Chapter 1

Introduction —(*J. Singer*)

.

Progress: 70%

Structural reordering 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. *Could we have a cartoon picture here?*

dynamic single assignment or simple single assignment is a _compiler_ IR also used for automatic parallelisation. Static means textualy.

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 is particularly relevant for program analysis that occurs inside a compiler, prior to execution. 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 address labelled as x) should be modified to store the value $(y + 1)$.

1 Definition of SSA

The seminal paper on SSA [4] gives an informal prose definition as follows:

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

This is the simplest, least constrained, definition of SSA. However there are various, more specialized, varieties of SSA, which impose further constraints on programs. Such

not defined yet.
Say something such as:
"every variable use as to be statically defined (no path from the program entry to the use with any def)

constraints are generally expressed in terms of dominance properties of variable definitions, or the presence or absence of certain pseudo-functions at control-flow merge points. Each distinct SSA variety has specific characteristics—many of these are discussed in Chapter **??**.

too vague. I would remove.
extensions are:
- more live-range splitting
- more information attached to the phi (eg gated SSA)
- SSA for memory
...

One important property that holds for all varieties of SSA, including the simplest definition above, is *referential transparency*. 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 `then` branch following a statement such as:

```
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.

```
x := 1;
y := x + 1;
x := 2;
z := x + 1;
```

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 $x1$ and $x2$ are equal.

```
x1 := 1;
y  := x1 + 1;
x2 := 2;
z  := x2 + 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 this is useful for debugging the compiler

by adding integer subscripts to original variable names; however in general this is an unimportant implementation feature.

The ϕ-*function* is the most important SSA concept to grasp. It is a special statement, known as a *pseudo-assignment* function. Appel [2] calls it a 'notational fiction.' [1] The purpose of a ϕ-function is to merge values from different incoming paths, at control-flow merge points.

Consider the following example:

```
x := input();
if (x == 42)
    y := 1;
else
    y := x*2;
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 $y1$ and $y2$. However the `print` statement could use either variable, dependent on the outcome of the `if` conditional test. A ϕ-function introduces a new variable $y3$, which takes the value of either $y1$ or $y2$. Thus the SSA version of the program is:

```
x := input();
if (x == 42)
    y1 := 1;
else
    y2 := x*2;
y3 := \phi(y1,y2);
print(y3);
```

in fact SSA not necessary restricted to CFG representation eg Cliff Click's representation See with Sebastian Hack about see of node (same IR in libfirm) and Lennart Beringer for placing phi-nodes elsewhere

In terms of their position, ϕ-functions are always placed at the head of basic blocks. A ϕ-function at block b has n parameters if there are n incoming control-flow paths to b. The behaviour 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.

It is important to note that, if there are multiple ϕ-functions at the head of a basic block, then these are executed 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 Section **??**). When ϕ-functions are eliminated in the SSA destruction phase, they are replaced by ~~parallel~~ copy operations, as described in Section **??**.

[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.

Yes for programmable architectures.
For circuit synthesis the phi
are actually implemented

Strictly speaking, ϕ-functions are not directly executable, 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 only used during static analysis of the program. They are removed before any program interpretation or execution takes place. There are various executable extensions of ϕ-functions, such as γ-functions, which take an extra parameter to specify which source value to assign to the target variable. Section **??** presents further information on these advanced extensions.

add chater ref

too vague

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

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

In the SSA version below, we abuse conventional C/Java style loop syntax to enable better program comprehension. We require initial variable assignments to be executed before each evaluation of the loop conditional test. These ϕ-functions in the loop hearder merge incoming definitions from before the loop for first iteration, and from the loop body for subsequent iterations. (Obviously a syntactically correct transformation would rely on `goto` statements to achieve the same effects.)

what do you mean here?

```
x1 := 0;
y1 := 0;
while ( x2 := \phi(x1, x3);
        y2 := \phi(y1, y3);
        x2 < 10
      ) {
  y3 := y2 + x2;
  x3 := x2 + 1;
}
print(y2)
```

It is important to note that the static single assignment property 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 defined with fresh values at each loop iteration.

3 Comparison with Classical Data Flow Analysis

Data flow analysis collects information about programs 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.

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. 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.

The rest of this section presents a trivial example data flow analysis: **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. Figure 1 shows the lattice of data flow facts. The analysis will assign an abstract value from this lattice to each variable at each point in the program text.

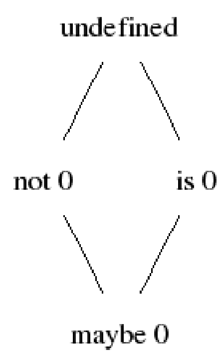

Fig. 1. Lattice of data flow facts for non-zero value analysis

Figure 2(a) shows an example program CFG. The full control flow structure of the program is given, however it only shows definition statements relating to variable x. Figure 2(b) shows the SSA version of this example program. Full details of the SSA construction algorithm are given in Chapter **??**. For now, it is sufficient to see that:

1. Integer subscripts have been used to rename variable x from the original program.
2. We have assumed an implicit definition of x_0 at the entry point of the program.
3. A ϕ-function has been inserted at the appropriate control-flow merge point where multiple reaching definitions of x converged in the original program.

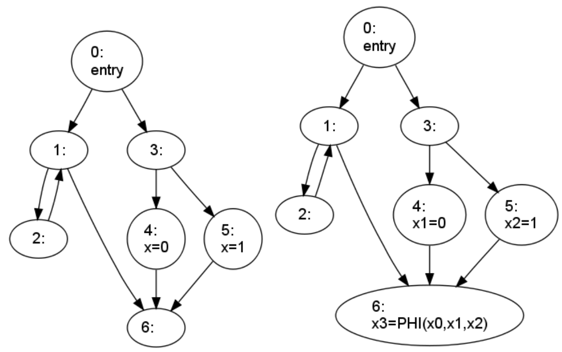

Fig. 2. Example control flow graph for non-zero value analysis, only showing relevant definition statements for variable x, in both (a) non-SSA and (b) SSA form.

With classical data flow analysis on the CFG in Figure 2(a), we would compute information about variable *x* for each of the seven nodes in the CFG, using suitable data flow equations. This might give results such as those shown in Table 1.

CFG node	abstract value of *x*
0	undefined
1	undefined
2	undefined
3	undefined
4	is 0
5	not 0
6	maybe 0

Table 1. Results of the CFG data flow analysis

Using SSA-based data flow analysis on Figure 2(b), we compute information about each variable based on a simple analysis of its definition statement. This gives us four data flow facts, one for each SSA version of variable *x*, as shown in Table 2.

This 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).

SSA variable	abstract value
x_0	undefined
x_1	is 0
x_2	not 0
x_3	maybe 0

Table 2. Results of the SSA data flow analysis

Part II *FIXME - forward ref* of this textbook gives a comprehensive treatment of SSA-based data flow analysis.

4 SSA in Context

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 [5] and program dependence web [9]. 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 [10,1,3]. 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.

4.2 Current Usage

The majority of current commercial and open-source compilers use SSA as a key intermediate representation for program analysis. SSA is generally enabled with the `-O` flag in ahead-of-time compilers (since SSA construction is ~~fairly expensive, and~~ only required for optimization). Similarly for just-in-time compilers, only the *hot* methods will be recompiled with SSA-based optimizations.

this is NOT expensive for an ahead-of-time compiler!

Recent optimizing compiler infrastructures, e.g. LLVM, use SSA from the ground up. Other compilers, e.g. GCC, began development before SSA was well-characterized or widely known. In such cases, SSA support can be back-ported into the original optimization framework. For instance, GCC uses SSA as of version 4.0 [7,8].

I do not know if it is already in production, but there are solutions with the byte code already in SSA (or some annotations to be able to build it quickly)

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.

5 Benefits of SSA

switch this section with section 4.

In this chapter, we have introduced the notion of SSA. We now conclude by enumerating some of the benefits that it provides.

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 variable def-use relationships, which underpin data flow analysis. There are three major advantages that SSA enables:

Program runtime benefit Certain compiler optimizations can be more effective when operating on programs in SSA form. These include the class of *control-flow insensitive analyses*, e.g. [6].

go through every chapter of the book and add the corresponding reference (instead of reference to a paper).

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 3.

add to this:
- implicit use-def chains
- structural properties of live-ranges (such as dominance)

=> forward propagation super easy, liveness check

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.

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

I would like you to say here that some people may think that SSA is bad because:
- it increases too much the number of variables
- we do not know how to get rid of phi functions and it adds too many copies
- it is a property that is difficult to update

One of the goal of this book it to convince the reader that this is not correct. See SSA destruction chapter (coalescing is easy opt problem); see update chapter (update can be a black box of the IR).
Optimizing consists of destructing and reconstructing. Sticking on the original code (ie trying to keep the original variables up to the op-code) avoids many transformations. Here the SSA destruction is easy, so lets get rid of all the original variables to build a nice, developer friendly IR: SSA Form

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. Andrew W. Appel. *Modern Compiler Implementation in* {*C,Java,ML*}. Cambridge University Press, 1997.
3. 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.
4. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
5. 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.
6. 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.
7. Diego Novillo. Tree SSA—a new optimization infrastructure for GCC. In *Proceedings of the First Annual GCC Developers' Summit*, pages 181–193, 2003.
8. Diego Novillo. Design and implementation of Tree SSA. In *Proceedings of the Second Annual GCC Developers' Summit*, pages 119–130, 2004.
9. 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.
10. 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.