CHAPTER 1

Propagating Information using SSA

F. Brandner
D. Novillo

Progress: 85% Minor polishing 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 to determine if a given transformation is applicable, to estimate its profitability, or guarantee correctness.

Data flow analysis [9] 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 us to reduce the compilation time and memory consumption.

Traditionally, data flow analysis is performed on a *control flow graph* representation (CFG) of the input program. Nodes in the graph represent operations and edges the potential flow of program execution. Information on certain *program properties* is propagated among the nodes along the control flow edges until the computed information 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* [15] (also known as SSA-CCP). Instead of using the CFG they represent the input program as an *SSA graph* [4]. 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 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 [10]. However, not all data flow analyses can be modeled, we will thus investigate the limitations of the SSA-based approach.

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 an example of a data flow analysis that can be realized efficiently by the aforementioned engine, namely copy propagation in Section 1.4. We conclude and provide links for further reading in Section 1.5.

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 particular constant value a variable may take, or the set of program points that are reachable at run-time. Liveness information, for example, is critical 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 considering all potential 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.

Property Space: A key concept for data flow analysis is the representation of the property space via *partially ordered sets* $(L, \sqsubseteq)$, where L represents some interesting program property and $\sqsubseteq$ a reflexive, transitive, and anti-symmetric relation. Using the $\sqsubseteq$ relation, *upper* or *lower bounds*, as well as *least upper* and *greatest lower bounds*, can be defined for subsets of L.

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. A chain is said to *stabilize* if there exists an index m, where $\forall i > m \colon l_i = l_m$. An analogous definition can be given for *descending chains*.

Program Representation: The functions of the input program are represented as control flow graphs, where the nodes represent operations, or instructions, and edges denote the potential flow of execution at run-time. Data flow information is then propagated from one node to another adjacent node along the respective graph edge using *in* and *out* sets associated with every node. In the simplest case this requires a simple copy, but if a node has multiple incoming edges, the information from those edges has to be combined using the meet or join operator.

Sometimes, it is helpful to reverse the flow graph to propagate information, i.e., reverse the direction of the edges in the control flow graph. Such analyses are termed *backward analyses*, while those using the regular flow graph are *forward analyses*.

Transfer Functions: Aside from the control flow, the operations of the program need to be accounted for during analysis, i.e., an abstract model of the operations' effects on the data flow information. Every operation is thus mapped to a *transfer function*, which transforms the information available from the *in* set of the operation's flow graph node and stores the result in the corresponding *out* set.

1.2.1 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 (minimal) fixed point* (MFP) using an iterative work list algorithm. The work list contains edges of the flow graph that have to be revisited by first combining the information from the *out* set of the edge's source node with the *in* set of the edge's target node using the meet or join operator, then applying the transfer function of the target node, and finally propagating the obtained information to all successors of the target node by appending them to the work list. The algorithm terminates when the data flow information stabilizes and the work list becomes empty.

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 of times a flow edge is processed. Recalling the definition of chains from

before, 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, where $|E|$ represents the number of edges. Since the number of edges is bounded by the number of graph nodes $|V|$, i.e., $|E| \leq |V|^2$, this gives $O(|V|^2 \cdot h)$. Note that the height of the lattice often depends on properties of the input program, which might ultimately yield bounds worse than cubic in the number of graph nodes. For instance, for copy propagation the lattice consists of the cross product of many smaller lattices, each representing the potential values of a variable occurring in the program. The total height of the lattice thus directly depends on the number of variables in the program.

In terms of memory consumption, every node of the flow graph is associated with complete *in* and *out* sets, where the stored information is often unrelated to the particular graph node and only needed to propagate the data flow information to all relevant program points.

1.3 Data Flow Propagation under SSA Form

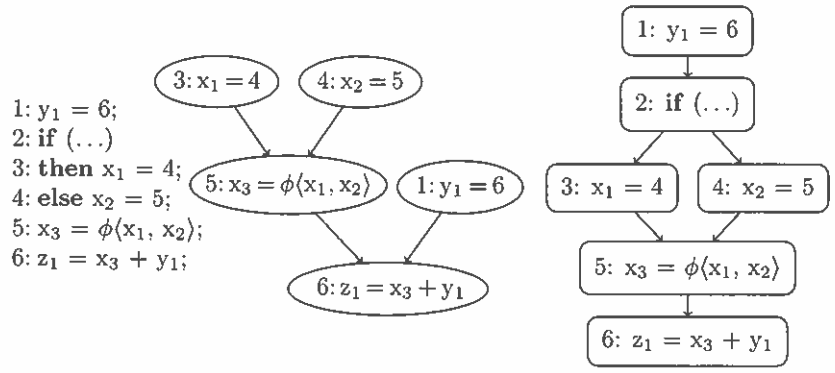
SSA form allows to solve a large class of data flow problems more efficiently than the standard fixed point solution presented previously. The basic idea is to directly propagate information computed at the unique definition of a variable to all its uses. Intermediate program points on regular control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

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. In the following, we assume that possibly many ϕ-operations are associated with the corresponding CFG nodes at those join points.

Data flow analysis under SSA form relies on a specialized program representation based on *SSA graphs*, which resemble traditional *def-use chains* and simplify the propagation of data flow information. The nodes of an SSA graph correspond to the operations of the program, including the program's ϕ-operations that are represented by dedicated nodes in the graph. The edges of the graph connect the unique definition of a variable with all its uses, i.e., edges represent true dependencies.

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



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

Fig. 1.1 Example program and its SSA graph.

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 variable defined by the respective ϕ-operation.

Consider for example, the code excerpt shown in Figure 1.1, 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, down to its unique use at the end. The traditional CFG representation causes the propagation to pass though several intermediate program points. These program points are concerned with computations of the variables x_1, x_2, and x_3 only and are thus irrelevant for the computation. The SSA graph representation, on the other hand, allows to directly propagate the desired information, 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 1.1. 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 shortcomings of the SSA graph this information cannot be derived. It is thus important to use both, the control flow graph and the SSA graph, during data flow analysis in order to obtain the best possible results.

1. Initially, every edge in the CFG is marked not executable and the *CFGWorkList* is seeded with the outgoing edges of the control 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 control 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 *CFGWorkList* visit its operation.
   - When the target node has a single outgoing non-executable edge append that edge to the *CFGWorkList*.
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 control 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

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 using:

- a *complete lattice* representing the property space,
- a set of *transfer functions* for the operations of the program,
- a *control flow graph* capturing the program's execution flow, and
- an *SSA graph* representing data dependencies.

We again seek a maximal (minimal) 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 control flow graph but along the edges of the SSA graph. For regular uses the propagation is then straightforward due to the fact that every use receives its value from a unique definition. Special care has to be taken only for ϕ-operations, which select a value among their operands depending on the incoming control 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 SSA graph node. The *in* and *out* sets used by the traditional approach – see Section 1.2 – are obsolete, since ϕ-operations already provide the required buffering. In addition, the control flow graph is used 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.

The algorithm processes two work lists, the *CFGWorkList*, containing edges of the control flow graph, and the *SSAWorkList*, which consists of edges from the SSA graph. It proceeds by removing the top element of either of those lists and pro-

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 control flow edge is marked executable.
   b. In the case of conditional branches, update the operation's data flow information. Determine which of the outgoing control flow edges are reachable from the corresponding control flow graph node by examining the branch's condition(s) and append the respective non-executable edges to the *CFGWorkList*.
   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

cessing the respective edge as indicated by Algorithm 1. Throughout the algorithm operations of the program are visited to update the work lists and propagate information as shown by Algorithm 2. We will highlight the most relevant steps of the algorithms in more detail in the following paragraphs.

In step 3 of the main algorithm, control flow edges are processed that were encountered to be executable for the first time in the course of the analysis. Whenever such a control 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 control 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 control flow 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 through the SSA graph.

Regular operations as well as ϕ-operations are visited by Algorithm 2 when the corresponding control flow graph node has become executable or whenever the data flow information of one of their predecessors in the SSA graph changed. At ϕ-operations the information from multiple control flow paths is combined using the usual meet or join operator. However, only those operands where the associated control flow edge is marked executable are considered. 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, control flow edges are appended to the *CFGWorkList* 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 1.1 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 control flow edges in the program will eventually be marked executable. This will trigger the evaluation of the constant assignments to the variables x_1, x_2,

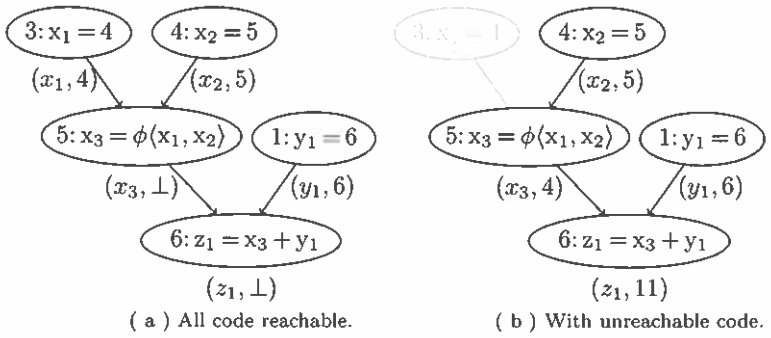


Fig. 1.2 Sparse conditional data flow propagation using SSA graphs.

and y_1. The transfer functions immediately yield that the variables are all constant, holding the values 4, 5, and 6 respectively. This new information will trigger the reevaluation of the ϕ-operation of variable x_3. As both of its incoming control flow edges are marked executable, the combined information yields $4 \sqcap 5 = \bot$, i.e., the value is known not to be a particular constant value. 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 1.2a. 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 1.2b. Neither the control 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 enables the analysis to show that the assignment to variable z_1 is, in fact, constant as well.

1.3.3 Discussion

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 control flow graph, on the other hand, are processed at most once. This leads to an upper bound in execution time of $O(|E_{SSA}| \cdot h + |E_{CFG}|)$, where E_{SSA} and E_{CFG} represent the edges of the SSA graph and the control flow graph respectively. The size of the SSA graph increases with respect to the original non-SSA program. Measurements indicate that this growth is linear, 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 storing the *in* and *out* sets capturing the complete property space on every program point, it is sufficient to associate every node in the SSA graph with the data flow information of the corresponding variable only. This leads to considerable savings in practice.

1.3.4 Limitations

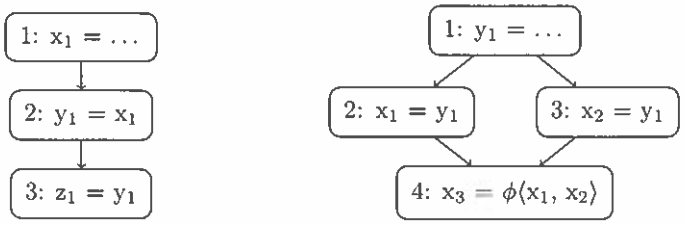
Unfortunately, the presented approach also has its limitations, which arise from two sources: (1) the exclusive propagation of information between data-dependent operations and (2) the semantics and placement of ϕ operations. The former issue prohibits the modeling of data flow problems that propagate 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 general backward problems.

Consider, for example, the well known problem of available expressions 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. The SSA graph does not cover those points, as it propagates information directly from definitions to uses without any intermediate steps.

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 control flow graph. SSA graphs are consequently not suited to model backward problems in general.

1.4 Example – Copy Propagation

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 example. Copy propagation under 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



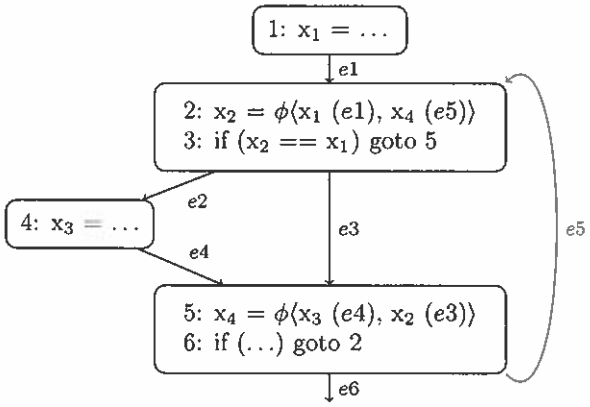
(a) Transitive copy relations. (b) Copy relations on ϕ-operations.

Fig. 1.3 Analysis of copy-related variables.

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 1.3a, 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 used previously for constant propagation. 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 of Figure 1.3b. This is accomplished by choosing the join operator such that a copy



move to next page

Fig. 1.4 ϕ-operations in loops often obfuscate copy relations.

relation is propagated whenever the copy-of values of all the ϕ-operation's operands match. 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 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 1.4. 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:

1. Assuming that the value assigned to variable x_1 is not a copy, the data flow information for this variable is lowered to $\bot$, the SSA edges leading to operations 2 and 3 are appended to the *SSAWorkList*, and the control flow graph edge $e1$ is appended to the *CFGWorkList*.
2. Processing the control 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 control flow graph edges are executable for the conditional branch. Examining the condition shows that only edge $e3$ is ~~reachable~~ executable and thus needs to be added to the work list.
3. Control 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 yields 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 control flow edges $e5$ and $e6$ to be added to the work list.
4. Now, control 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 control flow edge $e6$ leads to the exit node of the control 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 handle 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 control flow graph once the set of executable operations has been discovered.

1.5 Further Reading

Traditional data flow analysis is well established and well described in numerous papers. The book by Nielsen, Nielsen, and Hankin [9] gives an excellent introduction to the theoretical foundations and practical applications. For reducible flow graphs the order in which operations are processed by the work list algorithm can be optimized [6, 8, 9], allowing to derive tighter complexity bounds. However, relying on reducibility is problematic, because the flow graphs are often *not* reducible even for proper structured languages, e.g., reversed control flow graphs for backward problems can be, and in fact almost always are, irreducible even for programs with reducible control flow graphs. Furthermore, experiments have shown that the tighter bounds not necessarily lead to improved compilation times [3].

Apart from computing a fixed point (MFP) solution, traditional data flow equations can also be solved using a more powerful approach called the *meet over all paths* (MOP) solution, which computes the *in* data flow information for a basic block by examining *all* possible paths from the start node of the control flow graph. Even though more powerful, computing the MOP solution is often harder to compute or even undecidable [9]. Consequently, the MFP solution is preferred in practice.

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* form (SSI) [14] 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 [13]. Bodík uses an extended SSA form, *e*-SSA, to eliminate array bounds checks [1]. Ruf [12] introduces the *value dependence graph*, which captures both control and data dependencies. He derives a sparse representation of the input program, which is suited for data flow analysis, using a set of transformations and simplifications.

The *sparse evaluation graph* by Choi et.al [2] 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 [7], is based on single-entry/single-exit regions. The resulting graph is usually less sparse, but is also less complex to compute. Ramalingam [11] 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 [5] instead examine the data flow equations, eliminate redundancies, and apply simplifications to them.

References

1. Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the Conference on Programming Language Design and Implementation*, pages 321–333, New York, NY, USA, 2000. ACM.
2. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '91: Proceedings of the Symposium on Principles of Programming Languages*, pages 55–66, New York, NY, USA, 1991. ACM.
3. Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. An empirical study of iterative data-flow analysis. In *CIC '06: Proceedings of the Conference on Computing*, pages 266–276, Washington, DC, USA, 2006. IEEE Computer Society.
4. 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.
5. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *CC '94: Proceedings of the Conference on Compiler Construction*, pages 357–373, London, UK, 1994. Springer-Verlag.
6. Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *POPL '73: Proceedings of the Symposium on Principles of Programming Languages*, pages 207–217, New York, NY, USA, 1973. ACM.
7. Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI '93: Proceedings of the Conference on Programming Language Design and Implementation*, pages 78–89, New York, NY, USA, 1993. ACM.
8. John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.
9. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
10. Diego Novillo. A propagation engine for GCC. In *Proceedings of the GCC Developers Summit*, pages 175–184, 2005.
11. Ganesan Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.
12. Erik Ruf. Optimizing sparse representations for dataflow analysis. In *IR '95: Proceedings of the Workshop on Intermediate Representations*, pages 50–61, New York, NY, USA, 1995. ACM.
13. Jeremy Singer. Sparse bidirectional data flow analysis as a basis for type inference. In *APPSEM '04: Web Proceedings of the Applied Semantics Workshop*, 2004.
14. Jeremy Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, UK, 2005.
15. M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.