side. Using `def` and `use`, one can then state the unique definition and strictness properties, and well-formedness. Formally, we say that a SSA function is well-formed if it satisfies the following predicate:

```
Record wf_ssa_function (f:SSA.function) : Prop := {
  fn_ssa:          unique_def f;
  fn_strict:       ∀ x u d, use f x u → def f x d → dom f d u;
  fn_wf_block:     block_nb_args f;
  fn_block_at_jp: ∀ jp, join_point jp f ↔ fn_phicode f jp ≠ None;
  fn_normalized:  ∀ jp pc, join_point jp f → In jp (succs f pc) →
                           fn_code f pc = Some (Inop jp);}.
```

where predicates `unique_def` and `block_nb_args` respectively capture that a function satisfies the unique definitions property and the structural constraint about arguments.

In the sequel, we show that conversion to SSA yields well-formed programs. Besides, our SSA-based optimizations will assume that the input SSA programs are well-formed; in turn, we prove for each of them that output programs satisfy well-formedness.

Semantics The notion of SSA state is similar to the notion of RTL state, except that the type of registers and current function are modified into `SSA.reg` and `SSA.function` respectively. The small-step operational semantics is defined on SSA programs that satisfy the structural constraints introduced in the previous paragraph. Formally, we define `SSA.step` as a relation between pairs of (SSA) states and a trace of events. The definition follows the one of `RTL.step`, except for instructions of the form $(\mathtt{Inop\ pc'})$, where one distinguishes whether $\mathtt{pc'}$ is a junction point or not. In the latter case, the semantics coincide with the RTL semantics, i.e. the program point is updated in the semantic state. If on the contrary $\mathtt{pc'}$ is a junction point, then one executes the ϕ-block attached to $\mathtt{pc'}$ before the control flows to $\mathtt{pc'}$. Executing ϕ-blocks on the way to $\mathtt{pc'}$ avoids the need to instrument the semantics of SSA with the predecessor program point, and crisply captures the intuitive meaning given to ϕ-blocks by Cytron *et al* (see Section 2). Note in particular that the normalization procedure ensures that the predecessor of a junction point is an `Inop` instruction. This greatly simplifies the definition of the semantics, and subsequently the proofs about SSA programs.

```
Inductive instr := ...

Inductive phiinstr :=
  | Iphi (args: list SSA.reg)
         (res: SSA.reg).

Definition phiblock:= list phiinstr.
```

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

```
Inductive step: SSA.state → trace → SSA.state → Prop :=
  | ex_Inop_njp: ∀ s f sp pc rs m pc',
       fn_code f pc = Some(Inop pc') →
       ¬ join_point pc' f →
       step (State s f sp pc rs m) ϵ (State s f sp pc' rs m)

   | ex_Inop_jp: ∀ 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 (State s f sp pc rs m) ϵ (State s f sp pc' (phistore k rs phib) m)
   ...

Fixpoint phistore k rs phib : SSA.regset :=
  match phib with
    | nil => rs
    | (Iphi args res)::phib =>
      match nth_error args k with
        | None => rs
        | Some arg => (phistore k rs phib)#res ← (rs#arg)
      end
  end.
```

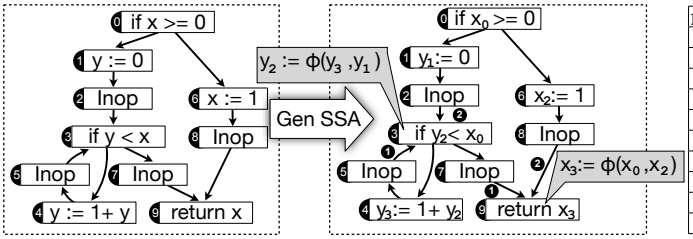
Fig. 5. Syntax and semantics of SSA (excerpt)

Following conventional practice, ϕ-blocks are given a parallel (big-step) semantics. This is formally embedded in the rule for `phistore` in Figure 5. When reaching a junction point $\mathtt{pc}'$ from its $\mathtt{k}$th predecessor, we update the register set `rs` for each register `res` assigned in the ϕ-block `phib` with the value of $\mathtt{rs}\#\mathtt{arg}$, where `arg` is the $\mathtt{k}$th operand in the ϕ-function of `res` (written $\mathtt{nth\_error\ args\ k} = \mathtt{Some\ arg}$). With the same notations, a well-formed SSA function satisfies the *parallel assignment* property:

```
∀ arg res, In (Iphi args res) phib →
   nth_error args k = Some arg → (phistore k rs phib) # res = rs # arg
```

4 Translation validation of SSA generation

Modern compilers typically follow the algorithm by Cytron *et al* [9] to generate a minimal SSA form of programs in almost linear time w.r.t. the size of the program. The algorithm proceeds in four steps: (i) it computes the dominator tree of the CFG using the Lengauer and Tarjan algorithm [15]; (ii) it builds the dominance frontier using a bottom-up traversal of the dominator tree; (iii) for each variable, it places ϕ-functions using iterated dominance frontier; (iv) at last, it uses a top-down traversal of the dominator tree to rename each def and use of RTL variables with correct indexes. Programming



pc	(Γ pc x)	(Γ pc y)	(live pc)
0	0	0	{x}
1	0	0	{x}
2	0	1	{x, y}
3	0	2	{x, y}
4	0	2	{x, y}
5	0	3	{x, y}
6	0	0	{ }
7	0	2	{x}
8	2	0	{x}
9	3	0	{x}

Fig. 6. A RTL program, its pruned SSA form and the corresponding type information

efficiently the algorithm in Coq and proving formally its correctness is a significant challenge—even verifying formally Step (i) requires to formalize a substantial amount of graph theory. Instead, we provide a new validation algorithm that checks in linear time that a SSA program is a correct SSA form of an input RTL program. The algorithm is complete w.r.t. minimal SSA form, and can be enhanced by a liveness analysis to handle pruned and semi-pruned SSA forms. In order to be used in a certified compiler chain, we also show that our validator guarantees preservation of behaviors.

Translation validation of SSA conversion is performed in two passes. The first pass performs a structural verification on programs: given a RTL function `f` and a SSA function `tf`, it verifies that `tf` satisfies all clauses of well-formedness except strictness, and that the code of `f` can be recovered from its SSA form `tf` simply by erasing ϕ-blocks and variable indices—the latter property is captured formally by the proposition `structural_spec f tf`. The second pass relies on a type system to ensure strictness and semantics-preservation. Overall the pseudo-code of the validator is:

```
let SSA_validator (f: RTL.function) (tf: SSA.function): bool :=
  if    (check_blocks_are_wf tf)           (* ensures block_are_wf tf *)
     && (check_blocks_are_at_jp tf)        (* ensures block_at_jp tf *)
     && (check_normalized tf)              (* ensures normalization *)
     && (check_unique_def tf)              (* ensures unique_def tf *)
     && (check_structural_spec f tf)       (* ensures structural_spec f tf *)
  then (is_well_typed f tf) else false
```

where `is_well_typed f tf` is the predicate stating that the function is well-typed w.r.t. our type system for SSA form.

Type system The basic idea of our type system is to track for each variable its *last* definition; this is achieved by assigning to all program points a local typing, i.e., an element of `ltype` = `RTL.reg` $\rightarrow$ `idx`; we let γ range over local typings. Then, the global typing of an SSA function `tf` is an element of `gtype` $=$ `node` $\rightarrow$ `RTL.reg` $\rightarrow$ `idx`; we let Γ range over global typings.

The type system is structured in three layers. The lowest layer checks that RTL-like instructions make a correct use of variables. The middle layer checks that CFG edges are well-typed. Finally, the third layer of the type system defines the notion of well-typed function. Throughout this section, we use Figure 6 as a running example (an RTL program, its pruned SSA form and the corresponding type mapping).

```
Definition use_ok (uses:list SSA.reg)(γ:ltype):= ∀ r i, In (r,i) uses→ γ r=i.

Inductive wt_instr: ltype → SSA.instr → ltype → Prop :=
| wt_Inop: ∀ γ s, {γ} Inop s {γ}

| wt_Istore: ∀ γ chk addr args s src,
  use_ok (src::args) γ → {γ} Istore chk addr args src s {γ}

| wt_Icond: ∀ γ cond args s1 s2,
  use_ok args γ → {γ} Icond cond args s1 s2 {γ}

| wt_Ireturn_some: ∀ γ r, use_ok [r] γ → {γ} Ireturn (Some r) {γ}

| wt_Ireturn_none: ∀ γ, {γ} Ireturn None {γ}

| wt_Iop: ∀ γ op args s r i,
  use_ok args γ  → i ≠ dft → {γ} Iop op args (r,i) s {γ[r ← i]}

| wt_Iload: ∀ γ chk addr args s r i,
  use_ok args γ → {γ} Iload chk addr args (r,i) s {γ[r ← i]}

| wt_Icall: ∀ γ sig args s id r i,
  use_ok args γ → i ≠ dft → {γ} Icall sig id args (r,i) s {γ[r ← i]}
```

Fig. 7. Typing rules for instructions

Liveness As explained in Section 2, liveness information can be used to minimize the number of ϕ-functions in a SSA program; specifically, ϕ-blocks only need to assign live variables (in Figure 6, the variable y is live at point 3, and x is live at point 9). Hence, our type system is parametrized by a function `live` that models a liveness analysis. Formally, we require that the `live` function satisfies two properties (for a function `f`, their conjunction is denoted by (`wf_live f live`)): (i) if a variable is used at a program point, then it should be live at this point and (ii) a variable that is live at a given program point is, at the predecessor point, either live or assigned.

Our type system is able to handle different SSA forms through appropriate instantiations of `live`. Our formalization provides support for minimal SSA and pruned SSA forms, respectively by defining `live` respectively as the trivial over-approximation (for each point, it is the set of all the RTL variables), and the result of a standard liveness analysis. One could also support for semi-pruned forms, by instantiating `live` as the result of the block-local liveness analysis of [4].

The type system for instructions checks that RTL-like instructions make of correct use of variables, and that they do not redefine parameters; its formal definition is given in Figure 7. Judgments are of the form $\{\gamma\}$ `ins` $\{\gamma'\}$; intuitively, the judgment is valid if each variable `x` is used in `ins` with the index (γ `x`), and γ' maps each variable to its last definition after execution of `ins`. The typing rules are formalized as an inductive relation `wt_instr`; we briefly comment on some rules. Several rules correspond to instructions that do not define variables, so the input and output local typings are equal. For such rules, one simply checks that the instruction makes a correct use of the variables. The typing rule for (`Inop pc`) states that for every local typing γ, (`Inop pc`) makes a correct use of variables. The typing rule for `Icond` checks that the variables used in the

```
Inductive wt_edge (f:SSA.function)(Γ:gtype)(live:Regset.t):node→ node → Prop:=
| wt_edge_not_jp: ∀ i j ins
  (NOTJP : fn_code f i = Some ins ∧ fn_phicode f j = None)
  (WTI :   {Γ i} ins {Γ j}),
  (wt_edge f Γ live i j)

| wt_edge_jp: ∀ i j ins block
  (JP:  fn_code f i = Some ins ∧ fn_phicode f j = Some block)
  (USES:∀ args r i, In (Iphi args (r,i)) block → phiuse_ok r args (preds f j) Γ)
  (ASSIG: ∀ r i, assigned (r,i) block → r ∈ live ∧ (Γ j r) = i ∧ i ≠ dft)
  (NASSIG: ∀ r, (∀ i, ¬ (assigned (r,i) block)) → (Γ i r = Γ j r) ∨ r ∉ live),
  (wt_edge f Γ live i j).

Definition wt_function (f:SSA.function)(Γ:gtype)(live:node→ Regset.t):Prop:=
(∀ i j, is_edge f i j → wt_edge f Γ (live j) i j)
∧ (∀ i or, fn_code f i = Some (Ireturn or) → {Γ i} Ireturn or {Γ i})
∧ (∀ p, In p (fn_params f) → ∃ r, p = (r, Γ (fn_entrypoint f) r)).
```

Fig. 8. Typing rules for edges and functions

guard are consistent with the input local typing (in Figure 6, the uses of $\mathtt{x}_0$ and $\mathtt{y}_2$ at point 3 are consistent with the intput local typing: $(\Gamma\ 3\ \mathtt{x}) = 0$ and $(\Gamma\ 3\ \mathtt{y}) = 2$). In the case of the instruction `Iop`, which defines the variable $(\mathtt{r}, \mathtt{i})$, the output local typing is $\gamma[\mathtt{r} \leftarrow \mathtt{i}]$, i.e. the input local typing updated for the initial variable `r`. From this program point onwards, the new version for `r` is the one indexed with `i`, and this is the one that should be used later on, until another version for `r` is defined (in Figure 6, the definition of $\mathtt{x}_2$ at point 6 makes the local typing change for variable `x` between points 6 and 8). Note that each time a variable is defined, we demand its index to be different from the index `dft` assigned to parameters at the onset of the program (in the example, the default index is 0). This prevents that a parameter is redefined during execution, which would violate the unique definition property.

Typing rules for edges and functions The typing rules for edges ensure that ϕ-blocks make a correct use of definitions w.r.t. a global typing Γ. There are two rules—modelled by the clauses of the inductive relation `wt_edge` in Figure 8. The first rule considers the case where the edge does not end in a junction point; in this case, typing the edge is equivalent to typing the corresponding instruction. The second rule considers the case where the edge ends in a junction point; in this case, the typing rule checks the ϕ-block attached to it—structural constraints impose that the instruction is an `Inop`, so we do not need to type-check the instruction. Hypothesis `USES` ensures that the ϕ-arguments passed to ϕ-functions are consistent w.r.t. all incoming local typings: its kth argument should be the version of the initial variable brought by the kth predecessor of the join point (we omit the formal definition of `phiuse_ok`). Hypothesis `ASSIG` ensures that the ϕ-block is compatible with the output local typing; Hypothesis `NASSIG` ensures that variables not assigned in the ϕ-block are either dead, or the incoming indices are the same. In Figure 6, the ϕ-function for `x` makes correct uses of it because its first argument $\mathtt{x}_0$ matches $(\Gamma\ 7\ \mathtt{x}) = 0$ and $\mathtt{x}_2$ matches $(\Gamma\ 8\ \mathtt{x}) = 2$. The local typing at point 9 takes into account the definition of $\mathtt{x}_3$ inside the block by setting $(\Gamma\ 9\ \mathtt{x})$ to 3. More-

over, no ϕ-function is required for y at point 9 since y $\notin$ (live 9), and no ϕ-function is required for x at point 3, since $(\Gamma\ 2\ \mathtt{x}) = (\Gamma\ 5\ \mathtt{x})$.

Finally, a function is well-typed w.r.t. global typing Γ if the local typing induced by Γ at the entry point fn_entrypoint is consistent with the parameters, and all edges and return instructions are well-typed.[4]

Implementation For the clarity of exposition, we have described a non-executable type checker which assumes that structural constraints are satisfied. The Coq implementation of the type system is in fact a bit more complex. In particular, it performs type inference rather than type checking; for efficiency reasons, the algorithm performs a single, linear scan of the program, and checks the list of arguments of ϕ-functions only once per junction point, rather than once per incoming edge for a given join point. On the benchmarks given in Section 7, our efficient implementation is ten times faster than a type checker derived from the non-executable type system.

Properties of the validator

Strictness All SSA programs accepted by the type system are strict. It follows that only well-formed SSA functions will be accepted by the validator.

```
Theorem wt_strict: ∀ f tf Γ live,
wf_live f live → wt_function tf Γ live →
∀ (xi : SSA.reg) (u d : node), use tf xi u → def tf xi d → dom tf d u.
```

The proof of wt_strict relies on two auxiliary lemmas about local typings in well-typed functions. The first lemma states that if a variable $(\mathtt{x}, \mathtt{i})$ is used at point pc, then it must be that $(\Gamma\ \mathtt{pc}\ \mathtt{x} = \mathtt{i})$. The second lemma states that whenever $(\Gamma\ \mathtt{pc}\ \mathtt{x} = \mathtt{i})$, the definition point of variable $(\mathtt{x}, \mathtt{i})$ dominates pc.

Soundness The validator is sound in the sense that if it accepts a RTL program f and an SSA form tf, then all behaviors of tf are also behaviors of f. Since CompCert already shows the very general result that the existence of a lock-step forward simulation implies preservation of behaviors, it is sufficient to exhibit such a simulation:

```
Theorem validator_correct : ∀ (prog:RTL.program) (tprog:SSA.program),
SSA_validator prog tprog = true →
 ∀ s1 t s2, RTL.step s1 t s2 →
   ∀ s1', s1 ≃ s1' → ∃ s2', SSA.step s1' t s2' ∧ s2 ≃ s2'.
```

where the binary relation $\simeq$ between semantic states of RTL and SSA carries the invariants needed for proving behavior preservation. For instance, two regular states are related by $\simeq$ if their memory states, stack pointers, and program counters are equal, their function descriptions are suitably related, e.g. by structural_spec, and their register states rs and rs$'$ agree, i.e. satisfy (agree (Γ pc) rs rs$'$ (live pc)), where

```
Definition agree (γ:ltype) (rs:RTL.regset) (rs':SSA.regset) (live:Regset.t):=
  ∀ r, r ∈ live → rs#r  = rs'#(r, γ r).
```

[4] It is not sufficient to require that all edges are well-typed since return instructions do not correspond to any edge.

Agreement is at the heart of the proof. It captures the semantics of local typings by making explicit how, at a given program point, variables of `f` should be interpreted in terms of the new variables in `tf`. The definition of $\simeq$ is completed by defining equivalence of stackframes; this relation basically lifts to the callstack all the invariants enforced by $\simeq$.

Completeness An essential property of our type system is that it accepts all the SSA programs that are output by the algorithm by Cytron *et al* [9]. The idea of the proof is as follows. First, one defines for each RTL normalized program `f` a global typing Γ. Second, we show that all instructions of the program `tf` output by our implementation are typable. Then, we show that all edges are typable if we omit the constraints about correct use; the proof relies crucially on the fixpoint characterization of the iterated dominance frontier, as given in work of Cytron *et al* [9]. Finally, one shows that all constraints about correct use are satisfied, and hence the program `tf` is typable with Γ. The appendix provides a more detailed sketch of the completeness proof.

5 Conversion out of SSA

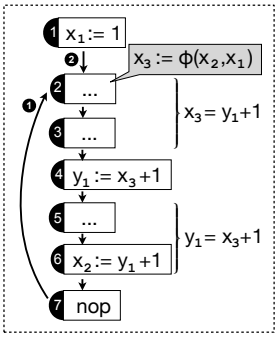
We have programed and verified a simple de-SSA algorithm that transforms SSA programs into RTL programs—so that they can be further processed by CompCert back end. The idea is to substitute each ϕ-function with one variable copy at each predecessor of junction points. Thanks to the single-instruction graph of RTL, replacing ϕ-functions with copies ensures soundness of the transformation, since critical edges are automatically splitted by code insertion—a critical edge is an edge whose entry has several successors and exit has several predecessors (see [4]). Pleasingly, the representation of programs inherited from CompCert deflates the penalty cost of splitting edges—on the contrary, algorithms that operate on *basic-block graphs* carefully avoid edge splitting, at the cost of making de-SSA algorithms significantly more complex. On the negative side, our current implementation of de-SSA fails on SSA programs with non-parallel ϕ-blocks, i.e. in which some variable is both used and defined. Future work includes making de-SSA total, using the formalization of the parallel moves algorithm [23]—which transforms a set of parallel moves into an equivalent sequence of elementary moves (using additional temporaries), and that is already used in CompCert.

Concerning the correctness of the transformation, we proceed by giving a forward simulation between the SSA program and the RTL program after de-SSA. The simulation requires the RTL program to perform several steps to simulate a (big-step) execution of a ϕ-block by the initial SSA program.

6 Translation validation of SSA-based optimizations

In this section, we introduce the *equation lemma* that support the view of programs in SSA form as *systems of equations*. We then illustrate how to reason about a simple SSA-based optimization, namely copy propagation. Finally, we formalize and prove correct a GVN optimization.

Equation lemma The SSA representation provides an intuitive reading of programs: one can view the unique definition of a variable as an equation, and by extension one can view SSA programs as systems of equations. For instance, the definitions of $\mathtt{x}_3$ and $\mathtt{y}_1$ respectively induce the two equations $x_3 = y_1 + 1$ and $y_1 = x_3 + 1$. There is however a pitfall: the two equations entail $x_3 = x_3 + 2$, and hence are inconsistent. In fact, equations are only valid at program points which are dominated by the definition that induce them, as captured formally by the *equation-lemma* of SSA:




```
Lemma equation_lemma : ∀ prog d op args x succ f m rs sp pc s,
    wf_ssa_program prog →
      reachable prog (State s f sp pc rs m) →
      fn_code f d  = Some (Iop op args x succ) →
      sdom f d pc →
      eval_operation sp op (rs##args) m = Some (rs#x).
```

where `reachable` is a predicate that defines reachable states. In practice, it is often convenient to rely on a corollary that proves the validity of the defining equation of `x` at program points where `x` is used – thus avoiding reasoning on the dominance relation. The formal statement of the corollary is obtained by replacing the hypothesis `sdom f d pc` by the hypothesis `use f x pc`; the proof of the corollary makes an essential use of the strictness property of well-formed SSA programs.

We conclude this paragraph with a succinct account of applying the corollary to prove the soundness of copy propagation (CP)—recall that CP will search for copies $\mathtt{x} := \mathtt{y}$ and replace every use of `x` by a use of `y`. Suppose `pc` is a program point where such a replacement has been done. Every time `pc` is reached during the program execution, we are able to derive, using the corollary, that $\mathtt{rs}\#\mathtt{y} = \mathtt{rs}\#\mathtt{x}$, where `rs` is the current register state because (i) `y` is the right hand side of the definition of `x` and (ii) `pc` was a use point of `x` in the initial program. On non-SSA forms, the reasoning is more involved since one has to prove that the reaching definition for `x` is unique at `pc`, and that no redefinition of `y` can occur in between.

Global Value Numbering Our implementation of GVN is made of two components. The first one is an efficient but untrusted analysis, written in OCaml, for computing numberings of SSA programs. From an abstract interpretation point of view, the analysis—which follows [1]—computes a fixpoint in the abstract domain of congruence partitions, where partitions are modelled as mappings $\mathcal{N} : \mathtt{reg} \to \mathtt{reg}$, and ordered w.r.t. reverse inclusion of equivalence kernels—recall that the equivalence kernel of $\mathcal{N}$ is the relation $\sim$ defined by $\mathtt{x} \sim \mathtt{y}$ if and only if $\mathcal{N}\ x = \mathcal{N}\ y$. Viewing the result of the analysis as a post-fixpoint is the key to our second component, a validator that checks whether a numbering $\mathcal{N}$ is indeed a post-fixpoint of the analysis on a program `p`, and if so returns an optimized SSA program `tp`. The validator is programmed in Coq, and is accompanied with a proof that optimized programs preserve the behaviors of the original programs.

The notion of valid numbering is formally defined in Figure 9. First, we define for each numbering $\mathcal{N}$ the relation $\equiv^{\mathcal{N}}$ as the smallest reflexive relation identifying:

```
Inductive ≡^𝒩 : reg → reg → Prop :=
| GVN_refl : ∀ x, ≡^𝒩 x x
| GVN_Iop : ∀ x y pc1 pc2 op args1 args2 pc1' pc2'
    fn_code f pc1 = Some(Iop op args1 x pc1') → same_number 𝒩 args1 args2 →
    fn_code f pc2 = Some(Iop op args2 y pc2') → ≡^𝒩 x y
| GVN_Phi : ∀ x y pc args_x args_y
    fn_phicode f pc = Some phib → same_number 𝒩 args_x args_y →
    (Iphi args_x x) ∈ phib → (Iphi args_y y) ∈ phib → ≡^𝒩 x y.

Definition GVN_spec (𝒩:reg → reg) : Prop :=
(∀ x y, 𝒩 x = 𝒩 y → param f x → param f y→ x=y)∧(∀ x y, 𝒩 x = 𝒩 y → ≡^𝒩 x y).
```

Fig. 9. Valid numbering

(i) registers whose assignments share the same operator and corresponding arguments are equivalent w.r.t. $\mathcal{N}$ (predicate `same_number`); (ii) registers that are defined in the same ϕ-block with equivalent arguments.Then, a numbering $\mathcal{N}$ is said to be valid if its equivalence kernel does not contain a pair of distinct function parameters and moreover it is included in $\equiv^{\mathcal{N}}$. The latter ensures the intended post-fixpoint property.

The crux of the correctness proof of the GVN validator is the correctness lemma for a valid numbering: if $\mathcal{N}$ is a valid numbering for `f`, and `rs` is a register state that can be reached at program point `pc`, and `x` and `y` are two registers whose definition strictly dominate `pc`, then $\mathcal{N}\ \mathtt{x} = \mathcal{N}\ \mathtt{y}$ entails that `rs` holds equal values for `x` and `y`:

```
Lemma valid_numbering_correct : ∀ prog s sp pc rs m,
    wf_ssa_program prog → GVN_spec 𝒩 →
      reachable prog (State s f sp pc rs m) → gamma 𝒩 pc rs.
```

where `gamma` is defined by

```
Definition gamma (𝒩:reg → reg) (pc:node) (rs: regset) : Prop :=
  ∀ x y: reg, def_sdom f x pc → def_sdom f y pc → 𝒩 x = 𝒩 y → rs#x = rs#y.
```

and `def_sdom f x pc` states that the definition of `x` in `f` strictly dominates `pc`. For an illustration of the property, consider Figure 3; here registers $\mathtt{x}_2$ and $\mathtt{y}_2$ share the same numbering; they are indeed equal just after the assignment of $\mathtt{y}_2$—but not before.

Next, we describe the Coq implementation for optimizing SSA programs. The implementation takes as input a numbering $\mathcal{N}$, and a partial mapping `crep` that takes as input a register `x` and program point `pc` and returns, if it exists, a register `y` such that `x` and `y` are related by the equivalence kernel of $\mathcal{N}$, and the definition of `y` strictly dominates `pc`. For efficiency reasons, we do not check the correctness of `crep` a priori, but lazily during the construction of the optimized program. The optimizer proceeds as follows: first, it checks whether $\mathcal{N}$ satisfies the predicate `GVN_spec`. Then, for each assignment (`Iop op args x pc`) of the original SSA program, the optimizer searches whether `crep` provides a canonical representative `y` for `x` at program point `pc`. If so, it checks whether the definition of `y` strictly dominates `pc`; this is achieved by means of a dominance analysis, computed directly inside Coq with a standard dataflow framework *a la* Kildall. Provided `y` is validated, we can safely replace the previous instruction by a move from `y` to `x`.

We conclude by commenting briefly on the soundness proof of the transformation. The proof follows a standard forward simulation proof where the correctness of the numbering is proved at the same time as the simulation itself. Noticeably, the normal form of CFG turned out to be extremely valuable for this proof. Indeed, consider a step from a program point pc to program point pc′: we have to prove that (gamma $\mathcal{N}$ pc′ rs) holds, asumming (gamma $\mathcal{N}$ pc rs). We reason by case analysis: if the instruction at pc is not an Inop instruction, we know by normalization that pc′ is not a junction point. In this case, (def_sdom f x pc′) is equivalent to (def_sdom f x pc) $\vee$ (def f x pc) which is particularly useful to exploit the hypothesis that (gamma $\mathcal{N}$ pc rs) holds.

7 Implementation and experimental results

We have plugged in Compcert 1.8.2 our SSA middle-end composed of (i) a Coq normalization (ii) an Ocaml SSA generator and its Coq validator; (iii) an Ocaml GVN inference tool and its Coq validator; (iv) a Coq de-SSA transformation. Our formal development adds 15.000 lines of Coq code and 1.000 lines of Ocaml to the 80.000 lines of Coq and 1.000 lines of Ocaml provided in CompCert 1.8.2. It does not add any axioms to CompCert. We use the Coq extraction mechanism to obtain a SSA-based certified compiler, that we evaluate experimentally using the CompCert benchmarks. These include around 75.000 lines of C code, and fall into three categories of programs, ranging from 20 to 5.000 lines of C code: a set of small computation kernels, a raytracer, and the theorem prover Spass[5]. Below we briefly comment on three key points: efficiency of the SSA validator; effectiveness of the GVN optimizer; efficiency of generated code.

Efficiency of SSA validator In order to be practical, validators must be more efficient than state-of-the-art implementations of the transformations that they validate. At first sight, this criterion may seem too demanding for SSA, since generation into SSA form is performed in almost linear time. However, experimental results are surprisingly good: overall converting a program into SSA form takes approximately twice longer than type-checking the output program. In more detail, the times for SSA generation—specialized to pruned SSA—distribute as follows: (i) 9% for normalization of RTL; (ii) 37% for liveness analysis of RTL (the liveness analysis is provided in the CompCert distribution); (iii) 35% for conversion to SSA using the untrusted OCaml implementation (based on state-of-the-art algorithms); (iv) 19% for validation using the verified validator. This distribution appears to be uniform on all benchmarks except on the biggest functions where the liveness analysis exhibits a non-linear complexity.

Effectiveness of GVN optimizer We measure the effectiveness of our GVN analyzer by performing a GVN-based CSE right after (Local Value Numbering) LVN-based CSE implemented by CompCert, and counting how many additional Iop instructions can be optimized by this additional CSE phase. In order to keep a fair comparison, we allow CompCert CSE to optimize around function calls—this is disabled in CompCert to keep the register pressure low. The overall improvement is significative: in comparison with CompCert's CSE, our global CSE optimizes an additional 25% of Iop instructions.

[5] Spass is the largest one with 69.073 LoC, and we only use it to evaluate the compilation time.

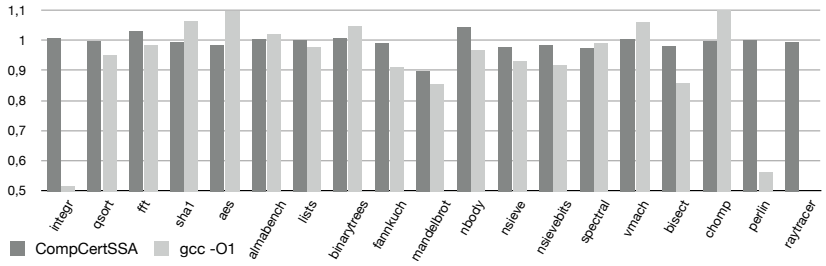


Fig. 10. Execution times of generated code

Efficiency of generated code In order to assess the efficiency of generated code, we have compiled the set of benchmarks with three compilers: CompCert, our version of CompCert extended with a SSA middle-end (CompCert-SSA), and `gcc -O1`. The chart in Figure 10 gives the execution times *relative* to Compcert (shorter bars mean faster) on PowerPC. The test suite is too small to draw definite conclusions, but the results are encouraging. Our version of CompCert performs slightly better than CompCert. We expect that performance improves significantly by enhancing our middle-end with additional optimizations, and by relying on an SSA-based register allocator.

8 Related Work

We focus on most closely related work and refer to [16] for an overview of mechanized compiler correctness, and to [24] for an annotated bibliography on SSA.

Machine-checked formalizations Blech *et al* [3] use the Isabelle/HOL proof assistant to verify the generation of machine code from a representation of SSA programs that relies on term graphs. While graph-based representations may be useful for the untrusted parts of our compiler, they increase the complexity of giving a formal semantics to SSA, and make it a greater challenge to verify SSA-based optimizations. They do not provide an algorithm to convert into SSA form, and leave as future work proving the correctness of SSA-based optimizations.

Mansky and Gunter [18] use Isabelle/HOL to formalize and verify the conversion of CFG programs into SSA form. However, their transformation is not designed to yield compact SSA forms – it inserts ϕ-functions at every junction point and for every variable. Moreover, it is not clear whether their semantics of SSA can be used to reason about SSA-based optimizations. Specifically, their semantics is based on an instrumentation of the state which stores for each variable x the index i corresponding to the last assigned instance x_i of x. As a result, their semantics is limited to SSA programs where ϕ-functions are of the form $x_0 := \phi(x_1, \ldots, x_n)$, i.e. deal with only one initial program variable. On the contrary, SSA-based optimizations typically return programs in which ϕ-functions takes as arguments instances of different variables.

Zhao *et al* [30] formalize the LLVM intermediate representation in the Coq proof assistant. They define and relate several formal semantics of LLVM, including a static and dynamic semantics. Further, they show how simple code motions can be validated using a simulation relation based on symbolic evaluation, and plan to extend the method to other transformations such as dead code elimination or constant propagation.

Finally, there are several machine-checked accounts of Continuation Passing Style (CPS) translations, e.g. [10, 7], which are closely related to conversion to SSA form [2].

Translation validation and type systems Menon *et al* [20] propose a type system that can be used to verify memory safety of programs in SSA form. However, their system does not enforce the SSA property, which must be verified by an external tool.

Matsuno and Ohori [19] define a type system equivalent to the SSA form: every typable program is given a type annotation that make explicit def-use relations. Their type system is similar to ours except they type check one program w.r.t. def-use annotations while we type check in parallel a RTL and a SSA program. Further, they show that common optimizations such as dead code elimination and common sub-expression elimination are type-preserving. On the other hand, they do not prove the semantics preservation of the optimizations.

Stepp *et al* [25] report on a translation validator for LLVM. Their validator uses Equality Saturation [26], a method which views optimizations as equality analyses. However, their tool does not validate GVN. Tristan *et al* [27] report on an independent effort to build a translation validator for LLVM's inter-procedural optimizations. This tool supports GVN, but is currently not certified.

9 Conclusion and Future Work

Our work shows that verified and realistic compilers can rely on a SSA-based middle-end that implements state-of-the-art algorithms, and opens the way for a new generation of verified compilers based on SSA.

A priority for further work is to achieve a tighter integration of our compiler middle-end into CompCert. There are three immediate objectives: (i) enhancing our SSA middle-end to handle memory aliases as done by CompCert RTL-based middle-end, (ii) implementing a SSA-based register allocator [12], and (iii) verifying more SSA-based optimizations, including lazy code motion [14]—we expect that our implementation of GVN will provide significant leverage there. Eventually, it should be possible to shift all CompCert optimisations into the SSA middle-end. In the longer term, it would be appealing to apply our methods to LLVM, building on [27, 25, 30].

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88*. ACM, 1988.
2. A.W. Appel. Ssa is functional programming. *SIGPLAN Notices*, 33:17–20, 1998.

3. J.O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing code generation from ssa form: A comparison between two formal correctness proofs in isabelle/hol. In *COCV'05*, ENTCS. Elsevier, 2005.
4. P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Soft.–Pract. and Exp.*, 1998.
5. P. Briggs, K.D. Cooper, and L.T. Simpson. Value numbering. *Soft.–Pract. and Exp.*, 1997.
6. Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves. Fast copy coalescing and live-range identification. In *PLDI '02*. ACM, 2002.
7. A. Chlipala. A verified compiler for an impure functional language. In *POPL'10*, pages 93–106. ACM, 2010.
8. The Coq proof assistant. `http://coq.inria.fr`.
9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 1991.
10. Z. Dargaye and X. Leroy. Mechanized verification of cps transformations. In *LPAR'07*, LNCS, pages 211–225. Springer-Verlag, 2007.
11. GCC, the GNU compiler collection. `http://gcc.gnu.org/`.
12. S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA form. In *CC*, LNCS. Springer-Verlag, 2006.
13. J. Knoop, D. Koschtzkil, and B. Steffen. Basic-block graphs: Living dinosaurs? In *CC*, LNCS. Springer-Verlag, 1998.
14. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *PLDI'92*, pages 224–234, 1992.
15. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1979.
16. X. Leroy. A formally verified compiler back-end. *J. of Autom. Reason.*, 43(4), 2009.
17. The LLVM compiler infrastructure. `http://llvm.org/`.
18. W. Mansky and E. Gunter. A framework for formal verification of compiler optimizations. In *ITP'10*. Springer-Verlag, 2010.
19. Y. Matsuno and A. Ohori. A type system equivalent to static single assignment. In *PPDP'06*. ACM, 2006.
20. V. Menon, N. Glew, B.R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *POPL'06*. ACM, 2006.
21. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI'00*, pages 83–94. ACM, 2000.
22. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, LNCS, pages 151–166. Springer-Verlag, 1998.
23. L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with coq: Formal verification of a compilation algorithm for parallel moves. *J. of Autom. Reason.*, 2008.
24. Ssa bibliography. `http://www.cs.man.ac.uk/~jsinger/ssa.html`.
25. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV'11*, LNCS, pages 737–742. Springer-Verlag, 2011.
26. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL'09*. ACM, 2009.
27. J. B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI'11*. ACM, 2011.
28. J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *PLDI'09*, pages 316–326. ACM, 2009.
29. J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL'10*, pages 83–92. ACM, 2010.
30. J. Zhao, S. Zdancewic, S. Nagarakatte, and M. Martin. Formalizing the LLVM intermediate representation for verified program transformation. In *POPL'12*. ACM, 2012.

Sketch Proof of Completeness of the SSA validator

We first review the well-known characterization of the iterated dominance frontier as a fixpoint of the operator J. Given a set S of nodes, $J(S)$ is defined to be the set of all nodes j such that there are two non-empty CFG paths that start at two distinct nodes in S and converge at j. For any set of nodes S, the iterated dominance frontier of S, $DF^+(S)$ satisfies $DF^+(S) = J(S \cup DF^+(S))$ ([9] page 467). For a variable x, we note def_x the set of program points where x is defined in `f`. We note D_x the set $\mathrm{def}_x \cup DF^+(\mathrm{def}_x)$. By definition of minimal-SSA form, S_x is the set of program points where any instance of x is defined in `tf`.

Let `f` be a normalized RTL program and let `tf` be the SSA program generated from `f` by a dominance-based algorithm. We first build using a depth-first-search (DFS) traversal of the CFG a global typing Γ. Then, we show such that `tf` is typable with Γ in our type system.

In order to explain the construction of Γ, we focus on a particular RTL variable x. Each time we reach a program point j, its father j' in the DFS tree has already been treated and $(\Gamma\ j'\ x)$ is thus already defined. To define $(\Gamma\ j\ x)$, we distinguish two cases. If j is not a junction point, then j' is the unique predecessor of j and we assign $(\Gamma\ j'\ x)$ to $(\Gamma\ j\ x)$ if x is not assigned at j' in `f` or we assign k to $(\Gamma\ j\ x)$ if a x_k is assigned at j' in `tf`. If j is a junction point, we assign $(\Gamma\ j'\ x)$ to $(\Gamma\ j\ x)$ if x is not assigned in the ϕ-block at j or we assign k to $(\Gamma\ j\ x)$ if a x_k is assigned by a ϕ-function at j. By construction, such a global typing Γ satisfies the following property: if $(\Gamma\ i\ x) = k$, then there exists a definition of x_k in D_x that reaches the program point i without meeting another definition of an instance of x (*i.e.* a point in D_x).

Now we prove that `tf` is typable with Γ; we consider a trivial live information `live_full` (all the set of RTL variables). We postpone the discussion of typing constraints about uses (predicate `use_ok` in Figure 7) to the end of the paragraph. We consider all edges (i, j) in the CFG and prove that the property (`wf_edge f Γ live_full i j`) holds. If j is not a junction point, then one applies the rule `wt_edge_not_jp`. If j is a junction point, we consider two cases again. If $i = j'$ (the father of j in the DFS tree) the constraints `ASSIG` and `NASSIG` hold by construction of Γ, and the edge is typable. Otherwise, if $i \neq j'$, we consider a variable x not assigned in the ϕ-block in i. By construction of Γ, $(\Gamma\ j\ x) = (\Gamma\ j'\ x)$. Now we must that prove $(\Gamma\ i\ x) = (\Gamma\ j'\ x)$. If the property would not hold, one could conclude from the property above that there exist two distinct points pc_i and $\mathrm{pc}_{j'}$ such that a definition of an instance of x occurs in pc_i (resp. $\mathrm{pc}_{j'}$) and there is a path from pc_i (resp. $\mathrm{pc}_{j'}$) that reaches i (resp. j') without meeting any another point in D_x. This implies that $j \in J(D_x)$. But $D_x = J(D_x)$ by the characterization of iterated dominance frontier, and hence $j \in D_x$. Therefore, an instance of x should then be assigned by a ϕ-function in j. This is a contradiction. This shows Γ is typable, except for constraints about uses.

Let us now look further at the type system without these constraints: if $(\Gamma\ i\ x) = k$, then there exists a definition of x_k at point pc_k such that pc_k dominates i. To find the right subscript of an usage of x at a program point i, the algorithm of Cytron et al. [9] works as follows: it walks up in the dominator tree starting from i and stops at the first point $\mathrm{pc}_{k'}$ in D_x and takes the index k' such that $x_{k'}$ is assigned at this point. Using the same notations, we have to show that necessarily $k = k'$. If $\mathrm{pc}_{k'} = \mathrm{pc}_k$, this holds

trivially. Otherwise, since both $\mathrm{pc}_{k'}$ and pc_k dominate i, either $\mathrm{pc}_{k'}$ dominates pc_k or pc_k dominates $\mathrm{pc}_{k'}$. The first case is impossible because $\mathrm{pc}_{k'}$ is the first ancestor of i in the dominator tree. In the second case, $\mathrm{pc}_{k'}$ does not dominate pc_k. Therefore there exists a path p_1 from the entry to pc_k that does not meet $\mathrm{pc}_{k'}$. Now we know it exists a path p_2 from pc_k to i that never meets a point in D_x. The concatenation of p_1 and p_2 gives us a path from entry to i that never meets pc_k. This is a contradiction with the fact that pc_k dominates i. This concludes the proof of completeness w.r.t minimal SSA.

Completeness with regards to pruned-SSA form can be shown easily by observing that both the algorithm and the type system make the same use of the liveness information (a dead initial variable does not require a ϕ-function).

A Proof sketch of `wt_strict`

We start by giving the formal definition of the predicate `use`:

```
Inductive use_phicode : reg → node → Prop :=
| upc_intro : ∀ pc pred k arg args dst phib
   (PHIB: fn_phicode f pc = Some phib)
   (ASSIG : In (Iphi args dst) phib)
   (KARG : nth_error args k = Some arg)
   (KPRED : index_pred f pred pc = Some k),
   use_phicode arg pred.

Inductive use : reg → node → Prop :=
| u_code : ∀ x pc, use_code x pc → use x pc
| u_phicode : ∀ x pc, use_phicode x pc → use x pc.
```

where the predicate `use_code` is defined in the obvious way (a variable is used if it appears in the right hand side of an assignment, in the condition of an `Icond` instruction, as an argument of a function call...). We also give the formal definition of a well-formed live information:

```
Record wf_live  (live: positive → Regset.t): Prop :=  {
  wf_live_incl : ∀ pc pc' x, RTL.is_edge f pc pc' →  x ∈ (live pc') →
  (x ∈ (live pc) ∨ RTL.assigned_code f pc x) ;

  wf_live_use : ∀ pc x, RTL.use_code x pc → x ∈ (live pc)
}.
```

With these definitions, we now show the following theorem:

```
Theorem wt_strict: ∀ f tf Γ live,
wf_live f live → wt_function tf Γ live →
∀ (xi : SSA.reg) (u d : node), use tf xi u → def tf xi d → dom tf d u.
```

Under the hypotheses, suppose $(\texttt{use tf xi u})$ and $(\texttt{def tf xi d})$. Suppose that `xi` is $(\texttt{x}, \texttt{i})$. The result is immediate when $\texttt{u} = \texttt{d}$. Now, suppose they are different, and that $\neg\ (\texttt{dom tf d u})$. Then, there exists a path p from the entry of `tf` to `u` that does not go through `d`. By applying the following lemma:

```
Lemma use_gamma : ∀ f tf Γ live, wt_function f tf live Γ → wf_live f live →
  ∀ x i u, use f (x,i) u → Γ u x = i.
```

We then obtain that $(\Gamma\ \texttt{u}\ \texttt{x}) = \texttt{i}$. It remains to show that `x` is live at `u` in order to conclude a contradiction by using the following lemma:

```
Lemma gamma_def: ∀ f tf Γ live, wt_function f tf live Γ → wf_live f live →
  ∀ p pc x i d, path tf (fn_entrypoint tf) p pc → def tf (x,i) d →
  (Γ pc x) = i → x ∈ (live pc) → In d (pc::p).
```

When (`use_code tf xi u`), we know `x` is live at point `u` by using (`wf_live f live`) and that (`structural_spec f tf`). Now, if (`use_phicode tf xi u`), we use the well-typedness of the edge from `u` to the ϕ-block at, say, `pc`. The register (`x`, `i`) is an argument, and hence a version for `x` is assigned in the block. The type system specification demands that `x` is live at `pc`. We hence know `x` is live at `u`, thanks to the `wf_incl` field of (`wf_live f live`) record, and the fact that `x` cannot be assigned at `pc` (the function is normalized).

B Proof of `validator_correct`

```
Theorem validator_correct: ∀ (prog:RTL.program) (tprog:SSA.program),
SSA_validator prog tprog = true →
 ∀ s1 t s2, RTL.step s1 t s2 →
   ∀ s1', s1 ≃ s1' → ∃ s2', SSA.step s1' t s2' ∧ s2 ≃ s2'.
```

The proof proceeds by nested case-analysis on the kind of semantic state of `s1`, the relation $\simeq$, and intruction at the program point under consideration. We treat here the main cases, which are when and the instructions are (i) `Iop` and (ii) `Inop` when a ϕ-block is attached at its successor point. Consider `s1` $=$ (`RTL.State s f sp pc rs m`) and `s1'` $=$ (`SSA.State ts tf sp pc rs' m`), such that (`agree` (Γ `pc`) `rs rs'` (`live pc`)).

- Suppose (`Iop op args res pc'`) is the instruction at `pc` in `f`. Hence, `f` makes a step towards the state `s2` $=$ (`RTL.State s f sp pc'` (`rs#res` $\leftarrow$ `v`) `m`). By the hypothesis (`structural_spec f tf`), we know that there is, at point `pc` in `tf`, an instruction (`Iop op args'` (`res`, `i`) `pc'`), and syntax normalization ensures that `pc'` is not a junction point. Hence, no ϕ-block is attached to it in `tf`: the matching state is thus `s2'` $=$ (`SSA.State ts tf sp pc'` (`rs'#`(`res`, `i`) $\leftarrow$ `v`) `m`). In fact both expressions defined by `op` and respectively `args` and `args'` evaluates to the same value `v`: first, the instruction is well-typed, so that it makes correct uses of its variables, with regards to (Γ pc). Second, `rs` and `rs'` agree w.r.t (Γ `pc`). Finally, all uses are live, by hypothesis on `live`. Finally, resulting states are still in the relation $\simeq$, since the update of the local typing specified by the typing rule of the edge (`pc`, `pc'`) takes into account the actual update of the register states in the semantic step.
- Suppose now (`Inop pc'`) is the instruction at `pc` in `f`, with `pc'` a junction point. In this case, `s2` $=$ (`RTL.State s f sp pc' rs m`). We here take for matching state `s2'` $=$ (`SSA.State ts tf sp pc'` (`phi_store k p rs'`) `m`) where `p` is the ϕ-block at `pc'` and `k` is such that `index_pred tf pc pc'` $=$ `Some k`. To show the resulting states stay in the relation, we prove that executing a ϕ-block preserves the agreement between register states (as long at the edge (`pc`, `pc'`) is well typed. Let `x` be an RTL variable that is live at `pc'`. Then, we know that it is live at `pc`, by the definition of `wf_incl` and normalization.
  If no version of `x` is assigned in the block, then we use the agreement between `rs` and `rs'` at `pc`. Otherwise, we reason similarly than in the case of `Iop`. We first use hypothesis `ASSIG` in Figure 8: we have to show that variable `x` and (`x`, (Γ `pc' x`))

have the same value in the new register states, and this is the case, thanks to constraints we impose on the format of ϕ-blocks, as well as the hypothesis `USES` in Figure 8: if the `k`th argument of the ϕ-function is $(\mathtt{x}, \mathtt{j})$, then it means that $(\Gamma\ \mathtt{pc}\ \mathtt{x}) = \mathtt{j}$, and we can conclude using the agreement of register states at `pc`.

All other cases are treated similarly in the full formalization, except for executing a function return, where we need to use some invariants about register states of the caller just before executing the function call (available in the `match_stackframe` predicate, which basically lifts all invariants from regular execution states to the whole callstack).