UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

A Tool Kit for Constructing Optimal Interprocedural Data Flow Analyses

Jens Knoop, Bernhard Steffen
Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau

Oliver Rüthing*
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preußerstraße 1-9
D–24105 Kiel

MIP-9413
November 1994

* Part of the work was done while the author was supported by the Deutsche Forschungsgemeinschaft grant La 426/11-1.

Abstract

We present a tool kit for the construction of optimal interprocedural data flow analyses for programs with mutually recursive procedures, global and local variables, formal value and reference parameters. The tool kit contains *specification tools*, which allow concise specifications, and *computation tools* that can directly be fed with these specifications. Moreover, proving an analysis to be optimal only requires knowledge about the specification. The benefits of the tool kit are demonstrated by presenting a collection of practically relevant optimal interprocedural data flow analyses ranging from classical bit-vector problems as e.g. *reaching definitions* and *very busy expressions* to more sophisticated applications as the optimal *elimination of interprocedural partial redundancies*.

Keywords

Interprocedural data flow analysis, program optimization, bit-vector data flow analyses, available expressions, reaching definitions, live variables, very busy expressions, use-definition-chains, code motion, partial redundancy elimination, partial dead code elimination, strength reduction.

Contents

1 Motivation

Data flow analysis (DFA) is concerned with the static analysis of programs in order to support the generation of efficient object code by "optimizing" compilers. Typically, designers of a DFA are faced with the problem of how to construct an algorithm that *determines* the set of all program points of an argument program that satisfy a given property of interest. Although this problem has been studied in detail for the *intraprocedural* case, the construction of *interprocedural* analyses is still characterized by heuristics and ad-hoc techniques.

The point of this paper is to support the designer of a DFA by providing him with a 'tool kit' that structures and simplifies the construction of provably correct and optimal *interprocedural* DFA-algorithms. This tool kit evolves as a distillation of the stack-based framework of [Kn, KS1] for interprocedural DFA, which is unique in capturing programs with mutually recursive procedures, global and local variables, formal value and reference parameters. It consists of *specification tools* allowing concise specifications of interprocedural DFAs, and *computation tools* determining the (optimal) data flow information directly from and according to a given specification.

In fact, using the tool kit the construction of interprocedural DFA-algorithms reduces, essentially as in the intraprocedural case, to specifying the data flow information of interest and the way in which this information is modified by means of elementary statements. The only difference to the intraprocedural case is the necessity of specifying the effect of return nodes in order to deal with local variables and formal parameters of (recursive) procedures.

Once the DFA has been specified, the corresponding DFA-algorithm can automatically be derived by means of the generic computations tools. This does not require any further knowledge about the underlying interprocedural framework or the specifics of the computation tools. The same also applies to the proofs of optimality. Based on our tool kit these proofs are split into four usually rather straightforward substeps that concern the domain of the specification only.

The practicality of the tool kit is illustrated by presenting a collection of optimal interprocedural DFA-algorithms solving well-known DFA-problems. It should be noted that these interprocedural algorithms are as efficient as their intraprocedural counterparts.

Related Work

There is a great variety of papers dealing with specific problems of interprocedural DFA, as e.g., *constant propagation* [CCKT, GT], *partial redundancy elimination* [Mo, MR], *interprocedural slicing* [HRB], *alias information* [Ban, Co, CK2, CK3, ERH, LH, My, We], or information whether a particular variable is *used*, *modified* or *preserved* by a procedure [Bth, CK1, HS, Ro].

On the other hand, unifying approaches aiming at a framework for interprocedural DFA have been considered e.g., by Cousot/Cousot [CC2], Barth [Ba1, Ba2], Sharir/Pnueli [SP], Jones and Muchnick [JM], and more recently by Bourdoncle [Bo]. These approaches mainly address correctness of a DFA, and they do not properly deal with local variables of recursive procedures: proper treatment of recursive procedures would require a mechanism to store information about local variables when treating a recursive call. Moreover, they have a foundational character: the use of the techniques would require a detailed understanding of the underlying frameworks. In contrast, our approach is 1) general: it is not restricted to certain problem classes, 2) it is optimal even in the presence of recursive procedures, and 3) it is tailored for practical use: all the details that are irrelevant for a specific application are hidden.

This approach is based on the stack-based framework for interprocedural DFA of [KS1]. The point of this framework was the introduction of so-called return functions, which allowed to distinguish between effects on global and local variables after returning from a procedure call. This turned out to be the key for constructing an algorithm for computing the intuitively desired solution of an interprocedural DFA-problem, that for the first time handled local variables properly. Later on this framework has been generalized to reference and procedure parameters [Kn].

The stack-based framework providing the foundation for this paper, can also be regarded as the theoretical backbone for proving the correctness of the efficiency-oriented interprocedural DFA-algorithms that have recently been introduced by Reps, Horwitz, and Sagiv, and Duesterwald, Gupta, and Soffa.

In [RHS] Reps, Horwitz, and Sagiv propose an algorithm for solving interprocedural DFA-problems over finite lattices in a way capturing global and local variables, and value and reference parameters. The treatment of local variables was made explicit in [Re3]. Two separate functions are used to mimic the return functions introduced in [KS1]: One function for extracting the globally relevant part from the data flow information valid at the termination time of the called procedure, and one function extracting from the data flow information valid immediately before the call, the data flow information about local variables that must be re-established after the return from the call. This *implicit* treatment of return functions allows them (1) to reduce an interesting class of DFA-problems to graph reachability problems that can efficiently be solved, and (2) to compute the effect of procedure calls in a by-need fashion. In fact, this enables the treatment of some potentially infinite abstract domains [SRH].

However, the introduction of the two 'artifical' functions introduces some additional paths in the graph used to represent the program that do not correspond to a standard program-execution path. As a consequence, when one encodes a problem in their framework(s), the proof of correctness for the encoding must account for these paths. This proof can be simplified by means of the Interprocedural Coincidence Theorem of [KS1]: it is sufficient to prove the equivalence to the *IMFP*-solution. This does not require the consideration of stacks at all, which must be part of any sound *IMOP*-solution capturing local variables and value parameters.

In [DGS] Duesterwald, Gupta, and Soffa propose an algorithm for demand-driven interprocedural DFA working for programs with global and local variables, value and reference parameters. The point of their algorithm is to compute data flow information for a given program point without performing an exhaustive analysis of the argument program. In practice, this may be used e.g. for the debugging of a program during its development. The effect of the return functions of [KS1] to handle local variables can problem specifically be encoded in the binding functions, which realize the mapping between the address space of the calling procedure and the called procedure. Like in [RHS] a formal correctness proof for this approach depends on the correspondence between the computed solution and the operational understanding of a procedural program, and can be established by means of the results of [Kn, KS1]. Another algorithm for demand-driven DFA, which is based on magic-set transformations, has recently been proposed by Reps [Re1, Re2].

Structure of the Paper

In Section 2 we revisit the standard framework of intraprocedural DFA, advocating that it can directly be thought of as a tool kit for constructing optimal intraprocedural DFA-algorithms. In Section 3 this is generalized to the stack-based framework for interprocedural DFA of [Kn, KS1]. This results in a tool kit for constructing optimal interprocedural DFA-algorithms. Section 4 illustrates the handling of reference parameters. The benefits of the tool kit are subsequently illustrated in Section 5 by presenting a collection of practically relevant interprocedural data flow analyses. Finally, Section 6 contains our conclusions and Appendix A the generic algorithms representing the computation tools.

2 Revisiting Intraprocedural DFA in a Tool Kit View

Intraprocedural DFA is characterized by a separate and independent investigation of the procedures of a program. Here it is common to represent procedures as *directed flow graphs* $G=(N,E,\mathbf{s},\mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent the statements and edges $(n,m) \in E$ the nondeterministic branching structure of the corresponding procedure. $pred_G(n)=_{df} \{\, m \mid (m,n) \in E \,\}$ and $succ_G(n)=_{df} \{\, m \mid (n,m) \in E \,\}$ denote the set of all immediate predecessors and successors of a node n, respectively. $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G, which are assumed to possess no predecessors and successors, respectively. A *finite path* in G is a sequence $(n_1,\ldots,n_q)$ of nodes such that $(n_j,n_{j+1}) \in E$ for $j \in \{1,\ldots,q-1\}$. $\mathbf{P}_G[m,n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}_G[m,n)$ the set of all finite paths from m to a predecessor of n. Moreover, ε denotes the unique path of length 0. Finally, all nodes $n \in N$ are assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

2.1 Optimal DFA-Algorithms

Given a *program property* ϕ of interest, data flow analysis aims at determining the set of all program points enjoying this property. A typical example, which serves as a running example throughout Section 2, is the availability of a program term t. Intuitively, a term t is *available* at a program point n, if on every program path reaching node n the last modification of an operand of t is followed by a computation of t. The formal definition requires the predicates $Comp(n)$ indicating that t is *computed* in n (i.e., t has a right hand side occurrence in n), and $Transp(n)$ indicating that t is *transparent* in n (i.e., no variable occurring in t is modified by the assignment of node n).

Definition 2.1 (Availability)
A term t is available *at a node $n \in N$ [in signs: $Avail(n)$]* $\Longleftrightarrow_{df}$
$\forall\, (n_1,\ldots,n_k) \in \mathbf{P}_G[\mathbf{s},n).\ \exists\, 1 \le i \le k.\ Comp(n_i) \ \wedge\ \forall\, i \le j \le k.\ Transp(n_j).$

Optimality of a DFA-algorithm is then defined by

Definition 2.2 (Optimality of a DFA-Algorithm)
Let ϕ be a program property, and $\mathcal{A}$ be a DFA-algorithm. $\mathcal{A}$ is optimal *with respect to ϕ if and only if $\mathcal{A}$ determines for every flow graph G the set of all nodes enjoying property ϕ.*

Technically, a DFA-algorithm computes an annotation of the argument flow graph with *data flow information* that characterizes the program property of interest. In terms of our running example, an optimal DFA-algorithm would typically compute for every node n a Boolean value indicating, whether t is available at n or not.

2.2 The Intraprocedural Tool Kit

The intraprocedural tool kit contains *specification tools* for a concise and elegant specification of data flow analyses, and a *computation tool* computing the corresponding data flow information.

The Intraprocedural Tool Kit	
Specification Tools:	1) *Complete Semi-Lattices* $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$
	2) *Local Semantic Functionals* $[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$
Computation Tool:	3) *Generic Fixed Point Algorithm A.1*

Remark: *The tool kit is formulated for forward analyses. As usual, backward analyses can be treated by forward analyses in the tool kit simply by inverting the flow of control (cf. [He]). Also bidirectional analyses can be handled, but require different techniques to prove their optimality, because they lack a natural operational characterization. The best way to overcome this problem is to decompose bidirectional analyses into sequences of unidirectional ones. This is exemplified in Section 5.5: the originally bidirectional flow of data flow information (cf. [Mo, MR]) is structured in a backward analysis followed by a forward analysis. In fact, we do not know of any (practical) DFA that cannot be decomposed in this way.*

Intuitively, the elements of the lattice $\mathcal{C}$ express the data flow information of interest, and the local semantic functional $[\![\ ]\!]$ gives meaning to every node of a flow graph G in terms of a transformation on $\mathcal{C}$. In fact, the tool kit view only requires to specify $\mathcal{C}$, a local semantic functional $[\![\ ]\!]$, and a *start information* $c_0 \in \mathcal{C}$, which is assumed to represent the data flow information, which is valid on entering the procedure under consideration.

Specifying an Intraprocedural DFA-Algorithm	
Specify	1) a complete semi-lattice $\mathcal{C}$
	2) a local semantic functional $[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$
	3) a start information $c_o \in \mathcal{C}$

The specifying components of a data flow analysis can directly be fed into the generic Algorithm A.1, which annotates the argument flow graph with the corresponding data flow information. Algorithm A.1 deals with the case, where the required information is defined as the *greatest* solution of Equation System 2.3. *Least* solutions are dealt with in a dual fashion.

Equation System 2.3 (The Equation System of the *MFP*-Strategy (Alg. A.1))

$$\mathbf{dfi}(n) \quad = \quad \begin{cases} c_0 & \text{if } n = \mathbf{s} \\ \bigsqcap \{ [\![\, m \,]\!](\mathbf{dfi}(m)) \,|\, m \in pred_G(n) \} & \text{otherwise} \end{cases}$$

The greatest solution of Equation System 2.3, denoted by $\mathbf{dfi}_{c_0}$, is well-known as the solution of the *maximal fixed point* strategy in the sense of Kam and Ullman [KU].

$$\textbf{The } \textit{MFP}\textbf{-Solution:} \quad \forall \, n \in N \; \forall \, c_0 \in \mathcal{C}. \;\; MFP_{([\![\;]\!], c_0)}(n) \; = \; \mathbf{dfi}_{c_0}(n)$$

We present the generic fixed point Algorithm A.1 only in the Appendix of this paper, since once it has been implemented, the designer of the DFA-algorithm does not need to know any details about this algorithm. It is important to note that this carries over to the interprocedural tool kit as well.

The table below shows the specification of the DFA-algorithm for availability.

Example: Availability	
1) *Complete Semi-Lattice:*	The lattice of Boolean truth values with *false* < *true* $\mathcal{B} = (\{false, true\}, \wedge, \leq, false, true)$
2) *Local Semantic Functional:*	$[\![\;]\!]_{av} : N \rightarrow (\mathcal{B} \rightarrow \mathcal{B})$ defined by $\forall \, n \in N \; \forall \, b \in \mathcal{B}. \; [\![\, n \,]\!]_{av}(b) =_{df} Transp\,(n) \wedge (Comp\,(n) \vee b)$
3) *Start Information:*	$false \in \mathcal{B}$

Choosing the lattice of Boolean truth values reflects the fact that a term is either available at a particular program point or not; the definition of the local semantic functional is based on the intuition that a term is available at the exit of a node n, if it is *transparent* in n (i.e., it is not modified in n), and *computed* in n or available at the entry of n; finally, the choice of the start information reflects that we do not assume any information about the environment of the flow graph being analyzed.

2.3 Proof Obligations

First, it is important that a local semantic functional can easily be extended to cover finite paths as well by defining for every path $p = (n_1, \ldots, n_q) \in \mathbf{P}_G[m, n]$

$$[\![\, p \,]\!] =_{df} \begin{cases} id_{\mathcal{C}} & \text{if } p \equiv \varepsilon \\ [\![\, (n_2, \ldots, n_q) \,]\!] \circ [\![\, n_1 \,]\!] & \text{otherwise} \end{cases}$$

where $id_{\mathcal{C}}$ denotes the identity on $\mathcal{C}$. This extension is the key to the *meet over all paths* (*MOP*) strategy in the sense of Kam and Ullman [KU], which provides a 'run-time oriented' definition of a flow graph annotation with data flow information. The solution of this strategy, which "meets" all informations belonging to a program path reaching the program point under consideration, is defined by:

The *MOP***-Solution:** $\forall\, n \in N \;\forall\, c_0 \in \mathcal{C}.\; MOP_{([\![\ ]\!], c_0)}(n) = \bigsqcap \{\, [\![\, p \,]\!](c_0) \mid p \in \mathbf{P}_G[\mathbf{s}, n) \,\}$

Although the *MOP*-solution is in general not directly computable, because there may be infinitely many program paths reaching a particular program point, it is conceptually important, since it allows to decompose the *completeness* proof for a DFA-algorithm, establishing the optimality up to termination, into two steps that are in general considerably easier to prove: First, the proof of the coincidence of the (computed) maximal fixed point solution with the meet over all paths solution. Second, the proof of the equivalence of the meet over all paths solution and the program property of interest.

The third step, the *correctness* proof, guaranteeing termination of the computation tool with the maximal fixed point solution, completes the optimality proof of the DFA-algorithm under consideration.

Completeness

Step 1: The key for proving the coincidence of the *MFP*-solution and the *MOP*-solution is the well-known Coincidence Theorem of Kam and Ullman [KU], which gives a sufficient condition for the coincidence of the meet over all paths solution and the maximal fixed point solution.

Theorem 2.4 (Coincidence Theorem)
Given a flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, the MFP-solution and the MOP-solution coincide, i.e., $\forall\, n \in N \;\forall\, c_0 \in \mathcal{C}.\; MOP_{([\![\ ]\!], c_0)}(n) = MFP_{([\![\ ]\!], c_0)}(n)$, if the semantic functions $[\![\, n \,]\!]$, $n \in N$, are distributive.[1]

Step 2: This step closes the gap between the meet over all paths solution and the program property of interest. In general, this gap is considerably small, since both notions are defined in terms of individual contributions of program paths over the same local properties of nodes (cf. running example). Thus, this gap is usually bridged by a straightforward induction on the length of program paths.

[1] $[\![\, n \,]\!]$ is called *distributive* if and only if $\forall C \subseteq \mathcal{C}.\; [\![\, n \,]\!](\bigsqcap C) = \bigsqcap \{[\![\, n \,]\!](c) \mid c \in C\}$.

Correctness

Step 3: The remaining proof that the generic Algorithm A.1 terminates with the maximal fixed point solution is a consequence of the following lemma:

Lemma 2.5 (Correctness Lemma)
Given a flow graph G, a complete semi-lattice $\mathcal{C}$, and a local semantic functional $[\![\ ]\!]$, the generic Algorithm A.1 terminates with the MFP-solution, if $\mathcal{C}$ satisfies the descending chain condition,[2] *and the semantic functions $[\![\, n \,]\!]$, $n \in N$, are monotonic.*[3]

Since the first step requires distributivity, which implies monotonicity, the only new requirement is the descending chain condition of $\mathcal{C}$.

Summary

Proving Optimality of an Intraprocedural DFA-Algorithm	
Prove that	1) the lattice $\mathcal{C}$ satisfies the *descending chain condition* 2) the local semantic functions, $[\![\, n \,]\!]$, $n \in N$, are *distributive* 3) the *MOP*-solution and the program property ϕ of interest are *equivalent*
Obtain that	the specified intraprocedural DFA is *optimal* in the sense of Definition 2.2

Note that the optimality proof does not require any knowledge about the generic fixed point algorithm.

It is well-known that the monotonicity of the semantic functions is already sufficient to guarantee that the *MFP*-solution is a *safe* approximation of the *MOP*-solution (cf. [KU]), i.e.,

$$\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ \mathit{MFP}_{([\![\ ]\!],c_0)}(n) \sqsubseteq \mathit{MOP}_{([\![\ ]\!],c_0)}(n)$$

Thus our tool kit can easily be modified to capture safety only.

Figure 1 illustrates the three proof obligations for establishing optimality of a DFA-algorithm in their context. In fact, all these steps can easily be verified for our running example. In particular, both implications of the third proof obligation, $\mathit{MOP}_{([\![\ ]\!]_{av},\mathit{false})}(n) \iff \mathit{Avail}(n)$, $n \in N$, follow by simple induction.

[2] A semi-lattice C satisfies the *descending chain condition*, if for all sequences of elements of C with $c_1 \sqsupseteq c_2 \sqsupseteq \ldots c_i \sqsupseteq \ldots$, there is an index j satisfying $c_k = c_j$ for all $k \geq j$.

[3] A function $f : C \to D$, C and D complete semi-lattices, is called *monotonic* if and only if $\forall c, c' \in C.\ c \sqsubseteq c'$ implies $f(c) \sqsubseteq f(c')$.

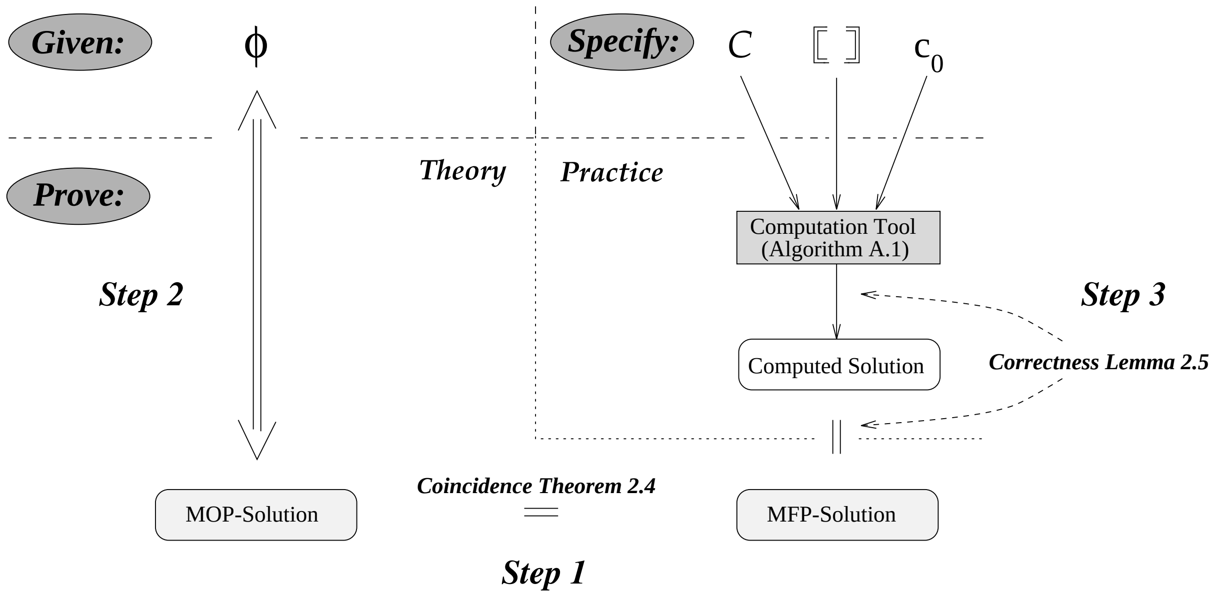

Figure 1: Illustrating the Intraprocedural Situation

3 The Interprocedural Tool Kit

The tool kit for constructing optimal interprocedural DFA-algorithms is distilled from the stack-based framework for interprocedural DFA of [Kn, KS1]. Here we recall the parts of this framework, which are essential for the application of the tool kit.

In the interprocedural setting we consider programs Π as systems $(\pi_0, \pi_1, \ldots, \pi_k)$ of (mutually recursive) procedure definitions, where every $\pi \in \Pi$ has a list of formal value and formal reference parameters, and a list of local variables. π_0 is assumed to denote the *main procedure* and therefore cannot be called. π_1 up to π_k are the *procedure declarations* of Π. For simplicity we assume that there is no (static) procedure nesting except that π_0 encloses π_1 up to π_k.[4] Thus, the variables of the main program are global variables of the procedures, and can be accessed by them. Programs Π are represented as systems $S = (G_0, G_1, \ldots, G_k)$ of *flow graphs* with disjoint sets of nodes N_i and edges E_i, in which every procedure π of Π (including the main procedure π_0) is represented as a *directed flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$ in the sense of Section 2. $\mathbf{N}^S =_{df} \bigcup \{ N_i \mid i \in \{0, \ldots, k\}\}$ denotes the set of all nodes of S, and $\mathbf{N}^S_C \subseteq \mathbf{N}^S$ the set of all nodes representing procedure calls. Additionally, we define the set of nodes $N' =_{df} (\mathbf{N}^S \backslash \mathbf{N}^S_C) \cup \{ n_C, n_R \mid n \in \mathbf{N}^S_C \}$, where every procedure call node n of S is replaced by a *call node* n_C and a *return node* n_R, which are introduced for modelling the parameter transfer and the initialization of local variables on entering and leaving a procedure called by $n \in \mathbf{N}^S_C$: Let n represent the procedure call statement *call* $\pi(a_1, ..., a_q, t_1, ..., t_r)$, where π has formal reference parameters $f_1, ..., f_q$, formal value parameters $g_1, ..., g_r$, and local variables $v_1, ..., v_s$, then n_C and n_R represent the assignments $(f_1, ..., f_q, g_1, ..., g_r, v_1, ..., v_s) := (a_1, ..., a_q, t_1, ..., t_r, \Lambda, ..., \Lambda)$ and $(f_1, ..., f_q, g_1, ..., g_r, v_1, ..., v_s) := (\Lambda, ..., \Lambda)$, respectively, where Λ denotes a special value "undefined".

Additionally, we define $N'_C =_{df} \{ n_C \mid n \in \mathbf{N}^S_C \}$, and $N'_R =_{df} \{ n_R \mid n \in \mathbf{N}^S_C \}$. Finally, we

[4] Integrating static procedure nesting is straightforward (cf. [Kn]).

introduce the function $fg : \mathbf{N}^S \rightarrow S$, which maps every node of S to the flow graph, in which it is contained, the function $callee : \mathbf{N}_C^S \rightarrow S$, which maps every procedure call node to the flow graph representing the called procedure, the function $caller : S \rightarrow \mathcal{P}(\mathbf{N}_C^S)$, which maps every flow graph to the set of nodes calling it, and the functions $start : S \rightarrow \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ and $end : S \rightarrow \{\mathbf{e}_0, \ldots, \mathbf{e}_k\}$, which map every flow graph of S to its start node and end node, respectively.

3.1 Optimal Interprocedural DFA-Algorithms

Clearly, the notion of optimality defined for intraprocedural DFA-algorithms carries over to the interprocedural setting: An interprocedural DFA-algorithm is *optimal* with respect to a program property ϕ if and only if it determines the set of nodes of an argument flow graph system enjoying property ϕ.

3.2 The Interprocedural Tool Kit

As its intraprocedural counterpart, the interprocedural tool kit offers *specification tools* and *computation tools*.

The Interprocedural Tool Kit	
Specification Tools:	1) *Complete Semi-Lattices* $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$ 2) *Local Semantic Functionals* $[\![\ ]\!]' : N' \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ 3) *Return Functionals* $\mathcal{R} : \mathbf{N}_C^S \rightarrow (\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C})$
Computation Tools:	4) *Generic Fixed Point Algorithm A.2* 5) *Generic Fixed Point Algorithm A.1*

The specification tools 1) and 2) and the computation tool 5) are already contained in the intraprocedural tool kit except that local semantic functionals are extended here to deal with parameter transfers. This, however, does not introduce any new subtleties, since parameter transfers can be handled in our tool kit in terms of ordinary assignment statements (cf. Section 3). New are the *return functionals*, and the generic fixed point Algorithm A.2, which are described below.

Return functionals give meaning to procedure call nodes in terms of a function from $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$. Intuitively, the first argument of a return function $\mathcal{R}(n)$ represents the data flow information that is valid immediately *before* a procedure call, and the second argument the data flow information that is valid immediately *before* finishing it. The return function extracts from these arguments the data flow information, which is valid immediately *after* finishing the procedure call. This is necessary in order to deal with the difference between local and global variables.

Specifying an Interprocedural DFA-Algorithm	
Specify	1) a complete semi-lattice $\mathcal{C}$ 2) a local semantic functional $[\![\ ]\!]' : N' \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ 3) a return functional $\mathcal{R} : \mathbf{N}_C^S \rightarrow (\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C})$ 4) a start information $c_o \in \mathcal{C}$

The specification of an interprocedural DFA consists of four components, which as in the intraprocedural case, can directly be fed into the computation tools (see table above). As no detailed knowledge of the tools is required, we only present their underlying intuition. Details can be found in the Appendix.

The key tool of the interprocedural tool kit is the generic fixed point Algorithm A.2, which computes the *effect* of procedure call nodes with respect to $\mathcal{C}$, $[\![\ ]\!]'$, and $\mathcal{R}$. To this end it computes for every node of S two functions $[\![\![n]\!]\!]$ and $[\![n]\!]$ on $\mathcal{C}$. Intuitively, $[\![\![n]\!]\!]$ transforms a data flow information assumed to be valid at the entry of the procedure containing n into the data flow information, which is valid before the execution of n, and $[\![n]\!]$ describes the effect of transforming a data flow information by n. In particular, if n is a procedure call node, $[\![n]\!]$ represents the full effect of the procedure called by n in this particular context. The functions $[\![\![n]\!]\!]$ and $[\![n]\!]$ evolve as the greatest fixed points of Equation System 3.1, which we report here without further details, as they are unimportant for the use of the tool kit (see also Algorithm A.2).

Equation System 3.1 (Effects of Procedures)

$$[\![\![n]\!]\!] =_{df} \begin{cases} id_{\mathcal{C}} & \text{if } n \in \{\, \mathbf{s}_0, \ldots, \mathbf{s}_k \,\} \\ \bigsqcap \{ [\![m]\!] \circ [\![\![m]\!]\!] \mid m \in pred_{fg(n)}(n) \} & \text{otherwise} \end{cases}$$

and

$$[\![n]\!] =_{df} \begin{cases} [\![n]\!]' & \text{if } n \in \mathbf{N}^S \backslash \mathbf{N}_C^S \\ \mathcal{R}(n) \circ (id_{\mathcal{C}}, [\![n_R]\!]' \circ [\![\mathbf{e}]\!] \circ [\![\![\mathbf{e}]\!]\!] \circ [\![n_C]\!]') & \text{otherwise } (\, \mathbf{e} =_{df} end(callee(n))\,)^5 \end{cases}$$

Subsequently, the computed effects of procedures $[\![n]\!]$, $n \in \mathbf{N}_C^S$, are fed into the generic fixed point Algorithm A.1, which annotates the argument flow graph system with data flow information according to the start information $c_0 \in \mathcal{C}$. This annotation is given by the greatest solution of Equation System 3.2.

Equation System 3.2 (The Equation System of the *IMFP*-Strategy (Alg. A.1))

$$\mathbf{idfi}(n) \quad = \quad \begin{cases} c_0 & \text{if } n \,=\, \mathbf{s}_0 \\ \bigsqcap \{ \, [\![m_C]\!]'(\mathbf{idfi}(m)) \mid m \in caller(fg(n)) \, \} & \text{if } n \in \{\mathbf{s}_1, \ldots, \mathbf{s}_k\} \\ \bigsqcap \{ \, [\![m]\!](\mathbf{idfi}(m)) \mid m \in pred_{fg(n)}(n) \, \} & \text{otherwise} \end{cases}$$

[5] $\forall\, c \in \mathcal{C}.\ \mathcal{R}(n) \circ (id_{\mathcal{C}}, [\![n_R]\!]' \circ [\![\mathbf{e}]\!] \circ [\![\![\mathbf{e}]\!]\!] \circ [\![n_C]\!]')(c) =_{df} \mathcal{R}(n)(id_{\mathcal{C}}(c), [\![n_R]\!]'([\![\mathbf{e}]\!]([\![\![\mathbf{e}]\!]\!]([\![n_C]\!]'(c)))))$.

For technical reasons the interprocedural maximal fixed point solution defined in [Kn, KS1] is defined in terms of one-component stacks. However, identifying one-component stacks with their single component delivers that the greatest solution of Equation System 3.2, denoted by $\mathbf{idfi}_{c_0}$, coincides with the solution of the *interprocedural* extension of the *maximal fixed point strategy* of Section 2, which is defined in full detail in [Kn, KS1].

The *IMFP***-Solution:** $\forall\, n \in \mathbf{N}^S \;\forall\, c_0 \in \mathcal{C}.\;\; \mathit{IMFP}_{([\![\;]\!]', \mathcal{R}, c_0)}(n) \;=\; \mathbf{idfi}_{c_0}(n)$

3.3 Proof Obligations

As in the intraprocedural case, local semantic functionals can be extended to cover *interprocedurally valid* program paths, i.e., program paths respecting the call and return behaviour of procedure calls (cf. [My, SP, Kn, KS1]). This extension allows an *interprocedural* version of the *meet over all paths* (*IMOP*) strategy (cf. [Kn, KS1]). The point of this extension is that it mimics the run-time stack of a run-time system in the DFA scenario.

For constructing an optimal interprocedural DFA-algorithm, however, these details are not necessary. Thus we simply present the solution of this strategy in order to illustrate that it matches the pattern of its intraprocedural counterpart.

Here $\mathbf{IP}[\mathbf{s}_0, n)$ stands for the set of all interprocedurally valid paths from the start node $\mathbf{s}_0$ of the program to the program point n under consideration, and $[\![\;]\!]_{stack}$ is a semantic functional derived from $[\![\;]\!]'$ and $\mathcal{R}$, which works on data flow analysis stacks.

The *IMOP***-Solution:**

$$\forall\, n \in N' \;\forall\, c_0 \in \mathcal{C}.\; \mathit{IMOP}_{([\![\;]\!]', \mathcal{R}, c_0)}(n) = \bigsqcap \{\, [\![\, p \,]\!]_{stack}(c_0) \mid p \in \mathbf{IP}[\mathbf{s}_0, n) \,\}$$

As in the intraprocedural case, the *IMOP*-solution gives us a handle to decompose the optimality proof of an interprocedural DFA-algorithm into considerably small steps allowing modular proofs.

Completeness

Step 1: As in the intraprocedural case, the first step consists of an application of a coincidence theorem. In this case the following version is required (cf. [Kn, KS1]):

Theorem 3.3 (Interprocedural Coincidence Theorem)
Given a flow graph system $S = (G_0, G_1, \ldots, G_k)$*, the IMFP-solution and the IMOP-solution coincide, i.e.,* $\forall\, n \in \mathbf{N}^S \;\forall\, c_0 \in \mathcal{C}.\; \mathit{IMOP}_{([\![\;]\!]', \mathcal{R}, c_0)}(n) = \mathit{IMFP}_{([\![\;]\!]', \mathcal{R}, c_0)}(n)$*, if the local semantic functions* $[\![\, n \,]\!]'$*,* $n \in N'$*, and the return functions* $\mathcal{R}(n)$*,* $n \in \mathbf{N}^S_C$*, are distributive.*

Step 2: The second step closes the gap between the *IMOP*-solution and the program property of interest. For the same reasons as in the intraprocedural case, this gap is in general small and therefore considerably easy to prove.

Correctness

Step 3: Finally, it must be checked that the two fixed point algorithms terminate with the interprocedural maximal fixed point solution. Here the following lemma is central, which guarantees that the algorithms terminate with the desired solution.

Lemma 3.4 (Interprocedural Correctness Lemma)
Given a complete semi-lattice $\mathcal{C}$, a local semantic functional $[\![\ ]\!]'$, and a return functional $\mathcal{R}$, we have:

1. *Algorithm A.2 terminates with the full effect of the procedures with respect to $\mathcal{C}$, $[\![\ ]\!]'$, and $\mathcal{R}$, if the functions $[\![\, n \,]\!]'$, $n \in N'$, and $\mathcal{R}(n)$, $n \in \mathbf{N}_C^S$, are monotonic, and the function lattice $[\,\mathcal{C} \to \mathcal{C}\,]$ satisfies the descending chain condition.*
2. *Algorithm A.1 terminates for every flow graph system S with the IMFP-solution, if the functions $[\![\, n \,]\!]'$, $n \in N'$, and $\mathcal{R}(n)$, $n \in \mathbf{N}_C^S$, are monotonic, and $\mathcal{C}$ satisfies the descending chain condition.*

In contrast to the intraprocedural case, not only the lattice $\mathcal{C}$ itself must satisfy the descending chain condition, but also the function lattice $[\,\mathcal{C} \to \mathcal{C}\,]$ expressing the effects of procedures, which poses a much stronger restriction on $\mathcal{C}$.

As in the intraprocedural case, the *IMFP*-solution is still a safe approximation of the *IMOP*-solution, if the semantic functions and the return functions are only monotonic.

Summarizing, optimality of an interprocedural DFA-algorithm can be proved in four steps, which are illustrated in Figure 2 below. Note that none of these steps requires any knowledge about the generic fixed point algorithms or the underlying framework. Moreover, 1) is trivial if $\mathcal{C}$ is finite, which is the case in most practical applications.

Proving Optimality of an Interprocedural DFA-Algorithm	
Prove that	1) the function lattice $[\,\mathcal{C} \to \mathcal{C}\,]$ satisfies the *descending chain condition* 2) the local semantic functions $[\![\, n \,]\!]'$, $n \in N'$, are *distributive* 3) the return functions $\mathcal{R}(n)$, $n \in \mathbf{N}_C^S$, are *distributive* 4) the *IMOP*-solution and the program property ϕ of interest are *equivalent*
Obtain that	the specified interprocedural DFA is *optimal* in the sense of Section 3.1

4 Reference Parameters and Aliasing

Up to now the interprocedural tool kit was presented without explicitely referring to the effects of variable aliasing due to reference parameters. This is possible because all aliasing

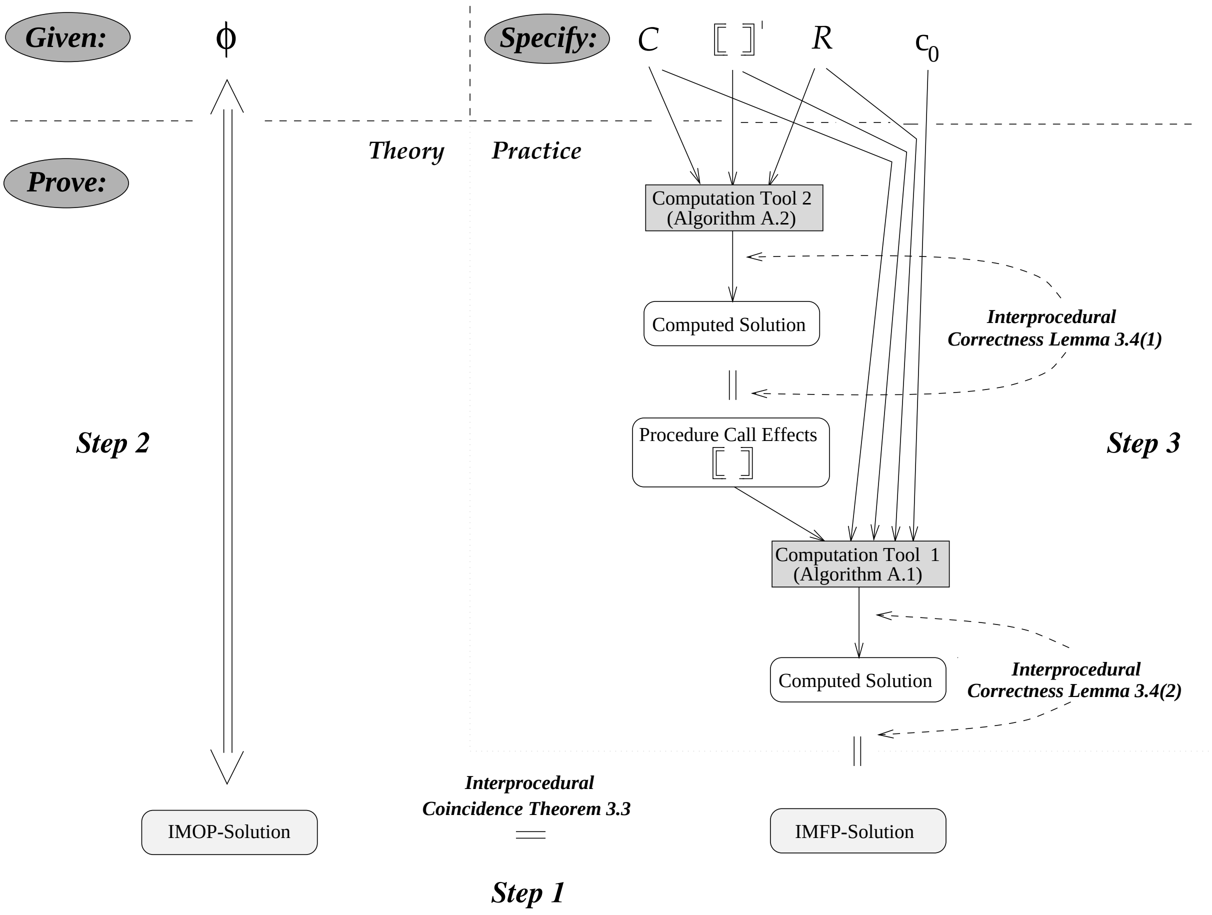

Figure 2: Illustrating the Interprocedural Situation

effects are encoded in the local semantic functions associated with elementary statements. Specifying a concrete DFA-algorithm, they must be made explicit by defining the local semantic functions accordingly. Technically, this is achieved by parameterizing the local semantic functions in the sets of *may* and *must* aliases of the variables and procedures of the program under consideration. The instantiations of these parameterized functions define the semantic functions for the tool kit. Subsequently, the instantiated functions can directly be fed into the computation tools of the tool kit.

Following this approach, the specification of a DFA can be done relatively to the alias-information computed in a preprocess e.g. by one of the algorithms reported in [Ban, Bth, Co, CK1, CK2, CK3, ERH, LH, My, We]. Of course, the preciseness of the resulting interprocedural analysis depends on the preciseness of the computed alias-information, but this is an orthogonal matter.

In the remainder of the paper we assume that for every variable v and procedure G of the program under consideration $MstAliases_G(v)$ and $MayAliases_G(v)$ denote the set of *must-aliases* and *may-aliases* of v in G, respectively. Given a term t, the induced notions of 'must-alias' and 'may-alias' representations of t are important for a number of DFA-problems considered in Section 5. Intuitively, a term t' is a must-alias (may-alias) representation of t, if it results from t by replacing some of the variables occurring in t by variables that are must-aliases (may-aliases) of the ones they replace.

These notions allow us to use alias-information in a black-box fashion for defining the local semantics and return functions of a DFA that take care of all aliasing effects. Demonstrating this in terms of our running example, we replace the predicates *Comp* and *Transp* by two predicates *A-Comp* and *A-Transp* that are parameterized in alias-information:

$$\forall\, n \in N'.\ \textit{A-Comp}\,(n) =_{df} \textit{Comp}\,(n)\ \vee\ \textit{Comp}_{\,MstAl}(n)$$

and

$$\forall\, n \in N'.\ \textit{A-Transp}\,(n) =_{df}\ \textit{Transp}\,(n)\ \wedge\ \begin{cases} \textit{true} & \text{if } n \in N_C' \cup N_R' \\ \textit{Transp}_{\,MayAl}(n) & \text{otherwise} \end{cases}$$

Intuitively, given a term t the predicate $\textit{A-Comp}\,(n)$ is true, if t itself (i.e., $\textit{Comp}\,(n)$) or one of its must-alias representations is computed at node n (i.e., $\textit{Comp}_{\,MstAl}(n)$). Similarly, for ordinary nodes, i.e. $n \in N' \backslash (N_C' \cup N_R')$, the predicate $\textit{A-Transp}\,(n)$ is true, if neither an operand of t (i.e., $\textit{Transp}\,(n)$) nor any of their may-aliases are modified by the assignment of node n (i.e., $\textit{Transp}_{\,MayAl}(n)$). For call and return nodes, i.e. $n \in N_C' \cup N_R'$, it is true if no operand of t is modified (i.e., $\textit{Transp}\,(n)$). This reflects the difference between ordinary assignments to reference parameters and parameter transfers to reference parameters, which are pointer updates leaving the memory untouched.

5 A Collection of Optimal Interprocedural DFAs

In this section we present a collection of optimal interprocedural data flow analyses in order to illustrate the simplicity and elegance of their tool kit based construction. We start with the interprocedural versions of *available expressions*, *reaching definitions*, *live variables*, and *very busy expressions*, which in [He] are used as representatives of a taxonomy of DFA-problems, in which data flow problems are classified according to the direction of data flow (forward, backward), and the desired fixed point solution (least, greatest). All four problems are practically relevant; e.g., information on reaching definitions of variables allows the computation of *definition-use-chains*, which is important e.g. for the automatic parallization of sequential programs. Subsequently, we present the interprocedural extension of the basic version of the algorithm for lazy code motion of [KRS1], which optimally eliminates all interprocedural partial redundancies of a program (cf. [Kn, KS2]).

All analyses satisfy the four conditions guaranteeing their optimality. Moreover, all analyses are as efficient as their corresponding intraprocedural versions, i.e., they have the same worst case time complexity.

For the remainder of this section we assume an arbitrary, but fixed flow graph system $S = (G_0, G_1, \ldots, G_k)$. $\textit{GlobVar}(S)$ denotes the set of *global variables* of S, i.e., the set of variables that are declared in the main procedure of S, which are accessible in all procedures of S. Additionally, $\textit{Var}(t)$ and $\textit{LhsVar}(n)$ denote the set of variables occurring in a term t and on the left hand side of the assignment of node n, respectively. Moreover, $\textit{GlobId}(t)$ and $\textit{LocId}(t)$ are abbreviations of $\textit{GlobVar}(S) \cap \textit{Var}(t)$ and $\textit{Var}(t) \backslash \textit{GlobVar}(S)$, respectively. Given a term t, we additionally need the functions $\textit{NoSurvMod} : N' \to \mathcal{B}$ and $\textit{PotAccessible} : S \to \mathcal{B}$. Intuitively, *NoSurvMod* indicates that if the memory location of a variable v of t is modified by a node n, then it will not survive the end of the current procedure call, i.e., the corresponding memory location has been allocated during the execution of the current procedure call. Similarly, *PotAccessible* indicates that the memory locations of all variables v of t that are accessible immediately before entering G remain accessible after entering it, i.e., by v itself or by one of its must-aliases. Formally, these functions are defined as follows:

$$\forall\, n \in N'.\ \mathit{NoSurvMod}(n) =_{df} \begin{cases} \mathit{true} & \text{if } n \in N'_C \cup N'_R \\ \mathit{Transp}_{\mathit{LocId}}(n) \ \wedge\ \mathit{Transp}_{\mathit{GlobId}}(n) & \text{otherwise} \end{cases}$$

where for all nodes $n \in N' \backslash (N'_C \cup N'_R)$ the predicates $\mathit{Transp}_{\mathit{LocId}}$ and $\mathit{Transp}_{\mathit{GlobId}}$ are defined by:

$$\mathit{Transp}_{\mathit{LocId}}(n) =_{df} \mathit{LocId}(t) \cap \mathit{MayAliases}_{fg(n)}(\mathit{LhsVar}(n)) = \emptyset$$

$$\mathit{Transp}_{\mathit{GlobId}}(n) =_{df} \mathit{GlobId}(t) \cap (\mathit{LhsVar}(n) \cup \mathit{MayAliases}_{fg(n)}(\mathit{LhsVar}(n))) = \emptyset$$

The predicate *NoSurvMod* is true for all nodes $n \in N'_C \cup N'_R$, since the assignments of call and return nodes only modify memory locations allocated in the procedure call under consideration. Similarly, it is true for ordinary nodes, i.e., $n \in N' \backslash (N'_C \cup N'_R)$, if no global identifier of t is modified by the assignment of node n (i.e., $\mathit{Transp}_{\mathit{GlobId}}(n)$), and if no may-alias of a local identifier is modified (i.e., $\mathit{Transp}_{\mathit{LocId}}(n)$). This reflects the fact that 'direct' modifications of local identifiers (i.e., $\mathit{LocId}(t) \cap \mathit{LhsVar}(n) \neq \emptyset$) only affect memory locations allocated in the procedure call under consideration.

$$\forall\, G \in S.\ \mathit{PotAccessible}(G) =_{df} \forall\, v \in \mathit{Var}(t).\ v \in \mathit{GlobVar}(S) \ \vee\ \mathit{MstAliases}_G(v) \neq \emptyset$$

A procedure G satisfies the predicate *PotAccessible*, if all variables of t are global variables of S (i.e., $v \in \mathit{GlobVar}(S)$), or have a *must-alias* in G (i.e., $\mathit{MstAliases}_G(v) \neq \emptyset$).

5.1 Available Expressions

In this section we specify an interprocedural DFA for computing the availability of a given program term t. Essentially, a term t is *interprocedurally available* at a program point n, if on every program path reaching node n the last modification of an operand of t (in the sense of the predicate $\neg$*A-Transp*) is followed by a computation of t (in the sense of the predicate *A-Comp*). The term "interprocedurally available" indicates the fact that there are potentially infinitely many copies of local variables of recursive procedures. This must be taken care of in any formal definition of interprocedural program properties. We omit the rather straightforward but technically tedious definitions here. They can be found in [Kn].

1. *Lattice*: $(\mathcal{B}^2, \wedge, \leq, (\mathit{false}, \mathit{false}), (\mathit{true}, \mathit{true}))$

2. *Local Semantic Functional*: $[\![\ ]\!]'_{av} : N' \to (\mathcal{B}^2 \to \mathcal{B}^2)$ defined by

$$\forall\, n \in N'\ \forall\, (b_1, b_2) \in \mathcal{B}^2.\ [\![\, n \,]\!]'_{av}(b_1, b_2) =_{df} (b'_1, b'_2)$$

where

$$b'_1 =_{df} \mathit{A\text{-}Transp}(n) \wedge (\mathit{A\text{-}Comp}(n) \vee b_1)$$

$$b'_2 =_{df} \begin{cases} b_2 \wedge \mathit{NoSurvMod}(n) & \text{if } n \in N' \backslash N'_C \\ \mathit{true} & \text{otherwise} \end{cases}$$

3. *Return Functional*: $\mathcal{R}_{av} : \mathbf{N}_C^S \to (\,\mathcal{B}^2 \times \mathcal{B}^2 \to \mathcal{B}^2\,)$ defined by

$$\forall\, n \in \mathbf{N}_C^S\ \forall\,(\,(b_1,b_2),\ (b_3,b_4)\,) \in \mathcal{B}^2 \times \mathcal{B}^2.\ \mathcal{R}_{av}(n)(\,(b_1,b_2),\ (b_3,b_4)\,) =_{df} (b_5,b_6)$$

where

$$b_5 =_{df} \begin{cases} b_3 & \text{if } \mathit{PotAccessible}\,(\mathit{callee}(n)) \\ (b_1 \vee \mathit{A\text{-}Comp}\,(n)) \wedge b_4 & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

4. *Start Information*: $(\mathit{false}, \mathit{true}) \in \mathcal{B}^2$

Whereas the first bit of a data flow information expresses availability information essentially as in the intraprocedural case, the second bit keeps track of modifications of variables of t that have to be maintained after returning from a (recursive) procedure call. In particular, it allows us to distinguish different copies of the local variables of recursive procedures, which is important for an interprocedural framework capturing local variables.

The intuition behind the return functional is as follows. If all memory locations of variables of t are accessible in a procedure G, entering G is essentially the same as traversing a normal edge. Thus, in this case t is available after the call (i.e., $b_5 = \mathit{true}$), if it is available immediately before leaving G (i.e., $b_3 = \mathit{true}$). Otherwise, i.e., if there is at least one variable of t, whose memory location is not accessible throughout G, t is available after the call, if it is available before the call (i.e., $b_1 = \mathit{true}$) or occurs as an actual parameter (i.e., $\mathit{A\text{-}Comp}\,(n)$), and if no memory location of a variable of t is modified as a side-effect of the call, which has been allocated by a previous procedure call (i.e., $b_4 = \mathit{true}$).

The intuition behind the return functional also applies to the other analyses presented in this paper.

5.2 Very Busy Expressions

A term t is *very busy* at a program point n, if it is used on every terminating program path before any of its operands is modified. Thus the flow of information is opposite to the one of the availability problem, and a backward analysis is needed. Technically this is taken care of by inverting the edges in the argument flow graph system (cf. [He]). Note that this requires to interchange the roles of call nodes and return nodes in an interprocedural framework.

1. *Lattice*: $(\,\mathcal{B}^2, \wedge, \leq, (\mathit{false}, \mathit{false}), (\mathit{true}, \mathit{true})\,)$

2. *Local Semantic Functional*: $[\![\ ]\!]_{vbe}' : N' \to (\,\mathcal{B}^2 \to \mathcal{B}^2\,)$ defined by

$$\forall\, n \in N'\ \forall\,(b_1,b_2) \in \mathcal{B}^2.\ [\![\, n \,]\!]_{vbe}'(b_1,b_2) =_{df} (b_1', b_2')$$

where

$$b_1' =_{df} \mathit{A\text{-}Comp}\,(n) \vee (\mathit{A\text{-}Transp}\,(n) \wedge b_1)$$

$$b_2' =_{df} \begin{cases} b_2 \wedge \mathit{NoSurvMod}(n) & \text{if } n \in N' \backslash N_R' \\ \mathit{true} & \text{otherwise} \end{cases}$$

3. *Return Functional*: $\mathcal{R}_{vbe} : \mathbf{N}_C^S \to (\, \mathcal{B}^2 \times \mathcal{B}^2 \to \mathcal{B}^2 \,)$ defined by

$$\forall\, n \in \mathbf{N}_C^S \;\forall\, (\, (b_1, b_2),\, (b_3, b_4)\,) \in \mathcal{B}^2 \times \mathcal{B}^2.\; \mathcal{R}_{vbe}(n)(\, (b_1, b_2),\, (b_3, b_4)\,) =_{df} (b_5, b_6)$$

where

$$b_5 =_{df} \begin{cases} b_3 & \text{if } \mathit{PotAccessible}\,(\mathit{callee}(n)) \\ \mathit{A\text{-}Comp}\,(n) \;\vee\; (b_1 \wedge b_4) & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

4. *Start Information*: $(\mathit{false}, \mathit{true}) \in \mathcal{B}^2$

5.3 Reaching Definitions

A *definition* of a variable v is an occurrence of v on the left hand side of an assignment. A definition *reaches* a program point n, if there is a modification free path from the node containing the definition to n. The problem of reaching definitions is to determine for every definition of a variable the set of all program points reached by it. The corresponding interprocedural DFA, which is a forward analysis, is specified below, where $\mathit{Definition}\,(n)$ is a predicate, which is true for the node containing the definition of v under consideration, and false otherwise. Note that the memory location of $v = \mathit{Var}(\mathit{Definition}\,(n))$ is either accessible in a procedure or not. There are no 'mixed situations' or 'partial accesses' as in the case of terms, which simplifies the specification.

1. *Lattice*: $(\, \mathcal{B},\, \vee\,, \geq, \mathit{true}, \mathit{false}\,)$
2. *Local Semantic Functional*: $[\![\;]\!]'_{rd} : N' \to (\, \mathcal{B} \to \mathcal{B}\,)$ defined by

$$\forall\, n \in N' \;\forall\, b \in \mathcal{B}.\; [\![\, n \,]\!]'_{rd}(b) =_{df} \mathit{Definition}\,(n) \;\vee\; (\mathit{A\text{-}Transp}\,(n) \;\wedge\; b)$$

3. *Return Functional*: $\mathcal{R}_{rd} : \mathbf{N}_C^S \to (\, \mathcal{B}^2 \to \mathcal{B}\,)$ defined by

$$\forall\, n \in \mathbf{N}_C^S \;\forall\, (b_1, b_2) \in \mathcal{B}^2.\; \mathcal{R}_{rd}(n)(b_1, b_2) =_{df} \begin{cases} b_2 & \text{if } \mathit{PotAccessible}\,(\mathit{callee}(n)) \\ b_1 & \text{otherwise} \end{cases}$$

4. *Start Information*: $\mathit{false} \in \mathcal{B}$

5.4 Live Variables

A variable v is *live* at a program point n, if there is definition free program path from n to a use site of v, i.e., to a node where v has a right hand side occurrence. The corresponding analysis problem is concerned with the determination of all program points, where a given program variable is live. This problem requires a backward analysis. As in the reaching definition problem there are no 'mixed situations', because we are dealing with variables only. In the following specification, $\mathit{A\text{-}Used}\,(n)$ is a predicate, which is true exactly for the nodes containing a right hand side occurrence of v or of one of its *must-aliases*.

1. *Lattice*: $(\, \mathcal{B},\, \vee\,, \geq, \mathit{true}, \mathit{false}\,)$

2. *Local Semantic Functional*: $[\![\ ]\!]'_{lv} : N' \to (\mathcal{B} \to \mathcal{B})$ defined by

$$\forall\, n \in N'\ \forall\, b \in \mathcal{B}.\ [\![\, n\,]\!]'_{lv}(b) =_{df} \mathit{A\text{-}Used}(n) \vee (\mathit{A\text{-}Transp}(n) \wedge b)$$

3. *Return Functional*: $\mathcal{R}_{lv} : \mathbf{N}_C^S \to (\mathcal{B}^2 \to \mathcal{B})$ defined by
   $\forall\, n \in \mathbf{N}_C^S\ \forall\, (b_1, b_2) \in \mathcal{B}^2.$

$$\mathcal{R}_{lv}(n)(b_1, b_2) =_{df} \begin{cases} b_2 & \text{if } \mathit{PotAccessible}(\mathit{callee}(n)) \\ \mathit{A\text{-}Used}(n) \vee b_1 & \text{otherwise} \end{cases}$$

4. *Start Information*: $\mathit{false} \in \mathcal{B}$

5.5 Optimal Elimination of Interprocedural Partial Redundancies

A computation of a term t is *partially redundant*, if it is computed twice on some program paths. Following Morel and Renvoise [MR], our partial redundancy elimination algorithm is based on the idea of computing terms "as early as possible" in a program, and carrying the computed values in temporaries to the original computation sites. Intuitively, "possible" means here that the computed value is very busy at the computation point, i.e., that it is used on every terminating program path. This is necessary in order to preserve the semantics of the program. "As early as possible" means that an earlier computation would either be not very busy, or would not deliver the desired value due to a subsequent modification.

Recently, in [KRS1] an algorithm for eliminating intraprocedural partial redundancies was presented that directly realizes this intuition: given a program term t, it determines the set of program points, where t is *very busy*,[6] and the set of program points, where t is *earliest*. Subsequently, it initializes a temporary $\mathbf{h}$ at all nodes that are both very busy and earliest, and replaces all original occurrences of t by $\mathbf{h}$. This results in a *computationally optimal* program, i.e., in a program that cannot be improved further by means of partial redundancy elimination.

This result can straightforward be interprocedurally generalized to capture alias information [Kn].[7] One only needs to replace the intraprocedural versions of the analysis by their interprocedural counterparts specified in Section 5.2 for very business and Section 5.5.1 for earliestness, and to modify the replacement phase to also capture all 'must-alias' occurrences of t by $\mathbf{h}$.[8]

5.5.1 Earliestness

In this section we present the interprocedural version of the earliestness analysis of [KRS1], which is the last unspecified ingredient of our partial redundancy elimination algorithm.

[6] Very business is called *down-safety* in [KRS1, KRS2].

[7] As in the intraprocedural case this requires some edge splitting, which is typical for partial redundancy elimination (cf. [DRZ, KRS1, KRS2, RWZ]).

[8] Interprocedurally, $\mathbf{h}$ has to be declared in the main procedure, if t does not contain local variables; otherwise in that procedure, of which it contains local variables (This assumes that the program is *distinguished*, i.e., different defining occurrences of identifiers use different identifiers). In fact, the corresponding optimization results in *interprocedurally* computationally optimal programs (cf. [Kn, KS2]).

Intuitively, a computation t is earliest at a program point n, if there is program path from the start node of the program to n, on which the computation of t is never very busy after the last modification of one of its operands. The predicate *VeryBusyExp* denotes the result of the very-busy-analysis specified in Section 5.2.

1. *Lattice*: $(\mathcal{B}^2, (\vee, \wedge), (\geq, \leq), (true, false), (false, true))$

2. *Local Semantic Functional*: $[\![\ ]\!]'_{ea} : N' \to (\mathcal{B}^2 \to \mathcal{B}^2)$ defined by

$$\forall\, n \in N'\ \forall\, (b_1, b_2) \in \mathcal{B}^2.\ [\![\, n\,]\!]'_{ea}(b_1, b_2) =_{df} (b_1', b_2')$$

   where

$$b_1' =_{df} \neg A\text{-}Transp\,(n) \vee (\neg\, VeryBusyExp\,(n) \wedge b_1)$$

$$b_2' =_{df} \begin{cases} b_2 \wedge NoSurvMod(n) & \text{if } n \in N' \backslash N_C' \\ true & \text{otherwise} \end{cases}$$

3. *Return Functional*: $\mathcal{R}_{ea} : \mathbf{N}_C^S \to (\mathcal{B}^2 \times \mathcal{B}^2 \to \mathcal{B}^2)$ defined by

$$\forall\, n \in \mathbf{N}_C^S\ \forall\, (\,(b_1, b_2),\ (b_3, b_4)\,) \in \mathcal{B}^2 \times \mathcal{B}^2.\ \mathcal{R}_{ea}(n)(\,(b_1, b_2),\ (b_3, b_4)\,) =_{df} (b_5, b_6)$$

   with

$$b_5 =_{df} \begin{cases} b_3\ \wedge\ \neg\, VeryBusyExp\,(end(callee(n))) & \text{if } PotAccessible\,(callee(n)) \\ \neg b_4\ \vee\ (\neg\, VeryBusyExp\,(n) \wedge b_1) & \text{otherwise} \end{cases}$$

$$b_6 =_{df} b_2 \wedge b_4$$

4. *Start Information*: $(true, true) \in \mathcal{B}^2$, if $Var(t) \subseteq GlobVar(S)$; $(false, true) \in \mathcal{B}^2$ otherwise.

5.5.2 The Transformation

The power of extending the very-busy/earliest-transformation of [KRS1] interprocedurally is illustrated in the example of Figure 3, which uses the convention that a procedure with name π, formal reference parameters $f_1, ..., f_q$, formal value parameters $g_1, ..., g_r$, and local variables $v_1, ..., v_s$ is denoted by

$$\pi\ (REF\ f_1, ..., f_q;\ VAL\ g_1, ..., g_r);\ VAR\ v_1, ..., v_s$$

Given the alias-information $MstAliases_{\pi_1}(x) = MayAliases_{\pi_1}(x) = \{a\}$ and $MayAliases_{\pi_1}(y) = \{a, b\}$, our algorithm is unique in producing the result of Figure 4, which is exceptional, because it eliminates the partially redundant computations of $a + b$ at node **14**, **17**, **23**, **28**, **29**, and **32**, of $c * d$ at node **9**, of $d * e$ at node **8** and **10**, of $c * u$ at node **31**, and of $u + v$ at node **24**, **26** and **30** by replacing them by computations at nodes **4**, **15**, **18** and **30**, at node **1**, at node **3** and **25**, at node **12**, and at node **22**, respectively.
The very-busy/earliest-transformation can be refined to additionally minimize the lifetimes of introduced temporaries or to capture strength reduction in the fashion of [KRS1, KRS2] and [KRS3], respectively. Furthermore, following the lines of [KRS4] it can be extended to realize partial dead code elimination.

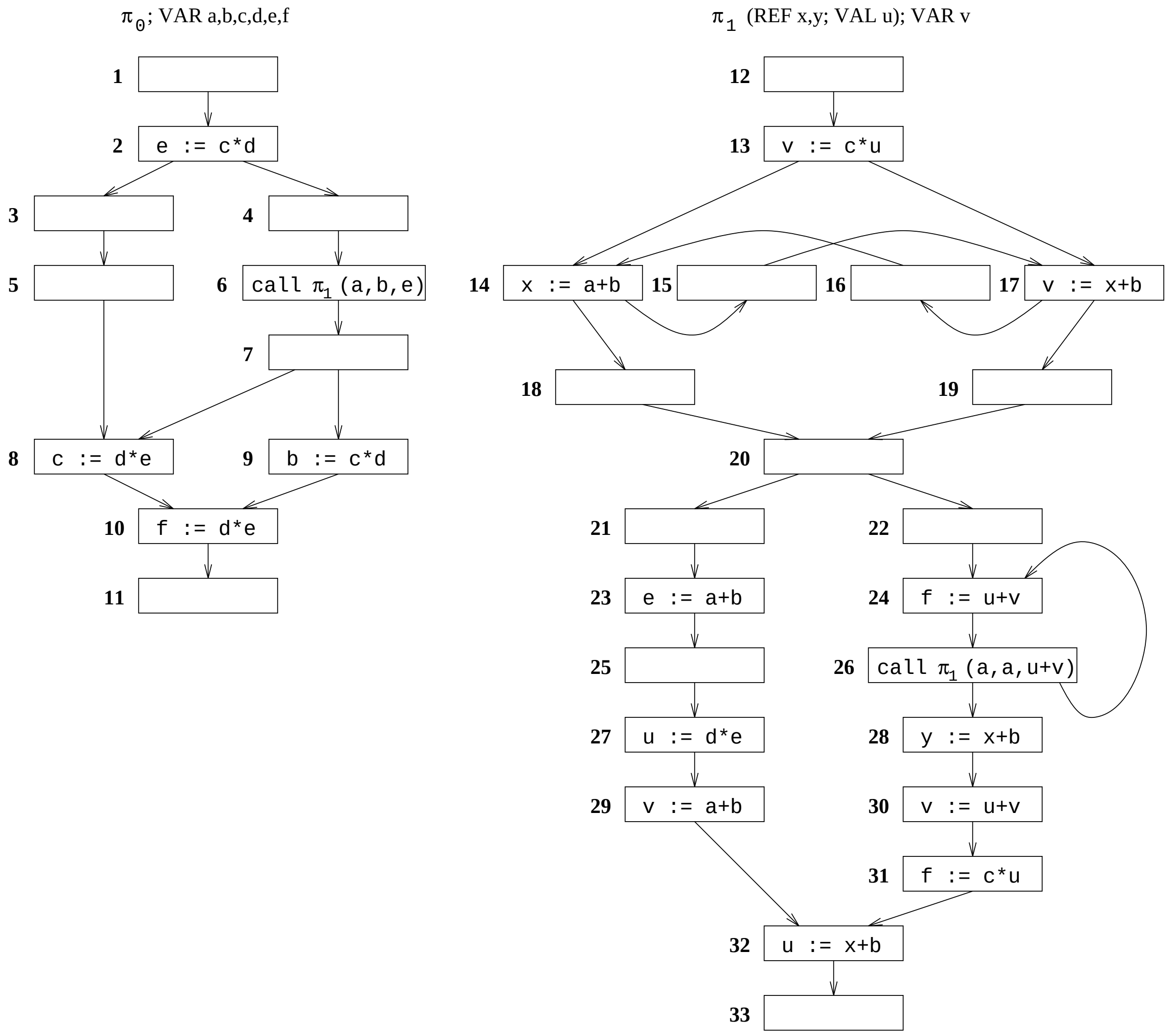

Figure 3: A Program with Interprocedural Partial Redundancies

6 Conclusions

We have presented a tool kit for constructing optimal interprocedural data flow analyses for programs with mutually recursive procedures, global and local variables, formal value and formal reference parameters. The tool kit contains *specification tools* allowing concise specifications of data flow analyses, and *computation tools* that can directly be fed with the specifications. Moreover, proving an analysis to be optimal only requires knowledge about the specification. The practicality of the tool kit has been illustrated by presenting a collection of optimal interprocedural DFA-algorithms solving well-known DFA problems, which are as efficient as their intraprocedural counterparts. Moreover, the tool kit is open for extensions: Following the lines of [Kn] it can be extended to capture procedural procedure parameters.

References

[Ba1] Barth, G. Interprozedurale Datenflußsysteme. Habilitationsschrift, University of Kaiserslautern, Germany, 1981.

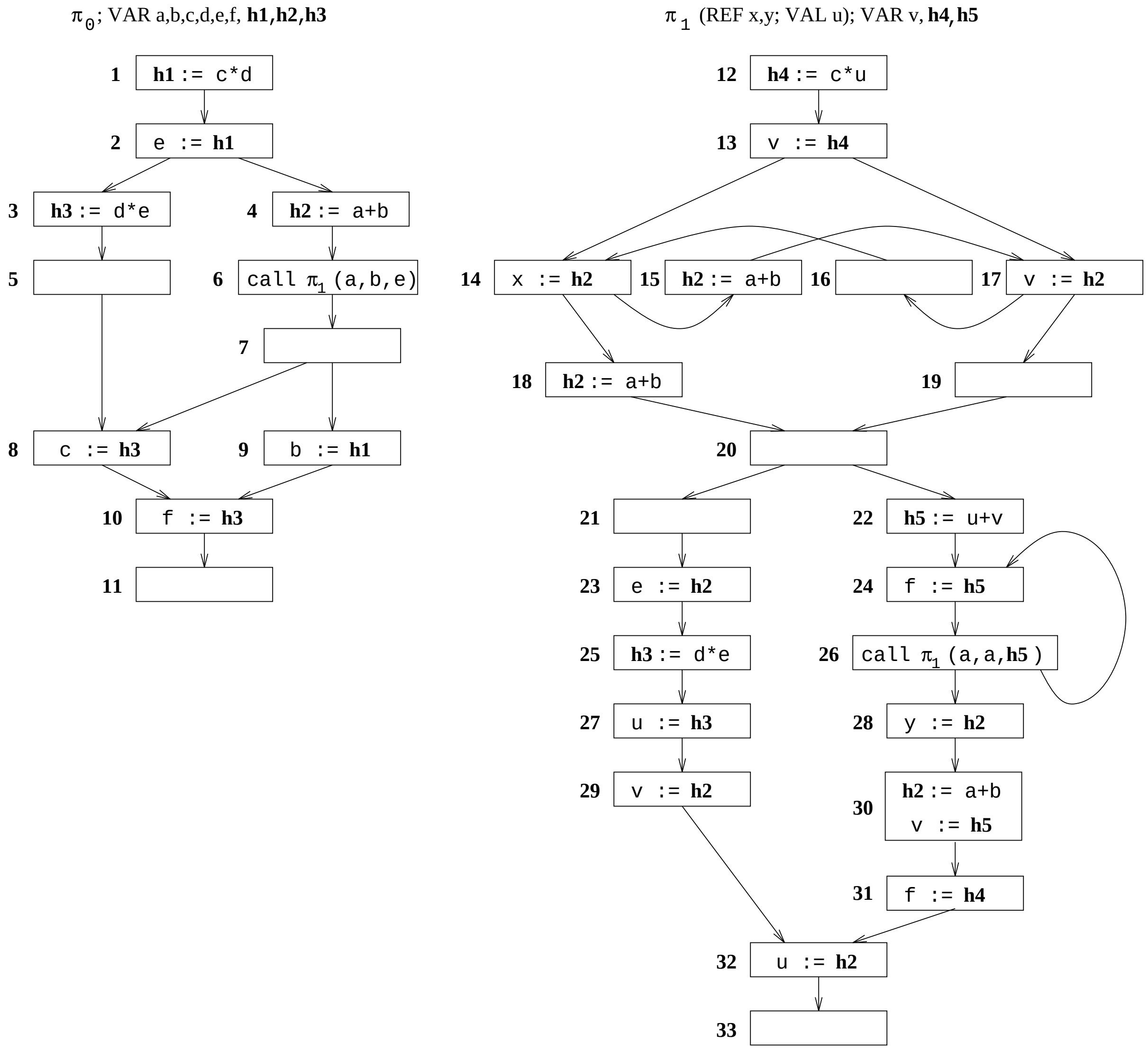

Figure 4: Optimal Elimination of Interprocedural Partial Redundancies

[Ba2] Barth, G. Interprocedural data flow systems. In *Proceedings of the 6^{th} GI-Conference*, Dortmund, Germany, Springer-Verlag, LNCS 145 (1983), 49 - 59.

[Ban] Banning, J. P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the 6^{th} Annual ACM Symposium on Principles of Programming Languages (POPL'79)*, San Antonio, Texas, 1979, 724 - 736.

[Bth] Barth, J. M. An interprocedural data flow analysis algorithm. In *Conference Record of the 4^{th} Annual ACM Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, California, 1977, 119 - 131.

[Bo] Bourdoncle, F. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Proceedings of the 2^{nd} International Symposium on Programming Language Implementation and Logic Programming (PLILP'90)*, Linköping, Sweden, Springer-Verlag, LNCS 456 (1990), 307 - 323.

[Co] Cooper, K. D. Analyzing aliases of reference formal parameters. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL'85)*, New Orleans, Louisiana, 1985, 281 - 290.

[CC1] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, California, 1977, 238 - 252.

[CC2] Cousot, P., and Cousot, R. Static determination of dynamic properties of recursive procedures. In *Proceedings of the 2nd IFIP TC-2 Working Conference on Formal Description of Programming Concepts*, St. Andrews, N. B., Canada (August 1977). Neuhold, E. J. (Ed.), New York: North-Holland, 1978, 237 - 277.

[CCKT] Callahan, D., Cooper, K. D., Kennedy, K. W., and Torczon, L. M. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, Palo Alto, California, *SIGPLAN Notices 21*, 7 (1986), 152 - 161.

[CK1] Cooper, K. D., and Kennedy, K. W. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, Montreal, Canada, *SIGPLAN Notices 19*, 6 (1984), 247 - 258.

[CK2] Cooper, K. D., and Kennedy, K. W. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI'88)*, Atlanta, Georgia, *SIGPLAN Notices 23*, 7 (1988), 57 - 66.

[CK3] Cooper, K. D., and Kennedy, K. W. Fast interprocedural alias analysis. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, Austin, Texas, 1989, 49 - 59.

[De] Deutsch, A. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 230 - 241.

[DRZ] Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, *SIGPLAN Notices 27*, 7 (1992), 212 - 223.

[DGS] Duesterwald, E., Gupta, R., and Soffa, M. L. Demand-driven computation of interprocedural data flow. To appear in *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL'95)*, San Francisco, California, January 23 - 25, 1995.

[ERH] Emami, M., Rakesh, G., and Hendren, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 242 - 256.

[GT] Grove, D., and Torczon, L. Interprocedural constant propagation: A study of jump function implementation. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI'93)*, Albuquerque, New Mexico, *SIGPLAN Notices 28*, (1993), 90 - 99.

[He] Hecht, M. S. Flow analysis of computer programs. Elsevier, North-Holland, 1977.

[HRB] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26 - 60.

[HS] Harrold, M. J., and Soffa, M. L. Efficient computation of interprocedural definition-use chains. In *Transactions on Programming Languages and Systems 16*, 2 (1994), 175 - 204.

[JM] Jones, N. D., and Muchnick, S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82)*, Albuquerque, New Mexico, 1982, 66 - 74.

[Kn] Knoop, J. Optimal interprocedural program optimization: A new framework and its application. PhD dissertation, Department of Computer Science, University of Kiel, 1993. To appear as monograph in the series *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, *SIGPLAN Notices 27*, 7 (1992), 224 - 234.

[KRS2] Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems 16*, 4 (1994), 1117 - 1155.

[KRS3] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

[KRS4] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 147 - 158.

[KS1] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, Paderborn, Germany, Springer-Verlag, LNCS 641 (1992), 125 - 140.

[KS2] Knoop, J., and Steffen, B. Efficient and optimal interprocedural bit-vector data flow analyses: A uniform interprocedural framework. Department of Computer Science, University of Kiel, Technical Report 9309 (1993), 22 pages.

[KU] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, (1977), 309 - 317.

[LH] Larus, J. R., and Hilfinger, P. N. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI'88)*, Atlanta, Georgia, *SIGPLAN Notices 23*, 7 (1988), 21 - 34.

[Mo] Morel, E. Data flow analysis and global optimization. In: Lorho, B. (Ed.). Methods and tools for compiler construction. Cambridge University Press, 1984, 289 - 315.

[My] Myers, E. W. A precise inter-procedural data flow algorithm. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, Williamsburg, Virginia, 1981, 219 - 230.

[MJ] Muchnick, S. S., and Jones, N. D. (Eds.). Program flow analysis: Theory and applications. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[MR] Morel, E., and Renvoise, C. Interprocedural elimination of partial redundancies. In [MJ], 1981, 160 - 188.

[Re1] Reps, T. Solving demand versions of interprocedural analysis problems. In *Proceedings of the 5th International Conference on Compiler Construction (CC'94)*, Edinburgh, UK, Springer-Verlag, LNCS 786 (1994), 389 - 403.

[Re2] Reps, T. Demand interprocedural program analysis using logic databases. In Ramakrishnan, R. (Ed.): *Applications of logic databases*. Kluwer Academic Publishers, Boston, Massachusetts, 1994.

[Re3] Reps, T. Private communication, November 1994.

[Ro] Rosen, B. K. Data flow analysis for procedural languages. *Journal of the ACM 26*, 2 (1979), 322 - 344.

[RHS] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. To appear in *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL'95)*, San Francisco, California, January 23 - 25, 1995.

[RWZ] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, San Diego, California, 1988, 2 - 27.

[SP] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. In [MJ], 1981, 189 - 233.

[SRH] Sagiv, M., Reps, T., and Horwitz, S. Precise interprocedural dataflow analysis with applications to constant propagation. Submitted for publication.

[We] Weihl, W. E. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL'80)*, Las Vegas, Nevada, 1980, 83 - 94.

A Computation Tools

Algorithm A.1 (Computing the (Interprocedural) Maximal Fixed Point Solution)

Input: *A flow graph G (*interprocedurally*: a flow graph system S), a lattice $\mathcal{C}$ satisfying the descending chain condition, a start information $c_0 \in \mathcal{C}$, and for every node n of G a monotonic local semantic function $[\![\, n \,]\!]$ (*interprocedurally*: for every node n of N'_C the monotonic local semantic function $[\![\, n \,]\!]'$ of the specification, and for every node n of S the monotonic local semantic function $[\![\, n \,]\!]$ as computed by Algorithm A.2).*

Output: *An annotation of G (*interprocedurally*: of S) with the maximal fixed point solution MFP (*interprocedurally*: IMFP) (stored in dfi).*

Remark: *The variable workset controls the iterative process. Its elements are tuples, whose first components are nodes of the flow graph G (*interprocedurally*: the flow graph system S), and whose second components are elements of $\mathcal{C}$ specifying a new approximation for the (interprocedural) maximal fixed point solution of the node of the first component. The auxiliary variable meet stores the result of the most recent meet operation. Identifying a single flow graph with a flow graph system consisting of the main procedure only allows a uniform notation for the intra- and interprocedural case.*

(Initialization of the annotation array dfi and the variable workset)
FORALL $m \in \mathbf{N}^S \backslash \{\mathbf{s}_0\}$ **DO** $dfi[m] := \top$ **OD**;
$dfi[\mathbf{s}_0] := c_0$;
$workset := \{\, m \mid [\![\, m \,]\!](dfi[m]) \sqsubset \top \,\}$;

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
  LET $m \in workset$
    BEGIN
      $workset := workset \backslash \{\, m \,\}$;
      (Update the "environment" of node m)
      FORALL $n \in succ_{fg(m)}(m)$ **DO**
        $meet := dfi[n] \sqcap [\![\, m \,]\!](dfi[m])$;
        IF $dfi[n] \sqsupset meet$
          THEN
            $dfi[n] := meet$;
            $workset := workset \cup \{\, n \,\}$
        FI
      OD;
      IF $m \in \mathbf{N}^S_C$
        THEN
          $meet := dfi[start(callee(m))] \sqcap [\![\, m_C \,]\!]'(dfi[m])$;
          IF $dfi[start(callee(m))] \sqsupset meet$
            THEN
              $dfi[start(callee(m))] := meet$;
              $workset := workset \cup \{\, start(callee(m)) \,\}$ **FI**
      FI
    END
OD.

Algorithm A.2 (Computing the Effects of Procedures)

Input: *A flow graph system* S*, a lattice* $\mathcal{C}$*, whose function lattice* $[\,\mathcal{C} \to \mathcal{C}\,]$ *satisfies the descending chain condition, for every node* n *of* N' *a monotonic local semantic function* $[\![\, n \,]\!]'$*, and for every node* n *of* $\mathbf{N}_C^S$ *a monotonic return function* $\mathcal{R}(n)$.

Output: *An annotation of* S *with functions* $[\![\![\, n \,]\!]\!] : \mathcal{C} \to \mathcal{C}$ *(stored in gtr, which stands for* global transformation*), and* $[\![\, n \,]\!] : \mathcal{C} \to \mathcal{C}$ *(stored in ltr, which stands for* local transformation*) representing the greatest solution of Equation System 3.1.*

Remark: *The variable workset controls the iterative process. Its elements are tuples, whose first components are nodes* $m \in \mathbf{N}^S$ *of the flow graph system* S*, and whose second components are functions* $f : \mathcal{C} \to \mathcal{C}$*, which specify a new approximation for the function* $[\![\![\, m \,]\!]\!]$ *of the node of the first component. Note that due to the mutual interdependence of the definitions of* $[\![\![\ ]\!]\!]$ *and* $[\![\ ]\!]$ *the iterative approximation of* $[\![\![\ ]\!]\!]$ *is superposed by an interprocedural iteration step, which updates the current approximation of the effect* $[\![\ ]\!]$ *of call nodes. The auxiliary variable meet stores the result of the most recent meet operation.*

(Initialization of the annotation arrays gtr and ltr and the variable workset)
FORALL $m \in \mathbf{N}^S$ **DO**
    $gtr[m] := \top_{[\,\mathcal{C} \to \mathcal{C}\,]}$;
    IF $m \in \mathbf{N}_C^S$
        THEN $ltr[m] := [\![\, m_R \,]\!]' \circ \top_{[\,\mathcal{C} \to \mathcal{C}\,]} \circ [\![\, m_C \,]\!]'$
        ELSE $ltr[m] := [\![\, m \,]\!]'$
    FI
OD;
$workset := \{\, (\mathbf{s}, id_{\mathcal{C}}) \mid \mathbf{s} \in \{\mathbf{s}_0, \ldots, \mathbf{s}_k\} \,\} \;\cup$
$\qquad\qquad \{\, (n, f) \mid f = \bigsqcap \{\, ltr[m] \circ gtr[m] \mid m \in pred_{fg(n)}(n) \,\} \sqsubset \top_{[\,\mathcal{C} \to \mathcal{C}\,]} \,\}$;

(Iterative fixed point computation)
WHILE $workset \neq \emptyset$ **DO**
    LET $(m, f) \in workset$
        BEGIN
            $workset := workset \backslash \{\, (m, f) \,\}$;
            $meet := gtr[m] \sqcap f$;
            IF $gtr[m] \sqsupset meet$
                THEN
                    $gtr[m] := meet$;
                    IF $m \in \{\mathbf{e}_i \mid i \in \{0, \ldots, k\}\}$
                        THEN
                            FORALL $l \in caller(fg(m))$ **DO**
                                $ltr[l] := \mathcal{R}(l) \circ (id_{\mathcal{C}}, [\![\, l_R \,]\!]' \circ ltr[m] \circ gtr[m] \circ [\![\, l_C \,]\!]')$;
                                $workset := workset \cup \{(n, ltr[l] \circ gtr[l]) \mid n \in succ_{fg(l)}(l) \,\}$
                            OD
                        ELSE
                            $workset := workset \cup \{\, (n, ltr[m] \circ gtr[m]) \mid n \in succ_{fg(m)}(m) \,\}$
                    FI
            FI
        END
OD.