Verified Construction of Static Single Assignment Form

Sebastian Buchwald Denis Lohner Sebastian Ullrich
Karlsruhe Institute of Technology, Germany
{sebastian.buchwald, denis.lohner}@kit.edu sebastian.ullrich@student.kit.edu

Abstract

Modern compilers use intermediate representations in static single assignment (SSA) form, which simplifies many optimizations. However, the high implementation complexity of efficient SSA construction algorithms poses a challenge to verified compilers. In this paper, we consider a variant of the recent SSA construction algorithm by Braun et al. that combines simplicity and efficiency, and is therefore a promising candidate to tackle this challenge. We prove the correctness of the algorithm using the theorem prover Isabelle/HOL. Furthermore, we prove that the algorithm constructs pruned SSA form and, in case of a reducible control flow graph, minimal SSA form. To the best of our knowledge, these are the first formal proofs regarding the quality of an SSA construction algorithm. Finally, we replace the SSA construction of the CompCertSSA project with code extracted by Isabelle's code generator to demonstrate the applicability to real world programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Verified compilers, Isabelle/HOL, SSA form, SSA construction

1. Introduction

Many modern compilers employ intermediate representations in *static single assignment* (SSA) form for their optimization phases, including GCC [1], Java HotSpot [14], libFirm [2] and LLVM [15]. SSA form makes def-use relations explicit and simplifies many optimization algorithms such as common sub-expression elimination or invariant code motion [5].

For the construction of SSA form, the literature offers multiple algorithms that differ in implementation complexity, runtime efficiency and output size. Production compilers choose a fast algorithm that produces only necessary ϕ functions at the cost of increased implementation complexity. This algorithm class includes the one by Cytron et al. [11] as well as the one by Sreedhar and Gao [22], which produce minimal SSA form and can be augmented with liveness information to additionally produce pruned SSA form.

Contrarily, verified compilers need algorithms that are simple enough to prove their correctness. An example is the algorithm by Aycock and Horspool [6] that uses a two-phased approach. First, it places ϕ functions for each variable in every basic block. In the second step, it iteratively removes trivial ϕ functions, i.e., ϕ functions that reference at most one value other than itself. Aycock and Horspool provide a hand-written proof that their approach produces minimal SSA form for reducible control flow graphs (CFGs).

More recently, Braun et al. [9] presented an efficient SSA construction algorithm that produces minimal and pruned SSA form. The algorithm constructs SSA form directly from the AST, which makes it suited for compilers with intermediate representations that rely on SSA form, like libFirm [2]. Moreover, the algorithm uses only basic data structures if minimality is required only for reducible CFGs. Thus, it is a promising algorithm for verified compilers. In this paper, we use the theorem prover Isabelle/HOL to reason about the correctness and quality of the SSA construction algorithm. Our contributions are:

- A machine-checked correctness proof for a functional variant of Braun et al.'s SSA construction algorithm.
- A machine-checked proof that the algorithm produces pruned SSA form.
- A machine-checked proof of the general fact that removing all trivial ϕ functions leads to minimal SSA form for reducible CFGs. This implies that the algorithm by Braun et al. as well as the algorithm by Aycock and Horspool produce minimal SSA form for reducible CFGs.
- The integration of our work into the verified C compiler CompCertSSA [8] that allows us to evaluate the code extracted by Isabelle's code generator on real world programs.

Section 2 of this paper defines SSA form and describes various SSA construction algorithms including Braun et al.'s. In Section 3, we define our abstract Isabelle/HOL representation of SSA form. Section 4 presents our functional implementation of Braun et al.'s algorithm in Isabelle/HOL that is suitable for formal verification. The proofs are separated into correctness (Section 5) and minimality (Section 6). In Section 7, we discuss the integration into the CompCertSSA compiler and its runtime efficiency. Section 8 discusses related work and Section 9 concludes this paper.

2. Background

In this section, we review the definition of SSA form and various SSA construction algorithms and discuss the selection of Braun et al.'s algorithm as the paper's subject. Furthermore, we list the Isabelle tools and frameworks used in the paper.

2.1 SSA Form

SSA form was introduced and defined by Rosen et al. [19]. It is based on the *control flow graph* (CFG) of the source program. A CFG is in SSA form if every variable contained in the CFG's basic blocks is statically assigned at most once. In other words, SSA form does not track the mutable *variables* of the source program

CC'16, March 1718, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892211

but merely their *values*. In this paper, we will call these identifiers *SSA values*, whereas *variables* will always refer to identifiers in the original CFG.

In the general case, SSA form cannot be achieved by renaming variables alone. If the definitions of two distinct SSA values reach the same basic block, a ϕ *function* may be inserted in order to decide which definition to use. A ϕ function has as many parameters as its basic block has predecessors. When control flow enters the basic block through an edge, the ϕ function evaluates to the corresponding parameter. We will call SSA values defined by ϕ functions ϕ *definitions* and all other definitions *simple definitions*.

2.2 SSA Construction Algorithms

Cytron et al. [11] first described an SSA construction algorithm that is both efficient and leads to only moderate increase in program size. Specifically, the algorithm is based on the observation that ϕ functions are only needed at the iterated dominance frontiers of basic blocks containing a variable definition. Cytron et al. call a valid set of ϕ function placements that adheres to this condition *minimal*. They present an algorithm to compute the dominance frontiers and ϕ placements and, using a set of sample programs, argue that the transformation is effectively linear for practical inputs.

The total number of placed ϕ functions, however, is not necessarily minimal yet. An SSA CFG (a CFG in SSA form) may further be *pruned* to eliminate ϕ functions that (transitively) only have other ϕ definitions as users. Choi et al. [10] extend Cytron et al.'s algorithm with liveness analysis to produce both pruned and minimal SSA form.

Sreedhar and Gao [22] use *DJ-graph*s, an extension of dominator trees, to "obtain, on average, more than five-fold speedup over [Cytron et al.'s] algorithm." It is used in e.g. LLVM for this reason [3].

In contrast to the complex dominance frontiers computation, Aycock and Horspool [6] propose a simple construction algorithm consisting of two phases: The *RC phase* ("really crude phase") constructs a maximal SSA form by inserting ϕ functions for every variable into every basic block. The *minimization phase* then iteratively eliminates a special class of ϕ functions we will call *trivial* ϕ functions using the following rules:

1. A ϕ definition of form $x = \phi(x, \ldots, x)$ can be trivially removed.
2. A ϕ definition of form $x = \phi(v_1, \ldots, v_n)$ where there is another SSA value y such that $v_1, \ldots, v_n \in \{x, y\}$ can be removed if all occurrences of x are subsequently replaced by y.

While this class of redundant ϕ functions has already been described by Rosen et al. in the original SSA paper, Aycock and Horspool first proved that exhaustively applying these rules yields minimal SSA form for reducible CFGs. They conclude that the algorithm constructs SSA form on par with Cytron et al.'s algorithm for most constructs in programming languages and that the overhead of constructing the maximal SSA form is measurable but negligible when compared to the total compilation time.

Like Aycock and Horspool, Braun et al. [9] describe an SSA construction algorithm they call "simple" because, in contrast to Cytron et al.'s algorithm, it does not depend on additional analyses. However, they also aim for efficiency and demonstrate that their algorithm's performance is on par with LLVM's implementation of Sreedhar and Gao's algorithm. An important property that leads to this result is that there is no need to construct a non-SSA CFG prior to executing the algorithm. Instead, the algorithm can construct SSA form directly while building the CFG and additionally employ simple optimizations on the fly.

```
readVariable(variable, block):
    if currentDef[variable] contains block:
        return currentDef[variable][block]
    if |block.preds| = 1:
        # Optimize the common case of one predecessor:
        # No phi needed
        val ← readVariable(variable, block.preds[0])
    else:
        # Break potential cycles with operandless phi
        phi ← new Phi(block)
        currentDef[variable][block] ← phi
        # Determine operands from predecessors
        for pred in phi.block.preds:
            phi.appendOperand(readVariable(variable, pred))
        val ← tryRemoveTrivialPhi(phi)
    currentDef[variable][block] ← val
    return val

tryRemoveTrivialPhi(phi):
    same ← None
    for op in phi.operands:
        if op = same || op = phi:
            continue # Unique value or self−reference
        if same ≠ None:
            # phi merges at least two values: not trivial
            return phi
        same ← op
    if same = None:
        # phi is unreachable or in the start block
        same ← new Undef()
    # Remember all users except phi itself
    users ← phi.users.remove(phi)
    # Reroute all uses of phi to same and remove phi
    phi.replaceBy(same)

    # Recurse for all phi users, which might have become trivial
    for use in users:
        if use is a Phi:
            tryRemoveTrivialPhi(use)
    return same
```

Figure 1. Pseudo code of the algorithm due to Braun et al. The code has been slightly edited for clarity. Calling `readVariable` for every variable and every block it is used in yields pruned and (for reducible CFGs) minimal SSA form.

Whereas the dominance frontier-based algorithms work from the variables' definition sites by computing their iterated dominance frontiers, Braun et al.'s algorithm starts at their use sites, implicitly ensuring prunedness of the output. In its most basic version, the algorithm searches backwards for reaching definitions of the variable used. If the search encounters a join point (a basic block with more than one predecessor), it places a ϕ function in the basic block and recursively continues the search in all predecessor blocks to look up the new ϕ function's parameters. The search stops at basic blocks containing a simple definition of the variable or a ϕ definition placed earlier. Therefore, the search never visits the same join point twice, ensuring termination. Braun et al. then extend the algorithm to remove trivial ϕ functions described by Aycock and Horspool's rules above, yielding minimal SSA form for reducible CFGs. Finally, by computing strongly connected components (SCCs) of redundant ϕ functions, Braun et al. achieve minimal SSA form for all inputs; however, we will focus on the reducible case for this paper. Figure 1 shows the relevant parts of the pseudo code implementation provided by Braun et al. [9].

```
locale graph =
fixes αe :: 'g ⇒ ('node × 'edgeD × 'node) set
 and αn :: 'g ⇒ 'node list
 and inEdges :: 'g ⇒ 'node ⇒ ('node × 'edgeD) list
assumes αn-correct:
  αn g ⊇ fst ‘ αe g ∪ snd ‘ snd ‘ αe g
assumes αn-distinct:
  distinct (αn g)
assumes inEdges-correct:
  set (inEdges g n) = {(f,d). (f,d,n) ∈ αe g}
begin
 definition predecessors g n ≡ map fst (inEdges g n)
end
```

Figure 2. Interface of the graph context used in the formalization. Some intermediary locales have been omitted for clarity.

2.3 Isabelle and Isabelle/HOL

The proofs of this paper are written in and verified by the interactive theorem prover Isabelle2015. We use Isabelle's default object logic called Isabelle/HOL. It reuses common syntax for most mathematical operations and basic functional notations.

Isabelle/HOL's basic types include *nat* and *bool*. Algebraic data types may be defined using the keyword **datatype**, e.g.

datatype *'a option = Some 'a | None*

Here *'a* is a type variable.

Non-recursive functions can be defined by **definition**, recursive ones by **function**. The function *the* :: *'a option* ⇒ *'a* unwraps an *option* value; the result of *the None* is the special value *undefined*. The function *Option.map* :: (*'a* ⇒ *'b*) ⇒ *'a option* ⇒ *'b option* applies a function to *Some* values and leaves *None* values unchanged. Type *'a* ⇀ *'b* denotes a partial function and is an alias for *'a* ⇒ *'b option*.

Further HOL types are pairs (*'a* × *'b*, unpacked by functions *fst* and *snd*), sets (*'a set*) and lists (*'a list*). Lists are either the empty list [] or the result of prepending an element x to another list xs, denoted by *x* # *xs*. *xs* ! *i* is the *i*th element of *xs* (0-indexed), the list concatenation operator is called @. The value of the list comprehension [*f x* . *x* ← *xs*] is the result of applying *f* to every item of *xs*. *f* ‘ *A* is the image of set *A* under the function *f*.

In propositions, free variables are implicitly universally quantified. For bundling assumptions, ⟦*P*; *Q*⟧ ⟹ *R* can be used as a shortcut for *P* ⟹ *Q* ⟹ *R*.

2.4 Graph Framework and Isabelle Locales

For representing the CFG structure, we use an abstract graph framework by Nordhoff and Lammich [18]. It contains a hierarchy of *locales*, an Isabelle abstraction mechanism comparable to ML's module system [7]. A locale *fixes* some set of operations (also called its *parameters*), then describes *assumptions* these operations should fulfill. Lemmas and definitions declared in the context of the locale may use any of these assumptions. Existing locales can be imported into new ones by use of the + operator.

Figure 2 shows a locale expression that describes the proof context for graphs we use in the formalization. The operations declared here are αe and αn, the projection of a graph onto its edges and its nodes, respectively, and *inEdges*. The latter function returns a list of edges instead of a set, so there is some non-specified but fixed linear order on incoming edges. We will later use this order to make a connection between incoming edges and positional arguments of a ϕ function.

By *interpreting* a locale, that is, instantiating its parameters with actual functions that fulfill the locale's assumptions, Isabelle substitutes the parameters in the locale's set of lemmas and definitions

```
locale CFG = graph +
fixes defs :: 'g ⇒ 'node ⇒ 'var set
 and uses :: 'g ⇒ 'node ⇒ 'var set
assumes defs-uses-disjoint: defs g n ∩ uses g n = {}
assumes defs-finite: finite (defs g n)
assumes uses-finite: finite (uses g n)

locale CFG-wf = CFG +
assumes def-ass-uses:
  ⟦v ∈ uses g m; g ⊢ Entry g−ns→m⟧
  ⟹ ∃ n ∈ ns. v ∈ defs g n
```

Figure 4. Definition of locales *CFG* and *CFG-wf*

accordingly and exports them to the outer context. Our formalization contains two interpretations of the locales. One that describes the CFGs of a simple While language from [25], and a generic interpretation that is used to produce a working OCaml version of the formalized algorithm. The first interpretation shows that the locale's assumptions are sane, while we use the second together with Isabelle's code generator to export the signatures of the locale's parameters as well as the instantiated definitions to other functional languages. For each exported definition, the code generator takes the definitional lemmas tagged as *code equations* and converts them to the specified target language. Our formalization contains code equations for all definitions needed by the functional variant of Braun et al.'s algorithm, ensuring the extracted OCaml version uses the same code the correctness proofs works on.

We extend the graph locale with two further definitions: a predicate $g \vdash n{-}ns{\to}m$ that holds iff there is a path (a list of nodes) *ns* from *n* to *m* in *g* (with *n* and *m* being included in *ns*), and a fixed node *Entry g* that is assumed to have no incoming edges, but to reach all other nodes. The full definitions can be found in the formalization [24].

3. CFG and SSA Representation

When formalizing an intermediate representation (IR) transformation such as SSA construction, one can choose to do so using a specific intermediate representation, which is mostly useful when verifying a concrete compiler implementation such as in the CompCertSSA project, or use an abstract representation that can be applied to a broad class of languages. We choose the latter option by abstractly modeling the minimum structure needed by the algorithm as locales, then later instantiate the model to show that all locale assumptions can be fulfilled.

Going back to the algorithm's definition, we see that while the algorithm uses the CFG's structure, it is not concerned about any edge labels. Of the basic blocks' contents, it is only interested in use sites to start its search from, and def sites to terminate the search at. Therefore, we abstract away the concrete content (such as a list of statements) of a block to merely the set of its def and use sites; Figure 3 shows these simplifications applied to a sample program. Since this abstraction loses the order of execution inside the block, basic blocks with intra-block data dependencies may have to be split prior to applying it.

Figure 4 shows the formalization of the CFG in the *CFG* locale, including a *definite assignment* assumption in a derived locale *CFG-wf*, meaning every variable must be defined before it is used. While this assumption is not needed for the definition of the construction algorithm, it will be necessary for both the minimization proof and the semantics preservation proof.

By extending the locale with ϕ functions, we finally define an abstract locale for SSA CFGs (Figure 5). Because the left-hand sides of ϕ definitions in a node must be mutually distinct, the set of all ϕ definitions in the graph can be characterized as a map-

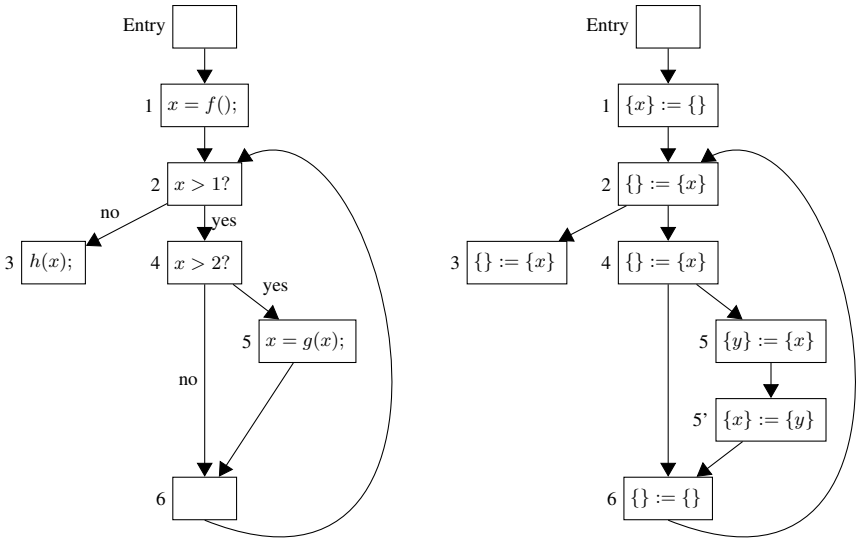

Figure 3. Original CFG and simplified CFG. The simplification removes all edge annotations and replaces block contents with the respective def and use sets, separated by := symbols. Block 5 has to be split and a temporary variable y has to be introduced to preserve the order of execution.

ping from *'node* × *'val*, a pair of containing node and the left-hand side, to the right-hand side, i.e., an argument list *'val list*. Function *phiDefs* extracts all identifiers of a block defined by ϕ functions. Assumption *phis-wf* ensures the length of an argument list corresponds to the number of predecessors of the containing block. Assumptions *simpleDefs-phiDefs-disjoint* and *allDefs-disjoint* describe the core property of SSA form: Every SSA value is defined at most once.

As with the *CFG* locale, we demand definite assignment in a derived locale *CFG-SSA-wf*. The only difference to *CFG-wf* is that *defs* and *uses* are replaced with *allDefs* and *allUses*, respectively. Finally, we derive from both *CFG-wf* and *CFG-SSA-wf* in order to describe a transformation from the former to the latter as a new locale *CFG-SSA-Transformed* (Figure 6). In it, we assume the existence of a function *var* that maps every SSA value to the original variable it belongs to; we will need this relation for the semantics preservation proof. We also use it to state that the transformation should not change the simple definition or use sets (modulo *var* application). Since we implicitly assume the transformation not to change the graph structure by using the same parameters for both locales, this means that the only changes the transformation is allowed to perform are adding ϕ functions and splitting variables into distinct SSA values.

Lastly, *CFG-SSA-Transformed* includes two assumptions that are satisfied by most SSA construction algorithms, including Cytron et al.'s. The first assumption states that a ϕ function's parameters must belong to the same original variable as the definition itself. In other words, the set of ϕ *nets* – ϕ functions that are transitively connected – is partitioned by the *var* function. The second assumption describes *Conventional SSA* (CSSA, [23]) Form: An SSA CFG is in CSSA form iff all variables of the same ϕ net are interference-free, i.e., a path from a definition to its use site may not contain another definition from the same ϕ net. The CSSA assumption is most important because Cytron et al.'s definition of SSA minimality, on which our minimality proof is based, implicitly requires it. Note, however, that integrating further on-the-fly optimizations into Braun et al.'s algorithm such as copy propagation (which we do not in this paper) may in general break both of these assumptions.

locale *CFG-SSA* = *CFG* +
fixes *phis* :: *'g* ⇒ *'node* × *'val* ⇀ *'val list*
assumes *phis-finite*:
  finite (*dom* (*phis g*))
assumes *phis-wf*:
  phis g (*n*,*v*) = *Some args*
  ⟹ *length* (*predecessors g n*) = *length args*
assumes *simpleDefs-phiDefs-disjoint*:
  defs g n ∩ *phiDefs g n* = {}
assumes *allDefs-disjoint*:
  n ≠ *m* ⟹ *allDefs g n* ∩ *allDefs g m* = {}
begin
 definition *phiDefs g n* ≡ {*v*. (*n*,*v*) ∈ *dom* (*phis g*)}
 definition *allDefs g n* ≡ *defs g n* ∪ *phiDefs g n*
 definition *phiUses g n* ≡ {*v*. ∃ *n'* ∈ *successors g n*.
  ∃ *v'* ∈ *phiDefs g n'*.
  (*n*,*v*) ∈ (*zip* (*predecessors g n'*) (*the* (*phis g* (*n'*,*v'*))))}
 definition: *allUses g n* ≡ *uses g n* ∪ *phiUses g n*
end

Figure 5. Definition of locale *CFG-SSA*

4. Construction

The imperative pseudo-code implementation of Braun et al.'s algorithm in Figure 1 combines the basic version achieving prunedness and its first extension achieving minimality for reducible inputs. While preserving the algorithm's structure and output, we instead define a functional version that is more amenable to formal verification by splitting the algorithm into these two parts:

1. transforming a valid CFG into a valid, pruned SSA CFG, that is, the functionality of `readVariable` in the pseudo code without the invocation of `tryRemoveTrivialPhi`.
2. transforming any reducible SSA CFG into a minimal one by removing any trivial ϕ functions, i.e., the functionality of `tryRemoveTrivialPhi`.

This mirrors Aycock and Horspool's two-phase algorithm except, of course, that our first phase is not "crude" at all but computes pruned SSA form instead of maximal SSA form.

```
locale CFG-SSA-Transformed =
  CFG-SSA-wf αe αn invar inEdges Entry defs uses phis +
  old: CFG-wf αe αn invar inEdges Entry oldDefs oldUses
fixes var :: 'val ⇒ 'var
assumes oldDefs-correct:
  oldDefs g n = var ` defs g n
assumes oldUses-correct:
  oldUses g n = var ` uses g n
assumes allDefs-var-disjoint:
  ⟦v ∈ allDefs g n; v' ∈ allDefs g n; v ≠ v'⟧
  ⟹ var v' ≠ var v
assumes phis-same-var:
  ⟦phis g (n,v) = Some vs; v' ∈ vs⟧
  ⟹ var v' = var v
assumes conventional:
  ⟦g ⊢ n−ns→m; n ∉ tl ns; v ∈ allDefs g n; v ∈ allUses g m;
   x ∈ tl ns; v' ∈ allDefs g x⟧
  ⟹ var v' ≠ var v
```

Figure 6. Definition of locale *CFG-SSA-Transformed*. Note that both base locales are instantiated with identical graph parameters, but different *defs* and *uses* parameters.

4.1 Basic Construction

For the first phase, we can build a very declarative implementation by further splitting it into two different tasks: looking up the places where to insert ϕ functions, and looking up the parameters of a single ϕ function. The following algorithm collects the set of the insertion points for a variable v starting from a node n.

```
function phiDefNodes-aux
 :: 'g ⇒ 'var ⇒ 'node list ⇒ 'node ⇒ 'node set where
  phiDefNodes-aux g v unvisited n =
    if n ∉ set unvisited ∨ v ∈ defs g n then {}
    else fold (op ∪)
      [phiDefNodes-aux g v (removeAll n unvisited) m .
        m ← predecessors g n]
      (if length (predecessors g n) ≠ 1 then {n} else {})
```

Parameter *unvisited*, a list of unvisited nodes, is passed along to ensure termination (unlike sets, Isabelle/HOL lists are implicitly finite). Finding a simple definition of v terminates the search, too. Otherwise, the function recurses into all predecessors of n and collects the respective result sets using a fold over the union operator. n is included if it is a join point.

If the recursion visits the entry node, it places an empty ϕ function in the node. This can be interpreted as "garbage in, garbage out": To create this situation, there must be a path in the input CFG from the entry node to a use site without a corresponding definition on it. The definite assignment assumption of locale *CFG-wf*, on which our correctness proof is based, disallows such a path, but it is still useful for the algorithm to be executable without this assumption.

We can now describe the locations of all ϕ functions of v using another fold over all use sites.

```
definition phiDefNodes :: 'g ⇒ 'var ⇒ 'node set where
  phiDefNodes g v ≡
    fold (op ∪)
      [phiDefNodes-aux g v (αn g) n . n ← αn g, v ∈ uses g n] {}
```

Having identified the locations of all ϕ functions, the remaining task is to look up their parameters. But first, we need a scheme for representing SSA values. A common scheme is to add a sequential index to all occurrences of a variable. However, in the simplified case of the formalization, for every original variable and node in the SSA CFG there may be at most one definition of an SSA value. Therefore, an SSA value can be uniquely identified by the following pair:

```
type-synonym ('node, 'var) ssaVal = 'var × 'node
```

Looking up a ϕ function's parameters is a simple backwards search that terminates at definitions in the input CFG and at join points. The definition of *phiDefNodes* guarantees that reached join points indeed contain a ϕ function for v.

```
function lookupDef
 :: 'g ⇒ 'node ⇒ 'var ⇒ ('node, 'var) ssaVal where
  lookupDef g n v =
    if v ∈ defs g n then (v,n)
    else case predecessors g n of
      [m] ⇒ lookupDef g m v
    | - ⇒ (v,n)
```

Figure 7 shows the actual output generated by the final algorithm and exemplifies the use of *ssaVal*.

With this, the three function parameters *defs*, *uses* and *phis* of the *CFG-SSA* locale can be instantiated with new functions using *ssaVal* instead of *'var*.

```
definition defs' :: 'g ⇒ 'node ⇒ ('node, 'var) ssaVal set where
  defs' g n ≡ (λv. (v,n)) ` defs g n
definition uses' :: 'g ⇒ 'node ⇒ ('node, 'var) ssaVal set where
  uses' g n ≡ lookupDef g n ` uses g n
definition phis' :: 'g ⇒ ('node, ('node, 'var) ssaVal) phis where
  phis' g ≡ λ(n,(v,m)).
    if m = n ∧ n ∈ phiDefNodes g v
    then Some [lookupDef g m v . m ← predecessors g n]
    else None
```

Function *defs'* simply lifts definitions in the input CFG to *ssaVal*s. *uses'* maps the original use sets via *lookupDef*. *phis'* first verifies the SSA value (v, m) is indeed a ϕ function in the node n, then looks up v's definition in each predecessor node.

These three functions constitute a valid *CFG-SSA* interpretation:

```
interpretation braun-ssa:
  CFG-SSA αe αn invar inEdges Entry defs' uses' phis'
```

To prove this theorem, the assumptions of both *CFG* and *CFG-SSA* have to be shown based on *defs'*, *uses'* and *phis'*. The proofs follow directly from the definitions.

To show that the SSA CFG is pruned, we first define what a *live* SSA value is. If an SSA value is used outside of a ϕ function, it is marked as live. Incrementally, if an SSA value marked as live is defined by a ϕ function, we mark its parameters as live. This iterative definition, which reflects Isabelle's implementation of inductive definitions as least fixed points, prevents an SSA value from being considered live simply because it is its own user.

```
definition phiArg g v v' ≡
  ∃ n vs. phis g (n,v) = Some vs ∧ v' ∈ vs
inductive liveVal :: 'val ⇒ bool where
  val ∈ uses g n ⟹ liveVal g val
| ⟦liveVal g v; phiArg g v v'⟧ ⟹ liveVal g v'
definition pruned ≡ ∀ g n val. val ∈ phiDefs g n ⟶ liveVal val
```

It follows that the constructed SSA CFG is pruned.

```
theorem phis'-pruned: braun-ssa.pruned
```

The proof works by showing that every SSA value (v, m) is live, where v is a variable and m a node in the result of *phiDefNodes-aux* for any use site n of v. By definition of *phiDefNodes-aux*, m and n must be connected by an interference-free path. By reverse induction over the path, the proof then iteratively shows the liveness of every ϕ function on it, up to and including (v, m) itself.

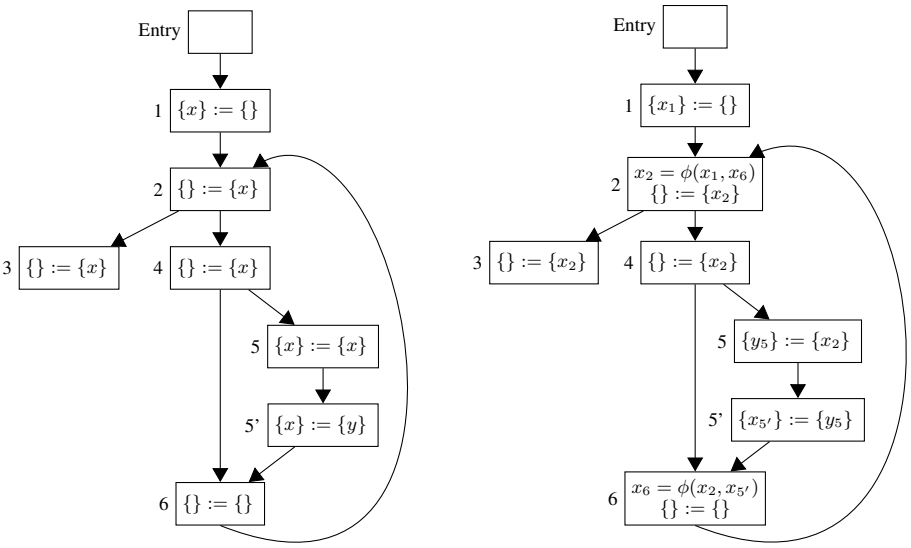

Figure 7. Simplified CFG and the SSA CFG generated by the algorithm. SSA values are printed by appending the number of the node the variable is defined in as an index to the variable name.

4.2 Removing Trivial Phi Functions

Braun et al. demonstrate the removal of trivial ϕ functions as an example of an on-the-fly IR optimization that can be combined with their basic construction algorithm. Whenever they complete a ϕ function, they check whether it is trivial. If it is, they remove it from the CFG and recursively check all of its former users, which, they argue, results in all trivial ϕ functions being removed in the end.

With an immutable CFG data structure, this approach becomes infeasible. Instead, we implement the transformation of removing a single trivial ϕ function, then close over it via a fixed-point iteration. This functional version is trivially correct in removing all trivial ϕ functions and, since the total number of ϕ functions is finite, always terminates. Additionally, the transformation is completely independent from the basic SSA construction, so we can implement it for arbitrary SSA CFGs and prove its properties independently from the preceding proofs.

Given a graph with a non-empty set of trivial ϕ functions and a user-defined function *chooseNext* that returns an arbitrary element of it[1], we remove the element by adapting a non-recursive `tryRemoveTrivialPhi` of the pseudo code. Figure 8 shows the definition of the new *uses* and *phis* functions – no other locale parameters are modified by the transformation. This also tells us what the fixed-point iteration should operate on: a pair of *uses* and *phis* functions, as shown in Figure 9. All other functions can be used from the outer, original context. Also note that we do not consider trivial ϕ functions $\phi(v) = (v, \ldots, v)$ that only depend on themselves as defined by Aycock and Horspool. One can easily see that such ϕ functions do not satisfy the definite assignment assumption. More formally, successfully completing the minimality proof later on will implicitly tell us that our definition of trivial ϕ functions is exhaustive.

For the proof, most work is related to proving that all SSA CFG invariants are maintained by the transformation. Many of these proofs are based on the important fact that the basic block of a trivial ϕ function is strictly dominated by the basic block of its other argument. Together, these proofs show that the functions from Figure 9 constitute a valid *CFG-SSA-Transformed* instantiation that is also free of trivial ϕ functions.

interpretation *notriv*:
  CFG-SSA-Transformed $\alpha e\ \alpha n$ *invar inEdges Entry*
  oldDefs oldUses defs uses'-all phis'-all
theorem $\neg$*notriv.redundant g*

Furthermore, the construction maintains prunedness.

theorem *pruned g* $\Longrightarrow$ *notriv.pruned g*

Intuitively, this result is immediately clear because we are only removing ϕ functions and reconnecting other ϕ functions so that they are still live. However, upon formalizing the proof, our inductive definition of the liveness predicate turned out to be unsuitable: To show that reconnecting a ϕ function works, we need to know the list of intermediary ϕ functions that connect it to a use site. Therefore, we need an equivalent but 'richer' liveness predicate exclusively for this proof.

inductive *liveVal'* :: *'g* $\Rightarrow$ *'val list* $\Rightarrow$ *bool*
where
  val $\in$ *uses g n* $\Longrightarrow$ *liveVal' g* [*val*]
 | $[\![$*liveVal' g* (*v*#*vs*); *phiArg g v v'*$]\!] \Longrightarrow$ *liveVal' g* (*v'*#*v*#*vs*)

lemma *liveVal-liveVal'*: *liveVal g v* $\longleftrightarrow \exists$ *vs. liveVal' g* (*v*#*vs*)

With the new predicate in place, the theorem can be proved by induction over the length of vs.

5. Proof of Correctness

While the previous sections proved that the algorithm indeed constructs pruned SSA form, they did not state any connection between the data flow in the input CFG and in the SSA CFG. This section provides a proof of the two CFGs being semantically equivalent according to a custom small-step semantics reflecting Cytron et al.'s SSA definition [11]:

> Along any control flow path, consider any use of a variable V (in the original program) and the corresponding use of V_i (in the new program). Then V and V_i have the same value.

[1] for example, the least element of it for variable types with a linear order

```
context CFG-SSA-wf begin
definition trivialPhi g v v' ≡ v' ≠ v ∧
  ∃n. phis g (n, v) = {v,v'} ∨ phis g (n, v) = {v'}
definition trivial g v ≡ ∃v'. trivialPhi g v v'
definition redundant g ≡ ∃v. trivial g v
end

context CFG-SSA-Transformed-notriv begin
definition substitution g ≡
  THE v'. trivialPhi g (chooseNext g) v'
definition substNext g ≡
  λv. if v = chooseNext g then substitution g else v
definition uses' g ≡ λn. substNext g ` uses g n
definition phis' g ≡ λ(n,v). if v = chooseNext g
  then None
  else map-option (map (substNext g)) (phis g (n,v))
end
```

Figure 8. Removing a single trivial ϕ function

```
definition cond g ≡ λ(u,p).
 CFG-SSA-wf.redundant ∝n inEdges defs (uses(g:=u)) (phis(g:=p)) g
definition step g ≡ λ(u,p). (
 CFG-SSA-Transformed-notriv.uses' ∝n defs (uses(g:=u)) (phis(g:=p)),
 CFG-SSA-Transformed-notriv.phis' ∝n defs (uses(g:=u)) (phis(g:=p)))
definition substAll g ≡ while (cond g) (step g) (uses g, phis g)
definition uses'-all g ≡ fst (substAll g)
definition phis'-all g ≡ snd (substAll g)
```

Figure 9. Fixed-point iteration. By referencing the locale functions explicitly, they can be supplied with adjusted *uses* and *phis* parameters.

```
definition step g m s v ≡
  if v ∈ oldDefs g m then Some m else s v
inductive sem
 :: 'g ⇒ 'node list ⇒ ('node, 'var) state ⇒ bool (- ⊢ -⇓-)
where
  g ⊢ Entry g−ns→last ns
  ⟹ g ⊢ ns⇓(fold (step g) ns Map.empty)

definition ssaStep g m i s v ≡
  if v ∈ defs g m then Some m
  else case phis g (m,v) of
    Some phiParams ⇒ s (phiParams ! i)
  | None ⇒ s v
inductive ssaSem
 :: 'g ⇒ 'node list ⇒ ('node, 'val) state ⇒ bool (- ⊢ -⇓s -)
where
  g ⊢ [Entry g]⇓s(ssaStep g (Entry g) 0 Map.empty)
| ⟦g ⊢ ns⇓s s; last ns = predecessors g m ! i⟧
  ⟹ g ⊢ (ns@[m])⇓s(ssaStep g m i s)
```

Figure 10. Definition of CFG and SSA CFG semantics

Small-step semantics define the *state* of the program after executing a single instruction or, in our case, a single basic block. In accordance to Cytron et al.'s definition, we characterize state as a mapping of live variables to their respective current values. Since the abstracted *CFG* locale has no notion of 'values', they can equivalently be replaced by the node each variable was last defined in.

```
type-synonym ('node, 'var) state = 'var ⇀ 'node
```

The small-step semantics on the original CFG simply records all definitions in a basic block and is extended to paths over blocks by a fold. The SSA semantics additionally executes phi functions,

```
definition pathsConverge g x xs y ys z ≡
  g ⊢ x−xs→z ∧ g ⊢ y−ys→z ∧
  length xs > 1 ∧ length ys > 1 ∧ x ≠ y ∧
  (∀j ∈ {0..< length xs}. ∀k ∈ {0..<length ys}. xs ! j = ys ! k
    ⟶ j = length xs − 1 ∨ k = length ys − 1)

definition pathsConverge' g x xs y ys z ≡
  g ⊢ x−xs→z ∧ g ⊢ y−ys→z ∧
  length xs > 1 ∧ length ys > 1 ∧ x ≠ y ∧
  butlast xs ∩ butlast ys = {}

lemma
  pathsConverge g x xs y ys z ⟷ pathsConverge' g x xs y ys z

definition necessaryPhi g v z ≡
  ∃n ns m ms. pathsConverge g n ns m ms z ∧
    v ∈ oldDefs g n ∧ v ∈ oldDefs g m
definition cytronMinimal g ≡
  ∀n v. phis g (n, v) ≠ None ⟶ necessaryPhi g (var v) n
```

Figure 11. Definition of path convergence and minimality due to Cytron et al.

for which it needs access to the traversed edge, necessitating an inductive definition. The full definitions are shown in Figure 10.

Recalling Cytron et al.'s definition, we expect a path to a use site in the transformed SSA CFG to yield the same state as in the original CFG. Indeed, the assumptions of the *CFG-SSA-Transformed* locale are strong enough so that we can show the semantics preservation for *any* instance of the locale.

```
theorem equiv: ⟦g ⊢ ns⇓s; g ⊢ ns⇓s s'; v ∈ uses g (last ns)⟧
  ⟹ s (var v) = s' v
```

We show by induction over the evaluation path that for any SSA value v and node n from which there is a path to a use site of v, the CFG state s of *var* v is equal to the SSA CFG state s' of v. Prominently, the proof uses the CSSA assumption in order to show that the SSA value v' last defined on the path with *var* $v' =$ *var* v is indeed v itself.

6. Proof of Minimality

In their definition of SSA form, Cytron et al. give an explicit description of places at which to insert ϕ functions: For any two non-empty paths that each start from an assignment to a variable V in the original program and converge at the same node, that node must hold a ϕ function for V. We will call this the *convergence property*. Cytron et al. call an SSA program minimal if it only contains ϕ functions satisfying this condition, i.e., minimality is the converse of the convergence property. Figure 11 shows our formalization of the definition, including Cytron et al.'s original definition of path convergence and a slightly simpler equivalent definition we used in the proofs.

For reducible SSA CFGs without trivial ϕ functions, Braun et al. give a proof that this minimality definition is satisfied by explicitly examining converging paths. In contrast, Aycock and Horspool's proof builds on both instance relationship graphs and Cytron et al.'s general minimality proof via iterated dominance frontiers, all of which would first have to be formalized separately. Therefore, we decided to formalize Braun et al.'s proof on top of our SSA locales.

```
theorem ⟦reducible g; ¬redundant g⟧ ⟹ cytronMinimal g
corollary reducible g ⟹ notriv.cytronMinimal g
```

The compact handwritten proof by Braun et al. expands to just under 1000 lines of Isabelle proof code, making it the largest proof section of this paper. Apart from few missing special cases, the basic structure of the proof did not need to be changed. There were

definition *serial-on A r* ≡
  ∀ *x* ∈ *A*. ∃ *y* ∈ *A*. (*x*,*y*) ∈ *r*

theorem *serial-on-finite-cycle*:
  ⟦*serial-on A r*; *A* ≠ {}; *finite A*⟧
  ⟹ ∃ *a* ∈ *A*. (*a*,*a*) ∈ r^+

lemma *convergence*:
  ⟦*necessaryPhi g* (*var v*) *n*; *g* ⊢ *n*−*ns*→*m*; *v* ∈ *allUses g m*;
   ∀ *x*. *x* ∈ *set* (*tl ns*) ⟶ *v* ∉ *allDefs g x*; *v* ∉ *defs g n*⟧
  ⟹ *phis g* (*n*,*v*) ≠ *None*

Figure 12. Additional formalizations for Braun et al.'s minimality proof

two bigger additions that were necessary to complete the formalized proof, the first of which was needed to show that an infinite chain of ϕ functions eventually leads to a cycle (which holds because our CFG is finite). The relation-based proof is factored out into a separate theory file and uses the formalization of the pigeonhole principle from the Isabelle standard library.

The second addition was more important: The handwritten proof implicitly uses the convergence property itself, without giving a proof for it. While Cytron et al. base SSA form on the convergence property via a "definition by construction," we would have to prove it by use of our more abstract SSA form assumptions. However, we realized that the convergence property as defined only holds for unpruned SSA form such as the output of Cytron et al.'s algorithm. Therefore we had to extend it with the additional assumption of the join node being reachable from a use site. Fortunately, the minimality proof succeeds even with this weaker version. See Figure 12 for the full definitions of the additions.

7. Applicability and Evaluation

Our formalization is based on locales that require an instantiation to fulfill certain assumptions. In order to show that these assumptions are consistent, we instantiated our formalizations for a simple While language. We also provide a generic instantiation from which we extract an implementation of our algorithm in OCaml.

For more realistic test programs, we integrated a generic version of the extracted code into the CompCertSSA compiler [8], which is written in Coq [17] and also extracted to OCaml. The original CompCertSSA compiler uses an unverified implementation of the SSA construction algorithm by Cytron et al. that merely computes the locations of ϕ functions, followed by a verified validator. The validator computes the ϕ functions' arguments and makes the compilation fail if the ϕ function locations are incorrect. As Cytron et al.'s algorithm only computes minimal SSA form, the CompCertSSA compiler also contains a verified live variable analysis to achieve pruned SSA form. The result of this analysis is used in the validator as well as in the ϕ placement pass. We will discuss the advantages and drawbacks of the verified validation approach in the related work section.

We replaced the algorithm by Cytron et al. and the subsequent validation code with the extracted Isabelle code. Due to missing interoperability between the two theorem provers, we had to add some unverified OCaml glue code to match the expected data structures. Assuming the glue code is correct and the invariants of both formal works are compatible, we obtain a formally verified compiler using our formalization of the SSA construction pass [4]. The version published in [24] is correct even without these assumptions, because it retains the live variable analysis and the validation pass, trading some amount of runtime for the formal guarantees on the Coq side.

We use the test programs of the CompCertSSA compiler to measure the performance of the modified construction pass. Table 1 shows the runtime of the extracted code. For comparison, the figure also contains the runtime of the SSA construction of CompCertSSA that bases on Cytron et al.'s algorithm. We applied some data refinement and optimizations until the runtime of our algorithm was on par with the runtime of the original SSA construction of CompCertSSA. Because of irreducible control flow, our algorithm generates 0.15% more ϕ functions on the test data than Cytron et al.'s.

We also created a family of simple test programs to investigate the scalability of the different algorithms. The test programs follow the following scheme but differ in the number of *if* statements:

```
int main(int argc, char *argv[]) {
    if (argc) { }
    ...
    if (argc) { }

    return 0;
}
```

Given n if statements, Braun et al.'s algorithm constructs n trivial ϕ functions that can be removed on the fly. Thus, the expected runtime of an efficient imperative implementation is $\mathcal{O}(n)$. We fit the runtime results for $100, 200, \ldots, 10000$ if statements against an $\mathcal{O}(n^a \cdot log(n)^b)$ function. For our extracted code, this results in $\mathcal{O}(n \cdot log(n)^2)$. Using the same tests for the implementation of Cytron et al.'s algorithm and the validator also results in $\mathcal{O}(n \cdot log(n)^2)$. On the same hardware as in Table 1, the SSA construction for the test program with 10000 if statements runs in 0.56 s for our algorithm and in 0.82 s for that of CompCertSSA.

8. Related Work

While this paper describes the first formal minimality proof of any SSA construction algorithm and the first formal correctness proof of Braun et al.'s algorithm known to the authors, other SSA construction algorithms have been verified correct by use of theorem provers.

Mansky and Gunter [16] describe and verify a simple SSA construction algorithm as an example of their generic Isabelle framework for verifying compiler optimizations. The algorithm is defined in TRANS, a language proposed by Kalvala et al. [13] for describing optimizations as CFG transformations, and operates on programs in a simplified programming language. Since the paper focuses on the use of the TRANS language and the proof framework, a concise algorithm that yields neither pruned nor minimal SSA form was chosen; it places ϕ functions wherever two different definitions of a variable are reachable (which is not necessarily at a dominance frontier). Mansky and Gunter prove that the algorithm is partially correct, that is, the return values of the original and the transformed program are the same. However, they do not consider termination. Our correctness proof shows neither explicitly because both return statements and branch conditions have been abstracted away, but the result of state equivalence at every point in the program implies both return value and termination equivalence.

Zhao et al. [27] also present a framework for formally verifying optimizations and prove the correctness of an implementation of Aycock and Horspool's algorithm to provide an example. The implementation is based on Vellvm [26], a formal semantics of LLVM's intermediate representation for the theorem prover Coq [17]. They also prove that return values are unchanged and additionally show that a transformed program terminates and does not enter a stuck state if the original program does the same.

Barthe et al. [8] use a validation approach within the CompCertSSA project: While the SSA construction itself is not verified, it is followed by a formally verified validator that checks whether

	Braun et al.					Cytron et al.				
Benchmark	Phase I	Phase II	Glue	Total	#ϕ	LV Analysis	ϕ Placement	Validation	Total	#ϕ
c	0.06 s	0.02 s	0.02 s	0.10 s	788	0.04 s	0.04 s	0.01 s	0.09 s	788
compression	0.00 s	0.02 s	0.00 s	0.02 s	594	0.01 s	0.01 s	0.00 s	0.02 s	594
raytracer	0.02 s	0.02 s	0.01 s	0.04 s	222	0.01 s	0.03 s	0.01 s	0.05 s	222
spass	0.79 s	1.08 s	0.53 s	2.41 s	15192	1.38 s	1.16 s	0.65 s	3.20 s	15168

Table 1. Runtime of different SSA construction phases in seconds on an Intel Core i7-3770 with 3.40 GHz and 16 GB RAM. For our extracted code (Braun et al.) we give the runtime of both phases: Phase I corresponds to constructing pruned SSA form, whereas Phase II is the removal of trivial ϕ functions. Column Glue gives the runtime of the (unverified) wrapper to match the data structures. The original SSA construction in CompCertSSA uses a live variable analysis (column LV Analysis) for both its phases, the unverified computation of ϕ locations using Cytron's algorithm and the verified validation of the construction. Columns Total are the total running times of the corresponding algorithms. Columns #ϕ are the total number of generated ϕ functions.

the construction produced correct SSA form and preserved the program semantics. The main advantage of the validation approach is its significantly reduced implementation effort. Furthermore, the validator is not restricted to a particular algorithm and might use a highly-tuned implementation that is too complex for direct verification. On the other hand, using a validator does not guarantee that the implementation itself is correct and terminates for all valid inputs. Moreover, if quality guarantees, such as prunedness or minimality of the SSA form, are of interest, a validator would also have to check the quality criterion. On the other hand, direct verification can offer quality guarantees without any impact on runtime performance.

Demange and Fernández de Retana [12] verify the destruction of SSA form as part of the CompCertSSA project. They first convert the SSA form to Conventional SSA (CSSA) form, and then transform the CSSA form to a non-SSA representation. For the latter transformation they provide two variants: A simple algorithm that is verified using their small-step operational semantics, and a variant with copy coalescing that uses a formally verified validator.

Schneider et al. [20, 21] present the functional intermediate language IL for verified compilers. Despite its functional interpretation, IL also provides an imperative interpretation IL/I. They state that the transformation of IL/I to IL corresponds to SSA construction, while the transformation of IL/F to IL corresponds to SSA destruction. To show the correctness of both transformations, they combine direct verification with formally verified validators.

All papers introducing SSA construction algorithms that are mentioned in Subsection 2.2 include handwritten proofs of minimality and/or prunedness in their works, but no actual proof of correctness by way of some formal semantics, which may anyway not be very meaningful when informally reasoning about the given pseudocodes. It is a unique feature of theorem provers to be able to make solid claims about the very code that can later be executed or embedded into a larger program.

9. Conclusion and Future Work

In this paper, we presented a functional variant of Braun et al.'s [9] SSA construction algorithm together with a proof that it preserves the program semantics. We further proved that the output is in pruned and, for reducible CFGs, minimal SSA form. We used Isabelle's code generator to export the algorithm's formalization to an efficient implementation in OCaml. We integrated the extracted code into the CompCertSSA compiler [8] and showed its applicability to real world programs. The runtime of the extracted code is on par with that of the original SSA construction of CompCertSSA.

In the future, the work could be extended to Braun et al.'s full algorithm, producing minimal SSA form even for irreducible CFGs. This involves the computation of strongly connected components, which would increase the proof complexity considerably.

The formalization including the proofs and integration into CompCertSSA can be found in the archive of formal proofs [24]. Ongoing work can be found on our project website [4].

Acknowledgments

We thank Joachim Breitner, Christoph Mallon, Manuel Mohr and Andreas Zwinkau for many fruitful discussions and valuable advice, and the anonymous reviewers for their helpful comments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) and grant Sn11/10-2.

References

[1] The GCC internals documentation. http://gcc.gnu.org/onlinedocs/gccint/SSA.html. Retrieved: 23 Oct. 2013.

[2] libFirm – The FIRM intermediate representation library. URL http://libfirm.org.

[3] Source code of the LLVM compiler infrastructure. http://llvm.org/viewvc/llvm-project/llvm/trunk/lib/Transforms/Utils/PromoteMemoryToRegister.cpp?revision=189169&view=markup. Revision 189169, 24 Aug. 2013.

[4] SSA construction project website. http://pp.ipd.kit.edu/ssa_construction.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11. ACM, 1988. ISBN 0-89791-252-7. doi: 10.1145/73560.73561.

[6] J. Aycock and N. Horspool. Simple generation of static single-assignment form. In *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2000. ISBN 978-3-540-67263-0. doi: 10.1007/3-540-46423-9_8.

[7] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, pages 1–31, 2013. ISSN 0168-7433. doi: 10.1007/s10817-013-9284-7.

[8] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, Mar. 2014. ISSN 0164-0925. doi: 10.1145/2579080.

[9] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In R. Jhala and K. Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2013. doi: 10.1007/978-3-642-37051-9_6.

[10] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming lan-*

guages, POPL '91, pages 55–66. ACM, 1991. ISBN 0-89791-419-8. doi: 10.1145/99583.99594.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

[12] D. Demange and Y. Fernández de Retana. Mechanizing conventional SSA for a verified destruction with coalescing. In *Compiler Construction*, 2016.

[13] S. Kalvala, R. Warburton, and D. Lacey. Program transformations using temporal logic side conditions. *ACM Transactions on Programming Languages and Systems*, 31(4):14:1–14:48, May 2009. ISSN 0164-0925. doi: 10.1145/1516507.1516509.

[14] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–86. IEEE Computer Society, 2004. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665.

[16] W. Mansky and E. Gunter. A framework for formal verification of compiler optimizations. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 2010. ISBN 978-3-642-14051-8. doi: 10.1007/978-3-642-14052-5_26.

[17] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0.

[18] B. Nordhoff and P. Lammich. Dijkstra's shortest path algorithm. *Archive of Formal Proofs*, Jan. 2012. ISSN 2150-914x. http://afp.sf.net/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.

[19] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27. ACM, 1988. ISBN 0-89791-252-7. doi: 10.1145/73560.73562.

[20] S. Schneider. Semantics of an intermediate language for program transformation. Master's thesis, Saarland University, 2013.

[21] S. Schneider, G. Smolka, and S. Hack. *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, chapter A Linear First-Order Functional Intermediate Language for Verified Compilers, pages 344–358. Springer International Publishing, Cham, 2015. ISBN 978-3-319-22102-1. doi: 10.1007/978-3-319-22102-1_23.

[22] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 62–73. ACM, 1995. ISBN 0-89791-692-1. doi: 10.1145/199448.199464.

[23] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 194–210, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66459-9. doi: 10.1007/3-540-48294-6_13.

[24] S. Ullrich and D. Lohner. Verified construction of static single assignment form. *Archive of Formal Proofs*, Feb. 2016. ISSN 2150-914x. http://afp.sf.net/entries/Formal_SSA.shtml, Formal proof development.

[25] D. Wasserrab and A. Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Proceedings of the 21st International Conference of Theorem Proving in Higher Order Logics*, pages 294–309, Montréal, Québec, Canada, Aug. 2008. Springer. doi: 10.1007/978-3-540-71067-7.

[26] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 427–440. ACM, 2012. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103709.

[27] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 175–186. ACM, 2013. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462164.