Institute of Software Technology

Department of Programming Languages and Compilers

University of Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Technical Report No. 2009/03

Implementing Sparse Flow-Sensitive Andersen Analysis

Stefan Staiger-Stöhr

February 13, 2009

CR-Classification: D.3.4, F.2.2, F.3.2

Implementing Sparse Flow-Sensitive Andersen Analysis

Stefan Staiger-Stöhr

Institute of Software Technology, University of Stuttgart
staiger@informatik.uni-stuttgart.de

Technical Report TR 2009/03, Faculty of Computer Science
February 13, 2009

Abstract. Andersen's analysis is the most influential pointer analysis known so far. This paper, which contains parts of the author's upcoming PhD thesis, for the first time presents a *flow-sensitive* version of that analysis. We prove that the flow-sensitive version still has the same cubic complexity. Thus, the higher precision comes without loss of asymptotic scalability. This contradicts common wisdom of flow-sensitivity being substantially more expensive.
Compared to other flow-sensitive pointer analyses, we have no expensive data-flow problem on the CFG. Instead, we simply propagate pointer targets along data-flow relations which we determine during the analysis. Our analysis in fact combines the computation of the interprocedural SSA data-flow representation and the uncovering of pointer targets. It also integrates the computation of control-flow relations. The analysis thus presents a new, sparse approach for the flow-sensitive solution of the central problems for data-flow based program analyses.
This paper also presents two extensions for higher precision. The first extension shows how the analysis can detect strong updates without increasing the complexity. The second extension describes a context-sensitive version which excludes unrealizable paths. Together this yields the first analysis of that precision which only has a complexity of $\mathcal{O}(n^4)$. This is a substantial improvement over the previous $\mathcal{O}(n^6)$ bound found by Landi.
Thus, in summary this report describes several theoretical advances in the field of flow-sensitive pointer analysis. It also provides details on the algorithms used for incremental SSA construction and context-sensitive pointer propagation.

1 Introduction

Powerful data-flow analyses rely on control-flow and pointer information to be precise. Consequently, many published approaches discuss the problem of determining pointer targets. To solve this problem one can benefit from an already existing data-flow representation. Thus we have interdependencies between pointer analysis (PTA) and data-flow analysis. This paper presents a combined analysis which benefits from the interactions between these analyses.

Surprisingly, the basic concept behind our analysis is very simple and elegant: alternating steps of determining data-flow edges and propagating pointer targets along these edges solve the problem. Remarkably, each of these steps uses unsound results from the other analysis, but a standard fixpoint iteration over the combination of them yields sound results. This approach is very simple and in fact related to the most influential pointer analysis known so far: Andersen's analysis [And94] basically is the flow-insensitive (FI) counterpart of our analysis in that it replaces the data-flow relations with the constraint graph. Although our generalization of Andersen's analysis seems simple and straight-forward, neither Andersen nor other researchers so far, to the best of our knowledge, were able to find a flow-sensitive (FS) version in that way. Moreover, compared to other FS-PTAs, our FS-Andersen has the advantage that it operates on the data-flow graph instead of the control-flow graph. This removes many problems related to the classical CFG-based approach.

The *significance* of our work is that is lifts the most influential PTA to the higher FS precision without increasing the complexity. It also describes an asymptotically faster FS-MOVP solution than previously known. We thus expect more interest in flow-sensitivity, and affordable higher precision for many important applications based on points-to analyses (e.g., finding bugs).

The *contributions* of this paper are:

- We solve the long-standing problem of making the most influential PTA, namely Andersen's analysis, flow-sensitive.
- We describe an extension to that flow-sensitive Andersen analysis which supports strong updates.
- We show that FS-Andersen has cubic complexity both with and without strong updates.
- We combine FS-Andersen with the construction of interprocedural SSA form. For data-flow analysis this yields a cubic algorithm to construct SSA from scratch with FS precision for real-world programs. For points-to analysis this gives a truly sparse FS-PTA.
- We extend FS-Andersen to compute the meet-over-all-valid paths (MOVP) solution, based on ideas of the IFDS framework. This for the first time applies IFDS to a (sparse) FS-PTA.
- We show how the FS-MOVP-Andersen can support strong updates and prove that both MOVP versions have complexity $\mathcal{O}(n^4)$. This is a substantial improvement over the previous $\mathcal{O}(n^6)$ bound found by Landi.

2 Related Work

The problem of determining pointer targets is central to static analyses and thus has attracted many researchers. This resulted in a large body of publications, so that we can only discuss a small portion of them here. An older article by Hind [Hin01] gives an overview and also introduces some dimensions of precision that help to classify PTAs. Unfortunately, this article also made the opinion popular that flow-sensitivity probably is not worth the price we have to pay for it.

This was based on earlier articles by Hind and Pioli [HP98,HP00]. However, these articles only considered small programs and did not combine flow-sensitivity with field- or context-sensitivity. Moreover, they compared their CFG-based FS-PTA with an FI-PTA that is more precise than usual in that it computes a solution per object *and function*. And finally, they did not investigate the benefits of higher precision for subsequent client analyses which clearly benefit from smaller points-to sets. Thus, we should be careful with the interpretation of their results. We hope that this paper reinforces the interest in flow-sensitivity, because our new analysis has several advantages over prior FS-PTAs.

Andersen's analysis [And94] is the most famous FI-PTA and probably the most influential PTA today. It uses a fixpoint iteration which repeatedly detects new constraints (over-estimated data-flow relations between objects) and propagates them to detect pointer targets at dereferences. The propagation, which basically solves the problem of dynamic transitive closure (for selected roots of the constraint graph), dominates the costs and makes it a cubic algorithm. Andersen also investigated the problem of making this approach flow-sensitive [And94, p. 147]. However, he did not succeed and just mentioned problems with that idea. This paper for the first time solves the challenge and thus opens the world of higher precision for the most influential PTA.

Many researchers used Andersen's simple approach and extended it in some way. The most popular idea to accelerate it was the detection and contraction of cycles in the constraint graph [FFSA98,HT01,HL07]. Reps [Rep98] showed how to apply the IFDS framework [RHS95] to a variant of Andersen's analysis to compute a FI-MOVP solution instead of the context-insensitive MOP (meet-over-all-paths) solution. Pearce developed an efficient field-sensitive version for C [Pea05,PKH07] which in the following was used by several other researchers. We also use his approach for the field-sensitive version of FS-Andersen.

Hasti and Horwitz [HH98] tried to make an FI-PTA flow-sensitive in that they iteratively construct SSA form on the PTA's result and then use the new variable names to improve the precision towards flow-sensitivity. However, nobody ever proved that this in fact results in a FS-PTA, and recently [HL09] even conjectured this being not the case. Also, the approach seems to be unable to support strong updates, an important benefit of FS-PTAs. In comparison, our analysis does not start with over-estimated results, trying to reduce the degree of over-estimation; instead, we start with unsound results and add more targets until we have a sound solution.

Real FS-PTAs are traditionally seen as classical data-flow problems. Consequently, researchers published approaches which solve the pointer analysis as a problem on the CFG [CWZ90,HP98]. However, using the CFG as a basis causes some problems. For example, strong updates prevent the problem from being monotone and distributive, and the analysis has to propagate and store points-to relations for several objects at every CFG node. Thus publications tried to move towards more data-flow oriented structures like the SEG or VDG [CBC93,Ruf95]. However, they were only able to use over-estimated data-flow because the detection of data-flow relations itself requires pointer targets. Our analysis elegantly

```
Preparation;
while not Fixpoint_Reached loop
   Update_Propagation_Graph;      -- step 1
   Propagate_Pointer_Targets;     -- step 2
end loop;
Finalization;
```

Fig. 1. Outline of generalized Andersen analysis

copes with this mutual dependency and for the first time uses the real data-flow relations as the basis. On the data-flow graph we simply have the problem of dynamic transitive closure, which is monotone and distributive.

[HL09] is a recent FS-PTA which uses SSA form for those variables that cannot be pointer targets. However, for other variables and heap objects they fall back to the classical, expensive CFG-model. [Tok07] describes an idea how the online construction of SSA form can accelerate an existing FS-PTA. However, the FS-PTA in his approach still solves a data-flow problem on the CFG and does not use SSA edges for propagation. We will use Tok's ideas for the incremental SSA construction in our approach.

[LR92] published a FS-PTA which achieves some context-sensitivity through inspection of alias sets. The runtime of this analysis, however, is in $\mathcal{O}(n^6)$. This paper improves the upper bound for the FS-MOVP problem to $\mathcal{O}(n^4)$.

[Kah08] described a pipeline of pointer analyses in which each stage reduces the input for the next stage. Our analysis is orthogonal to such ideas and thus can be combined with the pipeline approach to yield a faster algorithm.

In the past few years, researchers used binary decision diagrams (BDDs) to represent points-to sets [BLQ$^+$03,HL07,Wha07,WL04,ZC04,Zhu02,Zhu05,HL09]. Publications based on them showed considerable speed-ups. In the worst case, however, BDDs require exponential costs, and the behaviour depends on the variable ordering. A good variable ordering seems to require a lot of experimentation [BLQ$^+$03,Wha07]. In this paper, we use traditional data structures to achieve a more predictable analysis behaviour, including a cubic worst case. However, the analysis can also be used together with BDDs.

3 Generalization of Andersen's Analysis

Figure 1 shows an outline of our analysis. We use the *propagation graph* as generalization of Andersen's constraint graph. FI-Andersen still uses the constraint graph for this, while FS-Andersen uses the data-flow graph. Only step 1 differs for FS-/FI-Andersen: it adds new constraints (FI) or data-flow relations (FS) to the propagation graph. Step 2 updates its transitive closure. We briefly sketch how the analysis addresses several dimensions of precision.

Preparation: This part addresses the dimension of field-sensitivity. It extracts the program's (initial) objects and modifies the intermediate representation (IR).

Field-insensitive: creates objects of composite type, but not for fields. Accesses to fields are mapped to accesses of the composite object.
Field-sensitive [PKH07]: creates objects for fields, but not for composites. Accesses to the struct are mapped to accesses to the first field (except for struct copies, which map to copies for all fields).

Step 1: This part addresses the dimension of flow-sensitivity.
Flow-insensitive: Assignments (including parameter passing) are used to infer constraints between objects at newly discovered definitions and uses.
Flow-sensitive: Assignments (including parameter passing) yield data-flow edges from uses to definitions. Additionally, control-flow paths yield edges from definitions to uses (reaching definitions). Here we optionally respect strong updates. Section 4 will provide more details on how this can be implemented efficiently.

Step 2: This part addresses the dimension of context-sensitivity.
Context-insensitive: Standard approaches to compute the dynamic transitive closure compute the MOP solution. For example, we could annotate each node (in SSA: each definition and ϕ node) with a bitvector, having one bit per possible target. The propagation then uses a depth-first search starting at every new edge to update these bits. Here, strategies to contract cycles can be used optionally.
Context-sensitive: The IFDS-based idea to compute the MOVP solution is to compute summary edges for functions. They help to identify the call sites to which a target should (not) be propagated back. Section 5 will provide more details on how this can be implemented efficiently.

Finalization: This part prepares the results for following client analyses. FS it may prune the resulting SSA graph, and FI it may use one FS step to construct an SSA graph with FI pointer targets.

Using the FI version of step 1 gives one possible formulation of Andersen's analysis. In this paper, we focus on using the FS version. We will prove the correctness of this variant and discuss strong updates and MOVP propagation as extensions to that FS-Andersen analysis.

4 Sparse Flow-Sensitive Andersen Analysis

The flow-sensitive counterpart of Andersen's constraint graph is the data-flow graph. We use an interprocedural SSA form (ISSA) as the concrete shape of that graph to have a sparse analysis. This section first briefly describes ISSA form, then provides details on its incremental construction in step 1 of the analysis.

4.1 Interprocedural SSA Form

Interprocedural SSA form extends SSA [CFR$^+$91] to include representations for parameter passing and side-effects. We call these extensions *(sub)entries* and *(sub)exits*: entries are nodes in a function's CFG entry and represent artificial definitions for parameters, uninitialized variables and may-use side-effects. Exits are nodes in the CFG's exit and represent artificial uses for the return value and

may-def side-effects. Subexits are located right before a call and complement the callee's entries; locally in the caller they act as use nodes. Subentries are located immediately after a call and complement the callee's exits; locally in the caller they act as definition nodes.

A definition may be weak or strong. ISSA represents weak updates as definitions with an incoming edge (similar to a ϕ node with one predecessor). Subentries can also be weak or strong, depending on whether they represent a may- or must-def side-effect. At direct calls, we always model them as strong updates: In case a subentry v actually represents a may-def, there will be a data-flow path $d \rightarrow u \xrightarrow{*} v$ through the called function, where d is the reaching definition of v and u a subexit at the same call site as v. This interprocedural path propagates values (and thus pointer targets) like a local edge from d to v. Appendix A shows an example ISSA graph. Notice that, for the purpose of propagating pointer targets, we could also use a reduced form only containing all kinds of definitions.

We briefly recap important SSA properties that will be used in our incremental ISSA construction: a definition (including ϕ nodes, entries and subentries) dominates all of it uses, and a use node has exactly one definition. ϕ nodes reside in the iterated dominance frontier of such a definition. We can treat a ϕ node as combination of a definition and one use node per preceding basic block.

4.2 Incremental ISSA Construction

As with Andersen's constraint graph, we construct the ISSA graph incrementally during the analysis. Each iteration updates the graph based on new definitions and uses reported by the previous propagation step. In the first iteration, we treat all direct definitions and uses as new nodes in this sense.

The FS step 1 of our analysis loops over all new nodes and dispatches to expert functions for every kind of node. These experts might add induced nodes (such as ϕ nodes) to the set of new nodes, which will then be handled later in the same step 1. The actions executed in the expert functions to update the ISSA graph are inspired by Tok's work [Tok07].

Some of them have to find *the* reaching definition for the new node. This is where we benefit from the dominance properties and optionally respect strong updates. The reaching definition has to dominate the new node; thus we can use the CFG's dominance tree: walking up that tree until we find a definition (including ϕ nodes etc.) for the new node's object is an efficient solution. Figure 2 shows this function, using Is_Reaching_Def to compare the objects of N and $IDom$. It returns the reaching definition (if found) and a flag indicating whether a potential strong update resides in between that definition and the new node. If such a potential strong update later turns out to be in fact a strong update, it will be the reaching definition instead of the returned node. We therefore do not directly create the edge from the returned definition to the new node if indirect strong updates should be supported. Section 4.3 will provide more details on this strategy.

If the function could not find a reaching definition, we have to create an artificial one, namely an entry node. For a local variable of a non-recursive

```
function Find_Reaching_Def (N : Node) return (Node, Boolean)
is
   IDom : Node := N.Dominator; Blocked : Boolean := False;
begin
   while IDom ≠ null loop
      if Is_Reaching_Def (IDom, N) then return (IDom, Blocked);
      if Support_Strong_Updates and not Blocked and IDom ∈ B then
         Add N to IDom.Blocked_Nodes;
         Blocked := True;
         if not Is_General_Def (N) then return (IDom, Blocked);
      end if;
      IDom := IDom.Dominator;
   end loop;
   return (none, Blocked);
end Find_Reaching_Def;
```

Fig. 2. Searching the reaching definition via dominance tree

subprogram, this represents an uninitialized variable; otherwise it represents a may-use side-effect on a non-local object and also causes a subexit at all call sites to the function (the side-effect is propagated).

The following paragraphs provide details on the algorithms used to handle the different kinds of nodes. For all of them, let n be the new node in function f.

New use nodes, subexits, and exits. Use Find_Reaching_Def to determine the reaching definition d. If there is no potential strong update along the data-flow edge $d \to n$, then create the edge (if $d =$*none*, create an entry node and use it as d). If n is an exit node on a non-local object, add corresponding subentries at all relevant call sites, and link n to them.

New definitions and subentries. Add ϕ nodes in the dominance frontier of n (it is not necessary to place them in the *iterated* dominance frontier: this will happen automatically because the ϕ nodes in turn create other ϕ nodes in their dominance frontier). Search the reaching definition d via Find_Reaching_Def. If found, use the method shown in Figure 3 to adjust data-flow edges. Namely, if u is some use node attached to d with n dom u (assuming explicit use nodes at the end of preceding basic blocks for ϕ nodes), then move u from d to n: n now resides in between d and u, and a new search for the reaching definition of u would find n. This necessity for adjusting the data-flow edges is the reason why Find_Reaching_Def returns a reaching definition even if there is a potential strong update in between.

In case n is a weak update, insert a new edge $d \to n$ if there is no potential strong update in between. (To support indirect strong updates, an indirect definition with exactly one target is seen as must-def; if later a second target arrives, we also add this edge for the first target.) And finally, in case n defines a non-local object, create an exit node if it not already exists.

```
procedure Move_Dominated_Uses (From : Node; To : Node)
is
begin
   for all U ∈ succ (From) loop
      if To dom U then
         move U into succ (To);
      end if;
   end loop;
end Move_Dominated_Uses;
```

Fig. 3. Move dominated use nodes to the new definition

New ϕ nodes. A new ϕ node acts like a definition in its basic block and like a use node in all preceding basic blocks (the "entries" of n). Compared to a definition, there are basically two differences:

1. No need to create an exit node for a side-effect (because the definition inducing n already did it)
2. Use nodes of the reaching definition d for an entry of n may only be captured by n if d dom n.

That is, as for a new definition, first create the induced ϕ nodes in the dominance frontier. Then search the reaching definition d for every entry of n. If none found and no potential strong update was found on the way, create an entry and use it as d. Create the edge from d to the entry, if not blocked by a potential strong update. In case d dom n (can happen for at most one definition), inspect their use nodes to adjust the data-flow edges with Move_Dominated_Uses (d,n).

New entries. If n represents a non-local object, add the corresponding subexit nodes at call sites to f and link them to n.

We emphasize that these simple actions result in flow-sensitive side-effects without any complication by callgraph cycles. Moreover, there is no need for a precomputed callgraph approximation or the computation of some traversal order, as for example required by [CRL99].

We discussed one possible implementation for the problem of finding the reaching definition. It is basically a linear search in the CFG's dominance tree. It might improve efficiency in practice if one does not search along the CFG's dominance tree, but instead uses different, small dominance trees *per object* and function. Assume a pre- and postorder numbering of the CFG's dominance tree as for example used to perform dominance tests as constant-time interval inclusion tests [TGL06]. This also assigns an interval to every definition. These intervals can then be used to construct the reduced dominance tree per object which only includes the object's local definitions. Now the search for the reaching definition at some node n basically locates n's interval in the reduced dominance tree for n's object. This in general reduces the steps needed for the search.

Replacing the CFG-based search also is in line with our general idea of switching from the CFG to the data-flow representation. In fact, the reduced dominance tree per object can be used as a compact representation of the object's data-flow. However, when potential strong updates should be respected, an additional strategy of looking up the last node in B (see next subsection) between two given intervals is needed. This amounts to representing the graph B according to dominance relation using the intervals, too.

4.3 Support For Indirect Strong Updates

The basic idea of our strong updates extension is to treat potential strong updates as real strong updates until the opposite is proven (by a second pointer target). In the course of the analysis the number of pointer targets for an indirect definition increases monotonically. Therefore we only consider indirect definitions without targets and with a single target as potential strong updates. If there is already a single target, the indirect definition can only affect data-flow edges for that target, and only if the single target allows strong updates (e.g., this excludes heap objects and locals of recursive programs). If there is no target yet, we have to assume that the indirect definition might affect all objects for which we support strong updates and which can be targets of pointers (it for example cannot affect stack objects of which the address is never taken).

In a real execution, every definition d is a strong update and prevents the data-flow from preceding definitions of the same object to subsequent uses. A static analysis as ours overestimates the possible flows of control of all executions and thus has no linear ordering of definitions and uses as in an execution. Therefore, we have to restrict the set of "preceding" definitions and "subsequent" uses to definitions *definitively* preceding d and uses *definitively* following d in all executions. These relations are mainly captured by the notion of *interprocedural dominance*: we may only prevent data-flow edges from definitions p to use nodes u with p dom d and d dom u.

For indirect definitions with a single target, we can fall back to local dominance because of having side-effects and treating subentries as strong updates. For indirect definitions without a target, we use a so-called *blocking graph* B to achieve the same. A node in B represents a CFG position where a strong update or a node induced by such a definition might appear. As shown in Figure 2, the search for the reaching definition basically stops at such a node to prevent data-flow edges from preceding definitions. Once a first target for an indirect definition is found, it (and induced nodes) will be removed from B. Then we restart the search for the reaching definition for all nodes remembered in the removed nodes' sets Blocked_Nodes. Edges in the blocking graph point from a node to nodes induced by it.

To create B, we initially (in the preparation) add all indirect definitions. Then we create the closure of induced nodes: if a node $v \in B$ residing in function f dominates f's CFG exit, all call sites to f have to be added to B because a subentry for a must-def side-effect might appear there (v will be executed definitively when the call site is reached). Moreover, for all $v \in B$, the beginning

```
procedure Create_Blocking_Graph
is
begin
   Nodes := Edges := Phis := Phi_Pred (·) := ∅;
   for all v ∈ Indirect_Definitions loop
      add v to Nodes;
   end loop;
   -- add nodes induced by new nodes
   for all new v ∈ Nodes loop
      for all w ∈ DF(v) loop
         if w ∈ Nodes then
            for all p ∈ pred(w), v dom p loop
               add v → w[p] to Edges;
            end loop;
         else
            if w ∈ Phis then
               for all p ∈ pred(w), v dom p loop
                  add v to Phi_Pred (w,p);
               end loop;
               if Phi_Pred (w,p) ≠ ∅ ∀p ∈ pred (w) then
                  move w to Nodes and Phi_Pred (w,·) to Edges;
               end if;
            else
               add w to Phis;
               for all p ∈ pred(w), v dom p loop
                  Phi_Pred (w,p) := {v};
               end loop;
            end if;
         end if;
      end loop;
      if v dom CFG_Exit (func(v)) then
         for all c ∈ calls to func(v) loop
            if c ∉ Nodes then
               add c to Nodes;
            end if;
            add v → c to Edges;
         end if;
      end if;
   end loop;
   use Tarjan's algorithm to compute SCC-DAG;
end Create_Blocking_Graph;
```

Fig. 4. Construction of the initial blocking graph

of basic blocks in the iterated dominance frontier for v has to be added to B because a ϕ node might appear there. However, we may only include places where the potential ϕ node would be created because of nodes in B for all preceding basic blocks (otherwise it will not be a strong update).

```
procedure Update_Blocking_Graph (V : Node)
is
begin
   for all (N,E) ∈ V.Blocked_Nodes loop
      -- restart search for reaching definition
      if Is_Phi_Node (N) then
         Postponed_New_Phi (N, E);
      elsif Is_Definition_Or_Subentry (N) then
         Postponed_New_Def (N);
      else
         Postponed_New_Use (N);
      end if;
   end loop;
   remove V from Nodes (B);
   for all S ∈ succ(V) loop
      S.Incoming_Edges := S.Incoming_Edges - 1;
      if S.Incoming_Edges = 0 then
         Update_Blocking_Graph (S);
      elsif Is_Phi_Candidate (S) and then
         not ∀p ∈ pred(S) ∃pred (S,p)∈B
      then
         Update_Blocking_Graph (S);
      end if;
   end loop;
end Update_Blocking_Graph;
```

Fig. 5. Updating the blocking graph when a first target arrives at V

Figure 4 presents the algorithm for creating B. Notice that we compute the SCC-DAG of B to be able to update i correctly in case of cycles. Figure 5 shows the function we call once a first target arrives at an indirect definition. This function updates B (using the SCC-DAG) to remove the indirect definition and all induced nodes. For every node removed in this way, we reconsider the blocked nodes (which in turn basically searches the reaching definition again).

5 Context-Sensitive Propagation Step

Reps [Rep98] already showed how IFDS [RHS95] can be used to achieve MOVP precision for FI-Andersen. We focus on applying IFDS to FS-Andersen. For this purpose, this section describes the MOVP version of step 2 for the analysis.

5.1 IFDS

IFDS was designed to compute the MOVP solution for data-flow problems on the interprocedural CFG (ICFG; "supergraph" in [RHS95]). For that, it creates the exploded ICFG by duplicating each ICFG node with all possible data-flow facts (adding one special fact λ). The problem's transfer functions then translate to minigraphs connecting the data-flow fact nodes for two subsequent ICFG nodes. Together they constitute the exploded ICFG's edges. Finally, the so-called tabulation algorithm solves a special graph reachability problem on the exploded ICFG to compute the MOVP solution.

In our case, we have to compute the MOVP solution on the ISSA graph. Analogously to the ICFG version, we create the exploded ISSA graph in annotating each node with all possible pointer targets. The transfer functions are simple, namely the identity for most ISSA edges, and a constructor for a given pointer target if the node creates it via address-taking or heap allocation.

However, applying the tabulation algorithm to this only computes the transitive closure for the current iteration. We need the dynamic transitive closure, and there seems to be no good incremental version of the tabulation algorithm. Moreover, we want to optionally distinguish heap allocations in the calling context. This and the desire to have an algorithm similar to the depth-first search (DFS) used for the MOP solution led us to a new algorithm.

5.2 Propagation with Abstract and Concrete Targets

Our approach distinguishes *abstract* and *concrete* pointer targets. An abstract target is always connected to some entry node: it is a placeholder for all pointer targets of that entry and thus represents a set of context-dependent targets. An abstract target for an entry of function f is only used at ISSA nodes in f. Once an abstract target for entry e arrives at a local exit node x, we have a *summary edge* $e \to x$ that we store at e. Likewise we store *shortcut edges* from entries to local subexits and to local dereferences. Definitions (including ϕ nodes etc.) now have both a set of concrete and abstract targets to denote their points-to set. At a dereference pointing to an abstract target, we create ISSA nodes for all targets of the corresponding entry.

The propagation of pointer targets correspondingly consists of two different DFS routines: one for concrete targets, and the other one for abstract targets. Concrete targets are context-independent and thus basically propagated as usual; the exception is at call sites where they should be translated into an abstract target for the callee. Here we distinguish two cases:

1. If the callee's entry has not been visited yet, we simply process it using the DFS for abstract targets.
2. Otherwise we use the entry's summary and shortcut edges:
   - If the concrete target is not already known at the entry, propagate it to dereferences (locally and in transitive callees) using shortcut edges.
   - For all exits reachable via summary edges from the entry, continue the DFS for the concrete target at the caller's subsequent subentry.

The propagation of abstract targets has to translate the target both at call sites and after returning from the function. Let e be the entry of function f to which the current abstract target is connected. At a call site, we proceed similar to a concrete target, except that we have to use the shortcut edges for *all* targets of e if there is no current concrete target (may happen in the incremental setup). At an exit node x, we now respect the semantics of function calls: at a subentry node following x in some caller, the abstract target translates to all targets of e *coming from that call site*. To realize this, we simply inspect the points-to set of *the* subexit preceding e at the given call site (stored at the subexit's definition).

The differentiation of abstract and concrete targets allows us to distinguish context-dependent and context-independent targets. This in turn allows us to distinguish heap allocations in the calling context: if a concrete target at some exit node stems from a locally allocated heap object, we can create subobjects per subentry following the exit node. For a subobject, the subentry then acts as the allocation site, so that the simple strategy also works transitively. However, we have to be careful within callgraph cylces and if more than one exit of the same function propagates the same heap object (here we should use only one subobject per caller).

This sketches our new algorithm. Notice that strong updates can be supported as in the context-insensitive version, since that only affected step 1. It is not difficult to derive an incremental version of the propagation: for this, step 1 of the analysis collects a list of new edges, and step 2 processes this list. For each new edge $u \rightarrow v$, the propagation starts for all concrete and abstract targets of u not yet present at v, with some care at interprocedural edges (to simulate the actions described above). Appendix C contains an example to illustrate our ideas.

Figures 6 and 7 show the general algorithms for the two DFS. The current concrete target is stored in the global variable Current_Target, whereas the current abstract target is stored in Current_Entry. The function Heap_Refinement is used to distinguish heap allocations in the context. It basically creates a new subobject for the given heap object. The Visit_*_Node functions for definitions check whether we already visited that node with the current target, to avoid endless recursion and doing the same work again. At a dereference, the current target is used to create a new ISSA node. The context-sensitive case is shown in Figure 8.

At an exit node, the propagation for an abstract target executes the special actions shown in Figure 9 to respect the semantics of function calls. At a subexit, both kinds of propagations have to translate the current target into the abstract target of the callee, if not yet visited. Otherwise shortcut and summary edges must be used. Figure 10 shows these actions for the context-sensitive case, while Figure 11 shows it for the case of a concrete target arriving at the subexit. The utility function to use shortcut edges is shown in Figure 12.

```
procedure Visit_CI (V : Node) is
begin
   if Is_Dereference (V) then
      Add_Target (V, Current_Target);
   elsif Is_Subexit (V) then
      Visit_CI_Subexit (V);
   elsif Is_Exit (V) and then
      Is_Locally_Allocated (V, Current_Target)
   then
      Heap_Refinement (V);
   else
      if Visit_CI_Node (V) then
         return;
      end if;
      for all S∈succ (V) loop
         Visit_CI (S);
      end loop;
   end if;
end Visit_CI;
```

Fig. 6. DFS for the propagation of concrete targets

```
procedure Visit_CS (V : Node) is
begin
   if Is_Dereference (V) then
      Visit_CS_Dereference (V);
   elsif Is_Exit (V) then
      Visit_CS_Exit (V);
   elsif Is_Subexit (V) then
      Visit_CS_Subexit (V);
   else
      if Visit_CS_Node (V) then
         return;
      end if;
      for all S∈succ (V) loop
         Visit_CS (S);
      end loop;
   end if;
end Visit_CS;
```

Fig. 7. DFS for the propagation of abstract targets

```
procedure Visit_CS_Dereference (V : Node) is
begin
   Add V to Derefs (Current_Entry);
   if Current_Target = None then
      for all T∈Targets(Current_Entry) loop
         Add_Target (V, T);
      end loop;
   else
      Add_Target (V, Current_Target);
   end if;
end Visit_CS_Dereference;
```

Fig. 8. Actions for an abstract target at a dereference

```
procedure Visit_CS_Exit (V : Node) is
   Old_Entry : constant Entry := Current_Entry;
begin
   Add V to Exits (Current_Entry);
   for all S∈Succ (V) loop
      P := Pred (Old_Entry) at Callsite (S);
      for all T∈Abstract_Targets (P) loop
         Current_Entry := T;
         Visit_CS (S);
      end loop;
      if Current_Target = None then
         for all T∈Targets (P) loop
            Current_Target := T;
            Visit_CI (S);
         end loop;
         Current_Target := None;
      elsif Current_Target∈Targets (P) then
         Visit_CI (S);
      end if;
   end loop;
   Current_Entry := Old_Entry;
end Visit_CS_Exit;
```

Fig. 9. Respecting function call semantics at an exit node

```
procedure Visit_CS_Subexit (V : Node) is
   Old_Entry : constant Entry := Current_Entry;
begin
   Add V to Subexits (Current_Entry);
   for all S∈Succ(V) loop
      -- S is an entry node
      if #Targets (S) = 0 then
         if Current_Target = None then
            for all T∈Targets (Old_Entry) loop
               Add T to Targets (S);
            end loop;
         else
            Add Current_Target to Targets (S);
         end if;
         Current_Entry := S;
         Visit_CS (S); -- creates summaries, visits successors
      else -- use summaries
         -- propagate target(s) down
         if Current_Target = None then
            for all T∈Targets (Old_Entry) loop
               if T∉Targets (S) then
                  Propagate_Down (S, T);
               end if;
            end loop;
         else
            if Current_Target∉Targets (S) then
               Propagate_Down (S, Current_Target);
            end if;
         end if;
         -- continue propagation at current call site
         for all X∈Exits (S) loop
            N := Succ (X) at Callsite (V);
            Visit_CS (N);
         end loop;
      end if;
   end loop;
   Current_Entry := Old_Entry;
end Visit_CS_Subexit;
```

Fig. 10. Actions when an abstract target arrives at a subexit

```
procedure Visit_CI_Subexit (V : Node) is
begin
   for all S∈Succ(V) loop
      -- S is an entry node
      if #Targets (S) = 0 then
         Add Current_Target to Targets (S);
         Current_Entry := S;
         Visit_CS (S); -- creates summaries, visits successors
      else -- use summaries
         -- propagate target down
         if Current_Target∉Targets (S) then
            Propagate_Down (S, Current_Target);
         end if;
         -- continue propagation at current call site
         for all X∈Exits (S) loop
            N := Succ (X) at Callsite (V);
            Visit_CI (N);
         end loop;
      end if;
   end loop;
end Visit_CI_Subexit;
```

Fig. 11. Actions when an concrete target arrives at a subexit

```
procedure Propagate_Down (E : Entry; T : Object) is
begin
   for all S∈Derefs (E) loop
      Add_Target (S, T);
   end loop;
   for all X∈Subexits (E) loop
      for all S∈succ (X) loop
         if #Targets (E) > 0 and then
            T∉Targets (E)
         then
            Propagate_Down (S, T);
         end if;
      end loop;
   end loop;
end Propagate_Down;
```

Fig. 12. Using shortcut edges to propagate a new target for an entry

6 Correctness of the Flow-Sensitive Andersen Analysis

Termination of the analysis is obvious: the data-flow graph grows monotonically (since an edge is only replaced or refined by a path), and a finite upper bound is given by the complete graph using every object as possible pointer target. However, using unsound intermediate results requires a closer look at the correctness of the analysis. We thus provide a formal correctness proof, at first for the context-insensitive version without strong updates. Strong updates will be considered later, and the correctness of the context-sensitive version stems from that of IFDS.

Let $\mathcal{E}$ be an execution of the program being analyzed, executing read and write accesses $V = v_1, \ldots, v_n$ in this order. For each $v_i \in V$ let $mem(v_i)$ be the memory location accessed (extensions to structured memory blocks can be added easily). For a read access $v_i \in V$ let $def(v_i)$ be the write access on the same memory location having the largest subscript smaller than i, thus defining the value read by v_i. In case there is no such write access, and for v_i being a write access, let $def(v_i)$ undefined.

Definition 1. *The* execution graph $G_{exe}(\mathcal{E})$ *consists of the nodes V and the edges given by $def(v_i) \to v_i$ for all $i \leq n$ such that $def(v_i)$ is defined.*

For simplicity, we assume a language without pointer arithmetics or parallelism. We also require the full intermediate representation (IR) or suitable summaries for missing parts to be available to the analysis. We assume $\mathcal{E}$ does not dereference a pointer without valid target, because then the behaviour is undefined.

Let O be a mapping from memory locations to analysis objects, assigning a single analysis object to every $mem(v_i)$. Applying O normally maps several different memory locations to the same object, merging different nodes of G_{exe} into one.

Definition 2. *Two nodes $v_i, v_j \in G_{exe}$ are* equivalent *if they are the same kind of access, have the same position in the IR, and if $O(mem(v_i)) = O(mem(v_j))$. We denote the equivalence class of v_i by $[v_i]$.*

Now define G'_{exe} to be the execution graph where equivalence classes of nodes are collapsed into single nodes. Let G_{ana} be the data-flow graph computed by our analysis (irrespective of the concrete shape, i.e. we may assume direct connections between definitions and uses instead of ISSA form). We inductively show $G'_{exe} \subseteq G_{ana}$, thereby proving correctness. The induction considers the v_i in turn and shows both $[v_i] \in G_{ana}$ and that we know the pointer target of this node in case it is a pointer access. Assuming correctness of the reaching-definitions analysis, this implies that G_{ana} also includes the edges of G'_{exe}.

The induction starts with v_1. v_1 as the first action in $\mathcal{E}$ cannot be pointer-indirect, thus $[v_1]$ is included in the initial data-flow graph. If v_1 is a pointer definition, the target must be literally there in the source code and is thus known to the analysis.

Symbol	Meaning
n, m	nodes / edges of the ISSA graph
t_{calls}	targets for indirect calls
n_{ir}	nodes in the input IR
$t_{initial}$	potential targets
f, c	subprograms and call sites
g	non-local objects
i	number of iterations

Table 1. Symbols used to determine the complexity

Now consider some $v_i \in G_{exe}$. If v_i is not the result of some pointer dereference, $[v_i]$ is already added to G_{ana} at the beginning. Otherwise there is a $v_j, j < i$, being a pointer dereference resulting in v_i. Inductively, we have $[v_j] \in G_{ana}$ and know the target, i.e., $O(mem(v_i))$. But this results in the creation of the pointer-indirect node $[v_i]$ in step 2 of some iteration.

Thus in all cases, $[v_i] \in G_{ana}$, and by applying a reaching-definitions analysis also $[def(v_i)] \to [v_i] \in G_{ana}$ for a read access. By induction we then know the target of $[def(v_i)]$, and thus the next execution of step 2 propagates this target to $[v_i]$. Similarly, the target for a write access is either literally given or comes from the (use node of the) right-hand side of the assignment. This proves:

Theorem 1. *The analysis terminates with sound results.*

6.1 Correctness with Strong Updates

Consider $[v_j] \to [v_i] \in G'_{exe}$ for some $v_i, v_j \in G_{exe}$ with $j < i$. By induction, we still have $[v_i], [v_j] \in G_{ana}$, and we know the pointer target at $[v_j]$. The edge $[v_j] \to [v_i]$ now might be delayed by an indirect definition without targets or with a single target. But because of interprocedural dominance, $\mathcal{E}$ executed that indirect definition as some v_k before v_i ($k < i$), and likewise $j < k$. By induction, we already know at least one target for $[v_k]$; thus, if the data-flow edge is affected, $O(mem(v_k)) = O(mem(v_i))$ and therefore $v_j \neq def(v_i)$. That means: it is correct for that situation not to create $[v_j] \to [v_i]$ if we have exactly one target for $[v_k]$. If $\mathcal{E}$ later executes other elements of the same equivalence class, a second target for $[v_k]$ (if applicable) makes the indirect definition weak and also creates the edge.

7 Complexity of the Flow-Sensitive Andersen Analysis

We use the symbols explained in table 1 to compute the size of the results and the flow-sensitive analysis' complexity. It is reasonable to assume that, as programs grow, the number of subprograms increases – but neither the number of parameters nor the number of statements per subprogram. Under this assumption, the number of parameters as well as the size of dominance trees and dominance

frontiers are bounded by constants per subprogram. We again start with the context-insensitive version without strong updates, and add these aspects later. For the field-sensitive version, g grows, but can still be bound by n_{ir}, so that we assume no impact on the asymptotic complexity.

7.1 Size of the results

Lemma 1. *The ISSA graph has at most $n \in \mathcal{O}(g * f)$ nodes.*

Proof. Within a subprogram, there is at most one node per expression (two for calls) for a locally accessed object. Additionally, there can be one ϕ node per object and basic block. The number of local nodes for an object is thus linear in the size of a subprogram, which is a constant. At most $g + const$ many objects are locally visible per subprogram.

In the following, let $|CG| = f + c + t_{calls}$ be the size of the callgraph. In practice we find $t_{calls} \ll n_{ir}$ and thus a linear bound on $|CG|$.

Lemma 2. *The ISSA graph has at most $m \in \mathcal{O}(g * |CG|)$ edges.*

Proof. For a call edge we have only constantly many edges for parameter passing and two per non-local object. The number of interprocedural edges is thus bounded by $\mathcal{O}(g * (c + t_{calls}))$. There are only constantly many edges for copy assignments in a subprogram, and thus at most $\mathcal{O}(f)$ of them in total. Definitions, uses, subexits, and exits have at most one incoming edge. ϕ nodes have one incoming edge per preceding basic block, but this again is a constant. With the previous lemma we therefore have at most $\mathcal{O}(g * f)$ intraprocedural edges.

7.2 Runtime

Theorem 2. *The runtime is in $\mathcal{O}(t_{initial} * g * |CG|)$.*

Proof. Instead of counting iterations and determining the costs per iteration, we compute the sum for step 1 of all iterations (σ_1) and likewise the sum for step 2 of all iterations (σ_2). Initialization is linear in the program size. It is therefore negligible and the complexity is given by $\sigma_1 + \sigma_2$.

Step 1. Step 1 considers every node of the final ISSA graph once, giving n steps through the worklist. Searching for a reaching definition has constant costs because of the constant height of a (precomputed) dominance tree. All these searches therefore have total costs of $\mathcal{O}(n)$ (constantly many searches per node). Adding subentries, subexits, and their connections to exits and entries costs $\mathcal{O}(g * |CG|)$. The size of a (precomputed) dominance frontier is a constant. The costs for adding ϕ nodes are thus given by $\mathcal{O}(n)$. The number of uses connected to a definition, ϕ node, entry, or subentry is a constant because there are at most constantly many nodes per subprogram and object. Since a dominance test is in $\mathcal{O}(1)$, capturing dominated uses therefore has costs of $\mathcal{O}(n)$. Using the lemmas we thus find $\sigma_1 \in \mathcal{O}(n + m) = \mathcal{O}(g * |CG|)$.

Step 2. Let us assume an incremental propagation which operates on the list of newly added edges and stores already propagated targets at all kinds of definitions. It considers every edge only once as new and thus as starting point for a propagation of targets. But because we store the targets at the nodes, an edge is visited only once per target over all iterations. Then $\sigma_2 = \mathcal{O}(t_{initial} * (n + m))$, which proves the theorem.

Corollary 1. *FS-Andersen has the same complexity as FI-Andersen.*

Proof. Using $t_{calls} \ll n_{ir}$, we have $t_{initial}, g, f, c, t_{calls} \ll n_{ir}$.

Thus, as with FI-Andersen, the propagation dominates the runtime and makes both FI-Andersen and FS-Andersen cubic algorithms. FS-Andersen of course offers higher precision and additionally comes up with a full data-flow representation. In practice, online cycle detection proved useful to accelerate FI-Andersen, so we expect similar benefits for FS-Andersen; however, it does not improve the complexity.

7.3 Complexity with Strong Updates

Treating indirect definitions with a single target as strong updates does not cause any more costs. For indirect definitions without targets, we have the costs of managing the blocking graph B and restarting the search for a reaching definition once a first target is known.

Creating and maintaining the blocking graph B adds and removes every indirect definition, call site, and basic block beginning at most once. It is thus linear in the size of the callgraph (without indirect calls) plus the number of indirect definitions. In terms of complexity, this is already captured by the initialization's complexity.

The search for a reaching definition of some node u can be restarted several times, but at most once for all local dominators. This number is a constant, and a single search also has constant costs. Respecting indirect definitions without targets thus adds costs of $\mathcal{O}(n)$.

Summing all up, we can see that the strong updates extension actually does not increase the analysis' complexity.

7.4 Complexity for Context-Sensitive Version

The context-sensitive extension computes summary edges which we add to m. We can have at most g^2 summary edges per call site. Thus, we now have $m \in \mathcal{O}(g * |CG| + g^2 * c)$. We consider the version without context-sensitive heap allocations.

Propagating concrete targets is a normal DFS if we ignore the actions at subexits. We therefore account for it with costs $\mathcal{O}(t_{initial} * (n + m))$. This also holds for abstract targets if we ignore the actions at (sub)exits and entries, but this time we visit a node at most once per local entry. Since there are at most

g entries per function, this adds $\mathcal{O}(g * (n + m))$. An entry is visited once per initial target; the down-propagation along shortcut edges visits at most all local subexits from every entry. The terms used so far for the propagation thus already account for these costs.

At an exit, we at most loop over all successors of the exit for all g entries. Here we have costs g per edge to a successor, so we add $\mathcal{O}(g^2 * (c + t_{calls}))$. At a subexit we inspect reachable exits if the following entry e was already visited. When the summary edges to these exits were stored at e, the subexit did not carry the abstract or concrete target that we currently propagated to the subexit. Thus, the costs are accounted for by doubling the costs for exits. Summing all up gives the time bound of

$$\mathcal{O}((g + t_{initial})(n + m) + (c + t_{calls})g^2).$$

Again, the incremental version of the propagation achieves the same time bound over all iterations as one run over the final graph would have. With the bounds for n and m, and using $t_{initial} < g$, we find:

Theorem 3. *FS-MOVP-Andersen has the complexity* $\mathcal{O}(g^2 * |CG| + g^3 * c)$, *which is in* $\mathcal{O}(n_{ir}^4)$.

8 Summary, Future Work and Conclusions

In this paper we presented a novel combined analysis which simultaneously computes pointer targets and a data-flow representation. From a pointer analysis perspective, it is a sparse flow-sensitive version of the most influential analysis as invented by Andersen. Despite the higher precision it has the same cubic complexity and thus might clear the way for widespread use of flow-sensitivity. From a data-flow analysis perspective, it is a cubic algorithm to compute interprocedural SSA form (including side-effects) from scratch, respecting pointer-indirect operations with high precision.

In contrast to most other flow-sensitive pointer analyses, our analysis is truly sparse: it does not solve an expensive (non-monotone, non-distributive) data-flow problem on the CFG, but instead solves (monotone and distributive) standard dynamic transitive closure on the data-flow graph. To be even more sparse, we were able to use SSA form as the shape of this data-flow graph for all objects. Operating on a data-flow graph has the advantage that we have to store the targets for the single object represented at a data-flow node only. In contrast, CFG-based analyses have to manage points-to relations for all objects at all CFG nodes. Moreover, CFG-based propagation is more complicated, while we have to solve a standard problem. We expect that the numerous published ideas to accelerate the computation of dynamic transitive closure for FI-Andersen are also highly applicable to our FS-Andersen analysis.

This directly leads over to the next step of our work, namely an empirical evaluation. We are currently in the progress of implementing and evaluating the

generalized analysis as outlined in Section 3. This will allow us to directly compare FI-Andersen and FS-Andersen with different settings for field- and context-sensitivity. Preliminary results show that FS-Andersen is able to analyze at least 200 KLoC even without BDDs and any cycle detection or similar accelerations. This would be substantially better than previous non-BDD-based flow-sensitive analyses.

The context-sensitive version of our analysis for the first time gave a $\mathcal{O}(n^4)$ bound on the FS-MOVP pointer problem. This improves the previous $\mathcal{O}(n^6)$ bound found by Landi. Future work will try to use our abstract targets to improve the precision even further with two ideas:

1. In alias-free situations we might use abstract targets as objects for ISSA nodes, not just for pointer targets. This reduces the size of the graph and allows a more precise propagation of side-effects to only those callers that cause the side-effect.
2. Abstract targets might also be useful in the search for strategies to support strong updates for heap objects: a single abstract target at a dereference may be used to detect a *local* strong update irrespective of the entry's concrete targets. However, the side-effect might not be strong, so that additional strategies are needed.

Our promising results encourage us to investigate further the benefits of our combined analysis as a solution to the hot core topics of program analysis. The work presented in this report is part of the author's PhD thesis which is expected to be finished in summer 2009. The thesis will describe the ideas of this report in more detail and it will contain the results of our empirical evaluation.

References

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[BLQ+03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114, New York, NY, USA, 2003. ACM.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, New York, NY, USA, 1993.

[CFR+91] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant Context Inference. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, New York, NY, USA, 1999. ACM Press.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *PLDI '90: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310. ACM, 1990.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. *SIGPLAN Notices*, 33(5):85–96, 1998.

[HH98] Rebecca Hasti and Susan Horwitz. Using Static Single Assignment Form to Improve Flow-insensitive Pointer Analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 97–105, 1998.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE '01: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, Snowbird, UT, USA, 2001. ACM Press.

[HL07] Ben Hardekopf and Calvin Lin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *PLDI '07: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, New York, NY, USA, 2007.

[HL09] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, New York, NY, USA, 2009. ACM.

[HP98] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*, pages 57–81, London, UK, 1998. Springer-Verlag. Extended version of the SAS paper in Lecture Notes in Computer Science, vol. 1503.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *ISSTA '00: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, New York, NY, USA, 2000. ACM.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *PLDI '01: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, New York, NY, USA, 2001. ACM.

[Kah08] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–259, New York, NY, USA, 2008. ACM.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI '92: Proceedings of the Conference on Programming Language Design and Implementation*, pages 235–248, New York, NY, 1992. ACM Press.

[Pea05] David J. Pearce. *Some Directed Graph Algorithms and Their Application to Pointer Analysis*. PhD thesis, University of London, February 2005.

[PKH07] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007. Article 4.

[Rep98] Thomas Reps. Program Analysis via Graph Reachability. *Information and Software Technology*, 40(11-12):701–726, December 1998.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *PLDI '95: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, New York, NY, USA, 1995. ACM.

[TGL06] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient Flow-Sensitive Interprocedural Data-flow Analysis in the Presence of Pointers. In *CC '06: Proceedings of the 15th International Conference on Compiler Construction*, pages 17–31. Springer, 2006.

[Tok07] Teck Bok Tok. *Removing Unimportant Computations in Interprocedural Program Analysis*. PhD thesis, University of Texas, August 2007.

[Wha07] John Whaley. *Context-Sensitive Pointer Analysis Using Binary Decision Diagrams*. PhD thesis, Stanford University, Stanford, CA, USA, March 2007.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the Conference on Programming Language Design and Implementation*, pages 131 – 144. ACM Press, June 2004.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–157, New York, NY, USA, 2004. ACM.

[Zhu02] Jianwen Zhu. Symbolic pointer analysis. In *ICCAD '02: Proceedings of the IEEE/ACM International Conference on Computer-aided Design*, pages 150–157, New York, NY, USA, 2002. ACM.

[Zhu05] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *DAC '05: Proceedings of the 42nd Conference on Design Automation*, pages 831–836, New York, NY, USA, 2005. ACM.

Appendices

A Example ISSA Graph

Figure 13 shows an excerpt of an example program that we use throughout the appendices to illustrate the analysis. Both h and k call g which in turn calls f. The most interesting points are the indirect definitions in f and g and the impact of how precise we capture them.

```
int **p, **q;
int *x,*y;
int a,b;
void f() {**p = 1; q = p;}
void g() {x = &a; f(); *x = 3;}
void h() {a = 4; p = &x; g(); x = *q; b = a;}
void k() {b = 2; p = &y; y = &b; g(); y = *q;}
```

Fig. 13. Example program

Figure 14 shows the interprocedural SSA graph for the example as computed with the context-insensitive FS-Andersen analysis. We can see a few (sub)entries and (sub)exits for side-effects as well as the definitions and uses for direct and indirect operations. Edges are oriented in data-flow direction and are also shown for the data-flow from right to left at an assignment. We do not show interprocedural edges in order to avoid an overcrowded diagram. They would connect a caller's subexits with the callee's entries, and the callee's exits with the caller's subentries.

The context-insensitive version determines both x and y as targets for p and q. In f, this results in an indirect definition for both a and b. The corresponding side-effects propagate back to all callers. Context-insensitive, dereferencing q in h and k yields two targets, but one of them is a result of invalid paths. Without the support for strong updates, all indirect definitions are weak. This causes the indirect definition for a in g to be weak although in fact we have only one target that would allow a strong update.

Notice that, if for example the $use(q)$ in h is considered first in iteration 1, it causes a local $entry(q)$ to be created; later, once the side-effect on q is propagated (within the same iteration), the corresponding $subentry(q)$ captures the use node. The final pruning then removes $entry(q)$ because it has no outgoing edges. Similarly, there is no $subentry(b)$ in h because the final pruning removed it. The ISSA graph shown in the figure assumes that all exit nodes of h and k are used in some caller; otherwise, pruning would affect them, too.

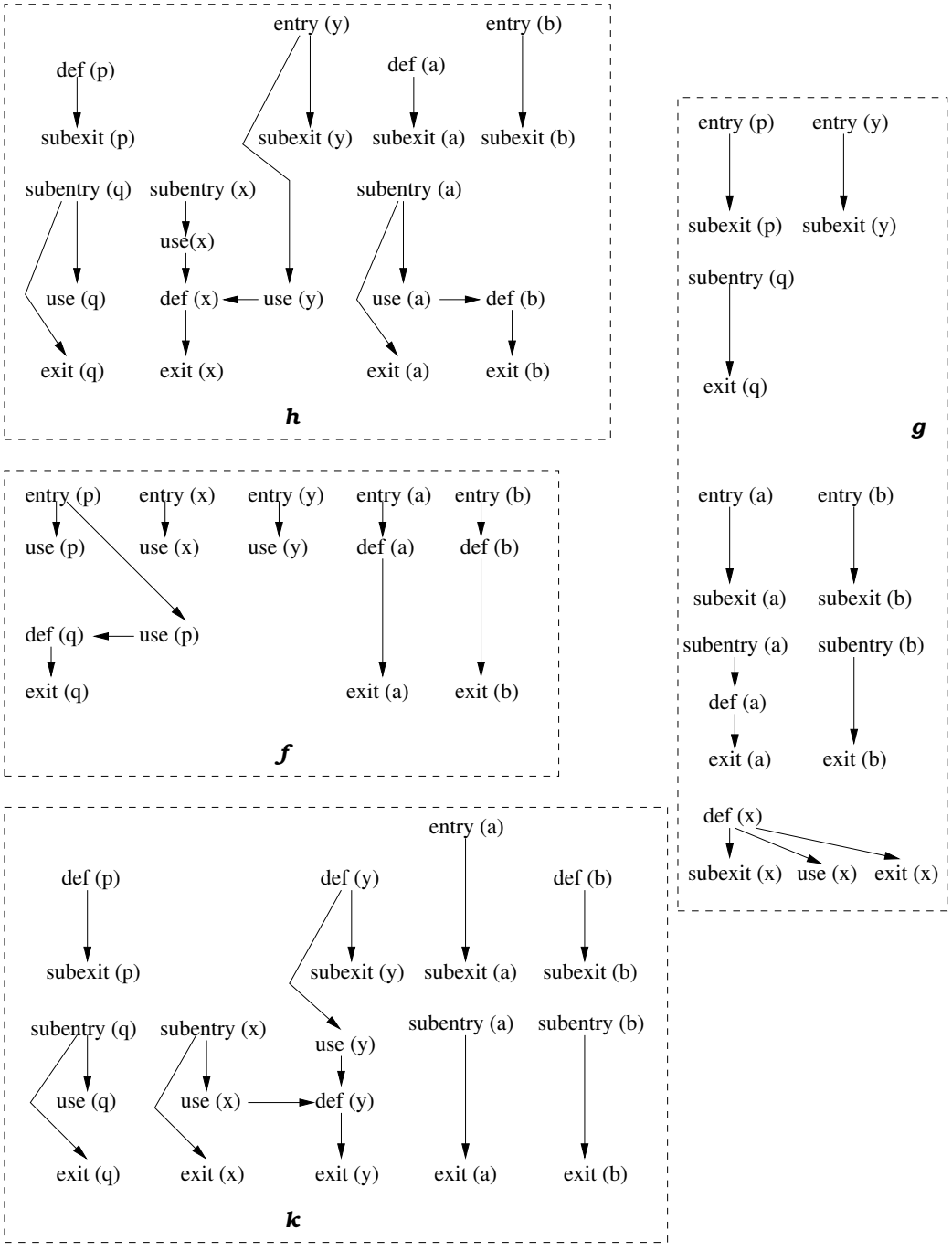

Fig. 14. ISSA graph for the example program without interprocedural edges

B Example with Strong Updates

Table 2 shows the initial blocking graph B for the example program. It contains all indirect definitions and the side-effects that could be induced by them. As soon as a target for an indirect definition is found, it will be removed from the graph, including outgoing edges and induced nodes which then no longer have incoming edges.

The effect of the blocking graph as respected in the search for the reaching definition is for example that the $use(a)$ in h will be blocked at the preceding call to g. The table also shows this set of blocked nodes per node in B. Remember that only ISSA nodes for potential pointer targets can be blocked. Since the program never takes the address of p and q, they cannot be targets. Thus, all nodes referring to them will never be blocked.

Node	Position	Induced nodes	Blocked_Nodes
1	`**p = 1`	3	-
2	`*x = 3`	4, 5	$exit_g(x)$
3	subentry after call to f	4, 5	$use_g(x)$
4	subentry after call to g in h	-	$use_h(a), use_h(q)$
5	subentry after call to g in k	-	$use_k(q), exit_k(y), exit_k(b), exit_k(x)$

Table 2. Initial blocking graph and nodes blocked in iteration 1

Delaying the search for a reaching definition (temporarily) reduces the number of data-flow edges and thus the paths along which pointer targets are propagated. In this case, the first iteration still propagates both x and y as targets for p to `**p = 1`, but no targets arrive at any indirect definition. The second iteration transitively connects the indirect uses for x and y in f with their definitions in g and k, respectively. This allows the propagation of a and b as targets for the indirect definition in f. This in turn removes nodes 1 and 4 from the blocking graph. The following step 1 then again searches the definition for the previously blocked node $use_g(x)$ (and succeeds), which also allows the propagation of the single target a to `*x = 3`. At this time, all remaining nodes will be removed from B. `*x = 3` then has only one target and is thus considered a strong update – no edge to the preceding subentry for a will be created, and this will remain in effect until the fixpoint is reached. As a consequence, the subentry has no outgoing edge and can be pruned, which in turn also prunes all nodes of a in f as they are unused. Continuing the pruning removes all remaining nodes of a in g and then determines the definition for a in h as unused; in k, subexit and entry for a will be removed. Thus, in the end the small improvement of respecting the indirect strong update removes a substantial part of the ISSA graph. Figure 15 shows the resulting graph, again without interprocedural edges.

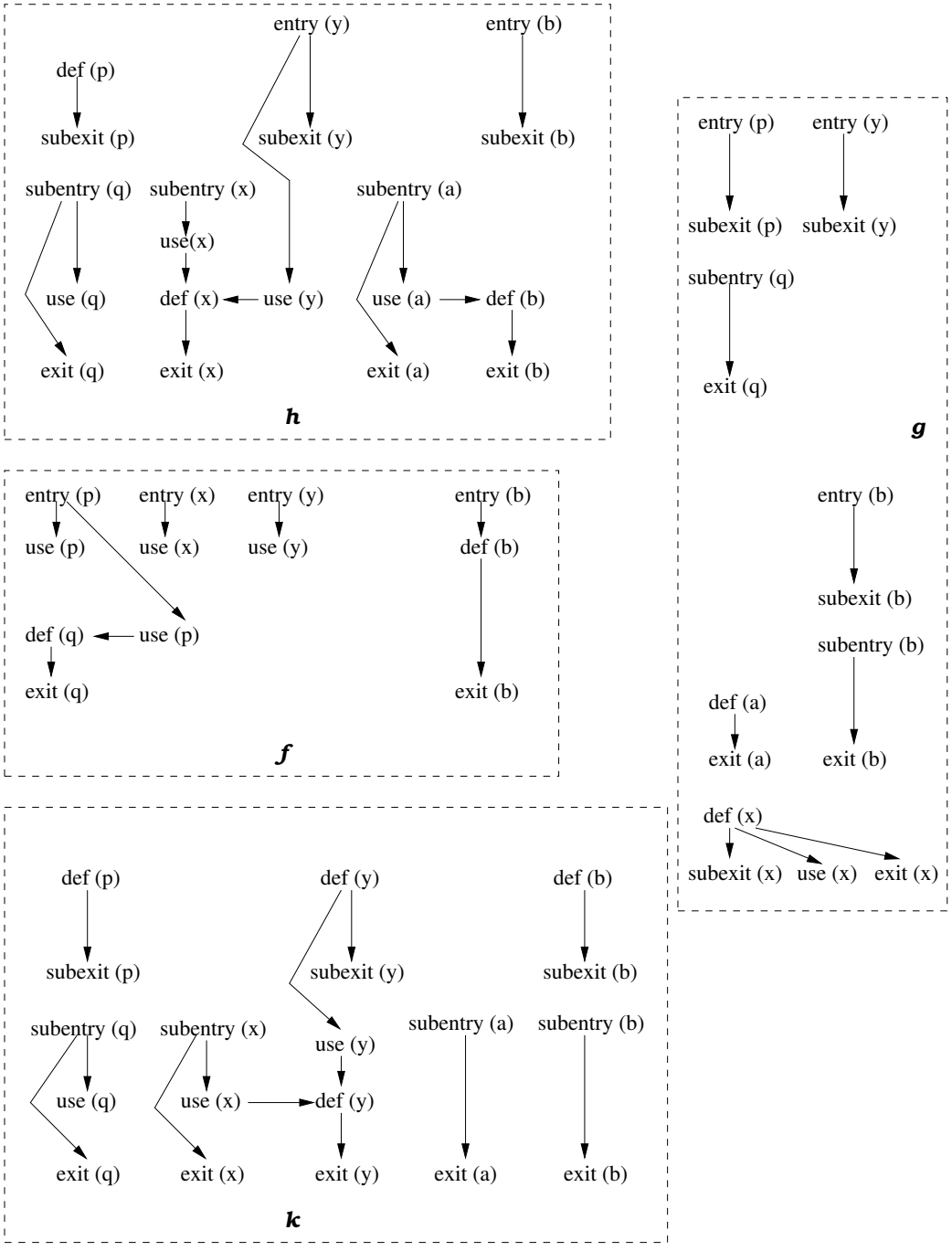

Fig. 15. ISSA graph for the example when strong updates are supported

C Context-Sensitive Example

Using our context-sensitive propagation increases the precision for the example both with and without strong updates. Let us first consider the situation without strong updates. In function f we now propagate the abstract target corresponding to $entry_f(p)$ to q's definition and thus to $exit_f(q)$. At the call site in g, this translates to whatever we passed as targets to $entry_f(p)$ – in this case, the abstract target corresponding to $entry_g(p)$. This in turn arrives at $exit_g(q)$ and needs to be translated similarly at the call sites in h and k. That is where the increased precision comes in: in h, we only had x as a target, and in k we only had y. Therefore, q in both functions now only points to a single target, whereas the context-insensitive version determined two targets. A subsequent optimization now could remove both `y = *q` and `x = *q` because they only assign a variable to itself. Figure 16 shows the result. For clarity we also annotated the definitions and subentries with their points-to sets. Notice that although we were able to improve the points-to sets, for example the side-effect on y is still propagated to both h and k.

Figure 17 shows the result for the combination of both extensions, strong updates and context-sensitive propagation. Here the effects also combine: the indirect strong update for a allows to prune several nodes, and the higher precision in propagation reduces the targets for q.

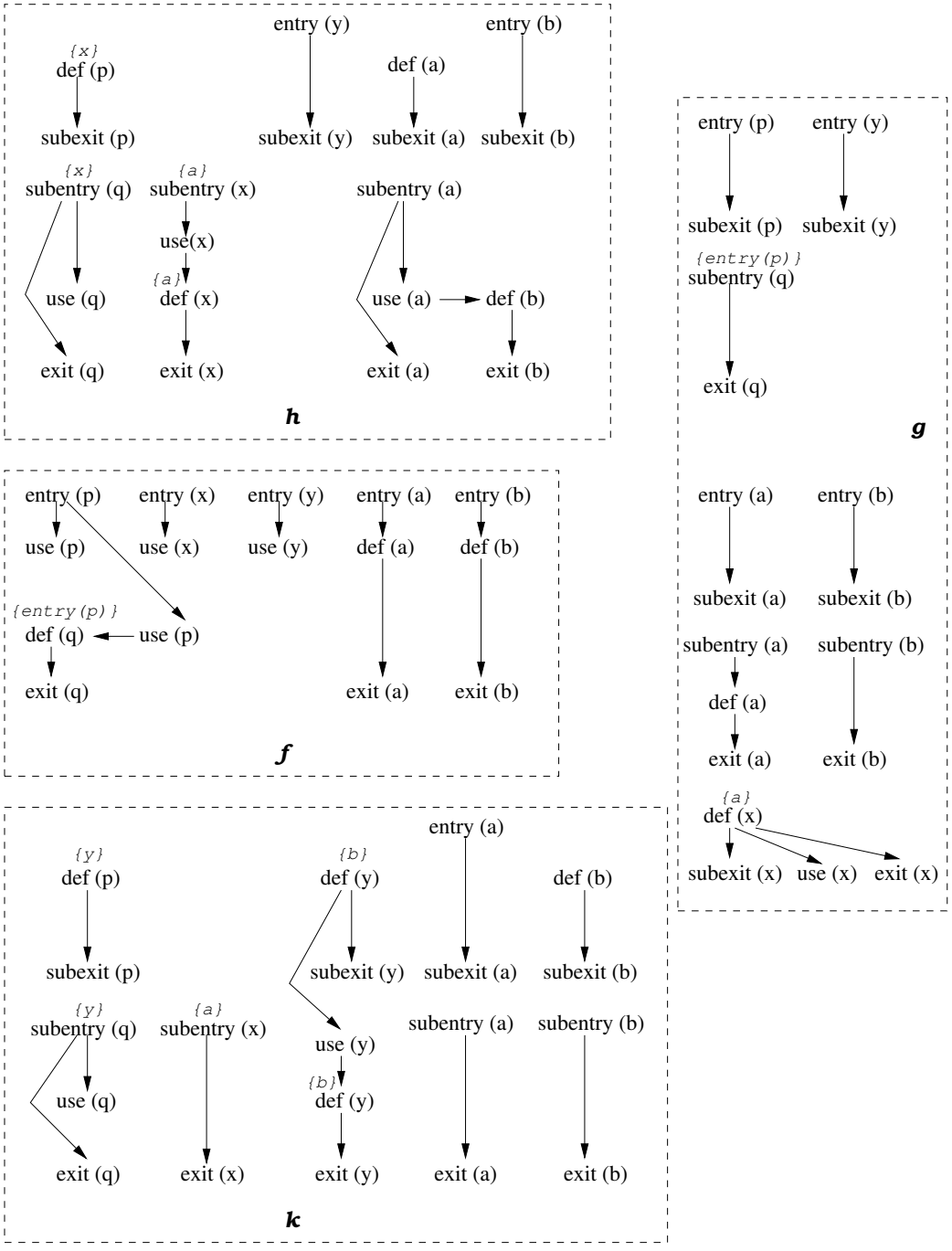

Fig. 16. ISSA graph for the example with context-sensitive propagation

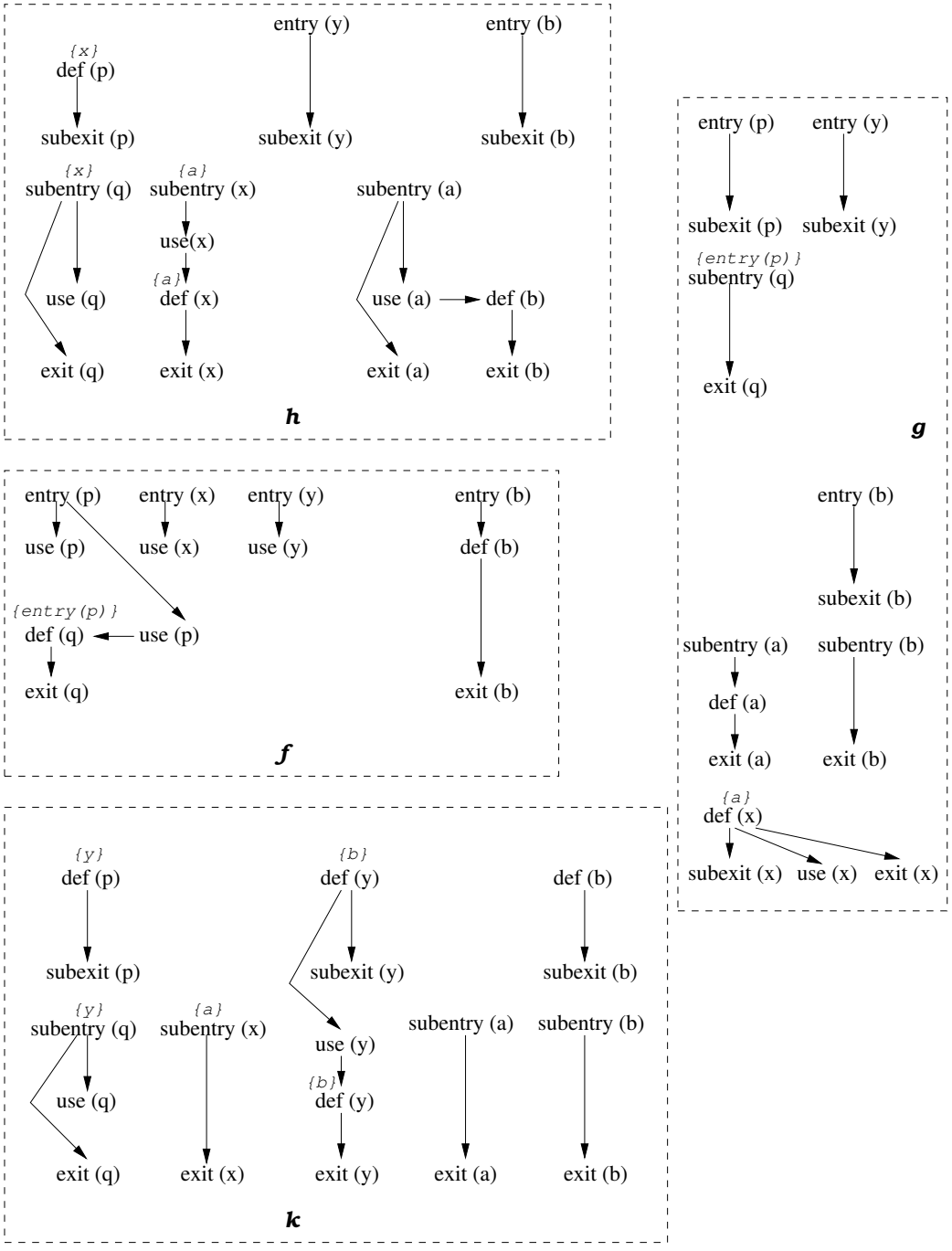

Fig. 17. ISSA graph with context-sensitive propagation and strong updates