CHAPTER 1

Graphs and gating functions

J. Stanier

It is not really to "aid" but rather to "support" the development of analysis algorithms.

Progress: 90%

Text and figures formating in progress

This whole introduction is way to long and verbose. It can be drastically reduced...

1.1 Introduction

Many compilers represent the input program as some form of graph in order to aid analysis and transformation. A cornucopia of program graphs have been presented in the literature and implemented in real compilers. Therefore it comes as no surprise that a number of program graphs use SSA concepts as the core principle of their representation. These range from very literal translations of SSA into graph form to more abstract graphs which are implicitly SSA. We ~~aim to~~ introduce a selection of program graphs which use SSA concepts, and examine how they may be useful to a compiler writer.

"surprise"? well, this is not really the case, SSA was developed to address a very specific need - support in an direct way, def-use chains and webs in a program to more efficiently support analysis and optimizations that have to reason about the live times of the values.

precisely the point I was trying to make with the comment on the right...

One of the seminal graph representations is the Control Flow Graph (CFG), which was introduced by Allen to explicitly represent possible control paths in a program. Traditionally, the CFG is used to convert a program into SSA form. Additionally, representing a program in this way makes a number of operations simpler to perform, such as identifying loops, discovering irreducibility and performing interval analysis techniques.

Again I would say supporting.

The CFG models control flow, but many graphs model *data flow*. This is useful as a large number of compiler optimizations are based on data flow. The graphs we consider in this chapter are all data flow graphs, representing the data dependencies in a program. We will look at a number of different SSA-based graph representations. These range from those which are a very literal translation of SSA into a graph form to those which are more abstract in nature. An introduction to each graph will be given, along with diagrams to show how sample programs look when translated strange construct.

into that particular graph. Additionally, we will touch on the literature describing a usage of a given graph with the application that it was used for.

1.2 Data Flow Graphs

The data flow graph (DFG) is a directed graph $G = (V, E)$ where the edges E represent the flow of data from the result of one operation to the input of another. An instruction executes once all of its input data values have been consumed. When an instruction executes it produces a new data value which is propagated to other connected instructions.

This statement of target code not being directly generated is somewhat misleading. I would rephrase this.

Whereas the CFG imposes a total ordering on instructions, the DFG has no such concept, nor does the DFG contain whole program information. Thus, target code cannot be generated directly from the DFG. The DFG can be seen as a companion to the CFG, and they can be generated alongside each other. With access to both graphs, optimisations such as dead code elimination, constant folding and common subexpression elimination can be performed effectively. However, keeping both graphs updated during optimisation can be costly and complicated.

1.3 The SSA Graph

This paragraph goes too fast. In fact I would suggest introducing first the notion of SSA and then the graph itself. Then the author goes on about reversing the edges. Please first give an example of a graph and the reversed so that we can see what you mean. In fact it might be much more natural introduce the Value Dependence Graph after the more traditional CFG with phi-nodes or phi-instructions.

We begin our exploration with a graph that is very similar to SSA: the SSA Graph. Notice that many different variations exist in the literature. For us, an SSA Graph consists of vertices which represent operations (such as `add` and `load`) or ϕ-functions, and directed edges connect uses to definitions of values. The outgoing edges from a vertex represent the arguments required for that operation, and the ingoing edge(s) to a vertex represents the propagation of that operation's result(s) after it has been computed. This graph is therefore a *demand-based* representation. In order to compute a vertex, we must first *demand* the results of the operands and then perform the operation indicated on that vertex. Reversing the direction of each edge would provide the widely used *data-based* representation. The SSA Graph can be constructed from a program in SSA form by *explicitly* adding use-definition chains. We present some sample code in Figure 1.1 which is then translated into an SSA Graph. Note that each node mixes up the operation and the variable(s) it defines, as actual data structures might be able to find one from the other.

The textual representation of SSA is much easier for a human to read. However, the primary benefit of representing the input program in this form is that the compiler writer is able to apply a wide array of graph-based optimizations by using standard graph traversal and transformation techniques. It is possible to augment the SSA Graph to model memory dependencies. This can be achieved by adding *state edges* that enforce an order of interpretation. These edges are extensively used in

```
begin: a0 = 0;
       i0 = 0;
loop:  a1 = φ(a0,a2);
       i1 = φ(i0,i2);
       if i1 > 100 goto end;
       a2 = a1 * i1;
       i2 = i1 + 1;
       if a2 > 20 goto end;
       goto loop;
end:   a3 = φ(a1,a2);
       i3 = φ(i1,i2);
       print(a3 + i3);
```

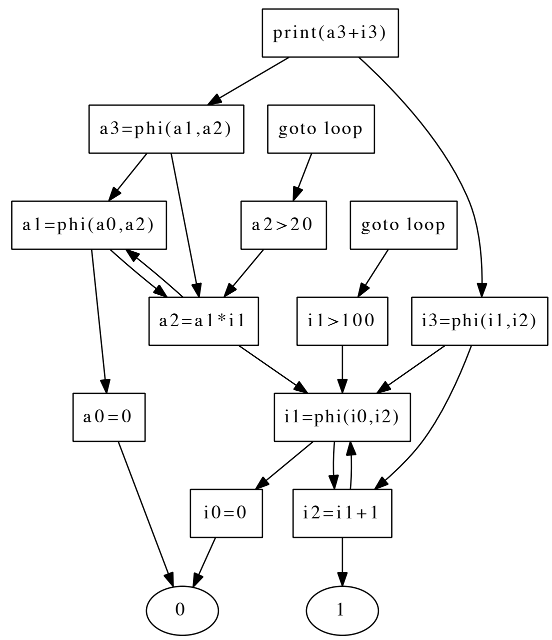


Fig. 1.1 Some SSA code translated into a (demand-based) SSA graph.

the Value State Dependence Graph, which we will look at later, after we touch on the concept of gating functions.

In the literature, the SSA Graph has been used to detect a variety of induction variables in loops, also for performing instruction selection techniques, operator strength reduction, rematerialization, and has been combined with an extended SSA language to aid compilation in a parallelizing compiler. The reader should note that the exact specification of what constitutes an SSA Graph changes from paper to paper. The essence of the intermediate representation (IR) has been presented here, as each author tends to make small modifications for their particular implementation.

1.3.1 Finding induction variables with the SSA Graph

We illustrate the usefulness of the SSA Graph through a basic induction variable (IV) recognition technique. A more sophisticated technique is developed in Chapter **??**. Given that a program is represented as an SSA Graph, the task of finding induction variables is simplified. A *basic linear induction variable* i is a variable that appears only in the form:

```
i = 10
loop
     ...
     i = i + k
     ...
endloop
```

where k is a constant or loop invariant. A simple IV recognition algorithm is based on the observation that each basic linear induction variable will belong to a non-trivial strongly connected component (SCC) in the SSA graph. SCCs can be easily discovered in linear time using any depth first search traversal. Each such SCC must conform to the following constraints:

This is a good example.

- The SCC contains only one ϕ-function at the header of the loop.
- The SCC contains only addition and subtraction operators, and the right operand of the subtraction is not part of the SCC (no `i = n − i` assignments).
- The other operand of each addition or subtraction is loop invariant.

This technique can be expanded to detect a variety of other classes of induction variables, such as wrap-around variables, non-linear induction variables and nested induction variables. Scans and reductions also show a similar SSA Graph pattern and can be detected using the same approach.

1.4 Program Dependence Graph

The Program Dependence Graph (PDG) represents both control and data dependencies together in one graph. The PDG was developed to aid optimizations requiring reordering of instructions and graph rewriting for parallelism, as the strict ordering of the CFG is relaxed and complemented by the presence of data dependence information. The PDG is a directed graph $G = (V, E)$ where nodes V are statements, predicate expressions or region nodes, and edges E represent either control or data dependencies. Thus, the set of all edges E has two distinct subsets: the control dependence subgraph E_C and the data dependence subgraph E_D. Similar to the CFG, a PDG also has two nodes `ENTRY` and `EXIT`, through which control flow enters and exits the program respectively. For this purpose, it is assumed that every node of the CFG is reachable from the entry node and can reach the exit node.

I would insert a paragraph here.

Statement nodes represent instructions in the program. Predicate nodes test a conditional statement and have true and false edges to represent the choice taken on evaluation of the predicate. Region nodes group control dependencies with identical source and label together. If the control dependence for a region node is satisfied, then it follows that all of its children can be executed. Thus, if a region node has three different control-independent statements as immediate children, then these could potentially be executed in parallel. Diagrammatically, rectangular nodes represent statements, diamond nodes predicates, and circular nodes are region nodes. Dashed edges represent control dependence, and solid edges represent data dependence. Loops in the PDG are represented by back edges in the control dependence subgraph. We show an example code translated into a PDG in Figure 1.2.

"falls itself" is awkward and I cannot really understand what it means. Why not "is accomplished in two steps"?

obviously missing.

This section is way too long. You can easily shorten it bey possibly avoiding it altogether and refering to the original paper or summarizing the core idea.

It i really unbalance when comparing to other graphs described in this chapter.

Construction of the PDG is tackled in two steps from the CFG: construction of the control dependence subgraph and construction of the data dependence subgraph. The construction of the control dependence subgraph falls itself into two steps. First, control dependence between statements and predicate nodes are computed; then region nodes are added. A node w is said to be *control dependent on node u along CFG edge* (u, v) or simply *control dependent on edge* (u, v) if w post-dominates v and w does not *strictly* post-dominate u. Control dependence between nodes is nothing else than post-dominance frontier, i.e., dominance frontier on the reverse CFG. A slight difference in its construction process is that the corresponding control dependence edges from u to w is labelled by the boolean value taken by the predicate computed in u when branching on edge (u, v). To compute the control dependence subgraph, the postdominator tree is constructed for the procedure. Then, the `ENTRY` node is added with one edge labelled true pointing to the CFG entry node, and another labelled false going to the CFG exit node. Then, let S consist of all edges (A, B) in the CFG such that B is not an ancestor of A in the postdominator tree. Each of these edges has an associated label true or false. Then, each edge in S is considered in turn. Given (A, B), the postdominator tree is traversed backwards from B until we reach A's parent, marking all nodes visited (including B) as control dependent on A with the label of S.

repeated sentence structure. Reads poorly.

Next, region nodes are added to the PDG. Each region node summarizes a set of control conditions and "groups" all nodes with the same set of control conditions together. Region nodes are also inserted so that predicate nodes will only have two successors. To begin with, an unpruned PDG is created by checking, for each node of the CFG, which control region it depends on. This is done by traversing the postdominator tree in postorder, and using a hash table to map sets of control dependencies to region nodes. For each node N visited in the postdominator tree, the hash table is checked for an existing region node with the same set CD of control dependencies. If none exists, a new region node R is created with these control

The use of an hash-table is really an implementation details and could easily be omitted.

Where is the PDG for this example?

```
begin: i = 1;
loop:  if i > 100 goto end;
       a = 2 * B[i];
       A[i] = a;
       i = i + 1;
       if a > 20 goto end;
       goto loop;
end:   return a;
```

Fig. 1.2 Some code translated into a PDG. Nodes associated with the evolution of the induction variable i are omitted.

dependencies and entered into the hash table. R is made to be the only control dependence predecessor of N. Next, the intersection INT of CD is computed for each immediate child of N in the postdominator tree. If $INT = CD$ then the corresponding dependencies are deleted from the child and replaced with a single dependence on the child's control predecessor. Then, a pass over the graph is made to make sure that each predicate node has a unique successor for each truth value. If more than one exists, the corresponding edges are replaced by a single edge to a freshly created region node that itself points to the successor nodes.

Again, this description is too detailed I think and borrowed straight from the original paper about PDG construction.

A statement B that is to be executed after a statement A in the original sequential ordering of the program depends on A in the following situations: (flow) B reads to a storage location that was lastly accessed by A through a write; (anti) B writes to a storage location previously accessed through a read by A; (output) A and B both have a write access to the same storage location. Side effects can also dictate the insertion of a dependence between A and B to force the sequential ordering of the final schedule. Memory accesses can not always be analyzed with enough precision. In the presence of a may-alias between two consecutive accesses, a conservative dependence is to be inserted also. Memory access locations often vary with the iterations of the enclosing loops. Dependence analysis can take advantage of some abstract representations of the access function such as when the memory address can be represented as an affine function of the induction variables. Not only does this enables a refinement of the alias information, but also the dependence can be labelled with a distance vector as a function itself of the loop indices. As an example, a loop indexed by i, that would access array $A[i]$ twice, first as a write at iteration i, then as a read at iteration $2i + 1$, would lead to a flow dependence of distance $i + 1$.

This is a side discussion about the limitations of data dependence analysis and the example at the end is rather obvious. I woul dsuggest trimming down this section and just refering to the conservative approach compilers much take when dealing with data dependence.

The PDG can also be constructed during parsing. Sequential code can be derived from a PDG, but generating the *minimal size* CFG from a PDG turns out to be an NP-Complete problem. The PDG's structure has been exploited for generating code for vectorisation, and has also been used in order to perform accurate program slicing and testing.

This last paragraph comes out of nowehere and is somewhat inconsequent.

1.4.1 *Detecting parallelism with the PDG*

Not sure the general audience for this book is familiar with an hyper-block.

The structure of the PDG allows for parallelism to be detected easily. On a regular CFG representation, a scheduler based on data dependencies will generally restrict its scope to hyper-blocks. In this context, code transformations such as loop unrolling or if-conversion (see Chapter **??**) that effectively change control dependencies into data dependencies can expose instruction level parallelism. However, the PDG can *directly* be used to detect parallelism. As an example, any node of a CFG loop, that is not contained in an SCC of the PDG (considering *both* control and data dependence edges) can be vectorized. In the example in Figure 1.2, since the statement `A[i]=a` in the loop do not form an SCC in the PDG, it can be vectorized provided array expansion of variable a. On the other hand, because of the circuit involving the test on a, the statement `a=2*B[i]` cannot.

It would be better instead to have a discussion of what it means to be in a SCC and how that captures the opportunities for concurrency.

Where is the PDG for this example?

1.5 Gating functions and GSA

In SSA form, ϕ-functions are used to identify points where variable definitions converge. However, they cannot be directly *interpreted*, as they do not specify the condition which determines which of the variable definitions to choose. By this logic, we cannot directly interpret the SSA Graph. Being able to interpret our IR is a useful property as it gives the compiler writer more information when implementing optimizations, and also reduces the complexity of performing code generation. Gated Single Assignment form (GSA; sometimes called Gated SSA) is an extension of

Good - short and to the point.

SSA with *gating functions*. These gating functions are directly interpretable versions of ϕ-nodes, and replace ϕ-nodes in the representation. We usually distinguish the three following forms of gating functions:

- The ϕ_{if} function explicitly represents the condition which determines which ϕ value to select. A ϕ_{if} function of the form $\phi_{if}(P, V_1, V_2)$ has P as a predicate, and V_1 and V_2 as the values to be selected if the predicate evaluates to true or false respectively. This can be read simply as *if-then-else*.
- The ϕ_{entry} function is inserted at loop headers to select the initial and loop carried values. A ϕ_{entry} function of the form $\phi_{entry}(V_{init}, V_{iter})$, has V_{init} as the initial input value for the loop, and V_{iter} as the iterative input. We replace ϕ-functions at loop headers with ϕ_{entry} functions.
- The ϕ_{exit} function determines the value of a variable when a loop terminates. A ϕ_{exit} function of the form $\phi_{exit}(P, V_{final})$ has P as predicate and V_{final} as the definition reaching beyond the loop.

please upgrade the figures with the phi symbol rather then the "phi" representation to keep the text consistent with the figure.

```
begin: i = 1;
loop:  if i > 100 goto end;
       a = 2 * B[i];
       A[i] = a;
       i = i + 1;
       if a > 20 goto end;
       goto loop;
end:   return a;
```

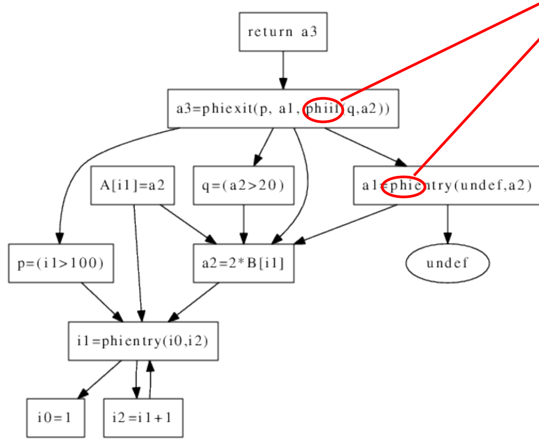


Fig. 1.3 A graph representation of our sample code in GSA form.

It is easiest to understand these gating functions by means of an example. Figure 1.3 shows how our earlier code in Figure 1.2 translates into GSA form. Here, we can see the use of both ϕ_{entry} and ϕ_{exit} gating functions. At the header of our sample loop, the ϕ-function has been replaced by a ϕ_{entry} function which determine between the initial and iterative value of i. After the loop has finished executing, the nested ϕ_{exit} functions selects the correct live-out version of a.

This example shows several interesting points. First, the semantic of both the ϕ_{exit} and ϕ_{if} are strict in their gate: here a_1 or $\phi_{exit}(q, a_2)$ are not evaluated before p is known [1]. Similarly, a ϕ_{if} function that results from the nested if-then-else code

[1] As opposed to the ψ-function described in Chapter **??** that would use a syntax such as $a_3 = \phi((p \wedge \neg q)?a_1, (\neg p \wedge q)?a_2)$ instead.

of Figure 1.4 would be itself nested as $a = \phi_{if}(p, \phi_{if}(q, a_2, a_3), a_1)$. Second, this representation of the program does not allow for an interpreter to decide whether an instruction with a side effect (such as $A[i_1] = a_2$ in our running example) has to be executed or not. Finally, computing the values of gates is highly related to the simplification of path expressions: in our running example a_2 should be selected when the path $\neg p$ followed by q (denoted $\neg p.q$) is taken while a_1 should be selected when the path p is taken; for our nested if-then-else example, a_1 should be selected either when the path $\neg p.r$ is taken or when the path $\neg p.\neg r$ is taken which simplifies to $\neg p$. Diverse approaches can be used to generate the correct nested ϕ_{if} or ϕ_{exit} gating functions.

I'm lost about what you mean by "that" here. A little bit of redundancy in the text may be good here.

```
a1 = ...
if (p) then
   if (q) then (a2 = ...)
   else (a3 = ...)
else
   if (r) then (...)
   else (...)
```

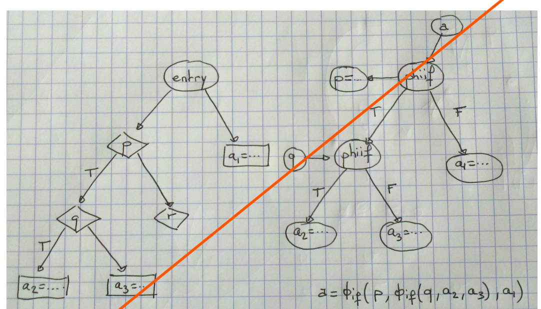


Fig. 1.4 (a) A structured code; (b) the CDG (with region nodes ommited); (c) the DAG representation of the nested gated ϕ_{if}

The more natural way uses a data flow analysis that computes, for each program points and each variable, its unique reaching definition and the associated set of reaching paths. This set of paths is abstracted using a *path expression*. If the code is not already under SSA, and if at a merge point of the CFG its predecessor basic blocks are reached by different variables, a ϕ-function is inserted. The gates of each operand is set to the path expression of its corresponding incoming edge. If a unique variable reaches all the predecessor basic blocks, the corresponding path expressions are merged. Of course, a classical path compression technique can be used to minimize the number of visited edges. One can observe the similarities with the ϕ-function placement algorithm described in Section **??**.

There also exists a relationship between the control dependencies and the gates: from a code already under strict and conventional SSA form, one can derive the gates of a ϕ_{if} function from the control dependencies of its operands. This relationship is illustrated by Figure 1.4 in the simple case of a structured code.

These gating functions are important as the concept will form components of the Value State Dependence Graph later. GSA has seen a number of uses in the literature including analysis and transformations based on data flow. With the diversity of applications (see chapters **??** and **??**), many variants of GSA have been proposed.

Those variations concern the correct handling of loops in addition to the computation and representation of gates.

By using gating functions it becomes possible to construct IRs based solely on data dependencies. These IRs are sparse in nature compared to the CFG, making them good for analysis and transformation. This is also a more attractive proposition than generating and maintaining both a CFG and DFG, which can be complex and prone to human error. One approach has been to combine both of these into one representation, as is done in the PDG. Alternatively, we can utilize gating functions along with a data flow graph for an effective way of representing whole program information using data flow information.

1.5.1 Backwards symbolic analysis with GSA

Overall I liked this subsection but the example is not that compelling in the sense that it should include longer expression so that we can see a big benefit in using the backwards symbolic analysis.

GSA is useful for performing symbolic analysis. Traditionally, symbolic analysis is performed by forward propagation of expressions through a program. However, complete forward substitution is expensive and can result in a large quantity of unused information and complicated expressions. Instead, *backward*, demand-driven substitution can be performed using GSA which only substitutes *needed* information. Consider the following program:

```
R: JMAX = Expr
S: if(P) then J = JMAX - 1
         else J = JMAX
T: assert(J ≤ JMAX)
```

Fig. 1.5 A program on which to perform symbolic analysis.

If forwards substitution were to be used in order to determine whether the assertion is correct, then the symbolic value of J must be discovered, starting at the top of the program in statement R. Forward propagation through this program results in statement T being $assert((if\ \mathtt{P}\ then\ \mathtt{Expr}-1\ else\ \mathtt{Expr}) \leq \mathtt{Expr})$, thus the *assert* statement evaluates to true. In real, non-trivial programs, these expressions can get unnecessarily long and complicated.

repeated thought

Using GSA instead allows for backwards, demand-driven substitution. The program above has the following GSA form:

Using this backwards substitution technique, we start at statement T, and follow the SSA links of the variables from $\mathtt{J}_3$. This allows for skipping of any intermediate statements that do not affect variables in T. Thus the substitution steps are:

The backwards substitution then stops because enough information has been found, avoiding the redundant substitution of $\mathtt{JMAX}_1$ by Expr. In non-trivial programs this can greatly reduce the number of redundant substitutions, making symbolic analysis significantly cheaper.

```
R: JMAX₁ = Expr
S: if(P) then J₁ = JMAX₁ - 1
         else J₂ = JMAX₁
   J₃ = φif(P,J₁,J₂)
T: assert(J₃ ≤ JMAX₁)
```

Fig. 1.6 Figure 1.5 in GSA form.

$$\begin{aligned} \mathtt{J}_3 &= \phi_{if}(\mathtt{P},\mathtt{J}_1,\mathtt{J}_2) \\ &= \phi_{if}(\mathtt{P},\mathtt{JMAX}_1 - 1,\mathtt{JMAX}_1) \end{aligned}$$

Fig. 1.7 Substitution steps in backwards symbolic analysis.

1.6 Value State Dependence Graph

The gating functions defined in the previous section were used in the development of a sparse data flow graph IR called the Value State Dependence Graph (VSDG). The VSDG is a directed graph consisting of operation nodes, loop and merge nodes together with value and state dependency edges. Cycles are permitted but must satisfy various restrictions. A VSDG represents a single procedure: this matches the classical CFG.

An example VSDG is shown in Figure 1.8. In (a) we have the original C source for a recursive factorial function. The corresponding VSDG (b) shows both value and state edges and a selection of nodes.

1.6.1 Definition of the VSDG

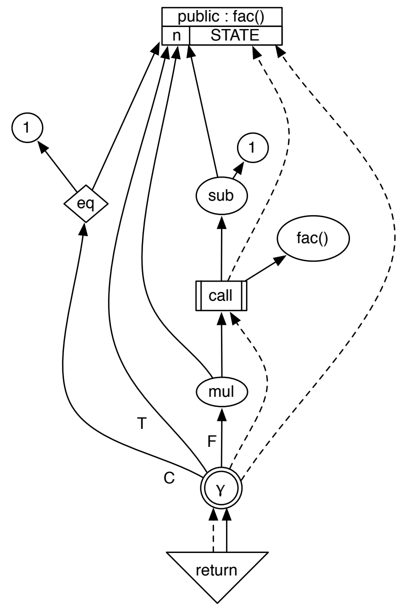
A VSDG is a labelled directed graph $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ consisting of nodes N (with unique entry node N_0 and exit node N_∞), value dependency edges $E_V \subseteq N \times N$, state dependency edges $E_S \subseteq N \times N$. The labelling function ℓ associates each node with an operator.

The VSDG corresponds to a reducible program, e.g. there are no cycles in the VSDG except those mediated by θ (loop) nodes.

Value dependency (E_V) indicates the flow of values between nodes. State dependency (E_S) represents two things; the first is essential sequential dependency required by the original program, e.g. a given `load` instruction may be required to follow a given `store` instruction without being re-ordered, and a `return` node in general must wait for an earlier loop to terminate even though there might be no value-dependency between the loop and the `return` node. The second purpose is that state dependency edges can be added incrementally until the VSDG corresponds to a unique CFG. Such state dependency edges are called *serializing* edges.

```
int fac(int n) {
    int result;
    if(n == 1)
       result = n;
    else
       result = n * fac(n - 1);
    return result;
}
```

(a)



(b)

Fig. 1.8 A recursive factorial function, whose VSDG illustrates the key graph components—value dependency edges (solid lines), state dependency edges (dashed lines), a `const` node, a `call` node, a γ-node, a conditional node, and the function entry and exit nodes.

The VSDG is implicitly represented in SSA form: a given operator node, n, will have zero or more E_V-consumers using its value. Note that, in implementation terms, a single register can hold the produced value for consumption at all consumers; it is therefore useful to talk about the idea of an output *port* for n being allocated a specific register, r, to abbreviate the idea of r being used for each edge (n_1, n_2) where $n_2 \in succ(n_1)$.

1.6.2 Nodes

There are four main classes of VSDG nodes: value nodes (representing pure arithmetic), γ-nodes (conditionals), θ-nodes (loops), and state nodes (side-effects). The majority of nodes in a VSDG generate a value based on some computation (add, subtract, etc) applied to their dependent values (constant nodes, which have no dependent nodes, are a special case).

1.6.3 γ-Nodes

The γ-node is similar to the ϕ_{if} gating function in being dependent on a control predicate, rather than the control-independent nature of SSA ϕ-functions.

A γ-node $\gamma(C,T,F)$ evaluates the condition dependency C, and returns the value of T if C is true, otherwise F.

We generally treat γ-nodes as single-valued nodes (contrast θ-nodes, which are treated as tuples), with the effect that two separate γ-nodes with the same condition can be later combined into a tuple using a single test. Figure 1.9 illustrates two γ-nodes that can be combined in this way. Here, we use a pair of values (2-tuple) of values for the T and F ports. We also see how two syntactically different programs can map to the same structure in the VSDG.

```
a) if (P)
x = 2, y = 3;
else
x = 4, y = 5;

b) if (P) x = 2; else x = 4;
...
if (P) y = 3; else y = 5;
```

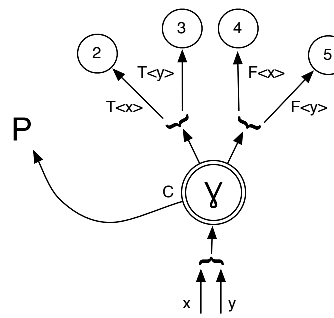


Fig. 1.9 Two different code schemes (a) & (b) map to the same γ-node structure.

Why are you opting for the nomenclature of ports here? This sounds like a Petri-net legacy. I would suggests maintaining a tuple representation by naming these as elements of the tuple.

1.6.4 θ-Nodes

The θ-node models the iterative behaviour of loops, modelling loop state with the notion of an *internal value* which may be updated on each iteration of the loop. It has five specific ports which represent dependencies at various stages of computation.

A θ-node $\theta(C,I,R,L,X)$ sets its internal value to initial value I then, while condition value C holds true, sets L to the current internal value and updates the internal value with the repeat value R. When C evaluates to false computation ceases and the last internal value is returned through the X port.

this is a fancy word for termination...

A loop which updates k variables will have: a single condition port C, initial-value ports $I_1,\ldots,I_k$, loop iteration ports $L_1,\ldots,L_k$, loop return ports $R_1,\ldots,R_k$, and loop exit ports $X_1,\ldots,X_k$. The example in Figure 1.10 also shows a pair (2-tuple) of values being used for I,R,L,X, one for each loop-variant value.

The θ-node directly implements pre-test loops (`while`, `for`); post-test loops (`do...while`, `repeat...until`) are synthesised from a pre-test loop preceded by

```
j = ...
for(i = 0;
i < 10;
++i)
--j;

... = j;
```

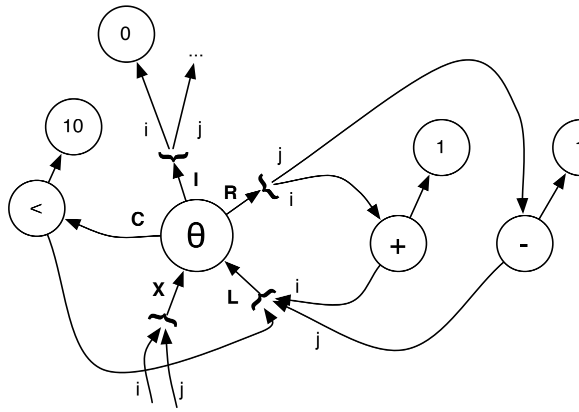


Fig. 1.10 An example showing a `for` loop. Evaluating the **X** port triggers it to evaluate the **I** value (outputting the value on the **L** port). While **C** evaluates to true, it evaluates the **R** value (which in this case also uses the θ-node's **L** value). When **C** is false, it returns the final internal value through the **X** port. As `i` is not used after the loop there is is no dependency on the `i` port of **X**.

a duplicate of the loop body. At first this may seem to cause unnecessary duplication of code, but it has two important benefits: (*i*) it exposes the first loop body iteration to optimization in post-test loops (cf. loop-peeling), and (*ii*) it normalizes all loops to one loop structure, which both reduces the cost of optimization, and increases the likelihood of two schematically-dissimilar loops being isomorphic in the VSDG.

1.6.5 State Nodes

Loads and stores compute a value and state. The `call` node takes both the name of the function to call and a list of arguments, and returns a list of results; it is treated as a state node as the function body may read or update state.

We maintain the simplicity of the VSDG by imposing the restriction that *all* functions have *one* return node (the exit node N_∞), which returns at least one result (which will be a state value in the case of `void` functions). To ensure that function calls and definitions are able to be allocated registers easily, we suppose that the number of arguments to, and results from, a function is smaller than the number of physical registers—further arguments can be passed via a stack as usual.

Note also that the VSDG neither forces loop invariant code into nor out-of loop bodies, but rather allows later phases to determine, by adding serializing edges, such placement of loop invariant nodes for later phases.

1.6.6 Dead node elimination with the VSDG

By representing a program as a VSDG, many optimisations become trivial. For example, consider dead node elimination (Figure 1.11). This combines both dead code elimination and unreachable code elimination. Dead code generates VSDG nodes for which there is no value or state dependency path from the `return` node, i.e., the result of the function does not in any way depend on the results of the dead nodes. Unreachable code generates VSDG nodes that are either dead, or become dead after some other optimisation. Thus, a *dead node* is a node that is not postdominated by the exit node N_∞. To perform dead node elimination, only two passes are required over the VSDG resulting in linear runtime complexity: one pass to identify all of the live nodes, and a second pass to delete the unmarked (i.e., dead) nodes. It is safe because all nodes which are deleted are guaranteed never to be reachable from the `return` node.

Input: A VSDG $G(N, E_V, E_S, N_\infty)$ with zero or more dead nodes.
Output: A VSDG with no dead nodes.

```
Procedure DNE(G) {
    WalkAndMark(N∞, G);
    DeleteMarked(G);
}
Procedure WalkAndMark(n, G) {
    if n is marked then finish;
    mark n;
    foreach (node m ∈ N ∧ (n, m) ∈ (E_V ∪ E_S)) do
        WalkAndMark(m);
}
Procedure DeleteMarked(G) {
    foreach (node n ∈ N) do
        if n is unmarked then delete(n);
}
```

Everybody (the target audience for this book, of course) should understand the algorithm from the text given its simplicity. I see no reason to include this Fig. 1.11.

Fig. 1.11 Dead node elimination on the VSDG.

1.7 Further readings

A compiler's intermediate representation can be a graph, and many different graphs exist in the literature. We can represent the control flow of a program as a Control Flow Graph (CFG) [1], where straight-line instructions are contained within basic blocks and edges show where the flow of control may be transferred to once leaving that block. A CFG is traditionally used to convert a program to SSA form [7]. We can also represent programs as a type of Data Flow Graph (DFG) [9, 10], and SSA can be represented in this way as an SSA Graph [6]. An example was given that used the SSA Graph to detect a variety of induction variables in loops [31, 15]. It has also been used for performing instruction selection techniques [11, 22], operator strength reduction [6], rematerialization [5], and has been combined with an extended SSA language to aid compilation in a parallelizing compiler [25].

The Program Dependence Graph (PDG) as defined by Ferrante et al. [14] represents control and data dependencies in one graph. Their definition of control dependencies that turns out to be equivalent to post-dominance frontier leads to confusions at it uses a non standard definition of post-dominance. We choose to report the definition of Bilardi and Pingali [**?**]. Section **??** mentions possible abstractions to represent data dependencies for dynamically allocated objects. Among others, the book of Darte et al. [8] provides a good overview of such representations. The PDG has been used for program slicing [21], testing [3], and widely for parallelization [13, 12, 23, 4]. We showed an example of how the PDG directly exposes parallel code.

Gating functions can be used to create directly interpretable ϕ-functions. These are used in Gated Single Assignment Form. Alpern et al. [2] presented a precursor of GSA for structured code, to detect equality of variables. This chapter adopts their notations, i.e. a ϕ_{if} for a if-then-else construction, a ϕ_{entry} for the entry of a loop, and a ϕ_{exit} for its exit. The original usage of GSA was by Ballance et al. [20] as an intermediate stage in the construction of the Program Dependence Web IR. Further GSA papers replaced ϕ_{if} by γ, ϕ_{entry} by μ, and ϕ_{exit} by η. Havlak [16] presented an algorithm for construction of a simpler version of GSA—Thinned GSA—which is constructed from a CFG in SSA form. The construction technique sketched in this chapter is developed in more detail in [26]. GSA has been used for a number of analyses and transformations based on data flow. The example given of how to perform backwards demand-driven symbolic analysis using GSA has been borrowed from [27]. If conversion (see Chapter **??**), converts control dependencies into data dependencies. To avoid the potential loss of information related to the lowering of ϕ-functions into conditional moves or select instructions, gating ψ-functions (see Chapter **??**) can be used.

We then described the Value State Dependence Graph (VSDG) [18], which is an improvement on a previous, unmentioned graph, the Value Dependence Graph [30]. It uses the concept of gating functions, data dependencies and state to model a program. We gave an example of how to perform dead node elimination on the VSDG. Detailed semantics of the VSDG are available [18], as well as semantics of a related IR: the Gated Data Dependence Graph [29]. Further study has taken place on the problem of generating code from the VSDG [28, 19, 24], and it has also been used to perform a combined register allocation and code motion algorithm [17].

References

Please make the references more homogeneous - include the authors first initial only and abbreviate "Proceedings' as "Proc.", "Symposium" as "Symp", "Conference" as "Conf." and a few others. It can save you a little bit of space and looks way better..

1. Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.
2. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11. ACM, 1988.
3. Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, New York, NY, USA, 1993. ACM.
4. W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1989. ACM.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 311–321, New York, NY, USA, 1992. ACM.
6. Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001.
7. 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.
8. Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 1st edition, 2000.
9. J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.
10. J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.
11. Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA-graphs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on languages, compilers, and tools for embedded systems*, pages 31–40, New York, NY, USA, 2008. ACM.
12. J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 582–592, New York, NY, USA, 1988. ACM.
13. Jeanne Ferrante and Mary Mace. On linearizing parallel code. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1985. ACM.
14. 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.
15. Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995.

16. Paul Havlak. Construction of thinned gated single-assignment form. In *In Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 477–499. Springer Verlag, 1993.
17. Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In *In Proc. 12th International Conference on Compiler Construction (CC03) (April 2003*, pages 1–16, 2003.
18. Neil E. Johnson. Code size optimization for embedded processors. Technical Report UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, November 2004.
19. Alan C. Lawrence. Optimizing compilation with the Value State Dependence Graph. Technical Report UCAM-CL-TR-705, University of Cambridge, Computer Laboratory, December 2007.
20. 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.
21. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.
22. Stefan Schäfer and Bernhard Scholz. Optimal chain rule placement for instruction selection based on SSA graphs. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 91–100, New York, NY, USA, 2007. ACM.
23. B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In *SPAA '90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, pages 350–359, New York, NY, USA, 1990. ACM.
24. James Stanier. Ph.D. thesis. Technical report, University of Sussex, School of Informatics, October 2011.
25. Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *In Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 43–52, 1993.
26. Peng Tu and David Padua. Efficient building and placing of gating functions. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 47–55, New York, NY, USA, 1995. ACM.
27. Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 414–423, New York, NY, USA, 1995. ACM.
28. Eben Upton. Optimal sequentialization of gated data dependence graphs is NP-complete. In *PDPTA*, pages 1767–1770, 2003.
29. Eben Upton. Compiling with data dependence graphs, 2006.
30. Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–310, New York, NY, USA, 1994. ACM.
31. Michael Wolfe. Beyond induction variables. *SIGPLAN Not.*, 27(7):162–174, 1992.