Mechanizing conventional SSA for a verified destruction with coalescing
Delphine Demange, Yon Fernandez de Retana

To cite this version:
Delphine Demange, Yon Fernandez de Retana. Mechanizing conventional SSA for a verified destruction with coalescing. 25th International Conference on Compiler Construction, Mar 2016, Barcelona, Spain. hal-01378393

HAL Id: hal-01378393
https://hal.archives-ouvertes.fr/hal-01378393
Submitted on 14 Oct 2016

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Mechanizing Conventional SSA for a Verified Destruction with Coalescing

Delphine Demange
University of Rennes 1 / IRISA / Inria, France
delphine.demange@irisa.fr

Yon Fernandez de Retana
University of Rennes 1 / IRISA / Inria, France
yon.fernandez-de-retana@irisa.fr

Abstract
Modern optimizing compilers rely on the Static Single Assignment (SSA) form to make optimizations fast and simpler to implement. From a semantic perspective, the SSA form is nowadays fairly well understood, as witnessed by recent advances in the field of formally verified compilers. The destruction of the SSA form, however, remains a difficult problem, even in a non-verified environment. In fact, the out-of-SSA transformation has been revisited, for correctness and performance issues, up until recently. Unsurprisingly, state-of-the-art compiler formalizations thus either completely ignore, only partially handle, or implement naïvely the SSA destruction. This paper reports on the implementation of such a destruction within a verified compiler. We formally define and prove the properties of the generation of Conventional SSA (CSSA) which make its destruction simple to implement and prove. Second, we implement and prove correct a coalescing destruction of CSSA, à la Boissinot et al., where variables can be coalesced according to a refined notion of interference.

This formalization work extends the CompCertSSA compiler, whose correctness proof is mechanized in the Coq proof assistant. Our CSSA-based, coalescing destruction removes, on average, more than 99% of introduced copies, and leads to encouraging results concerning spilling during post-SSA register allocation.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification - Correctness Proofs; F.3.1 [Logics and meanings of programs]: Specifying and Verifying and Reasoning about Programs - Mechanical verification

General Terms Languages, Verification

Keywords Verified Compilers, conventional SSA, coalescing

1. Introduction
The Static Single Assignment (SSA) form [11] is an intermediate representation of code where variables are statically assigned exactly once. Thanks to the considerable strength of this property, the SSA form simplifies the definition of many optimizations, and improves their efficiency, as well as the quality of their results. It is therefore not surprising that modern compilers (e.g., GCC or LLVM) rely heavily on the SSA form. The powerful properties of SSA are enabled by the uniqueness of variable definition points. To ensure this criteria on non-linear control-flow graphs (with branches and junction points), SSA uses dedicated instructions, placed at junction points. For a junction point with \( n \) predecessors, the so-called \( \phi \)-functions, of the form \( x := \phi(x_1, x_2, \ldots, x_n) \), select, at run-time, according to the control-flow graph, the right definition to use among all definitions \( x \), reaching that junction point. The corresponding \( x_i \) is then assigned to \( x \). All the \( \phi \)-functions at a same junction point are considered as a whole, and interpreted with a parallel semantics. If \( \phi \)-functions are indeed the key ingredient of SSA, they are not directly available as machine instructions, hence the need to go out of the SSA form. The standard way of integrating SSA-based techniques in a compilation chain is to (i) first convert to the SSA form, then (ii) perform SSA-based code analyses and optimizations, and (iii) re-convert back to a non-SSA representation of code.

Destructing SSA. The destruction of the SSA form (i.e., \( \phi \)-functions elimination) can be seen as a way of compiling \( \phi \)-functions, or implementing them with regular instructions of the language at hand, namely, copies. Eliminating \( \phi \)-functions in a non-naive way (without introducing too many copies) is a notoriously difficult problem. Indeed, the out-of-SSA transformation has kept being revisited, for correctness and performance issues, from its introduction in the early 90’s up until recently. We give a brief summary of these revisions, in chronological order. A more comprehensive summary can be found in [13].

The initial destruction proposed by Cytron et al. [11] consists essentially in introducing a copy \( x := x_i \) at each \( i \)-th predecessor of a junction point. Copies are then coalesced using a Chaitin-style coalescing [10]. This destruction is then identified by Briggs et al. [8] as incorrect in the presence of critical edges in the control-flow graph, after aggressive value-numbering or copy propagation (the lost-copy problem). Other cases of bugs can arise after copy folding, if the parallel semantics of \( \phi \)-function blocks is not carefully considered (the swap problem). They propose two new algorithms to correct these issues.

The work of Sreedhar et al. [17] constitutes a real progress in the understanding of the essence of the SSA destruction. They identify a subclass of SSA, Conventional SSA (CSSA), in which the elimination of \( \phi \)-functions is, so-to-say, obviously correct. In fact, CSSA is defined in [17] as the SSA form in which all \( \phi \)-related variables can be coalesced together without changing the semantics of the program. They observe that, while the conversion of code to SSA ensures a CSSA form, many optimizations break this

* This work was supported by Agence Nationale de la Recherche, grant number ANR-14-CE28-0004 DISCOVER.

Copyright notice will appear here once 'preprint' option is removed.

[2016/10/12]
property (including copy propagation and common sub-expression elimination based on value-numbering \([7]\)). Hence, they propose to
destroy SSA by first converting to CSSA, inserting fresh copies to
ensure a \(\phi\)-congruence property: all variables related (transitively)
by \(\phi\)-functions should have disjoint live-ranges. They progressively
refine the copy insertion strategy so as to minimize the number of
inserted copies.

More recently, Boissinot et al. \([5]\) revisit the problem, with a
resolute emphasis on the need for a conceptually simple destruction
of SSA, to ensure both the correctness, and the performance of
the destruction. To ensure the correctness of the destruction, they
propose to first generate CSSA following the most naive copy in-
sertion strategy of Sreedhar et al. \(\text{(we will come back to this}
information in more details). The destruction phase is then seen as
a classic and generic coalescing problem, in which the notion of non-
interference is refined to include, in addition to liveness information,
some value information, which can be easily computed or ap-
proximated in SSA. Interestingly, the copies they insert are \(\text{parallel}
copies\), to mimic the way \(\phi\)-function blocks operate, and to allow
for more coalescing opportunities. Parallel copies remaining after
the coalescing are sequentialized in a final step. Boissinot et al. \([5]\)
make a clear case for correctness consideration, pointing out some
subtle inaccuracies in \([17]\) related to the liveness information used
to check interferences between variables. They even provide some
high-level proof sketches about the correctness of the CSSA genera-
tion and non-interferences of \(\phi\)-related variables.

**SSA in Verified Compilers.** From a semantic perspective, SSA is
nowadays fairly well understood, as witnessed by recent advances
in the field of formally verified compilers. The earliest mechanized
formalizations date back from the work of Blech et al. \([3]\). Since
then, the SSA generation has been studied in more realistic con-
texts. The CompCertSSA \([1]\) project uses a complete, verified val-
idator, for the generation of pruned-SSA, based on dominance fron-
tiers \([11]\). The Vellvm project proves in \([18, 19]\) a simplified ver-
sion of the LLVM SSA generation, based on register promotion.
Recently, the generation algorithm proposed by Braun et al. \([5]\)
has been formalized in Isabelle/HOL \([9]\).

Apart from the generation itself, some progress has also been
made on the formalization of the useful invariants and properties
of SSA that ease the reasoning when it comes to proving optimiza-
tions. This includes some semantic invariants such as strictness, ba-
sic equational reasoning, dominance-region reasoning \([1, 12, 19]\),
with concrete applications to the formal proof of Sparse Condi-
tional Propagation and Common Subexpression Elimination based
on Global-Value-Numbering based \([12]\), or Copy Propagation and
micro memory optimizations \([19]\).

The destruction of SSA, however, has not received much atten-
tion yet from the verified compilation community. In fact, even if
Sreedhar et al. \([17]\) and Boissinot et al. \([5]\) provide some high-level
arguments to justify the correctness of their SSA destruction stra-
ey, it is unclear whether the non-interference of \(\phi\)-related variables
is a sufficient condition to prove the correctness of the \(\phi\)-function
elimination, or if the transformation has to establish stronger in-
variants for corner cases not yet identified, despite the scrutiny of
compiler writers.

As a matter of fact, state-of-the-art compiler formalizations
either completely ignore, only partially handle, or implement
naively the SSA destruction. Indeed, so far, the SSA destruction
in CompCertSSA \([1, 12]\) is partial, requiring that \(\phi\)-blocks behave
the same when executed sequentially or in parallel. On programs
that do not satisfy this criterion, the compiler fails. This restriction
disallows many optimizations. Furthermore, on programs that meet
this requirement, the elimination of \(\phi\)-instructions is done naively,
with massive copy insertion, thus impacting the performance of
the subsequent register allocation phase, which in practice, results
in a substantial amount of spill/reload code. Consequently, most
of the gain that we could hope from SSA optimizations is lost in
the backend. Vellvm \([18, 19]\) does not consider this problem at
all, focusing on SSA construction and optimizations. The compiler
backend (and thus the destruction of SSA) is not included in the
verified part of the project.

**Contributions.** We argue that integrating a complete (i.e., non-
failure), non-trivial SSA destruction in a verified compiler raises
an interesting question. Indeed, proving the correctness of the
\(\phi\)-function elimination requires to establish a precision result about
the underlying liveness analysis. In contrast, most of the time,
verified compilers can confine themselves to proving the soundness
of static analyses (imprecise analysis just results in less optimized
code). In the case of SSA destruction, the situation is different: if
\(\phi\)-related variables have overlapping live-ranges, or if the liveness
analysis is too coarse to detect their disjointness, one would not be
able to prove that removing all \(\phi\)-functions preserves the program
semantics. Here, we observe that the full CSSA generation (Method
I of Sreedhar et al. and Boissinot et al.) ensures that the required
precision can be met, by inserting fresh copies for all \(\phi\)-instructions
arguments and destinations (referred to as \(\phi\)-resources in the rest of
the paper).

This paper reports on the implementation of such a destruc-
tion within the verified compiler CompCertSSA, whose correctness
proof is mechanized using the Coq proof assistant. We implement
an SSA destruction à la Boissinot et al. \([5]\). This comprises
the generation of CSSA using parallel copies, the implementation of
a variable coalescing on CSSA using a refined notion of interfer-
ence, and the sequentialization of remaining parallel copies. More
technically, we make the following contributions:

- We implement the generation of CSSA, and prove it is semantic
  preserving. Despite its conceptual simplicity (introduce fresh

![Figure 1: Overview of our SSA middle-end and SSA destruction (in grey)](image-url)
variables and parallel copies, and rename $\phi$-instructions accordingly), the proof is quite technical to complete.

- We formally prove that in the resulting CSSA code, $\phi$-resources of a same $\phi$-instruction have disjoint live-ranges. This precision result, combined with the freshness property of inserted copies, allows us to prove that, independently of any sophisticated coalescing, eliminating all $\phi$-functions of the generated CSSA code is semantics preserving.

- We implement a CSSA-based coalescing algorithm and prove its correctness. To do this, we formalize the refined notion of non-interference of $\phi$, integrating the SSA-value. To account for heuristics in the coalescing algorithm, we rely on verified a posteriori validators.

- On the practical side, the CSSA-based coalescing removes, on average, more than 99% of the (massively) introduced copies. This leads to encouraging results concerning spilling and reloading during post-SSA register allocation. This validates empirically the utility of such a coalescing destruction.

The middle-end of CompCertSSA, extended with the new SSA destruction is depicted in Figure 2. It is plugged in CompCert at the level of the RTL intermediate language, that is a 3-address, unstructured representation of code in a control-flow graph (CFG), with virtual registers. After a normalization phase of RTL, we generate from there the SSA form of the code, and perform SSA-based optimizations. The generation and optimization phases are described in Section 3. The work presented in this paper is represented inside node 4, integrating the SSA-value. After a normalization phase of the program, Program a) is the initial normalized RTL version and Program c) is obtained by propagating the copy at node 4 to node 6.

![Figure 2: Example programs. Program a) is the initial normalized RTL program. Program b) is the SSA version and Program c) is obtained by propagating the copy at node 4 to node 6.](http://compcertssa.gforge.inria.fr/node4.png)

In Section 2, we briefly recall the necessary background notions relative to verified compilation. In Section 3, we present our formalization of the SSA form (syntax and semantics), and its important property of disjoint live-ranges for $\phi$-resources. In Section 4, we present a basic destruction of CSSA, that only eliminates $\phi$-instructions, without further coalescing. We give the main structure of its proof of semantic correctness. The destruction of CSSA presented in Section 4, improves it by incorporating a refined notion of non-interference leveraging the SSA-value of variables. In Section 5, we present some experimental results measuring the effectiveness of this verified coalescing on program benchmarks. Finally, Section 6 concludes and discusses some future directions of research.

In the paper, we choose not to present results in the Coq syntax, to account for unfamiliar readers. The formal development is available online at [http://compcertssa.gforge.inria.fr/](http://compcertssa.gforge.inria.fr/).
removed, without affecting the semantics of the function, thanks to their disjoint live-ranges.

Now, consider Program c) in Figure 2. It is obtained after a simple copy propagation in Program b). In this program, removing the phi-instruction and merging $x_3$, $x_1$, and $y$ would be incorrect. Indeed, Program c) is an SSA program that is not conventional anymore: the variables $x_3$, $x_1$, and $y$ (arguments of the phi-instruction at point 6) are both live at program point 2, hence their live-range intersect, and it would be incorrect to merge variables $y$ and $x_1$. Indeed, the assignment of $x_1$ would erase the one of $y$.

The goal of the conversion to SSA is to re-establish the property of disjoint live-range for phi-resources. The idea of the algorithm favored in Boissinot et al. [5] is to introduce fresh variables and parallel copies of phi-variables at junction points and their predecessors, so that the phi-function becomes isolated. That is, for each phi-instruction $x_0 := \phi(x_1, \ldots, x_n)$ at a junction point $pc$, we introduce fresh variables $a_0, \ldots, a_n$, and do the following:

- The phi-instruction is replaced by $a_0 := \phi(a_1, \ldots, a_n)$.
- A copy $a_i := a_0$ is added to a parallel copy block at $pc$.
- A copy $a_i := x_i$ is added to a parallel copy block at the $i$th predecessor of $pc$, for all $i \in \{1, \ldots, n\}$.

In Program d) of Figure 3, fresh variables $a_0, a_1, a_2$ are introduced to replace the phi-instruction $x_3 := \phi(x_1, y)$, and parallel copies are inserted at nodes 3, 5, and 6 (depicted in dark gray). As expected, all variables $a_i$ have now disjoint live-ranges.

In the rest of this section, we present how we model the syntax, semantics, and properties of SSA. Most of its properties are inherited from the SSA intermediate language, and preserved by the CSSA generation. The distinguishing property of CSSA, live-range disjointness, is presented in Section 3.4.

### 3.1 Abstract Syntax

The syntax of a SSA function is given in Figure 4. A SSA function has a name, a list of parameters, an entry point, and a code mapping from a set of nodes to instructions (nop, arithmetic operations, copies, conditionals, ...). Each instruction carries its successors nodes. Hence, the CFG of the function is completely embedded in the graph code. The SSA function has also a phi code mapping from nodes to phi-blocks (lists of phi-instructions of the form $x := \phi(x)$), and a parcode mapping from nodes to parallel copy blocks (lists of pcopy, see Figure 3).

![Figure 3: Example programs. Program c) is an optimized SSA program, Program d) is obtained by conversion to SSA, and Program e) is obtained by eliminating phi-functions.](image)

### 3.2 Semantics

We use the following notations. We write $f \circ g(x) = f(g(x))$ for function composition. If $\vec{x}$ is a vector of variables $(x_1, \ldots, x_n)$, then for $k \in \{1, \ldots, n\}$, we write $\vec{x} \cdot k$ for $x_k$. 

Figure 4 presents selected rules defining the small-step operational semantics of SSA, expressed as a labelled transition system on execution states, whose transitions are of the form $\sigma \xrightarrow{k} \sigma'$. For the instruction set we consider here in the paper, the observable event $e$ labelling the transition will be the silent event and execution states $\sigma$ (defined in Figure 5) are simple tuples $(f, pc, rs)$ gathering the function $f$ being executed, the current program point $pc$ (a node in the code pointing the next instruction to execute), and the current local environment $rs$, (finitely) mapping variables.

1 We present here the core instruction set, abstracting many technical details from the full language, e.g., memory accesses and function calls. Our formalization does handle the full instruction set, enabling the compilation of ISO C90 / ANSI-C programs.

2 The full formalization of the language does handle non-silent transitions.
### 3.3 Unique Definitions and Strictness

Each variable of a CSSA function has a unique definition point (either in the code, or in a φ-instruction, or in a copy of a parallel copy block). In our formalization, we have to formally define this notion, and prove that it is ensured. Here, to smooth the presentation, we omit the formal definition, and will just assume a function \( \text{def}_f() : \text{reg} \rightarrow \text{node} \) computing the unique node in a function \( f \) where a variable is defined.

More interesting is the notion of strictness: in a CSSA function, each variable use must be strictly dominated by its definition point. To account for the presence of parallel copy blocks and φ-blocks at junction points and their predecessors, we use the following definition.

**Definition 1 (Strict dominance).** In function \( f \), a variable \( x \) strictly dominates node \( pc \), written \( x \succ pc \), whenever one the following holds:

- \( \text{def}_f(x) \) strictly dominates \( pc \) in the CFG of \( f \)
- \( x \) is the destination of a φ-function at point \( pc \)

### 3.4 Live-range Splitting

We turn now our attention to the important properties established by the conversion to CSSA. First, we observe that in [5], proof sketches justifying the correctness of the destruction informally rely on the precise way fresh copies are inserted to isolate each φ-functions inside a same φ-block. In particular, fresh names are introduced for each φ-functions. We will call this the φ-resources disjointness property.

**Definition 2 (φ-resources disjointness).** A CSSA function \( f \) satisfies the φ-resources disjointness property if all variables appearing in a φ-function (anywhere in \( f.\text{phicode} \)) are pairwise distinct.

In particular, this ensures that no variable will appear twice in a φ-block. Now, Definition 2 only capture that φ-functions do not conflict regarding variables names. What remains to formalize is that φ-functions are indeed isolated from the rest of the code, in terms of live-ranges. In the literature, this is what is often called the live-range splitting of CSSA conversion. We will use the following definition:

**Definition 3 (Split live-ranges).** In a function \( f \), variables \( x \) and \( y \) have disjoint live-ranges (written \( x \perp y \)) when \( x \not\in \text{live}^\text{out}_f(\text{def}_f(y)) \) and \( y \not\in \text{live}^\text{out}_f(\text{def}_f(x)) \), and \( \text{def}_f(x) \neq \text{def}_f(y) \).

In this definition, we formally define the liveness information \( \text{live}^\text{out}_f(pc) \) as an inductive predicate, characterizing the set of \( x \), in the CFG of \( f \), such that there exists a path from \( pc \) to a use point of \( x \), with no re-definition of \( x \) in between. The condition \( \text{def}_f(x) \neq \text{def}_f(y) \) is a technical requirement simplifying the reasoning about parallel-copy and φ-function blocks (see discussion below). Thanks to the way fresh copies are inserted during conversion to CSSA, this extra condition will be met, by construction, on all variables appearing in a φ-instruction.

Indeed, we can prove that the CSSA form generated by the algorithm satisfies the two above properties.

**Lemma 3.** Let \( f \) a function produced by the conversion to CSSA. Then the two following properties hold:

- \( x \) is the destination of a copy in a parallel copy block, and \( pc \) is a junction point.

Given this definition, we can prove that the following property holds:

**Lemma 2 (Strictness).** In a function CSSA \( f \), if a variable \( x \) is used at a node \( pc \), then \( x \succ pc \). Moreover, no variable is both assigned and used in a same parallel copy block.

Note that the analogous property for φ-blocks is a consequence of the general strictness in the CFG, because arguments to a φ-function at a junction point are considered used at the predecessor, as is standard in the SSA literature [11].
• $f$ satisfies the $\phi$-resources disjointness property
• for any node $pc$, such that $f.\text{fn}\_\text{phi}\_\text{code}(pc) = \phi b$, it is the case, for each $\phi$-instruction $x_0 := \phi(x_1, \ldots, x_n)$ in $\phi b$, that, for all $0 \leq i, j \leq n, i \neq j \Rightarrow x_i \perp x_j$.

Proof. The proof of this lemma essentially results from the freshness of new variables in inserted copies. Once the freshness property is proved, the proof of the non-interference goes through. We first prove that for each $\phi$-resource (i.e., source and destination of a $\phi$-function), it is defined and used at the same point. Next, as they cannot be used anywhere else, they become dead just out the junction point.

In Section 4 we rely on this lemma to prove that the elimination of $\phi$-instructions (by coalescing all variables of a single $\phi$-instruction), is correct.

Discussion. Compared to the $\phi$-congruence property of Sreedhar et al., the two above properties are stronger. With these properties, we make explicit the fact that the SSA generation (Method 1) generates short live ranges for $\phi$-resources. In contrast, the $\phi$-congruence property is global on the CFG, which, we argue, would make the reasoning more difficult. In particular, the definition of $\phi$-congruence classes [12] allows the case where $x_d := \phi(x, x_2)$ and $y_g := \phi(y, z)$ belong to a same $\phi$-block. Being able to prove that eliminating these two $\phi$-instructions is correct would require to justify that $x_d$ and $y_g$ could also be replaced by a common representative. But reasoning on their run-time value would require to consider two, a priori, completely unrelated execution paths. Boissinot et al. already pointed at that it is simpler to reason on the “naïve” CSSA generation, and that, at the same time, the subsequent coalescing algorithm successfully coalesces most of introduced copies. Our experimental results (Section 6) confirm this. We hence decide to take full advantage of the two above properties, allowing for a local reasoning at junction points.

4. Non-coalescing CSSA Destruction

In this section, we present a simple CSSA destruction, from CSSA to RTL $\phi$ function. The syntax of RTL $\phi$ is the same as CSSA, without any phi code field in the function record:

$$
tf \equiv \text{function}_{\text{RTL } \phi}$$

$$
tf ::= \begin{cases} &\text{name} = \text{id}; \quad \text{params} = \hat{x}; \\
&\text{entry} = pc; \quad \text{code} = \iota; \\
&\text{parcode} = \mu \\
\end{cases}$$

Accordingly, the semantics of RTL $\phi$ and CSSA are similar, the only difference being the semantic rule NotJP, which is simpler in RTL $\phi$ because of this absence of $\phi$-blocks.

The simple destruction only removes all $\phi$-instructions, by merging variables that appear in a same $\phi$-instruction. This allows us to characterize the essential properties that allow for elimination of $\phi$-instructions. Coalescing-based destruction will be presented in the next section as an extension of this one.

4.1 Algorithm

Here, to lighten the presentation, we will assume that CSSA functions are structurally well-formed, and satisfy Lemma 4. The algorithm for this simple destruction is the following:

\[
\text{destruct}(f) = \\
\text{let } R := \text{classes}(f) \text{ in} \\
\{ \text{name} := f.\text{name}; \\
\text{params} := \text{map } R f.\text{params}; \\
\text{entry} := f.\text{entry}; \\
\text{code} := pc \mapsto \text{map}_{\text{native}} R (f.\text{code}(pc)); \\
\text{parcode} := pc \mapsto \text{map}_{\text{par}} R (f.\text{parcode}(pc)) \}
\]

First, a mapping $R$, is computed by the function classes. It is a mapping from variables to a unique representative. Then, the destruction consists in (i) removing completely the $\phi$-code from the function, and (ii) applying the mapping $R$ (seen as a variable renaming) to all parameters and instructions of the function $f$ (in the regular instruction graph $f$ code, as well as on the parallel copy blocks graph $f$ parcode).

4.2 Properties of Variable Renaming $R$

More precisely, the computed $R$ maps each variable appearing as an argument of a $\phi$-instruction to the destination of the $\phi$-instruction. Other variables will just be mapped to themselves. Note that this computation only makes sense if the same variable does not appear as a $\phi$-argument of two distinct instructions (it could otherwise map to two different destinations). This is effectively the case in our CSSA form (by Definition 2). Another important point to note is that no liveness analysis is required for this destruction. Indeed, by construction of CSSA (Lemma 3), the live-ranges of variables used in $\phi$-functions are known to be disjoint. To sum up, we (provably) characterize the computed mapping in the following way.

Lemma 4. For each $\phi$-instruction $x := \phi(x_1, \ldots, x_n)$ of a $\phi$-block $\phi b$ at node $pc$, and all $k \in \{1, \ldots, n\}$, we have $R(x_k) = x$. For all variables not used in any $\phi$-instruction, we have $R(x) = x$.

We then prove the main property of our $R$ mapping, namely that, if a variable is indeed renamed during the destruction, then it will be renamed into a variable that does not interfere with it:

Lemma 5. For each pair of distinct variables $x, x'$ of function $f$, if $R(x) = R(x')$, then $x \perp x'$.

Proof. Such $x$ and $x'$ are necessarily $\phi$-resources of a same $\phi$-instruction, by Lemma 4. We conclude using the live-range splitting property ensured by Lemma 3.

4.3 Correctness Proof

The proof that the destruct function is semantically correct is done by exhibiting a lock-step simulation between the two functions. Intuitively, we aim to show that the execution of the two functions match, step by step. To do this, we need to maintain the invariant that the value of all the necessary variables (i.e., variables live entering current program point) is preserved by the renaming process.

Often, when reasoning about SSA code analysis and optimizations, one only needs to track a correctness invariant about the variables whose definition strictly dominates the current program point $pc$. Indeed, for many SSA-to-SSA optimizations, at all point, if the value of a variable influences program execution, we can prove that we are executing a portion of code dominated by the definition of this variable.

In the present case, this is no longer true. We need to track the preservation of values for all live variables, a strictly stronger property. Indeed, as soon as a variable becomes dead, we cannot show its value is preserved, even in a code region that is dominated by its definition, because its name could have been merged with another one. Hence, the invariant must be relaxed.

The simulation relation $\sim$ between execution states of functions $f$ and $tf$ is defined as follows:

Definition 4 (State matching $\sim$). Let $tf = \text{destruct}(f)$ be the transformed function of $f$. Then $(f, rs, pc) \sim (tf, rs', pc')$ if:

• $pc = pc'$
• $\forall x, x \in \text{live}^f(pc) \cup \text{Ddep}_f(pc) \Rightarrow rs(x) = rs'(R(x))$. 
Here, \( \text{Djp}_f(pc) \) denotes the set of variables that are defined at junction point \( pc \), i.e., in a \( \phi \)-block, or in parallel copy block at a junction point. Variables in this set are not necessarily live. Hence, we maintain the invariant for a larger set of variables. This technique allows us to work with a simpler definition of liveness at junction points, at the granularity of the CFG. We now give some highlights about the proof that the relation \( \sim \) is indeed a lock-step simulation.

**Lemma 6.** Let \( \sigma_1, \sigma_2 \) be execution states of \( f \), and suppose that \( \sigma_1 \rightarrow_{\text{CSSA}} \sigma_2 \). Let \( \sigma_1' \), an execution state of \( f \) such that \( \sigma_1 \sim \sigma_1' \). Then, there exists a state \( \sigma_2' \) such that \( \sigma_1' \rightarrow_{\text{RTL}} \sigma_2' \) and \( \sigma_2 \sim \sigma_2' \).

**Proof.** The proof is done by a case analysis on the step relation. We present here the most interesting cases only. Let \( \sigma_1 = (f, pc, rs) \), and \( \sigma_1' = ((f, pc, rs') \) two states satisfying the hypotheses of the preceding lemma.

Let us first consider the case where \( \sigma_1 \) is such that the next instruction to execute is a single copy instruction \( x ::= y \). Let now \( r \) be a variable live out of \( pc \) (that is, live entering its successor). We want to prove that local environments match after executing the copy, i.e., \( rs'[x \leftarrow y](r) = rs'[(R(x) \leftarrow R(y))(R(r))] \). There are several cases to consider:

- If \( r = x \), the proof is straightforward: we can apply to \( y \) the induction hypothesis because it is live entering \( pc \), being used in \( pc \), and not defined at \( pc \) because of the code strictness property of Lemma 2.
- If \( r \neq x \) and \( R(r) \neq R(x) \), then \( r \) was already live entering \( pc \), and the copy does not modify \( r \) or \( R(r) \), so the equality follows from the matching relation between \( rs \) and \( rs' \).
- If \( r \neq x \) but \( R(r) = R(x) \), \( r \) is as before already live entering \( pc \), but the value of \( R(r) \) could have changed. Because \( r \neq x \), \( R(r) = R(x) \) means \( r \perp x \), which is impossible, because \( r \) is live out of the definition point of \( x \).

The most difficult case is when the instruction at \( pc \) is a nop(\( pc' \)), and \( pc' \) is a junction point \( pc' \) (rule NoplP in Figure 5), which requires to simulate the execution of a parallel copy block \( pc' \) at \( pc \), followed by a \( \phi \)-block phib and a parallel copy block \( pc' \) at \( pc' \) only one step. In fact, we proceed in two steps, by proving the matching relation between local environments is preserved, but with some adjustments with respect to the variables on which the matching holds.

First, we prove a lemma that propagates the necessary intermediate invariants after application of parch. Mainly, it propagates the invariant over the local environment for variables that are live entering \( pc' \) or assigned in parch. Morally, the simulation of a block of parallel copies is similar to that of a single copy. In particular, we take advantage of the fact that, with this basic notion of interference, we know that after renaming, no variable is assigned twice, because \( x \perp y \) cannot happen for variables defined in a same point. We rely on two technical helper lemmas. The first lemma characterizes the effect of a renamed block on the representative \( R(r) \) of a variable \( r \) that is not assigned in parch.

**Lemma 7.** Let \( r \) be a variable that is not a destination in block parch. If for each copy \( x ::= y \) appearing in parch, we have \( R(r) \neq R(x) \), then \([\text{map}_{\text{mov}} \ R \text{ parch}]rs'[R(r)] = rs'[R(r)] \). The second lemma is similar, but for variables that are assigned in the block.

**Lemma 8.** Let \( r \) be a variable that is a destination of a copy \( y ::= y \) of the block parch. If for any other copy \( x ::= y \) we have \( R(r) \neq R(x) \), then \([\text{map}_{\text{mov}} \ R \text{ parch}]rs'[R(r)] = rs'[R(y)] \).

This means, in fact, that the copies of parallel copy blocks remain indeed parallel after renaming with \( R \).

We give a highlight of the proof of this invariant propagation after applying parch. Let \( r \) be a variable live entering the junction point \( pc' \), successor of \( pc \). We proceed by a case disjunction on whether the variable is assigned or not in parch.

- If \( r \) is assigned in a copy \( x ::= y \) at parch, then \( R(r) \neq R(x) \) for each copy \( x ::= y' \) of parch with \( x \neq r \), otherwise \( r \) and \( x \) would be defined at the same point, so \( r \perp x \) could not hold. We can therefore apply Lemma 8.
- If \( r \) is not assigned in parch, we also have \( R(r) \neq R(x) \) for each copy \( x ::= y' \) of parch, because \( r \) is live out of point \( pc \) (because live entering \( pc' \)), and \( x \) defined at \( pc \), so \( r \perp x \) could not hold. We can hence apply Lemma 7.

Second, we propagate further the matching between environments after executing the \( \phi \)-block phib, for variables that are live at \( pc' \) or assigned in parch or assigned at phib. Here, Lemma 4 is crucial. Indeed, the fact that in each \( \phi \)-instruction, all variables are mapped to the same representative, basically means that a \( \phi \)-instruction \( x_0 := \phi(x_1, \ldots, x_n) \) would be renamed into \( x_0 := \phi(x_0, \ldots, x_0) \), and would hence not modify the local environment. This justifies the \( \phi \)-block elimination.

At last, we achieve the propagation of the invariant after the second parallel copy block parch, using a reasoning similar to the previous one.

5. **CSSA Destruction with Coalescing**

This destruction extends the previous one by allowing to remove useless copies. In contrast, we make more use of verified \( a \text{ post} \text{riori} \) validators in place of direct proof. This approach yields the same guarantees of correctness on the output program, while having two main advantages: the proof is generally significantly simpler, but more importantly, validators are robust against fined-tuned adjustments done regarding heuristics of computations. For example, the use of coalescing priorities, such as processing the junction point before predecessors as we do (because it’s a more frequently used execution path), does not affect the proof.

### 5.1 Algorithm

The algorithm for the extended destruction is the following:

```plaintext
destructure(f) =
let live := live_analysis(f) in
let \( V_f \) := cssa_value_ext(f) in
let ninterfere := ninterfere_test(f, live, \( V_f \)) in
let (\( R, \) classes) := build_classes_ext(f, ninterfere) in
if check(R, classes, ninterfere) 
& & check_v(f, \( V_f \)) then
  \( \text{name} := f \text{name}; \)
  \( \text{params} := \text{map } R f \text{params}; \)
  \( \text{entry} := f \text{entry}; \)
  \( \text{code} := pc \mapsto \text{map}_{\text{code}} \{f (f \text{code}(pc)); \}
\text{parcode} := pc \mapsto \text{clean(map}_{\text{mov}} R \{f (f \text{parcode}(pc))\}) \}
else \text{Error}
```

7 2016/10/12
As previously, the destruction consists in computing a variable renaming $R$, and then applying it to the whole code of a function. There are however two noticeable differences.

First, $R$ is now computed by an external, untrusted OCaml program, build_classes_ext, whose result is a posteriori validated, as indicated by check, against a specification. If the validator succeeds, then we proceed to next phase. If not, then, here, the whole destruction phase fails. In practice, the validator never fails.

The details of the computation of classes and properties (ensured by the checker) are presented in Section 5.3, but they rely on a pre-computed liveness (live, computed and proved in Coq) and CSSA-value information $(V_f)$, computed in OCaml and checked by a formally proved validator check_v.

Second, in addition to applying the variable renaming and eliminate $\phi$-instructions, some copy elimination is performed (clean in renamed parallel copy blocks. We remove trivial copies of the form $x := x$, effectively eliminating copies whose variables were coalesced. We also remove redundant copies, i.e., copies with the same destinations but not necessarily trivial (see Section 5.4). In fact, we must ensure that all such redundant copies are removed before the copy serialization phase, so that parallel copy blocks satisfy the “windmill” condition of Rideau et al. [16].

5.2 Non-interference Refined with CSSA-value

Since the early work of Chaitin et al. [10], the ultimate criterion to decide whether two variables can be merged is that they have disjoint live-ranges, or the same value at execution time. The latter condition, as proposed by Boissinot et al. [5], can be easily approximated using the notion of SSA-value. The SSA-value is a symbolic approximation (i.e., an expression) of the run-time value of an SSA variable. This can be easily computed thanks to the unique definition property of SSA. Here, as in [5], we particularize the SSA-value to variable copies. More formally:

**Definition 5 (CSSA-value).** The CSSA-value function $V_f$ of a SSA function $f$, is a function from variables to variables that satisfies the following properties:

- If $x := y$ is a copy (parallel or not) of $f$, then $V_f(x) = V_f(y)$.
- If $x := \text{insn}(y)$ where insn is not a copy, then $V_f(x) = x$.

We compute such a function $V_f$ in OCaml by a depth-first traversal of the control flow graph of $f$ as in Boissinot et al. [5]. The resulting $V_f$ is validated a posteriori by a verified validator check_v, ensuring that $V_f$ satisfies Definition [5].

The main property of a CSSA-value (as specified in Definition 5), on which we rely to prove the correctness of the destruction, is that a variable and its CSSA-value evaluate to the same value at run-time, at program points dominated by the definition point of the variable.

**Lemma 9.** For each reachable execution state $\sigma = (f, pc, rs)$ of the SSA function $f$, then for each variable $r$ such that $r \succ pc$, we have $rs(r) = rs(V_f(r))$.

**Proof.** The proof is done by induction on the number of steps of the execution reaching that state, with a dominance-based reasoning, as presented in [12], and using the property that $\text{def}_f(V_f(r))$ dominates $\text{def}_f(r)$, which is proved, using Definition 5 by induction on the CFG paths leading to $\text{def}_f(r)$.

As a direct corollary, we get:

**Lemma 10.** For each reachable execution state $\sigma = (f, pc, rs)$ of a SSA function $f$, for all pairs of variables $x, y$ of $f$ such that $x \succ pc$, $y \succ pc$ and $V_f(x) = V_f(y)$, we have $rs(x) = rs(y)$.

This lemma will be useful in the correctness proof of the transformation, to prove that it is correct to merge two variables with equal CSSA-values. For example, if a variable $r$ is merged with a distinct variable $x$ appearing in a copy $x := y$, and $r$ is live out of this point, we want to be able to prove that the value of the variable issued from the fusion of $x$ and $r$ is unchanged by this copy.

We can now define the extended notion of non-interference from Boissinot et al.:

**Definition 6.** Two variables $x$ and $y$ of a SSA function $f$ do not interfere, written $x \perp_f y$, if $x \perp y$, or if $V_f(x) = V_f(y)$.

The non-interference check $\text{intersect}$ is implemented in Coq, using the liveness and CSSA-value precomputations, and is proved formally correct with respect to Definition 6.

5.3 Properties of Variable Renaming $R$

The coalescing algorithm build_classes_ext starts by putting all variables appearing in a same $\phi$-instruction in a same class. The other variables of the program are at this point in singleton classes. Then, each block of parallel copies is traversed and for each copy $x := y$, we check (with intersect) whether we can merge the coalescing classes of $x$ and $y$, that is, if there is no interference between a variable of the coalescing class of $x$ and that of $y$. The resulting $R$ associates to each variable $r$, the representative of its class. The computation of coalescing classes yields two maps: a map from a set of representative to coalescing classes (classes), and the map $R$. The validation a posteriori of coalescing classes check ensures that, in each class, variables do not interfere, and that each variable $r$ belongs to the class of $R(r)$ (hence, does not interfere with any other member of the class). The current validator check is a quadratic algorithm (in the size of the class), that checks interference between all pairs of variables in a class.

The main properties established on $R$ by the validator can be stated as follows:

**Lemma 11.** For each pair of variables $r, r'$ of a SSA function $f$, if $R(r) = R(r')$, then $r \perp r'$. For each $x, y$ in a same $\phi$-instruction, we have $R(x) = R(y)$.

This lemma is analogous to Lemma 4. Now we have to take into account the CSSA-value, and the fact that $R$ can act on variables not used in a $\phi$-instruction, but appearing in parallel copies, can be added by the algorithm to the coalescing class of variables of a $\phi$-instruction, if it does not interfere with all the variables in the class. Two distinct $\phi$-instructions of distinct $\phi$-blocks can be merged if each pair of variables forming them do not interfere. As a future extension, two $\phi$-instructions $x := \phi(x_1, \ldots, x_n)$ and $y := \phi(y_1, \ldots, y_m)$ of a same $\phi$-block could eventually be merged if for all $k \in \{1, \ldots, n\}$, $V_f(x_k) = V_f(y_k)$, but with our current definition of interference, the variables $x$ and $y$ interfere.

5.4 Correctness Proof

The proof of the extended destruction follows the same architecture as the basic version of Section 4. We exhibit a lock-step simulation between the source and target functions. Interestingly, the chosen simulation relation ($\sim$) is the same. The difference in the proof lies in the specification of the variable renaming, which now includes the possibility of coalesced variables to have equal CSSA-values, and also on the elimination of trivial parallel copies of the form $x := x$. This, in turn, requires us to generalize some lemmas, in particular the ones characterizing the semantics of parallel copy blocks.
Case of a Simple Copy Instruction. Let \( x := y \) be a copy at \( pc \), and \( r \) a variable live at \( pc \). We have to prove \( rs[x ← y](r) = rs'[(R(x) ← R(y))(R(r))] \). The only case that differs significantly is the one with \( r \neq x \) and \( R(r) = R(x) \). Then \( r \) is already live entering \( pc \), but the value of \( R(r) \) could have changed. In fact, \( R(r) = R(x) \) means that \( x \) and \( r \) do not interfere, but because \( r \) is live out of \( pc \), this means that \( r \) and \( x \) do not interfere (Lemma 11) thanks to same CSSA-value, so \( r \) and \( y \) have same CSSA-value (a copy propagates the CSSA-value). The property of CSSA-value of Lemma 10 then states that \( rs(r) = rs(y) \), but we also have \( rs(y) = rs'(R(y)) \) (because \( y \) is live at \( pc \)), which allows to conclude.

Case of a Junction Point Predecessor. In this case, \( pc \) branches to a junction point \( pc' \) (rule NopJP). As in Section 4, we need to prove intermediate propagations of the \( \sim \) invariant after applying \( \phi \)-elim and \( \phi \)-parcb.

While CSSA invariants guarantee that no variable is assigned more than once, this is no longer the case in the renamed block: several \( \phi \)-destinations could be mapped to the same \( \phi \)-value, not because they have disjoint live-ranges, but because they could have the same \( \phi \)-value. Indeed, two copies \( x := y \) and \( x' := y' \) of parcb can be such that \( R(x) = R(x') \) if \( V_f(x) = V_f(x') \), which happens if \( V_f(y) = V_f(y') \) (by propagation of CSSA-value through copies). We hence proved general versions of Lemmas 7 and 8.

Lemma 12. Let \( r \) be a variable that is not a destination in block parcb. If for each copy \( x := y \) appearing in parcb, we either have \( R(r) \neq R(x) \) or \( rs'(R(r)) \neq rs'(R(y)) \), then:

\[
\text{[clean(map}_{\phi \text{mov}} R \text{parcb}]} \text{[rs'}(R(r)) = rs'(R(r))]
\]

In particular, now, in the case where \( R(r) = R(x) \), the lemma implies that the value of \( R(r) \) is unchanged.

Lemma 13. Let \( r \) be a variable that is a destination of a copy \( r := y \) of the block parcb. If for any other copy \( x := y' \), we have \( R(r) \neq R(x) \) or \( rs'(R(r)) = rs'(R(y')) \), then:

\[
\text{[clean(map}_{\phi \text{mov}} R \text{parcb}]} \text{[rs'}(R(r)) = rs'(R(y'))]
\]

Another point to note is that \( \text{[clean(map}_{\phi \text{mov}} R \text{parcb}]} \) represents the block parcb to which not only \( R \) has been applied to all variables, but also in which trivial copies of the form \( x := x \) have been removed. This removal is tricky to justify: if \( x \) appears in the same parallel copy block as destination of another copy \( x := y \), we need to know that the sources \( x \) and \( y \) have the same value (otherwise the parallel copy block would not be well-defined semantically).

The proof of invariant propagation after applying parcb follows the same schema as in the non-coalescing case, but we have to take into account the CSSA-value. For example, in the case where \( r \) is in a copy \( r := y \) at \( parcb \) and there is another copy \( x := y \) at \( parcb \) such that \( R(r) = R(x) \), we want to prove that \( rs'(R(y)) = rs'(R(y')) \) as before. Because they are defined in the same node, we cannot have \( r \perp x \) as in the non-coalescing case, so \( r \) and \( x \) have equal CSSA-values. By CSSA-value propagation across copies, \( y \) and \( y' \) have equal CSSA-value too: as these are live entering \( pc \), we can apply Lemma 13.

We can conclude applying Lemma 13. Other cases are similar.

6. Experimental Results

The proofs presented so far establish the semantic correctness of the destruction, but it does not account for the quality of coalescing. In this section, we evaluate the practical gain of integrating such a destruction in CompCertSSA.

To do so, we rely on the Coq extraction mechanism, that produces efficient OCaml code from the formalization, and get an executable version of the CompCertSSA verified C compiler.

Benchmark Programs. We use a set of test programs taken from the CompCert test suite, the SPEC2006 benchmarks and WCET-related reference benchmarks. These represent around 192,600 lines of C code, each program ranging from thousands of lines of C code, to tens of thousands. This comprises small programs such as some cryptographic functions as aes.c, bigger programs such as compression algorithms, or the first order logic prover \textit{spass} (http://www.spass-prover.org/) with tens of thousand lines.

Evaluation Criteria. We want to evaluate the impact of the coalescing destruction on the generated code, as independently as possible from the rest of the compiler chain. Typically, assessing the overall performance of the CompCertSSA compiler would be premature, and out of the scope of the present paper.

Hence, we will be comparing three different, but similar compilers: CompCert\(^*\) without the SSA middle-end, CompCertSSA using the old, partial destruction (deSSA) and CompCertSSA using the new, complete, and coalescing destruction (CSSA).

One possible criterion for measuring the effects of coalescing is the execution time of compiled programs. However, programs in the benchmark suite we use have, for most of them, too short execution times to make the observed variations significant. Hence, we use more fine-grained measurements: namely the number of remaining copies (for the two variants of CompCertSSA) and the impact on spilling and reloading on the subsequent register allocation phase (for CompCert and the two CompCertSSA variants).

Number of Remaining Copies. Results are given in Figure 5. For each program, we give the number of parallel copies (i) introduced when converting to CSSA, (ii) introduced by the previous destruction of SSA (deSSA), and (iii) remaining in RTL after coalescing.

On average, more than 99% of the introduced copies are eliminated. More precisely, on average over all the copies introduced by all the programs from Figure 6, we get 99.96% of eliminated copies. And on average, 99.93% of copies are eliminated, by file. As for \textit{spass}, we get nearly 100% of eliminated copies over a total of 24574 introduced copies (only 4 copies remain). Results are similar for \textit{bzip2} and \textit{raytracer}.

This high percentage of copy elimination can be explained by the fact that the current CompCertSSA optimizations do not introduce many interferences in \( \phi \)-blocks. Indeed, this would have made the old destruction phase fail to compile the resulting programs.

One reason for copies not to be coalesced, with the current state of CompCertSSA, is that two distinct \( \phi \)-arguments of a same \( \phi \)-function in SSA could be live out of the junction point (so, typically, after a copy propagation). Then, these two variables interfere for liveness reasons, and cannot be put in the same coalescing class. Optimizations making a variable appear multiple times in a \( \phi \)-block could also lead to non-eliminated copies: for example, if a variable \( r \) appears in SSA as an argument of two \( \phi \)-instructions in a same block, \( x_d := \phi(x) \) and \( y_d := \phi(y) \), then \( r \) will appear as a source of two parallel copies in CSSA, and only one copy will eventually be eliminated. Indeed, eliminating both copies would require us to merge the classes of \( x_d \) and \( y_d \). Currently, this is not possible with our non-interference definition. However, as mentioned in Section 5.3 this limitation could be overcome by extending the CSSA-value propagation through \( \phi \)-functions.

Spilling and Reloading. Figure 7 gathers results on the number of spilling and reloading instructions for programs \textit{spass}, \textit{raytracer}, and...
bzip2 and a sum over the CompCert test suite. For spass, the number of spills decreases from 4300 to 3453 (a 20% improvement), and the number of reloadings from 6742 to 5430 (a 19% improvement). On bzip2 spills decrease from 2410 to 450 (81% improvement). On raytracer, we observe a slight regression from 261 to 271 (4% regression). We observe a decrease from 842 spills to 576 on the CompCert test suite (32% improvement). Overall, the new destruction allows to reach spilling and reloading rate that are close to CompCert 2.1 (without SSA): for example 7% less spilling for spass, but 15% more spilling for bzip2. Note this remains a significant improvement when compared to the 520% of extra spilling we use to obtain with the old, non-coalescing destruction.

Discussion. The results in terms of remaining copies are satisfying, and overall, the improvement in spilling and reloading is significant, compared to the previous destruction. In fact, the coalescing destruction leads to results comparable to CompCert 2.1. The minor regression in spilling on raytracer means however that there remains room for improvement. In particular, it would be interesting to study whether this is due to a too aggressive coalescing, increasing live ranges [14]. In the end, our empirical validation could of course benefit from a more thorough benchmarking process, but these preliminary results are promising.

7. Conclusions and Future Work

In this paper, we have presented a formalization of the SSA destruction, based on the conversion to conventional SSA, and a subsequent coalescing algorithm. First, we identify the necessary properties of SSA for a non-coalescing destruction, using the live-range splitting property of φ-resources and disjointness of variables in φ-blocks. We then proved an extension of this destruction, that uses a coalescing algorithm to eliminate useless introduced copies. The results in terms of eliminated copies are very satisfying and match our expectations, with more than 99% of eliminated copies, and an overall spilling that decreased significantly.

Development size. Table[1] gives an overview of the Coq development size, as given by program coqc, organized thematically. We give the total number of source code lines, and the number of proof lines. This provides an idea of the proof effort, although these numbers are not always linearly correlated with the actual difficulty.

![Figure 6: Coalescing results](image1)

![Figure 7: Spilling (top) and Reloading (bottom)](image2)
of the proofs: some proofs are long and tedious but without any real technical difficulty (like the CSSA generation), and some other proofs could probably get along with some factorizations.

### Validation and direct proof

We used two approaches to verify our transformations: direct proof of Coq algorithms, and validation a posteriori of non-proved algorithms by formally proved validators, which gives the same correctness guarantees on the generated code. Validation a posteriori sometimes leads to simpler and more maintainable proofs, because the validator proof depends less on the actual algorithm. This is especially true for the coalescing algorithm, where we have truly benefitted from the validation approach during the fine-tuning of coalescing heuristics.

### Non detailed transformations

Some proofs we do not detail in this paper are the semantic preservation of the CSSA generation from SSA, and the guarantee that CSSA-invariants hold. The main time-consuming task was to reason about the freshness of introduced variables and proving the unique definition property, which required us to prove that in each φ-block each variable appeared once, but also that two distinct φ-blocks had not common variables. Another phase not detailed here is the sequentialization of parallel computation: it is currently done by a standard dataflow analysis, but as shown by Boissinot et al. [4], in the particular case of SSA, it is possible to use a more specific computation that exploits structural data structures that are not optimal, often introducing a logarithmic factor in the algorithms, e.g., when using tree data structures instead of hash tables. We also occasionally compute data invariants hold. The main time-consuming task was to reason about the freshness of introduced variables and proving the unique definition property, which required us to prove that in each φ-block each variable appeared once, but also that two distinct φ-blocks had not common variables. Another phase not detailed here is the sequentialization of parallel copy blocks. This part is essentially a proof reuse of the formalization by Rideau et al. [16].

### Future work

In this work, we mainly focus on formalizing and characterizing the properties justifying the correctness of the SSA destruction, leaving compilation time and memory usage as secondary objectives. First, using Coq forced us to rely on purely functional data structures that are not optimal, often introducing a logarithmic factor in the algorithms, e.g., when using tree data structures instead of hash tables. We also occasionally compute data structures that could be virtualized. For example, Boissinot et al. [5] and Sreethar et al. [17] propose techniques to introduce copies only when they are really needed, instead of introducing them first, and eliminating them a posteriori. In some cases, we implement non optimal algorithms. In particular, coalescing class computation and validation are currently quadratic in the size of the classes. However, there exist linear algorithms to check for interference between two classes of variables. For example, Boissinot et al. [5] propose such an algorithm that relies on a specific ordering of variables in classes. A future step for improving our destruction could be to formalize such an algorithm. Another example is the liveness computation: it is currently done by a standard dataflow analysis, but as shown by Boissinot et al. [4], in the particular case of SSA, it is possible to use a more specific computation that exploits structural properties of SSA, and that gives better results on memory usage, while staying competitive in terms of compilation time.

The SSA destruction we study in this work is done prior to register allocation. While this strategy is the most frequent in real world compilers, others approaches have been proposed. In particular, Hack et al. [14] propose to perform register allocation directly on the SSA form. But spilling can lead to a variable needing to be reloaded at several points in the program, thus breaking the unique definition property of SSA, and requiring an on-the-fly re-conversion to SSA. While this approach is interesting, it seems currently hard to adopt in a formally verified compiler such as CompCertSSA.

### References


### Table 1: Lines of source code

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Total</th>
<th>Proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSSA and RTLg Syntax and semantics</td>
<td>793</td>
<td>15</td>
</tr>
<tr>
<td>CSSA generation</td>
<td>8372</td>
<td>5545</td>
</tr>
<tr>
<td>Non-coalescing destruction</td>
<td>3689</td>
<td>2723</td>
</tr>
<tr>
<td>Destruction with coalescing</td>
<td>6612</td>
<td>4567</td>
</tr>
<tr>
<td>De-parallelization</td>
<td>2657</td>
<td>1453</td>
</tr>
<tr>
<td>Utility lemmas</td>
<td>1407</td>
<td>822</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>23530</strong></td>
<td><strong>15125</strong></td>
</tr>
</tbody>
</table>