# Data-Flow Analysis for Multi-Core Computing Systems: A Reminder to Reverse Data-Flow Analysis

Jens Knoop[1]

Institute of Computer Languages
TU Vienna, 1040 Vienna, Argentinierstr. 8 / E185.1, Austria
knoop@complang.tuwien.ac.at

**Abstract.** The increasing demands for highly performant, proven correct, easily maintainable, extensible programs together with the continuous growth of real-world programs strengthen the pressure for powerful and scalable program analyses for program development and code generation. Multi-core computing systems offer new chances for enhancing the scalability of program analyses, if the additional computing power offered by these systems can be used effectively. This, however, poses new challenges on the analysis side. In principle, it requires program analyses which can be easily parallelized and mapped to multi-core architectures. In this paper we remind to *reverse data-flow analysis*, which has been introduced and investigated in the context of demand-driven data-flow analysis, as one such class of program analyses which is particularly suitable for this.

**Keywords.** Multi-core computing systems, scalable program analysis, reverse data-flow analysis, demand-driven data-flow analysis

## 1 Motivation

Functional and non-functional program properties such as performance, maintainability, extensibility, reusability, the demand for proven correctness (of at least certain aspects) of a program and the continuously fast growth of the size of real-world programs push the need for ever more powerful and scalable program analyses.

Against this background, the advent and the growing dissemination of multi-core computing systems offers appealing new chances for improving the scalability of program analyses. This, however, poses new challenges on the structure of program analyses in order to exploit the additional computing power offered by multi-core computing systems effectively. In principle, this requires program analyses which can be easily parallelized and mapped to multi-core architectures. In this paper we remind to *reverse data-flow analysis*, which has been introduced and investigated in the context of *demand-driven data-flow analysis*, as one such class of program analyses which we consider particularly suitable for this.

Dagstuhl Seminar Proceedings 08161
Scalable Program Analysis
http://drops.dagstuhl.de/opus/volltexte/2008/1575

While conventional data-flow analysis (DFA) can be considered a whole program analysis as it computes a property of interest for every program point, demand-driven data-flow analysis (DD-DFA) does so for a specifically selected program point of interest only. This is called a *data-flow query*, which can usually be answered very efficiently as often only a small portion of the program needs to be investigated.

For illustration consider the example of Figure 1, which we recall from [1]. In this example the green and pink highlighted program regions suggest that detecting the redundancy of the computation of $a + b$ in the left loop, which allows the replacement of this computation by a reference to variable $x$, and the non-redundancy of the computation of $c+b$ in the right loop, which prevents such a replacement, does not require to analyze the whole and possibly huge program but that the analysis of the program can be restricted to the highlighted program parts, i.e., the program parts which actually determine the information at the program points of interest. In this example, these are the source nodes of the edges $e$ and $f$ with the use sites of $a + b$ and $c + b$ in the left and in the right loop, respectively. Note also that the example suggests that the two data-flow queries can be processed in parallel, e.g. by two different cores of a multi-core computing system.

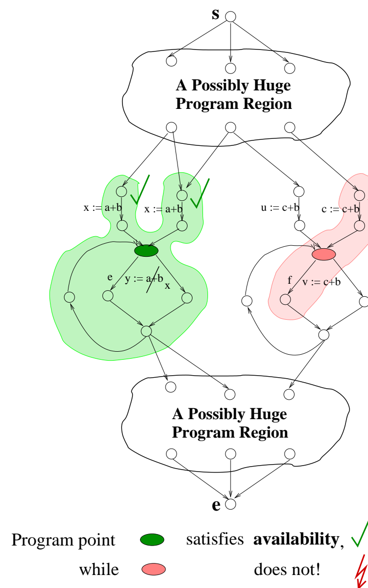


**Fig. 1.** Motivating Example

Demand-driven DFA aims at answering such queries efficiently (cf. [2,3,4,5,6,7,8,9,10,11,12]). Particularly successful in practice turned out be an approach for DD-DFA based on *reverse* data-flow analysis (RDFA), which has

been pioneered by Duesterwald et al. (cf. [5,7,8]). In contrast to other approaches to DD-DFA, which are designed and tailored for a specific DFA problem, e.g., the construction of the call-graph of an object-oriented program as in [2], the approach of Duesterwald et al. offers a framework supporting the construction of demand-driven analyses for various problems.

In this paper we reconsider reverse data-flow analysis focusing on two main issues. First, on the duality of reverse data-flow analysis and classical data-flow analysis and their precise relationships, which is commonly left informally and implicitly in part in previous work. Second, on the suitabability of RDFA-based DD-DFA for parallelization, which suggests its appropriateness for multi-core computing systems. In spirit, the current paper is thus in line with the works of [1] and [13] for an intraprocedural sequential setting and an intraprocedural parallel setting allowing parallel programs with a fork/join-parallelism. In this paper we consider an interprocedural setting.

We will present the interprocedural version of the *Reverse Safety and Coincidence Theorem*, which characterizes the correctness and precision of interprocedural reverse DFA and complements its well-known counterpart of classical interprocedural DFA, the Interprocedural Safety and Coincidence Theorem of Sharir and Pnueli [14]. The *Reverse Interprocedural Safety and Coincidence Theorem* highlights the duality of classical and reverse DFA for the interprocedural setting of [14]. Moreover, we will present the interprocedural version of the *Link Theorem* (cf. [13,1]). This theorem complements its counterparts for the intraprocedural and parallel settings of [1] and [13]. The *Interprocedural Link Theorem* captures the relationship between classical and reverse interprocedural DFA. Together the *Reverse Interprocedural Safety and Coincidence Theorem* and the *Interprocedural Link Theorem* constitute the formal foundation for constructing interprocedural demand-driven data-flow analyses based on reverse interprocedural DFA. Particularly important in practice is the large class of bit-vector data-flow analyses. For this class of analyses, the results of classical interprocedural DFA can be computed by means of their demand-driven reverse counterparts. Moreover, these analyses are tailored for parallelization and thus for usage on multi-core computing systems.

Together this provides the key to scalable implementations of a variety of powerful and widely used optimizations such as redundancy elimination, dead-code elimination, constant propagation, and array bounds check elimination [4,5,7,6,8,13,1,15] on multi-core computing systems.

## 2 Preliminaries

We consider the interprocedural setting, which has been introduced by Sharir and Pnueli in their pioneering work on interprocedural data-flow analysis [14]. In this setting, programs consist of a finite number of procedures without parameters and global variables only. The procedures of a program can be (mutually) recursive and statically nested, and each program is assumed to have a unique

*main procedure*. This is a distinct procedure, which is executed on calling the program and cannot be called by other procedures.

As in [14], we represent procedures by flow graphs, and programs with procedures by flow-graph systems and interprocedural flow graphs.

*Flow Graphs.* A *flow graph* is a directed graph $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set $N$, edge set $E$, and a unique *start node* $\mathbf{s}$ and *end node* $\mathbf{e}$, which are assumed to be free of incoming and outgoing edges, respectively (cf. Figure 1). We assume that the nodes of a flow graph represent the program points, and the edges the (elementary) statements (assignments, etc.) and the control flow of the underlying program. Edges without a label are assumed to represent the empty statement "skip." As usual the control flow is nondeterministically interpreted in order to avoid undecidabilities.

For a node $n$ and an edge $e$ of a flow graph, $pred(n)$ and $succ(n)$ denote the set of all immediate predecessors and successors of $n$, and $src(e)$ and $dst(e)$ the source node and the destination node of $e$. A *finite path* in $G$ is a sequence of edges $\langle e_1, \ldots, e_q \rangle$, where $dst(e_j) = src(e_{j+1})$ for $j \in \{1, \ldots, q-1\}$. It is a path from $m$ to $n$, if $src(e_1) = m$ and $dst(e_q) = n$. The set of all finite paths from $m$ to $n$ is denoted by $\mathbf{P}[m, n]$. Without losing generality we assume that each node of a flow graph $G$ lies on a path from its entry $\mathbf{s}$ to its exit $\mathbf{e}$.

*Flow-Graph Systems.* A *flow-graph system* $S =_{df} \langle G_0, \ldots, G_k \rangle$ represents each procedure of a program by a flow graph in the sense of the previous paragraph (cf. Figure 2). We assume that the sets of nodes $N_i$ and edges $E_i$, $0 \leq i \leq k$, are pairwise disjoint and that $G_0$ represents the main procedure of the underlying program. For brevity, we usually write $\mathbf{s}$ and $\mathbf{e}$ instead of $\mathbf{s}_0$ and $\mathbf{e}_0$. By $N =_{df} \bigcup \{ N_i \mid i \in \{0, \ldots, k\}\}$ and $E =_{df} \bigcup \{ E_i \mid i \in \{0, \ldots, k\}\}$ we denote the sets of all nodes and edges of a flow-graph system. Additionally, we denote the set of *call edges* of $S$, i.e., the subset of edges representing a procedure call, by $E_{call} \subseteq E$.

*Interprocedural Flow Graphs.* The *interprocedural flow graph* is derived from a flow-graph system $S$ by melting the flow graphs of $S$ into a single graph $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$. This graph results from $S$ by replacing each call edge $e$ of $S$ by a *call edge* $e_c$ and a *return edge* $e_r$. The call edge $e_c$ connects the source node of $e$ with the start node of the flow graph called; the return edge $e_r$ connects the end node of this flow graph with the destination node of $e$ (cf. Figure 2). In the following we denote the set of all call and return edges of $G^*$ by $E^*_c$ and $E^*_r$, respectively. Moreover, $E^*_{call} =_{df} E^*_c \cup E^*_r$ denotes the union of call and return edges.

*Interprocedural Paths.* The notion of a path of a flow graph can be extended to interprocedural flow graphs. While, however, every path connecting two nodes of an intraprocedural flow graph is valid in the sense of representing a possible run-time execution (up to non-determinism), this does not hold for interprocedural flow graphs. In order to be valid, interprocedural paths need to respect the

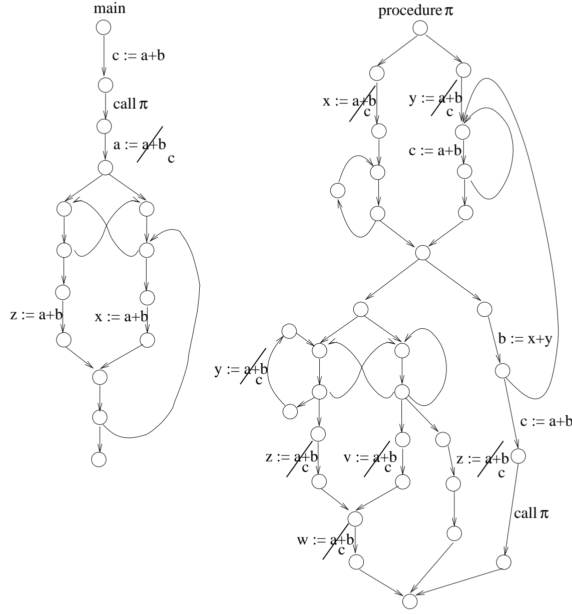


**Fig. 2.** Flow-Graph System

call/return behaviour of procedure calls (cf. [14]). Identifying matching call and return edges in $G^*$ with opening and closing parentheses "(" and ")," respectively, the set of interprocedurally valid paths corresponds to the prefix-closed language of balanced parentheses (cf. [16]). Hence, considering the sequence of edges of a path a word of a formal language, the set of intraprocedurally valid paths forms a regular language, while the set of interprocedurally valid paths forms a context-free language.[1] In the following we denote the set of interprocedurally (valid) paths connecting two nodes $m$ and $n$ by $\mathbf{IP}[m, n]$.

*Complete Interprocedural Paths.* Complete interprocedural paths are important for capturing the global abstract semantics of procedure calls. An interprocedural path leading from the start node $\mathbf{s}_i$ of a procedure $G_i$, $i \in \{0, \ldots, k\}$, to a node $n$ located inside $G_i$ is called *complete*, if each procedure call occurring on $p$ is completed by a subsequent return. Intuitively, this ensures that the occurrences of $\mathbf{s}_i$ and $n$ belong to the same incarnation of the procedure underlying $G_i$. It is worth noting that the subpaths of a complete interprocedural path corresponding to a procedure call are either disjoint or properly nested. We denote the set of all complete paths leading from $\mathbf{s}_i$ to $n$ by $\mathbf{CIP}[\mathbf{s}_i, n]$.

[1] In [14] an algorithmic definition of interprocedurally valid paths is provided, in [17] a definition in terms of a context-free language.

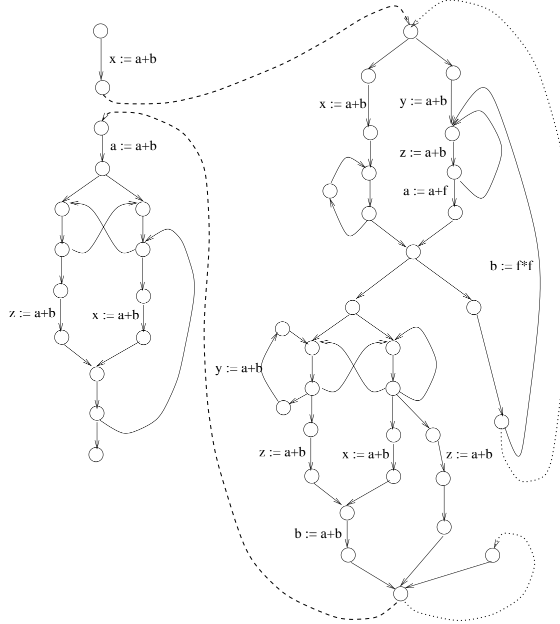


**Fig. 3.** Interprocedural Flow Graph

*Further Notations.* For a flow-graph $S$, the polymorphic function *flowGraph* : $N \cup E \rightarrow S$ maps the nodes and edges of $S$ to the flow graph containing them; the function *callee* : $E_{call} \rightarrow S$ maps every call edge to the flow graph representing the called procedure; the function *caller* : $S \rightarrow \mathcal{P}(E_{call})$ maps every flow graph to the set of edges calling it, and the functions *start* : $S \rightarrow \{\mathbf{s}_0, \ldots, \mathbf{s}_k\}$ and *end* : $S \rightarrow \{\mathbf{e}_0, \ldots, \mathbf{e}_k\}$ map the flow graphs of $S$ to their start nodes and end nodes, respectively. These functions will be helpful for the formal development in this paper.

## 3 (Classical) Interprocedural Data-Flow Analysis

Intuitively, DFA aims at computing information about the run-time behaviour of a program at compile time. This information is typically modelled by the elements of an appropriate complete lattice $\mathcal{C}$. Together with a *data-flow functional* $[\![\ ]\!]' : E^* \rightarrow \mathcal{C} \rightarrow \mathcal{C}$, which assigns abstract meaning to the elementary statements of a program, the lattice of *data-flow facts* $\mathcal{C}$ forms an *abstract semantics* of the program, which is tailored for the problem under consideration (cf. [18]). For the parameterless setting considered here, the call and return edges of $G^*$ are associated with the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$.

In the following, we assume that the top element of a data-flow lattice represents "unsatisfiable (inconsistent)" information, and that it is invariant under

each data-flow function. Apparently, such an element can always be added, if necessary. This assumption ensures that the reverse data-flow functions of a DFA problem are always well-defined (cf. Section 4).

Given a data-flow functional $[\![\ ]\!]'$, and a data-flow fact $c_{\mathbf{s}}$, the task of DFA can conceptually be considered to compute an annotation function $ann : N \to \mathcal{C}$ annotating each node $n$ of the flow-graph system $S$ with a data-flow fact $c$ reflecting and respecting the constraints given by $[\![\ ]\!]'$ and $c_{\mathbf{s}}$. In particular, the annotation function shall satisfy $ann(\mathbf{s}) = c_{\mathbf{s}}$.

In the following we recall the essence of the interprocedural data-flow framework of Sharir and Pnueli to accomplish this [14]. The key of this approach is to extend the intraprocedural versions of the *meet-over-all-paths* (*MOP*) approach and the *maximal-fixed-point* (*MaxFP*) approach (cf. [19]) to the interprocedural setting. We recall these two approaches next.

***The IMOP-Approach.*** The definition of the *interprocedural meet-over-all-paths* (*IMOP*) approach and the annotation function it induces results from the inductive extension of an abstract semantics to (finite) interprocedural paths $p$, which is straightforward. For a start information $c_{\mathbf{s}} \in \mathcal{C}$ and a node $n \in N^*$, the *IMOP*-solution is defined by:

$$\textit{IMOP}\textbf{-Solution: } \textit{IMOP}_{c_{\mathbf{s}}}(n) =_{df} \bigsqcap \{\, [\![\, p \,]\!]'(c_{\mathbf{s}}) \mid p \in \mathbf{IP}[\mathbf{s}, n] \,\}$$

***The IMaxFP-Approach.*** In contrast to the *IMOP*-approach, which has an operational flavour, the *interprocedural maximal-fixed-point* (*IMaxFP*) approach is characterized by two dependent equation systems. Intuitively, the greatest solution of the first equation system captures the abstract semantics of the procedures of a program; based on this solution, the greatest solution of the second equation system defines the annotation function of the overall approach.

**Equation System 1 (Semantics of Procedures (2nd-Order))**

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

*and*

$$[\![\, e \,]\!] = \begin{cases} [\![\, e \,]\!]' & \text{if } e \in E \backslash E_{call} \\ [\![\, end(caller(e)) \,]\!] & \text{otherwise} \end{cases}$$

Note that the functions $[\![\, e \,]\!]$, $e \in E_{call}$, of the greatest solution of Equation System 1 describe the semantics of procedure call edges (cf. Main Lemma 1). Denoting the greatest solution of Equation System 1 by $[\![\ ]\!]$ and $[\![\ ]\!]$, too, the annotation function, which is induced by the overall approach, is then given by the greatest solution of Equation System 2, denoted by $\textsf{inf}^{\,*}_{c_{\mathbf{s}}}$, where **inf** reminds to *data-flow fact*.

**Equation System 2 (The *IMaxFP*-Equation System)**

$$\textsf{inf}\,(n) = \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s}_0 \\ \bigsqcap \{\, \textsf{inf}\,(src(e)) \mid e \in caller(flowGraph(n)) \,\} & \text{if } n \in \{\mathbf{s}_1, \ldots, \mathbf{s}_k\} \\ \bigsqcap \{\, [\![\, (m,n) \,]\!](\textsf{inf}\,(m)) \mid m \in pred_{flowGraph(n)}(n) \,\} & \text{otherwise} \end{cases}$$

$$\textit{IMaxFP}\textbf{-Solution: } \textit{IMaxFP}_{c_s}(n) =_{df} \textsf{inf}^{\,*}_{c_s}(n)$$

***Interprocedural Safety and Coincidence.*** As for the intraprocedural setting, the relevance of the *IMOP*-approach stems from the fact that the annotations it induces can intuitively be understood as the "strongest" data-flow facts possible with respect to the abstract semantics under consideration. The *IMOP*-solution provides thus the surveyor's rod of precision of an interprocedural DFA algorithm. Conversely, the relevance of the *IMaxFP*-approach stems from the fact that in practice the annotation function it induces can effectively (and often even efficiently) be computed. Safety and coincidence are the commonly used terms to relate the *IMaxFP*-solution to the *IMOP*-solution and to characterize its precision with respect to the *IMOP*-solution.

Intuitively, safety means that the *IMaxFP*-solution is a conservative, i.e. a lower approximation of the *IMOP*-solution. Coincidence means equality of the two solutions, and hence precision of the *IMaxFP*-solution with respect to the *IMOP*-solution. Monotonicity and distributivity of the data-flow functions are sufficient to ensure safety and coincidence. This is summarized in the Interprocedural Safety and Coincidence Theorem 1 recalled below (cf. [14]).

**Theorem 1 (Interprocedural Safety and Coincidence).**
*The IMaxFP-solution*

1. *...is a lower approximation of the IMOP-solution, i.e.,* $\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.$ $\textit{IMaxFP}_{c_s}(n) \sqsubseteq \textit{IMOP}_{c_s}(n)$, *if the data-flow functional* $[\![\ ]\!]'$ *is monotonic* (***Safety***).
2. *...coincides with the IMOP-solution, i.e.,* $\forall\, c_s \in \mathcal{C}\ \forall\, n \in N.$ $\textit{IMaxFP}_{c_s}(n) = \textit{IMOP}_{c_s}(n)$, *if the data-flow functional* $[\![\ ]\!]'$ *is distributive.* (***Coincidence***).

Fundamental for proving this theorem is the Main Lemma 1. Intuitively, it states that the semantics of a procedure coincides with the greatest solution of Equation System 1 of the end node of this procedure, if all data-flow functions are distributive. If they are monotonic, then it is still a safe, i.e., conservative approximation.

**Lemma 1 (Main Lemma (2nd Order)).**
*For all* $e \in E_{call}$*, we have:*

1. $[\![\, e \,]\!] \sqsubseteq \bigsqcap\{[\![\, p \,]\!]' \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, *if the data-flow functional* $[\![\ ]\!]'$ *is monotonic,*
2. $[\![\, e \,]\!] = \bigsqcap\{[\![\, p \,]\!]' \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$, *if the data-flow functional* $[\![\ ]\!]'$ *is distributive.*

## 4 Reverse Interprocedural Data-Flow Analysis

The data-flow functional $[\![\ ]\!]'$ of an abstract semantics induces a *reverse* counterpart $[\![\ ]\!]'_R : E \to (\mathcal{C} \to \mathcal{C})$ (cf. [20,21]). This is the key to reverse data-flow analysis.

For every edge $e \in E$ and data-flow fact $c \in \mathcal{C}$, the reverse data-flow function is defined by

$$[\![ e ]\!]_R'(c) =_{df} \bigsqcap \{ c' \mid [\![ e ]\!]'(c') \sqsupseteq c \}$$

Intuitively, a reverse data-flow function $[\![ e ]\!]_R'$ maps a data-flow fact $c$ assumed be to valid at the destination of $e$ to the "weakest" data-flow fact $\hat{c}$, which must be valid at the origin of $e$ in order to ensure the validity of $c$ after executing $e$.

In terms of program verification, classical DFA aims intuitively at computing "strongest post-conditions," while reverse DFA aims at computing "weakest pre-conditions." As we will see later in this paper, this holds formally whenever the underlying data-flow functional $[\![ \ ]\!]$ is distributive.

Lemma 2 and Lemma 3 demonstrate the close relationship between an abstract semantics and its reverse counterpart (cf. [20,21]).

**Lemma 2.** *1.* $[\![ e ]\!]_R'$ *is well-defined and monotonic.*
*2. If* $[\![ e ]\!]'$ *is distributive, then* $[\![ e ]\!]_R'$ *is additive.*

**Lemma 3.** *1.* $[\![ e ]\!]_R' \circ [\![ e ]\!]' \sqsubseteq Id_{\mathcal{C}}$*, if* $[\![ e ]\!]'$ *is monotonic.*
*2.* $[\![ e ]\!]' \circ [\![ e ]\!]_R' \sqsupseteq Id_{\mathcal{C}}$*, if* $[\![ e ]\!]'$ *is distributive.*

In terms of abstract interpretation (cf. [18]), Lemma 3 means that a distributive data-flow function and its reverse counterpart constitute a *Galois connection* (cf. [22]), or equivalently, that a distributive data-flow function and its reverse counterpart are a pair of *adjunct* functions.

It is worth noting that the development so far in this section is exactly the same as for the intraprocedural base case of reverse data-flow analysis (cf. [1]), which is because of the absence of parameters and local variables in the setting of [14].

We are now ready to proceed with highlighting the duality of reverse and classical DFA by presenting the *reverse interprocedural join-over-all-paths* (*R-IJOP*) approach and the *reverse interprocedural minimal-fixed-point* (*R-IMinFP*) approach for the setting of [14]. Conceptually, they are the reverse counterparts of the *IMOP*- and the *IMaxFP*-approach.

***The R-IJOP-Approach.*** Like a data-flow functional $[\![ \ ]\!]'$, its reverse counterpart $[\![ \ ]\!]_R'$ can inductively be extended to (finite) interprocedural paths. To simplify the definition of the fixed-point counterpart of the *R-IJOP*-approach, we assume that the program point of interest $q$ is different from the start node $\mathbf{s}$, and that $\mathbf{q}$ is a fresh node, which has the same predecessors as $q$ but no successors.[2] Note that in the definition of the *R-IJOP*-solution, the "meet" is replaced by the "join," and that paths are considered in the opposite direction of control flow. This reflects that reverse DFA aims at computing "weakest pre-conditions" rather than "strongest post-conditions."

$$\textit{R-IJOP}\textbf{-Solution:} \quad \textit{R-IJOP}_{c_q}(n) =_{df} \bigsqcup \{ [\![ p ]\!]_R'(c_q) \mid p \in \mathbf{IP}[n, \mathbf{q}] \}$$

[2] This does not cause any subtleties. For $\mathbf{s}$ the reverse DFA problem is trivial. For other nodes $q$ and their copies $\mathbf{q}$, the *IMOP*-solutions coincide because of $\mathbf{IP}[\mathbf{s}, q] = \mathbf{IP}[\mathbf{s}, \mathbf{q}]$.

***The*** *R-IMinFP*-***Approach.*** The *reverse interprocedural minimal-fixed-point* (*R-IMinFP*) approach is the fixed-point counterpart of the *R-IJOP*-approach. Like the *IMaxFP*-approach it is a two-step approach separating the second order treatment of procedures from the first order characterization of the *R-IMinFP*-solution. Compared to the *IMaxFP*-approach it is worth noting that the roles of start nodes and end nodes of a flow graph of $S$ are interchanged.

**Equation System 3 (Reverse Semantics of Procedures (2nd Order))**

$$[\![\, n \,]\!]_R = \begin{cases} Id_{\mathcal{C}} & \text{if } n \in \{\, \mathbf{e}_0, \ldots, \mathbf{e}_k \,\} \\ \bigsqcup\{[\![\, (n,m) \,]\!]_R \circ [\![\, m \,]\!]_R \mid m \in succ_{flowGraph(n)}(n)\} & \text{otherwise} \end{cases}$$

*and*

$$[\![\, e \,]\!]_R = \begin{cases} [\![\, e \,]\!]_R' & \text{if } e \in E \backslash E_{call} \\ [\![\, start(caller(e)) \,]\!]_R & \text{otherwise} \end{cases}$$

Denoting the least solution of Equation System 3 by $[\![\ ]\!]_R$ and $[\![\ ]\!]_R$, the least solution of Equation System 4 denoted by $\mathbf{reqInf}^*_{c_q}$ is the solution of the *R-IMinFP*-approach, where **reqInf** reminds to *required data-flow fact*.

**Equation System 4 (The *R-IMinFP*-Equation System)**

$$\mathbf{reqInf}\,(n) = \begin{cases} c_q & \text{if } n = \mathbf{q} \\ \bigsqcup\{\, [\![\, (n,m) \,]\!]_R(\mathbf{reqInf}\,(m)) \mid m \in succ_{flowGraph(n)}(n) \,\} \ \sqcup \\ \bigsqcup\{\, \mathbf{reqInf}\,(start(callee((n,m)))) \mid (n,m) \in E_{call} \,\} & \text{otherwise} \end{cases}$$

$$\textit{R-IMinFP}\textbf{-Solution: } \textit{R-IMinFP}_{c_q}(n) =_{df} \mathbf{reqInf}^*_{c_q}(n)$$

It is worth noting that end nodes $m$, $m \in \{\mathbf{e}_0, \ldots, \mathbf{e}_k\}$, $m \neq q$, do not occur as special case in the equation system above. For these nodes $m$ we have $\textit{R-IMinFP}_{c_q}(m) = \bot$. This coincides in fact with the *R-IJOP*-solution at these nodes. Note that all paths starting at a node in $\{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$ are invalid (and that there is no path starting at $\mathbf{e}_0$ at all). They start with a closing parenthesis. Hence, $\textit{R-IJOP}_{c_q}(\mathbf{e}_i) = \bigsqcup\emptyset = \bot$. The duality of Equation System 4 to the one of the *IMaxFP*-approach becomes apparent when rewriting Equation System 2 equivalently as follows:

**Equation System 5 (The *IMaxFP*-Equation System / Version 2)**

$$\mathbf{inf}\,(n) = \begin{cases} c_{\mathbf{s}} & \text{if } n = \mathbf{s} \\ \bigsqcap\{\, [\![\, (m,n) \,]\!](\mathbf{inf}\,(m)) \mid m \in pred_{flowGraph(n)}(n) \,\} \ \sqcap \\ \bigsqcap\{\, \mathbf{inf}\,(src(e)) \mid n \in \{\mathbf{s}_1, \ldots, \mathbf{s}_k\} \ \wedge e \in caller(flowGraph(n)) \,\} \\ \quad \text{otherwise} \end{cases}$$

***Reverse Interprocedural Safety and Coincidence.*** We are now ready to present the reverse version of the interprocedural safety and coincidence theorem. It opposes the *R-IMinFP*-solution to the *R-IJOP*-solution. The *R-IMinFP*-solution is an upper approximation of the *R-IJOP*-solution, if the reverse data-flow functions are monotonic.[3] Both solutions coincide, if these functions are additive. Together with Lemma 2 and the inductive extension of Lemma 3 to paths, this allows us to prove.

**Theorem 2 (Reverse Interprocedural Safety and Coincidence).**
*The R-IMinFP-solution*

1. *...is an upper approximation of the R-IJOP-solution, i.e.,* $\forall\, c_q \in \mathcal{C}\ \forall\, n \in N.\ \textit{R-IMinFP}_{c_q}(n) \sqsupseteq \textit{R-IJOP}_{c_q}(n)$
2. *...coincides with the R-IJOP-solution, i.e.,* $\forall\, c_q \in \mathcal{C}\ \forall\, n \in N.\ \textit{R-IMinFP}_{c_q}(n) = \textit{R-IJOP}_{c_q}(n)$*, if the data-flow functional* $[\![\ ]\!]'$ *is distributive.*

Fundamental for proving this theorem is the reverse counterpart of the Main Lemma 1 dealing with the reverse global semantics of procedures.

**Lemma 4 (Reverse Main Lemma (2nd Order)).**
*For all* $e \in E_{call}$*, we have:*

1. $[\![\, e \,]\!]_R \sqsupseteq \bigsqcup\{[\![\, p \,]\!]_R^{\prime} \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$,
2. $[\![\, e \,]\!]_R = \bigsqcup\{[\![\, p \,]\!]_R^{\prime} \mid p \in \mathbf{CIP}[src(e), dst(e)]\}$*, if the data-flow functional* $[\![\ ]\!]'$ *is distributive.*

## 5 Classical and Reverse Interprocedural DFA: The Link

The Reverse Interprocedural Safety and Coincidence Theorem 2 focuses on the relationship of the *R-IMinFP*-solution and the *R-IJOP*-solution, and thus on reverse interprocedural DFA itself. In contrast, the Interprocedural Link Theorem 3 addresses the relationship between classical and reverse DFA (cf. [13,1]). In the version below, it focuses on distributive data-flow functionals, which are especially important for mimicing classical DFAs by their reverse counterparts.

**Theorem 3 (Interprocedural Link Theorem).**
*For distributive data-flow functionals* $[\![\ ]\!]'$ *we have:*

$$\forall\, c_{\mathbf{s}}, c_q \in \mathcal{C}\ \forall\, q \in N.\ \textit{R-IJOP}_{c_q}(q) \sqsubseteq c_{\mathbf{s}} \iff \textit{IMOP}_{c_{\mathbf{s}}}(q) \sqsupseteq c_q$$

In fact, the Interprocedural Reverse Safety and Coincidence Theorem 2 together with the Interprocedural Link Theorem 3 provides the foundation for constructing correct and precise demand-driven interprocedural DFAs based on reverse DFA. Particularly important are analyses, whose results for the program points investigated by the demand-driven DFA coincide with those of its

[3] Note that monotonicity of the reverse data-flow functional is always given. Hence, the reverse interprocedural safety theorem does not need an explicit premise.

underlying classical counterpart. In particular, this applies to the large class of bitvector data-flow analyses, which support a variety of very powerful and widely used optimizations such as redundancy elimination, dead-code elimination and array bounds check elimination (cf. [23]).

## 6 Application

We conclude our presentation with presenting the abstract semantics and its reverse counterpart for the *availability* analysis which is required to perform the analysis and optimization displayed in Figures 1 and 2. In Table 1, $\mathcal{B}_X$ denotes the set of Boolean truth values *true* and *false*, which is enriched by a third element *failure*, which serves as the (artificial) top element of the data-flow lattice. It can easily be checked that the data-flow functions of the availability problem are distributive and their reverse counterparts are additive. Hence, they satisfy the preconditions of the Reverse Interprocedural Coincidence Theorem 2 and the Interprocedural Link Theorem 3. Note that the specification of the availability analysis is the same as for the intraprocedural case except that here the specification is given for an interprocedural flow graph instead of an intraprocedural one (cf. [1]). As mentioned earlier, this is because of the absence of parameters and local variables in the setting of [14].

---

**Abstract semantics for *availability***:

1. *Data-flow lattice*: $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{B}_X, \wedge, \vee, \leq, \mathit{false}, \mathit{failure})$
2. *Data-flow functional*: $[\![\ ]\!]'_{av} : E^* \to (\mathcal{B}_X \to \mathcal{B}_X)$ defined by

$$\forall\, e \in E^*.\ [\![\, e \,]\!]'_{av} =_{df} \begin{cases} Cst^X_{true} & \text{if } Comp_{\,e} \wedge Transp_{\,e} \\ Id_{\mathcal{B}_X} & \text{if } \neg Comp_{\,e} \wedge Transp_{\,e} \\ Cst^X_{false} & \text{otherwise} \end{cases}$$

**Reverse abstract semantics for *availability***:

1. *Data-flow lattice*: $(\mathcal{C}, \sqcap, \sqcup, \sqsubseteq, \bot, \top) =_{df} (\mathcal{B}_X, \wedge, \vee, \leq, \mathit{false}, \mathit{failure})$
2. *Reverse data-flow functional*: $[\![\ ]\!]'_{av_R} : E^* \to (\mathcal{B}_X \to \mathcal{B}_X)$ defined by

$$\forall\, e \in E^*.\ [\![\, e \,]\!]'_{av_R} =_{df} \begin{cases} R\text{-}Cst^X_{true} & \text{if } [\![\, e \,]\!]'_{av} = Cst^X_{true} \\ R\text{-}Id_{\mathcal{B}_X} & \text{if } [\![\, e \,]\!]'_{av} = Id_{\mathcal{B}_X} \\ R\text{-}Cst^X_{false} & \text{if } [\![\, e \,]\!]'_{av} = Cst^X_{false} \end{cases}$$

*The functions $Id_{\mathcal{B}_X}$, $Cst^X_{true}$ and $Cst^X_{false}$ denote the identity on $\mathcal{B}_X$, and the extensions of the constant functions $Cst_{true}$ and $Cst_{false}$ from $\mathcal{B}$ to $\mathcal{B}_X$ leaving the argument failure invariant. The predicates $Comp_{\,e}$ and $Transp_{\,e}$ hold for edge $e$, if the computation under consideration is computed and not modified along edge $e$, respectively.*

**Table 1.** Abstract and Reverse Abstract Semantics for Availability.

---

It is worth noting that data-flow queries can naturally be processed in parallel. In the extreme case for each node of a program a data-flow query can be started. The combined results of these analyses provide then the same information as the underlying conventional data-flow analysis they are derived from but, dependently on the number of computing units available, more efficiently. Note that a smart implementation can also share and reuse results of already completed data-flow queries leading to a further speed-up in practice. For illustration consider again the example of Figure 1, which indeed suggests that the data-flow queries regarding the availability of the computations of $a+b$ and $c+b$ at the entrances of the edges $e$ and $f$ can be processed in parallel.

## 7 Conclusions

Exploiting the computing power of multi-core computing systems effectively to improve the scalability of program analyses requires program analyses which can easily be parallelized and mapped to such computing systems. In this paper we argue that reverse data-flow analyses are particularly suitable for this. This is important because a variety of practically relevant analysis problems and optimizations based thereon including the large class of bitvector problems can equivalently be solved by the reverse counterparts of a conventional data-flow analysis. Fundamental for this are two theorems on the correctness and precision of a reverse data-flow analysis and the relationship of its results to those of its underlying conventional counterpart. In this paper we presented these two theorems, the *Reverse Interprocedural Safety and Coincidence Theorem* and the *Interprocedural Link Theorem* for the interprocedural setting of [14], which complements the contributions of [1] and [13] for the intraprocedural sequential and an intraprocedural parallel setting, and closes a gap in previous related work.

## References


1. Knoop, J.: Data-flow analysis for hot-spot program optimization. In: Proceedings of the 14th Workshop "Programmiersprachen und Grundlagen der Programmierung" (Timmendorfer Strand, Germany). Bericht Nr. A-07-07, Institut für Softwaretechnik und Programmiersprachen, University of Lübeck, Germany (2007) 124 – 131
2. Agrawal, G.: Demand-driven construction of call graphs. In: Proc. 9th Int. Conf. on Compiler Construction (CC 2000). LNCS 1781, Springer-V., Germany (2000) 125 – 140
3. Babich, W.A., Jazayeri, M.: The method of attributes for data flow analysis: Part II. demand analysis. Acta Informatica **10 (3)** (1978) 265 – 272
4. Bodík, R., Gupta, R., Sarkar, V.: ABCD: Eliminating array bounds checks on demand. In: Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'00). Volume *35*,5 of ACM SIGPLAN Not. (2000) 321 – 333
5. Duesterwald, E.: A Demand-driven Approach for Efficient Interprocedural Data Flow Analysis. PhD thesis, Univ. Pittsburgh, Dept. of Comp. Science, Pittsburgh, PA (1996)

6. Duesterwald, E., Gupta, R., Soffa, M.L.: A demand-driven analyzer for data flow testing at the integration level. In: Proc. IEEE Int. Conf. on Software Engineering (CoSE'96). (1996) 575 – 586
7. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: Conf. Rec. 22nd Symp. Principles of Prog. Lang. (POPL'95), ACM, NY (1995) 37 – 48
8. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM Trans. Prog. Lang. Syst. **19** (1997) 992 – 1030
9. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. 3rd ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE'95). (1995) 104 – 115
10. Reps, T.: Demand interprocedural program analysis using logic databases. In Ramakrishnan, R., ed.: Applications of Logic Databases. Kluwer Academic Publishers (1994)
11. Reps, T.: Solving demand versions of interprocedural analysis problems. In: Proc. 5th Int. Conf. on Compiler Construction (CC'94). LNCS 786, Springer-V. (1994) 389 – 403
12. Yuan, X., Gupta, R., Melham, R.: Demand-driven data flow analysis for communication optimization. Parallel Processing Letters **7** (1997) 359 – 370
13. Knoop, J.: Parallel data-flow analysis of explicitly parallel programs. In: Proc. 5th Europ. Conf. on Parallel Processing (Euro-Par'99). LNCS 1685, Springer-V. (1999) 391 – 400
14. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In Muchnick, S.S., Jones, N.D., eds.: Program Flow Analysis: Theory and Applications. Prentice Hall, Englewood Cliffs, NJ (1981) 189 – 233
15. Soffa, M.L.: Tutorial: Techniques to improve the scalability and precision of data flow analysis. In: Proc. 6th Static Analysis Symposium (SAS'99). LNCS 1694, Springer-V., Germany (1999) 355 – 356 (Invited contribution).
16. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Conf. Rec. 22nd Annual Symp. on Principles of Programming Languages (POPL'95), ACM, NY (1995) 49 – 61
17. Knoop, J.: Optimal Interprocedural Program Optimization: A new Framework and its Application. PhD thesis, Univ. of Kiel, Germany (1993) LNCS Tutorial 1428, Springer-V., 1998.
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Rec. 4th Symp. Principles of Prog. Lang. (POPL'77), ACM, NY (1977) 238 – 252
19. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7** (1977) 305 – 317
20. Hughes, J., Launchbury, J.: Reversing abstract interpretations. In: Proc. 4th European Symp. on Programming (ESOP'92). LNCS 582, Springer-V., Germany (1992) 269 – 286
21. Hughes, J., Launchbury, J.: Reversing abstract interpretations. Science of Computer Programming **22** (1994) 307 – 326
22. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-V., Germany (1999)
23. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, CA (1997)