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

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. [1988] 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 $\phi$-instruction. At a control-flow junction point with, say, three predecessors, a $\phi$-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 [1991], 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 [Ananian and Rinard 1999] 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. [1990] to extend program dependency graphs with the SSA property, leading to an IR devoted to compilation targeting data-flow architectures and simulators. In GSA, $\phi$-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 $\phi$-instruction, the GSA form extends $\phi$-instructions with gates, i.e. Boolean conditions characterising the control-flow paths leading to that $\phi$-instruction and hence determining which $\phi$-argument has to be selected. For example, suppose an SSA $\phi$-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 $\phi$-instruction would be gated as follows $x_3 = \phi((c, x_1), (c, x_2))$, meaning if condition $c$ evaluates to true, then it will select $x_1$, otherwise it will select $x_2$. The selection of the $\phi$-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 information, so that they become "referentially transparent".

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. Our formalisation has been conducted 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.

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.

```c
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 $\varphi$-instructions that handle control-flow joins. At node 12, the $\varphi$-instruction expresses 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 $\varphi$-block (grey nodes in Fig. 2b) groups all the $\varphi$-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, $\varphi$-instructions are replaced by $\mu$- and $\gamma$-instructions, and new instructions called $\eta$-instructions are introduced. These instructions are meant to better reflect control-dependencies. Indeed, they include strictly more information than $\varphi$-instructions, and therefore allow for the formulation of different types of semantics, such as demand-driven or data-flow driven executions. An SSA $\varphi$-instruction becomes either a $\gamma$-instruction when it is located at a simple junction point (e.g. node 12), or a $\mu$-instruction when it is located at a loop-header (e.g. node 3). The $\gamma$-instructions are augmented with gates, namely predicates characterising the control-flow path corresponding to the reaching definition of each of the $\varphi$-arguments. For $\gamma$-instructions, the selection of the argument is based on the predicate evaluating to true. For instance, at node 12, predicates guarding the $\gamma$-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 $\mu$-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, $\mu$-instructions are of the form $x = \mu(x_0, x_1)$, where $x_0$ is the initial value before entering the loop, and $x_1$ is the version of $x$ modified inside the loop body. Intuitively, arguments of a $\mu$-instruction are not guarded like in $\gamma$-instructions, since no useful predicate helps to distinguish when $x_0$ or $x_1$ 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 $x_1$ with the True predicate wouldn’t always be correct. At loop exits, GSA introduces $\eta$-instructions (e.g. node 14). An $\eta$-instruction $x = \eta(c, x_p)$ selects a loop-defined variable $x_p$ when the loop-exit condition $c$ is satisfied. Intuitively, $\eta$-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 $\gamma$-, $\mu$- and $\eta$-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 [Barthe et al. 2014], which is an extension of the CompCert C compiler [Leroy 2009a]. CompCert is programmed and verified using Coq [Inria 2021]. 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 \([x]\) (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 \(e\) for the empty list. Vectors are written \(\overrightarrow{v}\), and \(v_k\) denotes the \(k\)-th element of \(\overrightarrow{v}\).

\[
i \ ::= \ \text{Inop}(l) \quad \text{no-op instr.} \\
| r \leftarrow \text{1op}(o, \overrightarrow{r}, l) \quad \text{arith. operator } o \text{ on } \overrightarrow{r} \\
| \text{Icond}(o, \overrightarrow{r}, l_1, l_2) \quad \text{conditional } o \text{ on } \overrightarrow{r} \\
| \ldots \quad \text{other RTL instructions} \\
I \ ::= l \mapsto i \quad \text{instruction map} \\
i_\phi \ ::= r \leftarrow \phi(\overrightarrow{r}) \quad \phi\text{-instruction} \\
\Phi \ ::= l \mapsto i_\phi \quad \phi\text{-block map} \\
f \ ::= (I, \Phi) \quad \text{SSA function}
\]

Figure 3. Syntax of SSA.

3.1 SSA Representation

An SSA function \(f\) (see Fig. 3) is modelled as a record made of a CFG \(I\) of instructions over pseudo-registers, and a map \(\Phi\) for \(\phi\)-instructions. \(I\) 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 \(o\) to a list of pseudo-registers \(\overrightarrow{r}\), stores its result in a register \(r\) 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 \(o\) applied to \(\overrightarrow{r}\). The \(\Phi\) map stores blocks of \(\phi\)-instructions. We write \(f.\Phi(l)\) for the \(\phi\)-block at node \(l\). A \(\phi\)-instruction written \(r_\phi \leftarrow \phi(\overrightarrow{r_\phi})\) assigns to \(r_\phi\) the value of one of its operands in \(\overrightarrow{r_\phi}\).

In SSA, there is a clear distinction between RTL-like instructions and \(\phi\)-instructions: both are stored in distinct maps; this simplifies the conversion to SSA, and allows for a smooth integration in CompCert. All \(\phi\)-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(\(\cdot\)) , capturing essential properties of SSA functions: structural constraints, unique definitions and strictness properties [Barthe et al. 2014].

The placement of \(\phi\)-instructions in SSA relies on the notion of dominance between nodes [Cytron et al. 1991]. 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 [Leroy 2009b]. In this paper, to simplify the presentation, and we omit $G$. For the sake of completeness, we keep the trace in small-step execution, although the instructions we present in this paper do not emit observable events, and thus step with the empty trace $\epsilon$.

Semantic states are written $S(f, l, rs, m)$. 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 $\phi$-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. $f \not\vdash l'$). Here, the standard RTL rule applies, as there is no $\phi$-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. $f \vdash l'$). In that case, a (potentially empty) $\phi$-block $b_\phi$ exists, and is executed before reaching the regular instruction at $l'$. All its $\phi$-instructions are executed in parallel and their result is assigned to their respective destination registers $b_\phi$, $k$ is determined by a conventional numbering of the predecessors of each of the CFG nodes $(\text{preds}(l'))_k = l$. The execution updates the registers state to $rs'$, so that the value of each register $r$ assigned in $b_\phi$ becomes the value of the $k$-th operand of the corresponding $\phi$-instruction in $rs$.

![Figure 4. Semantics of SSA (excerpt).](image)

4 Gated SSA Representation

To define our Gated SSA form, we take inspiration from previous existing work on GSA [Arenaz et al. 2008; Campbell et al. 1993; Ding et al. 2014; Ferrante et al. 1987; Havlak 1994; Ottenstein et al. 1990; Tu and Padua 1995a,b]. 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.

4.1 Syntax of GSA

The definition of a function is shown in Fig. 5. It includes three maps, a map $I$ for the CFG over regular instructions (as in SSA), a map $M$ for merge instructions, i.e. $\gamma$- and $\mu$-instructions, and a map $E$ for $\eta$-instructions. This separation makes it a natural extension of SSA, because only the map $E$ has to be newly constructed in GSA. The map $M$ has the same structure as $\Phi$ in SSA: $\phi$-instructions have either been converted to either $\mu$- or $\gamma$-instructions. Each of the $M$ and $E$ 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 $\mu$- and $\gamma$-instructions, and replace the SSA $\phi$-instructions. $\mu$-instructions, written $r_d \leftarrow \mu(r_0, r_1)$, 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 $r_1$ to $r_d$. This instruction does not have
predicates guarding its arguments, and therefore behaves just like its \( \phi \)-instruction counterpart, based on the executed control-flow. \( \gamma \)-instructions, written as \( r_d \leftarrow \gamma((p, r)) \), are the other replacement for \( \phi \)-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 \( \gamma \)-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 \( x_5 \).

Second, selection instructions are extra instructions, introduced when generating GSA, and have no counterpart in SSA. All \( \eta \)-instructions, written \( r_d \leftarrow \eta(p, r_p) \), are placed at loop exit nodes. These instructions indicate a termination condition \( p \) of the loop, which asserts that the register \( r_p \) is ready to use, and can be assigned to \( r_d \). Intuitively, they behave like predicates moved.

Predicates play an essential role in the semantics of \( \eta \)- and \( \gamma \)-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 \( \gamma \) or \( \eta \) 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 \( a \) and operands \( \gamma \). To reflect path condition composition, we need to include the conjunction \( \land \) of predicates, for nested conditionals, and the disjunction \( \lor \) 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 \( \gamma \)-, \( \mu \)- or \( \eta \)-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 \( \phi \)-blocks are executed.

To define the semantics of \( \gamma \)- and \( \eta \)-instructions, we first need to define the semantics for GSA predicates. The evaluation of predicates \( p \models c \) is defined by induction on the structure of the predicate (see Fig. 6, where \( \models \) 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 1nop(\( l' \)) instructions, as for the \( \phi \)-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 1nop 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 \( T(f, l, r, m) \). 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 NJoJoin), the register state is only updated by \( \eta \)-instructions. Otherwise (rule Join), \( l' \) is a junction point, and the merge-block must be executed. The \( \eta \)-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 \( \eta \)-instructions and merge instructions. Rule \( \text{Eta} \) (top of Fig. 7) defines how to execute a non-empty list of \( \eta \)-instructions. Executing an \( \eta \)-instruction \( r_d \leftarrow \eta(q, r) \) requires predicate \( q \) to evaluate to 1: it must hold in \( rs \). All \( \eta \)-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 \( \phi \)-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 \( \mu \)-instructions (rule \text{Merge}p), the selection is done only via \( k \), designating the index of the control-flow predecessor that leads to \( l' \), as was done for \( \phi \)-instruction. In
We ensure that our semantics stays deterministic by choosing the first such \( n \) whenever two predicates \( q_n \) and \( q_{n+1} \) would be simultaneously true. However, as we explain in Section 5.2, predicates guarding \( \gamma \)-arguments are provably mutually exclusive in GSA.

## 5 Conversion from SSA to GSA

To convert SSA into GSA, the \( \phi \)-instructions need to be replaced by \( \gamma \)- or \( \mu \)-instructions, depending on whether the \( \phi \)-instruction was at a simple junction point or at a loop header. In addition to that, extra \( \eta \)-instructions need to be added to loop exits. The difficulty of the transformation is the calculation of the predicates for \( \gamma \)- and \( \eta \)-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 \( \eta \)-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 [Herklotz et al. 2022].

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 \( \eta \)-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 \( \gamma \)- and \( \mu \)-instruction from \( \phi \)-instructions, (ii) insert an \( \eta \)-instruction for each \( \mu \)-instruction, and (iii) register renaming for the inserted \( \eta \)-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 \( \eta \)-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 $\eta$-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 $\phi$-block directly to the final merge-block, but without having to consider how $\eta$-instructions are inserted, and thus how the renaming is performed. We therefore generate a predicate matrix $P$ containing all the predicates that were generated during the translation and a renaming map $R$ 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 Section 5.2 and Section 5.4 respectively. Finally, the properties we need about $P$, $R$ 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 SMT-Coq [Keller 2019]. We explain our encoding in Section 6.1.

Definition 5.1 (SSA to GSA Translation Specification).

\[ f = (I, \Phi) \quad \text{and} \quad \text{rename}(R, I_t, M_t, \epsilon_t) = (I_r, M_r, \epsilon_r) \]

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 $f = (I, \Phi)$ and $tf = (I_{tr}, M_{tr}, \epsilon_{tr})$. In the second line of the specification, we require the predicate matrix $P$ and the renaming map $R$ to satisfy properties that tell us enough about their correctness so that we can prove that the right argument will be picked in $\gamma$-instructions, and that the predicates in $\eta$-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-instruction). We explain these code correspondences in Section 5.3.

Finally, the renaming is performed after the three kinds of maps are generated, so that variables introduced by an $\eta$-instruction are used by their definition.

5.2 Specification of GSA Predicates

In GSA, predicates guard the register selection in $\gamma$- and $\eta$-instructions. Hence, they should reflect the dynamic control-flow of the program. Essentially, one must generate a predicate for each path to the $\gamma$-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 [1981]. The suggested algorithm is used 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 $\eta$-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 $P$, associating predicates, in $P$, to pairs of nodes in the CFG of the initial SSA function $f$. Morally, the predicate associated to $(i, j)$, written $P_{i,j}$, 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 $x > 50 \land x_1 < 9$.

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 \( P_{ik} \) 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, \( P_{ij} \) ought to hold, as soon as one of the paths from \( i \) to \( j \) corresponding to a predicate \( P_{ik} \land c_{k,j} \), where \( k \) is a predecessor of \( j \), has been taken.

We formalise this intuition with the three-place relation \( \text{P coh} \ (i, j) \); it states, for a function \( f \), a predicate matrix \( P \), and two nodes \( i \) and \( j \) that the predicate \( P_{ij} \) is, locally, coherent for nodes \( i \) and \( j \).

**Definition 5.2 (Local Coherence).** Let \( f \) be an SSA function, \( P \) a predicate matrix, and \( i \) and \( j \) be two nodes in the CFG of \( f \). The relation \( \text{P coh} \ (i, j) \) is defined as follows:\(^1\):

\[
\begin{align*}
\text{CohNDom} & \quad i \ngtr j \quad \text{P coh} \ (i, j) & \quad \text{CohEQ} & \quad \text{isTrue}(P_{ii}) \\
\text{CohSDom} & \quad i \ngtr j \quad \sqrt{P_{ik} \land c_{k,j}}_{j \rightarrow j} \quad \text{P coh} \ (i, j)
\end{align*}
\]

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 \( P_{ij} \) 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 \( P \). 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 \( p|j \), which replaces any atomic condition \( c \), or \( \bar{c} \) 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 \( \phi \)-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

---

\[^1\]The definition of semantic implication on predicates is standard: \( P \Rightarrow Q \) means that, for any register state \( rs \), if \( P \) evaluates to 1, then so does \( Q \).
to specify which variables should be considered as relevant to the truthfulness of predicate \( P_{i,j} \); 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.

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

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 SSA function, i.e., at future \( \gamma \)- and \( \eta \)-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 \( \phi \)-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 \( \gamma \)-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 \( P \). They are said to be mutually exclusive, written \( p_1 \parallel p_2 \), whenever for all register states \( rs \) they cannot both evaluate to true, i.e. if \( rs \models p_1 \downarrow 1 \), then \( rs \models p_2 \downarrow 1 \).

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 \( \gamma \)-instruction’s pair of arguments be mutually exclusive. Hence, we only consider non-loop-headers junction points – loop-header \( \phi \)-instructions are future \( \mu \)-instructions, that do not resort on predicates.

**Definition 5.5 (Well-Exclusive Predicate Matrix).** Let \( f \) be an SSA function. A predicate matrix \( P \) is well-exclusive for \( f \), written \( P \Rightarrow \), when for all node \( n \varphi \) in \( f \) that is not a loop header, and that holds a \( \phi \)-block, and any possible strict-dominator \( d \) of \( n \varphi \), i.e. \( d > n \varphi \), the following holds: for any two distinct nodes \( n_1, n_2 \in \text{preds}(n \varphi) \), we have \( P_{d,n_1} \parallel P_{d,n_2} \).

We now summarise the calculation and validation of GSA predicates. For any \( \phi \)-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 \( P \), on which we globally apply the projection operator \( \downarrow \) on all predicates columns, i.e. \( P_{i,j} \) is replaced by the projection \( P_{i,j,l} \). 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 \( \eta \)-instructions hence also preserves the CFG structure.

We thus introduce three code-correspondences: relation \( \approx \) handles the regular instruction maps, relation \( \approx_M \) handles the merge-block map, and relation \( \approx_{\gamma} \) handles the selection-block map.

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

\[
\begin{align*}
I(l) &= I_\varphi(l) \quad d > l \\
I &\equiv \downarrow I_l \\
P \vdash \Phi(l)_i &\approx^{l_\varphi} \mathcal{M}(l)_i \\
P \vdash r_d &\leftarrow \phi(r_0, r_1) \approx^{l_\varphi}_\varphi r_d \leftarrow \mu(r_0, r_1) \\
&\forall k, r'_k = (P_{n,preds(l)_k}, r_k) \\
P \vdash r_d &\leftarrow \phi(r'_k) \approx^{l_\varphi}_{\varphi} r_d \leftarrow \gamma(r') \\
I(l) &= [\text{nop}(l')] \\
h &> l \\
&\forall r_d p_s, r_d \leftarrow \eta(p_s, r_s) \in E(l) \Rightarrow p_s = P_{h,l} \\
P \vdash I &\approx_{\gamma} \mathcal{E}
\end{align*}
\]

Relation \( \approx \) is straightforward: both functions should have identical instructions at a node \( l \). Relation \( \approx_M \) states that, at \( l \), all \( \varphi \)-instructions and merge-instructions are pairwise related through \( \approx_\varphi \). Recall that \( \varphi \)-instructions are converted either to \( \mu \)- or \( \gamma \)-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 \( \varphi \)-instruction being related either to a \( \gamma \)-instruction or to a \( \mu \)-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 \( \phi \)-instruction is converted to a \( \gamma \)-instruction at node \( l \): to each register argument \( r_k \) in the \( \phi \)-instruction, we associate in the \( \gamma \)-instruction the predicate \( P_{n,preds(l)_k} \); 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 \( \approx_{\eta} \) states that \( \eta \)-instructions are inserted only at nodes \( l \) holding an \( \text{InOp} \) instruction in \( f \), and that the predicate used to select the register \( r_\eta \) in the \( \eta \)-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 \( \eta \)-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 \( \eta \)-instructions.

Each of the loop-exit nodes will hold an \( \eta \)-instruction \( r_\eta \leftarrow \eta(p, r_\mu) \), where \( r_\eta \) must be fresh, \( p \) is the loop-exit predicate, and \( r_\mu \) is the variable defined by the corresponding \( \mu \)-instruction. So, in the renaming phase, we need to keep track of how each variable defined using a \( \mu \)-instruction will be copied to the fresh register \( r_\eta \) at node \( \mathsf{lnext} \). 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 \( R \) is valid with respect to \( f \) and \( tf \), written \( f, tf \models R \) \( \checkmark \), if and only if the following two conditions hold.

1. For all \( r_\mu \) such that \( R(r_\mu) = \left( \left( r_\eta, l_\eta \right) \right) \), (i) all \( r_\eta \) are fresh in \( f \) and register \( r_\eta \) is not fresh in \( f \), (ii) there exists \( l_\mu \) with \( r_\mu \leftarrow \mu(r_\mu), r_\eta \in tf \cdot M(l_\mu) \), and (iii) there exists \( r_\eta \leftarrow \eta(p, r_\mu) \in tf \cdot E(l_\eta) \).
2. For all \( l \) and \( r_\eta \leftarrow \phi(\overline{f} \in f) \Phi(l) \), if \( R(r_\eta) = \left( \left( r_\eta, l_\eta \right) \right) \) then \( \mathsf{lnext} > l_\eta \) for all \( i \).

The implementation of the renaming pass is as follows. For any register \( r \) used at node \( n \), if \( (r_\eta, l_\eta) \in R(r) \), then \( r \) is renamed to \( r_\eta \) if \( l_\eta > n \). Register \( r \) is left unchanged otherwise. Indeed, if \( l_\eta > n \), because the renaming map is valid (Definition 5.7), we prove that an \( \eta \)-instruction \( r_\eta \leftarrow \eta(p, r_\mu) \) necessarily dominates node \( n \), and therefore the new register \( r_\eta \) should be used instead of \( r \). The actual renaming pass is bijective, assuming that the renaming \( R \) is valid with respect to the initial SSA function. We apply this renaming process using \( R \) on each of the maps \( M, E \) and \( I \), through the function rename\((R, I, M, E) = (I', M', E')\), 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 \( T_{\mathsf{init}} \) emits a trace \( t \) only if running \( P_c \) from its initial state \( S_{\mathsf{init}} \) emits the trace \( t \).

\[
\forall P_c, P_g . \quad \mathsf{Safe}(P_c) \land \mathsf{Comp}(P_c) = \mathsf{OK}(P_g)
\implies \quad (\forall t . \quad \vdash S_{\mathsf{init}} \Downarrow t \implies \vdash T_{\mathsf{init}} \Downarrow t').
\]

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, \( \cdot \mathrel{\approx} \cdot \), 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 \( P \) and renaming map \( R \), with \( P, R, f \models tf \).

\[
\forall S_1, T_1 . \quad \vdash S_1 \Downarrow t 
\implies S_2 \Downarrow t \land \vdash S_2 \approx T_2 \implies \vdash T_1 \Downarrow t'.
\]

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

\[
S_1 \overset{\approx}{\rightarrow} T_1
\]

\[
S_2 \Downarrow t \implies \overset{t}{\Downarrow} \rightarrow T_2
\]

### 5.6 Simulation Relation

We now describe the simulation relation \( \cdot \mathrel{\approx} \cdot \), which relates an SSA semantic state \( S \) to a GSA semantic state \( T \). As is often the case in simulation proofs, the main difficulty lies in defining that very simulation relation. We define the relation as follows\(^2\), and we prove it satisfies Lemma 5.9.

\(^2\)We only present the simulation relation on standard semantic states. The details about other states can be found in our Coq development.
The basic case states that if register \( r \) is fresh, is when it exists in the renaming map \( R \). The agreement between register state \( rs \) in SSA and \( rs' \) in GSA, written \( R \models I \) \( rs \approx rs' \), is defined in Definition 5.10. 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 \( \eta \)-instruction.

### 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 \( \gamma \)- and \( \eta \)-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 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 \( \phi \)-instructions and inserting \( \eta \)-instructions, we build up the predicate matrix \( P \). Each time a new predicate is needed, and if it is not already present, the entry \( P_r \) 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 SMT-Coq [Keller 2019] 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.

#### 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.

<table>
<thead>
<tr>
<th></th>
<th>Spec</th>
<th>Proof</th>
<th>OCaml</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompCert</td>
<td>59439</td>
<td>69487</td>
<td>28703</td>
<td>157629</td>
</tr>
<tr>
<td>CompCertSSA</td>
<td>15693</td>
<td>27868</td>
<td>3161</td>
<td>46722</td>
</tr>
<tr>
<td>Dom. completeness</td>
<td>1413</td>
<td>2735</td>
<td>0</td>
<td>4148</td>
</tr>
<tr>
<td>GSA</td>
<td>6320</td>
<td>9035</td>
<td>1433</td>
<td>16788</td>
</tr>
<tr>
<td>Syntax &amp; semantics</td>
<td>122</td>
<td>685</td>
<td>0</td>
<td>807</td>
</tr>
<tr>
<td>Generation</td>
<td>4359</td>
<td>4947</td>
<td>314</td>
<td>9620</td>
</tr>
<tr>
<td>SMTCoq integration</td>
<td>1839</td>
<td>3403</td>
<td>1119</td>
<td>6361</td>
</tr>
</tbody>
</table>

Finally, for \( \eta \)-instructions, the correctness comes directly from the simulation relation \( \approx \). 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 \( \eta \)-instruction.
is given in Section 6.3. The correctness of the SMT solver states that, if it finds that a negated\(^4\) 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 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 \((p_a \rightarrow_L p_b)\), taken to be the implication as defined by Łukasiewicz three-valued logic \(L_3\) [Borowski 1970], 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, \(L_3\) defines an implication where \(\frac{1}{2} \rightarrow 1 \equiv 1\), making it possible to formulate tautologies.

Validating \(\forall i,j. P\ coh (i,j)\) is straightforward: it corresponds directly to checking the coherence relation on the predicates, where the logical implication is translated into \(\rightarrow_L\). 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 \(p_a \rightarrow_L p_b \rightarrow_L \neg p_b\). For a node \(j\), and for \(m,n \in \text{preds}(j)\), the query \((P_{l,m} \rightarrow_L P_{l,n} \rightarrow_L \neg P_{l,n}) \land (P_{l,m} \rightarrow_L P_{l,n} \rightarrow_L \neg P_{l,m})\) proves the mutual exclusivity of predicates \(P_{l,m}\) and \(P_{l,n}\) 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 \(\phi\)-instruction to \(\mu\) or \(\gamma\)-instructions depends on whether the \(\phi\)-instruction is at a loop header or not. We use an efficient loop-header checker [Blazy et al. 2013; Bourдонcle 1993] that we can safely trust: its correctness does not affect the soundness of the translation. Similarly, we do not need to formally establish that \(\eta\)-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 \((\text{Inop instructions})\) 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 \(\mu\)-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 \(\gamma\)-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 SMTCooq

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 SMTCooq [Keller 2019], which includes a certificate checker for the veriT [Bouton et al. 2009] SMT solver for its internal SMT formulas. The main use-case of SMTCooq 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 SMTCooq formulas. In turn, we need to prove this translation sound. SMTCooq 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 SMTCooq 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

\(^4\)This negation is defined as setting 1 to 0, and both 0 and \(\frac{1}{2}\) to 1.
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. There are many ways in which the this 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, 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 made more efficient.

7 Related Work

There are various, different informal definitions of GSA in the literature. GSA was first introduced by Ottenstein et al. [1990] in 1990 as part of the program dependence web (PDW), which was inspired by the extended SSA form developed by Alpern et al. [1988]. It was later refined by Campbell et al. [1993], 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 [Ferrante et al. 1987], 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 [1994] 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 [1995a,b] 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 [Tarjan 1981]. 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 [Arenaz et al. 2008; Derrien et al. 2020; Ding et al. 2014; Oancea and Rauchwerger 2015; Tristan et al. 2011]. None of these transformations are formally verified. Among them, Tristan et al. [2011] 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 [Keller 2019]. 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 an 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 [Fleury 2020; Fleury and Weidenbach 2020].

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-formed properties similar to the SSA well-formedness from CompCertSSA.

Our semantics expresses the meaning of $\gamma$- and $\eta$-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 [Herklotz et al. 2022].

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


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