Mechanised Semantics for Gated Static Single Assignment

Yann Herklotz
Imperial College London
London, UK
yann.herklotz15@imperial.ac.uk

Delphine Demange
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
delphine.demange@irisa.fr

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
sandrine.blazy@irisa.fr

Abstract

The Gated Static Single Assignment (GSA) form was proposed by Ottenstein et al. in 1990, as an intermediate representation for implementing advanced static analyses and optimisation passes in compilers. Compared to SSA, GSA records additional data dependencies and provides more context, making optimisations more effective and allowing one to reason about programs as data-flow graphs.

Many practical implementations have been proposed that follow, more or less faithfully, Ottenstein et al.'s seminal paper. But many discrepancies remain between these, depending on the kind of dependencies they are supposed to track and to leverage in analyses and code optimisations.

In this paper, we present a formal semantics for GSA, mechanised in Coq. In particular, we clarify the nature and the purpose of gates in GSA, and define control-flow insensitive semantics for them. We provide a specification that can be used as a reference description for GSA. We also specify a translation from SSA to GSA and prove that this specification is semantics-preserving. We demonstrate that the approach is practical by implementing the specification as a validated translation within the CompCertSSA verified compiler.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program verification**; • **Software and its engineering → Semantics**.

Keywords: Verified Compilation, SSA, Gated SSA

ACM Reference Format:
Yann Herklotz, Delphine Demange, and Sandrine Blazy. 2023. Mechanised Semantics for Gated Static Single Assignment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3573105.3575681

CPP '23, January 16–17, 2023, Boston, MA, USA

ACM ISBN 979-8-4007-0026-2/23/01…$15.00
https://doi.org/10.1145/3573105.3575681

1 Introduction

Important program optimisations in compilers require global reasoning on the values that are computed by the program. More precisely, one has to be able to statically infer facts about program instructions that span several basic blocks in the program's control-flow graph. Furthermore, this global reasoning has to account for side-effects, such as variable modifications, memory writes, or observable outputs. More generally, dependencies between instructions have to be carefully analysed. Implementations of these optimisations are therefore subtle and particularly error-prone.

To overcome these difficulties, many techniques have been proposed by the compiler community. These range from adequate program intermediate representations (IRs) to sophisticated auxiliary data structures that complement the program representation. Most notably, Rosen et al. [24] proposed the static single assignment (SSA) form in the late 80's, which enables an extensive global redundancy elimination of computations. In SSA, each variable has a single definition point: each time a variable is modified in the initial program, a new version of that variable is introduced. To ensure this property on programs with branches and junction points, SSA provides a dedicated instruction, the ϕ-instruction. At a control-flow junction point with, say, three predecessors, a ϕ-instruction $x_4 = \phi(x_1, x_2, x_3)$ selects among the three versions x_1, x_2 and x_3 of x, the one which should be assigned to x_4, depending on the control-flow execution path that led to the junction point. The SSA form also has a built-in representation for use-definition (*use-def*) chains, which provide basic dependency information about the instructions of the program: an instruction using an SSA variable depends on the unique instruction defining that variable. This led to a wide range of *sparse* program optimisations, see e.g. Wegman and Zadeck [30], demonstrating the great success of SSA, which is now available in many production compilers.

Since its introduction, several extensions of SSA have been proposed to enrich the tracking of dependency information between program instructions, as well as the tracking of how properties propagate in SSA programs, see e.g. the Static Single Information form [2] and its use in static analysis. The extension we study in this paper is *Gated* SSA, which partially transforms control-flow dependencies into data dependencies. Gated SSA (GSA) was introduced by Ottenstein et al. [23] to extend *program dependency graphs* with

the SSA property, leading to an IR devoted to compilation targeting data-flow architectures and simulators. In GSA, ϕ-instructions are augmented with control-flow information, so that they become "referentially transparent" in the following sense: while in the traditional SSA form, one needs to keep track of the execution control-flow to select the correct argument of a ϕ-instruction, the GSA form extends ϕ-instructions with *gates*, i.e. Boolean conditions characterising the control-flow paths leading to that ϕ-instruction and hence determining which ϕ-argument has to be selected. For example, suppose an SSA ϕ-instruction $x_3 = \phi(x_1, x_2)$ is placed at a junction point whose corresponding branching point is a test on condition c. In GSA, the ϕ-instruction would be gated as follows $x_3 = \phi((c, x_1), (\overline{c}, x_2))$, meaning if condition c evaluates to true, then it will select x_1, otherwise it will select x_2. The selection of the ϕ-arguments is no longer guided by the dynamic control-flow predecessor that led there, but is uniquely determined by the gate that evaluates to true. Note that the gate's conditions will refer to program variables and are be evaluated with respect to the current execution state. Most of the successful applications of GSA can be found in the domain of parallelising compilers [3, 16, 29] where dependency analysis plays a crucial role. More recently, GSA has also been applied to the field of high-level synthesis [11], and thread divergence and thread aggregation for efficient compilation to GPUs [25].

Despite the impressive advances in the field of compiler verification, where optimisation techniques are becoming more realistic and closer to the ones used in production non-verified compilers, GSA-based techniques remain largely unexplored. This can be explained (i) by the lack of a reference GSA form, as the notion of dependence is carefully tuned to each precise use case, and (ii) by the absence of a clear and precise semantic description of GSA in the compiler literature, especially regarding the evaluation of gates. The GSA form is central and crucial for these compilation techniques, as it provides a useful base on top of which more sophisticated analyses and optimisations can be built.

In this paper, we aim to bridge the semantic gap existing between GSA and the well-understood SSA form. Providing a fully fledged and fully verified GSA code optimiser is far beyond the scope of the paper. Instead, we focus here on essential components of GSA, i.e. the specification, construction and semantics of gates. We conducted our formalisation in the Coq proof assistant. Our contributions are as follows:

1. We define a semantics for GSA in Section 4, featuring a control-flow insensitive semantics for gates;
2. We provide a formal specification of a GSA construction algorithm in Section 5, and we prove it implies the semantic preservation of SSA;
3. We apply our work to a realistic SSA-based middle-end, demonstrating empirically the validity and relative completeness of our specification in Section 6.

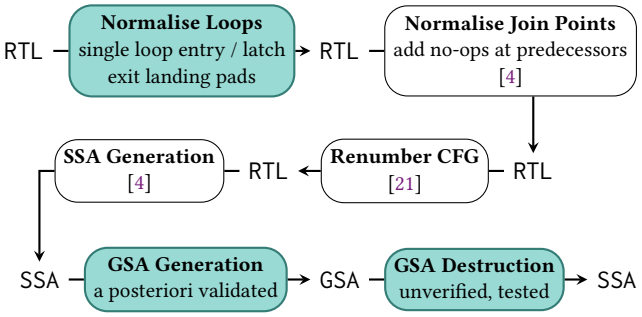


Figure 1. Key phases to integrate GSA into CompCertSSA. This work comprises the phases in the shaded boxes in addition to the definition of the GSA language. The other phases which are not shaded are provided by either CompCert or CompCertSSA.

This paper is organised as follows. First, Section 2 illustrates GSA form through an example. In Section 3, we recall the required background on CompCertSSA, the formally verified SSA middle-end where we integrate our formalisation. Section 4 defines our GSA representation. Section 5 details our conversion from SSA to GSA and its proof of correctness. Section 6 describes the key phases we implement to integrate GSA into CompCertSSA: this comprises a normalisation of loops, the translation from SSA to GSA, as well as an non-verified destruction phase of GSA back to SSA (see Fig. 1). Related work is discussed in Section 7, followed by concluding remarks.

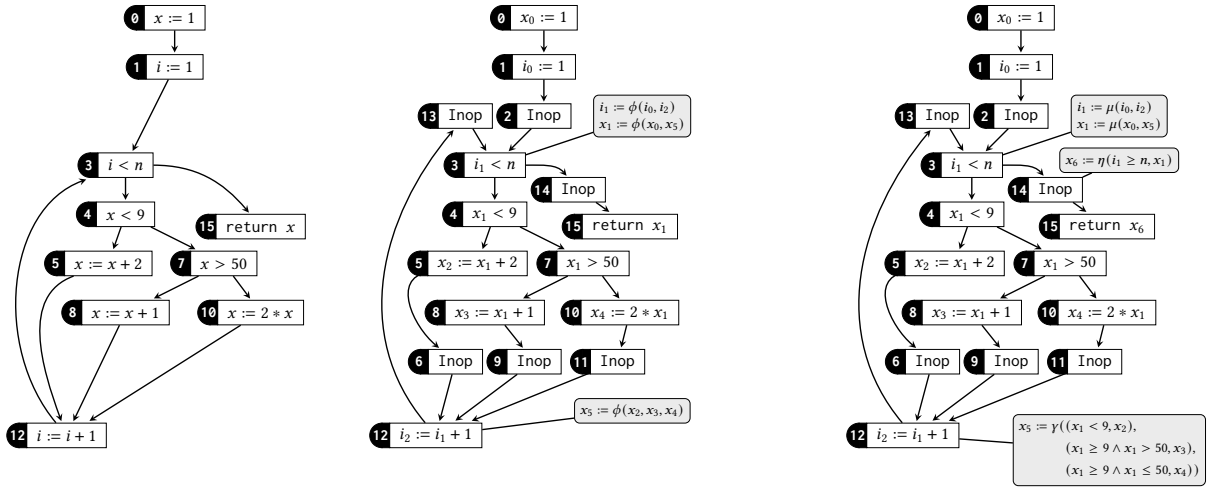
2 Motivating Example

We illustrate and explain informally the GSA form on a simple example program. Let us consider the following C code snippet, where function `f` takes as input an integer `n` and computes a result `x` using a for-loop.

```
int f(int n) {
  int x = 1;
  for (int i = 1; i < n; i++)
    if (x < 9) x = x + 2;
    else if (x > 50) x = x + 1;
    else x = 2 * x;
  return x;
}
```

In a compiler chain, programs are usually represented as a control-flow graph (CFG) of instructions, which simplifies their processing through analyses and optimisations. Fig. 2a illustrates this CFG representation, called RTL in CompCert.

In SSA (Fig. 2b), each variable is defined exactly once: this makes the link between the program point where a variable is defined and the program point where it is used explicit in the syntax. SSA extends RTL with ϕ-instructions that handle control-flow joins. At node 12, the ϕ-instruction expresses



(a) RTL form **(b)** SSA form **(c)** GSA form

Figure 2. Example program illustrating Gated SSA compared to RTL and SSA.

that x_5 gets the value of either x_2, x_3 or x_4, depending on whether the control-flow of execution dynamically originates from node 6, 9, or 11 respectively. A ϕ-block (grey nodes in Fig. 2b) groups all the ϕ-instructions at a given junction node (e.g. node 3). In addition to that, the CFG is normalised in the following sense: (i) only `Inop` instructions, i.e. no-op instructions, can lead to a junction point, and (ii) all nodes are syntactically reachable from the entry node.

GSA (Fig. 2c) in turn extends SSA. In GSA, ϕ-instructions are replaced by μ- and γ-instructions, and new instructions called η-instructions are introduced. These instructions are meant to better reflect control-dependencies. Indeed, they include strictly more information than ϕ-instructions, and therefore allow for the formulation of different types of semantics, such as demand-driven or data-flow driven executions. An SSA ϕ-instruction becomes either a γ-instruction when it is located at a simple junction point (e.g. node 12), or a μ-instruction when it is located at a loop-header (e.g. node 3). The γ-instructions are augmented with gates, namely predicates characterising the control-flow path corresponding to the reaching definition of each of the ϕ-arguments. For γ-instructions, the selection of the argument is based on the predicate evaluating to true. For instance, at node 12, predicates guarding the γ-arguments represent paths from node 4 to node 12; they are mutually exclusive, and each time node 12 is reached through the program execution, only one of them is satisfied.

Regarding loops, GSA uses two kinds of instructions. At loop headers, GSA uses μ-instructions, to reflect the loop construct; they update variables modified in the loop body, thereby handling loop-carried dependencies properly. When loops have a single entry point and a single latch, μ-instructions are of the form $x = \mu(x_0, x_i)$, where x_0 is the initial value before entering the loop, and x_i is the version of x modified inside the loop body. Intuitively, arguments of a μ-instruction are not guarded like in γ-instructions, since no useful predicate helps to distinguish when x_0 or x_i is the right version to use. Indeed, the execution is always flowing from the loop-latch back to the header, and yet, guarding the selection of x_i with the True predicate wouldn't always be correct. At loop exits, GSA introduces η-instructions (e.g. node 14). An η-instruction $x = \eta(c, x_\mu)$ selects a loop-defined variable x_μ when the loop-exit condition c is satisfied. Intuitively, η-instructions introduce new variables to decouple loop-carried dependencies from variable uses occurring after the loop has ended.

Another folklore intuition about GSA is that the γ-, μ- and η-instructions make it referentially transparent: each of them denotes an equality between the left-hand side variable and their arguments, that only holds when the *path condition* expressed by the Boolean predicate is true.

We argue that the main challenge in understanding GSA lies in the definition, specification, and construction of gates. First, depending on the application use cases, these gates reflect different subsets of dependencies. It is currently an open problem to determine which dependencies must be kept track of to correctly reflect the semantics of the initial SSA program. Second, it is currently unclear in the recent literature which algorithms are precisely used to generate these gates. Seminal papers are cited, but without giving many details about the concrete implementations, let alone a concise specification thereof. We argue that gates are the most critical component of GSA, and that they must be better understood and mastered by the community. Another challenge is that gates must be interpreted as path predicates, reified in the very code of GSA programs. Hence, for gates to make proper sense, we need to define what it means to evaluate a gate. In fact, the situation is much more subtle than

one might first think: gates involve program variables that may be defined or not, depending on the path that is actually taken during the execution. This raises several challenges related to the underlying logic behind the apparently simple syntax of gates. These are the challenging and interesting problems that we tackle in this work.

To do this, we employ a compiler-correctness approach. Our goal is to devise a reference GSA form, and provide it with a formal semantics reflecting the control-flow insensitivity of the semantics of gates. In this context, GSA is correct when it reflects all the dependencies required to, provably, mimic the semantics of the initial SSA program.

3 Background on SSA in CompCertSSA

We integrate our formalisation work into the CompCertSSA compiler [4], which is an extension of the CompCert C compiler [20]. CompCert is programmed and verified using Coq [18]. The compiler itself is written as a sequence of 20 compiler passes, from the CompCert C source language down to assembly, going through 8 intermediate languages. Among them is RTL (Register Transfer Language), a CFG-based representation of programs, and on which most CompCert's optimisations are performed. Compiler passes are either programmed and verified in Coq using simulation techniques, or programmed in OCaml and verified in Coq, using translation validation. The correctness theorem states that the compilation is semantics-preserving. It decomposes into theorems for each of the 20 compilation passes.

CompCertSSA extends CompCert with an SSA-based optimising middle-end. The middle-end is plugged in at the level of RTL, and includes a validated SSA construction, SSA-based optimisations, as well as an SSA destruction phase, going back to RTL, on which register allocation takes place. The SSA form in CompCertSSA was designed to be as close as possible to the RTL form. In particular, it reuses all regular instructions, arithmetic and conditional operators.

This section introduces the syntax and semantics of SSA of CompCertSSA, and recalls some related background notions.

Notations. For option types, we write $\lfloor x \rfloor$ (read: "some x") for the presence of value x, and $\emptyset$ (read: "none") for the absence of value. We write $h :: t$ for a list with head h and tail t, and ϵ for the empty list. Vectors are written $\vec{v}$, and v_k denotes the k-th element of $\vec{v}$.

3.1 SSA Representation

An SSA function f (see Fig. 3) is modelled as a record made of a CFG $\mathcal{I}$ of instructions over pseudo-registers, and a map Φ for ϕ-instructions. $\mathcal{I}$ is modelled as a partial map from nodes to single instructions. The instruction set includes arithmetic operations, memory loads and stores, conditional and unconditional jumps, function calls, and a return statement. Each instruction explicitly carries the labels of successor instructions. In the following, l ranges over node labels, and r ranges over pseudo-registers. The `Inop` instruction is a no-op instruction, it just branches to its explicit successor l. The `Iop` instruction applies an operator o_a to a list of pseudo-registers $\vec{r}$, stores its result in a register r_d and branches to its successor node l. The `Icond` instruction conditionally branches to l_1 or l_2 depending on the value of the condition operator o_c applied to $\vec{r}$. The Φ map stores blocks of ϕ-instructions. We write $f.\Phi(l)$ for the ϕ-block at node l. A ϕ-instruction written $r_\text{d} \leftarrow \phi(\vec{r_i})$ assigns to r_d the value of one of its operands in $\vec{r_i}$.

$$
\begin{array}{rcll}
i & ::= & \texttt{Inop}(l) & \text{no-op instr.} \\
 & | & r_\text{d} \leftarrow \texttt{Iop}\ (o_\text{a}, \vec{r}, l) & \text{arith. operator } o_\text{a} \text{ on } \vec{r} \\
 & | & \texttt{Icond}(o_\text{c}, \vec{r}, l_1, l_2) & \text{conditional } o_\text{c} \text{ on } \vec{r} \\
 & | & \ldots & \text{other RTL instructions} \\
\mathcal{I} & ::= & l \mapsto i & \text{instruction map} \\
i_\phi & ::= & r_\text{d} \leftarrow \phi(\vec{r}) & \phi\text{-instruction} \\
\Phi & ::= & l \mapsto \vec{i_\phi} & \phi\text{-block map} \\
f & ::= & (\mathcal{I}, \Phi) & \text{SSA function}
\end{array}
$$

Figure 3. Syntax of SSA.

$$
\frac{f.\mathcal{I}(l) = \lfloor \texttt{Inop}(l') \rfloor \qquad f \not\vee l'}{\vdash \mathcal{S}(f, l, rs, m) \xrightarrow{\epsilon} \mathcal{S}(f, l', rs, m)}
$$

$$
\frac{f.\mathcal{I}(l) = \lfloor \texttt{Inop}(l') \rfloor \qquad f \vee l' \qquad f.\Phi(l') = \lfloor b_\phi \rfloor \qquad \text{preds}(l')_k = l \qquad b_\phi, k \vdash rs \overset{\phi}{\rightsquigarrow} rs'}{\vdash \mathcal{S}(f, l, rs, m) \xrightarrow{\epsilon} \mathcal{S}(f, l', rs', m)}
$$

Figure 4. Semantics of SSA (excerpt).

In SSA, there is a clear distinction between RTL-like instructions and ϕ-instructions: both are stored in distinct maps; this simplifies the conversion to SSA, and allows for a smooth integration in CompCert. All ϕ-blocks, are located at junction points, and the CFG is normalised so that only `Inop` instructions can lead to a junction point. Last, SSA functions are equipped with a well-formedness predicate, wfSSA(.), capturing essential properties of SSA functions: structural constraints, unique definitions and strictness properties [4].

The placement of ϕ-instructions in SSA relies on the notion of dominance between nodes [10]. A node i dominates a node j (written $i \geq j$) if every path from the entry node to j goes through i. Node i strictly dominates j (written $i > j$) when $i \geq j$ but $i \neq j$. We write $i \not\geq j$ to mean that i does not dominate j. The (unique) immediate dominator of a node j is the strict dominator of j that does not strictly dominate any other strict dominator of j. In Fig. 2b, we have $1 > 3$, $5 \not\geq 12$ and node 4 is the immediate dominator of node 12.

3.2 Semantics of SSA

The small-step semantics of SSA is defined as a relation $G \vdash S \xrightarrow{t} S'$ between a global environment G and states S and S', associating to a program loaded in G the set of all its observable behaviours, comprising a trace t of external actions, emitted by certain specific instructions such as external function calls. Observable behaviours include terminating, diverging and going wrong [21]. In this paper, to simplify the presentation, and we omit G. For the sake of completeness, we keep the trace in small-step execution steps, although the instructions we present in this paper do not emit observable events, and thus step with the empty trace ϵ.

Semantic states are written $\mathcal{S}(f, l, rs, m)$. They carry the current function f, a program counter l, a map rs from pseudo-registers to values, and a memory state m. Other semantic states include call states and return states. For the sake of presentation, in this paper, we focus on regular states, as we only expose the intra-procedural part of SSA.

Figure 4 gives the most relevant semantic rules for SSA. The rules for instructions other than `Inop` closely match their RTL counter-part from CompCert, we thus omit them for space reasons. Indeed, the most important rules are the one for the `Inop` instruction, because ϕ-blocks can only be placed at junction points in the CFG, and only `Inop` can lead to a junction point. The first rule of Fig. 4 gives the rule for executing an `Inop` instruction at program point l, when its successor l' is not a junction point (i.e. $f \not\vee l'$). Here, the standard RTL rule applies, as there is no ϕ-block to execute, and execution steps to l' without modifying the registers state rs. The second rule corresponds to the case where l' is a junction point (i.e. $f \vee l'$). In that case, a (potentially empty) ϕ-block b_ϕ exists, and is executed before reaching the regular instruction at l'. All its ϕ-instructions are executed in parallel and their result is assigned to their respective destination registers ($b_\phi, k \vdash rs \overset{\phi}{\rightsquigarrow} rs'$). A given ϕ-instruction uses only one of its operands, the k-th, where k is determined by a conventional numbering of the predecessors of each of the CFG nodes ($\mathsf{preds}(l')_k = l$). The execution updates the registers state to rs', so that the value of each register assigned in b_ϕ becomes the value of the k-th operand of the corresponding ϕ-instruction in rs.

4 Gated SSA Representation

To define our Gated SSA form, we take inspiration from previous existing work on GSA [3, 9, 12, 13, 16, 23, 28, 29]. Precisely, we define GSA as an extension of the SSA form presented in the previous section. The syntax for GSA therefore follows the syntax for SSA closely, and many SSA concepts translate to GSA.

$$
\begin{array}{rcll}
i & ::= & \texttt{Inop}(l) & \text{no-op instr.} \\
 & | & r_\mathrm{d} \leftarrow \texttt{Iop}\ (o_\mathrm{a}, \vec{r}, l) & \text{arith. op. } o_\mathrm{a} \text{ on } \vec{r} \\
 & | & \texttt{Icond}(o_\mathrm{c}, \vec{r}, l_1, l_2) & \text{conditional } o_\mathrm{c} \text{ on } \vec{r} \\
 & | & \dots & \\
\mathcal{I} & ::= & l \mapsto i & \text{regular instr. map} \\
 & & & \\
i_\mathcal{M} & ::= & r_\mathrm{d} \leftarrow \mu(r_0, r_i) & \text{merge instr.} \\
 & | & r_\mathrm{d} \leftarrow \gamma(\overrightarrow{(p_i, r_i)}) & \\
\mathcal{M} & ::= & l \mapsto \overrightarrow{i_\mathcal{M}} & \text{merge-block map} \\
 & & & \\
i_\eta & ::= & r_\mathrm{d} \leftarrow \eta(p, r_\mathrm{s}) & \text{selection instr.} \\
\mathcal{E} & ::= & l \mapsto \overrightarrow{i_\eta} & \text{selection-block map} \\
 & & & \\
f & ::= & (\mathcal{I}, \mathcal{M}, \mathcal{E}) & \text{GSA function} \\
 & & & \\
c & ::= & (o_\mathrm{c}, \vec{r}) & \text{cond. op. } o_\mathrm{c} \text{ on } \vec{r} \\
p & ::= & \mathsf{False} \mid \mathsf{True} \mid \mathsf{Undef} & \text{predicates} \\
 & | & c \mid \overline{c} \mid p_1 \vee p_2 \mid p_1 \wedge p_2 &
\end{array}
$$

Figure 5. Syntax of GSA.

4.1 Syntax of GSA

The definition of a function is shown in Fig. 5. It includes three maps, a map $\mathcal{I}$ for the CFG over regular instructions (as in SSA), a map $\mathcal{M}$ for merge instructions, i.e. γ- and μ-instructions, and a map $\mathcal{E}$ for η-instructions. This separation makes it a natural extension of SSA, because only the map $\mathcal{E}$ has to be newly constructed in GSA. The map $\mathcal{M}$ has the same structure as Φ in SSA: ϕ-instructions have either been converted to either μ- or γ-instructions. Each of the $\mathcal{M}$ and $\mathcal{E}$ maps, which contain the new GSA instructions, are maps from program counters to blocks of GSA instructions.

We distinguish two categories of GSA instructions. First, *merge instructions* include μ- and γ-instructions, and replace the SSA ϕ-instructions. μ-instructions, written $r_\mathrm{d} \leftarrow \mu(r_0, r_i)$, are the simplest ones; they are placed at loop headers, and r_d is the loop-carried register. When the loop is initially reached, this instruction assigns the value of r_0 to r_d. When the loop is subsequently re-entered from its back-edge, the instruction assigns the value of r_i to r_d. This instruction does not have predicates guarding its arguments, and therefore behaves just like its ϕ-instruction counterpart, based on the executed control-flow. γ-instructions, written as $r_\mathrm{d} \leftarrow \gamma(\overrightarrow{(p_i, r_i)})$, are the other replacement for ϕ-instructions. Here, instead of having to rely on control-flow to select a register to assign to r_d, it includes, for each register r_i, a predicate p_i indicating when that register should be selected. Consider for instance the instruction $x_5 = \phi(x_2, x_3, x_4)$ at node 12 in Fig. 2b, and its corresponding γ-instruction at node 12 in Fig. 2c. Each predicate describes a path from dominator node 4 of node 12 to node 12. This is enough to discriminate paths with, respectively, x_2, x_3, or x_4 as a reaching definition for x_5.

$$\frac{}{rs \models_{\mathrm{p}} \mathsf{True} \Downarrow 1} \qquad \frac{}{rs \models_{\mathrm{p}} \mathsf{False} \Downarrow 0} \qquad \frac{}{rs \models_{\mathrm{p}} \mathsf{Undef} \Downarrow \frac{1}{2}}$$

$$\frac{rs \models_{\mathrm{c}} c \Downarrow b}{rs \models_{\mathrm{p}} c \Downarrow b} \qquad \frac{rs \models_{\mathrm{p}} p_1 \Downarrow b_1 \qquad rs \models_{\mathrm{p}} p_2 \Downarrow b_2}{rs \models_{\mathrm{p}} p_1 \vee p_2 \Downarrow b_1 \max b_2}$$

$$\frac{rs \models_{\mathrm{c}} c \Downarrow b}{rs \models_{\mathrm{p}} \overline{c} \Downarrow 1 - b} \qquad \frac{rs \models_{\mathrm{p}} p_1 \Downarrow b_1 \qquad rs \models_{\mathrm{p}} p_2 \Downarrow b_2}{rs \models_{\mathrm{p}} p_1 \wedge p_2 \Downarrow b_1 \min b_2}$$

Figure 6. Evaluation of GSA predicates.

Second, *selection instructions* are extra instructions introduced when generating GSA and have no counter-part in SSA. All η-instructions, written $r_{\mathrm{d}} \leftarrow \eta(p, r_{\mathrm{s}})$, are placed at loop exit nodes. These instructions indicate a termination condition p of the loop, which asserts that the register r_{s} is ready to use, and can be assigned to r_{d}. Intuitively, they behave like predicated moves. We assume that the loops are in loop-closed SSA form,[1] so that the register r_{s} is assigned by a corresponding μ-instruction at the loop header.

Predicates play an essential role in the semantics of η- and γ-instructions. In GSA, predicates are represented in the IR. They are defined according to the grammar given at the end of Fig. 5. We recall that in GSA, predicates are used to materialise path conditions under which a γ or η argument should be selected. Predicates are thus built out of atomic conditions c found in the initial SSA code, as well as their negation $\overline{c}$. Atomic conditions are pairs of a conditional operator o_{c} and operands $\vec{r}$. To reflect path condition composition, we need to include the conjunction $\wedge$ of predicates, for nested conditionals, and the disjunction $\vee$ of predicates, for sequencing conditionals. Predicates also include the two expected constants True and False, and the special constant Undef, representing a non-evaluable condition.

We do not need to add a negation operator on predicates, as instructions in SSA do not contain them either: branching is only possible through simple, basic conditions, rather than arbitrarily complex Boolean expressions (see Fig. 3). This syntax for predicates is also sufficient to treat switch branches, modulo an encoding of branching conditions.

4.2 Semantics of GSA

The semantics of GSA is essentially the same as the one of SSA, since both representations use the same set of regular instructions, and handle function calls similarly. The novelty is how to execute GSA specific instructions. At a high-level, whenever a γ-, μ- or η-block is reached, all its instructions are executed in parallel and their results are assigned to their respective destination registers. This is similar to how ϕ-blocks are executed.

[1] Loop-closed SSA form states that all variables defined within the loop are not used outside of the loop.

To define the semantics of γ- and η-instructions, we first need to define the semantics for GSA predicates. The evaluation of predicates $\models_{\mathrm{p}}$ is defined by induction on the structure of the predicate (see Fig. 6, where $\models_{\mathrm{c}}$ refers to the evaluation of conditions borrowed from SSA). We emphasise, however, that predicates are part of the syntax, and composed of (potentially negated) conditions that refer to program variables. Hence, we need a local environment rs to evaluate a predicate. Importantly, we also note that a predicate can either (i) evaluate to the Boolean values 0 and 1 or (ii) be *non-evaluable* (represented by an evaluation to $\frac{1}{2}$ in Fig. 6) since it can refer to program variables that may have never been in scope. This subtlety is in contrast with a simple Boolean semantics where predicates evaluate to either true or false.

The semantics of merge instructions is handled when executing `Inop`(l') instructions, as for the ϕ-instructions in SSA (see bottom of Fig. 7). It is also the case for selection instructions, which are inserted at loop exit landing pads implemented with `Inop` instructions. Note that a loop exit landing pad could very well be the predecessor of a junction point: in that case, we must handle both kinds of instructions.

The most important semantic rules for GSA are given in Fig. 7, where GSA semantic states are written $\mathcal{T}(f, l, rs, m)$. We again define two cases, whether l' is a junction point or not, as merge-blocks are only placed at junction points. If l' is not a junction point (rule NJoin), the register state is only updated by η-instructions. Otherwise (rule Join), l' is a junction point, and the merge-block must be executed. The η-block is first executed, updating rs to rs'; then the merge-block is executed, updating rs' to rs''.

Let us now explain in deeper detail how to execute η-instructions and merge instructions. Rule Eta (top of Fig. 7) defines how to execute a non-empty list of η-instructions. Executing an η-instruction $r_{\mathrm{d}} \leftarrow \eta(q, r)$ requires predicate q to evaluate to 1: it must hold in rs. All η-instructions are evaluated in parallel: the value of the predicates and register operands are determined in the current register state rs, and destination registers are updated.

The execution of a merge-block is similar to the execution of a ϕ-block: it assigns in parallel to destination registers the value (in rs') of one operand of each merge-instruction. The novelty here is *how* this operand is chosen for each instruction. For μ-instructions (rule Merge$_\mu$), the selection is done only via k, designating the index of the control-flow predecessor that leads to l', as was done for ϕ-instruction. In the case of a μ-instruction, we however ensure that it only has two predecessors: the loop header, and the loop latch. Hence, $k \in \{0, 1\}$ and either r_0 or r_1 is assigned to r_{d}.

In the case of γ-instructions (rule Merge$_\gamma$), the selection is guided by a predicate that evaluates to 1, and k does not play any role. Indeed, in rule Merge$_\gamma$, the n-th operand r_n is selected, for *some* n. In particular, q_n is not necessarily q_k. We ensure that our semantics stays deterministic by choosing the first such n, whenever two predicates q_{n_1} and q_{n_2}

$$\textsc{Eta}_{\emptyset}\ \frac{}{\emptyset \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs} \qquad \textsc{Eta}_{\epsilon}\ \frac{}{\lfloor \epsilon \rfloor \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs}$$

$$\textsc{Eta}\ \frac{i = r_{\mathrm{d}} \leftarrow \eta(q, r) \qquad rs \models_{\mathrm{p}} q \Downarrow 1 \qquad \lfloor b_\eta \rfloor \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs'}{\lfloor i :: b_\eta \rfloor \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs'[r_{\mathrm{d}} \mapsto rs(r)]}$$

$$\textsc{Merge}_{\gamma}\ \frac{i = r_{\mathrm{d}} \leftarrow \gamma(\overrightarrow{(q, r)}) \qquad rs \models_{\mathrm{p}} q_n \Downarrow 1 \qquad \lfloor b_{\mathcal{M}} \rfloor, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'}{\lfloor i :: b_{\mathcal{M}} \rfloor, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'[r_{\mathrm{d}} \mapsto rs(r_n)]}$$

$$\textsc{Merge}_{\mu}\ \frac{i = r_{\mathrm{d}} \leftarrow \mu(r_0, r_1) \qquad k \in \{0, 1\} \qquad \lfloor b_{\mathcal{M}} \rfloor, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'}{\lfloor i :: b_{\mathcal{M}} \rfloor, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'[r_{\mathrm{d}} \mapsto rs(r_k)]}$$

$$\textsc{NJoin}\ \frac{f.\mathcal{I}(l) = \lfloor \mathtt{Inop}(l') \rfloor \qquad f \not\gg l' \qquad f.\mathcal{E}(l) \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs'}{\vdash \mathcal{T}(f, l, rs, m) \xrightarrow{\epsilon} \mathcal{T}(f, l', rs', m)}$$

$$\textsc{Join}\ \frac{f.\mathcal{I}(l) = \lfloor \mathtt{Inop}(l') \rfloor \qquad f \gg l' \qquad f.\mathcal{E}(l) \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs' \qquad \mathrm{preds}(l')_k = l \qquad f.\mathcal{M}(l'), k \vdash rs' \overset{\mathcal{M}}{\rightsquigarrow} rs''}{\vdash \mathcal{T}(f, l, rs, m) \xrightarrow{\epsilon} \mathcal{T}(f, l', rs'', m)}$$

Figure 7. Semantics of GSA (excerpt)

would be simultaneously true. However, as we explain in Section 5.2, predicates guarding γ-arguments are provably mutually exclusive in GSA.

5 Conversion from SSA to GSA

To convert SSA into GSA, the ϕ-instructions need to be replaced by γ- or μ-instructions, depending on whether the ϕ-instruction was at a simple junction point or at a loop header. In addition to that, extra η-instructions need to be added to loop exits. The difficulty of the transformation is the calculation of the predicates for γ- and η-instructions, as well as proving the necessary properties about the generated predicates. This is the main focus of this paper, and we explain it below, together with the salient properties of the conversion. The new η-instructions additionally require fresh register names, and to update register uses accordingly. We abstract over the related administrative duties, which mainly comprise technicalities. Full details are present in the companion Coq development [17].

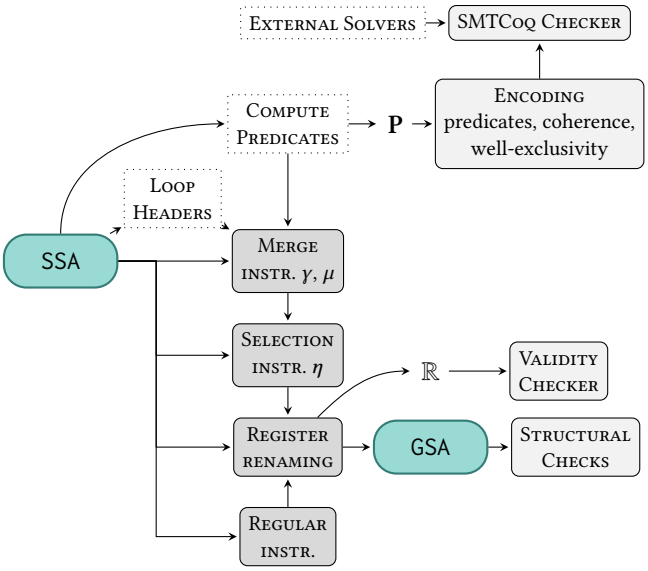


Figure 8. Overview of the translation from SSA to GSA.

Section 5.1 describes the main algorithm used to perform the translation from SSA to GSA, as well as its formal specification. In Section 5.2, we give more details about the generation of predicates, explain predicates invariants that hold in the specification. In Section 5.3, we explain how we specify the translation from SSA to GSA at the instruction level. Section 5.4 then goes over how the renaming of variables is performed after the new η-instructions have been inserted, and describes invariants about this renaming. Section 5.5 then covers the main correctness theorem and gives an intuitive description of how it is proven using the specification. Finally, Section 5.6 describes the main semantic invariant used to prove the correctness theorem of the translation.

5.1 Specification of GSA Construction

This section describes the translation going from SSA to GSA. A diagram with an overview of the translation is given in Fig. 8, where the main steps are in the center of the figure: (i) generate γ- and μ-instruction from ϕ-instructions, (ii) insert an η-instruction for each μ-instruction, and (iii) register renaming for the inserted η-instruction assignments. In addition, there are also a predicate computation step and a loop headers computation step which generate information used when generating merge- and η-instructions.

From an implementation point of view, the translation is done sequentially, meaning there will be an intermediate state of the code after the merge-instruction translation and after the η-instruction generation, where the GSA code does not account for the proper renaming yet. When writing a specification for this translation, one would want to relate the SSA ϕ-block directly to the final merge-block, but without having to consider how η-instructions are inserted, and

thus how the renaming is performed. We therefore generate a predicate matrix $\mathbf{P}$ containing all the predicates that were generated during the translation and a renaming map $\mathbb{R}$ which contains all the variables that are renamed and their new name, together with their original definition point. We described in further detail the predicate matrix and the renaming map in described in Section 5.4 and Section 5.2 respectively. Finally, the properties we need about $\mathbf{P}$, $\mathbb{R}$ and the CFG structure are validated after the translation, with dedicated validators. This includes making use of an untrusted external SMT solver, whose result is itself validated by SMTCoq [19]. We explain our encoding in Section 6.1.

Translation Specification. We now give in Definition 5.1 a formal specification for our translation from SSA to GSA. It relates an SSA function f to the translated GSA function tf, relative to $\mathbf{P}$ and $\mathbb{R}$. It is written $\mathbf{P}, \mathbb{R} \vdash f \equiv tf$.

Definition 5.1 (SSA to GSA Translation Specification)**.**

$$\frac{\begin{array}{c} f = (\mathcal{I}, \Phi) \qquad tf = (\mathcal{I}_{tr}, \mathcal{M}_{tr}, \mathcal{E}_{tr}) \\ \mathbf{P}\ \mathsf{coh} \qquad \mathbf{P} \ltimes \qquad f, tf \models \mathbb{R} \checkmark \\ \forall l, \mathcal{I} \simeq^{l}_{\iota} \mathcal{I}_t \qquad \forall l, \mathbf{P} \vdash \Phi \simeq^{l}_{\mathcal{M}} \mathcal{M}_t \qquad \forall l, \mathbf{P} \vdash \mathcal{I} \simeq^{l}_{\eta} \mathcal{E}_t \\ \mathsf{rename}(\mathbb{R}, \mathcal{I}_t, \mathcal{M}_t, \mathcal{E}_t) = (\mathcal{I}_{tr}, \mathcal{M}_{tr}, \mathcal{E}_{tr}) \end{array}}{\mathbf{P}, \mathbb{R} \vdash f \equiv tf}$$

We explain here the main components of the specification, and how it is structured, leaving a detailed and formal description for the next subsections.

The two functions are of the form $f = (\mathcal{I}, \Phi)$ and $tf = (\mathcal{I}_{tr}, \mathcal{M}_{tr}, \mathcal{E}_{tr})$. In the second line of the specification, we require the predicate matrix $\mathbf{P}$ and the renaming map $\mathbb{R}$ to satisfy properties that tell us enough about their correctness so that we can prove that the right argument will be picked in γ-instructions, and that the predicates in η-instructions will always evaluate to true when they are reached. We explain these requirements in Section 5.2 and Section 5.4.

The third line specifies how the code of the SSA and GSA functions match. We introduce one code-matching relation per type of instructions (regular instructions, merge-instruction, and selection-instructions). We explain these code correspondences in Section 5.3.

Finally, the renaming is performed after the threes kind of maps are generated, so that variables introduced by an η-instruction are used after their definition.

5.2 Specification of GSA Predicates

In GSA, predicates guard the register selection in γ- and η-instructions. Hence, they should reflect the dynamic control-flow of the program. Essentially, one must generate a predicate for each path to the γ-instruction, so that the predicate is true if and only if the path was picked.

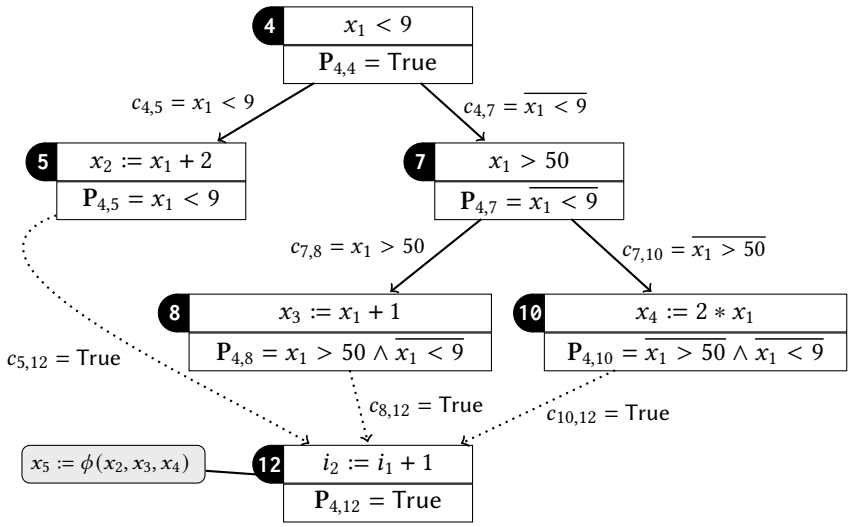
One solution to this is to leverage the solution to the single-source path problem expressed by Tarjan [26]. The suggested algorithm is use to build a regular expression on CFG edges, that matches all possible paths between a single source CFG node and all other CFG nodes. We then translate these regular expressions to predicates by collecting and composing all relevant conditions encountered on the paths of interest. Such predicates will, intuitively, only be true whenever a path in the language of that regular expression is taken. Kleene stars in regular expression express infinitely many possible paths with loops. Simply removing star-expressions during the translation of path expressions to predicates provably does not change the evaluation of the resulting predicate itself: the predicate characterising paths already accounts for cases where loops are not entered, hence entailing all paths described by the star-expression-free predicate.

To reason about the meaning of predicates in the proof of the semantic preservation, the solution to the single source path expression problem needs to be formalised. Presenting the details of Tarjan's algorithm is far beyond the scope of this paper. In fact, in this work, instead of verifying Tarjan's algorithm, we use a translation validation approach: we identify two properties, namely *coherence* and *well-exclusivity* on predicates, which are sufficient to prove the translation itself, allowing us to abstract away from the implementation technicalities of Tarjan's algorithm. We explain later in Section 6 how we validate these properties.

We first calculate predicates on the initial SSA function; then, we insert them in the GSA function, and the subsequent renaming of variables will account for the insertion of η-instructions. We note that predicates characterise sets of CFG paths: they hence constitute an information that is *global* on the CFG of the initial SSA function. We therefore express their essential properties relative to a *predicate matrix* $\mathbf{P}$, associating predicates, in $\mathbb{P}$, to pairs of nodes in the CFG of the initial SSA function f. Morally, the predicate associated to (i, j), written $\mathbf{P}_{i,j}$, should represent a set of paths from node i to node j in the CFG of f. For instance, for the CFG in Fig. 2c, $P_{4,8}$ is $x_1 > 50 \wedge \overline{x_1 < 9}$.

Coherence. The first property we formulate, *coherence*, relates to the semantic correctness of predicates. Intuitively, predicates should indeed be coherent with the CFG paths they are supposed to represent. In particular, for a given node j with m predecessors, the possible paths from a node i to each of the predecessors of j should enable a path from i to j, when extended with the (atomic) condition on the edge from that predecessor to j. This is visualised in Fig. 9, taken from Fig. 2b, but with elided `Inop` nodes. In this figure, we take node i to be node 4, and we write each predicate $\mathbf{P}_{4,k}$ at node k. An edge from node k to j is taken when the atomic condition $c_{k,j}$ holds. The coherence property intuitively states that, $\mathbf{P}_{i,j}$ ought to hold, as soon as one of the paths from i to j, corresponding to a predicate $\mathbf{P}_{i,k} \wedge c_{k,j}$, where k is a predecessor of j, has been taken.

We formalise this intuition with the three-place relation $\mathbf{P}$ coh (i, j); it states, for a function f, a predicate matrix



(a) SSA example program taken from Fig. 2, with elided `Inop` nodes (excerpt).

$$\begin{pmatrix} (\mathbf{P}_{4,5} \wedge c_{5,12}) \\ \vee(\mathbf{P}_{4,8} \wedge c_{8,12}) \\ \vee(\mathbf{P}_{4,10} \wedge c_{10,12}) \end{pmatrix}\downarrow_{12} \implies \mathbf{P}_{4,12}$$

$$= \begin{pmatrix} (x_1 < 9 \wedge \text{True}) \\ \vee(x_1 > 50 \wedge \overline{x_1 < 9} \wedge \text{True}) \\ \vee(\overline{x_1 > 50} \wedge \overline{x_1 < 9} \wedge \text{True}) \end{pmatrix} \implies \text{True}$$

(b) Example of local coherence $\mathbf{P}$ coh $(4, 12)$. The projection has no effect as only i_2 and x_5 are defined at node 12, but they do not appear in the predicates.

$$(\mathbf{P}_{4,7} \wedge c_{7,8})\downarrow_8 \implies \mathbf{P}_{4,8}$$

$$= \overline{x_1 < 9} \wedge x_1 > 50 \implies x_1 > 50 \wedge \overline{x_1 < 9}$$

(c) Example of local coherence $\mathbf{P}$ coh $(4, 8)$.

Figure 9. Illustration of the coherence property $\mathbf{P}$ coh (i, j) for a node i such that $i \succ j$.

$\mathbf{P}$, and two nodes i and j, that the predicate $\mathbf{P}_{i,j}$ is, locally, coherent for nodes i and j.

Definition 5.2 (Local Coherence)**.** Let f be an SSA function, $\mathbf{P}$ a predicate matrix, and i and j be two nodes in the CFG of f. The relation $\mathbf{P}$ coh (i, j) is defined as follows[2]:

$$\frac{i \not\succeq j}{\mathbf{P} \text{ coh } (i,j)}\ \textsc{CohNDom} \qquad \frac{isTrue(\mathbf{P}_{i,i})}{\mathbf{P} \text{ coh } (i,i)}\ \textsc{CohEq}$$

$$\frac{i \succ j \qquad \left(\bigvee_{k \in \text{preds}(j)} \mathbf{P}_{i,k} \wedge c_{k,j}\right)\downarrow_j \implies \mathbf{P}_{i,j}}{\mathbf{P} \text{ coh } (i,j)}\ \textsc{CohSDom}$$

In addition to the informal explanations we gave previously, coherence requires two other ingredients, appearing in Definition 5.2, to be meaningful and provable.

The first ingredient is dominance. For nodes i and j, we need to distinguish cases where i dominates j or not. Indeed, recall that a predicate used at a node j characterises paths from a dominator of j to j. Hence, we never need to consider paths from non-dominators of j to j. This explains the first rule CohNDom, which does not impose any constraint on those predicates. Now, when i dominates j, there are again two cases to consider: either $i = j$ or i strictly dominates j. In the first case (rule CohEq), we ask that predicate $\mathbf{P}_{i,i}$ always evaluates to true, so that it models an empty path from i to i. The second case (rule CohSDom) corresponds to the informal explanation given previously. We give two examples of this rule, illustrated in Fig. 9c (where $i = 4$ and $j = 12$) and Fig. 9b (where $i = 4$ and $j = 8$).

The second ingredient we need is related to the interaction between semantic implication of predicates and the evaluability of the predicates. Indeed, we need to make sure that semantic implication does not hold vacuously because of some yet-to-be-defined or outdated program variable appearing in the atomic conditions of the predicates in $\mathbf{P}$. Predicates sometimes reflect CFG paths that join in a non-structured way: some sub-paths might therefore involve conditions on locally defined program variables, and the definition points of variables appearing in predicates do not necessarily dominate the use point of a predicate.

To deal with this, we introduce a projection operator on predicates, written $p\downarrow_j$, which replaces any atomic condition c or $\overline{c}$ in a predicate p with True, as soon as condition c uses a program variable defined at node j in function f. In particular, when j is a junction point, variables defined by a ϕ-instruction are abstracted away. Note that we only project atomic conditions, and not the entire predicate. We only abstract the variables that actually *need* to be abstracted. Intuitively, the projection operator in Definition 5.2 allows us to specify which variables should be considered as relevant to the truthfulness of predicate $\mathbf{P}_{i,j}$; indeed, to prove coherence one would have to show that the projection does not change the truthfulness of the predicate, either by showing that the problematic variables are not present in the predicate, or by showing that they do not affect its evaluation.

Next, the following definition uses the coherence relation to express a global criteria on the entire predicate matrix.

[2]The definition of semantic implication on predicates is standard: $P \implies Q$ means that, for any register state rs, if P evaluates to 1, then so does Q.

Definition 5.3 (Coherent Predicate Matrix)**.** Let f be an SSA function. A predicate matrix $\mathbf{P}$ is said to be coherent, written $\mathbf{P}$ coh, when $\mathbf{P}$ coh (i, j) for all i and j in $\mathbf{P}$.

We emphasise here that the matrix does not need to include all pairs of nodes. In practice, it is sufficient to keep track of only the predicates required in the future GSA function, i.e., at future γ- and η-instruction nodes (e.g. the predicates of Fig. 9a). Hence, it is enough to build a matrix of dimension $(D + H) \times N$, where N is the size of the CFG, D is the number of (non-loop headers) nodes holding a ϕ-instruction and H the number of loop exit nodes. Informally, for each of the $D + H$ nodes, one predicate is required to describe paths from their immediate dominator to each of the N nodes in the CFG of the function.

Mutual exclusivity. The second property we must establish about the generated predicates is that they are sufficiently informative: they indeed allow for a proper selection of the arguments in γ-instructions. We formalise this property using a notion of mutual exclusivity of predicates, stating that they cannot be satisfied simultaneously.

Definition 5.4 (Mutually Exclusive Predicates)**.** Let p_1 and p_2 be two predicates in $\mathbb{P}$. They are said to be mutually exclusive, written $p_1 \ltimes p_2$, whenever for all register states rs they cannot both evaluate to true, i.e. if $rs \models_{\mathrm{p}} p_1 \Downarrow 1$, then $rs \models_{\mathrm{p}} p_2 \not\Downarrow 1$.

Naturally, we cannot ask for all predicates in a predicate matrix to be pairwise mutually exclusive. What we require is that predicates to be used for the selection of any future γ-instruction's pair of arguments be mutually exclusive. Hence, we only consider non-loop-headers junction points – loop-header ϕ-instructions are future μ-instructions, that do not resort on predicates.

Definition 5.5 (Well-Exclusive Predicate Matrix)**.** Let f be an SSA function. A predicate matrix $\mathbf{P}$ is *well-exclusive* for f, written $\mathbf{P} \ltimes$, when for all node n_ϕ in f that is not a loop header, and that holds a ϕ-block, and any possible strict-dominator d of n_ϕ, i.e. $d \succ n_\phi$, the following holds: for any two distinct nodes $n_1, n_2 \in \mathrm{preds}(n_\phi)$, we have $\mathbf{P}_{d,n_1} \ltimes \mathbf{P}_{d,n_2}$.

We now summarise the calculation and validation of GSA predicates. For any ϕ-instruction at node n in the initial SSA function f, we calculate a predicate characterising all paths from its immediate dominator node d to n. For each loop exit node n in f, we calculate a path predicate characterising all paths from the corresponding loop-header of n to n. We collect all these predicates in the predicate matrix $\mathbf{P}$, on which we globally apply the projection operator $.\downarrow_.$ on all predicates columns, i.e. $\mathbf{P}_{i,j}$ is replaced by the projection $\mathbf{P}_{i,j}\downarrow_j$. We finally check that the resulting predicate matrix is indeed coherent and well-exclusive. The details of the validator itself are given in Section 6.

5.3 Specification for GSA Instructions

We turn now to the specification of how SSA instructions are converted to GSA instructions. We do this by stating how the respective instruction maps of the initial SSA function f and the GSA function are relating, on a per-node basis. Indeed, such a one-to-one correspondence is possible, since the conversion to GSA does not modify the structure of the CFG. In fact, we insert loop exit landing pads prior to the GSA conversion; the insertion of η-instructions hence also preserves the CFG structure.

We thus introduce three code-correspondences: relation $. \simeq^{\cdot}_{\iota} .$ handles the regular instruction maps, relation $. \vdash . \simeq^{\cdot}_{\mathcal{M}} .$ handles the merge-block map, and relation $. \vdash . \simeq^{\cdot}_{\eta} .$ handles the selection-block map.

Definition 5.6 (Code-Correspondence Relations at Node l)**.**

$$\frac{\mathcal{I}(l) = \mathcal{I}_t(l)}{\mathcal{I} \simeq^{l}_{\iota} \mathcal{I}_t} \qquad \frac{d \succ l \qquad \forall i.\ \mathbf{P} \vdash \Phi(l)_i \approx^{l,d}_{\phi} \mathcal{M}(l)_i}{\mathbf{P} \vdash \Phi \simeq^{l}_{\mathcal{M}} \mathcal{M}}$$

$$\frac{}{\mathbf{P} \vdash r_{\mathrm{d}} \leftarrow \phi(r_0, r_i) \approx^{l,n}_{\phi} r_{\mathrm{d}} \leftarrow \mu(r_0, r_i)}$$

$$\frac{\forall k.\ r'_k = (\mathbf{P}_{n,\mathrm{preds}(l)_k}, r_k)}{\mathbf{P} \vdash r_{\mathrm{d}} \leftarrow \phi(\overrightarrow{r}) \approx^{l,n}_{\phi} r_{\mathrm{d}} \leftarrow \gamma(\overrightarrow{r'})}$$

$$\frac{\mathcal{I}(l) = \lfloor \mathtt{Inop}(l') \rfloor \qquad h \succ l \qquad \forall\ r_{\mathrm{d}}\ p_{\mathrm{s}}\ r_{\mathrm{s}}.\ r_{\mathrm{d}} \leftarrow \eta(p_{\mathrm{s}}, r_{\mathrm{s}}) \in \mathcal{E}(l) \Rightarrow p_{\mathrm{s}} = \mathbf{P}_{h,l}}{\mathbf{P} \vdash \mathcal{I} \simeq^{l}_{\eta} \mathcal{E}}$$

Relation $\simeq_\iota$ is straightforward: both functions should have identical instructions at a node l. Relation $\simeq_{\mathcal{M}}$ states that, at node l, all ϕ-instructions and merge-instructions are pairwise related through $\approx_\phi$. Recall that ϕ-instructions are converted either to μ- or γ-instructions, depending on whether they were placed at loop headers in f. In our specification, we do not need to distinguish between the two cases, and we allow for a ϕ-instruction being related either to a γ-instruction or to a μ-instruction (if it has only two arguments: a first one for the loop initialisation and a second one for the loop iteration). We make this (correct) specification permissive enough so that it makes it possible to abstract over the correctness of the calculation of loop-headers. The interesting case is when a ϕ-instruction is converted to a γ-instruction at node l: to each register argument r_k in the ϕ-instruction, we associate in the γ-instruction the predicate $\mathbf{P}_{n,\mathrm{preds}(l)_k}$: this expresses that r_k should be selected on paths from n to the k-th predecessor of l, with n a strict dominator of l.

Third, for the selection-block map, the relation $\simeq_\eta$ states that η-instructions are inserted only at nodes l holding an `Inop` instruction in f, and that the predicate used to select the register r_{s} in the η-instruction is expressing paths from a (loop-header) node h to l, with h strictly dominating l.

Note that in Definition 5.6, source and destination register names are under-constrained. Technically speaking, at this

point of the specification, the GSA function is *not* SSA. We re-establish the SSA property and register-use consistency using our global renaming post-phase that we describe next.

5.4 Specification of Register Renaming

Because GSA adds η-instructions, and register definitions need to remain unique, we need to (i) generate fresh register names, and (ii) to readjust register uses to keep them consistent: past a loop exit node, the fresh generated register name should be used instead of the initial one. To this end, we rely on a register renaming map, that we compute during the insertion of η-instructions.

Each of the loop-exit nodes will hold an η-instruction $r_\mathrm{d} \leftarrow \eta(p, r_\mathrm{s})$, where r_d must be fresh, p is the loop-exit predicate, and r_s is the variable defined by the corresponding μ-instruction. So, in the renaming phase, we need to keep track of how each variable defined using a μ-instruction will be copied to the fresh register r_d at node l_{exit}. We store all of this information in the following data-structure.

Definition 5.7 (Register Renaming Map Validity)**.** Let f be an SSA function. A register renaming map $\mathbb{R}$ is valid with respect to f and tf, written $f, tf \models \mathbb{R}\ \checkmark$, if and only if the following two conditions hold.

1. For all r_μ such that $\mathbb{R}(r_\mu) = \lfloor\overrightarrow{(r_\eta, l_\eta)}\rfloor$, (i) all r_{η_i} are fresh in f and register r_μ is not fresh in f, (ii) there exists l_μ with $r_\mu \leftarrow \mu(r_0, r_i) \in tf.\mathcal{M}(l_\mu)$, and (iii) there exists $r_\eta \leftarrow \eta(p, r_\mu) \in tf.\mathcal{E}(l_\eta)$.
2. For all l and $r_\mathrm{d} \leftarrow \phi(\vec{r}) \in f.\Phi(l)$, if $\mathbb{R}(r_\mathrm{d}) = \lfloor\overrightarrow{(r_\eta, l_\eta)}\rfloor$ then $l \succ l_{\eta_i}$ for all i.

The implementation of the renaming pass is as follows. For any register r used at node n, if $(r_\eta, l_\eta) \in \mathbb{R}(r)$, then r is renamed to r_η if $l_\eta \succ n$. Register r is left unchanged otherwise. Indeed, if $l_\eta \succ n$, because the renaming map is valid (Definition 5.7), we prove that an η-instruction $r_\eta \leftarrow \eta(p, r_\mu)$ necessarily dominates node n, and therefore the new register r_η should be used instead of r. The actual renaming pass is *bijective*, assuming that the renaming $\mathbb{R}$ is valid with respect to the initial SSA function. We apply this renaming process using $\mathbb{R}$ on each of the maps $\mathcal{M}, \mathcal{E}$ and $\mathcal{I}$, through the function $\mathrm{rename}(\mathbb{R}, \mathcal{I}, \mathcal{M}, \mathcal{E}) = (\mathcal{I}', \mathcal{M}', \mathcal{E}')$, yielding three renamed code maps.

5.5 Top-Level Correctness Theorem

The overall correctness theorem states the overall semantics preservation between the initial C code and the GSA code that is produced by the compiler.

Theorem 5.8. *Let P_c be a safe C program (i.e. that does not go wrong). Suppose the compilation of P_c succeeds, and produces a GSA program P_g. Then, running P_g from its initial state T_{init} emits a trace t only if running P_c from its initial state S_{init} emits the trace t.*

$$\forall P_c\ P_g.\quad Safe(P_c) \wedge \mathtt{Comp}(P_c) = \mathtt{OK}(P_g)$$
$$\implies (\forall t.\quad \vdash S_{init} \xrightarrow{t}{}^{*} \implies \vdash T_{init} \xrightarrow{t}{}^{*}).$$

As in CompCert's formal development, this backward simulation theorem is proven by showing a forward simulation between C and GSA, and then proving that the semantics of GSA is deterministic. The forward simulation itself can be decomposed into individual forward simulations, one for each of the compilation pass.

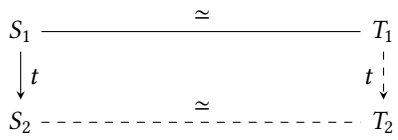
At the heart of the proof of the forward simulation for the conversion from SSA to GSA, we need to exhibit a binary (simulation) relation on execution SSA and GSA states, $\cdot \simeq \cdot$, which carry enough information to prove that both programs behave the same, i.e. emit the same observable trace.

Lemma 5.9 states the forward lock-step simulation diagram between GSA and SSA. It relates the states of an SSA function with the states of a GSA function, and shows that, for every execution step in SSA, there exists an execution step in GSA which ensures both states stay related.

Lemma 5.9. *Let f be an SSA function, such that $wfSSA(f)$. Let tf be the corresponding generated GSA function, with the companion predicate matrix $\mathbf{P}$ and renaming map $\mathbb{R}$, with $\mathbf{P}, \mathbb{R} \vdash f \equiv tf$.*

$$\forall S_1\ t\ S_2\ T_1.\quad \vdash S_1 \xrightarrow{t} S_2\ \wedge\ S_1 \simeq T_1\ \implies$$
$$\exists T_2.\quad \vdash T_1 \xrightarrow{t} T_2\ \wedge\ S_2 \simeq T_2.$$

This lemma can be visualised as follows, where solid lines are hypotheses, and dashed lines are conclusions.



5.6 Simulation Relation

We now describe the simulation relation $\cdot \simeq \cdot$, which relates an SSA semantic state $\mathcal{S}$ to a GSA semantic state $\mathcal{T}$. As is often the case in simulation proofs, the main difficulty lies in defining that very simulation relation. We define the relation as follows[3], and we prove it satisfies Lemma 5.9.

Definition 5.10 (Simulation Relation)**.**

$$\frac{f \hookrightarrow l \qquad \mathbb{R} \models_l rs \approx rs' \qquad \big(\forall d, d \succ l \implies rs \models_\mathrm{p} \mathbf{P}_{d,l} \Downarrow 1\big)}{\mathcal{S}(f, l, rs, m) \simeq \mathcal{T}(tf, l, rs', m)}$$

$$\frac{\forall r,\ r \notin \mathrm{fresh}(f) \implies rs'(r) = rs(r) \qquad \forall r\ r'\ n,\ \mathbb{R}(r') = \lfloor\overrightarrow{(r, n)}\rfloor \text{ and } n \succ l \implies rs'(r) = rs(r')}{\mathbb{R} \models_l rs \approx rs'}$$

[3]We only present the simulation relation on standard semantic states. The details about other states can be found in our Coq development.

Given an SSA function f, a GSA function tf that is a possible translation of f, their semantic states match at the current program point l, when their register states agree, and a further property related to predicates holds.[4]

The agreement between register state rs in SSA and rs' in GSA, written $R \models_l rs \approx rs'$, is defined in Definition 5.10. The basic case states that if register r is not fresh, it should always be equal to its counter part in SSA. The second case, when r is fresh, is when it exists in the renaming map $\mathbb{R}$. The relation holds when the definition of the renamed register strictly dominates the current point, which guarantees it will have the same value as the register before the renaming.

The last property we need is the fact that the predicates at the current node l always evaluate to true if $d \succ l$, namely we have $rs \models_{\mathrm{p}} \mathbf{P}_{d,l} \Downarrow 1$ for all strict dominators d of l. For junction points, we use the coherence property to prove that the next predicate will evaluate to true, as at least one predecessor evaluates to true. For instructions such as γ-instructions, μ-instructions and `Iop` instructions that modify a register, we rely on the fact that the modified register cannot appear in the predicate due to the projection of the predicate at the current node, to show that it will still evaluate to true.

Once we know that predicates evaluate to true at the current point l, we prove the behaviour of merge-instructions. μ-instructions are simple, as they behave like ϕ-instructions. However, the difficulty here is that a register r_{d} defined by a μ-instruction will likely be in $\mathbb{R}(r_{\mathrm{d}}) = \lfloor \overrightarrow{(r_\eta, l_\eta)} \rfloor$. We need to show that this renaming does not interfere, and we can therefore use $\forall i. l \succ l_{\eta_i}$, which comes from Definition 5.7 (2), to show that we are not updating a register that was also renamed somewhere at l.

Proving the correctness of γ-instructions relies on mutual exclusivity: we prove that the same register is picked in SSA and GSA, as we know that the predecessor's predicate evaluated to true, and that no other predecessor predicate can be true. The three-valued logic allows us to ignore paths that were never executed and whose conditions might not be evaluable. Then, we use the coherence property to prove that the predicate at the junction point will still to hold: at least one predecessor evaluates to true, and any register modified by the ϕ-instruction has been projected away already.

Finally, for η-instructions, the correctness comes directly from the simulation relation $\simeq$. The main problem in this proof is showing that the final register maps agree, as it is modifying fresh variables. This requires proving that the fresh variables exist in $\mathbb{R}$, and that the register that maps to it in $\mathbb{R}$ is the one being assigned by the η-instruction.

[4]We further need to maintain the invariant $f \hookrightarrow l$, stating that l is syntactically reachable in the CFG of f, to reason about the dominance relation.

Table 1. Number of lines of code (SLOC), generated using `coqwc` in our development relative to CompCertSSA and to CompCert. Validated OCaml code is also included.

	Spec	Proof	OCaml	Total
CompCert	59439	69487	28703	157629
CompCertSSA	15693	27868	3161	46722
Dom. completeness	1413	2735	0	4148
GSA	6320	9035	1433	16788
Syntax & semantics	122	685	0	807
Generation	4359	4947	314	9620
SMTCoq integration	1839	3403	1119	6361

6 Implementation of the GSA Construction within CompCertSSA

We implement our specification as a translation pass from SSA to GSA. The main difficulty is proving the coherence and the well-exclusivity of the predicate matrix, which is then used in the main SSA-to-GSA translation pass to assign the predicates to the γ- and η-instructions. In Section 6.1, we describe how we populate the predicate matrix, and the proofs of coherence and well-exclusivity. We give further details about our implementation in Section 6.2, including a the additional compiler passes that we added. Section 6.3 then covers the integration of the external SMT solver and its validation. Finally, Section 6.4 covers the main limitations of the current implementation. To get a sense of the scale of the implementation, we give in Table 1 the total number of lines, relative to CompCert and CompCertSSA.

6.1 Generation and Validation of the Predicates

While translating ϕ-instructions and inserting η-instructions, we build up the predicate matrix $\mathbf{P}$. Each time a new predicate is needed, and if it is not already present, the entry $\mathbf{P}_{i,.}$ is populated using Tarjan's algorithm. Even though i could be any dominator of the current node, we pick the immediate dominator to minimise the number of computed paths.

The correctness of the generated predicates is validated after-the-fact. To check coherence and well-exclusivity, we use unsatisfiability queries to an SMT solver, which outputs a certificate proving the unsatisfiability. This certificate can then be checked using the proof checker from SMTCoq [19] which we directly integrate as a validator in our translation. More information about the integration of the embedding of three-valued logic into SMTCoq formulas is given in Section 6.3. The correctness of the SMT solver states that, if it finds that a negated[5] predicate is unsatisfiable, then this predicate unconditionally evaluates to true. This correctness result is then used in the Coq proofs without having to trust the SMT solver itself. Despite the induced cost of checking SMT certificates, relying on a solver was key, in the course of

[5]This negation is defined as setting 1 to 0, and both 0 and $\frac{1}{2}$ to 1.

our formalisation, to cope with partial (or wrong) intuitions we could have had about the sufficient properties that GSA ought to satisfy to be correct.

We also extend the predicate language with an implication rule ($p_a \to_Ł p_b$), taken to be the implication as defined by Łukasiewicz three-valued logic $Ł_3$ [6], to be able to formulate all the needed properties. Interestingly, if one were to use the same definition of implication as in binary logic, the three-valued logic would have no tautologies, making it impossible to express SMT queries properly, as all values being undefined would always be an acceptable assignment. Instead, $Ł_3$ defines an implication where $\frac{1}{2} \to_Ł \frac{1}{2} \equiv 1$, making it possible to formulate tautologies.

Validating $\forall i\, j.\, \mathbf{P}\ \mathrm{coh}\ (i, j)$ is straightforward: it corresponds directly to checking the coherence relation on the predicates, where the logical implication is translated into $\to_Ł$. We then prove that this implies the coherence of the predicate matrix. Validating the mutual exclusivity of predicates is more involved. We encode the implication used in Definition 5.4, i.e. if p_a is 1, then p_b is either $\frac{1}{2}$ or 0, into the three-valued logic as $p_a \to_Ł p_b \to_Ł \neg p_b$. For a node j, and for $m, n \in \mathrm{preds}(j)$, the query $\left(\mathbf{P}_{i,m} \to_Ł \mathbf{P}_{i,n} \to_Ł \neg\mathbf{P}_{i,n}\right) \wedge \left(\mathbf{P}_{i,n} \to_Ł \mathbf{P}_{i,m} \to_Ł \neg\mathbf{P}_{i,m}\right)$ proves the mutual exclusivity of predicates $\mathbf{P}_{i,m}$ and $\mathbf{P}_{i,n}$ if it always evaluates to true, i.e. the negation of the predicate is unsatisfiable.

6.2 Conversion to GSA and Other Compiler Passes

Converting ϕ-instructions to μ- or γ-instructions depends on whether the ϕ-instruction is at a loop header or not. We use an efficient loop-header checker [5, 7] that we can safely trust: its correctness does not affect the soundness of the translation. Similarly, we do not need to formally establish that η-instructions are inserted at loop-exit nodes only, and it is sufficient to prove the correctness of the prior placement of loop-exit landing pads (`Inop` instructions) during an RTL to RTL pass. We avoid the need to reason about loop-header and exit nodes thanks to (i) the way we formulate our semantics for GSA instructions and (ii) the preservation of the code structure ensured by the generation algorithm of GSA from SSA.

Inserting loop-exit landing pads prior to the GSA translation has many other benefits. It saves us from having to insert new nodes in the function during the actual SSA to GSA translation, and the structure of the CFG is therefore preserved. All the properties of SSA, such as the dominance test, and the reachability of nodes can be reused directly from SSA and on the original graph, which reduces the amount of proofs that need to be performed.

Additionally, we normalise loops to ensure they have a single entry point and a single latch, so that we can generate μ-instructions with a well-defined, non-blocking semantics.

Finally, we implement an unverified and trusted compiler pass to translate GSA back to SSA, in order to generate machine code, and to be able to test programs by running them. Proving the correctness of this pass is left as future work. This translation is not easy to prove correct: one cannot assume that the order of the arguments of the γ-instruction corresponds to the order of the predecessors anymore.

Another technical point is that we need a completeness result for the dominance test to implement and prove our translation to GSA. Here, we needed to integrate a verified, but unpublished, formal development, proving the completeness of the dominance test of CompCertSSA (shown as part of CompCertSSA in Table 1).

6.3 Integration with SMTCoq

The proof of correctness for the validation of the coherence and well-exclusivity of the predicate matrix heavily relies on unsatisfiability checks guaranteeing that the property indeed always holds. Hence, the SMT solver should itself formally give this guarantee, either via a direct proof of correctness, or by generating proof certificates that would then be checked with a verified certificate checker.

In our GSA construction, we opted to use SMTCoq [19], which includes a certificate checker for the `veriT` [8] SMT solver for its internal SMT formulas. The main use-case of SMTCoq is to provide Coq tactics that call the SMT solver and solve goals while proving theorems in Coq, but we wanted to integrate the certificate checker itself into our validator, and eventually extract it to OCaml.

The certificate checker can already be extracted. However, to prove properties about GSA predicates, we need to embed their three-valued logic in terms of SMTCoq formulas. In turn, we need to prove this translation sound. SMTCoq supports linear arithmetic as a theory, so three-valued logic can be implemented using min and max functions (see Fig. 6).

The translation and its soundness proof are rather tedious: we perform a Tseytin transformation to flatten the predicate, before we can encode it efficiently as an SMTCoq formula using arrays and sharing of redundant formulas and atoms.

6.4 Limitations

Unsupported Features. Currently, our extracted formal development is able to compile all the expected CompCert test programs successfully, and only fails on programs with conditions that are dependent on memory, such as pointer equality checks. These are currently unsupported, as it would need a proof that stack allocations made when functions are called do not make invalid pointers valid again, which could change the result of executing the equality check. We could leverage the hypothesis of well-defined C program semantics to ensure this does not invalidate the construction of GSA.

Performance. The current validation increases compilation time significantly which will have to be addressed in

the future. Running CompCertGSA without any validation on the standard C tests included in CompCert takes around 4.5s. Then, running CompCertGSA with validation using Z3 as a trusted SMT solver (without the SMTCoq checker) takes around 156s. Finally, running CompCertGSA with full validation including SMTCoq takes around 1872s. This large difference in execution time is mainly due to SMTCoq having to use an older version of veriT as the SMT solver back end, which seems to have large differences in execution time for some inputs. However, there are many other ways in which the execution time could be improved. Firstly, the size of the predicates could be simplified, which could be done in an unverified and untrusted manner. Secondly, the handling of case statements could be improved, as a new predicate is introduced for each branch which complicates the predicates substantially. Finally, our interface to SMTCoq could be more efficient, as it currently performs many reads and writes to files to communicate with veriT, whereas queries can be built in memory when using Z3.

7 Related Work

There are various, different informal definitions of GSA in the literature. GSA was first introduced by Ottenstein et al. [23] in 1990 as part of the program dependence web (PDW), which was inspired by the extended SSA form developed by Alpern et al. [1]. It was later refined by Campbell et al. [9], describing different semantics that the PDW could have, such as standard control-flow semantics, data-flow semantics, or even demand-driven semantics. The purpose of the PDW was mainly to produce a referentially transparent program dependence graph [13], which could target more exotic architectures that relied purely on data-flow. There are therefore a large number of different gates that are defined to support various types of semantics.

However, GSA itself is well suited for symbolic analysis over control-flow boundaries, thereby allowing for more powerful optimisations than could be applied to SSA. Therefore, Havlak [16] introduced the Thinned Gated Single Assignment (TGSA) form, which retained the important parts of Ottenstein et al.'s GSA formulation related to symbolic analysis. Tu and Padua [28, 29] described a similar version of GSA and developed an efficient way of building GSA, leveraging Tarjan's algorithm to solve the single-source path expression problem [26]. But these are formulations of various different versions of GSA, that all behave slightly differently. In addition to that, they all operate over idealised languages and none of them have a formal semantics; the definitions of the SSA language that is extended is not always clear, as well as what the predicates exactly consist of, or how these are evaluated. On the contrary, our formal semantics of GSA operates over the RTL language of CompCert; it is formalised in Coq and validated by our proof of correctness of the construction pass.

There have been various recent uses of GSA to either perform equivalence checks or optimisations or high-level synthesis [3, 11, 12, 22, 27]. None of these transformations are formally verified. Among them, Tristan et al. [27] developed an algorithm to detect the equivalence of the LLVM CFG before and after optimisation passes, and used a monadic GSA language tracking the memory usage of each instruction.

Currently, our correctness proof relies on a SMT solver verifier, based on SMTCoq [19]. SMTCoq sends SMT queries to an external, untrusted SMT solver, and then validates the result by checking a proof certificate generated by the SMT solver. In our work, we rely on a extractable checker provided by SMTCoq, and we integrate this checker within our work by formally proving the correctness of our encoding of predicates, their evaluation, and the corresponding queries into the SMTCoq framework. IsaSAT is one of the few verified SAT solvers implemented in Isabelle [14, 15].

8 Conclusions and Future Work

We make a number of contributions towards the integration of GSA-based techniques into verified compilers. This includes providing the first formal semantics for GSA, proving the semantics preservation of a specification for the SSA to GSA conversion, and integrating the translation pass into CompCertSSA, demonstrating its feasibility.

Proving the correctness of the translation to GSA does not require formalising the notions of loop headers and loop exit nodes, however, expressing optimisations or analysis passes on GSA would require to do so. We could extend this work with a set of well-formedness properties similar to the SSA well-formedness from CompCertSSA.

Our semantics expresses the meaning of γ- and η-instructions with predicates, thus making them control-flow independent. While we focus on this aspect in the paper, in the future, we would like to formalise a data-flow or event-driven semantics for GSA, where all control-dependencies have been translated to data-dependencies. Such a language could be used as a target for translation validation of complex optimisations that are independent of control-flow. In addition to that, such a language could also be used to target back ends such as hardware directly.

Artefact Availability

The formal development, including its proofs is available as an artefact [17].

Acknowledgments

We would like to thank John Wickerson and the anonymous reviewers for their helpful feedback. This work was partially funded by the CNRS via the INS2I (Appel Unique 2022) and by the EPSRC via the Research Institute for Verified Trustworthy Software Systems (VeTSS).

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/73560.73561

[2] C. Scott Ananian and Martin Rinard. 1999. *Static Single Information Form.* Technical Report. Master's Thesis, Massachussets Institute of Technology.

[3] Manuel Arenaz, Pedro Amoedo, and Juan Touriño. 2008. Efficiently Building the Gated Single Assignment Form in Codes with Pointers in Modern Optimizing Compilers. In *Euro-Par 2008 – Parallel Processing*, Emilio Luque, Tomàs Margalef, and Domingo Benítez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 360–369.

[4] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (March 2014), 35 pages. https://doi.org/10.1145/2579080

[5] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. 2013. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In *SAS (LNCS, Vol. 7935)*. Springer, 324–344.

[6] L. Borowski. 1970. *Selected Works of J. Łukasiewicz.* Nort Holland.

[7] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 735)*, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.). Springer, 128–141. https://doi.org/10.1007/BFb0039704

[8] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. 2009. veriT: An Open, Trustable and Efficient SMT-Solver. In *Automated Deduction – CADE-22* (Berlin, Heidelberg), Renate A. Schmidt (Ed.). Springer Berlin Heidelberg, 151–156. https://doi.org/10.1007/978-3-642-02959-2_12

[9] Philip L Campbell, Ksheerabdhi Krishna, and Robert A Ballance. 1993. Refining and defining the program dependence web. *Cs93-6, University of New Mexico, Albuquerque* (1993).

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[11] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. 2020. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (Nov. 2020), 4229–4239. https://doi.org/10.1109/tcad.2020.3012866

[12] Shuhan Ding, John Earnest, and Soner Önder. 2014. Single Assignment Compiler, Single Assignment Architecture: Future Gated Single Assignment Form*; Static Single Assignment with Congruence Classes. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 196–207. https://doi.org/10.1145/2544137.2544158

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[14] Mathias Fleury. 2020. *Formalization of logical calculi in Isabelle/HOL.* Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://tel.archives-ouvertes.fr/tel-02963301

[15] Mathias Fleury and Christoph Weidenbach. 2020. A Verified SAT Solver Framework including Optimization and Partial Valuations. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020 (EPiC Series in Computing, Vol. 73)*, Elvira Albert and Laura Kovács (Eds.). EasyChair, 212–229. https://doi.org/10.29007/96wb

[16] Paul Havlak. 1994. Construction of thinned gated single-assignment form. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 477–499.

[17] Yann Herklotz, Delphine Demange, and Sandrine Blazy. 2022. *CompCertGSA.* https://doi.org/10.5281/zenodo.6009632

[18] Inria 2021. *The Coq proof assistant reference manual.* Inria. http://coq.inria.fr Version 8.13.2.

[19] Chantal Keller. 2019. *SMTCoq: Mixing Automatic and Interactive Proof Technologies.* Springer International Publishing, Cham, 73–90. https://doi.org/10.1007/978-3-030-28483-1_4

[20] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* (2009).

[21] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

[22] Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable conditional induction variables (CIV) analysis. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 213–224. https://doi.org/10.1109/CGO.2015.7054201

[23] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

[24] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. ACM, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[25] Diogo Sampaio, Rafael Martins, Caroline Collange, and Fernando Magno Quintão Pereira. 2012. Divergence Analysis with Affine Constraints. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. 67–74. https://doi.org/10.1109/SBAC-PAD.2012.22

[26] Robert Endre Tarjan. 1981. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (July 1981), 594–614. https://doi.org/10.1145/322261.322273

[27] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

[28] Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/207110.207115

[29] Peng Tu and David Padua. 1995. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *Proceedings of the 9th International Conference on Supercomputing* (Barcelona, Spain) *(ICS '95)*. Association for Computing Machinery, New York, NY, USA, 414–423. https://doi.org/10.1145/224538.224648

[30] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136

Received 2022-09-21; accepted 2022-11-21