Fixed all the references (see the book.pdf).
Fixed the notations/figures/codes (for homogeneity).
Added some index.

CHAPTER 1

Introduction

J. Singer

Progress: 85%

Minor polishing in progress

In computer programming, as in real life, names are useful handles for concrete entities. The key message of this book is that having *unique names* for *distinct entities* reduces uncertainty and imprecision.

For example, consider overhearing a conversation about 'Homer.' Without any more contextual clues, you cannot disambiguate between Homer Simpson and Homer the classical Greek poet; or indeed, any other people called Homer that you may know. As soon as the conversation mentions Springfield (rather than Smyrna), you are fairly sure that the Simpsons television series (rather than Greek poetry) is the subject. On the other hand, if everyone had a *unique* name, then there would be no possibility of confusing 20th century American cartoon characters with ancient Greek literary figures.

removed the picture here as I am not sure we are allowed to include it :-(

This book is about the *static single assignment form* (SSA), which is a naming convention for storage locations (variables) in low-level representations of computer programs. The term *static* indicates that SSA relates to properties and analysis of program text (code). The term *single* refers to the uniqueness property of variable names that SSA imposes. As illustrated above, this enables a greater degree of precision. The term *assignment* means variable definitions. For instance, in the code

$$x = y + 1;$$

the variable x is being assigned the value of expression $(y+1)$. This is a definition, or assignment statement, for x. A compiler engineer would interpret the above assignment statement to mean that the lvalue of x (i.e., the memory location labelled as x) should be modified to store the value $(y+1)$.

1.1 Definition of SSA

The simplest, least constrained, definition of SSA can be given using the following informal prose:

removed the citation (not in the main body).

> *“ A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text. ”*

However there are various, more specialized, varieties of SSA, which impose further constraints on programs. Such constraints may relate to graph-theoretic properties of variable definitions and uses, or the encapsulation of specific control-flow or data-flow information. Each distinct SSA variety has specific characteristics. Basic varieties of SSA are discussed in Chapter **??**. Part **??** of this book presents more complex extensions.

One important property that holds for all varieties of SSA, including the simplest definition above, is *referential transparency*: i.e., since there is only a single definition for each variable in the program text, a variable's value is *independent of its position* in the program. We may refine our knowledge about a particular variable based on branching conditions, e.g. we know the value of x in the conditionally executed block following an `if` statement beginning:

$$\texttt{if}\ \ (x == 0)$$

however the *underlying value* of x does not change at this `if` statement. Programs written in pure functional languages are referentially transparent. Referentially transparent programs are more amenable to formal methods and mathematical reasoning, since the meaning of an expression depends only on the meaning of its subexpressions and not on the order of evaluation or side-effects of other expressions. For a referentially opaque program, consider the following code fragment.

$$\begin{aligned} x &= 1; \\ y &= x + 1; \\ x &= 2; \\ z &= x + 1; \end{aligned}$$

A naive (and incorrect) analysis may assume that the values of y and z are equal, since they have identical definitions of $(x + 1)$. However the value of variable x depends on whether the current code position is before or after the second definition of x, i.e., variable values depend on their *context*. When a compiler transforms this program fragment to SSA code, it becomes referentially transparent. The translation process involves renaming to eliminate multiple assignment statements for the same variable. Now it is apparent that y and z are equal if and only if x_1 and x_2 are equal.

$$x_1 = 1;$$
$$y = x_1 + 1;$$
$$x_2 = 2;$$
$$z = x_2 + 1;$$

1.2 Informal Semantics of SSA

In the previous section, we saw how straightline sequences of code can be transformed to SSA by simple renaming of variable definitions. The *target* of the definition is the variable being defined, on the left-hand side of the assignment statement. In SSA, each definition target must be a unique variable name. On the other hand, variable names can be used multiple times on the right-hand side of any assignment statements, as *source* variables for definitions. Throughout this book, renaming is generally performed by adding integer subscripts to original variable names. In general this is an unimportant implementation feature, although it can prove useful for compiler debugging purposes.

The ϕ-function is the most important SSA concept to grasp. It is a special statement, known as a *pseudo-assignment* function. Some call it a "notational fiction." [1] The purpose of a ϕ-function is to merge values from different incoming paths, at control-flow merge points.

removed the citation (Appel)

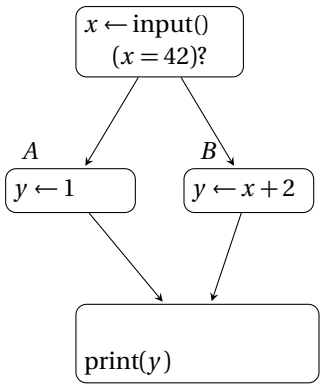
Consider the following code example and its corresponding control flow graph (CFG) representation:

Added the CFG representation

```
x = input();
if (x == 42)

then
  y = 1;
else
  y = x + 2;
end

print(y);
```



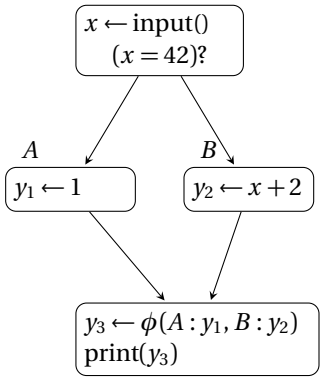
There is a distinct definition of y in each branch of the `if` statement. So multiple definitions of y reach the `print` statement at the control-flow merge point. When a compiler transforms this program to SSA, the multiple definitions of y are renamed as y_1 and y_2. However the `print` statement could use either variable,

[1] Kenneth Zadeck reports that ϕ-functions were originally known as *phoney*-functions, during the development of SSA at IBM Research. Although this was an in-house joke, it did serve as the basis for the eventual name.

dependent on the outcome of the `if` conditional test. A ϕ-function introduces a new variable y_3, which takes the value of either y_1 or y_2. Thus the SSA version of the program is:

```
x = input();
if (x == 42)

then
  y1 = 1;
else
  y2 = x + 2;
end
y3 = φ(y1, y2);
print(y3);
```



In terms of their position, ϕ-functions are generally placed at control-flow merge points, i.e., at the heads of basic blocks that have multiple predecessors in control-flow graphs. A ϕ-function at block b has n parameters if there are n incoming control-flow paths to b. The behavior of the ϕ-function is to select dynamically the value of the parameter associated with the actually executed control-flow path into b. This parameter value is assigned to the fresh variable name, on the left-hand-side of the ϕ-function. Such pseudo-functions are required to maintain the SSA property of unique variable definitions, in the presence of branching control flow. Hence, in the above example, y_3 is set to y_1 if the flow comes from basic-block A, and set to y_2 if it flows from basic-block B. Notice that the CFG representation here adopts a more expressive syntax for ϕ-functions than the standard one, as it associates predecessor basic block labels B_i with corresponding SSA variable names a_i, i.e. $a_0 = \phi(B_1 : a_1, \ldots, B_n : a_n)$. When not ambiguous, basic block labels will be omitted.

Moved this point from below. Illustrated using the example. Used the notation Bi:ai in the example.

It is important to note that, if there are multiple ϕ-functions at the head of a basic block, then these are executed in parallel, i.e., simultaneously *not* sequentially. This distinction becomes important if the target of a ϕ-function is the same as the source of another ϕ-function, perhaps after optimizations such as copy propagation (see Chapter **??**). When ϕ-functions are eliminated in the SSA destruction phase, they are sequentialized using conventional copy operations, as described in Chapters **??** and **??**. This subtlety is particularly important in the context of register allocated code (see Chapter **??**, e.g. Figure **??**).

Strictly speaking, ϕ-functions are not directly executable in software, since the dynamic control-flow path leading to the ϕ-function is not explicitly encoded as an input to ϕ-function. This is tolerable, since ϕ-functions are usually only used during static analysis of the program. They are removed before any program interpretation or execution takes place. However, there are various executable extensions of ϕ-functions, such as ϕ_{if} or γ functions (see Chapter **??**), which take an extra predicate parameter to replace the control dependence that dictates

not always.

tried to rephrase as it was too fuzzy.

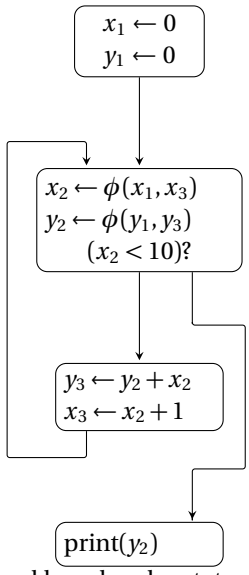
the argument the ϕ-function should select by a condition dependence. Such extensions are useful for program interpretation (see Chapter **??**), if-conversion (see Chapter **??** or), or hardware synthesis (see **??**).

Added the context (ie something else than analysis)

We present one further example in this section, to illustrate how a loop control-flow structure appears in SSA. Here is the non-SSA version of the program and its corresponding control flow graph SSA version:

Added the CFG version

```
x = 0;
y = 0;
while(x < 10){
   y = y + x;
   x = x + 1;
}
print(y)
```



The SSA code has two ϕ-functions in the compound loop header statement that merge incoming definitions from before the loop for the first iteration, and from the loop body for subsequent iterations.

It is important to outline that SSA should not be confused with (*dynamic*) single assignment (DSA or simply SA) form used in automatic parallelization. *Static* single assignment does not prevent multiple assignments to a variable during program execution. For instance, in the SSA code fragment above, variables y_3 and x_3 in the loop body are redefined dynamically with fresh values at each loop iteration.

Mentioned SA. Some people from automatic parallelisation make the confusion.

Full details of the SSA construction algorithm are given in Chapter **??**. For now, it is sufficient to see that:

Moved this from next section that was about comparison with dense data-flow analysis.

1. Integer subscripts have been used to rename variables x and y from the original program.
2. A ϕ-function has been inserted at the appropriate control-flow merge point where multiple reaching definitions of the same variable converged in the original program.

removed the undef. Removed it in the example. Prefer not to go into such details for now.

1.3 Comparison with Classical Data Flow Analysis

I changed the layout of the section a little bit. Removed the lattice, put 2 variables in the example, simplified it not to have undef.

As we will see with more details in Chapters **??** and **??**, one of the major advantage of SSA form concerns data flow analysis. Data flow analysis collects information about programs at compile time in order to make optimizing code transformations. During actual program execution, information flows between variables. Static analysis captures this behaviour by propagating *abstract* information, or data flow facts, using an operational representation of the program such as the control flow graph (CFG). This is the approach used in classical data flow analysis.

Just explained why we are talking about data-flow analysis here.

Often, data flow information can be propagated more efficiently using a *functional*, or *sparse*, representation of the program such as SSA. When a program is translated into SSA form, variables are renamed at definition points. For certain data flow problems (e.g. constant propagation) this is exactly the set of program points where data flow facts may change. Thus it is possible to associate data flow facts directly with variable names, rather than maintaining a vector of data flow facts indexed over all variables, at each program point.

Figure 1.1 illustrates this point through an example of non-zero value analysis. For each variable in a program, the aim is to determine statically whether that variable can contain a zero integer value at runtime. Here 0 represents the fact that the variable is null, $\emptyset$ the facts that it is non-null, and $\top$ the fact that it is maybe-null. With classical dense data flow analysis on the CFG in Figure 1.1(a), we would compute information about variables x and y for each of entry and exit of the six basic-blocks in the CFG, using suitable data flow equations. Using sparse SSA-based data flow analysis on Figure 1.1(b), we compute information about each variable based on a simple analysis of its definition statement. This gives us six data flow facts, one for each SSA version of variables x and y.

We do not need to bother the reader with some useless formalism here.

Was before.

For other data flow problems, properties may change at points that are not variable definitions. These problems can be accommodated in a sparse analysis framework by inserting additional pseudo-definition functions at appropriate points to induce additional variable renaming. See Chapter **??** for one such example. However, this example illustrates some key advantages of the SSA-based analysis.

1. Data flow information *propagates directly* from definition statements to uses, via the def-use links implicit in the SSA naming scheme. In contrast, the classical data flow framework propagates information throughout the program, including points such as node 2 where the information about x does not change, or is not relevant.
2. The results of the SSA data flow analysis are *more succinct*. There are four data flow facts (one for each SSA version of x) versus seven facts for the non-SSA program (one fact about variable x per CFG node).

Part **??** of this textbook gives a comprehensive treatment of some SSA-based data flow analysis.

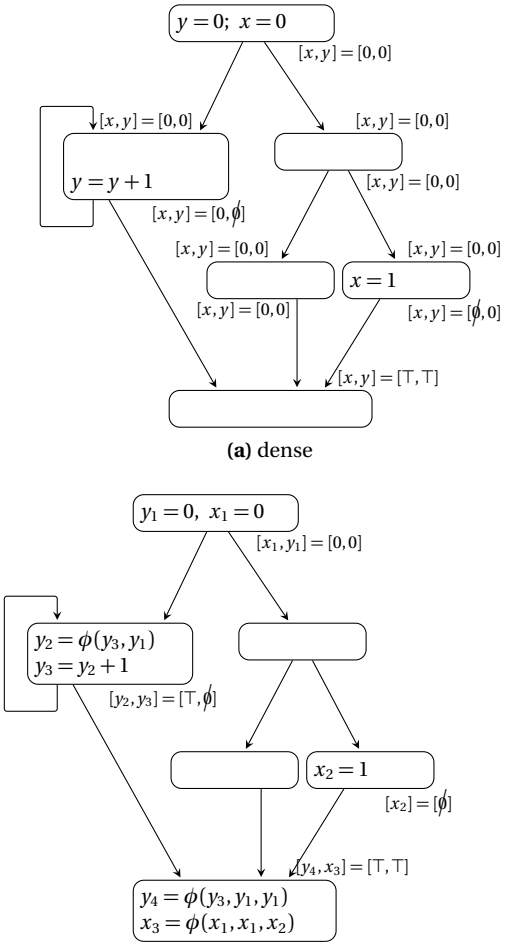


Fig. 1.1 Example control flow graph for non-zero value analysis, only showing relevant definition statements for variable x and y.

1.4 SSA in Context

1.4.1 Historical Context

Throughout the 1980s, as optimizing compiler techology became more mature, various intermediate representations (IRs) were proposed to encapsulate data

dependence in a way that enabled fast and accurate data flow analysis. The motivation behind the design of such IRs was the exposure of direct links between variable definitions and uses, known as *def-use chains*, enabling efficient propagation of data-flow information. Example IRs include the program dependence graph [3] and program dependence web [5]. Chapter **??** gives further details on dependence graph style IRs.

Static single assignment form was one such IR, which was developed at IBM Research, and announced publicly in several research papers in the late 1980s [6, 1, 2]. SSA rapidly acquired popularity due to its intuitive nature and straightforward construction algorithm. The SSA property gives a standardized shape for variable def-use chains, which simplifies data flow analysis techniques.

1.4.2 Current Usage

Added more compilers. Removed the references (don't know if I should have). Removed the part on -O that was not true for all compilers (some of them always use SSA). Removed the section on compilers that were roginaly not under SSA and are now... I think this was useless or incomplete.

The majority of current commercial and open-source compilers, including GCC, Open64, Sun's HotSpot JVM, IBM's Java Jikes RVM, Chromium V8 JavaScript engine, Mono, or LLVM, use SSA as a key intermediate representation for program analysis. As optimizations in SSA are fast and powerful, SSA is increasingly used in just-in-time (JIT) compilers that operate on a high-level target-independent program representation such as Java byte-code, CLI byte-code (.NET MSIL), or LLVM bitcode.

Initially developed to facilitate the development of high-level program transformations, SSA form has gained much interest due to its favorable properties that often allow to simplify algorithms and reduce computational complexity. Today, SSA form is even adopted for the final code generation phase (see Part **??**), i.e., the back-end. Several industrial and academic compilers, static or just-in-time, use SSA in their back-ends, e.g., LLVM, Java HotSpot, LAO, libFirm, Mono. Many industrial compilers that use SSA form perform SSA elimination before register allocation, including GCC, HotSpot, Jikes, and LLVM. Recent research on register allocation (see Chapter **??**) even allows to retain SSA form until the very end of the code generation process.

1.4.3 SSA for High-Level Languages

So far, we have presented SSA as a useful feature for compiler-based analysis of low-level programs. It is interesting to note that some high-level languages enforce the SSA property. The SISAL language is defined in such a way that programs automatically have referential transparency, since multiple assignments are not permitted to variables. Other languages allow the SSA property to be applied on a per-variable basis, using special annotations like `final` in Java, or `const` and `readonly` in C#.

The main motivation for allowing the programmer to enforce SSA in an explicit manner in high-level programs is that *immutability simplifies concurrent programming*. Read-only data can be shared freely between multiple threads, without any data dependence problems. This is becoming an increasingly important issue, with the trend of multi- and many-core processors.

High-level functional languages claim referential transparency as one of the cornerstones of their programming paradigm. Thus functional programming supports the SSA property implicitly. Chapter **??** explains the dualities between SSA and functional programming.

1.5 About the Rest of this Book

In this chapter, we have introduced the notion of SSA. The rest of this book presents various aspects of SSA, from the pragmatic perspective of compiler engineers and code analysts. The ultimate goals of this book are:

1. To demonstrate clearly the *benefits* of SSA-based analysis.
2. To dispell the *fallacies* that prevent people from using SSA.

This section gives pointers to later parts of the book that deal with specific topics.

1.5.1 Benefits of SSA

SSA imposes a strict discipline on variable naming in programs, so that each variable has a unique definition. Fresh variable names are introduced at assignment statements, and control-flow merge points. This serves to simplify the structure of variable *def-use* (see Section **??**) relationships and *live ranges* (see Section **??**), which underpin data flow analysis. Part **??** of this book focus on data flow analysis using SSA.

There are three major advantages that SSA enables:

Compile time benefit Certain compiler optimizations can be more efficient when operating on SSA programs, since referential transparency means that data flow information can be associated directly with variables, rather than with variables at each program point. We have illustrated this simply with the non-zero value analysis in Section 1.3.

Compiler development benefit Program analyses and transformations can be easier to express in SSA. This means that compiler engineers can be more productive, in writing new compiler passes, and debugging existing passes. For example, the *dead code elimination* pass in GCC 4.x, which relies on an underlying SSA-based intermediate representation, takes only 40% as many lines of

code as the equivalent pass in GCC 3.x, which does not use SSA. The SSA version of the pass is simpler, since it relies on the general-purpose, factored-out, data-flow propagation engine.

Program runtime benefit Conceptually, any analysis and optimization that can be done under SSA form can also be done identically out of SSA form. Because of the compiler development mentioned above, several compiler optimizations show to be more effective when operating on programs in SSA form. These include the class of *control-flow insensitive analyses*, e.g. [4].

This remark made to me many times is correct. There is a program runtime benefit because SSA is well suited to develop efficient optimizations with less efforts.

1.5.2 Fallacies about SSA

Some people believe that SSA is too cumbersome to be an effective program representation. This book aims to convince the reader that such a concern is unnecessary, given the application of suitable techniques. The table below presents some common myths about SSA, and references in this first part of the book containing material to dispell these myths.

Shortened the myth sentence

Myth	Reference
SSA greatly increases the number of variables	Chapter **??** reviews the main varieties of SSA, some of which introduce far fewer variables than the original SSA formulation.
SSA property is difficult to maintain	Chapters **??** and **??** discuss simple techniques for the repair of SSA invariants that have been broken by optimization rewrites.
SSA destruction generates many copy operations	Chapters **??** and **??** present efficients and effective SSA destruction algorithms.

References

1. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM.
2. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM.
3. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
4. Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 97–105, 1998.
5. Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 257–271, New York, NY, USA, 1990. ACM.
6. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.