HAL
open science

Formal Verification of an SSA-based Middle-end for CompCert

Gilles Barthe, Delphine Demange, David Pichardie

To cite this version:

Gilles Barthe, Delphine Demange, David Pichardie. Formal Verification of an SSA-based Middle-end for CompCert. [University works] 2011. inria-00634702v3

HAL Id: inria-00634702

https://inria.hal.science/inria-00634702v3

Submitted on 2 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Verification of an SSA-based Middle-end for CompCert ⋆

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

[1] IMDEA Software Institute
[2] University of Pennsylvania
[3] Harvard University / INRIA Rennes - Bretagne Atlantique

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 that is used by many compilers. In fact, it has remained an open problem to verify formally an SSA-based compiler middle-end. 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, we address two problems raised by Leroy [22]: giving a simple and intuitive formal semantics to SSA, and leveraging the global properties of SSA to reason locally about program optimizations.

1 Introduction

Static single assignment Static single assignment (SSA) form [16] 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 and LLVMC [23], 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 has been 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 SSA programs, 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,

⋆ Partially funded by Spanish project TIN2009-14599 DESAFIOS 10, and Madrid Regional project S2009TIC-1465 PROMETIDOS, and French project ANR Verasco, FNRAE ASCERT and Bretagne Regional project CertLogS.

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 [22], a compiler that is programmed and verified in the Coq proof assistant 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 it foregoes relying on an SSA-based middle end. In [22], 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.

and 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.

The structural properties of SSA are well-identified in the literature, and some proofs of SSA-based analyses and transformations can be found [16, 14, 8]. What is missing in those works is the semantic counterparts of those properties. The proofs are traditionally based on how the SSA-based algorithms work and the information they compute (i.e. properties of the CFG). In particular, the semantic properties and invariants established by the SSA generation algorithm are never expressed precisely. This is probably due to the lack of a definition of a semantics for SSA that would be both formal and close to the intuitive definition given in the seminal papers [1, 16].

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 [28, 27]—increasingly favored by CompCert [34, 35]—for converting programs into SSA form and for GVN. Specifically, we program in Coq verified checkers that validate *a posteriori* results of untrusted computations, and we implement in OCaml efficient algorithms for these computations; we rely on Cytron *et al* algorithm [16] for

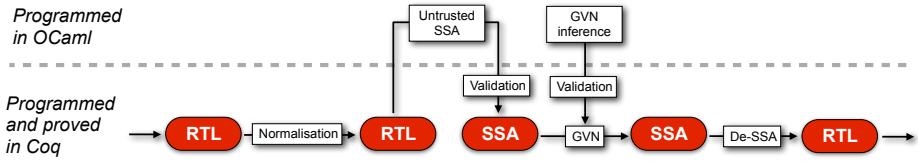


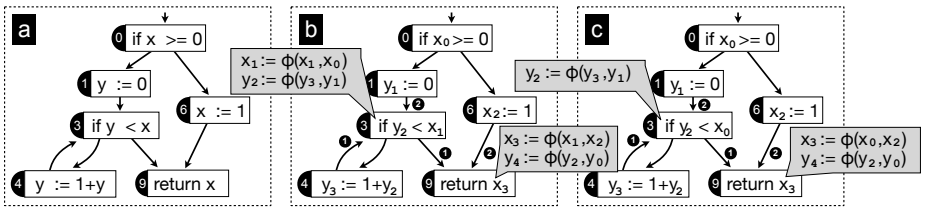
Fig. 1: The SSA Middle-end

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 in Coq. In addition, our work addresses the two issues raised by Leroy [22]. First, we give a simple and intuitive operational semantics for SSA; the semantics follows the informal description given in [16], and does not require any artificial state instrumentation. Second, we define on SSA programs two global properties, called strictness and equational form, allowing 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.

This paper supersedes [5]. The main differences are a proof of completeness for the SSA translation validator, a description of the implementation of the type inference, a more detailed description of the conversion out of SSA and more precise measurements of the GVN optimizer efficacy. The companion Coq development is available online [15].

Contents The paper is organized as follows. Section 2 provides a brief primer on SSA and CompCert. In Section 3, we recall the syntax and semantics of RTL, the CompCert IR at which we plug our middle-end, and we explain the preprocessing of RTL prior to the middle-end. Section 4 defines the SSA language used by our middle-end. Conversion to and out of SSA forms are presented in Section 5 and 8 respectively. Section 7 presents SSA-based optimizations. The correctness of these optimizations rely on a core lemma, which we present in Section 6. We conclude with experimental results in Section 9 and related work in Section 10. 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 in-



(a) Example program (b) Maximal SSA form (c) Minimal SSA form

Fig. 2: Example program and its SSA forms

troducing 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 written according to the pattern

```
Inductive R : A → B → Prop :=
  | Rule1: ∀ a b, ... → R a b
  | Rule2:... → R a b
```

meaning that the relation `R` is a binary predicate (indicated by `Prop`, the type of propositions in Coq) whose arguments are of types `A` and `B` respectively. The relation `R` is defined by two rules `Rules1` and `Rules2`, describing when the proposition (`R a b`) holds for elements `a` and `b` (the hypotheses are indicated by dots).

2 Background

2.1 Static Single Assignment form

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

Converting into SSA form 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 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 node 6. In program b), these two definitions of x 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 point 9 and node 6 is the second one. The semantics of ϕ-functions is given in the seminal paper by Cytron *et al* [16]:

> 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.

Maximal, minimal and pruned SSA There may be several SSA forms for a single program CFG. Figure 2 gives alternative SSA forms for a same initial program. In the maximal SSA form (Figure 2b), a ϕ-function is inserted for all program variables, at each join point. 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 an SSA form with a minimal number of ϕ-functions.

The minimal SSA form is informally specified as follows: a ϕ-function is needed for a given variable at join points that can be reached by at least two distinct definitions of that variable in the initial program. This is captured by the notion of convergence point of CFG paths starting at two distinct definition points of a variable (the join operator in [16]).

Consider the program examples in Figures 2a and 2c. Two definitions of y (at point 1 and 4) can reach the join point 3: a ϕ-instruction is required at node 3 in Program 2c. On the other hand, there is only one definition of x (the initial implicit definition of x) that reaches that point in Program 2a and no ϕ-function is inserted for x at point 3 in Program 2c.

Algorithmically, it is more efficient to determine the placement of ϕ-functions of minimal SSA using the equivalent notion of *dominance frontier*.

Definition 1 (Dominance relation). *A node i in a CFG dominates another node j if every path from the entry node of the CFG to j contains i. The dominance is said to be strict if additionally $i \neq j$.*

Definition 2 (Dominance frontier). *For a node i of a CFG, the* dominance frontier *$DF(i)$ of 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 $DF(S) = \bigcup_{i \in S} DF(i)$.*

Definition 3 (Iterated dominance frontier). *The* iterated dominance frontier *$DF^+(S)$ of a set of nodes S is $\lim_{i \to \infty} DF^i(S)$, where $DF^1(S) = DF(S)$ and $DF^{i+1}(S) = DF(S \cup DF^i(S))$.*

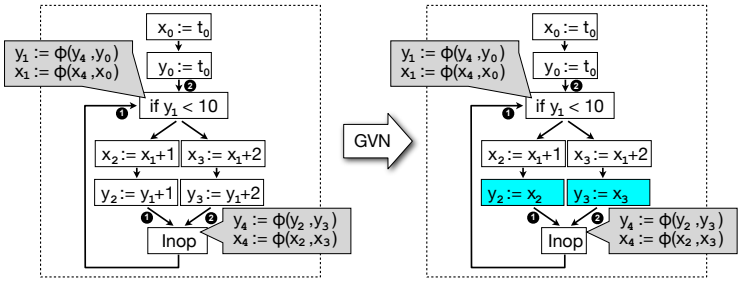


Fig. 3: Common sub-expression elimination (CSE) using GVN

Efficient algorithms for computing the dominance frontiers rely on an effective representation of the dominance relation, the dominator tree. It relies on the notion of immediate dominator.

Definition 4 (Immediate dominator). *The immediate dominator of a node j, written $idom(j)$ is the closest strict dominator of j on every path from the entry node to j. It is uniquely determined.*

Definition 5 (Dominator tree). *It is defined as follows. The start node is the root of the tree. Each node's children are the nodes it immediately dominates.*

In a *minimal SSA* program generated by Cytron et al.'s algorithm, every ϕ-function of an instance x_i of an original variable x appears in a junction point j if and only if j belongs to the iterated dominance frontier of the set of definition nodes of x in the original program.

However, one can achieve more compact SSA forms by observing that, at any junction point, dead variables need not to be defined by a ϕ-function. The intuition is captured by the notion of *pruned SSA* form: a program is in *pruned SSA* form when the ϕ-functions appear at the iterated dominance frontiers 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 (there is a path from j to a use of x that does not redefine x). Compared to minimal SSA (Figure 2c), pruned SSA detects that the ϕ-function for y at point 9 can be removed. Finally, *semi-pruned* SSA forms can be seen as a good trade-off between a minimal SSA form that is not compact enough, and pruned SSA forms that are sometimes too costly to compute (pathological CFGs with can make the liveness analysis intractable). The liveness analysis used for semi-pruned SSA is local to basic blocks: it is hence less precise, but more efficient.

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. Furthermore, several optimizations are naturally formulated on

SSA. 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, 11] allow to compute efficiently such numberings. We briefly explain one such numbering in Section 7.

Figure 3 illustrates how GVN can be used to eliminate redundant computation. The left program is the original code; in this program, for each i, x_i and y_i are assigned the same value number. 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 the numbering is global to the whole program; in fact, any block-local analysis would fail to discover the equality $x_2 = y_2$.

2.2 CompCert

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 go 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).
```

Each phase of the compiler is formally proved relying on simulation techniques, and the formal development of CompCert provides the general correctness theorems of the simulation diagrams. Some parts of the CompCert compiler are not directly proved in Coq. This is the case of the register allocation [22], which is based on a graph coloring algorithm. The graph coloring algorithm is written in OCaml, and then validated a posteriori by a checker written in Coq. The correctness proof of the checker (stating that if a coloring is accepted by the validator, then this is indeed a valid coloring) ensures this compilation phase has the same guarantees than a transformation that would be written and proved directly in Coq, with the additional benefit of abstracting from complex implementation details and heuristics.

3 The RTL language

Our middle-end is plugged at the level of the RTL language in CompCert. This section presents briefly this language. RTL stands for Register Transfer Language. It is a CFG-based three-address like representation of the code, where most of the existing optimisations are performed (constant propagation, removal of redundant cast, tail call detection, local value numbering and a register allocation that includes copy propagation).

3.1 Syntax and semantics

The syntax and semantics of RTL is given in Figure 4. An RTL program is defined as a set of global variables, a set of functions, and an entry node. 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 CFG node to a single instruction, and we stick to this important design choice of CompCert. As explained by Knoop *et al* [19], it allows for simpler implementations of code manipulations and simplifies correctness proofs of analyses or 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`)— we do not discuss here 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 type of states is defined as the tagged union of regular states, call states and return states (Figure 4). We focus on regular states, as we only expose here 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 where variables dereferenced in the C source program reside), the current program point, the registers state (a mapping of local variables to values) and the global memory. The semantics also includes a global environment (of type `genv`) mapping function names and global variables to memory addresses.

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: (`Inop` $\mathtt{pc}'$) branches to the next program point $\mathtt{pc}'$. (`Iop op args res` $\mathtt{pc}'$) performs the arithmetic operation `op` over the values of registers `args` (written `rs##args`), stores the result in `res` (written `rs#res` $\leftarrow$ `v`), and branches to $\mathtt{pc}'$. The instruction (`Iload chk addr args res` $\mathtt{pc}'$) loads a `chk` memory quantity

```
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).

Definition code := PTree.t instr.   type of code 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_entrypoint: node              entry node
}.

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: ∀ ge s f sp pc rs m pc',
      fn_code f pc = Some(Inop pc') →
      step ge (State s f sp pc rs m) ϵ (State s f sp pc' rs m)

 | ex_Iop: ∀ 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) ϵ (State s f sp pc' (rs#res←v) m)

 | ex_Iload: ∀ ge 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 ge (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)

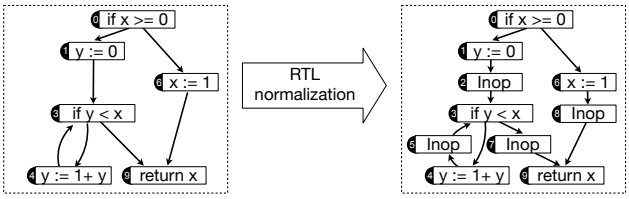


Fig. 5: An RTL program and its normalized version

from the address determined by the addressing mode `addr` and the values of the `args` registers, stores the memory quantity just read into `res`, and branches to $\mathtt{pc}'$.

3.2 Normalizing RTL syntax

Before generating the SSA form of an RTL code, we rely on a structural normalization phase of the RTL code (see Figure 1) that we have added to CompCert, prior to the middle-end proper.

This normalization phase consists of transforming an RTL program into another one, with additional structural constraints on the CFGs of functions. We normalize an RTL CFG so that the only instruction that can lead to a junction point is an (`Inop pc`) instruction. Figure 5 shows an example RTL program and its normalized version.

This normalization phase has been programmed and proved in Coq. One could think this normalization phase is quite anecdotal. But this structural constraint will carry over the SSA form of RTL programs, and will allow for lightening the formal development of our SSA middle-end. As will be pointed out in the next sections, this impacts the formal definitions of the syntax of SSA, but also greatly simplifies its semantics. This also lightens the definition and the proof of our SSA validator, the GVN-based CSE, and the SSA deconstruction. Also, we take care during this normalization to remove from the function CFG all the nodes that are not syntactically reachable from the entry node. Having CFGs with all nodes reachable simplifies many formal definitions, including the one of dominance. Strictly speaking, an unreachable node is dominated by any other node in a graph. So, by eliminating such nodes, we also eliminate this corner case from the definitions.

4 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 capturing essential properties of SSA forms.

```
Definition reg := RTL.reg * idx                    indexed registers

Inductive instr := ...                             RTL-like instructions
                                                   (operating on SSA.regs)
Inductive phiinstr :=
 | Iphi (args: list SSA.reg) (res: SSA.reg).       φ-functions

Definition phiblock:= list phiinstr.               φ-blocks

Record function := {
   fn_sig: signature;            function 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 node
}.
```

Fig. 6: Syntax of SSA

4.1 SSA programs

Syntax Our definition of SSA program distinguishes between RTL-like instructions 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 6 introduces the syntax of SSA.

Compared to RTL functions, SSA functions operate on indexed registers of type `SSA.reg`, and include an additional field `fn_phicode` mapping junction points to ϕ-blocks. The latter are modelled as lists of ϕ-functions, each of the form (`Iphi args res`), where `res` is an indexed register, and `args` a list of indexed registers.

We define structural constraints that allow giving an intuitive semantics to SSA programs. First, we require that the domain of the function `fn_phicode` be the set of junction points. Second, we require that all ϕ-functions in a ϕ-block have the same number of arguments as the number of predecessors of that block. Our last requirement is the normalization criterion of the CFG of SSA functions: all predecessors of a junction point must be (`Inop pc`) instructions.

Strict SSA We consider two essential properties of SSA forms: unique definitions and strictness [12]. 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 CFG paths, and two predicates `dom` and `sdom` for dominance and strict dominance. We also prove many properties of the dominance relation, such as its reflexivity, transitivity,

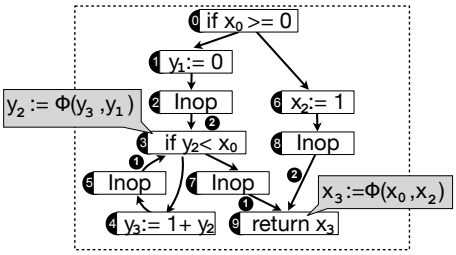


Fig. 7: Example (normalized) SSA program – The variable y_2 is defined at node 3 (in the ϕ-block attached to that program point), and used at points 3 and 4. The variable x_2 is defined at node 6 and used at node 8, the second predecessor of the junction point 9, where x_2 is the second argument of the ϕ-function.

and anti-symmetry. Then, one must define the two predicates `def` and `use` of type `SSA.function` → `SSA.reg` → `node` → `Prop` such that proposition `def f x pc` (respectively `use f x pc`) holds iff the register `x` is defined (resp. used) at node `pc` in the code of the function `f`. Predicate `def` is defined in the obvious way. The definition of `use` is more involved, because of ϕ-functions. A variable is used either by an RTL-like instruction or a ϕ-function:

```
Definition use (f:SSA.function) (x:reg) (pc:node) : Prop :=
  use_code f x pc ∨ use_phicode f x pc.
```

where predicate `use_code` defines when a variable is used in the RTL-like code. It is defined straightforwardly: 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 *etc.* We now explain predicate `use_phicode`. The widely adopted convention is to view ϕ-functions as lazily evaluated. Hence, the kth argument of a ϕ-function is used at the kth predecessor of the corresponding block.

```
Inductive use_phicode : SSA.function → reg → node → Prop :=
 | upc_intro : ∀ f 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)        arg is the kth element of args
    (KPRED : index_pred f pred pc = Some k),     pred is the kth predecessor of pc in f
    use_phicode f arg pred.
```

This matches the semantics we formally define in Section 4.2: ϕ-functions are executed along the edge leading to the ϕ-block. This definition also allows reusing the traditional notion of strictness defined on non-SSA programs. Figure 7 illustrates the definition of predicates `def` and `use`.

Using predicates `def` and `use`, one can then state the unique definition and strictness properties, that defines the strict SSA form. We omit the formal definition of `unique_def` (it is as expected but rather verbose).

```
Definition unique_def (f: SSA.function) := ...

Definition strict (f: SSA.function) :=
  ∀ x u d, use f x u → def f x d → dom f d u.
```

```
Inductive step: SSA.genv → SSA.state → trace → SSA.state → Prop :=
  | ex_Inop_njp: ∀ 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) ϵ (State s f sp pc' rs m)

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

Fixpoint phistore (k:nat) (rs:SSA.regset) (phib:phiblock) : 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. 8: Semantics of SSA (excerpt)

Well-formed SSA programs Finally, the well-formedness of SSA programs is formally defined by the following predicates (the `Record` must be interpreted as a conjunction):

```
Record wf_ssa_function (f:SSA.function) : Prop := {
fn_ssa:        unique_def f;
fn_wf_block:   block_nb_args f;
fn_strict:     strict f;
fn_block_jp:   ∀ jp, join_point jp f ↔ fn_phicode f jp ≠ None;
fn_norm:∀ jp pc, join_point jp f → jp∈(succs f pc) →fn_code f pc=Some(Inop jp)
}.
```

The predicate `block_nb_args` states that ϕ-functions arguments are consistent with the number of predecessors of the CFG node holding the block. In the sequel, we show that the conversion to SSA yields well-formed programs. Besides, our SSA-based optimizations will assume that the input SSA programs are well-formed; in turn, each of the transformations must be proved to preserve well-formedness.

4.2 Semantics

SSA state are similar to RTL states, except that the type of registers and current function are modified into `SSA.reg` and `SSA.function` respectively. We describe now the semantics of SSA programs.

Exploiting normalization for an intuitive semantics The small-step operational semantics is defined on SSA programs that satisfy the structural constraints introduced in the previous paragraph (`wf_ssa_function`).

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 (`Inop pc'`), where one distinguishes whether `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 `pc'` is a junction point, then one executes the ϕ-block attached to `pc'` before the control flows to `pc'`.

Executing ϕ-blocks on the way to `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 ensures that the predecessor of a junction point is an `Inop` instruction. This greatly simplifies the definition of the semantics (ϕ-block can only be executed after an `Inop`), and subsequently the proofs about SSA programs.

Parallel execution of ϕ-blocks Following conventional practice, ϕ-blocks are given a parallel (big-step) semantics. In fact, the SSA generation algorithm ensures, by construction, that the ϕ-functions arguments are never assigned by a distinct ϕ-function in the same block. So this parallel semantics seems to be of little help. But later optimizations will exploit this semantics, that makes explicit the independence of ϕ-arguments with regards to ϕ-function destinations [18, 7].

The semantics of ϕ-blocks is formally defined with `phistore` (Figure 8). When reaching a join point `pc'` from its `k`th predecessor, we update the register set `rs` for each register `res` assigned in the ϕ-block `phib` with the value of register `arg` in `rs` (written `rs#arg`), where `arg` is the `k`th operand in the ϕ-function of `res` (written `nth_error args k = Some arg`). With the same notations, `phistore` satisfies, on well-formed SSA functions, a *parallel assignment* property:

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

5 Translation validation of SSA generation

Modern compilers typically follow the algorithm by Cytron *et al* [16] 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) Build the CFG dominator tree using the algorithm of [21]
(ii) Compute dominance frontiers (bottom-up traversal of the dominator tree)
(iii) ϕ-functions are placed at iterated dominance frontiers of RTL variables
(iv) Rename definitions and uses of RTL variables with the correct indexes (top-down traversal of the dominator tree).

Programming 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 an 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, as presented in Section 2.1. In order to be used in a certified compiler chain, we also show that our validator is sound: it ensure the 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

```
Definition SSA_validator (f: RTL.function) (tf: SSA.function) (Γ: gtype): bool :=
  if    (check_blocks_are_wf tf)            (* ensures block_are_wf tf *)
     && (check_blocks_are_at_jp tf)         (* ensures block_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 Γ)
  else false
```

where `is_well_typed f tf` returns `true` when the function is well-typed with respect to the typing Γ (defined below) in our type system for SSA.

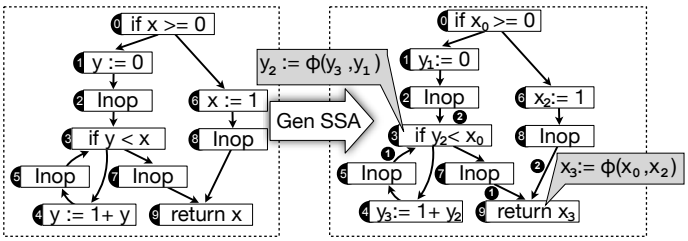
5.1 Type system

The basic idea of our type system is to track for each variable its *most recent* 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` $\rightarrow$ `ltype`; 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 9 as a running example (an RTL program, its pruned SSA form and its type mapping).

Liveness As explained in Section 2, a liveness information can be used to minimize the number of ϕ-functions in a SSA program. Specifically, ϕ-blocks need to assign only live variables. Hence, our type system is parametrized by a function `live` modelling a liveness analysis result, a mapping from CFG nodes to sets of registers: (`live i`) is the set of registers that are live at node `i`.

Formally, the type system does not need to know much about the liveness information, and how it is computed. We only demand that the `live` function



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. 9: An RTL program, its pruned SSA form and a valid typing information

satisfies two properties: (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. For a function `f`, the conjunction of these two properties is denoted by the Coq record (`wf_live f live`):

```
Record wf_live (f: RTL.function) (live: node → Regset.t):= {
    wf_live_use: ∀ pc x, use_code f x pc → x ∈ (live pc)

    wf_live_incl: ∀ pc pc' x,
        is_edge f pc pc' →  x ∈ (live pc') →
        x ∈ (live pc) ∨ assigned_code f pc x ;
}.
```

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` 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 [2]. One could also support semi-pruned forms, by instantiating `live` as the result of the block-local liveness analysis of [10]. All these three liveness information can be shown to be well-formed.

Example 1 (Liveness information). In the last column of the table in Figure 9, we give the liveness information calculated about the variables of the initial RTL function. This information will be used by the validator for validating the pruned SSA form of the program in Figure 9. For instance, the variable y is live at node 3, since it is used at node 3. This variable is however dead (i.e. not live) at point 1 because it is defined (but not used) at this point of the program: it is hence redefined before it is used. At point 6, neither x or y are live. Indeed, the variable x is defined (but not used) at this point (thus redefined before being used at point 9) and the variable y is not used on any path starting at point 6.

Typing rules for instructions The type system for instructions checks that RTL-like instructions make a correct use of variables, and that they do not redefine parameters; its formal definition is given in Figure 10.

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 (through `use_ok`). 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 guard are consistent with the local typing input

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 node 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.

The type system relies on the following convention for function parameters: each of them is given a default index `dft` (in the example of Figure 9, the default index is 0). The first phase of the validator (`check_unique_def`) ensures that parameters are not redefined inside the body of the function.

Example 2 (Typing instructions). We illustrate the typings of instructions with Figure 9. Consider the input local typing at point 3. The uses of $\mathtt{x}_0$ and $\mathtt{y}_2$ are consistent with it, since $(\Gamma\ 3\ \mathtt{x}) = 0$ and $(\Gamma\ 3\ \mathtt{y}) = 2$. The definition of $\mathtt{x}_2$ at node 6 makes the local typing change for variable `x` between nodes 6 and 8: it changes from $(\Gamma\ 6\ \mathtt{x}) = 0$ to $(\Gamma\ 8\ \mathtt{x}) = 2$.

Typing rules for edges and functions The typing rules for edges ensure that ϕ-blocks make a correct use of definitions with regards to a global typing Γ. There are two rules—modelled by the clauses of the inductive relation `wt_edge` in Figure 10.

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: 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. There are three constraints:

- `USES` ensures that the ϕ-arguments `args` passed to ϕ-functions are consistent with 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`
- `ASSIG` ensures that the output local typing is consistant with the definitions in the ϕ-block
- `NASSIG` ensures that, if the variables is not assigned in the ϕ-block, then it means that either it is dead, or the incoming indices for this variable are the same for all predecessors

```
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  →
    {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 →
    {γ} Icall sig id args (r,i) s {γ[r ← i]}

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 k, In (Iphi args (r,k)) block → phiuse_ok r args (preds f j) Γ)
 (ASSIG: ∀ r k, assigned (r,k) block → r ∈ live ∧ (Γ j r) = k)
 (NASSIG: ∀ r, (∀ k, ¬ (assigned (r,k) 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 r, fn_code f i = Some (Ireturn r) → {Γ i} Ireturn r {Γ i})
  ∧ (∀ p, In p (fn_params f) → ∃ r, p = (r, Γ (fn_entrypoint f) r)
                                  ∧ (Γ (fn_entrypoint f) r) = dft).
```

Fig. 10: Type system

Example 3 (Typing ϕ-functions). In Figure 9, the ϕ-function for `x` at point 9 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 node 9 takes into account the definition of $\mathtt{x}_3$ in the block by setting $(\Gamma\ 9\ \mathtt{x})$ to 3. Moreover, no ϕ-function is required for `y` at node 9 since $\mathtt{y} \notin (\mathtt{live\ 9})$, and no ϕ-function is required for `x` at node 3, since $(\Gamma\ 2\ \mathtt{x}) = (\Gamma\ 5\ \mathtt{x})$.

Finally, a function is well-typed with regards to global typing Γ if the local typing induced by Γ at the entry node `fn_entrypoint` is consistent with the parameters, and all edges and return instructions are well-typed. Return instructions do not correspond to any edge, we thus need to add this constraint explicitely.

5.2 Strictness

All SSA programs accepted by the type system are in strict SSA form. 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 f 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 (explained below) about local typings for well-typed functions. The first lemma states that if a variable x_k is used at node i, then it must be that $(\Gamma\ \mathtt{i}\ \mathtt{x} = \mathtt{k})$.

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

The second lemma states that, whenever $(\Gamma\ \mathtt{i}\ \mathtt{x} = \mathtt{k})$, the definition point of variable x_k dominates i.

```
Lemma def_gamma : ∀ f tf Γ live,
  wf_live f live →
  wt_function f tf live Γ →
  ∀ x i d, Γ u x = i → def tf (x,i) d → dom tf d u.
```

Under the hypotheses of `wt_strict`, suppose that x_i is used at point u and defined at point d. By `use_gamma`, we get that $(\Gamma\ u\ x) = i$. We conclude by applying `def_gamma` to get that d dominates u.

5.3 Soundness

The SSA generation phase, as any other phase of a formally verified compiler must be proved correct in the following sense: all behaviors of the SSA form `tf` are also behaviors of the corresponding initial RTL function `f`. In our case, where `tf` is generated by the untrusted generator and validated a posteriori, we have to prove that if the validator accepts the pair `f` and `tf`, then all behaviors of `tf` are also behaviors of `f`.

CompCert already provides the general result that a lock-step forward simulation implies preservation of behaviors, it is thus sufficient to exhibit such a simulation, under the assumption that the validator accepts the pair of programs (i.e. all pairs of RTL and SSA functions in these two programs pass the validator presented in Section 5.1). Such a simulation consists of the three following lemmas:

```
Variable prog:RTL.program.
Variable tprog:SSA.program.

Hypothesis valid_OK : SSA_validator prog tprog = true.

Lemma match_initial_states:
  ∀ s1, RTL.initial_state prog s1 →
  ∃ s2, SSA.initial_state tprog s2 ∧ s1 ≃ s2.

Lemma match_final_states:
  ∀ s1 s2 r, s1 ≃ s2 → RTL.final_state s1 r → SSA.final_state s2 r.

Lemma match_step :
    ∀ s1 t s2, RTL.step (genv prog) s1 t s2 →
    ∀ s'1, s1 ≃ s'1 → ∃s'2, SSA.step (genv tprog) s'1 t s'2 ∧ s2 ≃ s'2.
```

where the binary relation $\simeq$ between semantic states of RTL and SSA carries the invariants needed for proving behavior preservation.

Simulation relation In particular, $\simeq$ should track the correspondance between the registers of semantics states. To do so, we need to capture the semantics of local typings, that specify the correspondance between the variables of `f` and `tf`. This corresponds to the following property:

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

This intuitively means that the value of an initial RTL register `r` is equal to the value of its current version $(\mathtt{r}, \gamma)$ (determined by the local typing γ) in the SSA function. The idea is then to require that, after each computation step, the register states of the RTL and SSA functions agree, with respect to the local typing at the current program point. Note that we will be able to prove such a correspondance only for live variable, and that it is actually sufficient for proving behavior preservation.

Now, defining $\simeq$ only in terms of agreement is not enough to make the proof of simulation go through. We have to constrain more the way RTL and SSA states match. For instance, matching states should have the same memory states and stack pointers. Further, their program counters should be equal. Finally, we add locally to the relation $\simeq$ other invariants relative to the function descriptions of semantic states (e.g. the well-formedness of the SSA function and the well-typedness of the pair of functions).

Formally, the $\simeq$ relation is defined with the inductive `match_states` below, where we omit, for the sake of brevity, the case for relating semantic states of function calls.

```
Inductive match_states : RTL.state → SSA.state → Prop :=
  | match_states_reg: ∀ s f sp pc rs m ts tf rs' Γ live
      (STACKS: match_stackframes s ts)
      (SPEC:   wt_function f tf Γ live)
      (SSA:    wf_ssa_function tf)
      (LIVE:   wf_live f live)
      (AGREE:  agree (Γ pc) rs rs' live),
      (RTL.State s f sp pc rs m) ≃ (SSA.State ts tf sp pc rs' m)

  | match_states_return: ∀ s v m ts
      (STACKS: match_stackframes s ts),
      (RTL.Returnstate s v m) ≃ (SSA.Returnstate ts v m)

   where "s≃t" := (match_states s t).
```

Note that we also define a matching relation for stackframes; this relation basically lifts the invariants of the current functions to the whole callstack of $\simeq$. This way, at each function call return, the invariants for the caller are available through the matching relation over the stackframes of the callee. This avoids to define (rather clumsily) a global hypothesis on the pair of whole RTL and SSA programs stating the invariants hold for all the functions composing the programs.

Proof sketch 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 $\texttt{s1} = (\texttt{RTL.State s f sp pc rs m})$ and $\texttt{s1}' = (\texttt{SSA.State ts tf sp pc rs}' \texttt{ m})$, such that $(\texttt{agree } (\Gamma \texttt{ pc}) \texttt{ rs rs}' (\texttt{live pc}))$.

- Suppose $(\texttt{Iop op args res pc}')$ is the instruction at `pc` in `f`. Hence, `f` makes a step towards the state $\texttt{s2} = (\texttt{RTL.State s f sp pc}' (\texttt{rs}\#\texttt{res} \leftarrow \texttt{v}) \texttt{ m})$. By the hypothesis $(\texttt{structural_spec f tf})$, we know that there is, at point `pc` in `tf`, an instruction $(\texttt{Iop op args}' (\texttt{res}, \texttt{i}) \texttt{ pc}')$, and syntax normalization ensures that $\texttt{pc}'$ is not a junction point. Hence, no ϕ-block is attached to it in `tf`: the matching state is $\texttt{s2}' = (\texttt{SSA.State ts tf sp pc}' (\texttt{rs}'\#(\texttt{res}, \texttt{i}) \leftarrow \texttt{v}) \texttt{ m})$. In fact both expressions defined by `op` and respectively `args` and $\texttt{args}'$ evaluates to the same value `v`: first, the instruction is well-typed, so that it makes correct uses of its variables, with regards to $(\Gamma\ pc)$. Second, `rs` and $\texttt{rs}'$ agree w.r.t $(\Gamma \texttt{ pc})$. 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 $(\texttt{pc}, \texttt{pc}')$ takes into account the actual update of the register states in the semantic step.
- Suppose now $(\texttt{Inop pc}')$ is the instruction at `pc` in `f`, with $\texttt{pc}'$ a junction point. In this case, $\texttt{s2} = (\texttt{RTL.State s f sp pc}' \texttt{ rs m})$. We here take for matching state $\texttt{s2}' = (\texttt{SSA.State ts tf sp pc}' (\texttt{phi_store k p rs}') \texttt{ m})$ with `p` is the ϕ-block at $\texttt{pc}'$ and `k` is such that $\texttt{index_pred tf pc pc}' = \texttt{Some k}$. 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

$(\mathtt{pc}, \mathtt{pc}')$ is well typed. Let $\mathtt{x}$ be an RTL variable that is live at $\mathtt{pc}'$. Then, we know that it is live at $\mathtt{pc}$, by the definition of `wf_incl` and normalization. If no version of $\mathtt{x}$ is assigned in the block, then we use the agreement between $\mathtt{rs}$ and $\mathtt{rs}'$ at $\mathtt{pc}$. Otherwise, we reason similarly than in the case of `Iop`. We first use hypothesis `ASSIG` in Figure 10: we have to show that variable $\mathtt{x}$ and $(\mathtt{x}, (\Gamma\ \mathtt{pc}'\ \mathtt{x}))$ 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 10: if the $\mathtt{k}$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 $\mathtt{pc}$.

All other cases are treated similarly in the full formalization, except for executing a function call or return. At function call, we have to prove a partial invariant about the caller (that holds just before calling the function), and the invariants for the callee. The former will then be used at the callee's return.

5.4 Completeness of the type system

An essential property of our type system is that it accepts all the SSA programs generated by the algorithm by Cytron *et al* [16].

Theorem 1 (Type system completeness). *Let* `f` *be a normalized RTL function and let* `tf` *be the SSA function generated from* `f` *by Cytron et al.'s algorithm. Then there exists Γ such that* `SSA_validator f tf` Γ `= true`.

Proving this theorem requires to identify some key properties about the algorithm presented in [16], which we recall in the Section 5.4. Given this specification, we show in Section 5.4 how to build a global typing, that we prove valid in Section 5.4.

This proof is not formalized in the Coq proof assistant. It would require formalizing the specification in Coq, and proving that the actual running algorithm satisfies this specification. Hence, we would not need to run the validator anymore: by the soundness of our type system, we could deduce a full correctness proof of the SSA generation algorithm a la Cytron.

Specification of Cytron et al.'s algorithm We first review the well-known characterization of the iterated dominance frontier as a fixpoint of the *join* operator J, as well as some properties of the Cytron et al.'s algorithm.

Definition 6 (Join 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, i.e. they both end at j.*

Lemma 1 (Iterated dominance characterization). *For any set of nodes S, the iterated dominance frontier of S, $DF^+(S)$ satisfies $DF^+(S) = J(S \cup DF^+(S))$.*

Proof. See [16], page 467. □

Let `f` be an RTL function, and `tf` the SSA form generated by Cytron's algorithm. For a variable x of `f`, we write def_x the set of definition points of x in `f`, and $\mathrm{def}(x)$ for the (unique) definition point of the variable in `tf`. We express now the way Cytron's algorithm defines the set of definition points of the versions of x in `tf`, and how it determines the right index to use in `tf` when x is used at some point in `f`.

Lemma 2 (Minimal SSA - Definitions). *Define* $D_x = \mathrm{def}_x \cup DF^+(\mathrm{def}_x)$. D_x *is the set of program points where an instance of* x *is defined in* `tf`*, and* $DF^+(\mathrm{def}_x)$ *is the set of nodes where a* ϕ*-function for* x *is inserted.*

Proof. Theorem 2 in [16], page 468. □

Lemma 3 (Minimal SSA - Absence of ϕ-function). *If no instance of a variable* x *is assigned in the* ϕ*-block at node* n*, then a single definition of an instance of* x *reaches all predecessors of* n*, without any other instance of* x *is defined in between.*

Proof. The set of ϕ-functions required for the variable x is by definition [16] $J^+(\mathrm{def}_x)$. We conclude using the definition of the iterated join operator J^+. □

Corollary 1. *If no instance of a variable* x *is assigned in the* ϕ*-block at node* n*, then there exists an instance* x_k *of* x *whose definition strictly dominates* n*.*

Proof. The definition of x_k reaches all predecessors of j, and no instance of x is defined in between. In particular, x_k is defined at a common ancestor of all the predecessors of j. $\mathrm{def}(x_k)$ dominates all predecessors of j; it thus dominates j. □

Lemma 4 (Minimal SSA - Uses). *If* x *is used at point* i *in* `f`*, the variable* x_k *will be used at point* i *in* `tf`*, where* x_k *is the instance of* x *such that* $\mathrm{def}(x_k) \in D_x$ *is the closest ancestor of* i *in the dominator tree of* `f`*.*

Proof. See Lemmas 9 and 10 in [16], pages 473-474. □

Building a witness global typing Let `f` be an RTL function, and `tf` the SSA form generated by Cytron et al's algorithm. We explain now how to build a global typing Γ by a depth-first-search (DFS) traversal of the CFG of `tf`. Each time we reach a new program point j in the DFS, one of its predecessors i in the CFG has already been treated and $(\Gamma\ i)$ is already defined. To define $(\Gamma\ j)$, we distinguish two cases:

Case 1 If j is not a join point, for every RTL variable x, we define $(\Gamma\ j\ x)$ by case analysis:
- if no instance of x is assigned at i in `tf`, then we set $\Gamma\ j\ x = \Gamma\ i\ x$;
- if some instance x_k of x is assigned at i in `tf`, then we set $\Gamma\ j\ x = k$;

Case 2 If j is a join point, for every RTL variable x, we define $(\Gamma\ j\ x)$ by case analysis on the ϕ-block b at j:

- if no instance of x is assigned in b, then we set $\Gamma\ j\ x = \Gamma\ i\ x$;
- if some instance x_k of x is assigned in b then we set $\Gamma\ j\ x = k$.

The global typing given in Figure 9 can actually be computed using this construction. Some properties about this witness global typing Γ can be derived, that we will use in the proof of the next paragraph.

Lemma 5 (Witness global typing: properties). *If $(\Gamma\ i\ x) = k$, then there exists x_k such that $\mathrm{def}(x_k)$ dominates i and any shortest CFG path p from $\mathrm{def}(x_k)$ to i (excluded) does not go through another definition of an instance of x, i.e. a point in D_x.*

Proof. We proceed by induction on the construction of Γ.

- Base case. For all variable x, $(\Gamma\ Entry\ x) = dft$ for all x, and their definition point is the entry point by convention. The condition on shortest paths is trivial since it is emtpy.
- Induction case. Consider the CFG edge (i, j). We proceed by case analysis on j:

  Suppose j is not a junction point. By definition of Γ, there are two cases:
  - $(\Gamma\ j\ x) = k$ because x_k is defined at i. Here, i dominates j, and the shortest path from i to j contains only i and j.
  - $(\Gamma\ j\ x) = (\Gamma\ i\ x)$ because no instance of x is defined at point i. Applying the induction hypothesis, we get that there is x_k such that $\mathrm{def}(x_k)$ dominates i and the shortest path p from $\mathrm{def}(x_k)$ to i does not go through another definition of an instance of x. But i dominates j. By transitivity of the dominance relation, we get that $\mathrm{def}(x_k)$ dominates j. The mininal path $[\mathrm{def}(x_k); \ldots; i; j]$ does not contain any other definition of an instance of x because i does not define a version of x.

  Suppose now j is a junction point. There are again two cases.
  - $(\Gamma\ j\ x) = k$ because an instance x_k is defined in the ϕ-block at point j. We conclude by the reflexivity of dominance.
  - $(\Gamma\ j\ x) = (\Gamma\ i\ x)$ because no instance of x is defined in the ϕ-block at j. Let $(\Gamma\ i\ x) = k$. Here, the induction hypothesis does not permit to conclude.
    In this case, j is not in the iterated dominance frontier of any point in def_x (Lemma 3). Then, by Corollary 1, we get that $\mathrm{def}(x_k)$ dominates j.

□

The witness global typing is a correct typing Now we prove that `tf` is typable with Γ as defined in the previous section. We first consider that `tf` has been generated with a trivial live information `full_live`, containing at each program point the set of all the RTL variables.

We consider all edges (i, j) in the CFG of `tf`, and have to prove that the property (`wf_edge f Γ full_live i j`) holds. We postpone the discussion of typing pruned and semi-pruned SSA versions at the end of the paragraph.

First, we concentrate on verifying that the constraints on the variable definitions are satisfied. We will check that the typing constraints about variables uses (predicate `use_ok` in Figure 10) in a separated lemma.

Lemma 6 (Constraints on definitions). *Let (i, j) be an edge in the CFG of* `tf`*. Then* (`wf_edge f Γ full_live i j`) *holds except for constraints about variable uses.*

Proof. We distinguish two cases.

- Case 1. If j is not a junction point, then i is the sole predecessor of j in the CFG of `tf`, and $(\Gamma\ j)$ is defined in terms of $(\Gamma\ i)$. In this case, we apply the rule `wt_edge_not_jp`.
- Case 2. If j is a junction point. We have to prove that rule `wt_edge_jp` is applicable. We consider two cases.
  - Case 2.1. If i is the predecessor of j in the DFS traversal, $\Gamma\ j$ is defined in terms of $\Gamma\ i$, and the constraints `ASSIG` and `NASSIG` hold by definition of Γ. Therefore the edge is typable.
  - Case 2.2. Let i' be the predecessor of j in the DFS, and suppose $i \neq i'$. We have to prove that `ASSIG` and `NASSIG` hold. Let b the ϕ-block at point j.

    `ASSIG` Let x_k be assigned in b. The live information we use here is `full_live`, thus x is live at point j. Additionally, we have $(\Gamma\ j\ x) = k$ by construction.

    `NASSIG` Let x be an RTL variable such that no instance of x is assigned in block b. Because we use `full_live`, we have to show that $(\Gamma\ j\ x) = (\Gamma\ i\ x)$.

    By definition of Γ, we know that $(\Gamma\ j\ x) = (\Gamma\ i'\ x)$. It is thus sufficient to prove that $(\Gamma\ i\ x) = (\Gamma\ i'\ x)$.

    If the property would not hold, one could conclude from the Lemma 5 that there exist two distinct points ℓ and ℓ' such that a definition of an instance of x occurs in ℓ and ℓ' and there is a path from ℓ (resp. ℓ') that reaches i (resp. i') without meeting any other point in D_x. This implies that $j \in J(D_x) = DF^+(\mathrm{def}_x)$. This leads to a contradiction, as it would mean that j holds a ϕ-node for x (Lemma 2). Therefore, an instance of x should be assigned by a ϕ-function in b. This is a contradiction.

This shows that `tf` is typable with Γ, except for constraints about uses. □

Lemma 7 (Variable uses). *Let (i, j) be an edge in the CFG of* `tf`*. Whenever an instance x_k of x is used at point i in* `tf`*, we have $(\Gamma\ i\ x = k)$.*

Proof. Suppose that $(\Gamma\ i\ x = k')$, with $k' \neq k$. Then, by Lemma 5, we know that $\mathrm{def}(x_{k'})$ dominates i. But x_k is used at point i. By Lemma 4, we hence know that $\mathrm{def}(x_k)$ dominates i. Hence, $p_k = \mathrm{def}(x_k)$ and $p_{k'} = \mathrm{def}(x_{k'})$ both dominate i. Therefore, by the property of the dominance relation, either p_k dominates $p_{k'}$ or $p_{k'}$ dominates p_k. We distinguish three cases:

- Case 1. If $p_k = p_{k'}$, we can conclude directly.
- Case 2. Suppose p_k strictly dominates $p_{k'}$. In this case, p'_k would be between p_k and i in the dominator tree. Then, the closest ancestor of i in the dominator tree that belongs to D_x would be $p_{k'}$, and the index used for x at point i should be, by Lemma 4, p'_k. This is a contradiction.
- Case 3. Suppose $p_{k'}$ strictly dominates p_k. Then, by antisymmetry of the dominance relation, p_k does not dominate $p_{k'}$. This means that there exists a CFG path p from the entry to $p_{k'}$ that does not go through p_k.
  But $(\Gamma\ i\ x) = k'$. Thus, by Lemma 5, we know it exists a CFG path p' from $p_{k'}$ to i that never meets another point in D_x.
  The concatenation of p and p' gives us a path from the entry node of the CFG to i, that never goes through p_k. This contradicts the fact that p_k dominates i.

□

Corollary 2 (Constraints on variable uses). *Let (i, j) be an edge in the CFG of* `tf`*. Then the constraints on the variable uses required for predicate* (`wf_edge f Γ full_live i j`) *to hold are satisfied.*

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).

5.5 Implementation

For the sake of clarity, we have described a non-executable type checker which assumes that structural constraints are satisfied. For efficiency reasons, the Coq implementation of the type system is in fact a bit more complex. In particular, it performs type inference rather than type checking. Additionally, it 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 9, our implementation is ten times faster than a type checker derived naively from the non-executable type system of Figure10. We now give an overview of the implementation.

The untrusted SSA generator does not actually computes the whole code of the SSA form of a function. It provides to the type checker the information

that is strictly necessary. We will call this information a hint; it is made of two maps. The first maps indicates for each CFG node, the instance index this node potentially defines. The second map provides the same kind of information but for ϕ-blocks: at a given node, it indicates whenever a block is required, and what index to use for the definition of variables. The signature of the external generator for SSA is thus the following:

```
Definition SSA_hint := (PTree.t index) * (PTree.t (PTree.t index)).
Variable extern_SSA_gen: RTL.function → (node → Regset.t) → SSA_hint.
```

Then, given this hint, both the type inference and the code generation will be performed simulatenously:

```
Definition type_infer:
  RTL.function → (node → Regset.t) → SSA_hint → option SSA.function.
```

Since the hint might be incorrect, the type inference may not be able to generate any SSA function, hence the option type of its result. This type inference builds a global typing Γ using the SSA hint, in a way that is similar to the algorithm described in Section 5.4. We then prove that, whenever the inference is successful, the generated function is well typed in the type system described in Figure 10:

```
Theorem type_infer_correct: ∀ f tf live hint,
   wf_live f live →
   type_infer f live hint = Some tf →
   ∃ Γ, SSA_validator f tf Γ = true.
```

Finally, our SSA generation algorithm is described by the following snippet.

```
Definition ssa_gen (f: RTL.function) : option SSA.function :=
  let live := (LiveAnalysis f) in
  let hint := extern_gen_ssa f live in
  type_infer f live hint.
```

First, a liveness analysis (implemented in Coq) is performed on the RTL function. This liveness information is shared by the external untrusted SSA generator (written in OCaml) and the type inference. The external SSA generator computes the information (hint) required for the type inference to perform the actual SSA code generation, whilst verifying that the validity of the hint.

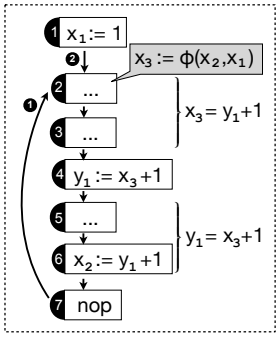
6 The SSA equational lemma

In this section, we introduce the *equation lemma* that supports 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. Using the equational lemma, we will be able in Section 7 to formalize and prove correct a GVN optimization.

6.1 Equational 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.

> *Because every assignment creates a new value name it cannot kill (i.e. invalidate) expressions previously computed from other values. In particular, if two expressions are textually the same, they are sure to evaluate the same result. [9]*



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 thus are inconsistent. In fact, equations are only valid at program nodes 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 intensively uses the strictness property of well-formed SSA programs.

6.2 Application with Copy Propagation

We conclude with a succinct account of applying the corollary to prove the soundness of copy propagation (CP)—recall that CP will search for copies `x := 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 `rs#y` = `rs#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.

7 Validation of Global Value Numbering

This typical SSA-based optimization assigns to variables an identifying number such that variables with the same number will hold equal values at execution

```
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. 11: Valid numbering

time. Several variations of the optimization have been proposed [1, 11]. They are generally presented has highly optimised iterative algorithms.

We follows [1] but clearly separate the optimisation into two phases. First, an untrusted analysis, written in OCaml, computes a numbering of SSA programs and for each program point where the numbering detects a redundant computation $x := e$, it provides a candidate y for replacing the previous operation by $x := y$. In a second phase, a validator checks the numbering and the proposed assignment simplification.

To achieve this separation of concerns it is useful to reconsider GVN from an abstract interpretation point of view: the analysis computes a fixpoint in the abstract domain of congruence partitions, where partitions are modelled as mappings $\mathcal{N}$: `reg` $\rightarrow$ `reg` that map a register to the canonical register of its equivalence class (its *number*). The abstract domain is ordered w.r.t. to a partial order $\sqsubseteq_{\text{GVN}}$ that coincides with the reverse inclusion of equivalence kernels—recall that the equivalence kernel of $\mathcal{N}$ is the relation $\sim_{\mathcal{N}}$ defined by $\mathtt{x} \sim \mathtt{y}$ if and only if $\mathcal{N}\ x = \mathcal{N}\ y$.

$$\mathcal{N}_1 \sqsubseteq_{\text{GVN}} \mathcal{N}_2 \text{ iff } \sim_{\mathcal{N}_1} \ \supseteq \ \sim_{\mathcal{N}_2}$$

The notion of valid numbering is formally defined in Figure 11. First, we define for each numbering $\mathcal{N}$ the relation $\equiv^{\mathcal{N}}$ as the smallest reflexive relation identifying: (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, for a numbering $\mathcal{N}$ to be valid (see `GVN_spec`), its equivalence kernel must not contain a pair of distinct function parameters and it must moreover be included in $\equiv^{\mathcal{N}}$. The latter ensures the intended post-fixpoint property: if we note n_{param} the numbering that associates each register to itself if it is a function parameter and a default register otherwise, then (`GVN_spec` $\mathcal{N}$) is equivalent to $F(\mathcal{N}) \sqsubseteq_{\text{GVN}} \mathcal{N}$ with F the operator defined by $F(\mathcal{N}) = n_{\text{param}} \cap \equiv^{\mathcal{N}}$.

$$\texttt{GVN_spec}\ \mathcal{N} \text{ iff } n_{\text{param}} \cap \equiv^{\mathcal{N}} \ \sqsubseteq_{\text{GVN}} \ \mathcal{N}$$

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 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 node pc, and x and y are two registers whose definition strictly dominate pc, then $\mathcal{N}$ x = $\mathcal{N}$ 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. The definition of def_sdom given below takes care of the case where x is assigned in a ϕ-block at pc (noted assigned_phi f pc x). Indeed, a ϕ-block at pc is actually executed before reaching pc while a normal assignement at pc will takes effect after leaving pc.

```
Inductive def_sdom (f:function) (x:reg) (pc:node) : Prop :=
| def_sdom_def_sdom : ∀ def_x,
  def f x def_x → sdom f def_x pc → ¬ assigned_phi f pc x → def_sdom f x pc
| def_sdom_def_phi :
  assigned_phi f pc x →  def_sdom f x pc.
```

Let us illustrate the gamma property with Figure 3; registers x_2 and y_2 share the same numbering; they are indeed equal just after the assignment of y_2 but not before.

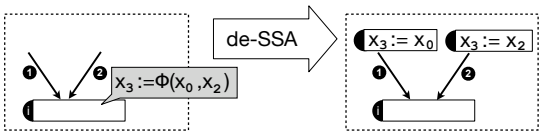
Next, we describe the Coq implementation for optimizing SSA programs. The implementation takes as input a numbering $\mathcal{N}$, and a partial mapping crep that takes as input a register x and node pc and returns, if it exists, a register y such that x and y are related by the equivalence kernel of $\mathcal{N}$, 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 checks whether crep provides a canonical representative y for x at node 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. It follows a standard forward simulation proof where the correctness of the numbering is proved at the same time as the simulation itself. Noticeably, the CFG normalization turned out to be extremely valuable for this proof. Indeed, consider a step from node pc to node 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) ∨ (def f x pc) which is particularly useful to exploit the hypothesis that (gamma $\mathcal{N}$ pc rs) holds.

8 Conversion out of SSA

The final phase of the middle-end converts SSA programs back to RTL programs, so that they can be further processed by the CompCert back-end, starting with register allocation. Several approaches have been proposed [30, 7]. As a first step, we decided to use the conversion described in [16]. The basic idea of this conversion is to substitute each ϕ-function with one variable copy at each predecessor of the junction points:



However, there are several pitfalls to avoid: performing naively the destruction of SSA by such copy insertions can lead to the non-preservation of behaviors. Two problems were identified by Briggs et al. in [10]: the presence of critical back-edges (that can lead to the so-called lost-copy problem) and the swap problem. We review both problems in the next sections, and explain how we tackle these two issues. As noted in [10], the swap problem is a particular case of the lost-copy problem but we tackle the issues differently in our development. We finish this section with an overview of the correctness proofs, that shows how the normalization phase can be exploited.

8.1 Critical edges

In the presence of critical back-edges in the progran CFG, the simple copy insertion described above becomes incorrect. We first recall the definition of a critical edge.

Definition 7 (Critical edge). *A critical edge is an edge (i, j) whose entry i has several successors and whose exit j has several predecessors.*

Figure 12a describes the situation where the exit of the critical edge (i, j) holds a ϕ-block. The problem here is that, the copies cannot be inserted at the predecessors, because they would be executed on some paths that initially did not reach the ϕ-function. This can lead to the well-known lost-copy problem in the presence of critical back-edges and optimizations such as copy folding (see [10]). But copies cannot either be inserted at the edge sink, because it would overwrite the values coming from the others predecessors.

(a) Critical edge - before splitting

(b) Splitted critical edge

Fig. 12: Critical edges and naive copy insertion

One solution to this problem is to split critical edges, as shown in Figure 12b. After the critical edge (i, j) has been splitted, the copies for replacing the ϕ-function can be safely inserted at the predecessors of the node holding the block (including the newly inserted node k). Compilers that operate on *basic-block* CFG graphs carefully avoid edge splitting for efficiency concern in later optimization stages. But this is at the cost of making de-SSA algorithms significantly more complex.

In our case, the normalization we impose on SSA programs pleasingly ensures the absence of critical edges in their CFG. One could fear that the critical edge splitting implied by the normalization could impact later phases of the compiler, but the representation of programs inherited from CompCert deflates this penalty cost. RTL graphs, and thus SSA code graphs, are single-instruction graphs: replacing ϕ-functions with copies automatically splits critical edges by the insertion of code.

8.2 The swap problem

One must also take care of the semantics of ϕ-blocks. They are given a parallel semantics, and, because of optimizations, it is not in general equivalent to a sequential interpretation. Indeed, performing copy propagation on SSA can modify the code, so that ϕ-functions argument and destination registers are no longer independent: a variable x_i can appear both as a source and a target of distinct ϕ-functions in a single ϕ-block. In this case, the copies inserted for converting out of SSA must be sequentialized. This can be done at the reasonable price of inserting at most one temporary variable [29].

In the current state of our development, our conversion out of SSA fails on such ϕ-blocks. This is not a limitation in practice, as the GVN optimization we perform on the code does not cause problems of that kind; from the SSA generation until its destruction, the parallel semantics of ϕ-blocks is ensured to be equivalent to the sequential one. We however plan to reuse the work of Rideau et al. [29] which provides an algorithm for transforming a set of parallel moves into an equivalent sequence of elementary moves (using additional temporaries). This algorithm is already used in CompCert when enforcing calling conventions during the compilation of function calls.

8.3 Correctness proof

For proving the transformation correct, we proceed by giving a forward plus 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. We also take advantage of the normalization in this proof: the execution of an `Inop` instruction leading to a junction point with a ϕ-block matches the corresponding inserted copies. Without the normalization, all RTL-like instructions would have resulted in a different case in the proof.

9 Implementation and experimental results

We have plugged in Compcert 1.8.2 our SSA middle-end made 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. It does not add any axioms to CompCert. We use the Coq extraction mechanism to obtain an SSA-based certified compiler, that we evaluate experimentally using the benchmark suite provided with the CompCert distribution. These include around 75.000 lines of C code, and fall into three categories of programs (from 20 to 5.000 LoC): small computation kernels, a raytracer, and the theorem prover Spass[4]. Below we briefly comment on three key points: efficiency of the SSA validator; effectiveness of the GVN optimizer; efficiency of generated code.

9.1 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.

9.2 Effectiveness of GVN optimizer

We measure the effectiveness of our GVN analyzer by performing a GVN-based CSE right after a (Local Value Numbering) LVN-based CSE implemented in

[4] Spass is the largest (69.073 LoC), we only use it to evaluate the compilation time.

	x86				PPC			
	`Iop`	LVN	GVN	GVN only	`Iop`	LVN	GVN	GVN only
c. kernels	3,494	163	55	216	3,142	422	54	472
raytracer	2,303	131	29	159	2,755	303	21	322
spass	51,640	122	19	99	52,451	392	43	306
TOTAL	57,437	416	103	474	58,348	1,117	118	1,100

Table 1: **GVN optimizer (x86 and PowerPC backends)** For each set of benchmarks, we count the number of initial `Iop` in the RTL function (column `Iop`), the number of `Iop` optimized away by the LVN-CSE optimization of CompCert (column LVN) and the number of `Iop` optimized away by our GVN-CSE optimization, right after CompCert's LVN-CSE (column GVN). We also measure the number of `Iop` that GVN optimizes away without any prior LVN-CSE (column GVN only).

CompCert. We count how many additional `Iop` instructions are optimized by this additional CSE phase. For efficiency concerns about the generated code, we need to keep the LVN phase that optimizes redundant memory loads (currently, this is not done by our GVN optimizer). To keep the comparison fair, we allow CompCert CSE to optimize around function calls—this is disabled in CompCert to keep the register pressure low. The results are given in Table 1, for two backends, x86 (left) and PowerPC (right). The overall improvement is significant. Our global CSE optimizes an additional 10% of `Iop` instructions on PowerPC and an additional 25% on x86.

We also measure how the GVN behaves, without the preliminary LVN optimization. Our global CSE manages to optimize all the `Iop` instructions that are optimized by LVN, except 2 for the small computation kernels, and 1 for the raytracer. For Spass, however, GVN only optimizes half the number of `Iop`. This is due to the fact that in CompCert's LVN, the redundant load elimination and CSE optimizations are interdependent (detecting some redundant loads helps in turn detecting new common sub-expressions, and common sub-expression elimination can lead to extra load redundancy detection).

9.3 Efficiency of the Generated code

To assess the efficiency of the generated code, we have compiled the benchmarks with three compilers: CompCert, our version of CompCert extended with a SSA middle-end (CompCertSSA), and `gcc -O1`. Figure 13 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.

During our experiments, we observed that the computation time of the allocator is sometimes rather long, and can result in a lot of spill code. The quality of

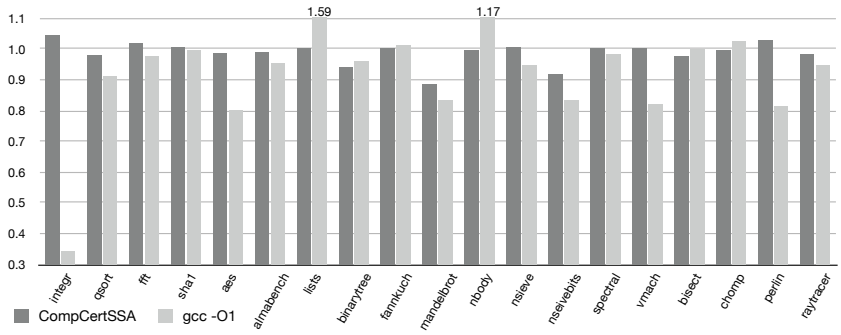


Fig. 13: Execution times of generated code

the allocation is essentially impacted by our current SSA deconstruction, that introduces many copies and artificial interferences between variables of a ϕ-block, imposing more constraints on the allocator. We expect that performance improves significantly by enhancing our middle-end with additional optimizations, and by refining our SSA deconstruction, with either techniques similar to [7] or an SSA-based register allocator [18]. We leave this interesting challenge for future work.

10 Related Work

10.1 Machine-checked formalizations

Blech *et al* [6] 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 the formal SSA semantics, 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 [24] use Isabelle/HOL to formalize and verify the conversion of CFG programs into SSA form. However, their transformation may yield non-minimal SSA, and does not aim extraction into efficient code. Moreover, it is not clear whether their semantics of SSA can be used to reason about optimizations.

Zhao *et al* [37] formalize the LLVM SSA intermediate representation in Coq. They define and relate several formal semantics of LLVM, including a static and dynamic semantics. More recently [36], they verify an SSA generation algorithm based on Aycock and Horspool algorithm [4]. This parallel work shares some similarities with us concerning the semantics of ϕ-blocks and the SSA equational lemma. In contrast to our work, their SSA generation algorithm is directly verified (and not *a posteriori* validated) but runs in quadratic time while our

generator and its validator run in almost linear time thanks to the Lengauer-Tarjan algorithm. Their SSA representation has not been used yet to verify a representative SSA-based optimizations such as GVN.

Finally, there are several machine-checked accounts of Continuation Passing Style translations, e.g. [17, 13], closely related to conversion to SSA form [3].

10.2 Translation validation and type systems

Menon *et al* [26] propose a type system that can be used to verify memory safety of programs in SSA form, but their system does not enforce the SSA property. Matsuno and Ohori [25] define a type system equivalent to SSA: every typable program is given a type annotation makeing explicit def-use relations. Their type system is similar to ours except they type check one program w.r.t. annotations while we type check a pair of a RTL and a SSA program. They show that common optimizations such as dead code elimination and CSE are type-preserving. But they do not prove the semantics preservation of the optimizations. Stepp *et al* [31] report on a translation validator for LLVM. Their validator uses Equality Saturation [32], which views optimizations as equality analyses. Their tool does not validate GVN. Tristan *et al* [33] independently report on an a translation validator for LLVM's inter-procedural optimizations. This tool supports GVN, but is currently not certified.

11 Conclusion and Future Work

The SSA form is a popular IR in the compilation community that has been used with great success in many program optimizations since its inception in the late 80's. The structural properties of unique definition and strictness, as well as the parallel semantics given to ϕ-blocks are the ingredients that led to this success.

If those properties seems rather simple and intuitive, the algorithms underlying the generation of SSA – who actually establish those properties – rely on complex properties of graphs (e.g. the dominator tree or dominance frontiers), that are difficult to justify formally. Moreover, the very semantics of the SSA form has kept for a long time at an informal level. As a consequence, the correctness proof of SSA-related algorithms (i.e. generation, optimizations, and destruction), were until very recently not formally proved correct. Over the past few years, some interesting attempts have be made to formalize the semantics of SSA, but these formalizations were rather distant from the intuitive semantics presented in the seminal papers. The correctness of SSA-based analyses and optimizations is usually proved using structural arguments on the CFG only, and the semantic properties and invariants of SSA remain unclear.

In this paper, we have defined a formal semantics for SSA, that is both close to the intuitive definition of the early papers, and amenable for formal reasoning, as witnessed by our fully verified SSA-based middle-end for the verified CompCert C compiler. Thanks to our choices made in the representation of programs,

this semantics integrates well in the CompCert architecture. The translation validation approach we use for the conversion to SSA and the GVN optimization allows the middle-end implementing state-of-the-art algorithms, while keeping close to the essence of those phases and to the high-level properties they should satisfy in order to preserve the behaviors of programs. The focused nature of our SSA validator makes it complete with regard to one of the reference implementations of the SSA generators [16], where ϕ-functions placement is determined using dominance-frontiers. We also identified and isolated the semantic counterpart of the structural properties of SSA into a dedicated invariant lemma – that holds at all points of the execution – on which rely the correctness proof of several SSA-based optimizations.

A priority for further work is to achieve a tighter integration of our middle-end into CompCert. There are three immediate objectives: (i) enhancing our SSA middle-end to handle memory aliases as done by CompCert's RTL-based middle-end, (ii) implementing an SSA-based register allocator [18], and (iii) verifying more SSA-based optimizations, including PRE [14], or lazy code motion [20]—we expect that our implementation of GVN will provide significant leverage there. Eventually, it should be possible to shift all CompCert optimizations into the SSA middle-end.

Acknowledgments We thank Xavier Leroy for his thoughtful feedback.

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. *Modern compiler implementation in ML: basic techniques*. Cambridge University Press, 1997.
3. A.W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33, 1998.
4. J. Aycock and R. N. Horspool. Simple generation of static single-assignment form. In *CC'00*, volume 1781 of *LNCS*, pages 110–124, 2000.
5. Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert. In *ESOP 2012*, volume 7211 of *LNCS*, pages 47–66. Springer-Verlag, 2012.
6. 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.
7. B. Boissinot, A. Darte, F. Rastello, B. Dupont de Dinechin, and C. Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proc. of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, Washington, DC, USA, 2009. IEEE Computer Society.
8. B. Boissinot, S. Hack, D. Grund, B. Dupont de Dinechin, and F. Rastello. Fast Liveness Checking for SSA form Programs. In *Proc. of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, New York, NY, USA, 2008. ACM.

9. M. M. Brandis and H. Mössenböck. Single-pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Trans. Program. Lang. Syst.*, 16(6), November 1994.
10. P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *SPE*, 1998.
11. P. Briggs, K.D. Cooper, and L.T. Simpson. Value numbering. *SPE*, 1997.
12. Philip Brisk. *Advances in Static Single Assignment form and register allocation.* PhD thesis, Los Angeles, CA, USA, 2006. AAI3254798.
13. A. Chlipala. A verified compiler for an impure functional language. In *POPL'10*. ACM, 2010.
14. F. Chow, S. Chan, R. Kennedy, S-M. Liu, R. Lo, and P. Tu. A New Algorithm for Partial Redundancy Elimination Based on SSA Form. In *Proc. of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, New York, NY, USA, 1997. ACM.
15. Companion web page, 2012. `http://compcertssa.gforge.inria.fr`.
16. 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.
17. Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *LPAR'07*, LNCS. Springer-Verlag, 2007.
18. S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA form. In *CC*, LNCS. Springer-Verlag, 2006.
19. J. Knoop, D. Koschützkil, and B. Steffen. Basic-block graphs: Living dinosaurs? In *CC*, LNCS. Springer-Verlag, 1998.
20. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *PLDI'92*, 1992.
21. T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1979.
22. X. Leroy. A formally verified compiler back-end. *JAR*, 43(4), 2009.
23. The LLVM compiler infrastructure. `http://llvm.org/`.
24. W. Mansky and E. Gunter. A framework for formal verification of compiler optimizations. In *ITP'10*. Springer-Verlag, 2010.
25. Y. Matsuno and A. Ohori. A type system equivalent to static single assignment. In *PPDP'06*. ACM, 2006.
26. 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.
27. G. Necula. Translation validation for an optimizing compiler. In *PLDI'00*. ACM, 2000.
28. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, LNCS. Springer-Verlag, 1998.
29. L. Rideau, B.P. Serpette, and X. Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *JAR*, 2008.
30. V.C. Sreedhar, R. Ju, D.M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *SAS'99*. Springer-Verlag, 1999.
31. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV'11*, LNCS. Springer-Verlag, 2011.
32. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL'09*. ACM, 2009.
33. J.B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI'11*. ACM, 2011.

34. J.B. Tristan and X. Leroy. Verified validation of lazy code motion. In *PLDI'09*. ACM, 2009.
35. J.B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL'10*. ACM, 2010.
36. J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI'13*. ACM, 2013.
37. J. Zhao, S. Zdancewic, S. Nagarakatte, and M. Martin. Formalizing the LLVM intermediate representation for verified program transformation. In *POPL'12*. ACM, 2012.