CHAPTER 1

Propagating Information using SSA

F. Brandner
D. Novillo

Progress: 50%

Review in progress

1.1 Overview

A central task of compilers is to *optimize* a given input program such that the resulting code is more efficient in terms of execution time, code size, or some other metric of interest. However, in order to perform optimizations and program transformations typically some form of *program analysis* is required in order to determine if a given transformation is applicable, to estimate its profitability, or guarantee correctness.

Data flow analysis [11] is a simple, yet powerful, approach to program analysis that is utilized by many compiler frameworks and program analysis tools today. We will introduce the basic concepts of traditional data flow analysis in this chapter and show that *static single assignment* form (SSA) facilitates the design and implementation of equivalent analyses, while leveraging properties of programs in SSA form allows to reduce the compilation time and memory consumption.

Traditionally, data flow analysis is performed on a *control flow graph* (CFG) representation of the input program, where nodes in the graph represent operations and edges the possible flow of program execution. Information on certain *program properties* is propagated among the nodes along the control flow edges until the computed information at the nodes stabilizes, i.e., no *new* information can be devised from the program.

The *propagation engine* presented in the following sections is an extension of the well known approach by Wegman and Zadeck for *sparse conditional constant*

propagation [18] (also known as SSA-CCP). Instead of using the CFG they represent the input program as an *SSA graph* [6]. Operations are again represented as nodes in this graph, however, the edges represent *data dependencies* instead of control flow. This representation allows a selective propagation of program properties among data dependent graph nodes only. As before, the processing stops when the information at the graph's nodes stabilizes. The basic algorithm is not limited to constant propagation and can also be applied to solve a large class of other data flow problems efficiently [12]. However, not all data flow analyses can be modeled. We will thus investigate the limitations of the SSA-based approach and briefly discuss problems that cannot be modeled.

The remainder of this chapter is organized as follows. First, the basic concepts of (traditional) data flow analysis are presented in Section 1.2. This will provide the theoretical foundation and background for the discussion of the SSA-based propagation engine in Section 1.3. We then provide examples of data flow analyses that can be realized efficiently by the aforementioned engine, namely copy propagation in Section 1.4.1 and value range propagation in Section 1.4.2. Finally, some examples of data flow problems are given that cannot be modeled using the generic propagation engine but still benefit from SSA properties in Section 1.5. We conclude and provide links for further reading in Section 1.6.

1.2 Preliminaries

Data flow analysis is at the heart of many compiler transformations and optimizations, but also finds application in a broad spectrum of analysis and verification tasks in program analysis tools such as program checkers, profiling tools, and timing analysis tools. This section gives a brief introduction to the basics of data flow analysis, due to space considerations, we cannot cover this topic in full depth.

As noted before, data flow analysis allows to derive information of certain interesting program properties that may help to optimize the program. Typical examples of interesting properties are: The set of *live* variables at a given program point, the constant value a variable may take, or the set of program points that are reachable at run-time. Liveness information is critical, for example, during register allocation, while the two latter properties help in simplifying computations and avoiding useless calculations as well as dead code.

The analysis results are gathered from the input program by propagating information among its operations along all possible execution paths. The propagation is typically performed iteratively until the computed results stabilize. Formally, a data flow problem can be specified using a *monotone framework* that consists of:

- a *complete lattice* representing the property space,
- a *flow graph* resembling the control flow of the input program,
- and a set of *transfer functions* modeling the effect of individual operations on the property space.

I would prefer to avoid this formalism

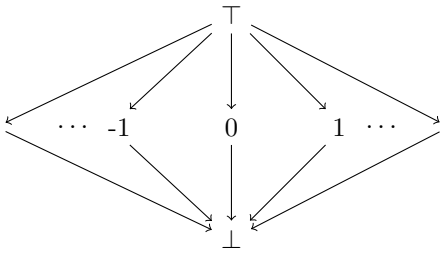


Fig. 1.1 Example of a lattice, which is commonly used to represent whether a variable is known to hold a constant value at a given program point.

1.2.1 Property Space

A key concept for data flow analysis is the representation of the property space via *partially ordered sets*. A partially ordered set $(L, \sqsubseteq)$ is a set L that is equipped with a reflexive, transitive, and anti-symmetric relation $\sqsubseteq\colon L \times L \to \{true, false\}$. Based on the relation an *upper bound* for subsets of L can be defined: $l \in L$ is the upper bound for a subset Y of L if $\forall l' \in Y\colon l' \sqsubseteq l$. Furthermore, $l \in L$ is the *least upper bound* if it is an upper bound and for any other upper bound l_o of $Y\colon l \sqsubseteq l_0$. Lower bounds and greatest lower bounds are defined analogously.

A particularly interesting class of partially ordered sets are *complete lattices*, where all subsets have a least upper bound as well as a greatest lower bound. These bounds are unique and are denoted by $\bigsqcup$ and $\bigsqcap$ respectively. In the context of program analysis the former is often referred to as the *join operator*, while the latter is termed the *meet operator*. Complete lattices have two distinguished elements, the *least element* and the *greatest element*, often denoted by $\bot$ and $\top$ respectively.

An *ascending chain* is a totally ordered subset $\{l_1, \ldots, l_n\}$ of a complete lattice, where for all $i, j \in \{1, \ldots, n\}, i < j\colon l_i \sqsubseteq l_j$. A chain is said to *stabilize* if there exists an $i_0 \in \mathbb{N}$, where $\forall i \in \mathbb{N}, i > i_0\colon l_i = l_{i_0}$; i_0 is then called the *length* of the chain.

Consider, for example, the lattice presented in Figure 1.1 that is commonly used to represent whether a given variable is known to hold a specific constant value at a given program point at run-time. The analysis and the corresponding optimization are often uniformly referred to as *constant propagation*. For this lattice, $\top$ indicates that no specific information on the variable's value is available. The symbol $\bot$, on the other hand, indicates that the analysis was not able to prove that the variable holds the same constant value in all cases. The other elements in the lattice denote that the variable is known to hold the respective value. The $\sqsubseteq$ relation is represented by the arrows in the figure. Clearly, all chains for this lattice are at most of length three ($\bot \sqsubseteq c \sqsubseteq \top, c \in \mathbb{N}$).

1.2.2 Program Representation

Individual functions of the input program are represented as separate flow graphs of the form $G = (V, E)$, where the nodes represent operations, or instructions, at a program point and edges denote the possible flow of control at run-time. The graph contains two distinguished nodes, the *start* node s and the *exit* node t. These two nodes are special in that the former is the only node without any predecessors and the latter is the only node without any successors.

During the analysis, data flow information is propagated from one node to another adjacent node along the respective graph edge. Every node is associated with data flow information using two sets, *in* and *out*. If the target node has a single incoming edge only, the propagation corresponds to a simple copy from the source node's *out* set to the target node's *in* set. However, in the more general case of multiple incoming edges, we need to combine the information from all those edges. This is accomplished by applying the join or meet operator of the lattice that represents the property space. For example, consider the lattice from Figure 1.1 and the flow graph shown in Figure 1.2. We can clearly say that, after executing operation 1 on the top left side, variable x holds the constant value of 4, as indicated by the *out* annotation $(x, 4)$ below. The same applies for operation 2 on the right, with the slight difference that the variable holds the value of 5. Combining this information yields $4 \sqcap 5 = \bot$, i.e., x does not hold a constant value at program point 3.

In many cases it is helpful to reverse the flow graph to propagate information, i.e., reverse the direction of the graph's edges. Analyses relying on this reversed flow graph are termed *backward analyses*, while those using the regular flow graph are called *forward analyses*. As shown before, computing whether a variable holds a certain constant value is a forward problem, while determining whether a computation is dead, i.e., not used, is a typical backward problem.

1.2.3 Transfer Functions

Aside from the control flow, the effect of the program's operations need to be accounted for during analysis. However, in contrast to the actual execution, we are only interested in an abstract model of the operations' effects with respect to the

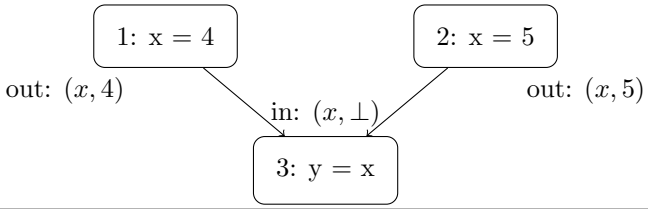


Fig. 1.2 Combining data flow information using the join operator.

data flow information. Operations are thus mapped to a set of abstract *transfer functions* of the form $f_{op}\colon L \to L$ that transform the information available from the *in* set of the respective flow graph nodes and store the result in the corresponding *out* set.

Continuing the example from before, a simple set of transfer functions can be defined, as shown in Table 1.1. Assignments to variables invalidate the information that is available on that particular variable, i.e., information is removed from the input set. However, in the case of a copy operation or the assignment of a constant value, new information is obtained, i.e., is appended to the input set. Care has to be taken that previous information is properly invalidated. Other operations not modifying any variable are associated with the identity function, i.e., all the information is preserved.

1.2.4 Solving Data Flow Problems

Putting all those elements together, i.e., a complete lattice, a flow graph, and a set of transfer functions, yields an instance of a monotone framework. Which, in fact, describes a set of *data flow equations* that need to be solved in order to retrieve the analysis result. A very popular and intuitive way of solving these equations is to compute the *maximal fixed point* (MFP) using an iterative work list algorithm. The work list contains operations that have to be reevaluated by first combining the information form all their predecessors in the flow graph, then applying their transfer function, and finally propagating the obtained information to all successors by appending them to the work list. The algorithm terminates when the data flow information stabilizes and the work list becomes empty [11].

It is obvious that a single flow edge can be appended several times to the work list in the course of the analysis. It may even happen that an infinite feedback loop prevents the algorithm from terminating. We are thus interested in bounding the number a given edge is processed. Recalling the definition of chains from Section 1.2.1, the *height* of a lattice is defined by the length of its longest chain. We can ensure termination for lattices fulfilling the *ascending chain condition*, which ensures that the lattice has finite height. Given a lattice with finite height h and a flow graph $G = (V, E)$ it is easy to see that the MFP solution can be computed in $O(|E| \cdot h)$ time, or, due to the fact that $|E| \leq |V|^2$, in $O(|V|^2 \cdot h)$. Note that the height of the lattice often depends on properties of the input program, which might ultimately

this is not trivial. provide example eg #variables for constant propagation

I do not like this: too formal.

Operation	Transfer Function
x = c, $c \in \mathbb{N}$	$f_{x=c}(L) = L \setminus \{(x, c') \mid (x, c') \in L\} \cup \{(x, c)\}$
x = y	$f_{x=y}(L) = L \setminus \{(x, c') \mid (x, c') \in L\} \cup \{(x, c) \mid (y, c) \in L\}$
x = ...	$f_{x=\ldots}(L) = L \setminus \{(x, c') \mid (x, c') \in L\}$
...	$f_{\ldots}(L) = L$

Table 1.1 A simplified set of transfer functions for constant propagation analysis.

yield bounds worse than cubic in the number of graph nodes. Furthermore, every node of the flow graph is associated with an *in* and an *out* set, in order to store data flow information and propagate preliminary analysis results to the neighbors in the graph. The sets often contain information that is unrelated to the particular nodes in question in order to ensure that the information is properly propagated to all relevant program points.

Would prefer to see this in the last section

For reducible flow graphs the order in which operations are processed by the work list algorithm can be optimized [8, 10, 11], allowing to derive tighter complexity bounds. However, relying on reducibility is problematic, because flow graphs are often *not* reducible even for proper structured languages, e.g., reversed flow graphs of backward problems can be, and in fact almost always are, irreducible even for programs with reducible flow graphs. Furthermore, experiments have shown that the tighter bounds not necessarily lead to improved compilation times [5].

control

control

Apart from computing a fixed point, data flow equations can also be solved using a more powerful approach called the *meet over all paths* (MOP) solution. Even though more powerful, computing the MOP solution is often harder to compute or even undecidable [11]. Consequently, the MFP solution is preferred in practice.

Give the intuition of what it is

1.3 Propagation Using the SSA Form

SSA form allows to solve a large class of data flow problems more efficiently than the standard fixed point solution presented previously. The basic idea is to directly propagate information computed at the unique definition of a variable to all its uses. Intermediate program points on regular control flow paths from the definition to those uses are skipped, reducing memory consumption and compilation time.

or from uses to def (eg deadcode)

1.3.1 Program Representation

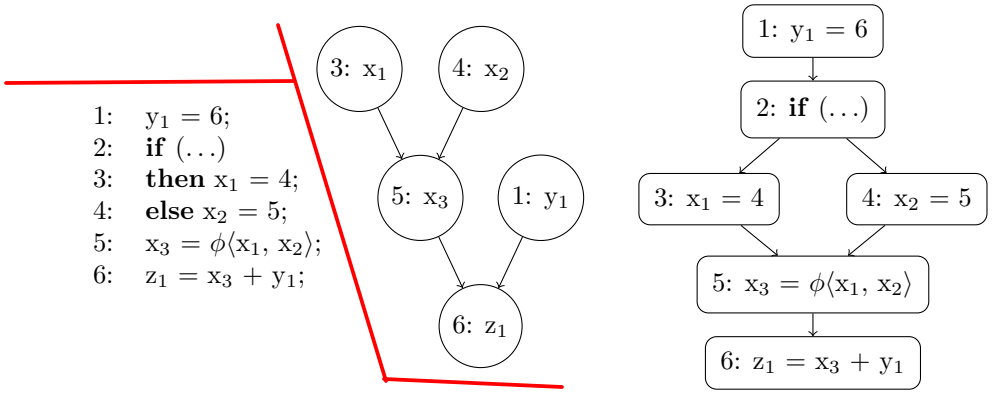
Programs in SSA form exhibit, aside from the regular operations of the original program, special operations called ϕ-operations, which are placed at join points of the original CFG – see Part **??** of this book for more information. In the following, we assume that possible many ϕ-operations are associated with the corresponding CFG nodes at those join points.

Reference for SSA graphs: Point directly to a chapter in the book?

Data flow analysis under SSA form relies on a specialized program representation based on *SSA graphs* [6], which allows to simplify the propagation of data flow information. The nodes of an SSA graph correspond to the operations of the program. In particular the program's ϕ-operations are represented by dedicated nodes in the SSA graph. The edges of the graph connect the unique definition of a variable with all its uses, i.e., an edge represents a true data dependence among two nodes.

SSA graph == use-def chains
- use the term use-def chains instead of SSA graph
- here you want def-use chains that has to be computed
- some simple "propagation" engines do _ not_ require def-use chains as they traverse the program using a topological order of the dom-tree (so cannot handle loops unless they iterate as much as the max loop depth)

As in graph chapter, put the operations in the nodes



(a) SSA pseudo code. (b) SSA-graph. (c) Control flow graph.

Fig. 1.3 Example program and its SSA graph.

An important property of SSA graphs is that they also capture, besides the data dependencies, the *relevant* join points of the program's CFG. A join point is relevant for the analysis, whenever the value of two or more definitions may reach a use by passing through that join. Due to the properties of SSA form, it is ensured that a ϕ-operation is placed at the join point and that the use has been properly updated to refer to the respective ϕ-operation. Other join points can safely be ignored, because uses are guaranteed to receive the same value on all paths due to the dominance property.

I would remove this useless sentence.
Actually this is the single assignment property that is important here not the dominance property.

Consider for example, the code excerpt shown in Figure 1.3, along with the corresponding SSA graph and CFG. Assume we are interested in propagating information from the assignment of variable y_1 at the beginning of the code excerpt down to its unique use at the end. Using the traditional CFG representation this requires the propagation to pass though several intermediate program points. These program points are concerned with computations of the variables x_1 through x_3 only and are thus irrelevant for the computation at hand. The SSA graph representation, on the other hand, allows to propagate the desired information directly without any intermediate steps. At the same time, we also find that the control flow join following the **if** is properly represented by the ϕ-operation defining the variable x_3 in the SSA graph.

Even though the SSA graph captures data dependencies and the relevant join points in the CFG, it lacks information on other *control dependencies*. However, analysis results can often be improved significantly by considering the additional information that is available from the control dependencies in the program's CFG. As an example consider once more the code excerpt shown in Figure 1.3. Assume that the condition associated with the **if** operation is known to be false for all possible program executions. Consequently, the ϕ-operation will select the value of x_2 in all cases, which is known to be of constant value 5. However, due to the shortcom-

1. Initially, every edge in the CFG is marked not executable and the *FlowWorkList* is seeded with the outgoing edges of the flow graph's *start* node. The *SSAWorkList* is empty.
2. Remove the top element of either of the two work lists.
3. If the element is a flow edge that is marked to be executable, do nothing, otherwise proceed as follows:
   - Mark the edge as executable.
   - Visit every ϕ-operation associated with the edge's target node.
   - When the target node was reached the first time via the *FlowWorkList* visit its operation.
   - When the target node has a single outgoing non-executable edge append that edge to the *FlowWorkList*.
4. If the element is an edge from the SSA graph, process the target operation as follows:
   a. When the target operation is a ϕ-operation visit that ϕ-operation.
   b. For regular target operations, examine the corresponding executable flag of the incoming edges of its corresponding flow graph node. If any of those edges is marked executable visit the operation, otherwise do nothing.
5. Continue with step 2 until both work lists become empty.

Algorithm 1: Sparse Data Flow Propagation

At this point this is not clear what you mean. Please precise (CFG & def-use chains)

ings of the SSA graph this information cannot be derived. It is thus important to use both graphs during data flow analysis in order to obtain the best possible results.

1.3.2 Sparse Data Flow Propagation

Similar to monotone frameworks for traditional data flow analysis, frameworks for *sparse data flow propagation* under SSA form can be defined. As before, such a framework consist of: (1) a complete lattice, (2) a set of transfer functions, (3) a control flow graph, and, additionally, (4) an SSA graph. We again seek a maximal fixed point solution (MFP) using an iterative work list algorithm. However, in contrast to the algorithm described before, data flow information is not propagated along the edges of the flow graph but along the edges of the SSA graph. For regular uses the propagation is straightforward due to the fact that every use receives its value from a unique definition. Special care has to be taken for ϕ-operations, which select a value among their operands depending on the incoming flow edges. The data flow information of the incoming operands has to be combined using the meet or join operator of the lattice. As data flow information is propagated along SSA edges that have a single source, it is sufficient to store the data flow information with the node. The *in* and *out* sets used by the traditional approach – see Section 1.2.2 – are obsolete, since ϕ-operations already provide the required buffering. Furthermore, the flow graph is used in order to track which operations are not reachable under any program execution and thus can be ignored safely during the computation of the fixed point solution.

regroup 3 & 4 together: a propagation graph made of CFG + def-use chains

The algorithm processes two work lists, the *FlowWorkList* containing edges of the flow graph and the *SSAWorkList*, which consists of edges from the SSA graph.

1. Compute the operation's data flow information:
   a. If the operation is a ϕ-operation, combine the data flow information from all its operands where the corresponding flow edge is marked executable.
   b. In the case of conditional branches, update the operation's data flow information. Determine which of the outgoing flow edges are reachable from the corresponding flow graph node by examining the branch's condition(s) and append the respective non-executable edges to the *FlowWorkList*.
   c. For regular operations, update the corresponding data flow information by applying its transfer function.
2. Whenever the data flow information changes append all outgoing edges of the corresponding SSA graph node to the *SSAWorkList*.

Algorithm 2: Visiting an Operation

It proceeds by removing the top element of either of those lists and processing the respective edge as indicated by Algorithm 1. Throughout the algorithm operations of the program may be visited in order to update the work lists and propagate information as shown by Algorithm 2. We will highlight the most relevant parts of the algorithms in the following paragraphs and explain some important aspects of their interaction.

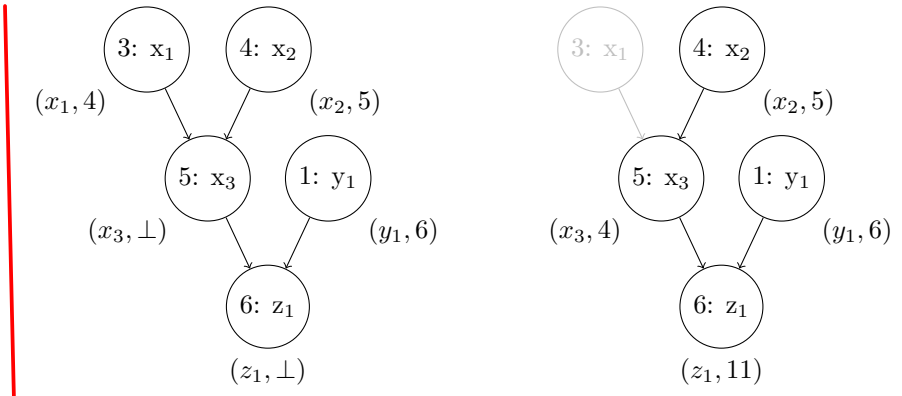
In step 3 of the main algorithm, flow edges are processed that were encountered to be executable for the first time in the course of the analysis. Whenever such a flow edge is processed, all ϕ-operations of its target node need to be reevaluated due to the fact that Algorithm 2a discarded the respective operands of the ϕ-operations so far – because the flow edge was not yet marked executable. Similarly, the operation of the target node has to be evaluated when the target node is encountered to be executable for the first time, i.e., the currently processed edge is the first of its incoming edges that is marked executable. Note that this is only required the *first* time the node is encountered to be executable, due to the processing of operations in Step 4b, which thereafter triggers the reevaluation automatically when necessary.

I would remove this

Regular operations as well as ϕ-operations are visited by Algorithm 2 when the corresponding flow graph node has become executable or whenever the data flow information of one of their predecessors in the SSA graph changed. ϕ-operations combine the information from multiple flow paths using the usual meet operator, but only considering those operands where the associated edge is marked executable. Conditional branches are handled by examining its conditions based on the data flow information computed so far. Depending on whether those conditions are satisfiable or not, flow edges are appended to the *FlowWorkList* in order to ensure that all reachable operations are considered during the analysis. Finally, all regular operations are processed by applying the relevant transfer function and possibly propagating the updated information to all uses by appending the respective SSA graph edges to the *SSAWorkList*.

As an example, consider the program shown in Figure 1.3 and the constant propagation problem. First, assume that the condition of the **if** cannot be statically evaluated, we thus have to assume that all its successors in the CFG are reachable. Consequently, all flow edges in the program will eventually be marked executable.

this is too vague. The important point is what is the result of "c==d" where V(c)=bottom & V(d)=3? If "V(c)=top" this is clear the result is top so both successors should be considered as executable. Can we safely consider that V(c) will eventually get a non bottom value and then consider the result of "bottom==3" to be bottom and mark both successors as non executable???



(a) All code reachable. (b) With unreachable code.

Fig. 1.4 Sparse data flow propagation using SSA graphs.

This will trigger the evaluation of the constant assignments to the variables x_1, x_2, and y_1. The transfer functions immediately yield that the variables are all constant holding the value 4, 5, and 6 respectively. This new information will trigger the reevaluation of the ϕ-operation of variable x_3. As both of its operands are reachable the combined information yields $3 \sqcap 5 = \bot$. Finally, also the assignment to variable z_1 is reevaluated, but the analysis shows that its value is not a constant as depicted by Figure 1.4a. If, however, the condition of the **if** is known to be false for all possible program executions a more precise result can be computed, as shown in Figure 1.4b. Neither the flow edge leading to the assignment of variable x_1 is marked executable nor its outgoing edge leading to the ϕ-operation of variable x_3. Consequently, the reevaluation of the ϕ-operation considers the data flow information of its second operand x_2 only, which is known to be constant. This finally enables to analysis to show that the assignment to variable z_1 is, in fact, constant as well.

During the course of the propagation algorithm, every edge of the SSA graph is processed at least once, whenever the operation corresponding to its definition is found to be executable. Afterward, an edge can be revisited several times depending on the height h of the lattice representing the analysis' property space. Edges of the flow graph, on the other hand, are processed at most once. This leads to an upper bound in execution time of $O(|E_{SSA}| \cdot h + |E_{FG}|)$, where E_{SSA} and E_{FG} represent the edges of the SSA graph and the flow graph respectively – see [18]. The size of the SSA graph increases with respect to the original non-SSA program. Measurements indicate that this growth is linear [6], yielding a bound that is comparable to the bound of traditional data flow analysis. However, in practice the SSA-based propagation engine outperforms the traditional approach. This is due to the direct propagation from the definition of a variable to its uses, without the costly intermediate steps that have to be performed on the CFG. The overhead is also reduced in terms of memory consumption. Instead of the *in* and *out* sets capturing the complete

#variables*#prog_points

property space on every program point in the program, it is sufficient to associate every node in the SSA graph with the data flow information of the corresponding variable. This leads to considerable savings in practice.

#SSA_variables

1.3.3 *Limitations*

Unfortunately, the presented approach also has its drawbacks in terms of general applicability. The problem arises from two sources: (1) the exclusive propagation of information between data-dependent operations and (2) the semantics and placement of ϕ operations. The former issue prohibits the modeling of data flow problems that propagate information to program points that are not directly related to either a definition or a use of a variable, while the latter prohibits the modeling of backward problems.

Actually this is fine if the property is (conservatively) correct for every point of the live-range.

Consider, for example, the well known problem of available expressions [11] that often occurs in the context of redundancy elimination. An expression is available at a given program point when the expression is computed and not modified thereafter on all paths leading to that program point. In particular, this might include program points that are independent from the expression and its operands, i.e., neither defines nor uses any of its operands. However, the SSA graph does not cover those program points as it propagates information directly from definitions to uses without any intermediate steps.

Why not having a single source would be a problem? Because of the "merged" In & Out? What about dead-code elimination?

Furthermore, data flow analysis using SSA graphs is limited to forward problems. Due to the structure of the SSA graph it is not possible to simply reverse the edges in the graph as it is done with flow graphs. For one, this would invalidate the nice property of having a single source for incoming edges of a given variable, as variables typically have more than one use. In addition, ϕ-operations are placed at join points with respect to the *forward* control flow and thus do not capture join points in the reversed flow graph. SSA graphs are consequently not suited to model backward problems in general.

A fully agree with this. This is the main reason for me.

1.4 Examples

Even though data flow analysis based on SSA graphs has its limitations, it is still a useful and effective solution for many interesting problems, as will be shown in the following sections.

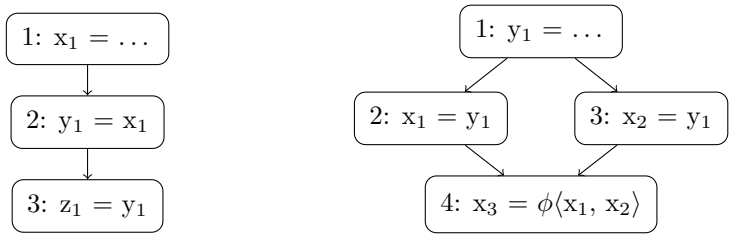
1.4.1 Copy Propagation

Copy propagation in SSA form is, in principle, very simple. Given the assignment $x = y$, all we need to do is traverse all the immediate uses of x and replace them with y, thereby effectively eliminating the original copy operation. However, such an approach will not be able to propagate copies past ϕ-operations, particularly those in loops. A more powerful approach is to split copy propagation into two phases. First, data flow analysis is performed in order to find copy-related variables throughout the program. Followed by a rewrite phase that eliminates spurious copies and renames variables.

The analysis for copy propagation can be described as the problem of propagating the *copy-of value* of variables. Given a sequence of copies as shown in Figure 1.5a. We say that y_1 is a *copy of* x_1 and z_1 is a *copy of* y_1. The problem with this representation is that there is no apparent link from z_1 to x_1. In order to handle transitive copy relations, all transfer functions operate on copy-of values instead of the direct source of the copy. If a variable is not found to be a copy of anything else, its copy-of value is the variable itself. For the example above, this yields that both, y_1 and z_1, are copies of x_1, which in turn is a copy of itself. The lattice of this data flow problem is thus similar to the lattice shown previously for constant propagation. However, the lattice elements correspond to variables of the program instead of integer numbers. The least element of the lattice represents the fact that a variable is a copy of itself.

Similarly, we would like to obtain the result that x_3 is a copy of y_1 for the example depicted in Figure 1.5b. This is accomplished by choosing the meet operator such that a copy relation is propagated whenever the copy-of values of all the ϕ-operation's operands match. So, when visiting the ϕ-operation for x_3, the analysis finds that x_1 and x_2 are both copies of y_1, which allows to propagate the desired result that x_3 is a copy of y_1.

The following example shows a more complex situation where copy relations are obfuscated by loops – see Figure 1.6. Note that the actual visiting order depends on the shape of the CFG and immediate uses, the ordering used here is meant for illustration only. Processing starts at the operation labeled 1, with both work lists empty and the data flow information $\top$ associated with all variables:



(a) Transitive copy relations. (b) Copy relations on ϕ-operations.

Fig. 1.5 Analysis of copy-related variables.

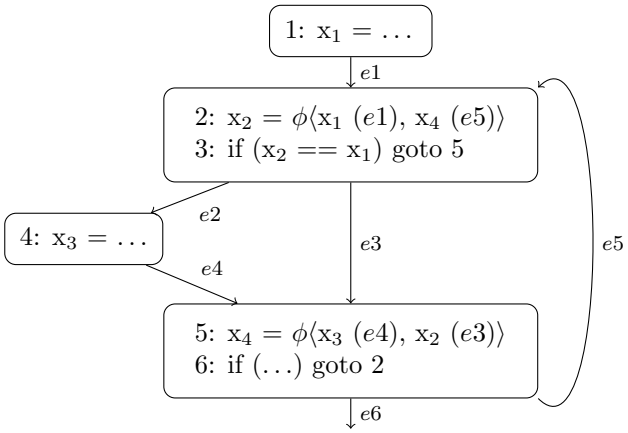


Fig. 1.6 ϕ-operations in loops often obfuscate copy relations.

1. Assuming that the value assigned to variable x_1 is not a copy, the data flow information for this variable is lowered to $\perp$, the SSA edges leading to operations 2 and 3 are appended to the *SSAWorkList*, and the flow graph edge $e1$ is appended to the *FlowWorkList*.
2. Processing the flow edge from the work list causes the edge to be marked executable and the operations labeled 2 and 3 to be visited. Since, edge $e5$ is not yet known to be executable the processing of the ϕ-operation yields a copy relation between x_2 and x_1. This information is utilized in order to determine which outgoing flow graph edges are executable for the conditional branch. Examining the condition shows that only edge $e3$ is reachable and thus needs to be added to the work list.
3. Flow edge $e3$ is processed next and marked executable for the first time. Furthermore, the ϕ-operation labeled 5 is visited. Due to the fact that edge $e4$ is not known the be executable, this allows to discover a copy relation between x_4 and x_1 (via x_2). The condition of the branch labeled 6 cannot be analyzed and thus causes its outgoing flow edges $e5$ and $e6$ to be added to the work list.
4. Now, flow edge $e5$ is processed and marked executable. Since the target operations are already known to be executable, only the ϕ-operation is revisited. However, variables x_1 and x_4 have the same copy-of value x_1, which is identical to the previous result computed in Step 2. Thus, neither of the two work lists is modified.
5. Assuming that the flow edge $e6$ leads to the exit node of the flow graph the algorithm stops after processing the edge without modifications to the data flow information computed so far.

The straightforward implementation of copy propagation, would have needed multiple passes to discover that x_4 is a copy of x_1. But the iterative nature of the propagation along with the ability to discover non-executable code allows to han-

dle even obfuscated copy relations. Moreover, this kind of propagation will only reevaluate the subset of operations affected by newly computed data flow information instead of the complete flow graph once the set of executable operations has been discovered.

1.4.2 Value Range Propagation

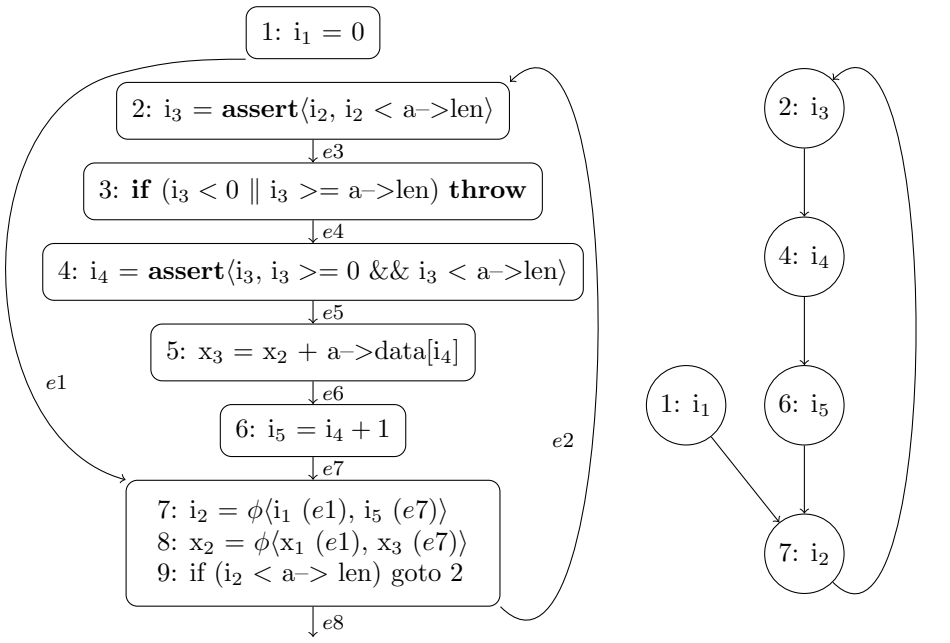
Value range propagation is similar to constant propagation, which served as a running example throughout this chapter. But instead of propagating single constant values, contiguous value ranges are propagated for all variables. For instance, the code in Figure 1.7 is extracted from a typical expansion of bound checking code in languages like Java. Notice how the bounds check at line 9 is redundant as variable *i* is guaranteed to take values in the range [0, a->len - 1]. An important source for range information are the conditions of branch operations, e.g., the condition associated with the **for** loop in the example.

Regular SSA graphs do not allow to take advantage of these conditions, because the range information only applies to the code region that is guarded by the respective branch. Furthermore, certain operations allow to derive range information as a side effect. Typical examples are arithmetic operations with implicit conditions such as the *minimum*, *maximum*, or the *absolute* operations. Trapping operations also provide information in case the execution succeeds, e.g., a pointer is known not to be `NULL` when a memory access succeeds. However, not all program points where this additional range information is available are captured by SSA graphs, which capture variable definitions, their uses, and ϕ-operations only.

The graph is thus preprocessed in order to split the live ranges of variables at appropriate program points whenever range information can be derived. The splitting can be accomplished by inserting artificial *assert* operations into the SSA graph that take an input variable and a condition as operands and return a copy of the input

```
1  struct array {
2    const int len;
3    int *data;
4  };
5
6  void
7  doit(array *a) {
8    for (int i = 0; i < a->len; ++i) {
9      if (i < 0 || i >= a->len)
10       throw;
11     x += a->data[i];
12   }
13 }
```

Fig. 1.7 Useless array bound checking code.



(a) Control flow with assertions. (b) Subset of the SSA graph.

Fig. 1.8 Splitting live ranges using additional *assert* expressions allows to derive more precise value ranges.

variable's value. Note that this *ad hoc* form of live range splitting could be generalized, e.g., by program representations such as *static single information* (SSI) [1, 16] or Extended SSA (*e*-SSA) [2] form – also see Chapter **??** of this book.

Reference: Missing reference to live range splitting chapter.

Following this live range splitting, a data flow problem is solved in order to propagate the range information available through the conditions of the assert expressions to the respective uses. The lattice representing the range information is more complex in structure than the previous examples. The lattice elements are divided into two classes, *regular* ranges and *anti* ranges, and consist of a lower and upper bound, where the bounds can be arbitrary symbolic expressions. Regular ranges specify that the actual values of a variable lie within the lower and upper bound, while for anti ranges the opposite is true. The meet operator is defined by combining the bounds such that the resulting range covers both of its arguments, i.e., in the case of regular ranges combining the information of $r_1 = [l_1, u_1]$ and $r_2 = [l_2, u_2]$ yields the range $[min(l_1, l_2), max(u_1, u_2)]$.

You can refer (in the conclusion) to abstract interpretation and the process of widening. Florian you can talk to Laure about it she might be able to give you some pointers.

However, a naive formulation of the lattice quickly leads to long or even infinite chains. The lattice thus needs to be restricted in order to guarantee termination of the analysis. As an example, consider a loop with an unconditional increment of a counter variable. This could lead to an infinite feedback look where the value range

associated with the counter grows infinitely on every iteration of the work list algorithm. In practice, infinite chains are prevented by choosing the transfer functions such that growth is restricted. For example, the popular open source compiler GCC, restricts the transfer functions of many operations, such as addition or subtraction, to constant singleton ranges only, i.e., ranges that consist of a single element that is known to be constant. Other compiler allow the range to grow within certain limits.

The augmented CFG of the original program from Figure 1.7 is shown in Figure 1.9a. Two additional *assert* expressions were inserted. The first, immediately following the conditional branch of the **for** loop, while the other is placed after the bounds check. For this example we are only interested in the value range of the counter variable i – and the corresponding variables in SSA form. Figure 1.9b thus only shows a subset of the complete SSA graph that is relevant for the counter variable. In the following the major steps of the data flow analysis are discussed:

1. The algorithm starts with the data flow information $\top$ assigned to all variables and by processing the operation labeled 1. The analysis derives the range $[0, 0]$ for variable i_1, which causes the ϕ-operations and the branch labeled 7-9 to be evaluated.
2. Since the flow edge $e7$ is not yet known to be executable the analysis derives the same range, $[0, 0]$, for variable i_2 of ϕ-operation 7.
3. Analyzing the condition of the branch operation 9 yields that both outgoing edges are executable, which triggers the analysis of the operations within the loop.
4. At first, however, examining the conditions of the *assert* expressions does not provide interesting ranges. The analysis merely retains the range $[0, 0]$ for both variables i_3 and i_4.
5. The analysis eventually reaches the increment of the counter variable labeled 6. As mentioned before, the transfer function for such increments has a huge impact on the quality of the analysis results and worst case complexity. For the sake of simplicity, assume that the increment is handled conservatively. The transfer function yields the range $[0, \infty]$ for variable i_5.[1]
6. Due to the updated range information of i_5 and the fact that the flow edge $e7$ is now known to be executable, the ϕ-operation of i_2 has to be reevaluated. Combining the range information computed by step 1 and 5 yields a new range $[0, \infty]$.
7. Note that at this point all flow edges have been marked executable, the following processing steps operate on the sparse SSA graph representation only and thus bypass unrelated computations.
8. First, the *assert* expression for variable i_3 is reevaluated. This time the expression's condition allows to clip the range of the input variable i_2 since the value of a->len is known to be smaller than ∞. The new range for i_3 is thus [0, a->len−1], which triggers the reevaluation of all its uses, in particular the array bounds check labeled with the number 3.

[1] Note that actual compilers have to assume a potential overflow of the counter variable, which further complicates the task of finding a proper transfer function.

9. The second *assert* expression is handled similarly and yields the same range, [0, a->len−1], for variable i_4. Again all of its uses are reevaluated.
10. Finally, the counter increment is reevaluated, however, the range for its variable i_5 does not change and the analysis terminates.

The final result of the analysis is used in order to eliminate array bounds checks, conditional branches, or unreachable code [2]. For example, the analysis result computed at Step 8 for the previous example, allows to safely eliminate the array bounds check. Furthermore, range information is valuable during the optimization of data representations, e.g., to reduce the number of bits for certain computations without loss of precision [17].

1.5 Beyond Sparse Data Propagation

We have seen that the generic propagation engine is capable to elegantly solve forward problems based on the sparse SSA graph. However, due to limitations of the graph, not all problems can be modeled. This section will highlight some problems where SSA form has proven helpful using specialized algorithms.

Dead Code Elimination

Dead code elimination (DCE) is a very important compiler optimization that eliminates computations that do not have any effect on the program output. This situation may arise from different sources. Code is called *unreachable* when it can never, under any circumstances, be executed at run-time. A common case of unreachable code is often related to conditional branches where the condition always evaluates to either true or false – the respective other case is unreachable, provided no other control path leads to the code. Another form of dead code arises from reachable computations, i.e., it is possible that the computations are carried out at run-time, that are never used by any subsequent computations. Dead or unreachable code can be removed in either case without impairing the program's correctness.

A popular algorithm for DCE, a typical backward data flow analysis, is due to Cytron et al. [6]. This algorithm exploits SSA form in order to eliminate both forms of dead code. It proceeds by initially marking all computations of a procedure to be dead. Computations are iteratively marked to be live using a work list algorithm according to the following criteria: (1) computations with observable side-effects such as I/O operations, function calls, and assignments to global or shared memory locations, (2) assignments to variables that are used by other live computations, (3) conditional branches where at least on control-dependent computation has been marked live. When the work list drains, no additional live computations can be discovered. The remaining code marked as dead can safely be eliminate. It is easy to

see that SSA form simplifies the processing of criteria (2) of this algorithm, since the relation between definitions and uses is captured naturally.

Live Variables

bof

Reference: Missing reference to liveness chapter.

Another important problem is the computation of live variables at all program points of a procedure or a subset thereof, which is usually solved using a backward data flow analysis. The approach is applicable to programs in SSA form, however, due to the increased number of variable names due to renaming during SSA construction, a considerable overhead is induced.

Fortunately, the live ranges, i.e., the set of program points where variables are live, offer properties under SSA form that facilitate the computation of liveness information. In fact, several options exist that allow to derive traditional liveness sets, where every program point is annotated with a set of live variables, or to perform on-demand liveness checks for individual variables [3]. In either case, the fact that uses of a variables are in any case dominated by its definition is exploited, which allows to prune the set of program point where variables are possibly live. We do not present details of these approaches and the involved algorithms, since Chapter **??** covers this topic in great detail.

1.6 Further Reading

Traditional data flow analysis is well established and well described in numerous papers. The book by Nielsen, Nielsen, and Hankin [11] gives an excellent introduction to the theoretical foundations and practical applications.

The sparse propagation engine, as presented in the chapter, is based on the underlying properties of SSA form. Other intermediate representations offer similar properties. *Static Single Information* (SSI) form [16] allows both backward and forward problems to be modeled by introducing σ operations, which are placed at program points where data flow information for backward problems needs to be merged [15]. Bodík uses an extended SSA form, *e*-SSA, in order to eliminate array bounds checks [2]. Ruf [14] introduces the *value dependence graph*, which captures both control and data dependencies. Using a set of transformations and simplifications a sparse representation of the input program can be derived which is suited for data flow analysis.

The *sparse evaluation graph* by Choi et.al [4] is based on the same basic idea as the approach presented in this chapter, intermediate steps are eliminating by bypassing irrelevant CFG nodes and merging the data flow information only when necessary. Their approach is closely related to the placement of ϕ-operations and similarly relies on the dominance frontier during construction. A similar approach, presented by Johnson and Pingali [9], is based on single-entry/single-exit regions.

The resulting graph is usually less sparse, but is also less complex to compute. Ramalingam [13] further extends these ideas and introduces the *compact evaluation graph*, which is constructed from the initial CFG using two basic transformations. The approach is superior to the sparse representations by Choi et.al as well as the approach presented by Johnson and Pingali.

The previous approaches derive a sparse graph suited for data flow analysis using graph transformations applied to the CFG. Duesterwald et.al [7] instead examine the data flow equations, eliminate redundancies, and apply simplifications to them.

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

References

1. C. S. Ananian. The static single information form. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1999.
2. Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 321–333, New York, NY, USA, 2000. ACM.
3. Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO '08: Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 35–44, New York, NY, USA, 2008. ACM.
4. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, New York, NY, USA, 1991. ACM.
5. Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. An empirical study of iterative data-flow analysis. In *CIC '06: Proceedings of the 15th International Conference on Computing*, pages 266–276, Washington, DC, USA, 2006. IEEE Computer Society.
6. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
7. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 357–373, London, UK, 1994. Springer-Verlag.
8. Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 207–217, New York, NY, USA, 1973. ACM.
9. Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 78–89, New York, NY, USA, 1993. ACM.
10. John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.
11. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
12. Diego Novillo. A propagation engine for GCC. In *Proceedings of the GCC Developers Summit*, pages 175–184, 2005.
13. Ganesan Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.
14. Erik Ruf. Optimizing sparse representations for dataflow analysis. In *IR '95: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, pages 50–61, New York, NY, USA, 1995. ACM.
15. Jeremy Singer. Sparse bidirectional data flow analysis as a basis for type inference. In *APPSEM'04: Web Proceedings of the Applied Semantics Workshop*, 2004.

16. Jeremy Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK, 2005.
17. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 108–120, New York, NY, USA, 2000. ACM.
18. M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.