Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

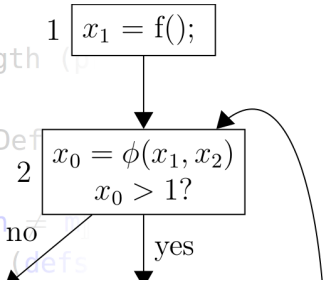
# Verified Construction of SSA Form

Bachelorarbeit von

**Sebastian Ullrich**

an der Fakultät für Informatik

```
assumes phis_wf:
  "⟦phis g (n,v) = Some args⟧ ⟹ length (p
assumes simpleDefs_phiDefs_disjoint:
  "n ∈ set (αn g) ⟹ defs g n ∩ phiDef
assumes all_defs_disjoint:
  "⟦n ∈ set (αn g); m ∈ set (αn g); n
   (defs g n ∪ phiDefs (phis g) n) ∩ (defs
```




| | |
|---|---|
| **Erstgutachter:** | Prof. Dr.-Ing. Gregor Snelting |
| **Zweitgutachter:** | Prof. Dr. rer. nat. Bernhard Beckert |
| **Betreuende Mitarbeiter:** | Dipl.-Inform. Denis Lohner<br>Dipl.-Inform. Sebastian Buchwald |

**Bearbeitungszeit:** 4. Juli 2013 – 29. Oktober 2013

**Verifizierte Konstruktion von SSA-Form**

*Static-Single-Assignment-Form* (SSA-Form) ist eine Eigenschaft von Zwischenrepräsentationen, die von vielen Compilern für die Optimierungsphase verwendet wird. Während der bekannteste Algorithmus zur Konstruktion von SSA-Form (Cytron et al. [9]) auf der Berechnung von iterierten Dominanzgrenzen basiert, stellen Braun et al. [7] einen bedeutend einfacheren und trotzdem effizienten Konstruktionsalgorithmus vor. Die vorliegende Arbeit präsentiert eine funktionale Version dieses Algorithmus und einen Beweis ihrer Korrektheit mittels des Theorembeweisers Isabelle und einer Big-Step-Semantik.

Die Implementierung des Algorithmus in Isabelle/HOL baut dabei auf einer abstrakten Repräsentation des Quellprogramms als Control Flow Graph (CFG) auf, in der jeder Block auf seine def- und use-Menge reduziert ist. Die Semantik arbeitet ebenfalls auf diesem Graph und stellt sicher, dass die SSA-Konstruktion sowohl in sich konsistent ist als auch def-use-Beziehungen des Originalprogramms erhält. Schließlich wird die abstrakte CFG-Repräsentation durch eine einfache While-Sprache instanziiert, um die Anwendbarkeit der Ergebnisse sicherzustellen.

**Abstract**

In this thesis, I prove the correctness of a new, simple yet efficient construction algorithm for static single assignment form described by Braun et al. [7]. The proof is based on a general, abstract control flow graph representation and big-step semantics and verified by the theorem prover Isabelle. The control flow graph is later instantiated by use of a simple While language to show the satisfiability of the representation's assumed properties.

# Contents

# 1 Introduction

Many modern compilers employ intermediate representations in *static single assignment* form (SSA form) for their optimization phases, including GCC [1], Java HotSpot [13], libFirm [15] and LLVM [14]. The simplicity of SSA form makes def-use relations explicit and simplifies common optimization algorithms such as common sub-expression elimination or branch elimination [3].

Multiple algorithms for constructing SSA form have been proposed and implemented in compilers. They differ in implementation complexity, runtime efficiency and output size. As an integral part of optimizing compilers, there is special interest in formally proving these algorithms to be correct, i.e. to preserve the semantics of the original program. This thesis presents a proof of the correctness of a new algorithm described in *Simple and Efficient Construction of Static Single Assignment Form* by Braun et al. [7]. The proof is formulated using the theorem prover Isabelle [19].

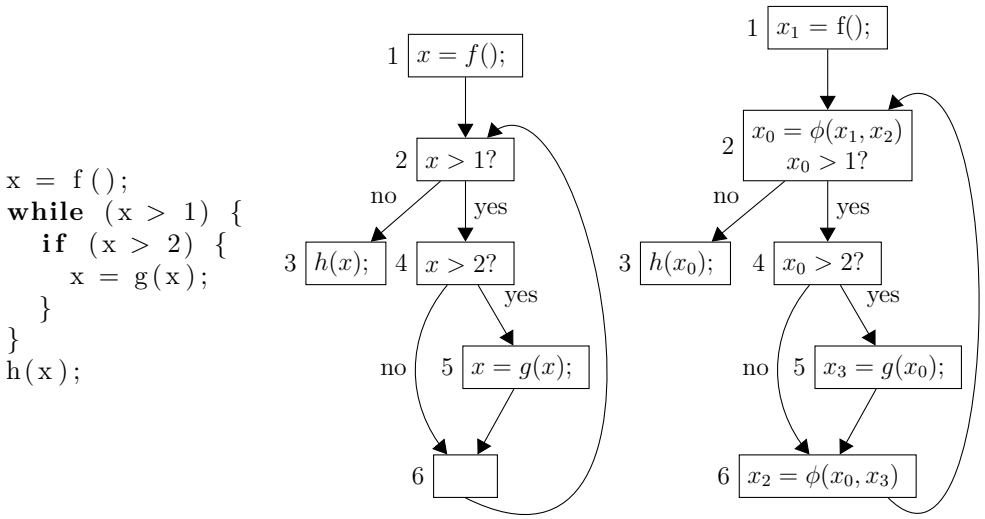
Section 2 of this thesis defines SSA form and describes various SSA construction algorithms including Braun et al.'s. In Section 3, I build an abstract Isabelle/HOL representation of SSA form. Section 4 presents a declarative, functional implementation of the basic version of Braun et al.'s algorithm in Isabelle/HOL that is suitable for a formal proof of correctness. The proof is presented in Section 5. Section 6 concludes the proof part by showing that the abstract representations used are indeed instantiable. Finally, in Section 7, I discuss the runtime efficiency of the implementation.

# 2 Background

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

## 2.1 SSA Form

SSA form was introduced and defined by Rosen et al. [20]. It is based on the *control flow graph* (CFG) of the source program. A CFG is constructed by partitioning the source code into *basic blocks*, contiguous sequences of statements featuring no jump targets (except possibly for the first instruction) and no jump instructions (except possibly for the last instruction). These blocks are then connected according to the program's jump operations, forming a directed graph. See Figure 1(b) for an example.




**(a)** program **(b)** CFG **(c)** SSA CFG

**Figure 1:** Example code and its CFG and SSA CFG representations. The code is partitioned into basic blocks to form the nodes of the CFG. Conditional jumps are described using "yes"/"no" edge annotations. To achieve SSA form, the variable $x$ is then split into multiple distinct identifiers by use of indices. Two $\phi$ functions have to be inserted for this.

A CFG is in SSA form if every variable is statically assigned at most once. In other words, SSA form does not track the mutable *variables* of the source program but merely their *values* – every time a new value is assigned to a variable in the original program, a unique identifier is inserted into the SSA CFG to track this value. In this thesis, I 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 two SSA values reach the same basic block, a $\phi$ *function* may be inserted in order to decide which definition to use. A $\phi$ function's parameter list has the same length as the list of the basic block's incoming control flow edges. When control flow enters the basic block through an edge, the $\phi$ function evaluates to the corresponding parameter. Figure 1(c) shows the result of transforming the example program into SSA form. I will call SSA values defined by $\phi$ functions $\phi$ *definitions* and all other definitions *simple definitions*.

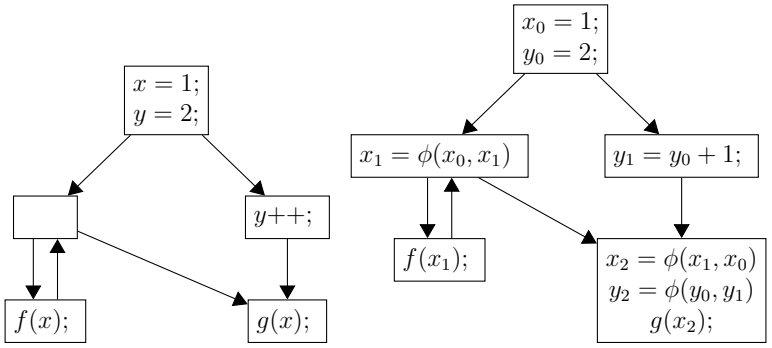
## 2.2 SSA Construction Algorithms

Cytron et al. [9] 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 $\phi$ functions are only needed at the iterated dominance frontiers of nodes containing a variable definition. Cytron et al. call a valid set of $\phi$ function placements that adheres to this condition *minimal* (see Figure 2(c) for an example of minimizing an SSA CFG). They present an algorithm to compute the dominance frontiers and $\phi$ placements and, using a set of sample programs, argue that the transformation is effectively linear for practical inputs.

The total number of placed $\phi$ functions, however, is not necessarily minimal yet. The SSA CFG may further be *pruned* to eliminate $\phi$ functions that (transitively) only have other $\phi$ definitions as users (Figure 2(d)). Choi et al. [8] extend Cytron et al.'s algorithm with liveness analysis to produce both pruned and minimal SSA form.

Sreedhar and Gao [21] use *DJ-graph*s, an extension of dominator trees, to "obtain, on average, more than five-fold speedup over [Cytron et al.'s] algorithm". While the measurements appear to be erroneous, the algorithm still outperforms Cytron et al.'s algorithm [6] and is used in e.g. LLVM for this reason [2].

In contrast to the complex dominance frontiers computation, Aycock and Horspool [4] propose a simple construction algorithm that starts with a maximal SSA form by inserting $\phi$ functions for every variable into every basic




**(a)** original CFG **(b)** unpruned, unminimized SSA

**(c)** minimized SSA **(d)** pruned and minimized SSA

**Figure 2:** Example of pruned and minimized SSA form. (b) shows a trivial SSA transformation of the input CFG (a). However, $x_1$ is redundant since its $\phi$ function depends on only one SSA value apart from $x_1$ itself. After removing the definition of $x_1$ and replacing all uses of $x_1$ with $x_0$, $x_2 = \phi(x_0, x_0)$ is redundant, too, and removing it yields minimal SSA form (c). Because $y_2$ has no users, it too can be removed, achieving pruned SSA form (d).

block. Redundant $\phi$ functions are then iteratively eliminated using the following rules:

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

Aycock and Horspool prove 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. [7] describe an SSA construction algorithm they call "simple" because, unlike Cytron et al.'s algorithm, it does not depend on additional analyses. However, they also aim for efficiency and demonstrate a slight increase in performance when replacing LLVM's highly-optimized implementation of Sreedhar and Gao's algorithm with their own 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 construction is based directly on the abstract syntax tree and can additionally employ simple optimizations on the fly.

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, Braun et al.'s algorithm searches backwards for reaching definitions of the variable used. If the search encounters a join point (a node with more than one predecessor), it places a $\phi$ function in the node and recursively continues the search in all predecessor nodes to look up the new $\phi$ function's parameters. The search stops on nodes containing a simple definition of the variable or a $\phi$ definition placed earlier. Therefore, the search never visits the same join point twice, ensuring termination. Figure 3 shows an example lookup. Braun et al. then extend the algorithm to remove redundant $\phi$ 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 $\phi$ functions, Braun et al. achieve minimal SSA form for all inputs.

It is the combination of simplicity and efficiency that made me choose Braun et al.'s algorithm as the subject of this thesis. The former property makes

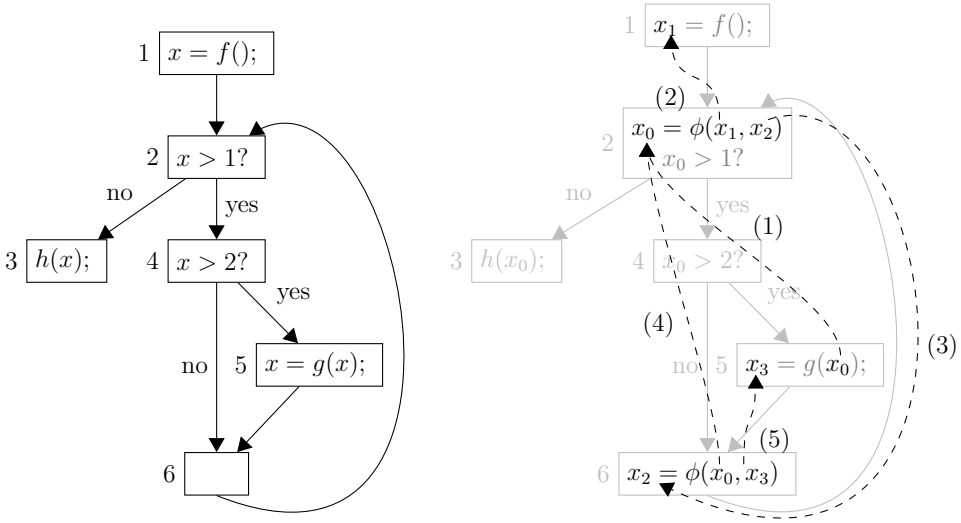



**Figure 3:** Input CFG and output of Braun et al.'s algorithm. The steps to resolve the use of $x$ in node 5 are pictured and numbered on top of the right graph. The algorithm starts in node 5 and searches for a definition through nodes 4 and 2 *(1)*. Since node 2 is a join point, a new $\phi$ definition $x_0$ is inserted, then the search continues recursively in each predecessor node to find the corresponding parameter. For the first parameter, a definition is found in node 1 *(2)*. The lookup for the second parameter enters node 6, a join point, places a new $\phi$ definition $x_2$ *(3)* and again recurses into its two predecessors. The first call is eventually stopped by the $\phi$ definition previously inserted in node 2 *(4)*, preventing the algorithm to loop forever, the second by the simple definition in node 5 *(5)*.

it amenable to formal proving without depending on further theories on additional data structures and analyses, whereas the latter one ensures the algorithm is actually relevant in practice, as evidenced by its experimental implementation in LLVM. The thesis will focus on the basic version of the algorithm, which still guarantees prunedness.

## 2.3 Isabelle and Isabelle/HOL

The proofs of this thesis are written in and verified by the interactive theorem prover Isabelle2013. Indeed, this very document is produced by Isabelle using its facility to embed LaTeX code between proofs. This approach ensures the theorems printed here are exactly the ones I proved.

Isabelle/HOL is Isabelle's default object logic and the one used in this thesis. 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. The function *the* :: *'a option* ⇒ *'a* unwraps an *option* object; the result of *the None* is underspecified. 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*), 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*.

## 2.4 Graph Framework and Isabelle Locales

Instead of operating on the CFG representation of a specific language, the algorithm is defined on an abstract graph framework by Simon Kohlmeyer [12]. It contains a hierarchy of *locales*, an Isabelle abstraction mechanism comparable to type classes [5]. A locale **fixes** some set of operations (also called its *parameters*), then describes *assumptions* these operations should fulfill. Lemmas 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 4 shows the graph locales used in this theory. The operations declared here are $\alpha e$ and $\alpha n$, the projection of a graph onto its edges and its nodes, respectively, *invar*, a predicate used by the implementation to disregard inconsistent instances of type *'g*, 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 $\phi$ function.

Before lemmas from a locale can be used, the locale has to be *interpreted*, that is, its parameters have to be instantiated with actual functions that fulfill its assumptions. For node types *'node* that have some linear order, the framework comes with an interpretation based on red-black trees. While the locales mentioned do not contain any lemmas, the interpretation con-

**locale** *graph* =
**fixes**
$\alpha e :: 'g \Rightarrow ('node \times 'edgeD \times 'node)\ set$ **and**
$invar :: 'g \Rightarrow bool$

**locale** *graph-nodes* = *graph* +
**fixes** $\alpha n :: 'g \Rightarrow 'node\ list$
**assumes** $\alpha n$-*correct*:
$invar\ g \Longrightarrow set\ (\alpha n\ g) = fst\ `\ \alpha e\ g \cup snd\ `\ snd\ `\ \alpha e\ g$

**locale** *graph-inEdges* = *graph* +
**fixes** *inEdges* :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD \times 'node)\ list$
**assumes** *inEdges-correct*:
$invar\ g \Longrightarrow set\ (inEdges\ g\ n) = \{(\text{-}, \text{-}, t).\ t = n\} \cap \alpha e\ g$

**Figure 4:** Locales of the graph framework used in this thesis

tains *code equations* that allow us to not only reason about Isabelle/HOL code using standard Isabelle proofs, but to execute it or export it to other functional languages. This thesis uses code equations to produce a working Haskell version of the same code the correctness proof works on.

# 3 SSA Representation

In this section, I incrementally build a locale appropriate for characterizing CFGs in SSA form. The first locale is *graph-path*, which extends *graph-inEdges* to define a predicate characterizing paths through a graph. Specifically, $g \vdash n{-}ns{\rightarrow}m$ holds iff there is a path (a list of nodes) *ns* from $n$ to $m$ in $g$. Both $n$ and $m$ are included in *ns*, i.e. they are the list's first and last element, respectively. The predicate is defined inductively as follows:

**definition** *predecessors* :: $'g \Rightarrow 'node \Rightarrow 'node\ list$ **where**
  $predecessors\ g\ n \equiv map\ fst\ (inEdges\ g\ n)$

**inductive** *path* :: $'g \Rightarrow 'node \Rightarrow 'node\ list \Rightarrow 'node \Rightarrow bool$ $(\text{-} \vdash \text{--}\rightarrow\text{-})$ **where**

$$\frac{n \in set\ (\alpha n\ g) \qquad invar\ g}{g \vdash n{-}[n]{\rightarrow}n}$$

$$\frac{g \vdash n{-}ns{\rightarrow}m \qquad n' \in set\ (predecessors\ g\ n)}{g \vdash n'{-}n' \#\ ns{\rightarrow}m}$$

The helper function *predecessors* will replace *inEdges* in the remaining document since the algorithm does not use *'edgeD*, the edge data annotations.

Continuing to a full definition of a CFG, locale *graph-Entry* extends locale *graph-path* by assuming every graph $g$ contains a special node *Entry* $g$ that has no predecessors and that there is a path from this entry node to every other node. Finally, locale *CFG* is defined as an extension of *graph-Entry* (Figure 5).

Remember that every node in the CFG represents a basic block in the original program. Instead of a list of statements of a specific language per node, we annotate each node with just the def and use set of the corresponding basic block – these are all the properties needed by a generic SSA construction algorithm. Since this abstraction loses the order of execution inside the node, the two sets are required to be disjoint in every node. Also, because the algorithm will later iterate over the use set of each node, it must be finite, which of course should be the case for all practical examples of CFGs. Finally, for convenience, we assume the graph invariant is fulfilled for every graph, so it can be excluded from the proof text and is only of concern when interpreting the locale. Figure 6 shows these simplifications applied to the sample program.

**locale** *graph-Entry* = *graph-path* +
**fixes** *Entry* :: *'g* ⇒ *'node*
**assumes** *Entry-in-graph*: *Entry g* ∈ *set* (*αn g*)
**assumes** *Entry-unreachable*: *invar g* ⟹ *inEdges g* (*Entry g*) = []
**assumes** *Entry-reaches*:
⟦*n* ∈ *set* (*αn g*); *invar g*⟧ ⟹ ∃ *ns*. *g* ⊢ *Entry g*−*ns*→*n*

**locale** *CFG* = *graph-Entry* +
**fixes** *defs* :: *'g* ⇒ *'node* ⇒ *'var set*
**fixes** *uses* :: *'g* ⇒ *'node* ⇒ *'var set*
**assumes** *defs-uses-disjoint*: *n* ∈ *set* (*αn g*) ⟹ *defs g n* ∩ *uses g n* = {}
**assumes** *uses-finite*: *finite* (*uses g n*)
**assumes** *invar*: *invar g*

**Figure 5:** Definition of locales *graph-Entry* and *CFG*

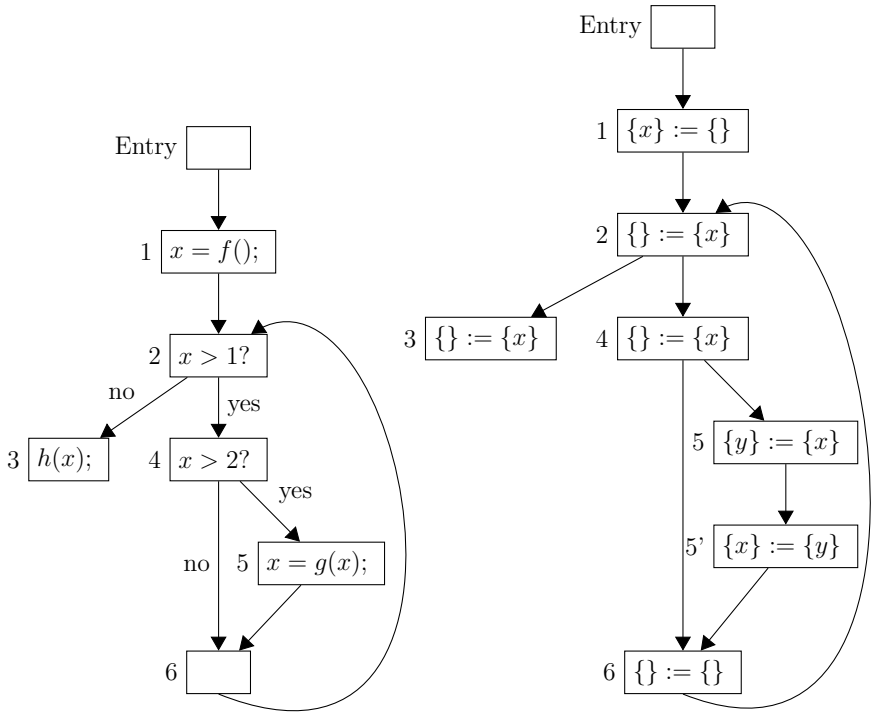



**Figure 6:** Original CFG and simplified CFG. The simplification removes all edge annotations and replaces block contents with the respective def and use sets, separated by ":=". Block 5 has to be split and a temporary variable $y$ has to be introduced to preserve the order of execution.

By extending this locale with $\phi$ functions we can finally define an abstract locale for well-formed SSA CFGs (Figure 7). Because the left-hand sides of $\phi$ definitions in a node are mutually distinct, the set of all $\phi$ definitions in the graph can be characterized as a mapping from *'node* × *'var*, a pair of containing node and the left-hand side, to the right-hand side, i.e. an argument list *'var list*. Function *phiDefs* extracts all identifiers of a block defined by $\phi$ 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 *all-defs-disjoint* describe the core property of SSA form: Every SSA value is defined at most once.

**type-synonym** (*'node*, *'var*) *phis* = *'node* × *'var* ⇀ *'var list*

**definition** *phiDefs* :: (*'node*, *'var*) *phis* ⇒ *'node* ⇒ *'var set* **where**
  *phiDefs phis n* ≡ {*v*. (*n*, *v*) ∈ *dom phis*}

**locale** *CFG-SSA* = *CFG* +
**fixes** *phis* :: *'g* ⇒ (*'node*, *'var*) *phis*
**assumes** *phis-wf*:
  ⟦*phis g* (*n*,*v*) = *Some args*⟧ ⟹ *length* (*predecessors g n*) = *length args*
**assumes** *simpleDefs-phiDefs-disjoint*:
  *n* ∈ *set* (*αn g*) ⟹ *defs g n* ∩ *phiDefs* (*phis g*) *n* = {}
**assumes** *all-defs-disjoint*:
  ⟦*n* ∈ *set* (*αn g*); *m* ∈ *set* (*αn g*); *n* ≠ *m*⟧ ⟹
   (*defs g n* ∪ *phiDefs* (*phis g*) *n*) ∩ (*defs g m* ∪ *phiDefs* (*phis g*) *m*) = {}

**Figure 7:** Definition of locale *CFG-SSA*

# 4 Construction

In this section, I develop a functional version of Braun et al.'s SSA construction algorithm. All code is contained in the *CFG* locale, so functions *defs* and *uses* may be used. The goal is to construct a corresponding SSA CFG, i.e. to define functions that describe a valid *CFG-SSA* interpretation inside this locale. In other words, given a valid *CFG* interpretation, the code should yield a valid *CFG-SSA* interpretation.

For the basic version of the algorithm, which generally does not yield minimal SSA form, we can build a very declarative implementation by splitting it into two different tasks: looking up the places to insert $\phi$ functions into, and looking up the parameters of a single $\phi$ function. The following algorithm collects a set of the insertion points for a variable $v$ starting from a node $n$.

```
fun 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 $\phi$ 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. Section 5 will introduce a *definite assignment* assumption to exclude such ill-defined programs.

We may now describe the locations of all $\phi$ functions using another fold over all use sites of $v$:

```
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 $\phi$ functions, the remaining task is to look up their parameters. First, we need a scheme for encoding SSA values.

A common scheme is to add a sequential index to all occurences of a variable. However, in the simplified case of this thesis, for every original variable and node in the SSA CFG there may be at most one $\phi$ definition and at most one simple definition of an SSA value for that variable. Therefore, an SSA value can be uniquely identified by the following triple:

```
datatype Def = SimpleDef | PhiDef
type-synonym ('node, 'var) ssaVal = 'var × 'node × Def
```

Looking up a $\phi$ function's parameters is a simple non-branching search:

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

Definitions in the input CFG are lifted to *SimpleDef*s. Any join point reached must contain a *PhiDef* because of *phiDefNodes*'s implementation.

Figure 8 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,SimpleDef)) ‘ 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,def)).
    if m = n ∧ n ∈ phiDefNodes g v ∧ def = PhiDef then
      Some [lookupDef g m v . m ← predecessors g n]
    else None
```

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

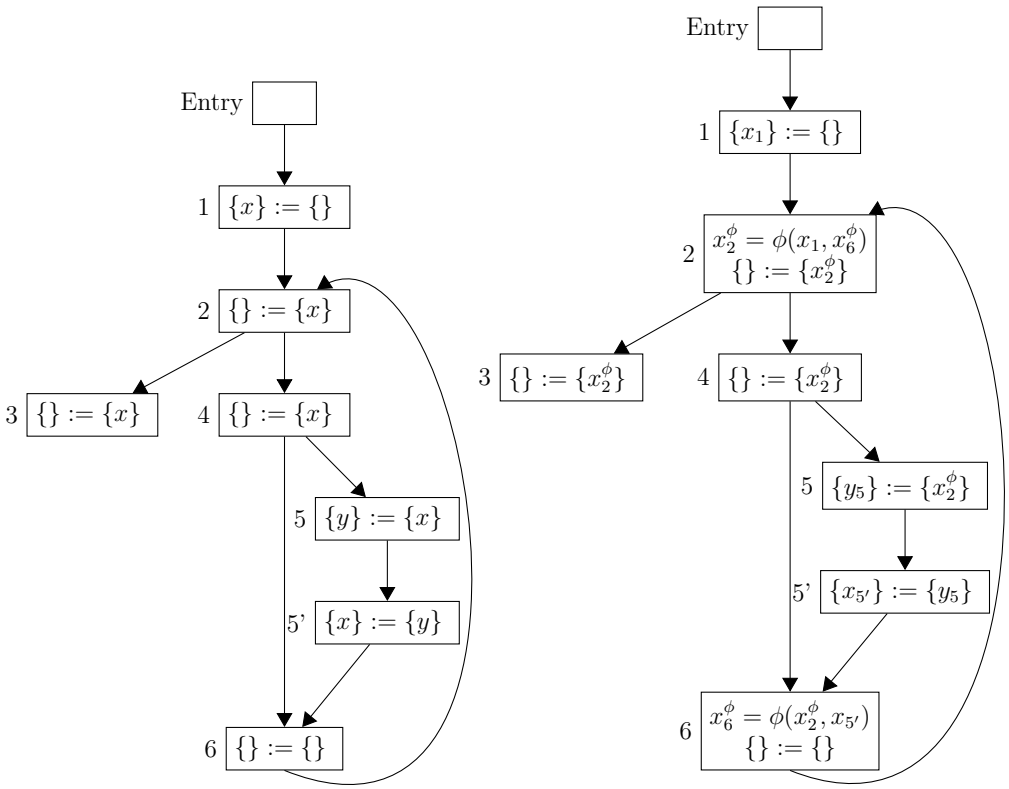



**Figure 8:** Simplified CFG and the SSA CFG generated by the algorithm. SSA values are printed by appending the number of the node the variable is defined in as an index to the variable name (in contrast to Figure 3 in which indices are sequential). *PhiDef*s are additionally marked with a superscripted $\phi$.

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

**theorem** *constructs-ssa*: *CFG-SSA* $\alpha e$ $\alpha n$ *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 and do not require any induction.

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 $\phi$ function, it is marked as live. Incrementally, if an SSA value marked as live is defined by a $\phi$ 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.

**inductive** *liveVal* :: (*'node*, *'var*) *ssaVal* ⇒ *bool* **where**

$$\frac{val \in uses'\ g\ n}{liveVal\ val} \qquad \frac{liveVal\ val \qquad val' \in set\ (the\ (phis'\ g\ (n,\ val)))}{liveVal\ val'}$$

It follows that all $\phi$ definitions are live:

**theorem** *phis'-pruned*: *val* ∈ *phiDefs* (*phis'* *g*) *n* ⟹ *liveVal val*

# 5 Proof of Correctness

While the previous section ends with a proof that the algorithm indeed constructs pruned SSA form, it does not state any connection between the input CFG and the SSA CFG. This section provides a proof of the two CFGs being semantically equivalent under a custom big-step semantics.

The *CFG* locale is missing any sort of *definite assignment* assumption, i.e. an assurance that on every path from the *Entry* node to some node $n$ where a variable $v$ is used, the path contains a node defining $v$. This was not needed for the definition of the algorithm. However, we will need it in this section to define a well-defined semantics. Figure 9 shows the definition of locale *CFG-wf*, which formalizes this assumption and is the context for the remaining part of the section.

Big-step semantics define the *state* of the program after executing a series of instructions. State is usually characterized as a mapping of live variables to their respective current values. Since the simplified *CFG* locale has no notion of "value", they can be equivalently replaced by the node each variable was last defined in.

**type-synonym** $('node, 'var)\ state = 'var \rightharpoonup 'node$

In the SSA CFG, the state is defined to be the mapping of a variable to both its last *actual* definition (an *ssaVal*) and its last *original* definition, the

**context** *CFG*
**begin**
  **definition** $defAss :: 'g \Rightarrow 'node \Rightarrow 'var \Rightarrow bool$ **where**
    $defAss\ g\ m\ v \equiv \forall ns.\ g \vdash Entry\ g{-}ns{\rightarrow}m \longrightarrow (\exists n \in set\ ns.\ v \in defs\ g\ n)$
**end**

**locale** *CFG-wf* = *CFG* +
  **assumes** *def-ass-uses*: $\forall m \in set\ (\alpha n\ g).\ \forall v \in uses\ g\ m.\ defAss\ g\ m\ v$
  **assumes** *Entry-no-defs*: $defs\ g\ (Entry\ g) = \{\}$

**Figure 9:** Definition of locale *CFG-wf* used for the correctness proof. *def-ass-uses* formalizes the definite assignment assumption. *Entry-no-defs* is an additional constraint that simplifies the semantics definitions and may easily be fulfilled by any interpretation by adding a new empty Entry node to the graph.

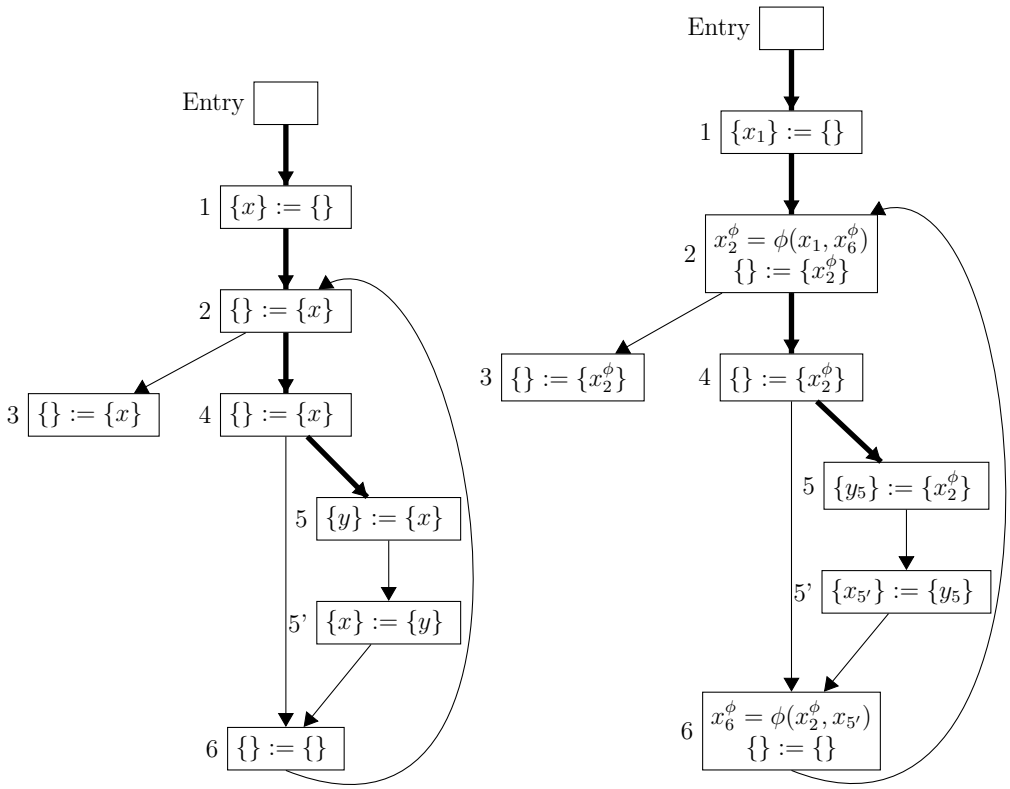



**Figure 10:** When executing the path through nodes 1, 2, 4, 5, the left CFG results in the state $\{x \to 1, y \to 5\}$ and the SSA CFG results in $\{x \to ((x, 2, PhiDef), 1), y \to ((y, 5, SimpleDef), 5)\}$. Note that the "original definitions" part of the SSA state matches the state in the original CFG.

definition left after resolving all $\phi$ functions on the path through the CFG (sufficiently described by its containing node).

**type-synonym** ('*node*, '*var*) *ssaState* = '*var* ⇀ ('*node*, '*var*) *ssaVal* × '*node*

Figure 10 provides an example of both states.

The "original definitions" part of *ssaState* is used for the equivalence proof. A *state* and an *ssaState* may be called equivalent if restricting the *ssaState* to its second part yields two identical maps. The "actual definitions" part, on the other hand, is needed to define a well-defined semantics. When encountering a $\phi$ function, execution is only well-defined if the $\phi$ function's parameter in question matches the actual definition in the state map.

Both semantics relate a graph and a path (list of nodes) through it to the state in the last node.

**definition** *updateState* ::
$'g \Rightarrow 'node \Rightarrow ('node, 'var)\ state \Rightarrow ('node, 'var)\ state$
**where**
$updateState\ g\ m\ s\ v \equiv if\ v \in defs\ g\ m\ then\ Some\ m\ else\ s\ v$

**inductive** *validState* :: $'g \Rightarrow 'node\ list \Rightarrow ('node, 'var)\ state \Rightarrow bool$
**where**

$$\frac{}{validState\ g\ [Entry\ g]\ Map.empty}$$

$$\frac{validState\ g\ ns\ s \qquad last\ ns \in set\ (predecessors\ g\ m)}{validState\ g\ (ns\ @\ [m])\ (updateState\ g\ m\ s)}$$

The semantics operating on the original CFG, *validState*, starts with an empty state in the entry node and uses the helper function *updateState* to update the map whenever it walks over a definition.

**definition** *ssaUpdateState* ::
$'g \Rightarrow 'node \Rightarrow nat \Rightarrow ('node, 'var)\ ssaState \Rightarrow ('node, 'var)\ ssaState$
**where**
$ssaUpdateState\ g\ m\ i\ s\ v \equiv$
$\quad if\ v \in defs\ g\ m\ then\ Some\ ((v,m,SimpleDef),m)$
$\quad else\ case\ phis'\ g\ (m,(v,m,PhiDef))\ of$
$\quad\quad Some\ phiParams \Rightarrow$
$\quad\quad\quad let\ (actualDef,origDef) = the\ (s\ v)\ in$
$\quad\quad\quad if\ actualDef = phiParams\ !\ i\ then\ Some\ ((v,m,PhiDef),origDef)$
$\quad\quad\quad else\ undefined$
$\quad\quad | None \Rightarrow s\ v$

**inductive** *ssaValidState* :: $'g \Rightarrow 'node\ list \Rightarrow ('node, 'var)\ ssaState \Rightarrow bool$
**where**

$$\frac{}{ssaValidState\ g\ [Entry\ g]\ Map.empty}$$

$$\frac{\begin{array}{c}ssaValidState\ g\ ns\ s \\ last\ ns = predecessors\ g\ m\ !\ i \qquad i < length\ (predecessors\ g\ m)\end{array}}{ssaValidState\ g\ (ns\ @\ [m])\ (ssaUpdateState\ g\ m\ i\ s)}$$

The semantics operating on the SSA CFG, *ssaValidState*, does so, too, and additionally marks such definitions as *SimpleDef*s via the helper function

*ssaUpdateState*. If there is no simple definition of $v$ in $m$, *ssaUpdateState* continues by looking for a $\phi$ function for $v$ in $m$. If there is one, it verifies the function's $i$th parameter (where $i$ is the index of the predecessor through which the node was entered) matches the entry in the current state, then updates the state with a *PhiDef*. The "original definition" part of the state is not updated in this case.

The first observation about both semantics is that every path yields a valid state:

**lemma** *validState-exists*:
**assumes** $g \vdash Entry\ g{-}ns{\rightarrow}n$
**obtains** $s$ **where** *validState g ns s*

**lemma** *ssaValidState-exists*:
**assumes** $g \vdash Entry\ g{-}ns{\rightarrow}n$
**obtains** $s$ **where** *ssaValidState g ns s*

The equivalence theorem is defined as following:

**theorem** *in-lockstep*:
**assumes** *validState g ns s* **and** *ssaValidState g ns* $s'$
**shows** $s = Option.map\ snd \circ s'$

This definition, however, is not yet sufficient. It would be trivially fulfilled by leaving $phis'$ empty. Only the existence of use sites necessitates inserting $\phi$ functions, which leads to the following additional theorems: Uses in the SSA CFG correspond to those in the input CFG and are well-defined (if computation reaches a use site, the used identifier must be the current "actual definition" in the state).

**theorem** *uses-uses′*:
**assumes** $n \in set\ (\alpha n\ g)$
**shows** *uses g n* = *fst* ‘ *uses′ g n*

**theorem** *uses′-welldefined*:
**assumes** *ssaValidState g ns* $s'$ **and** $(v,n,def) \in uses'\ g\ (last\ ns)$
**obtains** *val* **where** $s'\ v = Some\ ((v,n,def),\ val)$

The latter theorem may also be interpreted as saying that every use site of an SSA value is *dominated* by its def site, i.e. every execution path to the use site goes through the definition first.

# 6 Interpretation

The proof of correctness is moot if it relies on improper locale assumptions. Specifically, if any assumption or the set as a whole was unsatisfiable, any proposition could be trivially concluded. Therefore, I present an interpretation of the *CFG-wf* locale for CFGs of a simple While language. The construction algorithm is described by Kohlmeyer [12] and implemented by the following function:

*build* :: *cmd* ⇒ (*w-node*, *state edge-kind*) *graph*

The type *cmd* describes abstract syntax trees of the While language. The type ('*node*, '*edge*) *graph* together with its functions *mg-αe*, *mg-αn*, etc. constitutes a full interpretation of the graph locales. Whereas *edge-kind*, the type of the edge annotations, is not relevant to us, *w-node* is defined as being either the entry node (-*Entry*-), the exit node (-*Exit*-) or some sequentially labelled node (- *n* -).

We reuse the existing graph interpretation to build one for the CFGs of programs of type *cmd*:

**abbreviation** *wg-αe c* ≡ *mg-αe* (*build c*)
**abbreviation** *wg-αn c* ≡ *mg-αn* (*build c*)
**abbreviation** *wg-invar c* ≡ *mg-invar* (*build c*)
**abbreviation** *wg-inEdges c* ≡ *mg-inEdges* (*build c*)
**abbreviation** *wg-Entry c* ≡ (-*Entry*-)

**interpretation** *wg*: *graph-path wg-αe wg-αn wg-invar wg-inEdges*

The lemmas accompanying *build* allow us to prove most of *CFG-wf*'s assumptions. For example,

*stdEdges* ≡ {((-*Entry*-), (λ*x*. *True*)$_{\surd}$, (- *0* -)), ((-*Entry*-), (λ*x*. *False*)$_{\surd}$, (-*Exit*-))}
*build-edgesE1*: *e* ∈ *stdEdges* ⟹ *e* ∈ *mg-αe* (*build cmd*)

imply that *build c* includes edges outgoing from the entry node, thus we can prove our interpretation of the *graph-Entry* assumption that (-*Entry*-) is always a member of *αn g*:

**lemma** *wg-Entry-in-αn*: (-*Entry*-) ∈ *set* (*wg-αn c*)

One assumption the *build* function does not guarantee is the def and use sets being disjoint in every basic block. However, *build* already creates minimal basic blocks that each contain exactly one statement. Therefore, it is sufficient to split assignment statements of form $V{:=}e$, where $V$ is contained in $e$, in the original program by use of a new temporary variable (as seen in Figure 6). This is done by function *transform* :: *cmd* ⇒ *cmd*; its full defini-

tion and a proof of it being semantics preserving is included in the Isabelle theory files.

Chaining *build* and *transform* together yields a valid *CFG* interpretation. Functions *Defs* and *Uses* are from *cmd*'s theory module.

**abbreviation** *wg-tf-αe c* ≡ *mg-αe* (*build* (*transform c*))
**abbreviation** *wg-tf-αn c* ≡ *mg-αn* (*build* (*transform c*))
**abbreviation** *wg-tf-invar c* ≡ *mg-invar* (*build* (*transform c*))
**abbreviation** *wg-tf-inEdges c* ≡ *mg-inEdges* (*build* (*transform c*))
**abbreviation** *wg-tf-Entry c* ≡ (*-Entry-*)
**abbreviation** *wg-tf-defs c* ≡ *Defs* (*transform c*)
**abbreviation** *wg-tf-uses c* ≡ *Uses* (*transform c*)

**interpretation** *wg-tf*: *SSA-CFG.CFG wg-tf-αe wg-tf-αn wg-tf-invar wg-tf-inEdges wg-tf-Entry wg-tf-defs wg-tf-uses*

Having successfully interpreted locale *CFG*, we can finally export the algorithm to other functional languages such as Haskell:

**export-code** *wg-tf.defs′ wg-tf.uses′ wg-tf.phis′* **in** *Haskell*

For example, this is the translated type signature of *phiDefNodes*:

```
phiDefNodes :: Cmd -> String -> Set W_node
```

To show the applicability of the proof of correctness, we still have to interpret locale *CFG-wf*. However, *cmd* gives no guarantee of definite assignment and therefore neither does the output of *build*. The *typedef* keyword allows us to restrict *cmd* to instances that are wellformed.

**typedef** *wf-cmd* = {*c* . *wg-tf.defAssUses c*}

*typedef*s are only accepted if the resulting type is not empty. This is true for *wf-cmd* since the program *Skip* trivially exhibits the definite assignment property.
The existing functions operating on *cmd* can be *lifted* [10] to ones on *wf-cmd*. For example:

**setup-lifting** *type-definition-wf-cmd*

**lift-definition** *wf-αe* :: *wf-cmd* ⇒ (*w-node* × *state edge-kind* × *w-node*) *set* **is** *wg-tf-αe* **..**

Using the type invariant of *wf-cmd*, the interpretation of *CFG-wf* follows directly.

**interpretation** *wg-wf*: *SSA-Semantics.CFG-wf wf-αe wf-αn wf-invar wf-inEdges wf-Entry wf-defs wf-uses*

# 7 Evaluation

While runtime efficiency has never been a focus of the thesis, it should be noted that the algorithm can exhibit exponential runtime behavior because of the simplistic use of the *unvisited* list in the recursion. For example, in the following program with $n$ *if* structures, *phiDefNodes* $x$ walks every of the $2^n$ paths from $x$'s use to its definition.

```
x = f();
if (x > 1) {} else {}
...
if (x > 1) {} else {}
y = x;
```

To achieve at least polynomial runtime, the Isabelle theory files of this thesis include a more efficient implementation that is proved to be equivalent. It is based on an Isabelle/HOL implementation of depth-first search (DFS) by Nishihara and Minamide [18], ensuring that each node is visited only once. However, as Figure 11 shows, its runtime is still superlinear. A major factor that leads to this result is the use of the interpretation presented in Section 6: Instead of building the CFG of the program once, every single use of the CFG functions such as $\alpha n$ or *defs* wastefully invokes *build* on the whole graph again. When adding an interpretation for a real-world language where performance matters, one may decide to create a more elaborate interpretation in order to circumvent this problem.

| | Runtime (s) | |
|---|---|---|
| $n$ | Original | DFS-based |
| 7 | 0.223 | 0.021 |
| 8 | 0.497 | 0.031 |
| 9 | 1.179 | 0.047 |
| 10 | 2.799 | 0.061 |
| 11 | 6.483 | 0.098 |
| 20 | - | 0.637 |
| 30 | - | 3.138 |
| 40 | - | 10.375 |

**Figure 11:** Runtime of the presented and the improved implementation on the described input, in seconds. The code was executed in Isabelle/ML on a Core i5-3320M CPU with 2.6 GHz.

# 8 Related Work

While this thesis contains the first formal correctness proof of Braun et al.'s algorithm known to the author, other SSA construction algorithms have been verified 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. [11] 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 $\phi$ 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, but do not consider termination. My 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. [23] also present a framework for formally verifying optimizations and proof the correctness of an implementation of Aycock and Horspool's algorithm to provide an example. The implementation is based on Vellvm [22], 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.

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 operational 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 thesis, I presented a simplified, functional implementation of Braun et al.'s [7] SSA construction algorithm together with a proof that its output is in pruned SSA form and is semantically equivalent to the input CFG. The algorithm is also executable and can be exported to other functional languages.

In the future, this work could be extended into a correctness proof of the full algorithm. Even for reducible CFGs, this means the acts of locating $\phi$ insertion points (*phiDefNodes*) and looking up their parameters (*lookupDef*) would need to be intertwined, i.e. implemented as mutually recursive functions or a single big recursive function. The work would be concluded with an additional proof of minimality and a proof showing that Braun et al.'s and Cytron et al.'s minimality definitions are equivalent.

# References

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

[2] 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.

[3] 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.

[4] 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.

[5] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, pages 1–31, 2013.

[6] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *ACM Transactions on Computational Logic*, 50(3):375–425, May 2003.

[7] 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.

[8] 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 languages*, POPL '91, pages 55–66. ACM, 1991.

[9] 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.

[10] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL.

[11] 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.

[12] K.-S. Kohlmeyer. Funktionale Konstruktion und Verifikation von Kontrollflussgraphen, Aug. 2012. Bachelor's Thesis.

[13] 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.

[14] 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–. IEEE Computer Society, 2004.

[15] G. Lindenmaier. libFIRM – a library for compiler optimization research implementing FIRM. Technical Report 2002-5, Sept. 2002.

[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.

[17] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[18] T. Nishihara and Y. Minamide. Depth first search. *Archive of Formal Proofs*, June 2004. http://afp.sf.net/entries/Depth-First-Search.shtml, Formal proof development.

[19] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, Sept. 1989.

[20] 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.

[21] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing $\phi$-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 62–73. ACM, 1995.

[22] 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.

[23] 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.

# Erklärung

Hiermit erkläre ich, Sebastian Andreas Ullrich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

| | |
|---|---|
| Ort, Datum | Unterschrift |