Frameworks for Intra- and Interprocedural Dataflow Analysis

Craig Chambers, Jeffrey Dean, and David Grove

Department of Computer Science and Engineering
University of Washington
{chambers,grove,jdean}@cs.washington.edu

Technical Report 96-11-02
November 1996

Frameworks for Intra- and Interprocedural Dataflow Analysis

Craig Chambers, Jeffrey Dean,* and David Grove

Department of Computer Science and Engineering
University of Washington
{chambers,grove,jdean}@cs.washington.edu

UW CSE Technical Report 96-11-02

Abstract

Because dataflow analyses are difficult to implement from scratch, reusable dataflow analysis frameworks have been developed which provide generic support facilities for managing propagation of dataflow information and iteration in loops. We have designed a framework that improves on previous work by making it easy to perform graph transformations as part of iterative analysis, to run multiple analyses "in parallel" to achieve the precision of a single monolithic analysis while preserving modularity and reusability of the component analyses, and to construct context-sensitive interprocedural analyses from intraprocedural versions. We have implemented this framework in the Vortex optimizing compiler and used the framework to help build both traditional optimizations and non-traditional optimizations of dynamically-dispatched messages and first-class, lexically-nested functions.

1 Introduction

The heart of an optimizing compiler is its suite of dataflow analyses. Dataflow analyzers are hard to build from scratch. Fortunately, many analyses share mechanisms for managing information flowing through the program representation, for reaching fixpoint in the presence of loops, and so on, and it is common to develop reusable *dataflow analysis frameworks* or *generators* that encapsulate the recurring mechanisms [Tjiang & Hennessy 92, Hall *et al.* 93, Hendren *et al.* 93, Yi & Harrison 93, Alt & Martin 95, Dwyer & Clarke 96, Assmann 96, Adl-Tabatbai *et al.* 96]. A client dataflow analysis is implemented simply by instantiating or parameterizing the generic framework with the particulars of that analysis.

We have designed, implemented, and used a framework for implementing intra- and interprocedural dataflow analyses as part of the Vortex optimizing compiler [Dean *et al.* 96]. Our framework is like others in that it is based on a lattice-theoretic model of dataflow analysis [Kildall 73, Kam & Ullman 76]: clients provide an implementation of the domain of analysis, including operations to copy, merge, and compare domain elements, and the flow or transfer functions over domain elements for each kind of instruction in the control flow graph program representation. Our engine then propagates domain elements through the control flow graph in topological order, iterating to reach fixpoint in the presence of loops. Forward and backward analyses are supported uniformly.

Our framework includes additional features beyond those in previously-described frameworks. Our primary goal is to make it easy to implement and experiment with new intra- and interprocedural analyses. In addition, the capabilities of our framework are greatly influenced by Vortex's focus on optimizing higher-level languages featuring object-oriented dynamic dispatching and first-class functions, where complex graph transformations and analyses need to be interleaved for effective optimization. The main contributions of our framework are the following:

* Dean's current address: DEC WRL, 250 University Avenue, Palo Alto, CA 94301; jdean@pa.dec.com

- Clients can write transfer functions that perform **transformations as part of analysis**. This avoids the need for analyses to simulate the effect of later transformations in order to reach the best fixpoint (e.g., simulating dead assignment elimination or constant folding during live variables analysis or constant propagation, respectively), and it makes it practical for the effects of complex transformations such as procedure inlining and elimination of branches with constant predicates to be including during analysis. For example, inlining can make intraprocedural analysis much more effective, particularly in higher-level languages, but in the presence of dynamic dispatching or first-class functions, iterative analysis is needed to compute the possible callee(s) at call sites and enable the inlining. Consequently, it is important to be able to perform inlining, based on intermediate analysis results, while engaged in iterative analysis, in order to reach the best solutions [Chambers & Ungar 90]. Our framework manages the details of acting only tentatively on any transformations applied during iterative analysis, undoing the transformations' effects if iteration causes the code to later be reanalyzed.
- Clients can **compose intraprocedural dataflow analyses** so that they run "in parallel," interleaving the analyses' flow functions at each flow graph node. Each composed analysis can examine the intermediate dataflow information computed by the other composed analyses in order to share results and reach better fixpoints faster than would be possible by running each analysis separately in sequence [Click & Cooper 95]. In addition, complex monolithic analyses can be broken up into modular analysis components that may be reused as part of other analyses, and new analysis components can be written and added to existing composed analyses easily. For example, separate modules performing alias analysis, constant folding, common subexpression elimination, elimination of branches with constant predicates, receiver class set analysis, execution frequency estimation, and inlining can be composed to build an aggressive analysis for optimizing dynamic dispatches and indirect function calls.
- Clients can **construct interprocedural analyses directly from intraprocedural analyses**. The intraprocedural analysis determines whether the interprocedural analysis is flow-sensitive or -insensitive and whether flow-sensitive problems are forward or backward (the interprocedural framework itself is independent of these aspects), while the framework provides a uniform method for clients to control context sensitivity. Bottom-up and top-down summary analyses are special cases of our general framework, but fully interprocedurally flow-sensitive analyses are nearly as easy to build.

The next three sections of this paper describe each of these main contributions. Each section concentrates on the interface presented by our framework to clients, motivating it and explaining its advantages over previous interfaces; the full paper discusses some of the implementation details. Section 5 describes our experience using these interfaces and identifies some areas for future work, and section 6 compares our design to other analysis frameworks.

2 Defining Intraprocedural Analyses

Our framework performs dataflow analyses over control flow graphs whose nodes are individual three-address-code-style [Aho *et al.* 86] statements. Dataflow information at program points is drawn from a lattice of possible dataflow information. As is traditional in dataflow analysis (but opposite to the conventions followed in abstract interpretation), the top lattice element represents the best possible (most optimistic) information, while the bottom lattice element represents the worst possible (most conservative) information. At control flow merge points, the lattice meet operator takes the greatest lower bound of the inflowing information to compute the best conservative information possible after the merge. A flow function for each kind of statement maps inflowing domain elements to resulting domain elements. At loops, iterative approximation computes the greatest fixpoint solution to the domain equations represented by the flow functions. Figure 1 specifies the interface to our intraprocedural traversal framework, Figure 2 specifies the operations required by our framework of domain elements, Figure 3 specifies the main kinds of flow-function outcomes our framework supports (some additional more specialized outcomes are also supported), and Figure 4 provides an example client analysis, dead assignment elimination.

To define an analysis, a client must define:

- the data structure that represents the program-point-specific information (a concrete subclass of `AnalysisInfo`), along with the `copy`, `merge` (meet), and `as_general_as` (to test whether fixpoint is reached) operations over domain elements;
- the flow functions for the different kinds of statements in the control flow graph;
- the direction of analysis (`Forward` or `Backward`);
- the initial domain element at the entry arc (or exit arc, for a backward analysis); and
- as an optimization, whether the analysis is guaranteed not to require iteration (`NonIterative`), for instance for a flow-insensitive traversal that simply wishes to visit each node in the graph.

The most interesting aspect of our intraprocedural framework is its integration of transformations with analysis. A flow function's result either specifies the analysis information to propagate to the successor(s) of the statement being analyzed (`ContinueResult` or `ContinueBranchResult`), or ask the framework to perform a local graph transformation. Statement transformations include:

- `ReplaceResult`: replacement with another statement (e.g., constant-folding an arithmetic operation into a simple assignment),
- `ReplaceGraphResult`: replacement with an entire subgraph of statements (e.g., inlining or lowering high-level statements into sequences of lower-level statements), and
- `DeleteResult`: deletion (e.g., eliminating dead assignments or branches with constant predicates).

The framework *logically* modifies the control flow graph as indicated by the transformation, automatically deleting any newly unreachable code in the process, and then restarts the analysis at the program point before the transformed statement with the same analysis information as before the transformation. If a transformation is invoked during iterative analysis of a loop (e.g., inlining), based on dataflow information computed during iterative analysis (e.g., the set of possible receiver classes of the message), then the transformation may need to be undone if the dataflow information leading to the transformation becomes more conservative. The framework automatically manages performing transformations only logically during iterative analysis, applying them permanently to the control flow graph only when fixpoint has been reached; client flow functions do not need to be concerned with whether transformations they invoke may be based on overly optimistic input analysis information.

Special treatment may be required of merges that represent loop-heads, if the lattice domain is very tall. For example, if performing range analysis, the elements of the domain are tables mapping variables to ranges of the form [low..high], where low and high are constants. The meet of two ranges [low_1..high_1] and [low_2..high_2] is [min(low_1,low_2)..max(high_1,high_2)]. However, if this meet is used unchanged when analyzing some loop where an initially-constant counter is incremented, many iterations will be required to eventually reach fixpoint at the range [initial-value..max-int], assuming bounded integer arithmetic. To avoid this problem, our framework invokes a special `generalizing_merge` function at loop-heads, which by default just calls `merge` but in situations like range analysis with tall lattices an alternative merge

```
-- Main interface to intraprocedural dataflow analysis:
traverse(cfg:ControlFlowGraph,                  -- the graph to analyze
         dir:enum{Forward,Backward},            -- the direction of analysis
         mode:enum{Iterative,NonIterative},     -- whether iteration may be needed
         initial_info:AnalysisInfo,          -- the info at the start (end, if backwards) node
         flow_fn:                            -- the function invoked on each stmt in cfg:
             λ(stmt:CFGNode,                        -- the stmt being analyzed
               inflowing_info:AnalysisInfo          -- info before (after, if backwards) stmt
               ):AnalysisResult   -- info after (before, if backwards) stmt, or transformation
         ):void;
```

Figure 1: Main `traverse` Interface

```
-- Operations required of domain elements during any analysis:
abstract class AnalysisInfo {
   copy():AnalysisInfo;                        -- make a copy, for analyzing branches
   merge(other:AnalysisInfo):AnalysisInfo;     -- perform meet operation
};

-- Additional operations required of domain elements during iterative analyses:
abstract class IterativeAnalysisInfo isa AnalysisInfo {
   -- perform meet & widening operation at loop-heads:
   generalizing_merge(other:IterativeAnalysisInfo):IterativeAnalysisInfo {
      return merge(other) }     -- default to regular merge function
   -- used when testing whether analysis has reached fixpoint:
   as_general_as(other:AnalysisInfo):bool;
};
```

Figure 2: `AnalysisInfo` Interface for Domain Elements

```
-- Possible outcomes of a flow function:
abstract class AnalysisResult { };

-- No transformation; compute info after stmt:
class ContinueResult isa AnalysisResult {
   info:AnalysisInfo;           -- the info after the stmt (copied if multiple successors)
};
class ContinueBranchResult isa AnalysisResult {
   true_info: AnalysisInfo;     -- the info for the true successor
   false_info:AnalysisInfo;     -- the info for the false successor
};

-- Transformation: Replace stmt with another stmt or subgraph:
class ReplaceResult isa AnalysisResult {
   replacement:CFGNode;                -- the stmt to replace analyzed stmt with
};
class ReplaceGraphResult isa AnalysisResult {
   replacement:ControlFlowGraph;    -- the subgraph to replace stmt with
};

-- Transformation: Delete stmt or branch:
class DeleteResult isa AnalysisResult { };
class DeleteBranchResult isa AnalysisResult {
   kept_successor:CFGNode;  -- which successor to keep; branch & other successor is removed
};
```

Figure 3: `AnalysisResult` Interface

```
-- Definition of domain: live variables
class LiveVars isa IterativeAnalysisInfo {
   -- internal representation: a set of variable names:
   live_set:set[VariableName];
   -- standard operations used by framework:
   copy():LiveVars {
      return new LiveVars(live_set.copy()) }
   merge(other:LiveVars):LiveVars {
      return new LiveVars(live_set.union(other.live_set)) }
   as_general_as(other:LiveVars):bool {
      return live_set.contains(other.live_set) }
   -- additional operations used by flow functions:
   add(var:VariableName):void { live_set.add(var) }
   remove(var:VariableName):void { live_set.remove(var) }
   is_live(var:VariableName):bool { return live_set.includes(var) }
};
empty_live_vars():LiveVars { return new LiveVars(empty_set()) }

-- Main analysis & transformation function for dead assignment elimination:
eliminate_dead_assignments(cfg:ControlFlowGraph):void {
   traverse(cfg, Backward, Iterative, empty_live_vars(),
            λ(stmt:CFGNode, info:LiveVars):AnalysisResult {
                if is_dead(stmt, info) then
                     -- do transformation:
                     return new DeleteResult()
                else
                     -- no transformation; propagate info to predecessor(s)
                     update_info(stmt, info);
                     return new ContinueResult(info)
            })
}

-- Helper functions for identifying dead calculations and updating live vars
is_dead(stmt:CFGNode, info:AnalysisInfo):bool {
   return stmt.has_no_side_effects() and
          forall def in stmt.defs() do
              not info.is_live(def)
}
update_info(stmt:CFGNode, info:AnalysisInfo):void {
   forall def in stmt.defs() do
      info.remove(def)
   forall use in stmt.uses() do
      info.add(use)
}
```

Figure 4: Implementation of Dead Assignment Elimination

rule that generalizes the merged result more quickly (e.g., by extending one end or the other of the range to its limit) can be used. Generalizing merge corresponds to the widening operator used in abstract interpretation [Cousot & Cousot 77].

To avoid unnecessary copy operations, some operations may modify an `AnalysisInfo` data structure in place. The `merge` and `generalizing_merge` operations may modify their receiver argument in place and return it as the result, and a flow function may modify its incoming `AnalysisInfo` data structure in place when forming the data member of the resulting `ContinueResult` info. The `AnalysisInfo` data structure should *not* be modified if the flow function triggers a transformation, to enable the same data structure to be reused when restarting analysis after the transformation.

Several benefits accrue from our framework's integration of analysis and transformation. First, optimizations becomes simpler to write: there is only a single pass, not separate analysis and transformation passes, and there is no need to maintain any intermediate data structures recording the output of the analysis pass that the transformation pass reads. Second, by performing the transformations as part of analysis, optimizations potentially can achieve more precise results faster than is possible with iterated sequences of analysis and transformation. For example, in the following code fragment:

```
x := 0;
while ... do
    ...
    x := x + 1;
end
```

where there are no other references to `x`, our dead assignment elimination optimization will remove both assignments to `x`. In contrast, no algorithm that first identifies dead assignments and then separately deletes them will discover that the `x` operations are dead, unless the analysis is modified to simulate the effect of the dead assignment elimination transformation as part of its analysis. Transformations such as constant folding and dead assignment elimination are relatively easy to simulate during analysis,* but transformations like inlining and folding conditional branches with constant predicates have much more complex effects on the control flow graph, and simulating these sorts of transformations during analysis is much more difficult. With our framework, no redundant, potentially complicated simulation of transformations is necessary; the framework automatically does this simulation for clients.

3 Composing Intraprocedural Analyses

Intraprocedural analyses often can leverage off of each others' intermediate results, enabling integrated analyses potentially to compute more precise solutions faster than separate analyses run in sequence until none makes any more changes [Click & Cooper 95]. For example, a pass integrating constant propagation and folding, alias analysis, receiver class set analysis, and inlining would be more powerful than each run separately, since each analysis and transformation interacts with the others, enabling the others to work better. However, writing a single monolithic analysis that incorporates each of these analyses would be difficult, and useful component analyses could not be reused or run in isolation.

* A system whose flow functions use fixed GEN and KILL bit-vectors for each kind of statement, such as an analyzer produced by Sharlit when using path compression [Tjiang & Hennessy 92], is *not* able to simulate the effect of flow-dependent optimizations like these.

Our intraprocedural analysis framework supports building a single integrated dataflow analysis by composing a group of separately-written component dataflow analyses. A composed analysis has the same interface as any dataflow analysis, with a domain data structure (`ComposedAnalysisInfo`) supporting the appropriate operations, and a flow function (`process_composed_analysis`) that is invoked for each flow graph statement. Internally, the `ComposedAnalysisInfo` class maintains a vector of component `AnalysisInfo`s, one for each component analysis. Figure 5 specifies the `ComposedAnalysisInfo` domain, and Figure 6 sketches a client of the interface.

The composed analysis's flow function invokes each of the component analyses' flow functions on their corresponding `AnalysisInfo` inputs. If each of the component analyses' flow functions return a `ContinueResult` outcome, then the composed analysis's flow function returns a `ContinueResult` outcome, whose data is a `ComposedAnalysisInfo` wrapping the vector of the component analyses' `ContinueResult` data values. However, if one of the component analyses' flow functions triggers a transformation, then that transformation is performed and the composed analysis is restarted. The first component analysis to choose a transformation takes precedence over any later analyses in the vector of component analyses; clients of the composed analysis interface specify the order of component analyses when they create a composed analysis. It has been our experience that there is a natural ordering of transformation passes, but in general it might be necessary to extend the composed analysis interface to provide more control.

Ordinarily, flow functions can modify their incoming `AnalysisInfo` arguments if they are producing a `ContinueResult` or `ContinueBranchResult` outcome. However, to enable restarting analysis with unchanged dataflow information if a later composed analysis selects a transformation, composable analyses must use an alternative `AnalysisResult` outcome, `CommitContinueResult` or `CommitContinueBranchResult`. These alternative outcomes contain functions which, when invoked on the incoming analysis information, produce the outgoing analysis information, possibly by side-effecting the incoming analysis information in the process. In a "two-phase" style, the composed analysis flow function invokes each component analysis's flow function, which in phase one decides whether or not to select a transformation, without any side effects. Only if all component analyses forgo transformations does the composed analysis flow function invoke the `CommitContinueResult` outcome functions to run phase two of the component analyses' flow functions.[*]

Analyses composed together can interact simply by performing transformations that make other component analyses work better. Additionally, component analyses can "snoop" on each other's intermediate dataflow information, just as if they were implemented as a single monolithic pass. For example, the function inlining transformation can examine the current execution frequency estimate to decide whether a call site is important enough to optimize. The composed analysis framework supports direct component interaction by allowing clients to test for and look up the analysis information for other composed analyses by name. By checking whether other analyses are present, a given analysis can become a reusable analysis module, run with different combinations of other analyses or in isolation, without change.

[*] One could avoid the need for special `CommitContinueResult` outcomes by disallowing modification of analysis information during a flow function or by making copies of the incoming analysis information before invoking a component flow function, but we felt that the simplicity and efficiency of allowing modifications to analysis information without introducing lots of copies outweighed the additional interface complexity.

```
-- Any AnalysisInfo to be composed must be a subclass of ComposableAnalysisInfo:
class ComposableAnalysisInfo isa AnalysisInfo {
   composed_analyses:ComposedAnalysisInfo; -- a pointer to all other analyses
};

-- The ComposedAnalysisInfo domain
class ComposedAnalysisInfo isa IterativeAnalysisInfo {
   -- internal representation:
   component_names:vector[string]; -- names of the analyses
   infos:vector[ComposedAnalysisInfo]; -- component infos
   -- the flow functions for each component analysis:
   component_flow_fns:vector[λ(CFGNode, AnalysisInfo):AnalysisInfo];
   -- standard operations used by framework:
   copy():ComposedAnalysisInfo;
   merge(other:ComposedAnalysisInfo):ComposedAnalysisInfo;
   generalizing_merge(other:ComposedAnalysisInfo):ComposedAnalysisInfo;
   as_general_as(other:ComposedAnalysisInfo):bool;
   -- additional operations to query info for individual composed analyses:
   includes_pass(name:string):bool {       -- test whether composed with some other pass
      return desc.component_names.includes(name) }
   pass_info(name:string):AnalysisInfo { -- lookup the info for a composed analysis
      return infos[desc.component_names.find_index(name)] }
};

-- Main function for creating the initial composed analysis info at the start of an analysis:
new_initial_composed_analysis(
   components:vector[                 -- a vector of component analyses:
      record[name:string,                           -- name of the component analysis
             initial_info:ComposableAnalysisInfo,   -- initial info for the component
             flow_fn:λ(stmt:CFGNode,                -- the component's flow function
                       info:ComposableAnalysisInfo):AnalysisResult]]
   ):ComposedAnalysisInfo;

-- Predefined flow function for composed analyses:
process_composed_analysis(stmt:CFGNode, info:ComposableAnalysisInfo
                          ):AnalysisResult;
```

Figure 5: Interface for Composed Dataflow Analyses

```
optimize_calls(cfg:ControlFlowGraph):void {
   let initial_composed_info := new_initial_composed_analysis(
      [{name := "Inlining", initial_info := ..., flow_fn := ...},
       {name := "ConstantFolding", initial_info := ..., flow_fn := ...},
       {name := "AliasAnalysis", initial_info := ..., flow_fn := ...},
       {name := "ClassSetAnalysis", initial_info := ..., flow_fn := ...},
       {name := "FrequencyEstimation", initial_info := ..., flow_fn := ...}
      ]);
   traverse(cfg, Forward, Iterative, initial_composed_info,
            λ(stmt:CFGNode, info:ComposedAnalysisInfo):AnalysisResult {
                return process_composed_analysis(stmt, info)
            })
}
```

Figure 6: A Client of `ComposedAnalysisInfo`

4 Constructing Interprocedural Analyses

Interprocedural analyses are particularly important for higher-level languages with small procedures, frequent calls, and complex interprocedural data flow and data structures. Unfortunately, due to the difficulty of constructing and managing interprocedural analyses, implemented interprocedural analyses are relatively rare. Context-sensitive and/or flow-sensitive interprocedural analyses are even harder to build and consequently even rarer in practice.

Our framework strives to make interprocedural analyses, including context-sensitive and flow-sensitive ones, easy to construct, given three inputs:

- an intraprocedural version of the interprocedural problem to be solved,
- a function for specifying the degree of context sensitivity, and
- a program call graph.

The intraprocedural analysis determines whether the analysis is forward or backward and whether it is flow-sensitive or -insensitive. In the discussion that follows, input information to an intraprocedural analysis refers to information before the entry for forward problems or to information after the procedure's return for backward problems, and output information refers to the reverse. Input and output information can be drawn from different `AnalysisInfo` domains. Our interprocedural analysis interface is described in Figure 7, with Figure 8 defining a few common context sensitivity strategies. A sample client of our interprocedural framework, interprocedural constant propagation, is shown in Figures 9 and 10; some of the client code shown is part of intraprocedural constant propagation.

The main internal data structure used in interprocedural analysis is a table maintaining, for each call graph node, a set of input-output pairs representing the node's partial transfer function. The set is a singleton set for a context-insensitive analysis, while a context-sensitive analysis can have several pairs per node, one for each of the node's calling contexts. When interprocedural analysis completes, the final table is returned to the client. The client consults this table when transforming or compiling individual procedures; interprocedural analysis performs no transformations.

Bottom-up analyses have trivial input dataflow information (the single-point unit lattice), modeling the fact that bottom-up analyses ignore information flowing into the node from callers. Strictly top-down analyses have trivial output information. Interprocedurally flow-sensitive analyses maintain non-trivial information about each node's inputs and outputs.

The interprocedural algorithm recursively traverses the call graph, starting at the main node(s). When analyzing a node with input information (a calling context) that is not already found in the partial transfer function pairs, the context-sensitivity strategy function provided by the client is invoked. Given the set of contexts that have been previously analyzed and the new context to be analyzed, the function determines how the new context should be added to or merged in with the existing contexts, returning the set of previous contexts to throw away and the revised new context to use when analyzing the called procedure. The framework automatically reanalyzes callers that examined any contexts being thrown away.

Once the input context is determined by the context sensitivity function, the intraprocedural analysis function is invoked for that node. For the most part, this intraprocedural analysis runs like a regular intraprocedural analysis. However, when the intraprocedural analysis encounters a procedure call, it should return control to the interprocedural framework, to process the part of the call graph reachable from that call

```
-- Main entry point for interprocedural dataflow analysis
ip_traverse(cg:CallGraph,    -- the call graph to traverse
            -- initial inflowing info for each call graph node (used to start analysis of main node(s)):
            initial_input_info_fn :λ(n:CGNode):AnalysisInfo,
            -- initial outflowing info for each node (used to support recursion)
            initial_output_info_fn:λ(n:CGNode):AnalysisInfo,
            -- method for controlling how much calling context to maintain:
            context_sensitivity_fn:
              λ(prev_contexts:set[AnalysisInfo],  -- old contexts being maintained
                new_context:AnalysisInfo          -- new context being considered
                ):pair[set[AnalysisInfo],         -- old contexts to forget
                       AnalysisInfo],             -- new context to replace them with
            -- function to call to perform intraprocedural analysis on a call graph node:
            intra_fn:λ(n:CGNode,                  -- the node being analyzed
                       input_info:AnalysisInfo,   -- the inflowing info at that node
                       call_site_handler:         -- "call-back function" at call sites:
                          λ(site_id:CallSite,        -- the ID of the call site
                            input_info:AnalysisInfo  -- info entering call site
                            ):AnalysisInfo        -- returns info exiting call site
                       ):AnalysisInfo     -- the result of analyzing call graph node
   ):table[CGNode,           -- result of IP analysis:        for each node, a mapping from
           table[AnalysisInfo, AnalysisInfo]];  -- input info context to output info
```

Figure 7: Main `ip_traverse` Interface

```
context_insensitive_strategy(prev_contexts:set[AnalysisInfo],
                             new_context:AnalysisInfo
                             ):pair[set[AnalysisInfo],
                                    AnalysisInfo] {
   if prev_contexts.is_empty then
      return pair({}, new_context)  -- first time doing analysis; just analyze the new context
   else
      -- extract previous context analyzed (shouldn't have more than one):
      let prev_context := prev_contexts.only_element
      if prev_context.as_general_as(new_context) then
         -- previous context subsumes new context; reuse its result:
         return pair({}, prev_context)
      else
         -- merge old and new contexts, replace old with merged, and analyze merged:
         return pair(prev_contexts, new_context.merge(prev_context))
}
context_sensitive_strategy(prev_contexts:set[AnalysisInfo],
                           new_context:AnalysisInfo
                           ):pair[set[AnalysisInfo],
                                  AnalysisInfo] {
   -- always consider new context, never throw any old contexts away
   return pair({}, new_context)
}
new_bounded_context_sensitivity_strategy(bound:int):... {
   return λ(prev_contexts:set[AnalysisInfo],
            new_context:AnalysisInfo
            ):pair[set[AnalysisInfo],AnalysisInfo] {
                ... -- only maintain up to bound different contexts at one time
           }
}
```

Figure 8: Sample Context Sensitivity Strategies

```
-- Standard constant-propagation domain, for a single value
abstract class ConstInfo isa IterativeAnalysisInfo { ... };
   class TopConstInfo isa ConstInfo { ... };
   class IsAConstInfo isa ConstInfo { const_val:Constant; ... };
   class BottomConstInfo isa ConstInfo { ... };

-- Constant propagation domain for formals
class FormalsConstInfo isa IterativeAnalysisInfo {
   -- a mapping of the constantness of formals, by position:
   formals_info:vector[ConstInfo];
   ... -- other standard required functions of IterativeAnalysisInfos
};
new_bottom_formals(num_formals:int) {
   return new FormalsConstInfo(
      formals_info := new vector(num_formals, new BottomConstInfo)) }

-- Constant propagation domain for intraprocedural constant propagation, previously implemented:
class AvailableConstants isa IterativeAnalysisInfo {
   -- internal representation of available constants: a mapping from variables to constant info
   const_vars:table[VariableName,ConstInfo];
   ... -- other standard required functions
};
-- Subclass to support interprocedural extensions to constant propagation:
class IPAvailableConstants isa AvailableConstants {
   returned_constants:ConstInfo;   -- whether a constant is returned
   call_site_handler:λ(site_id:CallSite,   -- call-back to invoke at call sites
                       input_info:FormalsConstInfo):ConstInfo;
};
-- Initialize IPAvailableConstants info from inflowing FormalsConstInfo info
available_consts_from_formals(info:FormalsConstInfo):IPAvailableConstants;
-- Initialize FormalsConstInfo from call-site actuals and IPAvailableConstants info about them
new_formals_info_from_args(args:vector[VariableName],
                           info:IPAvailableConstants):FormalsConstInfo;
```

Figure 9: Implementation of Interprocedural Constant Propagation, Part 1

```
interprocedural_constant_propagation(cg:CallGraph
                        ):table[CGNode, table[FormalsConstInfo,ConstInfo]] {
   return
      ip_traverse(cg,
                  -- initialize all main nodes to have no constant formals:
                  λ(n:CFNode){ return new_bottom_formals(n.num_formals) },
                  -- in case of recursion, assume result of procedure is some unknown constant:
                  λ(n:CFNode){ return new TopConstInfo() },
                  -- select bounded context-sensitivity from standard palette:
                  new_bounded_context_sensitivity_strategy(3),
                  -- main "flow function" to analyze each node:
                  intraprocedural_constant_propagation)
}

intraprocedural_constant_propagation(
      n:CGNode, input_info:FormalsConstInfo,
      call_site_handler:λ(site_id:CallSite,
                          input_info:FormalsConstInfo):ConstInfo
      ):ConstInfo {
   let info := available_consts_from_formals(input_info, call_site_handler);
   traverse(n.control_flow_graph, Forward, Iterative, info,
            λ(stmt:CFGNode, info:AvailableConstants):AnalysisResult {
                return intra_const_prop(stmt, info)
            });
   return info.returned_constants
}

-- Selected intraprocedural flow functions for various kinds of statements:
-- Flow functions inherited from intraprocedural constant propagation unchanged:
intra_const_prop(stmt::Assign, info:AvailableConstants):AnalysisResult {
   -- bind result to the constant info representing the right hand side's value
   info.const_vars[stmt.lhs] := make_ConstInfo(stmt.rhs, info);
   return new ContinueResult(info) }
intra_const_prop(stmt::ALUOp, info:AvailableConstants):AnalysisResult {
   if stmt.is_pure() and stmt.args_are_constant(info) then
         -- constant-fold this operation, replacing with a simple assignment
         return new ReplaceResult(new Assign(stmt.lhs, stmt.Fold(info)))
   else
      -- update left-hand-side to be a non-constant and continue
      info.const_vars[stmt.lhs] := new BottomConstInfo();
      return new ContinueResult(info) }

-- Flow functions written specially for interprocedural constant propagation, overriding the intraprocedural
-- versions by dispatching on both the kind of statement and the kind of AnalysisInfo
intra_const_prop(stmt::Call, info::IPAvailableConstants):AnalysisResult {
   -- at a call, invoke the interprocedural call-back function to compute result constant
   let formals_info := new_formals_info_from_args(stmt.args, info);
   let result_info := (info.call_site_handler)(stmt.site_id, formals_info);
   info.const_vars[stmt.lhs] := result_info;
   return new ContinueResult(info) }
intra_const_prop(stmt::Return, info::IPAvailableConstants):AnalysisResult {
   -- remember the constant, if any, returned from this function
   info.returned_constants :=
      info.returned_constants.merge(info.const_vars[stmt.rhs]);
   return new ContinueResult(info) }
```

Figure 10: Implementation of Interprocedural Constant Propagation, Part 2

site and to compute the interprocedural output summary information about the call site to be used in the rest of the intraprocedural analysis. This control transfer is implemented through the `call_site_handler` call-back, given to the intraprocedural analysis at the start of analysis of the call graph node. The intraprocedural analysis invokes this call-back at each call site, passing the call site's identifier and input information and returning the output summary information for the callee(s) of that site back to the intraprocedural analysis. In this manner, the interprocedural and intraprocedural analyses alternate control during the interprocedural analysis of a program.

In the presence of recursion, a call graph node may be analyzed recursively before its first analysis has completed. To break the recursion in analysis, the first time a node is analyzed for a particular input context, an initial optimistic approximation to its output information is added to the set of input/output pairs. Any subsequent recursive calls with the same calling context will then immediately return the optimistic output information without recursive analysis. When the original analysis finally completes, the optimistic output information is compared to the real output information computed by analysis. If different, the framework replaces the overly optimistic output information with the real output information and reanalyzes any nodes whose analysis depended on the old output information. This strategy implements a standard iterative approximation algorithm to reach the greatest fixpoint solution, at the level of "loops" in the call graph.

Different algorithms for constructing call graphs embody different trade-offs between analysis time and precision of the resulting call graph. We wish our framework to be independent of the particular call graph construction algorithm, and the context-sensitivity strategy used by a particular client to be independent of the context-sensitivity strategy of the underlying call graph constructor or of any other interprocedural analysis client. All Vortex interprocedural clients run with our interprocedural analysis framework unchanged over a variety of call graph construction algorithms we have implemented.

One practical limitation to interprocedural analysis is coping with changes to the program source code. Reanalyzing the program interprocedurally from scratch and recompiling the entire program after a source code change would greatly limit the applicability of interprocedural analysis to standard interactive program development. We plan to extend Vortex to selectively reanalyze only those parts of the program affected by a program change and recompile only those procedures that depended on the changed interprocedural summary information. Vortex already includes support for selective recompilation for changes to simple kinds of interprocedural summary information [Chambers *et al.* 95].

5 Experience and Future Work

The framework described in this paper has been implemented in the Vortex compiler. We have used the intraprocedural analysis framework in essentially its current form since mid-1994. The composed analysis framework has been in use since mid-1995, although its interface was revised in early 1996 based on initial experience. The interprocedural analysis framework was designed and implemented in mid-1996.

Table 1 gives code sizes for the frameworks (excluding basic supporting data structures such as control flow graphs and call graphs) and for various analyses, in the order that they are executed during compilation. Some of the analyses are still more monolithically-integrated than we would like, such as the large constant propagation *et al.* pass, in part because these analyses were originally implemented before the composed analysis interface was implemented. We plan to reorganize these analyses into composed, modular components. Similarly, we expect to introduce new component analyses such as static and profile-guided

Table 1: Sizes of the Framework and Various Implemented Clients

Framework or Client		**Size (in lines of commented Cecil)**
Frameworks		
intraprocedural analysis framework		2,400
composed analysis framework		280
interprocedural analysis framework		1,200
Interprocedural Clients (excluding code shared with intraprocedural analyses, where applicable):		
may raise exception summary (bottom-up, flow-insensitive)		200
interprocedural closure escape analysis (bottom-up, flow-sensitive)		250
interprocedural constant propagation (context-sensitive, intra- and interprocedurally flow-sensitive)		300
Intraprocedural Clients (forward, iterative passes, unless specified otherwise):		
identifying last uses (backwards)		200
Composed Analysis:	class analysis	1,000
	compile-time message lookup, inlining, class & subclass testing	1,800
	splitting (eliminating partially redundant class & subclass tests)	900
	constant propagation & folding, copy propagation, common subexpression elimination, redundant load & store elimination, symbolic comparison elimination, closure escape analysis	4,500
dead store elimination (backwards)		250
dead assignment elimination (backwards)		200
closure creation delaying (eliminating partially unneeded closure creations)		400
environment creation delaying (eliminating partially unneeded environment creations)		375
clean-up (eliminating trivial branches and merges)		160
expansion I (lowering of high-level operations into mid-level ones) (non-iterative)		450
expansion II (lowering of mid-level operations into low-level ones) (non-iterative)		1,400
live variables & interference graph construction (backwards)		425
inserting register accesses and spill code (non-iterative)		900
simple instruction scheduling (non-iterative)		250

execution frequency analysis and compose these analysis with both the inlining transformation and the register interference graph construction algorithm to make their decisions more sensitive to execution frequency.

Some of the analyses listed in the table are run again between the first expansion pass and the second, such as constant propagation *et al.*, dead store elimination, dead assignment elimination, and graph clean-up. The constant propagation *et al.* pass is run both composed with other passes and on its own; it tests what other passes it is composed with as it runs in order to have the same code work in both contexts. The class analysis pass is reused by other parts of the compiler, such as several of the call graph construction algorithms.

Our intraprocedural analysis framework only supports easily optimizations that make local graph transformations (i.e., replace or delete the current statement). All of the existing analyses, save splitting, make only local transformations to the graph during analysis. Splitting may need to perform radical changes to the control flow graph, duplicating control paths to eliminate partially-redundant conditional tests [Chambers & Ungar 90]. To make these transformations safely in the face of composed iterative analyses, splitting waits until analysis has reached fixpoint before acting on any composed class analysis information, and when it is done transforming the graph it restarts analysis at a safe point before its graph transformations using a special `BackupAnalysisResult` outcome. It may be that other classical optimizations we have not implemented also require non-local graph transformations. We are currently implementing an intraprocedural loop invariant code motion optimization (and an interprocedural variation), which initially appears to need to make non-local graph transformations during iterative analysis. Fortunately, we have designed our analysis as first an iterative analysis to identify loop-invariant calculations, storing its results in a separate table, followed by a second non-iterative pass that inserts redundant calculations in loop preheaders. This loop-invariant code copying transformation is followed by the existing common subexpression elimination and copy propagation pass to eliminate or reduce the cost of the redundant calculations inside the loop. In this fashion, we are able to side-step the restriction against non-local graph transformations, albeit at some additional cost in client complexity.

We believe that we have achieved our main goal of making writing aggressive intra- and interprocedural optimizations easy. In particular, the intraprocedural framework manages and encapsulates many of the subtle details of performing graph transformations during iterative analysis. It took some time to debug this code, but clients can now rely on the framework to manage these details. Clients are no longer discouraged from performing sophisticated transformations during analysis because of implementation complexity.

It was not an explicit goal of our work to support the fastest possible analyses, although we do avoid repeating sequences of analyses and transformations until no more changes occur. Many of our analyses are quite time-consuming, but most of the time is spent in the client analysis code itself, not in the traversal framework. This occurs most often for analyses that maintain tables mapping variable names to information about those variables at each program point (e.g., the `AvailableConstants` class in Figure 9): the cost of searching large tables, copying them at control flow branches, and merging them at control flow merges gets high. We have considered several ways to speed up client analyses:

- **Denser bit-vector-based representations.** We tend to use relatively abstract, lattice-theoretic representations of dataflow information, which makes analyses easy to understand and extend but expensive to run. We modified some analyses to use faster bit-vector-based representations when the bit vector implements an appropriate higher-level abstraction such as a set. The set of live variables maintained by dead assignment elimination, shown in Figure 4, is one example of this. Compiler developers who are more concerned with compilation time could use compact representations more often than we do.

- **Copy-on-write tables.** Under this representation, the copy operation is implemented by making a new empty table that links back to the copied table. Modifications only happen to the head table, but searches may scan back through earlier tables, caching search results in the head table to speed later searches. Our preliminary experimentation with this approach has produced no clear results yet.
- **Sparser dataflow analysis model.** Our current analysis framework follows the procedure's control flow graph and analyzes all variables live at each program point in parallel, leading to large tables. An alternative approach follows the procedure's dataflow graph instead, propagating along each def/use arc a single piece of information describing that variable and avoiding the need to copy and update large tables [Choi *et al.* 91, Johnson & Pingali 93, Weise *et al.* 94]. This sort of analysis could work much better for analyses that track the flow of information along def/use arcs, such as constant propagation, but they may not be appropriate for analyses where information flows across "adjacent" instructions that are unrelated in a dataflow sense, such as common subexpression elimination or live variables analysis.

Another related area of future work is developing an incremental version of the interprocedural analysis framework, to support selective reanalysis and recompilation after incremental program changes during interactive program development, as discussed at the end of section 4.

6 Related Work

The basic theory for lattice-theoretic dataflow analysis was developed by Kildall [73] and Kam and Ullman [76]. Cousot and Cousot introduced abstract interpretation [77], which can be viewed as a more semantics-oriented approach to dataflow analysis, and widening operators to reach fixpoint in a finite amount of time even with infinitely-tall lattice domains. Knoop and Steffen discuss properties of an interprocedural version of monotone and distributive analysis frameworks [92]. Click and Cooper defined formally the circumstances in which two dataflow analyses should be run "in parallel" to reach better fixpoints than repeated sequences of the two analyses run separately [95]; this theory justifies our assertions that composed analyses can reach better fixpoints faster than iterated sequences of separate analyses. Masticola, Marlowe, and Ryder present the *k*-tuple dataflow analysis framework [95], a model for composing dataflow analyses that do not need to have a common single analysis direction.

A number of analysis frameworks have been described in the literature. Sharlit is an analyzer generator that supports iterative intraprocedural analyses [Tjiang & Hennessy 92]. Unlike our framework, it does not support composing analyses, running transformations "undoably" as part of analysis, or interprocedural analyses. It does have interesting support for "path compression" which enables faster analyses that compute summary flow functions for basic blocks or larger structures (in the style of interval analysis [Cocke 70]), for problems where flow functions are represented as GEN and KILL bit-vectors. The `cmcc` compiler [Adl-Tabatbai *et al.* 96], like our intraprocedural analysis framework, uses object-oriented design to organize the code and provide an abstract class for analysis information. However, it does not support composing analyses and/or transformations nor aid in building interprocedural analyses. Dwyer and Clarke describe another intraprocedural lattice-theoretic analysis architecture [Dwyer & Clarke 96]. They provide more parameterization options than our framework, enabling different program representation modules and analysis solvers to be substituted, although not all combinations of analysis, program representation, and solver make sense. Their architecture also supports a restricted kind of composed analysis. They do not address integrating transformations with analysis.

FIAT is a framework for managing interprocedural analyses [Hall *et al.* 93]. Like our interprocedural framework, it relies on underlying intraprocedural analysis to analyze individual procedures. Their framework includes a particular call graph construction algorithm, although it appears that alternative algorithms could be used. Unlike our framework, the FIAT system does not directly provide support for context-sensitive analyses, although clients can implement them by hand as evidenced by their procedure cloning algorithm [Cooper *et al.* 92]. FIAT recomputes individual interprocedural analyses on demand if their information is needed by some other analysis but is out of date. FIAT's compiler-independent representation for procedure-level information simplifies porting to other compilers and enables it to represent an analysis-specific abstraction of the procedure to save memory and perhaps analysis time, at the cost of extra representation-translation time and complexity for any particular compiler.

The McCAT optimizing C compiler includes a framework for context-sensitive interprocedural analyses [Hendren *et al.* 93]. This framework does not manage dataflow information or help in reaching fixpoints, but only provides the basic program representation and the program call graph over which clients write their analyses manually. McCAT includes a particular strategy for constructing a context-sensitive call graph, where each non-recursive path through the program call graph leads to a distinct calling context (producing a potentially exponential number of contexts) and function pointers are analyzed using an alias analysis that considers only stack locations, not heap locations. In contrast, our framework provides much more support for writing analyses beyond the base program representation, our interprocedural analysis framework is independent of the particular call graph construction algorithm, and our framework allows the client interprocedural analyses to specify their own policy for context-sensitivity.

PAG is a generator for both intra- and interprocedural dataflow analyzers [Alt & Martin 95]. PAG defines several specialized but restricted languages in which the syntax-tree program representation, the domain data structures, the flow functions, and the high-level analysis engine are defined. PAG's intraprocedural analysis is fairly conventional, lacking support for composing analyses and transformations and lacking Sharlit's path compression. Its interprocedural framework uses a model of context-sensitivity where each procedure has N entries of context, but there does not appear to be any support for dynamically computing the appropriate N for each procedure and analysis. Instead, they treat each non-recursive path through the program call graph, after each recursion has been unrolled by some factor k, as a distinct context, leading to even more contexts than produced by the McCAT system. In contrast, our framework allows the client analysis to conveniently specify policies for dynamically determining the appropriate context-sensitivity for each procedure independently.

System Z generates intra- and interprocedural analyzers from abstract interpreters specified in a special language [Yi & Harrison 93]. This system includes projection expressions which enable analyses to conveniently express more approximate analyses without directly modifying the underlying abstract interpreter. System Z does not appear to provide support for context-sensitive analyses, nor does it address combining abstract interpreters capturing different aspects of program behavior. It also suffers from the inefficiency of maintaining a persistent table mapping each program point (not just procedure entries as our framework does) to information deduced about that point.

Assmann has developed a model for integrated analyses and transformations where everything ranging from the program representation to the dataflow analysis information is represented by a particular graph model [Assmann 96]. Analyses and transformations are expressed as graph rewrite rules, which are "fired" whenever their preconditions over the input graph structure are satisfied, leading to more of a constraint-based flavor of program analysis. There is no support, however, for "tentatively" performing graph transformations during iterative analysis of loops; the graph rewrites are permanent, and the rule preconditions must encode whether the final result of analysis enables a transformation, thereby making it difficult to use the outcome of transformations to improve analysis. Multiple non-transforming analyses can easily be composed, simply by combining their rules into one larger set of rules, enabling the analyses to examine and act on each other's intermediate results.

Many researchers have explored issues in program representation and speeding program analysis [Ferrante *et al.* 87, Cytron *et al.* 89, Choi *et al.* 91, Dhamdhere *et al.* 92, Johnson & Pingali 93, Weise *et al.* 94, Aiken & Wimmers 92, Heintz 94, Jagannathan & Weeks 95, Sreedhar *et al.* 96]. Our work builds on a conventional control flow graph base, but it would be interesting to try to develop similar interfaces for client analyses that exploit the more advanced representations to run faster.

7 Conclusions

We have designed and implemented a framework that makes writing intra- and interprocedural dataflow analyses easier, and we have used this framework to develop all of the analyses currently in the Vortex optimizing compiler. Our framework improves on previous work principally in its mechanisms to support running multiple analyses in parallel and performing transformations as part of analysis, enabling a more modular approach to writing analyses and transformations while still reaching the most precise solutions, and in its support for constructing interprocedural analyses with customizable context sensitivity strategies. These new features are particularly useful in optimizations of dynamic dispatching and first-class functions, where multiple analyses need to be integrated and transformations like inlining need to be performed during iterative analysis. We plan to extend our framework to support selective interprocedural reanalysis and recompilation after programming changes, and we wish to investigate whether related frameworks can be developed that exploit more sophisticated program representations and analysis models for faster analysis. We hope that our work encourages other compiler writers to develop similar frameworks for their compilers.

Acknowledgments

As part of their graduate compilers class project, Adam Carlson, David Dion, Sujay Parekh, and Greg Linden implemented an early version of the interprocedural dataflow analysis framework. Tina Wong and MaryAnn Joy implemented parts of Vortex's constant propagation *et al.* pass. Greg DeFouw implemented one of Vortex's interprocedural call graph construction algorithms. Vassily Litvinov implemented other parts of the Vortex compiler system. Michael Ernst, Gary Leavens, and Vassily Litvinov provided extensive help in improving the paper's presentation.

This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), an NSF Research Initiation Award (number CCR-9210990), a grant from the Office of Naval Research (contract number N00014-94-1-1136), an Intel Foundation Graduate Fellowship, and gifts from Sun Microsystems, IBM, Xerox PARC, Pure Software, and Edison Design Group.

References

[Aho et al. 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Aiken & Wimmers 92] A. Aiken and E. Wimmers. Solving Systems of Set Constraints. In *Proceedings of the 7'th IEEE Symposium on Logic in Computer Science*, pages 329--340, Santa Cruz, CA, June 1992.

[Alt & Martin 95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Second INternational Symposium on Static Analysis*, pages 33--50. Springer-Verlag, September 1995.

[Assmann 96] Uwe Assmann. How to Uniformly Specify Program Analysis and Transformations with Graph Rewrite Systems. In *Proceedings of the CC'96. 6'th International Conference on Compiler Construction*, pages 121--135. Springer-Verlag, April 1996.

[Adl-Tabatbai et al. 96] Ali-Reza Adl-Tabatbai, Thomas Gross, and Guei-Yuan Lueh. Code Reuse in an Optimizing Compiler. In *OOPSLA'96 Conference Proceedings*, pages 51--68, San Jose, CA, October 1996.

[Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. *SIGPLAN Notices*, 25(6):150--164, June 1990. In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[Chambers et al. 95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *17th International Conference on Software Engineering*, pages 221--230, Seattle, WA, April 1995.

[Choi et al. 91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55--66, Orlando, Florida, January 1991.

[Click & Cooper 95] Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181--196, March 1995.

[Cocke 70] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20--24, July 1970.

[Cooper et al. 92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In *Proceedings of 1992 IEEE International Conference on Computer Languages*, pages 96--105, Oakland, CA, April 1992.

[Cousot & Cousot 77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238--252, Los Angeles, California, January 1977.

[Cytron et al. 89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25--35, Austin, Texas, January 1989.

[Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Dhamdhere et al. 92] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to Analyze Large Programs Efficiently and Informatively. *SIGPLAN Notices*, 27(7):212--223, July 1992. In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[Dwyer & Clarke 98] Matthew B. Dwyer and Lori A. Clarke. A Flexible Architecture for Building Data Flow Analyzers. In *17th International Conference on Software Engineering*, pages 554--564, Berlin, Germany, March 1998.

[Ferrante et al. 87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319--349, July 1987.

[Hall et al. 93] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In *The Sixth Anunual Workshop on Parallel Languages and Compilers*, August 1993.

[Heintze 94] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 306--317, Orlando, FL, June 1994.

[Hendren et al. 93] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A Practical Context-Sensitive Interprocedural Analysis Framework for C Compilers. Technical Report ACAPS Technical Memo 72, McGill University School of COmputer Science, July 1993.

[Jagannathan & Weeks 95] Suresh Jagannathan and Stephen Weeks. A Unified Framework of Flow Analysis in HIgher-Order Languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393--407, January 1995.

[Johnson & Pingali 93] Richard Johnson and Keshav Pingali. Dependence-Based Program Analysis. *SIGPLAN Notices*, 28(6):78--89, June 1993. In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.

[Kam & Ullman 76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158--171, January 1976.

[Kildall 73] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194--206, Boston, Massachusetts, October 1973.

[Knoop & Steffen 92] Jens Knoop and Bernhard Steffen. The Interprocedural Coincidence Theorem. In *Proceedings of the CC'92. 4'th International Conference on Compiler Construction*, pages 125--140. Springer-Verlag, October 1992.

[Masticola et al. 95] Stephen Masticola, Thomas J. Marlowe, and Barbara G. Ryder. Lattice Frameworks for Multisource and Bidirectional Data Flow Problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777--803, September 1995.

[Sreedhar et al. 96] Vugranam C. Sreedhar, Guang R. Gao, and Yong fong Lee. A New Framework for Exhaustive and Incremental Data Flow Analysis Using DJ Graphs. *SIGPLAN Notices*, pages 278--290, May 1996. In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.

[Tjiang & Hennessy 92] Steven W. K. Tjiang and John L. Hennessy. Sharlit -- A Tool for Building Optimizers. *SIGPLAN Notices*, 27(7):82--93, July 1992. In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[Weise et al. 94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation without Taxation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297--310, Portland, Oregon, January 1994.

[Yi & Harrison 93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic Generation and Management of Interprocedural Program Analyses. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246--259, Charleston, South Carolina, January 1993.