The Interprocedural Coincidence Theorem

Jens Knoop * Bernhard Steffen †

Abstract

We present an interprocedural generalization of the well-known (intraprocedural) Coincidence Theorem of Kam and Ullman, which provides a sufficient condition for the equivalence of the *meet over all paths* (*MOP*) solution and the *maximal fixed point* (*MFP*) solution to a data flow analysis problem. This generalization covers arbitrary imperative programs with recursive procedures, global and local variables, and formal value parameters. In the absence of procedures, it reduces to the classical intraprocedural version. In particular, our stack-based approach generalizes the coincidence theorems of Barth and Sharir/Pnueli for the same setup, which do not properly deal with local variables of recursive procedures.

1 Motivation

Data flow analysis is a classical method for the static analysis of programs that supports the generation of efficient object code by "optimizing" compilers (cf. [He, MJ]). For imperative languages, it provides information about the program states that may occur at some given program points during execution.

Theoretically well-founded are data flow analyses that are based on *abstract interpretation* (cf. [CC1]). The point of this approach is to replace the "full" semantics by a simpler more abstract version, which is tailored to deal with a specific problem. Usually, the abstract semantics is specified by a local semantic functional, which gives abstract meaning to every program statement in terms of a transformation function from a lattice $\mathcal{C}$ into itself. The elements of $\mathcal{C}$ express the data flow information of interest. The (global) abstract semantics then results from one of the following two globalization strategies; the "operational" *meet over all paths* (*MOP*) strategy, and the "denotational" *maximal fixed point* (*MFP*) strategy[1] in the sense of Kam and Ullman [KU][2].

The *MOP*-strategy directly mimics possible program executions: it "meets" (intersects) all information corresponding to program paths reaching the program point under consideration. This specifies the optimal result of a globalization but is in general not effective.

The *MFP*-strategy iteratively approximates the greatest solution of a system of equations that express consistency between pre-conditions and post-conditions that are given in terms of data flow information: the pre-condition of a statement must be implied by each of the post-conditions of the predecessors, and the post-condition must be implied by the result of transforming the pre-condition according to the (abstract) meaning of the statement. In general, this leads to a suboptimal but algorithmic description.

The (intraprocedural) Coincidence Theorem of [KU] states the coincidence of the *MOP*-solution and the *MFP*-solution in the case of *distributive* (local) semantic functionals (see Section 2). In this paper, we present an *interprocedural* generalization of this theorem (cf. Interprocedural Coincidence

*Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Preußerstr. 1-9, D-2300 Kiel 1 – Part of the work was done, while the author was supported by the Deutsche Forschungsgemeinschaft grant La 426/9-2.

†Lehrstuhl für Informatik II, Rheinisch-Westfälische Technische Hochschule Aachen, Ahornstr. 55, D-5100 Aachen.

[1] These are the strategies that lead to the *MOP*-solution and *MFP*-solution, respectively.

[2] The operational and denotational flavour of these two strategies becomes particularly apparent in the interprocedural setting.

Theorem 5.3), which covers arbitrary imperative programs with recursive procedures, global and local variables, and formal value parameters. In the absence of procedures, it reduces to the classical intraprocedural version of Kam and Ullman. The point of our generalization is the introduction of stacks of lattice elements as data flow information, which is necessary to properly deal with local variables of recursive procedures. These stacks directly mimic the run-time stacks used by run-time systems for maintaining the activation records of different procedure incarnations. Whereas the operational (*MOP*) strategy exhaustively makes use of the stack structure, it turns out that the denotational (*MFP*) strategy only needs stacks of length at most two (Remark 4.7).

Related Work

Semantically based reasoning about interprocedural data flow analysis was first considered by Cousot/ Cousot [CC2], Rosen [Ro], Jones and Muchnick [JM], and more recently by Bourdoncle [Bo].

Rosen [Ro] proved the correctness of an interprocedural data flow analysis algorithm computing a maximal fixed point solution that provides information whether a variable is used, modified or preserved. Thus his algorithm is tailored to specific problems, and it is not clear how to generalize his approach to arbitrary abstract interpretations. Such a general situation was investigated in [CC2] and [JM], leading to highly technical definitions that are difficult to apply. Moreover, as in [Ro], these two papers only address the correctness aspect but no optimality. This holds also for the approach of [Bo], which considers correctness of data flow analyses with respect to the collecting semantics of a program.

Both correctness and optimality were considered by Barth [Ba1, Ba2] and Sharir/Pnueli [SP], who independently proposed an interprocedural version of the Coincidence Theorem of Kam and Ullman. However, their approaches do not properly deal with local variables of recursive procedures, which would require to store information about the local variables when treating a recursive procedure call.

Structure of the Paper

After sketching the intraprocedural setting in Section 2, we recall the formal framework for interprocedural data flow analysis in Section 3, and define two versions of abstract semantics for programs with procedures in Section 4: an "operational" one caused by the *interprocedural meet over all paths* strategy and a "denotational" one caused by the *interprocedural maximal fixed point* strategy. Subsequently, Section 5 presents the main results, Section 6 sketches some applications, and Section 7 contains our conclusions. Finally, the Appendix provides the detailed data flow analysis algorithms.

2 The Intraprocedural Setting

In this section we summarize the intraprocedural setting for data flow analysis, which 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.[3] 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 $j \in \{1, .., q-1\}$. $\mathbf{P}[m,n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}[m,n)$ the set of all finite paths from m to a predecessor of n. Moreover, $lgth(p)$ denotes the number of node occurrences in p, and ε the unique path of length 0. Finally, we assume that every node $n \in N$ lies on a path from $\mathbf{s}$ to $\mathbf{e}$.

[3]The construction of flow graphs is described in [All].

Given a complete semi-lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$, whose elements are intended to express the relevant data flow information, the *local* abstract semantics of a flow graph G is given by a semantic functional

$$[\![\ ]\!] : N \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$$

which gives meaning to every node $n \in N$ in terms of a transformation on $\mathcal{C}$. For simplicity, it is assumed that **s** and **e** are associated with the identity on $\mathcal{C}$.

The local abstract semantics $[\![\ ]\!]$ can easily be extended to cover finite paths as well. For every path $p = (n_1, ..., n_q) \in \mathbf{P}[m, n]$, we define:

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

The (global) abstract semantics then results from one of the following two globalization strategies: the "operational" ***meet over all paths (MOP)*** strategy, and the "denotational" ***maximal fixed point (MFP)*** strategy in the sense of Kam and Ullman [KU].

The *MOP*-strategy directly mimics possible program executions: it "meets" (intersects) all informations, which belong to a program path reaching the program point under consideration:

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

This directly reflects our desires but is in general not effective.

The *MFP*-strategy iteratively approximates the greatest solution of a system of equations which specifies the consistency between pre-conditions and post-conditions that are expressed in terms of $\mathcal{C}$:

Equation System 2.1

$$\mathbf{pre}(n) = \begin{cases} c_0 & \text{if } n = \mathbf{s} \\ \bigsqcap \{ \mathbf{post}(m) \mid m \in pred_G(n) \} & \text{otherwise} \end{cases}$$

$$\mathbf{post}(n) = [\![n]\!](\mathbf{pre}(n))$$

Denoting the greatest solution of Equation System 2.1 with respect to the start information $c_0 \in \mathcal{C}$ by $\mathbf{pre}_{c_0}$ and $\mathbf{post}_{c_0}$, the solution of the *MFP*-strategy is defined by:

The *MFP*-Solution: $\forall n \in N \ \forall c_0 \in \mathcal{C}.\ MFP_{c_0}(n) = \mathbf{pre}_{c_0}(n)$

In general, this leads to a suboptimal but algorithmic description.

Thus we have two global notions of semantics here, an operational one, which precisely mimics our intention, and a denotational one, which has an algorithmic character. In fact, we consider the *MOP*-strategy as a mean for the direct specification of data flow analysis problems, and the *MFP*-strategy as an algorithmic realization of such problems[4]. This view rises the question of *correctness* (safety) or even *completeness* (optimality) of such algorithms. For the elegant answer to these questions we need two further notions: given a complete semi-lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$, a function $f : \mathcal{C} \rightarrow \mathcal{C}$ is called

- ***monotonic*** iff $\forall c, c' \in \mathcal{C}.\ c \sqsubseteq c'$ implies $f(c) \sqsubseteq f(c')$
- ***distributive*** iff $\forall \mathcal{C}' \subseteq \mathcal{C}.\ f(\bigsqcap \mathcal{C}') = \bigsqcap \{ f(c) \mid c \in \mathcal{C}' \}$

It is well-known that distributivity is a stronger requirement than monotonicity in the following sense:

[4] Explicit algorithms are presented in Appendix A.

Lemma 2.2
A function $f : \mathcal{C} \to \mathcal{C}$ *is monotonic iff* $\forall\, \mathcal{C}' \subseteq \mathcal{C}.\ f(\sqcap \mathcal{C}') \sqsubseteq \sqcap \{f(c) \mid c \in \mathcal{C}'\}$

As in this lemma, $\mathcal{C}$ will always denote a complete semi-lattice. We have (cf. [KU]):

Theorem 2.3 (Safety Theorem)
Given a flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$, *the MFP-solution is a* correct *(or* safe*) approximation of the MOP-solution, i.e.* $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MFP_{c_0}(n) \sqsubseteq MOP_{c_0}(n)$, *if all the semantic functions* $[\![n]\!]$ *are monotonic.*

Distributivity of the semantic functions yields completeness (optimality). This follows from the well-known intraprocedural Coincidence Theorem 2.4 of [KU]:

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

3 Interprocedural Notions

In the interprocedural setting we represent programs Π as systems $(\pi_0, \pi_1, ., \pi_k)$ of (recursive) procedure definitions, where every $\pi \in \Pi$ has a list of formal value parameters and a list of local variables. π_0 is assumed to denote the *main program* 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.[5] Thus, the variables of the main program are global variables of the procedures, and can be accessed by them.

The denotational (*IMFP*) approach and the operational (*IMOP*) approach require different representations of programs Π: *flow graph systems* and *interprocedural flow graphs*.

Flow Graph Systems

The denotational approach works on systems $S = (G_0, G_1, ., G_k)$ of *flow graphs* with disjoint sets of nodes N_i and edges E_i, in which every procedure π of Π (including the main program π_0) is represented as a *directed flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$ in the sense of Section 2. $\mathbf{N}^S =_{df} \bigcup \{N_i \mid i \in \{0, ., k\}\}$ denotes the set of all nodes of S, $\mathbf{E}^S =_{df} \bigcup \{E_i \mid i \in \{0, ., k\}\}$ the set of all edges of S, and $\mathbf{N}^S_C \subseteq \mathbf{N}^S$ the set of all nodes representing procedure calls. Finally, we need the following functions, where $\mathcal{P}$ denotes the power set operator:

- $fg : \mathbf{N}^S \to S$ with $fg(n) =_{df} G_i$ iff $n \in N_i$,
- $callee : \mathbf{N}^S_C \to S$ with $callee(n) =_{df} G_i$ iff n repesents a procedure call of G_i,
- $caller : S \to \mathcal{P}(\mathbf{N}^S_C)$ with $caller(G_i) =_{df} \{n \mid callee(n) = G_i\}$,
- $start : S \to \{\mathbf{s}_0, ., \mathbf{s}_k\}$ with $start(G_i) =_{df} \mathbf{s}_i$ for all $i \in \{0, ., k\}$ and
- $end : S \to \{\mathbf{e}_0, ., \mathbf{e}_k\}$ with $end(G_i) =_{df} \mathbf{e}_i$ for all $i \in \{0, ., k\}$.

Intuitively, fg maps every node of a flow graph system to its corresponding flow graph, *callee* every call node to the called procedure, *caller* every procedure to its set of call nodes, and *start* and *end* every procedure to its start node and its end node, respectively.

An illustrative flow graph system is given in Figure 1.

[5] Integrating static procedure nesting is straightforward.

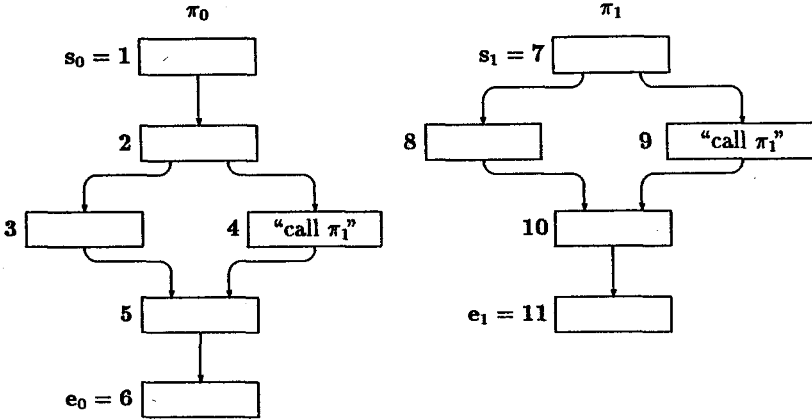

Figure 1: The Flow Graph System

Interprocedural Flow Graphs

The operational approach requires an explicit representation of the interprocedural control flow caused by procedure calls. This is achieved by combining the flow graphs of S to an *interprocedural flow graph* $G^* = (N^*, E^*, s^*, e^*)$, where s^* is given by s_0 and e^* by e_0 (cf. [My, SP]). In detail, G^* results from S by applying the following two step procedure to every node n of $\mathbf{N}_C^S$:

Algorithm 3.1 *Let S be a flow graph system, and $n \in \mathbf{N}_C^S$. Then*

1. *Replace n by two new nodes, the* call node n_C *and the* return node n_R *such that n_C has the same set of predecessors as n but no successors and n_R has the same set of successors as n but no predecessors*[6].

2. *Draw an edge from n_C to $start(callee(n))$ and from $end(callee(n))$ to n_R.*

N_C^* and N_R^* denote the set of all *call nodes* and *return nodes* in N^*, respectively, and $pred^*(n) =_{df} \{ m \mid (m,n) \in E^* \}$ the set of all immediate interprocedural predecessors of n. In the following, we will identify the set $\mathbf{N}^S$ of nodes of S with the set $N^* \setminus N_R^*$ of nodes of G^* to get an interpretation independent notion of program point.

Figure 2 shows the interprocedural flow graph that corresponds to the flow graph system of Figure 1.

Interprocedural Paths

The notion of finite path as introduced in Section 2 naturally applies to interprocedural flow graphs as well. However, due to the special nature of procedure calls not every finite path $p \in \mathbf{P}[m,n]$ represents a valid execution. For example in Figure 2 the path $(2, 4_C, 7, 8, 10, 11, 4_R)$ is possible, while the path $(2, 4_C, 7, 8, 10, 11, 9_R)$ is not. This led to the following definition of interprocedural (valid) paths [SP]:

Definition 3.2 (Interprocedural Path)

1. *Let $p \in \mathbf{P}[m,n]$. Then p is an* interprocedural path *iff the tuple $(m_1, ., m_r)$, which results from p by deleting all nodes in $N^* \setminus (N_C^* \cup N_R^*)$, is well-formed in the following sense:*

   - *if there is no return node in $(m_1, ., m_r)$, then $(m_1, ., m_r)$ is well-formed,*

[6] n_C is called to *match* n_R and vice versa.

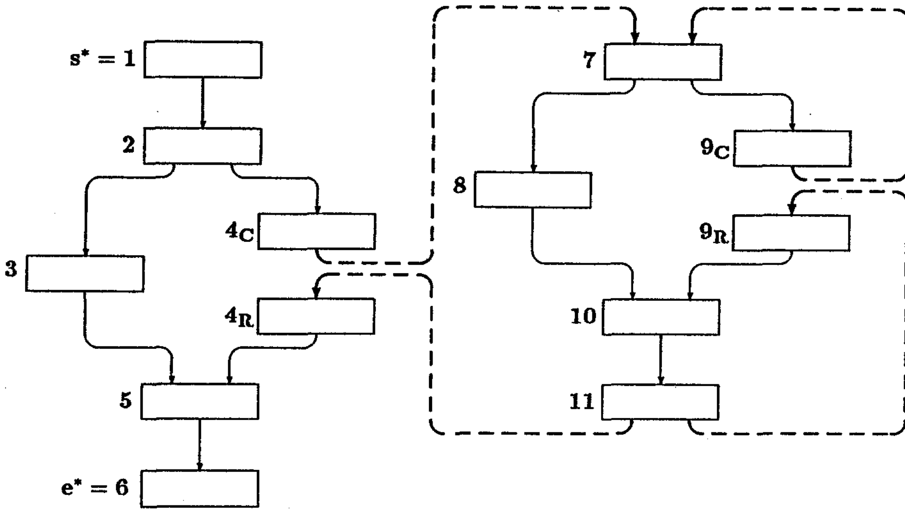

Figure 2: The Interprocedural Flow Graph

- *otherwise, let* j *be the smallest index in* $\{1,.,r\}$ *such that* m_j *is a return node. Then* $(m_1,.,m_r)$ *is well-formed iff* $j>1$ *and* m_{j-1} *is a call node matching* m_j*, and the remaining sequence after deleting* m_{j-1} *and* m_j *is well-formed too.*

2. *A call node and a return node of p are said to* correspond *to each other, if they are eliminated simultaneously in the procedure above.*

3. $\mathrm{IP}[m,n]$ *denotes the set of all* interprocedural paths *from* m *to* n*, and* $\mathrm{IP}[m,n)$ *the set of all interprocedural paths from* m *to a predecessor of* n.

Complete Interprocedural Paths

In order to determine the semantics of procedure calls, we need to deal with complete interprocedural paths p from *start*($fg(n)$) to n, which are characterized by the fact that all procedure calls in p have been completed by a subsequent return. This guarantees that the occurrences of *start*($fg(n)$) and n belong to the same procedure incarnation.

Definition 3.3 (Complete Interprocedural Path)

1. *An interprocedural path* $p=(n_1,\dots,n_k)\in \mathrm{IP}[start(fg(n)),n]$ *is called* complete *if it possesses equally many occurrences of procedure call and return nodes:*

$$|\;\{i \mid n_i \in N_C^*\}\;| = |\;\{i \mid n_i \in N_R^*\}\;|$$

2. $\mathrm{CIP}[start(fg(n)),n]$ *and* $\mathrm{CIP}[start(fg(n)),n)$ *denote the set of all* complete interprocedural paths *from start*($fg(n)$) *to* n*, and from start*($fg(n)$) *to a predecessor of* n*, respectively.*

That this actually realizes our intention is a consequence of the following property of interprocedural paths:

Lemma 3.4 *Let* $p=(n_1,\dots,n_k)\in \mathrm{IP}[m,n]$ *be an interprocedural path and* (n_i,n_j) *and* $(n_{i'},n_{j'})$ *two of its pairs of corresponding call and return nodes. Then the integer intervals* $[i:j]$ *and* $[i':j']$ *are either disjoint or one is included in the other.*

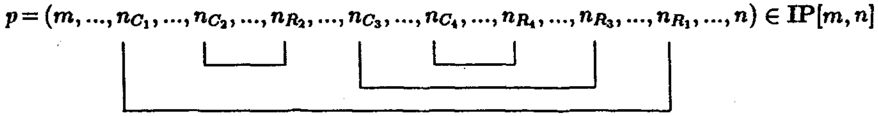

Figure 3: Complete Interprocedural Paths

This pattern is illustrated in Figure 3, where $\{ (n_{C_i}, n_{R_i}) \mid i \in \{1, .., 4\} \}$ are assumed to be pairs of corresponding call and return nodes of p.

The following lemma, which can easily be proved, will be important:

Lemma 3.5 *Let* $\mathbf{s} \in \{\mathbf{s}_0, ..., \mathbf{s}_k\}$, $p = (n_1, ..., n_k) \in \mathbf{IP}[\mathbf{s}, n]$ *and* (n_i, n_j) *a pair of corresponding call and return nodes. Then we have:*

$$(n_{i+1}, ..., n_{j-1}) \in \mathbf{CIP}[start(callee(n_i)), end(callee(n_i))]$$

Remark 3.6 If the underlying program Π has no procedures, the flow graph system S and the interprocedural flow graph G^* collapse to the flow graph G_0 of π_0. In this special case our framework coincides with the standard intraprocedural framework.

Conventions: Throughout the rest of this paper we assume an arbitrary but fixed program $\Pi = (\pi_0, \pi_1, ., \pi_k)$ with flow graph system $S = (G_0, G_1, ., G_k)$ and interprocedural flow graph $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$. Moreover, m and n, possibly indexed, are nodes of S or G^*, and for every node $n \in \mathrm{N}_C^S$, n_C and n_R denote its corresponding call node and return node in N^*, respectively.

4 Abstract Semantics

In this section we present new ***interprocedural*** versions of the meet over all paths strategy and the maximal fixed point strategy. They define the (global) semantics for interprocedural flow graphs and flow graph systems, respectively. The point of this presentation is the extension of the data flow information in a way that mimics run-time stacks as used in run-time systems.

4.1 The Local Semantic Functional

As its intraprocedural counterpart, the interprocedural meet over all paths (*IMOP*) solution directly records all possible program executions that lead to a particular program point. However, in the presence of recursive procedures it is necessary to work on stacks of lattice elements instead of the lattice itself, in order to record the part of the history which will become relevant after returning from (nested) procedure calls. Thus the local semantic functional has type

$$[\![\ ]\!] : N^* \to (STACK \to STACK)$$

where $STACK$ denotes the set of all non empty stacks with components of $\mathcal{C}$, which can be manipulated by means of the following operations:

- *newstack* : $\mathcal{C} \to STACK$
- *push* : $STACK \times \mathcal{C} \to STACK$
- *pop* : $STACK \to STACK$
- *top* : $STACK \to \mathcal{C}$

Intuitively, *newstack*(c) creates a new stack with single component c, *push* puts a new component on the top of the argument stack, *pop* removes the top component, and *top* delivers the content of the top component, while not affecting the argument stack[7]. Thus only the top components of the stacks can be affected by these operations.

STACK is an abstract version of the run-time stacks used by run-time systems for maintaining the activation records of different procedure incarnations. Intuitively, the top component of a stack contains the data flow information corresponding to the currently valid activation record[8], while the data flow informations of the remaining stack components correspond to activation records of preceding but not yet finished procedure calls. However, in contrast to a concrete run-time stack, where variables that are global for the currently activated procedure are accessed by means of static and dynamic link chains, the components of a data flow analysis stack are assumed to contain all information related with the current procedure incarnation, i.e. also the information related to global variables[9]. Thus the data flow analysis stacks directly reflect the nesting of procedure incarnations according to the current call sequence.

Formally, the local semantic functional $[\![\ ]\!]^*$ for this setting is defined by

$\forall n \in N^* \ \forall stk \in STACK.$

$$[\![n]\!]^*(stk) =_{df} \begin{cases} push(\,pop(stk),\ [\![n]\!]'(top(stk))\,) & \text{if } n \in N^* \backslash (N_C^* \cup N_R^*) \\ push(\,stk,\ [\![n]\!]'(top(stk))\,) & \text{if } n \in N_C^* \\ push(\,pop(pop(stk)),\ \mathcal{R}_n(\,top(pop(stk)),\ [\![n]\!]'(top(stk))\,)\,) & \\ & \text{if } n \in N_R^* \end{cases}$$

where $[\![\ ]\!]' : N^* \to (\mathcal{C} \to \mathcal{C})$ denotes the straightforward extension of the semantic functional of Section 2 to interprocedural flow graphs[10], and $\mathcal{R}_n : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a function as described below.

The intuition behind this definition is as follows:

The execution of an ordinary statement (i.e. $n \in N^* \backslash (N_C^* \cup N_R^*)$) only affects the currently valid activation record. Thus it can be modelled by simply modifying the top component of the stack representing the current data flow information.

A procedure call (i.e. $n \in N_C^*$) requires the generation of a new activation record. This is reflected by pushing a new element on the top of the stack, which results from modifying the top component of the stack according to the parameter transfer.

The treatment of return statements (i.e. $n \in N_R^*$) demonstrates the necessity of introducing stacks into the framework. Returning from a procedure call (i.e. $n \in N_R^*$) essentially requires removal of the activation record belonging to the called procedure and reactivation of its predecessor. However, one observation is important here. The effect of a (directly) recursive procedure to a global variable needs to be maintained, whereas the local variables must be reset to their values at call time. Thus we need to consider the data flow information valid immediately before entering the procedure (available in $top(pop(stk))$), as well as the information valid after executing its body (available in $[\![n]\!]'(top(stk))$), in order to compute the data flow information being valid after returning from the called procedure. The function $\mathcal{R}_n : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ models this computation. Thus popping the top component of the stack and replacing the subsequent component by

$$\mathcal{R}_n(\,top(pop(stk)),\ [\![n]\!]'(top(stk)))$$

reflects the whole process of completing a procedure call.

[7] We consider the operation *newstack* instead of the usual *emptystack* : $\to STACK$ here, in order to exclude empty stacks, which are irrelevant in our framework.

[8] Therefore, we are never dealing with empty stacks.

[9] Static and dynamic link chains are just a technical mean for getting efficient implementations of run-time systems. In our abstract framework, however, this aspect can be neglected without any harm (cf. Remark 4.7). Moreover, it allows us to work with local semantic functionals that affect only the top components of data flow analysis stacks.

[10] $[\![n]\!]^*(stk)$, $n \in N_R^*$, is only defined for stacks with at least two components, a fact, which is automatically taken care of in any reasonable analysis context.

4.2 The Structure of the Semantic Functions

Let $\mathcal{F}=_{df}[STACK \rightarrow STACK]$ denote the set of all functions from $STACK$ to $STACK$ and

$$\mathcal{F}_O=_{df}\{ f \in \mathcal{F} \mid \forall\, stk \in STACK.\; pop(f(stk)) = pop(stk) \}$$

$$\mathcal{F}_C=_{df}\{ f \in \mathcal{F} \mid \forall\, stk \in STACK.\; pop(f(stk)) = stk \}$$

$$\mathcal{F}_R=_{df}\{ f \in \mathcal{F} \mid \forall\, stk \in STACK.\; pop(f(stk)) = pop(pop(stk)) \}$$

Then we have:

Lemma 4.1

1. $\forall n \in N^* \backslash (N_C^* \cup N_R^*).\; [\![n]\!]^* \in \mathcal{F}_O$

2. $\forall n \in N_C^*.\; [\![n]\!]^* \in \mathcal{F}_C$

3. $\forall n \in N_R^*.\; [\![n]\!]^* \in \mathcal{F}_R$

Intuitively this means that the semantic function of an ordinary statement only affects the top component of the argument stack, that the semantic function of a call statement simply adds a new top component to the argument stack, and that a return statement replaces the upper two components of the argument stack by a new component. The following lemma is an easy consequence of these properties of $\mathcal{F}_O$, $\mathcal{F}_C$ and $\mathcal{F}_R$.

Lemma 4.2 $\forall f_r \in \mathcal{F}_R\; \forall f_o, f_o' \in \mathcal{F}_O\; \forall f_c \in \mathcal{F}_C.\quad f_o \circ f_o',\; f_r \circ f_o \circ f_c \in \mathcal{F}_O$

The formal development of the paper requires the following derived notions of monotonicity and distributivity:

Definition 4.3 (S-Monotonicity, S-Distributivity)
A function $f \in \mathcal{F}_O \cup \mathcal{F}_C \cup \mathcal{F}_R$ *is called*

- s-monotonic *iff* f_s *is monotonic*
- s-distributive *iff* f_s *is distributive*

where f_s*, the* significant part *of* f*, is defined according to the following two cases:*

- $f \in \mathcal{F}_O \cup \mathcal{F}_C$*: here* $f_s : \mathcal{C} \rightarrow \mathcal{C}$ *is defined by:* $f_s(c) =_{df} top(f(newstack(c)))$
- $f \in \mathcal{F}_R$*: here* $f_s : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ *is defined by*[11]*:* $f_s(c_1, c_2) =_{df} top(f(push(newstack(c_1), c_2)))$

The following lemma shows that the effort for checking the preconditions of the Interprocedural Safety Theorem 5.2 and the Interprocedural Coincidence Theorem 5.3 is comparable to the effort necessary for their intraprocedural counterparts (cf. Section 2 and 5).

Lemma 4.4 *For all* $n \in N^*$ *we have that* $[\![n]\!]^*$ *is s-monotonic (s-distributive) if*

$n \in N_R^*$: $[\![n]\!]'$ *and* $\mathcal{R}_n$ *are monotonic (distributive)*

$n \notin N_R^*$: $[\![n]\!]'$ *is monotonic (distributive)*

Conventions: In the following we consider s-monotonicity (s-distributivity) as a generalization of the usual monotonicity (distributivity) by identifying lattice elements with their unique representations as one-component stacks. Moreover, we extend the meet operation $\sqcap$ to work on stacks in the following way:

$$\forall\, STK \subseteq STACK.\; \sqcap STK =_{df} newstack(\sqcap \{ top(stk) \mid stk \in STK \})$$

Thus, the meet over a set of stacks is just the one-component stack containing the meet of all the top components in its single component.

[11]Note that $\mathcal{C} \times \mathcal{C}$ is a lattice, whenever $\mathcal{C}$ is.

4.3 The Interprocedural Meet Over all Paths Solution

Analogously to Section 2 the local abstract semantics $[\![\ ]\!]^*$ can be extended to cover finite interprocedural paths. For every path $p=(n_1,...,n_q) \in \mathbf{IP}[m,n]$, we define $[\![p]\!]^* : STACK \rightarrow STACK$ by

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

Now, as its intraprocedural counterpart (cf. [KU]), the interprocedural meet over all paths (*IMOP*) solution directly records all possible program executions leading to a particular program point. Here it is important that for any interprocedural path $p \in \mathbf{IP}[s^*,n)$ and any stack $stk \in STACK$, $top([\![p]\!]^*(stk))$ is the only data flow information relevant for node n after executing p, since all other components of $[\![p]\!]^*(stk)$ correspond to activation records that are not valid after p. Identifying one-component stacks with the content of their unique component, the formal definition of the interprocedural meet over all paths solution is given by:

The *IMOP*-Solution: $\forall n \in N^* \ \forall c_0 \in \mathcal{C}.\ IMOP_{c_0}(n) = \bigsqcap \{ [\![p]\!]^*(newstack(c_0)) \mid p \in \mathbf{IP}[s^*,n) \}$

4.4 The Interprocedural Maximal Fixed Point Solution

In addition to the equational characterization of the intraprocedural case (Equation System 2.1), flow graph systems need a preprocess, which determines the meaning of call nodes in terms of the meaning of the called procedures. This requires the introduction of an auxiliary semantic functional $[\![\![\ ]\!]\!]$, which gives meaning to whole flow graphs. Essentially, $[\![\![n]\!]\!]$ transforms data flow information that is assumed to be valid at the entry of the procedure that contains n into the corresponding data flow information being valid before an execution of n. In particular, $[\![\![e_i]\!]\!]$ is the meaning function of the i-th procedure[12]. Formally, the full preprocess for determining the meaning $[\![n]\!]$ of call nodes $n \in \mathbf{N}_C^S$ is characterized by:

Definition 4.5 *$[\![\![\ ]\!]\!] : \mathbf{N}^S \rightarrow (STACK \rightarrow STACK)$ and $[\![\ ]\!] : \mathbf{N}^S \rightarrow (STACK \rightarrow STACK)$ are defined as the greatest solution of the equation system given by:*

$$[\![\![n]\!]\!] =_{df} \begin{cases} id_{STACK} & \text{if } n \in \{s_0,..,s_k\} \\ \bigsqcap\{[\![m]\!] \circ [\![\![m]\!]\!] \mid m \in pred_{fg(n)}(n)\} & \text{otherwise} \end{cases}$$

and

$$[\![n]\!] =_{df} \begin{cases} [\![n]\!]^* & \text{if } n \in \mathbf{N}^S \backslash \mathbf{N}_C^S \\ [\![n_R]\!]^* \circ [\![\![end(callee(n))]\!]\!] \circ [\![n_C]\!]^* & \text{otherwise} \end{cases}$$

where id_{STACK} denotes the identity on $STACK$, and $\bigsqcap$ the "componentwise" meet operation on $\mathcal{F}_O$.[13]

The effect of a procedure call $n \in \mathbf{N}_C^S$ is determined in three steps reflecting the three phases of its execution:

- *Entering* the called procedure: $[\![n_C]\!]^*$ creates a new activation record by transforming the content of the top component of the stack according to the semantics of the call node and pushing it onto the stack. – Usually, the semantics of call nodes will reflect the parameter transfer.

[12] Remember, $[\![e_i]\!]' =_{df} id_{\mathcal{C}}$. Thus, e_i is related to the identity on $STACK$.

[13] $\forall f, f' \in \mathcal{F}_O.\ f \sqcap f' =_{df} f'' \in \mathcal{F}_O$ with $\forall stk \in STACK.\ top(f''(stk)) = top(f(stk)) \sqcap top(f'(stk))$. As usual, "$\sqcap$" induces an inclusion relation "$\sqsubseteq$" on $\mathcal{F}_O$ by: $f \sqsubseteq f'$ iff $f \sqcap f' = f$.

- *Evaluating* the call: $[\![\, end(callee(n)) \,]\!]$ computes the effect of the procedure body. Note that this affects the top component of the argument stack only.
- *Leaving* the called procedure: $[\![\, n_R \,]\!]^*$ removes the activation record related with the current procedure call by popping the top component from the stack, and replacing its subsequent component by the data flow information representing the effect of the procedure call relative to its call site.

Applying Lemma 4.2, we obtain

Lemma 4.6 $\forall n \in \mathrm{N}^S. \; [\![n]\!], \; [\![n]\!] \in \mathcal{F}_O$

Remark 4.7 Lemma 4.6 is important, since it shows that all the stacks occurring during the iterative computation of the *IMFP*-solution will have at most two components[14]. This is in contrast to the *IMOP*-strategy, where the size of stacks contributing to the *IMOP*-solution is in general unbounded. Moreover, it allows us to prove termination in the usual way.

After fixing the meaning of call nodes, $[\![\;]\!]$ plays essentially the same role as the local (abstract) semantic functional of Section 2. Formally, the interprocedural maximal fixed point strategy is characterized by Equation System 4.8. As its intraprocedural counterpart, this strategy labels every node n of N^S with a pre-information $\mathbf{pre}_{c_0}(n)$ and a post-information $\mathbf{post}_{c_0}(n)$, whose top components are the greatest solution of this equation system with respect to $c_0 \in \mathcal{C}$.

Equation System 4.8

$$\mathbf{pre}(n) = \begin{cases} newstack(c_0) & \text{if } n = \mathrm{s}_0 \\ \bigsqcap \{ [\![m_C]\!]^*(\mathbf{pre}(m)) \,|\, m \in caller(fg(n)) \} & \text{if } n \in \{\mathrm{s}_1, .., \mathrm{s}_k\} \\ \bigsqcap \{ \mathbf{post}(m) \,|\, m \in pred_{fg(n)}(n) \} & \text{otherwise} \end{cases}$$

$$\mathbf{post}(n) = [\![n]\!](\mathbf{pre}(n))$$

As before, identifying a stack having a single component only with the content of this component, we obtain as in the intraprocedural case:

The *IMFP*-Solution: $\forall n \in \mathrm{N}^S \; \forall c_0 \in \mathcal{C}. \; IMFP_{c_0}(n) = \mathbf{pre}_{c_0}(n)$

5 Main Results

The main step in the proof of our main results is taken by proving the following Main Lemma 5.1, whose proof is given in full detail in [KS1].

Lemma 5.1 (The Main Lemma)
For all $n \in \mathrm{N}_C^S$, we have, if the semantic functions $[\![m]\!]^*$, $m \in N^*$, are

1. *s-monotonic:* $[\![n]\!] \sqsubseteq \bigsqcap \{ [\![p]\!]^* \,|\, p \in \mathbf{CIP}[n_C, n_R] \}$
2. *s-distributive:* $[\![n]\!] = \bigsqcap \{ [\![p]\!]^* \,|\, p \in \mathbf{CIP}[n_C, n_R] \}$

[14]This is because the computation starts from a one-component stack $newstack(c_0)$ (cf. Equation System 4.8 and Algorithm A.3).

After having established this result, the Interprocedural Safety Theorem 5.2 and the Interprocedural Coincidence Theorem 5.3 can be proved almost as in the intraprocedural case. Thus we omit these proofs here[15].

As in the intraprocedural case, the first theorem states that the *IMFP*-solution is a correct approximation of the *IMOP*-solution, whenever all the local abstract semantic functions are s-monotonic:

Theorem 5.2 (Interprocedural Safety Theorem)
Given a flow graph system $S=(G_0,G_1,.,G_k)$ *and its corresponding interprocedural flow graph* $G^*=(N^*,E^*,\mathbf{s}^*,\mathbf{e}^*)$*, the IMFP-solution is a correct approximation of the IMOP-solution, i.e.* $\forall n \in \mathrm{N}^S\ \forall c_0 \in \mathcal{C}.\ IMFP_{c_0}(n) \sqsubseteq IMOP_{c_0}(n)$*, if the abstract semantics* $[\![n]\!]^*$ *of all nodes* $n \in N^*$ *is given by an s-monotonic function.*

Again, as in the intraprocedural case, s-distributivity of the semantic functions yields optimality (or completeness):

Theorem 5.3 (Interprocedural Coincidence Theorem)
Given a flow graph system $S=(G_0,G_1,.,G_k)$ *and its corresponding interprocedural flow graph* $G^*=(N^*,E^*,\mathbf{s}^*,\mathbf{e}^*)$*, the IMFP-solution and the IMOP-solution coincide, i.e.* $\forall n \in \mathrm{N}^S\ \forall c_0 \in \mathcal{C}.\ IMOP_{c_0}(n)=IMFP_{c_0}(n)$*, if the abstract semantics* $[\![n]\!]^*$ *of all nodes* $n \in N^*$ *is given by an s-distributive function.*

Note that Lemma 4.4 allows to check the s-monotonicity or s-distributivity of the semantic functions $[\![n]\!]^*$ simply by checking these properties for the semantic functions $[\![n]\!]'$ and the reduction functions $\mathcal{R}_n$. Thus the only additional effort in comparison to the intraprocedural case arises from checking the reduction functions.

6 Applications

In this section we sketch two applications of the Interprocedural Coincidence Theorem 5.3. We omit details here, since both examples require their own setup.

In [SK2] we propose an algorithm for *interprocedural constant propagation* and *constant folding*, which generalizes and improves all previous techniques for interprocedural constant propagation (cf. [CC2, CCKT, JM]). This algorithm determines all *finite interprocedural constants*, which are the interprocedural analogue to the set of finite constants introduced in [SK1]. As in the intraprocedural case, finite interprocedural constants have a purely *operational* characterization in the sense of the *IMOP*-strategy, and a purely *denotational* characterization in the sense of the *IMFP*-strategy. The Interprocedural Coincidence Theorem 5.3 yields the equivalence of these characterizations.

The second example concerns the interprocedural versions of the classical bit-vector data flow analyses, e.g. determining available expressions, reaching definitions, live variables, very busy (anticipatable) expressions (cf. [He])[16], and, more sophisticatedly, the optimal elimination of interprocedural partial redundancies[17]. In all these cases, the Interprocedural Coincidence Theorem 5.3 allows us to prove the optimality of our algorithms for programs with recursive procedures, global and local variables, and formal value parameters [KS2][18].

[15] Both proofs are given in [KS1].
[16] This application has also been suggested (without giving any details) by Sharir and Pnueli [SP].
[17] This problem is heuristically dealt with in [Mo, MR].
[18] A detailed presentation of interprocedural bit-vector data flow analyses is given in [KS3].

7 Conclusions

We have presented an interprocedural generalization of the well-known intraprocedural Coincidence Theorem of Kam and Ullman [KU], which covers arbitrary programs with recursive procedures, global and local variables, and formal value parameters. Our theorem, which reduces to the classical intraprocedural version in the absence of procedures, delivers a sufficient condition for the coincidence of the ***interprocedural meet over all paths*** strategy and the ***interprocedural maximal fixed point*** strategy, and it generalizes previous results (cf. [Ba1, Ba2, SP]), which do not deal properly with local variables of recursive procedures. Our results are formulated within the framework of abstract interpretation, thus covering a wide range of data flow analyses.

References

[All] Allen, F. E. Control flow analysis. *SIGPLAN Not. 5*, 7 (1970), 1 - 19.

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

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

[Bo] Bourdoncle, F. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Proceedings 2nd PLILP*, Linköping, Sweden, Springer-Verlag, LNCS 456 (1990), 307 - 323.

[CC1] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4th POPL*, Los Angeles, California, 1977, 238 - 252.

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

[CCKT] Callahan, D., Cooper, K. D., Kennedy, K. W., and Torczon, L. M. Interprocedural constant propagation. In *Proceedings SIGPLAN'86 Symp. on Compiler Construction*, SIGPLAN Not. *21*, 7 (1986), 152 - 161.

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

[JM] Jones, N. D., and Muchnick, S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings 9th POPL*, Albuquerque, New Mexico, 1982, 66 - 74.

[Ki] Kildall, G. A. A unified approach to global program optimization. In *Proceedings 1st POPL*, Boston, Massachusetts, 1973, 194 - 206.

[KS1] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. Aachener Informatik-Berichte Nr. 91-27, Rheinisch-Westfälische Technische Hochschule Aachen, Aachen, Germany, 1991.

[KS2] Knoop, J., and Steffen, B. Optimal interprocedural partial redundancy elimination. In *Addenda to Proceedings 4th CC*, Paderborn, Germany, October 5-7, 1992. Technical Report, Department of Computer Science, University of Paderborn, Germany, 1992.

[KS3] Knoop, J., and Steffen, B. Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework. To appear.

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

[La] Langmaack, H. On procedures as open subroutines. Part I. *Acta Informatica 2*, (1973), 311 - 333.

[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 *Proceedings 8th POPL*, 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.

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

[SK1] Steffen, B., and Knoop, J. Finite constants: Characterizations of a new decidable set of constants. In *Proceedings 14th MFCS*, Porąbka-Kozubnik, Poland, Springer-Verlag, LNCS 379 (1989), 481 - 491. An extended version appeared in: *Theoretical Computer Science 80*, 2 (1991), 303 - 318.

[SK2] Steffen, B., and Knoop, J. Finite interprocedural constants. To appear.

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

A Algorithms

This section provides algorithms that compute the *IMFP*-solution. First, we have an algorithm for the preprocess that determines the semantics of call nodes:

Algorithm A.1 (Computation of the Semantic Functionals $[\![\ ]\!]$ and $[\![\![\ ]\!]\!]$)

Input: *A flow graph system* $S = (G_0, G_1, ., G_k)$, *a complete semi-lattice* C, *for every node* $n \in N^S \backslash N^S_C$ *an s-monotonic function* $[\![n]\!]^* : STACK \rightarrow STACK \in \mathcal{F}_O$, *which is the identity for all nodes in* $\{s_0, ..., s_k, e_0, ..., e_k\}$. *Moreover, for every node* $n \in N^S_C$ *two s-monotonic functions* $[\![n_C]\!]^* : STACK \rightarrow STACK \in \mathcal{F}_C$ *and* $[\![n_R]\!]^* : STACK \rightarrow STACK \in \mathcal{F}_R$. *All semantic functions* $[\![n]\!]^*$, $n \in N^* \backslash N^*_R$, *are assumed to map stacks with top component* $\top$ *to stacks with top component* $\top$. *Analogously, all semantic functions* $[\![n]\!]^*$, $n \in N^*_R$, *are assumed to map stacks with upper two components* $\top$ *to stacks with top component* $\top$.

Output: *An annotation of* S *with functions* $[\![\![n]\!]\!] : STACK \rightarrow STACK$ *(stored in gtr) and* $[\![n]\!] : STACK \rightarrow STACK$ *(stored in ltr) that satisfy Definition 4.5.*

Remark: $\top_{\mathcal{F}_O} : STACK \rightarrow STACK \in \mathcal{F}_O$ *denotes the "universal" function which is assumed to "contain" every function* $f \in \mathcal{F}_O$, *and* id_{STACK} *is the identity on* $STACK$. *The variable workset controls the iterative process. Its elements are tuples, whose first components are nodes* $m \in N^S$ *of the flow graph system* S, *and whose second components are functions* $f : STACK \rightarrow STACK \in \mathcal{F}_O$

that 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 semantics* $[\ ]$ *of call nodes.*

```
(Initialization of the annotation arrays gtr and ltr and the variable workset)
FORALL m ∈ N^S DO
   gtr[m] := ⊤_{F_O};
   IF m ∈ N^S_C THEN ltr[m] := ⊤_{F_O} ELSE ltr[m] := [ m ]* FI
OD;
workset := { (s, id_STACK) | s ∈ {s_0, ..., s_k} };

(Iterative fixed point computation)
WHILE workset ≠ ∅ DO
   LET (m, f) ∈ workset
      BEGIN
         workset := workset \ { (m, f) };
         IF gtr[m] ⊐ gtr[m] ⊓ f
            THEN
               gtr[m] := gtr[m] ⊓ f;
               IF m ∈ {e_i | i ∈ {0, ., k}}
                  THEN
                     FORALL l ∈ caller(fg(m)) DO
                        ltr[l] := [[ l_R ]]* ∘ gtr[m] ∘ [[ l_C ]]*;
                        workset := workset ∪ {(n, ltr[l] ∘ gtr[l]) | n ∈ succ_{fg(l)}(l) }
                     OD
                  ELSE
                     workset := workset ∪ { (n, ltr[m] ∘ gtr[m]) | n ∈ succ_{fg(m)}(m) }
               FI
         FI
      END
OD.
```

In order to simplify the formulation of the central property of this algorithm, we abbreviate the values of $ltr[n]$ and $gtr[n]$ after the k-th execution of the while-loop by $ltr^k[n]$ and $gtr^k[n]$, respectively. The following theorem can now be proved in a straightforward fashion (cf. [Ki]):

Theorem A.2 $\forall n \in N^S.\ [\ n\ ] = \sqcap\{ltr^k[n] \mid k \geq 0\} \quad \wedge \quad [\![n]\!] = \sqcap\{gtr^k[n] \mid k \geq 0\}$
In particular, we have $\forall n \in N^S.\ [\![n]\!] = gtr[n] \ \wedge\ [\ n\ ] = ltr[n]$ *after termination of Algorithm A.1.*

The second algorithm computes the *IMFP*-Solution:

Algorithm A.3 (Computation of the *IMFP*-Solution)

Input: *A flow graph system* $S = (G_0, G_1, ., G_k)$, *the semantic functional* $[\![\ ]\!]$, *for every node* $n \in N^S_C$ *the function* $[\![n_C]\!]^*$, *and a start information* $c_0 \in C$. *All semantic functions* $[\![n]\!]$ *and* $[\![n_C]\!]^*$ *are assumed to be s-monotonic and to map stacks with top component* $\top$ *to stacks with top component* $\top$.

Output: *An annotation of S with data flow informations, i.e. an annotation with pre-informations (stored in pre) and post-informations (stored in post) of one-component stacks that characterize valid data flow information at the entry and at the exit of every node.*

Remark: *newstack*$(\top)$ *denotes the "universal" data flow information, which is assumed to "contain" every data flow information. The variable workset controls the iterative process. Its elements are tuples whose first components are nodes* $m \in N^S$ *of the flow graph system* S *and whose second components are elements of STACK specifying a new approximation for the pre-information of the node of the first component.*

```
(Initialization of the annotation arrays pre and post and the variable workset)
FORALL m ∈ N^S DO (pre[m], post[m]) := ( newstack(⊤), newstack(⊤) ) OD;
workset := { (s_0, newstack(c_0)) };

(Iterative fixed point computation)
WHILE workset ≠ ∅ DO
   LET (m, stk) ∈ workset
     BEGIN
       workset := workset \ { (m, stk) };
       IF pre[m] ⊐ pre[m] ⊓ stk
         THEN
           pre[m] := pre[m] ⊓ stk;
           post[m] := [[ m ]](pre[m]);
           workset := workset ∪ { (n, post[m]) | n ∈ succ_{fg(m)}(m) };
           IF m ∈ N_C^S
             THEN workset := workset ∪ { (start(callee(m)), [[ m_C ]]*(pre[m])) }
           FI
       FI
     END
OD.
```

As before, given a start information c_0, we abbreviate the values of *pre*$[n]$, and *post*$[n]$ after the k-th execution of the while-loop by $pre^k[n]$, and $post^k[n]$. In analogy to Theorem A.2 we have:

Theorem A.4 $\forall n \in N^S.\, IMFP_{c_0}(n) \;=\; \bigsqcap\{pre^k[n] \mid k \geq 0\}$
In particular, we have $\forall n \in N^S.\ IMFP_{c_0}(n) = pre[n]$ *after termination of Algorithm A.3.*