UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs

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

Jürgen Vollmer
Institut für Programm- und Datenstrukturen
Universität Karlsruhe
Vincenz-Prießnitz-Straße 1
D–76128 Karlsruhe

MIP-9409
August 1994

Abstract

In this paper we show how to construct optimal bitvector analysis algorithms for parallel programs with shared memory that are as efficient as their purely sequential counterparts, and which can easily be implemented. Whereas the complexity result is rather obvious, our optimality result is a consequence of a new Kam/Ullman-style Coincidence Theorem. Thus, the important merits of sequential bitvector analyses survive the introduction of parallel statements.

Keywords

Parallelism, interleaving semantics, synchronization, program optimization, data flow analysis, bitvector problems, definition-use chains, partial redundancy elimination, partial dead code elimination.

Contents

1 Motivation

Parallel implementations are of growing interest, as they are more and more supported by modern hardware environments. However, despite its importance [SHW, SW, WS], there is currently very little work on classical data flow analysis for parallel languages. Probably, the reason for this deficiency is that a naive adaptation fails [MP] and the straightforward correct adaptation needs an unacceptable effort, which is caused by considering all interleavings that manifest the possible executions of a parallel program.

Thus, either heuristics are proposed to avoid the consideration of all the interleavings [McD], or restricted situations are considered, which do not require to consider the interleavings at all. To our knowledge, the most relevant publications in this direction are [CH, GS]. In [CH] a situation without synchronization is considered, and in [GS] data independence of parallel components is required. Thus the result of a parallel execution does not depend on the particular choice of the interleaving. In [GS], this is exploited for the construction of an optimal and efficient algorithm determining the reaching-definition information.

In this paper we show how to construct arbitrary bitvector analysis algorithms for parallel programs with shared memory that

1. optimally cover the phenomenon of *interference*
2. are as *efficient* as their sequential counterparts and
3. easy to implement.

The first property is a consequence of a Kam/Ullman-style ([KU]) Coincidence Theorem for bitvector analyses stating that the *parallel meet over all paths (PMOP)* solution, which specifies the desired properties, coincides with our *parallel bitvector maximal fixed point ($PMFP_{BV}$)* solution, which is the basis of our algorithm. This result is rather surprising, as it states that although the various interleavings of the executions of parallel components are semantically different, they need not be considered during bitvector analysis, which is the key observation of this paper.

The second property is a simple consequence of the fact, that our algorithm behaves like standard bitvector algorithms, and the third property is due to the fact, that only a minor modification of the sequential bitvector algorithm needs to be applied after a preprocess consisting of a single fixed point routine (cf. Section 3.4).

Thus all the well-known algorithms for liveness, availability, very business, reaching definitions, definition-use chains (cf. [He]), partial redundancy elimination (cf. [DS, DRZ, KRS1, KRS2, MR]), partial dead code elimination (cf. [KRS2]) or strength reduction (cf. [Dh, JD1, JD2, KRS3]) can be adapted for parallel programs at almost no cost on the runtime and the implementation side.

The next section will recall the sequential situation, while Section 3 develops the corresponding notions for parallel programs. Subsequently, Section 4 sketches some applications of our algorithm and Section 5 contains the conclusions. The Appendix contains the detailed algorithm and a number of figures illustrating the formal development of the paper.

2 Sequential Programs

In this section we summarize the sequential setting of data flow analysis.

2.1 Representation

In the sequential setting 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.[1] Nodes $n \in N$ represent the statements, edges $(n, m) \in E$ the nondeterministic branching structure of the procedure under consideration, and $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G, which are assumed to possess no predecessors and successors, respectively, and to represent the empty statement `skip`. $pred_G(n) =_{df} \{\, m \mid (m, n) \in E \,\}$ 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. 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, $\lambda(p)$ denotes the number of node occurrences in p, and ε the unique path of length 0. Finally, every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

2.2 Data Flow Analysis

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 (cf. [He, MJ]). For imperative languages, DFA provides information about the program states that may occur at some given program points during execution. Theoretically well-founded are DFAs that are based on *abstract interpretation* (cf. [CC, Ma]). 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*

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

which gives abstract meaning to every program statement in terms of a transformation function from a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$ into itself, where the elements of $\mathcal{C}$ express the DFA-information of interest.[2]

Since $\mathbf{s}$ and $\mathbf{e}$ are assumed to represent the empty statement `skip` they are associated with the identity $Id_{\mathcal{C}}$ on $\mathcal{C}$. A local semantic functional $[\![\ ]\!]$ can easily be extended to cover finite paths as well. For every path $p = (n_1, \ldots, n_q) \in \mathbf{P}_G[m, n]$, define:

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

2.2.1 The *MOP*-Solution of a DFA

The *MOP*-solution — the solution of the *meet over all paths (MOP)* strategy in the sense of Kam and Ullman [KU] — defines the intuitively desired solution of a DFA. This

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

[2] In the following $\mathcal{C}$ will always denote a complete lattice.

strategy directly mimics possible program executions in that it "meets" (intersects) all informations belonging to a program path reaching the program point under consideration.

The *MOP*-Solution:

$$\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MOP_{(G,[\![\ ]\!])}(n)(c_0) = \bigsqcap \{\, [\![\, p\,]\!](c_0) \,|\, p \in \mathbf{P}_G[\mathbf{s}, n)\,\}$$

In fact, this directly reflects our desires, but is in general not effective.

2.2.2 The *MFP*-Solution of a DFA

The point of the *maximal fixed point (MFP)* strategy in the sense of Kam and Ullman [KU] is to iteratively approximate the greatest solution of a system of equations which specifies the consistency between pre-conditions expressed in terms of $\mathcal{C}$:

Equation System 2.1

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

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

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

For monotonic functionals,[3] this leads to a suboptimal but algorithmic description (see Algorithm A.1 in Appendix A), and the question of optimality of the *MFP*-solution was elegantly answered by Kildall [Ki1, Ki2], and Kam and Ullman [KU]:

Theorem 2.2 (The (Sequential) Coincidence Theorem)
Given a flow graph $G = (N, E, \mathbf{s}, \mathbf{e})$*, the MFP-solution and the MOP-solution coincide, i.e.* $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MOP_{(G,[\![\ ]\!])}(n)(c_0) = MFP_{(G,[\![\ ]\!])}(n)(c_0)$*, whenever all the semantic functions* $[\![\, n\,]\!]$*,* $n \in N$*, are distributive.*[4]

2.2.3 The Functional Characterization of the *MFP*-Solution

From interprocedural DFA, it is well-known that the *MFP*-solution can alternatively be defined by means of a functional approach [SP]. Here, one iteratively approximates the greatest solution of a system of equations specifying consistency between functions $[\![\, n\,]\!]$, $n \in N$. Intuitively, a function $[\![\, n\,]\!]$ transforms data flow information that is assumed to to be valid at the start node of the program into the data flow information being valid before the execution of n.

[3] A function $f : \mathcal{C} \to \mathcal{C}$ is called *monotonic* iff $\forall\, c, c' \in \mathcal{C}.\ c \sqsubseteq c'$ implies $f(c) \sqsubseteq f(c')$.

[4] A function $f : \mathcal{C} \to \mathcal{C}$ is called *distributive* iff $\forall\, C' \subseteq \mathcal{C}.\ \ f(\bigsqcap C') = \bigsqcap \{f(c) \,|\, c \in C'\}$. It is well-known that distributivity is a stronger requirement than monotonicity in the following sense: A function $f : \mathcal{C} \to \mathcal{C}$ is monotonic iff $\forall\, C' \subseteq \mathcal{C}.\ f(\bigsqcap C') \sqsubseteq \bigsqcap \{f(c) \,|\, c \in C'\}$.

Definition 2.3 (The Functional Approach)
The functional $[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$ *is defined as the greatest solution of the equation system given by:*

$$[\![n]\!] = \begin{cases} Id_{\mathcal{C}} & if\ n = \mathbf{s} \\ \sqcap\{[\![m]\!] \circ [\![m]\!] \mid m \in pred_G(n)\} & otherwise \end{cases}$$

The following equivalence result is important [KS]:

Theorem 2.4 $\forall\, n \in N\ \forall\, c_0 \in \mathcal{C}.\ MFP_{(G,[\![\ ]\!])}(n)(c_0) = [\![n]\!](c_0)$

The functional characterization of the *MFP*-solution will be the (intuitive) key for computing the parallel version of the maximal fixed point solution. As we are only dealing with Boolean values later on, this characterization can easily be coded back into the standard form.

3 Parallel Programs

As usual, we consider an imperative parallel programming language with an interleaving semantics. Formally, this means that we view parallel programs semantically as 'abbreviations' of usually much larger nondeterministic programs, which result from a product construction between parallel components. In fact, in the worst case, the size of the nondeterministic 'product' program grows exponentially in the number of parallel components of the corresponding parallel program. This immediately clarifies the dilemma of data flow analysis for parallel programs: even though it can be reduced to standard data flow analysis on the corresponding nondeterministic program, this approach is unacceptable in practice for complexity reasons. Fortunately, as we will see in Section 3.3, bitvector analyses, which are most relevant in practice, can be performed as efficiently on parallel programs as on sequential programs.

The following section establishes the notational background for the formal development and the proofs. One could therefore try to immediately continue with Section 3.3 and to 'backtrack' to Section 3.1 at need.

3.1 Representation

Syntactically, parallelism is expressed by means of a `par` statement whose components are assumed to be executed independently and in parallel on a shared memory.[5] As usual, we assume that there are neither jumps leading into a component of a `par` statement from outside nor vice versa.

Similarly to [GS], we represent a parallel program by a nondeterministic *parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ with node set N^* and edge set E^*. Except for subgraphs representing `par` statements a parallel flow graph is a nondeterministic flow graph in the sense of Section 2, i.e., nodes $n \in N^*$ represent the statements, edges $(m, n) \in E^*$ the nondeterministic branching structure of the procedure under consideration, and $\mathbf{s}^*$ and $\mathbf{e}^*$ denote the distinct *start node* and *end node*, which are assumed to possess no predecessors and successors, respectively. Like in Section 2, we assume that every node

[5] Integrating a replicator statement in order to allow a dynamical process creation is straightforward.

$n \in N^*$ lies on a path from $\mathbf{s}^*$ to $\mathbf{e}^*$, and that the start and end nodes of parallel flow graphs represent the empty statement `skip`. Additionally, $pred_{G^*}(n)=_{df} \{ m \mid (m,n) \in E^* \}$ and $succ_{G^*}(n)=_{df} \{ m \mid (n,m) \in E^* \}$ denote the set of all immediate predecessors and successors of a node $n \in N^*$, respectively.

A `par` statement as well as each of its components constitute themselves a parallel flow graph (cf. Figure 1 and Appendix C for illustration). The start node and end node of a graph representing a `par` statement have the start nodes and end nodes of the component flow graphs as their only successors and predecessors, respectively. $\mathcal{G}_{\mathcal{P}}(G^*)$ denotes the set of all subgraphs of G^* representing `par` statements. Moreover, if $G \in \mathcal{G}_{\mathcal{P}}(G^*)$, then $\mathcal{G}_{\mathcal{C}}(G)$ denotes the set of flow graphs of the parallel components of G, and $Nodes(G)=_{df} \bigcup \{ N' \mid G' \in \mathcal{G}_{\mathcal{C}}(G) \}$ the set of nodes occurring in one of them.[6] It is worth noting that for $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ every graph $G' \in \mathcal{G}_{\mathcal{C}}(G)$ as well as G itself is a single-entry/single-exit region of G^*.

Additionally, we introduce the abbreviations[7]

$$\mathcal{G}_{\mathcal{P}}{}^{max}(G^*)=_{df} \{ G \in \mathcal{G}_{\mathcal{P}}(G^*) \mid \forall\, G' \in \mathcal{G}_{\mathcal{P}}(G^*).\ G \subseteq G' \Rightarrow G = G' \}$$

for the set of *maximal* graphs of $\mathcal{G}_{\mathcal{P}}(G^*)$, and

$$N^*_N =_{df} \{ \mathbf{s} \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \} \quad \text{and} \quad N^*_X =_{df} \{ \mathbf{e} \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \}$$

for the set of start and end nodes of graphs of $\mathcal{G}_{\mathcal{P}}(G^*)$.

Moreover, the functions *start* and *end* map a parallel flow graph to the start node and end node of their argument, respectively, and the function *ParGraph* maps a node $n \in N^*_N$ to the uniquely determined flow graph G with $start(G) = n$.

Additionally, the function *pfg* yields for a node n of N^* that occurs in a component G of some flow graph $G' \in \mathcal{G}_{\mathcal{P}}(G^*)$, the smallest flow graph of $\mathcal{G}_{\mathcal{P}}(G^*)$ containing G; and it yields G^* otherwise, i.e.,

$$pfg(n)=_{df} \begin{cases} \bigcap \{ G' \in \mathcal{G}_{\mathcal{P}}(G^*) \mid n \in Nodes(G') \} & \text{if } n \in \bigcup \{ Nodes(G') \mid G' \in \mathcal{G}_{\mathcal{P}}(G^*) \} \\ G^* & \text{otherwise} \end{cases}$$

Similarly, the function *cfg* maps a node n occurring in a component flow graph of some graph $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ to the smallest component flow graph containing n, i.e.,

$$cfg(n)=_{df} \bigcap \{ G' \in \mathcal{G}_{\mathcal{C}}(G) \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge n \in N' \}$$

Both *pfg* and *cfg* are well-defined, since `par` statements in a program are either unrelated or properly nested.

Finally, given a parallel flow graph G we define a sequential flow graph G_{seq}, which results from G by replacing all nodes belonging to a component flow graph of some graph $G' \in \mathcal{G}_{\mathcal{P}}{}^{max}(G)$ together with all edges starting or ending in such a node by an edge leading from $start(G')$ to $end(G')$. Note that G_{seq} is a nondeterministic sequential flow graph in the sense of Section 2. This is illustrated in Figure 2, which shows the sequentialized version of the parallel flow graph of Figure 1 (see also Appendix C).

[6]We use the convention that node and edge set, and start and end node of a flow graph carry the same index as the flow graph itself. Hence, G' stands for the expanded version $G' = (N', E', \mathbf{s}', \mathbf{e}')$.

[7]For parallel flow graphs G and G' we define: $G \subseteq G'$ if and only if $N \subseteq N'$ and $E \subseteq E'$.

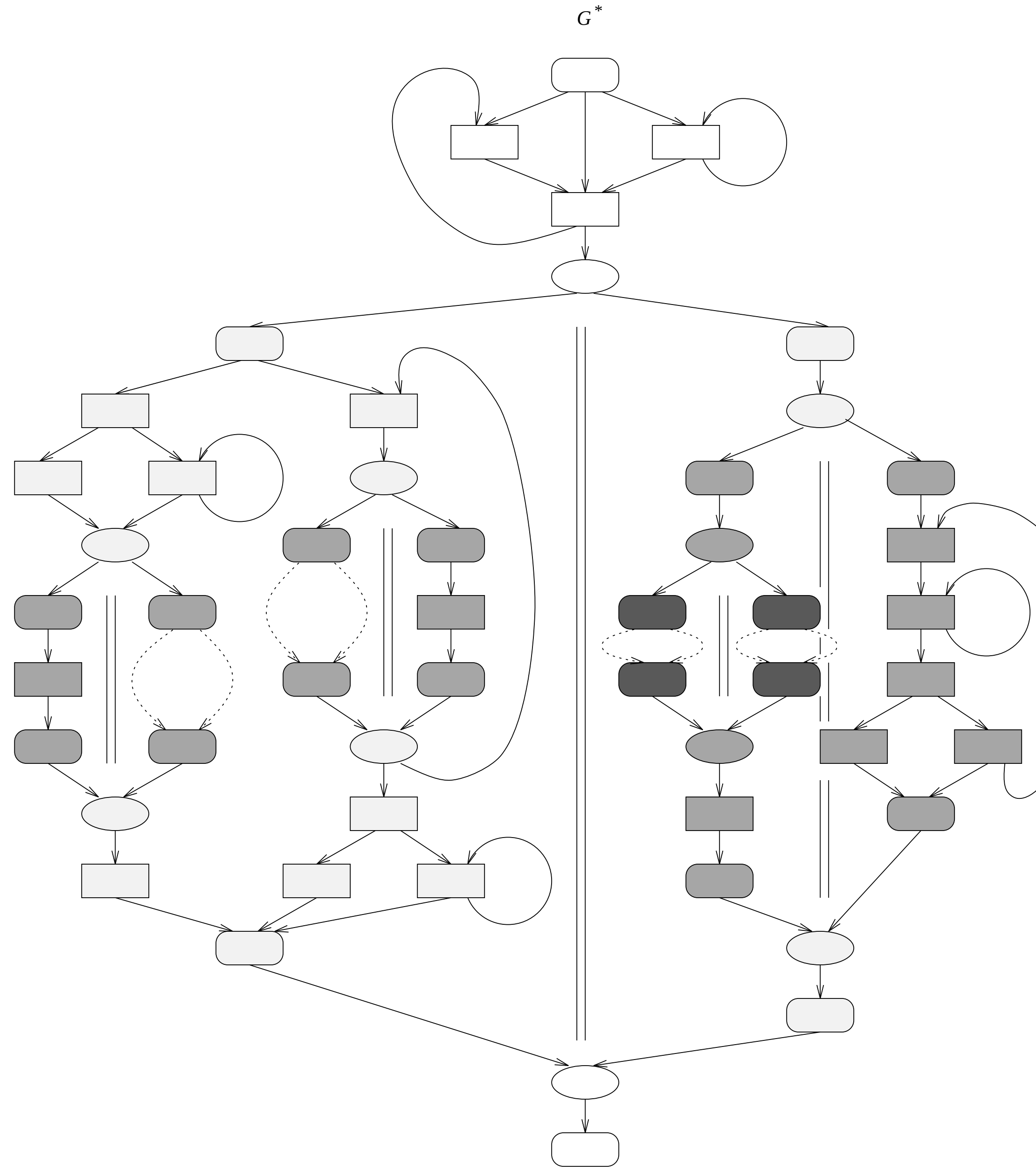

Figure 1: The Parallel Flow Graph G^*

Program Paths of Parallel Programs

As mentioned already, the interleaving semantics of an imperative parallel programming language can be defined via a translation that reduces parallel programs to (much larger) nondeterministic programs. However, there is also an alternative way to characterize the node sequences constituting a parallel (program) path, following in spirit the definition of an interprocedural program path as proposed by Sharir and Pnueli [SP]. They start by interpreting every branch statement purely nondeterministically, which allows to simply use the definition of *finite path* as introduced in Section 2. This results in a superset of the set of all interprocedurally valid paths, which they now define by means of an additional consistency condition. In our case, we are forced to define our consistency condition on arbitrary node sequences, as the consideration of interleavings invalidates the first step.

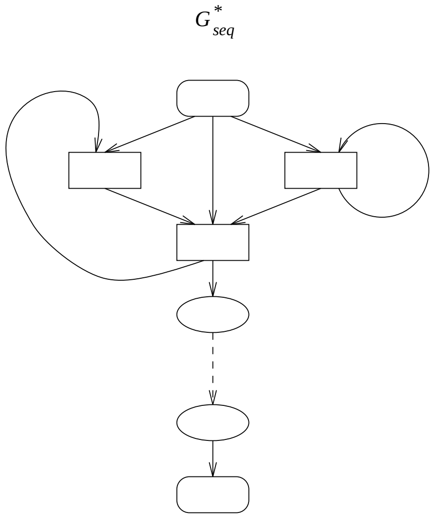

Figure 2: G^*_{seq}

Here, the following notion of well-formedness is important.

Definition 3.1 (*G*-Well-Formedness)
Let G be a (parallel) flow graph, and $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes. Then p is G-well-formed *if and only if*

1. $p{\downarrow}_{G_{seq}} \in \mathbf{P}_{G_{seq}}[start(G_{seq}), end(G_{seq})]$
2. $(\forall\, i \in \{1, \ldots, q\}.\ n_i \in N^*_N)\ (\exists\, j \in \{i+1, \ldots, q\}.\ n_j \in N^*_X)$
   $(\forall\, k \in \{i+1, \ldots, j-1\}.\ n_k \notin N^*_X).\ \forall\, G' \in \mathcal{G}_{\mathcal{C}}(ParGraph(n_i)).\ (n_{i+1}, \ldots, n_{j-1})$

   is G'-well-formed

where $p{\downarrow}_{G_{seq}}$ results from p by removing all nodes not in G_{seq}.

Now the set of parallel paths is defined as follows.

Definition 3.2 (Parallel Path)
Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel flow graph, and $p =_{df} (n_1, \ldots, n_q)$ be a sequence of nodes of N^*. Then:

1. *p is a* parallel path from $\mathbf{s}^*$ to $\mathbf{e}^*$ *if and only if p is G^*-well-formed.*
2. *p is a* parallel path from n_1 to n_q *if it is a subpath of some parallel path from $\mathbf{s}^*$ to $\mathbf{e}^*$.*

$\mathbf{PP}_{G^*}[m, n]$ denotes the set of all parallel paths from m to n, and $\mathbf{PP}_{G^*}[m, n)$ the set of all parallel paths from m to a predecessor of n, defined by

$$\mathbf{PP}_{G^*}[m, n) =_{df} \{(n_1, \ldots, n_q) \mid (n_1, \ldots, n_q, n_{q+1}) \in \mathbf{PP}_{G^*}[m, n]\}$$

Interleaving Predecessors

Given a sequential flow graph G, the set of nodes that might dynamically precede a node n is precisely given by the set of its static predecessors $pred_G(n)$. Given a parallel flow graph, however, the interleaving of statements of parallel components must be taken care of. In fact, nodes n occurring in a component of some `par` statement additionally have all nodes as dynamic predecessors, whose execution may be interleaved with that of n. We will denote these 'potentially parallel' nodes as *interleaving predecessors*. The set of all interleaving predecessors of a node $n \in N^*$ is recursively defined by means of the function $Pred_{G^*}^{Itlvg} : N^* \to \mathcal{P}(N^*)$, where $\mathcal{P}$ denotes the power set operator:

$$Pred_{G^*}^{Itlvg}(n) =_{df} \begin{cases} \emptyset & \text{if } pfg(n) = G^* \\ \\ Nodes(ParGraph(n)) \setminus Nodes(cfg(n)) \,\cup & \\ Pred_{G^*}^{Itlvg}(start(cfg(start(ParGraph(n))))) & \text{otherwise} \end{cases}$$

3.2 Data Flow Analysis of Parallel Programs

Like for a sequential program, a DFA for a parallel program is completely specified by means of a local semantic functional

$$[\![\ ]\!] : N^* \to (\mathcal{C} \to \mathcal{C})$$

that gives abstract meaning to every node of a parallel flow graph G^* in terms of a function from $\mathcal{C}$ to $\mathcal{C}$.

Like in the sequential case it is straightforward to extend a local semantic functional to cover also finite parallel paths. Thus, given a node n of a parallel program G^*, the parallel version of the *MOP*-solution is clear, and as in the sequential case, it marks the desired solution to the considered data flow problem:

The *PMOP*-Solution:

$$\forall\, n \in N^* \ \forall\, c_0 \in \mathcal{C}.\ PMOP_{(G^*, [\![\ ]\!])}(n)(c_0) = \bigsqcap \{\, [\![\, p\,]\!](c) \mid p \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n) \,\}$$

Refering to the nondeterministic 'product program', which explicitly represents all the possible interleavings, would allow us to straightforward adapt the sequential situation and to state a Coincidence Theorem. However, this would not be of much practical use, as this approach would require to define the *MFP*-solution relative to the potentially exponential product program. Fortunately, as we will see in the next section, for bitvector algorithms there exists an elegant and efficient way out.

3.3 Bitvector Analyses

Bitvector problems can be characterized by their simplicity of the local semantic functional

$$[\![\ ]\!] : N \to (\mathcal{B} \to \mathcal{B})$$

which specifies the effect of a node on a particular component of the bitvector (see Section 4 for illustration). Here $\mathcal{B}$ is the lattice $(\{\mathit{ff}, \mathit{tt}\}, \sqcap, \sqsubseteq)$ of Boolean truth values with

$f\!f \sqsubseteq tt$ and the logical 'and' as meet operation $\sqcap$, or its dual counterpart with $tt \sqsubseteq f\!f$ and the logical 'or' as meet operation $\sqcap$.

Despite their simplicity, bitvector problems are highly relevant in practice, as they include problems like liveness, availability, very business, reaching definitions, definition-use chains, partial redundancy elimination, partial dead code elimination or strength reduction.

We are now going to show, how to optimize the effort for computing the *PMOP*-solution. This requires the consideration of the semantic domain $\mathcal{F}_{\mathcal{B}}$ consisting of the monotonic Boolean functions $\mathcal{B} \to \mathcal{B}$. Obviously we have:

Proposition 3.3 *1. $\mathcal{F}_{\mathcal{B}}$ simply consists of the constant functions $Const_{tt}$ and $Const_{f\!f}$, together with the identity $Id_{\mathcal{B}}$ on $\mathcal{B}$.*

2. $\mathcal{F}_{\mathcal{B}}$, together with the pointwise ordering between functions, forms a complete lattice with least element $Const_{f\!f}$ and greatest element $Const_{tt}$, which is closed under function composition.

3. All functions of $\mathcal{F}_{\mathcal{B}}$ are distributive.

The key to the efficient computation of the 'interleaving effect' is based on the following simple observation, which pinpoints the specific nature of a domain of functions $M \to M$, M any set, that only consists of constant functions and the identity.

Lemma 3.4 (Main-Lemma)
Let $f_i : \mathcal{F}_{\mathcal{B}} \to \mathcal{F}_{\mathcal{B}}$, $1 \le i \le q$, $q \in I\!N$, be functions from $\mathcal{F}_{\mathcal{B}}$ to $\mathcal{F}_{\mathcal{B}}$. Then we have:

$$\exists\, k \in \{1, \ldots, q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k \ \wedge\ \forall\, j \in \{k+1, \ldots, q\}.\ f_j = Id_{\mathcal{B}}$$

The essence of this lemma for our application is that it restricts the way of possible interference within a parallel program: if there is any interference than this interference is subject to a single statement within a parallel component. Combining this observation with the fact, that for $m \in Pred_{G^*}^{Itlvg}(n)$, there exists a parallel path leading to n whose last step requires the execution of m, we obtain that the potential of interference, which in general would be given in terms of paths, is fully characterized by the set $Pred_{G^*}^{Itlvg}(n)$. In fact, considering the computation of universal properties that are described by maximal fixed points (the computation of minimal fixed points requires the dual argument), the obvious existence of a path to n that does not require the execution of any statement of $Pred_{G^*}^{Itlvg}(n)$ implies that the only effect of interference is 'destruction'. This motivates the introduction of the following predicate:

$NonDestructed : N^* \to \mathcal{B}$ defined by

$$\forall\, n \in N^*.\ NonDestructed(n) =_{df} \bigwedge \{\, [\![\, m \,]\!](tt) \mid m \in Pred_{G^*}^{Itlvg}(n) \,\}$$

which indicates that no node of a parallel component destroys the property under consideration, i.e. $[\![\, m \,]\!] \neq Const_{f\!f}$ for all $m \in Pred_{G^*}^{Itlvg}(n)$. Note that only the constant function induced by this predicate is used in Definition 3.7 to model interference, and in fact, Theorem 3.8 guarantees that this modelling is sufficient. Obviously, this predicate is easily and efficiently computable. Algorithm B.1 computes it as a side result.

Besides taking care of possible interference, we also need to take care of the synchronization required by nodes in N_X^*: in order to leave a parallel statement, all parallel components are required to terminate. The information that is necessary to model this effect can be computed by a hierarchical algorithm that only considers purely sequential programs. The central idea coincides with that of interprocedural analysis [KS]: we need to compute the effect of complete subgraphs, or in this case of complete parallel components. This information is computed in an 'innermost' fashion and then propagated to the next surrounding parallel statement. The following definition, which is also illustrated in Appendix C, describes the complete three-step procedure:

1. Terminate, if G does not contain any parallel components. Otherwise, select successively all maximal flow graphs $G' \in \mathcal{G}_{\mathcal{P}}(G)$ that do not contain a parallel statement, and determine the effect $[\![G']\!]$ of this (purely sequential) graph according to the equational system of Definition 2.3 with respect to the local semantic functional $[\![\ ]\!]'_{seq} : N'_{seq} \to \mathcal{F}_{\mathcal{B}}$ given by

$$[\![n]\!]'_{seq} =_{df} \begin{cases} Id_{\mathcal{B}} \sqcap Const_{NonDestructed(n)} & \text{if } n \in N_N^* \\ [\![pfg(n)]\!]^* & \text{if } n \in N_X^* \\ [\![n]\!] & \text{otherwise} \end{cases}$$

2. Compute the effect $[\![G'']\!]^*$ of the innermost parallel statements G'' of G by

$$[\![G'']\!]^* = \bigsqcap \{ [\![end(G'_{seq})]\!] \mid G' \in \mathcal{G}_{\mathcal{C}}(G'') \}$$

3. Transform G by replacing all innermost parallel statements $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'')$ by $(\{\mathbf{s}'', \mathbf{e}''\}, \{(\mathbf{s}'', \mathbf{e}'')\}, \mathbf{s}'', \mathbf{e}'')$, and replace the local semantics of $\mathbf{s}''$ and $\mathbf{e}''$ by $Id_{\mathcal{B}} \sqcap \bigsqcap\{ [\![n]\!] \mid n \in N''\}$ and $[\![G'']\!]^*$, respectively. Continue with step 1.

This three step algorithm is a straightforward hierarchical adaptation of the algorithm for computing the functional version of the *MFP*-solution for the sequential case. Only the third step realizing the synchronization at nodes in N_X^* needs some explanation, which is summarized in the following lemma.

Lemma 3.5 *The PMOP-solution of a parallel flow graph G that only consists of purely sequential parallel components $G_1, \ldots, G_k$ is given by:*

$$PMOP_{(G, [\![\ ]\!])}(end(G)) = \bigsqcap \{ [\![end(G_i)]\!] \mid 1 \leq i \leq k \}$$

Also the proof of this lemma is a consequence of the Main Lemma 3.4. As a single statement is responsible for the entire effect of a path, the effect of each complete path through a parallel statement is already given by some path through one of the parallel components (the one containing the vital statement). Thus in order to model the effect (or *PMOP*-solution) of a parallel statement, it is sufficient to meet the effects of all paths that are local to one of the components, and it is exactly this fact, which is formalized in Lemma 3.5.

Now the following theorem can be proved by means of a straightforward inductive extension of the functional version of the sequential Coincidence Theorem 2.2, which is tailored to cover complete paths, i.e. paths going from the start to the end of a parallel statement:

Theorem 3.6 (The Hierarchical Coincidence Theorem)
Let $G \in \mathcal{G}_{\mathcal{P}}(G^*)$ be a parallel flow graph, and $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have:

$$PMOP_{(G,[\![\ ]\!])}(end(G)) = [\![\, G \,]\!]^*$$

After this hierarchical preprocess the following modification of the equation system for sequential bitvector analyses is optimal:

Definition 3.7 *The functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ is defined as the greatest solution of the equation system given by:*[8]

$$[\![\, n \,]\!] = \begin{cases} Id_{\mathcal{B}} & \text{if } n = \mathbf{s}^* \\ [\![\, ParGraph(n) \,]\!]^* \circ [\![\, start(ParGraph(n)) \,]\!] \sqcap Const_{NonDestructed(n)} & \text{if } n \in N^*_X \\ \bigsqcap\{\, [\![\, m \,]\!] \circ [\![\, m \,]\!] \mid m \in pred_{G^*}(n)\} \sqcap Const_{NonDestructed(n)} & \text{otherwise} \end{cases}$$

This allows us to define the $PMFP_{BV}$-solution, a fixed point solution for the bitvector case, in the following fashion:

The $PMFP_{BV}$-Solution:

$PMFP_{BV(G^*,[\![\ ]\!])} : N^* \to \mathcal{F}_{\mathcal{B}}$ defined by $\forall\, n \in N^*\ \forall\, b \in \mathcal{B}.\ PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) = [\![\, n \,]\!](b)$

Like in the sequential case the $PMFP_{BV}$-strategy is practically relevant, because it can efficiently be computed (see Algorithm B.1 in Appendix B). The following theorem now establishes that it also coincides with the desired $PMOP$-solution.

Theorem 3.8 (The Parallel Bitvector Coincidence Theorem)
Let $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ be a parallel flow graph, and $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ a local semantic functional. Then we have that the PMOP-solution and the $PMFP_{BV}$-solution coincide, i.e.,

$$\forall\, n \in N^*.\ PMOP_{(G^*,[\![\ ]\!])}(n) = PMFP_{BV(G^*,[\![\ ]\!])}(n)$$

Proof. The proof follows the same pattern as for the known versions of the coincidence theorem (cf. [KS, KU]): Induction on the number of steps of the fixed point iteration for determining the $PMFP_{BV}$-solution for establishing

(A) $\quad PMOP_{(G^*,[\![\ ]\!])} \Rightarrow PMFP_{BV(G^*,[\![\ ]\!])}$

and induction on the length of a parallel path for the converse implication

(B) $\quad PMFP_{BV(G^*,[\![\ ]\!])} \Rightarrow PMOP_{(G^*,[\![\ ]\!])}$

Whereas the proof of (A) is only slightly altered, the proof of (B) requires some extra effort.

[8]Note that $[\![\ ]\!]$ is the straightforward extension of the functional defined in Definition 2.3. Thus the overloading of notation is harmless, as no reference to the sequential version is made in this definition.

Let $b \in \mathcal{B}$ and $n \in N^*$, and let us assume for (A) wlog. that $PMOP_{(G^*,[\![\ ]\!])}(n)(b) \;=\; tt$. Obviously, for every statement m, which can be executed in parallel with n, there exists a path in $\mathbf{PP}_{G^*}[\mathbf{s}^*, n)$, having m as its last component, i.e.

$$\mathbf{PP}_{G^*}[\mathbf{s}^*, n) \cap \mathbf{PP}_{G^*}[\mathbf{s}^*, m] \neq \emptyset$$

Thus $NonDestructed(n) = tt$. Now the rest of the proof is the 'standard induction' on the number of fixed point iterations mentioned above (cf. [KS, KU]), refined to take care of the distinction between 'standard' nodes and nodes taken from N_X^*, which requires the application of Theorem 3.6.

For (B), we can assume wlog. that

$$(*) \qquad PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) \;=\; tt$$

holds. In particular, this means that $NonDestructed(n) = tt$, i.e. none of the statements $m \in Pred_{G^*}^{Itlvg}(n)$ satisfies $[\![\, m \,]\!] = Const_{ff}$. Now it is the Main Lemma 3.4 which guarantees that this is already sufficient to guarantee that the standard sequential bitvector analysis is not interfered by any parallel statement. The proof is a slightly modified version of the standard induction on the path length:

Let $p = (n_1, \ldots, n_k) \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n)$ be a parallel path. Then we must show that

$$[\![\, n_k \,]\!] \circ \cdots \circ [\![\, n_1 \,]\!](b) \;=\; tt$$

In the case of $k = 0$ this is trivial, as only the start node $\mathbf{s}^*$ is reachable. Thus our assumption forces $b = tt$ as desired.

For $k \geq 1$ we need to distinguish the 'standard' cases from the case where $n \in N_X^*$. For the 'standard' case, let $0 \leq l \leq k$ be the index of the last step of p that was done by a predecessor $m \in pred_{G^*}(n)$, i.e. $[\![\, n_l \,]\!] = [\![\, m \,]\!]$. Such a step must exist, as $k \geq 1$, which excludes $n = \mathbf{s}^*$, and we know by induction:

$$PMFP_{BV(G^*,[\![\ ]\!])}(m)(b) \Rightarrow [\![\, n_{l-1} \,]\!] \circ \cdots \circ [\![\, n_1 \,]\!](b)$$

and therefore by monotonicity

$$[\![\, m \,]\!](PMFP_{BV(G^*,[\![\ ]\!])}(m)(b)) \Rightarrow [\![\, n_l \,]\!] \circ \cdots \circ [\![\, n_1 \,]\!](b)$$

On the other hand, assumption $(*)$ forces

$$[\![\, m \,]\!](PMFP_{BV(G^*,[\![\ ]\!])}(m)(b)) \;=\; tt$$

and therefore, together, as all the n_i, $l+1 \leq i \leq k$, are members of $Pred_{G^*}^{Itlvg}(n)$

$$[\![\, n_k \,]\!] \circ \cdots \circ [\![\, n_1 \,]\!] \;=\; tt$$

which completes the proof for the standard case.

Thus it remains to consider the case, where $n \in N_X^*$. In this case, assumption $(*)$ reads as follows:

$$tt \;=\; PMFP_{BV(G^*,[\![\ ]\!])}(n)(b) \;=\; [\![\, ParGraph(n) \,]\!]^* \circ [\![\, start(ParGraph(n)) \,]\!]$$

Now let $0 \leq l \leq k$ be the index corresponding to $m = start(ParGraph(n))$. Then we can apply the induction hypothesis in order to obtain:

$$PMFP_{BV(G^*,[\![\ ]\!])}(m)(b) \Rightarrow [\![\, n_{l-1} \,]\!] \circ \cdots \circ [\![\, n_1 \,]\!](b)$$

Now the application of Theorem 3.6 allows to complete the proof as in the standard case. □

3.4 Performance and Implementation

Our algorithm is based on a functional version of an *MFP*-solution, as it is common for interprocedural analyses. However, as bitvector algorithms only deal with Boolean values, proceeding argument-wise, would simply require to apply a standard bitvector algorithm twice. In particular, for regular program structures, all the nice properties of bitvector algorithms apply. In fact, for the standard version of Algorithm B.1 a single execution is sufficient, as we can start here with the same start information as the standard sequential analysis. Thus, even if we count the effort for computing the predicate *NonDestructed* separately, our analysis would simply be a composition of four standard bitvector analyses. In practice, however, our algorithm behaves much better, as the existence of a single destructing statement allows us to skip the analysis of large parts of the program. In fact, in our experience, the parallel version often runs faster than the sequential version on a program of similar size.

The same argumentation also indicates a way for a cheap implementation on top of existing bitvector algorithms. However, we recommend the direct implementation of the functional version, which to our experience, runs even faster than the decomposed standard version. This is not too surprising, as the functional version only needs to consider one additional value and does not require the argumentwise application.

4 Applications

As mentioned already, bitvector problems have a broad scope of applications ranging from simple analyses like determining whether some variable is live at a given program point or whether a program term has been computed on every program execution reaching a particular program point to more sophisticated applications like definition-use chains (cf. [He]), partial redundancy elimination (cf. [DS, DRZ, KRS1, KRS2, MR]), partial dead code elimination (cf. [KRS2]), or strength reduction (cf. [Dh, JD1, JD2, KRS3]).

Below we present the local semantic functionals of four bitvector problems in order to give a flavour of how a typical bitvector analysis looks like. Moreover, these analyses are all practically relevant, since they are the central components of two algorithms that eliminate all partially redundant computations in a program [KRS1], and remove all assignments in a program that are partially dead [KRS2], respectively.

Following [KRS1] all partial redundancies in a program can be eliminated by computing the set of program points where a computation is *up-safe*,[9] i.e., where it has been computed on every program path reaching the program point under consideration, or *down-safe*,[10] i.e., where it will be computed on every program continuation reaching the end node of the program. The DFA-problems for up-safety and down-safety are specified by the local semantic functionals $[\![\, n \,]\!]_{us}$ and $[\![\, n \,]\!]_{ds}$, respectively:

$$[\![\, n \,]\!]_{us}(b) =_{df} (b \vee Comp(n)) \wedge Transp(n) = \begin{cases} Const_{tt} & \text{if } Transp(n) \wedge Comp(n) \\ Id_{\mathcal{B}} & \text{if } Transp(n) \wedge \neg Comp(n) \\ Const_{ff} & \text{if } \neg Transp(n) \end{cases}$$

[9]Up-safety is also known as *availability*.

[10]Down-safety is also known as *very business* or *anticipability*.

$$[\![n]\!]_{ds}(b) =_{df} Comp(n) \lor (Transp(n) \land b) = \begin{cases} Const_{tt} & \text{if } Comp(n) \\ Id_{\mathcal{B}} & \text{if } \neg Comp(n) \land Transp(n) \\ Const_{ff} & \text{if } \neg Comp(n) \land \neg Transp(n) \end{cases}$$

In fact, based on the sets of up-safe and down-safe program points, the *busy code motion transformation* of [KRS1] eliminates all partial redundancies in a program.

Following [KRS2] all partially dead assignments in a program can be eliminated by successively moving assignments as far as possible in the direction of the control flow and by subsequently removing all assignments whose left hand side variable is dead after the execution of the assignment under consideration. In order to capture the second order effects of partial dead code elimination, this two step procedure is repeated until the programs eventually stabilizes. Below the local semantic functionals specifying the DFA-problems for the sinking of assignments $[\![n]\!]_{delay}$ and the detection of dead variables $[\![n]\!]_{dead}$ are presented, which are the central components of the algorithm of [KRS2]:

$$[\![n]\!]_{dead}(b) =_{df} \neg Used(n) \land (b \lor Mod(n)) = \begin{cases} Const_{tt} & \text{if } \neg Used(n) \land Mod(n) \\ Id_{\mathcal{B}} & \text{if } \neg Used(n) \land \neg Mod(n) \\ Const_{ff} & \text{if } Used(n) \end{cases}$$

$$[\![n]\!]_{delay}(b) =_{df} (b \lor Comp(n)) \land \neg LocBlocked(n)$$

$$= \begin{cases} Const_{tt} & \text{if } \neg LocBlocked(n) \land Comp(n) \\ Id_{\mathcal{B}} & \text{if } \neg LocBlocked(n) \land \neg Comp(n) \\ Const_{ff} & \text{if } LocBlocked(n) \end{cases}$$

Based on these two analyses the algorithm of [KRS2] succeeds in eliminating all assignments in a program that are partially dead.

5 Conclusions

We have shown how to construct optimal bitvector analysis algorithms for parallel programs with shared memory that are as efficient as their purely sequential counterparts, and which can easily be implemented. At the first sight, the existence of such an algorithm is rather surprising, as the interleaving semantics underlying our programming language is an indication for an exponential effort. However, the restriction to bitvector analysis constrains the possible ways of interference in such a way, that we could construct a fixed point algorithm that directly works on the parallel program without taking any interleavings into account. The algorithm is implemented on the *Fixpoint Analysis Machine* of [SBCKKMR].

References

[All] Allen, F. E. Control flow analysis. In *Proceedings of an ACM SIGPLAN Symposium on Compiler Optimization*, Urbana-Champaign, Illinois, *SIGPLAN Notices 5*, 7 (1970), 1 - 19.

[CC] 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 4^{th} International Symposium on Principles of Programming Languages (POPL'77), Los Angeles, California, 1977, 238 - 252.

[CH] Chow, J.-H., and Harrison, W. L. Compile time analysis of parallel programs that share memory. In *Conference Record of the 19^{th} International Symposium on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico, 1992, 130 - 141.

[Dh] Dhamdhere, D. M. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *Internat. J. Computer Math. 27*, (1989), 1 - 14 (+ 31 - 32).

[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.

[DS] Drechsler, K.-H., and Stadel, M. P. A variation of Knoop, Rüthing and Steffen's LAZY CODE MOTION. *SIGPLAN Notices 28*, 5 (1993), 29 - 38.

[GS] Grunwald, D., and Srinivasan, H. Data flow equations for explicitely parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Parallel Programming (PPOPP'93)*, *SIGPLAN Notices 28*, 7 (1993).

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

[JD1] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization. Part I. *Internat. J. Computer Math. 11*, (1982), 21 - 41.

[JD2] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization. Part II. *Internat. J. Computer Math. 11*, (1982), 111 - 126.

[Ki1] Kildall, G. A. Global expression optimization during compilation. Ph.D. dissertation, Technical Report No. 72-06-02, University of Washington, Computer Science Group, Seattle, Washington, 1972.

[Ki2] Kildall, G. A. A unified approach to global program optimization. In *Conference Record of the 1^{st} ACM Symposium on Principles of Programming Languages (POPL'73)*, Boston, Massachusetts, 1973, 194 - 206.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. To appear in *Transactions on Programming Languages and Systems*.

[KRS2] 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.

[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. 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.

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

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

[Ma] Marriot, K. Frameworks for abstract interpretation. *Acta Informatica 30*, (1993), 103 - 129.

[McD] McDowell, C. E. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing 6*, 3 (1989), 513 - 536.

[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. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2 (1979), 96 - 103.

[MP] Midkiff, S. P., and Padua, D. A. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Processing, Volume II*, St. Charles, Illinois, (1990), 105 - 113.

[St] Steffen, B. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming 21*, (1993), 115 - 139.

[SBCKKMR] Steffen, B., Burkart, O., Claßen, A., Klein, M., Knoop, J., Margaria, T., and Rüthing, O. The fixpoint analysis machine. Submitted for publication.

[SHW] Srinivasan, H., Hook, J., and Wolfe, M. Static single assignment form for explicitly parallel programs. In *Conference Record of the 20^{th} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, 1993, 260 - 272.

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

[SW] Srinivasan, H., and Wolfe, M. Analyzing programs with explicit parallelism. In *Proceedings of the 4^{th} International Conference on Languages and Compilers for Parallel Computing*, Santa Clara, California, Springer-Verlag, LNCS 589 (1991), 405 - 419.

[WS] Wolfe, M, and Srinivasan, H. Data structures for optimizing programs with explicit parallelism. In *Proceedings of the 1st International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, Springer-Verlag, LNCS 591 (1991), 139 - 156.

A Computing the *MFP*-Solution

Algorithm A.1 (Computing the *MFP*-Solution)

Input: *A flow graph* $G = (N, E, \mathbf{s}, \mathbf{e})$, *a local semantic functional* $[\![\ ]\!] : N \to \mathcal{F}_{\mathcal{B}}$, *and a function* $f_{init} \in \mathcal{F}_{\mathcal{B}}$ *reflecting the assumptions on the context in which the procedure under consideration is called. Usually,* f_{init} *is given by* $Id_{\mathcal{B}}$.

Output: *An annotation of* G *with functions* $[\![\, n \,]\!] \in \mathcal{F}_{\mathcal{B}}$, $n \in N$, *representing the greatest solution of the equation system of Definition 2.3. In fact, after termination of the algorithm the functional* $[\![\ ]\!]$ *satisfies:*

$$\forall\, n \in N.\ [\![\, n \,]\!] = MFP_{(G,[\![\ ]\!])}(n) = MOP_{(G,[\![\ ]\!])}(n)$$

BEGIN
    $MFP(G, [\![\ ]\!], f_{init})$
END.

where

PROCEDURE MFP $(G = (N, E, \mathbf{s}, \mathbf{e}) : SequentialFlowGraph;$
                        $[\![\ ]\!] : N \to \mathcal{F}_{\mathcal{B}} \ : LocalSemanticFunctional;$
                        $f_{start} : \mathcal{F}_{\mathcal{B}});$
VAR $f : \mathcal{F}_{\mathcal{B}}$;
BEGIN
    (Initialization of the annotation array gtr and the variable workset)
    FORALL $n \in N \backslash \{\mathbf{s}\}$ **DO** $[\![\, n \,]\!] := Const_{tt}$ **OD**;
    $[\![\, \mathbf{s} \,]\!] := f_{start}$;
    $workset := \{\, n \mid n = \mathbf{s} \vee [\![\, n \,]\!] = Const_{ff} \,\}$;

    (Iterative fixed point computation)
    WHILE $workset \neq \emptyset$ **DO**
        LET $n \in workset$
            BEGIN
                $workset := workset \backslash \{\, n \,\}$;
                $f := [\![\, n \,]\!] \circ [\![\, n \,]\!]$;
                FORALL $m \in succ_G(n)$ **DO**
                    IF $[\![\, m \,]\!] \sqsupset f$ **THEN** $[\![\, m \,]\!] := f; workset := workset \cup \{\, m \,\}$ **FI OD**
            END
    OD
END.

B Computing the $PMFP_{BV}$-Solution

Algorithm B.1 (Computing the $PMFP_{BV}$-Solution)

Input: *A parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$, *a local semantic functional* $[\![\ ]\!] : N^* \to \mathcal{F}_\mathcal{B}$, *a function* $f_{init} \in \mathcal{F}_\mathcal{B}$ *and a Boolean value* $b_{init} \in \mathcal{B}$, *where* f_{init} *and* b_{init} *reflect the assumptions on the context in which the procedure under consideration is called. Usually,* f_{init} *and* b_{init} *are given by* $Id_\mathcal{B}$ *and* ff, *respectively.*

Output: *An annotation of* G^* *with functions* $[\![\, G \,]\!]^* \in \mathcal{F}_\mathcal{B}$, $G \in \mathcal{G}_\mathcal{P}(G^*)$, *representing the semantic functions computed in step 2 of the three step procedure of Section 3.3, and with functions* $[\![\, n \,]\!] \in \mathcal{F}_\mathcal{B}$, $n \in N^*$, *representing the greatest solution of the equation system of Definition 3.7. In fact, after the termination of the algorithm the functional* $[\![\ ]\!]$ *satisfies:*

$$\forall\, n \in N^*.\ [\![\, n \,]\!] = PMFP_{BV(G^*,[\![\ ]\!])}(n) = PMOP_{(G^*,[\![\ ]\!])}(n)$$

Remark: *The global variables* $[\![\, G \,]\!]^*$, $G \in \bigcup\{\mathcal{G}_\mathcal{C}(G') \mid G' \in \mathcal{G}_\mathcal{P}(G^*)\}$, *each of which is storing a function of* $\mathcal{F}_\mathcal{B}$, *are used during the hierarchical computation of the* $PMFP_{BV}$-*solution for storing the global effect of graphs that are a component of some graph* $G \in \mathcal{G}_\mathcal{P}(G^*)$. *Additionally, the global variables* $harmful(G)$, $G \in \bigcup\{\mathcal{G}_\mathcal{C}(G') \mid G' \in \mathcal{G}_\mathcal{P}(G^*)\}$, *store whether* G *contains a node* n *with* $[\![\, n \,]\!] = Const_{ff}$. *These variables are used to compute the value of the predicate NonDestructed of Section 3.3. Finally, every flow graph* $G \in \mathcal{G}_\mathcal{P}(G^*)$ *is assumed to have a* rank, *which is recursively defined by:*

$$rank(G) =_{df} \begin{cases} 0 & \text{if } G \in {\mathcal{G}_\mathcal{P}}^{min}(G^*) \\ max\{\, rank(G') \mid G' \in \mathcal{G}_\mathcal{P}(G^*) \wedge G' \subset G \,\} + 1 & \text{otherwise} \end{cases}$$

where

$${\mathcal{G}_\mathcal{P}}^{min}(G^*) =_{df} \{\, G \in \mathcal{G}_\mathcal{P}(G^*) \mid \forall\, G' \in \mathcal{G}_\mathcal{P}(G^*).\ G' \subseteq G \Rightarrow G' = G \,\}$$

denotes the set of minimal *graphs of* $\mathcal{G}_\mathcal{P}(G^*)$.

BEGIN
  (Synchronization: Computing $[\![\, G \,]\!]^*$ *for all* $G \in \mathcal{G}_\mathcal{P}(G^*)$ *)*
  $GLOBEFF(G^*, [\![\ ]\!])$;

  (Interleaving: Computing the $PMFP_{BV}$*-Solution* $[\![\, n \,]\!]$ *for all* $n \in N^*$ *)*
  $PMFP_{BV}(G^*, [\![\ ]\!], f_{init}, b_{init})$
END.

where

PROCEDURE $GLOBEFF$ ($\underline{G = (N, E, \mathbf{s}, \mathbf{e})} : ParallelFlowGraph$;
            $\overline{[\![\ ]\!]} : N \to \mathcal{F}_\mathcal{B}$: $LocalSemanticFunctional$);
VAR i : *integer*;
BEGIN
  FOR $i := 0$ **TO** $rank(G)$ **DO**
    FORALL $G' \in \{G''_{seq} \mid G'' \in \mathcal{G}_\mathcal{P}(G) \wedge rank(G'') = i\}$ **DO**

FORALL $G'' \in \mathcal{G}_{\mathcal{C}}(G')$ *where* $G'' = (N'', E'', \mathbf{s}'', \mathbf{e}'')$ **DO**

LET $\forall\, n \in N''.\ [\![\, n \,]\!]'' = \begin{cases} Id_{\mathcal{B}} \sqcap Const_{\forall\, \bar{G} \in \mathcal{G}_{\mathcal{C}}(pfg(n)).\ \neg harmful(\bar{G})} & \textit{if } n \in N^*_N \\ [\![\, pfg(n) \,]\!]^* & \textit{if } n \in N^*_X \\ \overline{[\![\, n \,]\!]} & \textit{otherwise} \end{cases}$

BEGIN

$harmful(G'') := (\ |\{\, n \in N'' \mid [\![\, n \,]\!]'' = Const_{\mathit{ff}} \,\}| \geq 1\,);$

$MFP(G'', [\![\ ]\!]'', Id_{\mathcal{B}});$

$[\![\, G'' \,]\!]^* := [\![\, end(G'') \,]\!]^*$

END

OD;

$[\![\, G' \,]\!]^* := \bigsqcap\{\, [\![\, G'' \,]\!]^* \mid G'' \in \mathcal{G}_{\mathcal{C}}(G') \,\}$

OD

OD

END.

PROCEDURE $PMFP_{BV}$ $(G = (N, E, \mathbf{s}, \mathbf{e}) : ParallelFlowGraph;$
$\overline{[\![\ ]\!]} : N \to \mathcal{F}_{\mathcal{B}}\ : LocalSemanticFunctional;$
$f_{start} : \mathcal{F}_{\mathcal{B}};$
$harmful : \mathcal{B});$

VAR $f : \mathcal{F}_{\mathcal{B}};$

BEGIN

IF *harmful* **THEN FORALL** $n \in N$ **DO** $[\![\, n \,]\!] := Const_{\mathit{ff}}$ **OD**

ELSE

(Initialization of the annotation arrays $[\![\ ]\!]$ *and the variable workset)*

FORALL $n \in N \backslash \{\mathbf{s}\}$ **DO** $[\![\, n \,]\!] := Const_{tt}$ **OD**;

$[\![\, \mathbf{s} \,]\!] := f_{start};$

$workset := \{\, n \mid n = \mathbf{s} \vee [\![\, n \,]\!]' = Const_{\mathit{ff}} \,\};$

(Iterative fixed point computation)

WHILE $workset \neq \emptyset$ **DO**

LET $n \in workset$

BEGIN

$workset := workset \backslash \{\, n \,\};$

IF $n \in N \backslash N^*_N$

THEN

$f := \overline{[\![\, n \,]\!]} \circ [\![\, n \,]\!];$

FORALL $m \in succ_G(n)$ **DO**

IF $[\![\, m \,]\!] \sqsupset f$ **THEN** $[\![\, m \,]\!] := f;\ workset := workset \cup \{\, m \,\}$ **FI**

OD

ELSE

FORALL $G' \in \mathcal{G}_{\mathcal{C}}(ParGraph(n))$ **DO**

$PMFP_{BV}(G', \overline{[\![\ ]\!]}, [\![\, n \,]\!], \sum_{G'' \in \mathcal{G}_{\mathcal{C}}(ParGraph(n)) \backslash \{G'\}} harmful(G''))$ **OD**;

$f := [\![\, ParGraph(n) \,]\!]^* \circ [\![\, n \,]\!];$

IF $[\![\, end(ParGraph(n)) \,]\!] \sqsupset f$

THEN

$[\![\, end(ParGraph(n)) \,]\!] := f;$

$workset := workset \cup \{ \, end(ParGraph(n)) \, \}$

FI

FI

END

OD

FI

END.

Let $[\![\, n \,]\!]_{alg}$, $n \in N^*$, denote the final values of the corresponding variables after the termination of Algorithm B.1, and $[\![\, n \,]\!]$, $n \in N^*$, the greatest solution of the equation system of Definition 3.7, then we have:

Theorem B.2 $\forall \, n \in N^*. \; [\![\, n \,]\!]_{alg} = [\![\, n \,]\!]$

C Illustrating Figures

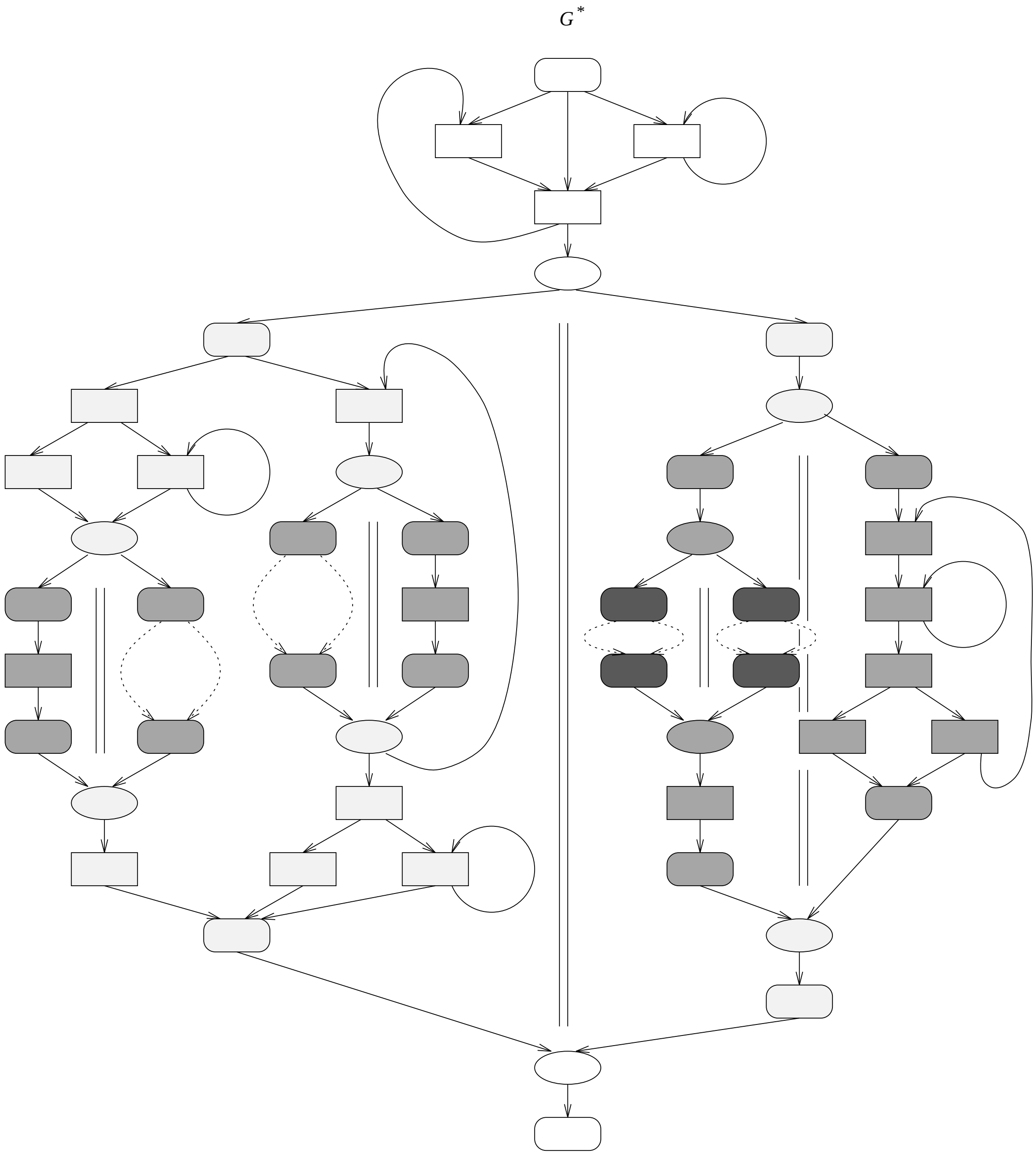

Figure 3: The Parallel Flow Graph G^*

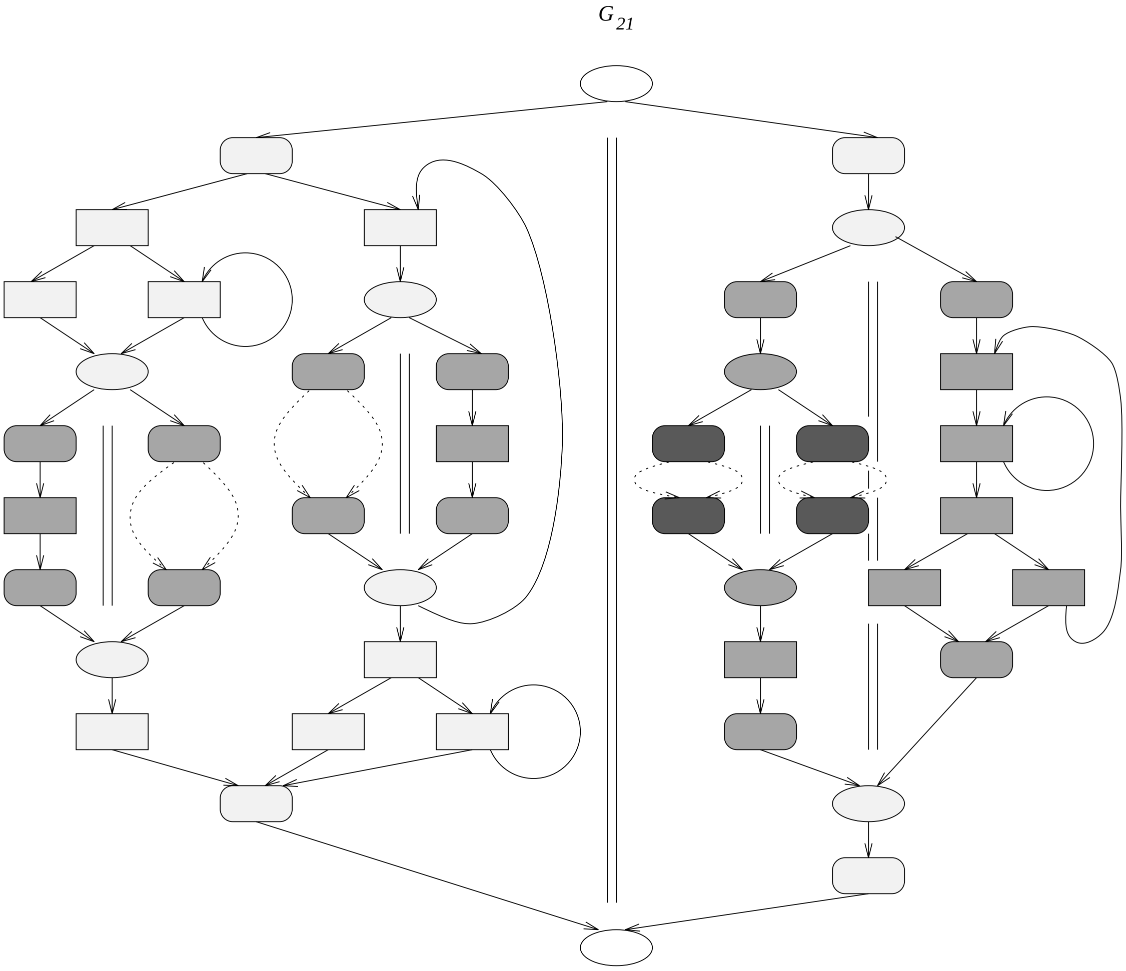

Figure 4: $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \wedge rank(G) = 2\} = \{G_{21}\}$

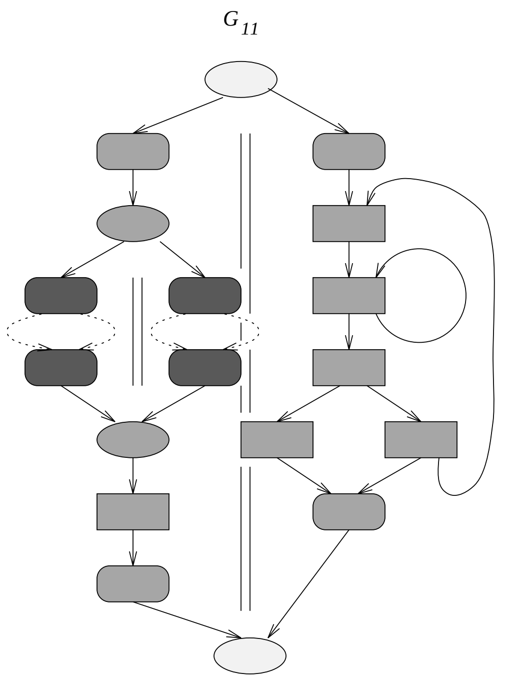

Figure 5: $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \,\wedge\, rank(G) = 1\} \;=\; \{G_{11}\}$

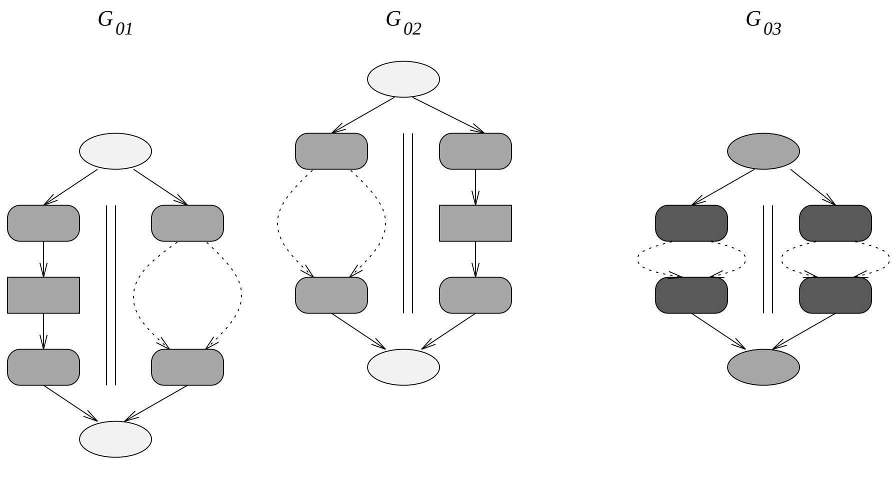

Figure 6: $\{G \mid G \in \mathcal{G}_{\mathcal{P}}(G^*) \,\wedge\, rank(G) = 0\} \;=\; \{G_{01}, G_{02}, G_{03}\}$

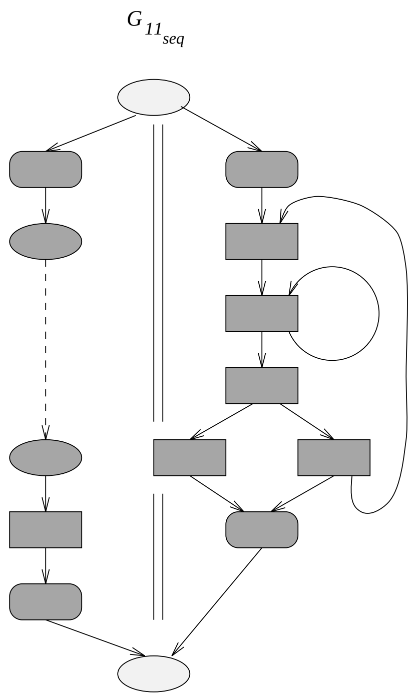

Figure 7: $\{G_{seq} \,|\, G \in \mathcal{G}_{\mathcal{P}}(G^*) \,\wedge\, rank(G) = 1\} \;=\; \{G_{11_{seq}}\}$

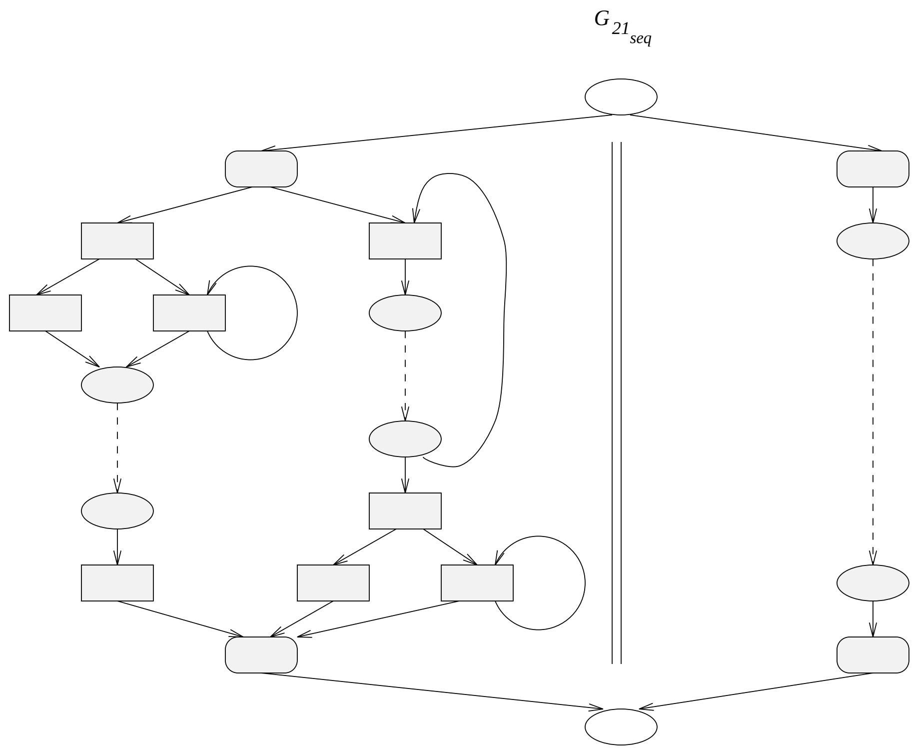

Figure 8: $\{G_{seq} \,|\, G \in \mathcal{G}_{\mathcal{P}}(G^*) \,\wedge\, rank(G) = 2\} \;=\; \{G_{21_{seq}}\}$

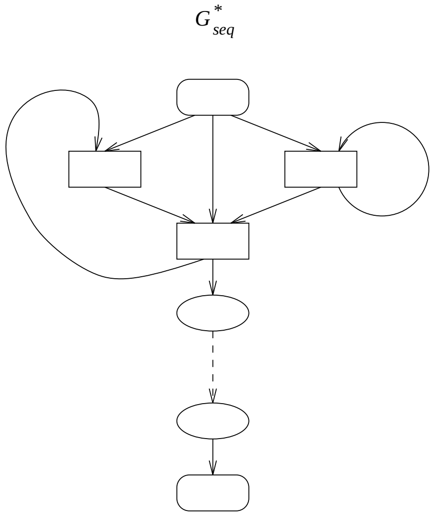

Figure 9: G^*_{seq}

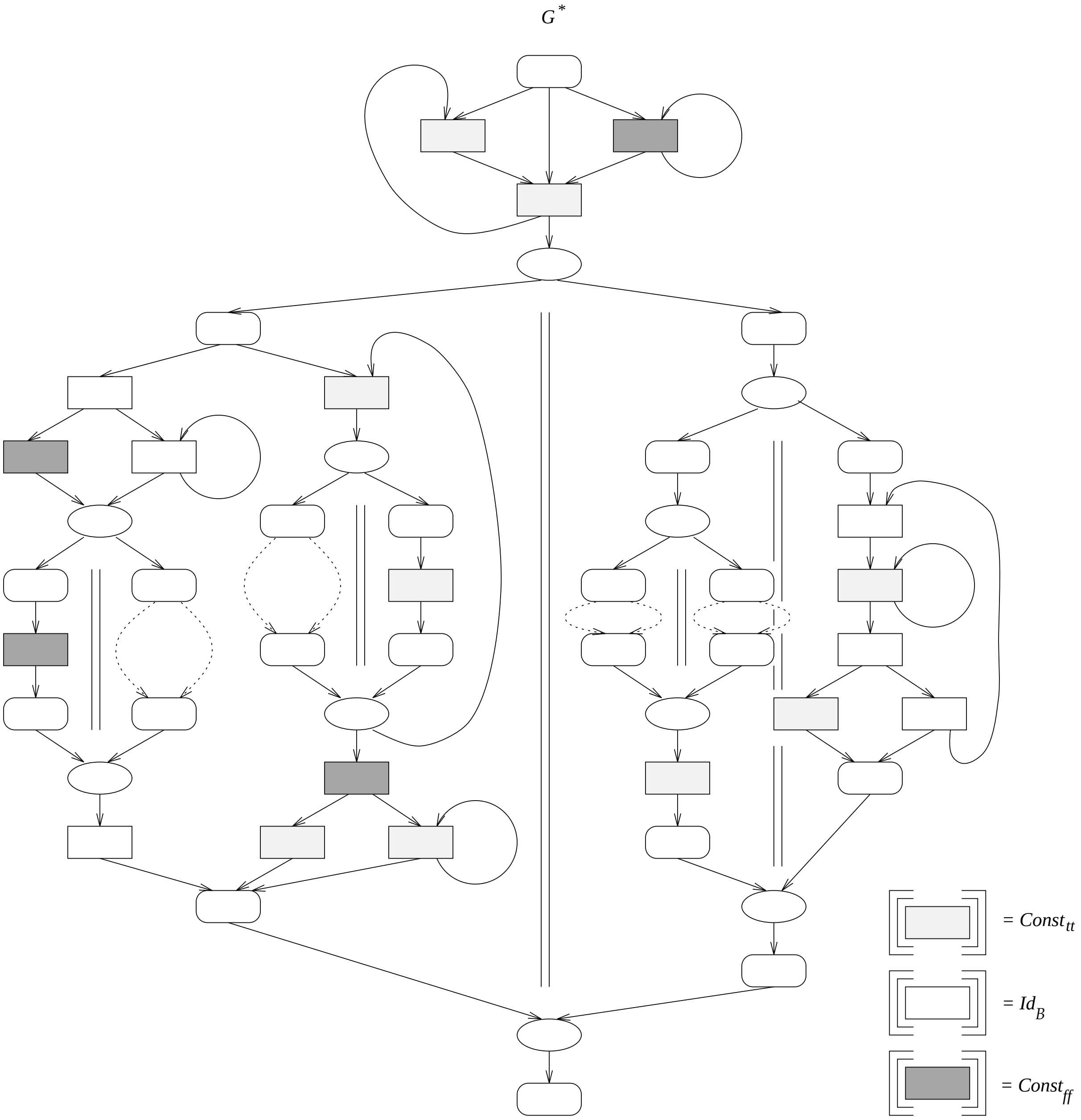

Figure 10: The Parallel Flow Graph G^* with a Local Semantic Functional $[\![\ ]\!]$

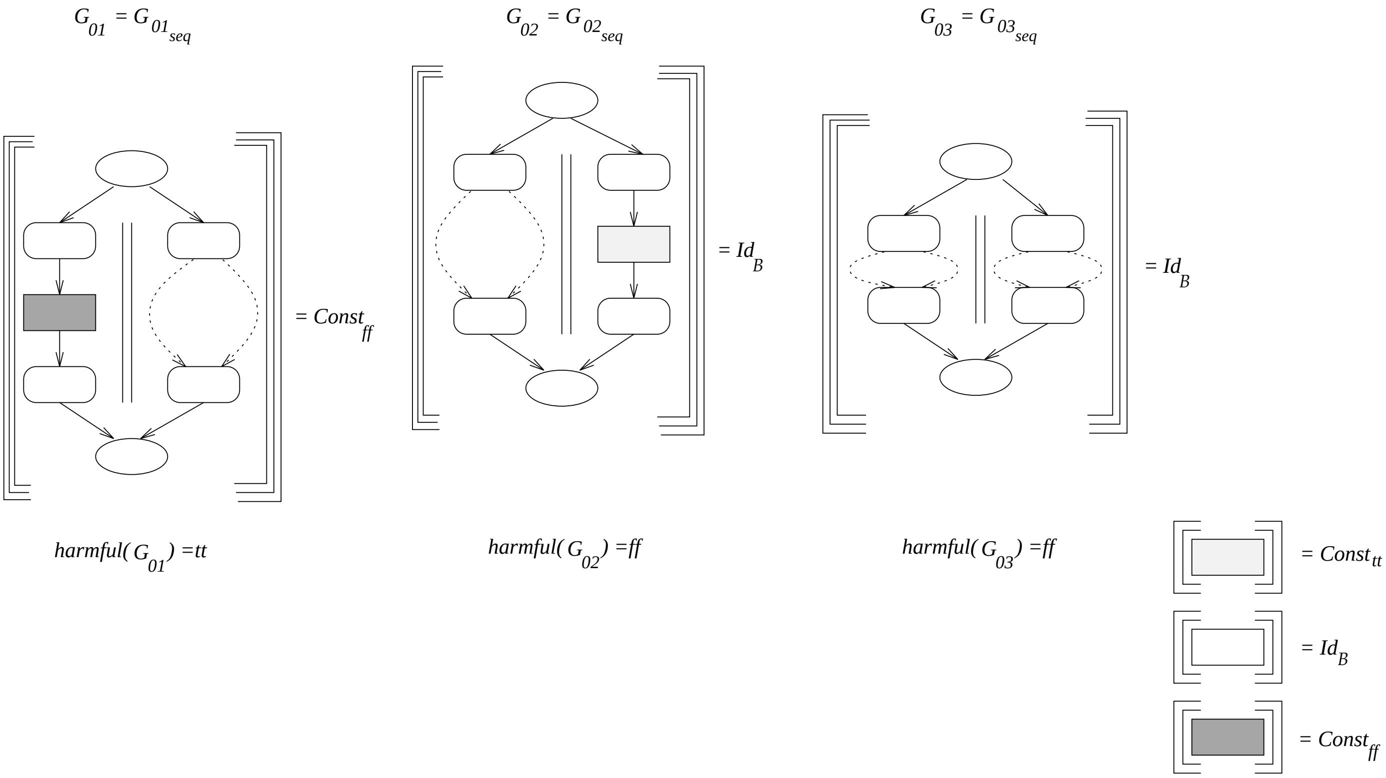

Figure 11: After the First Iteration of the Outermost For-Loop of Procedure GLOBEFF

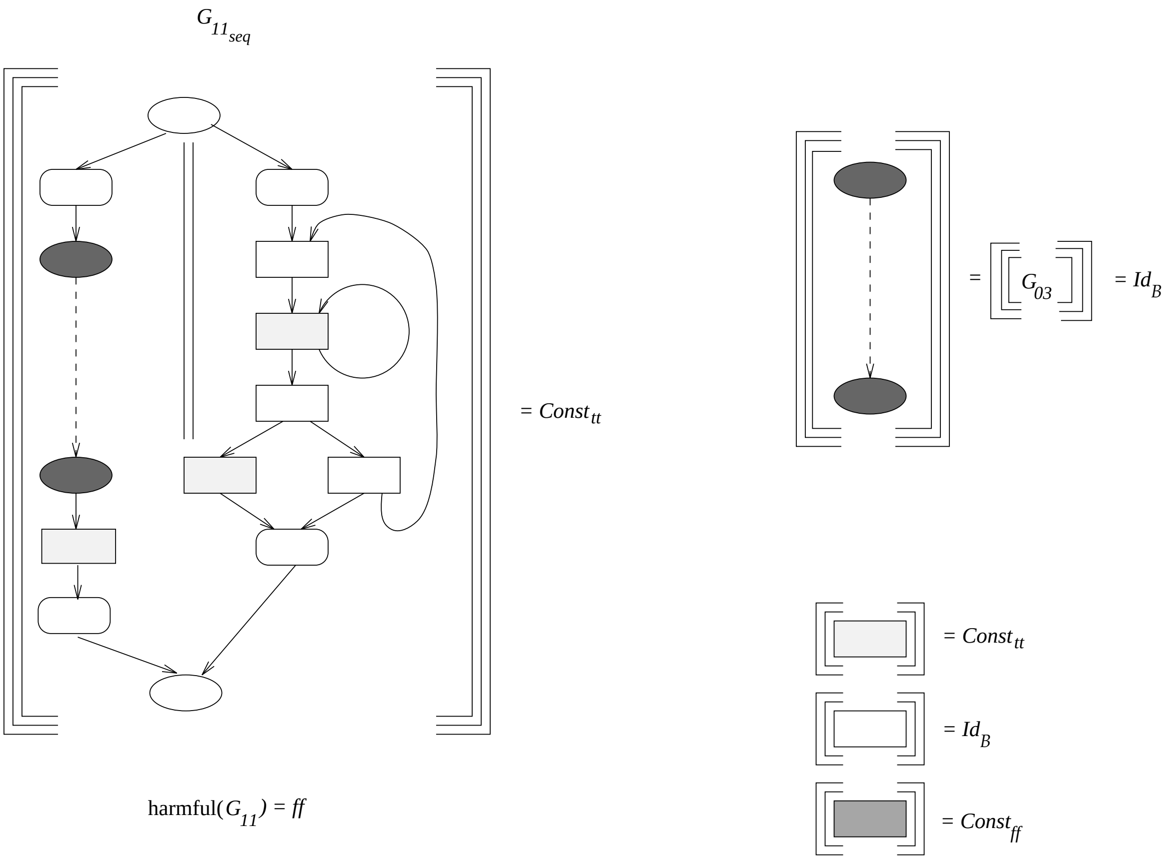

Figure 12: After the Second Iteration of the Outermost For-Loop of Procedure GLOBEFF

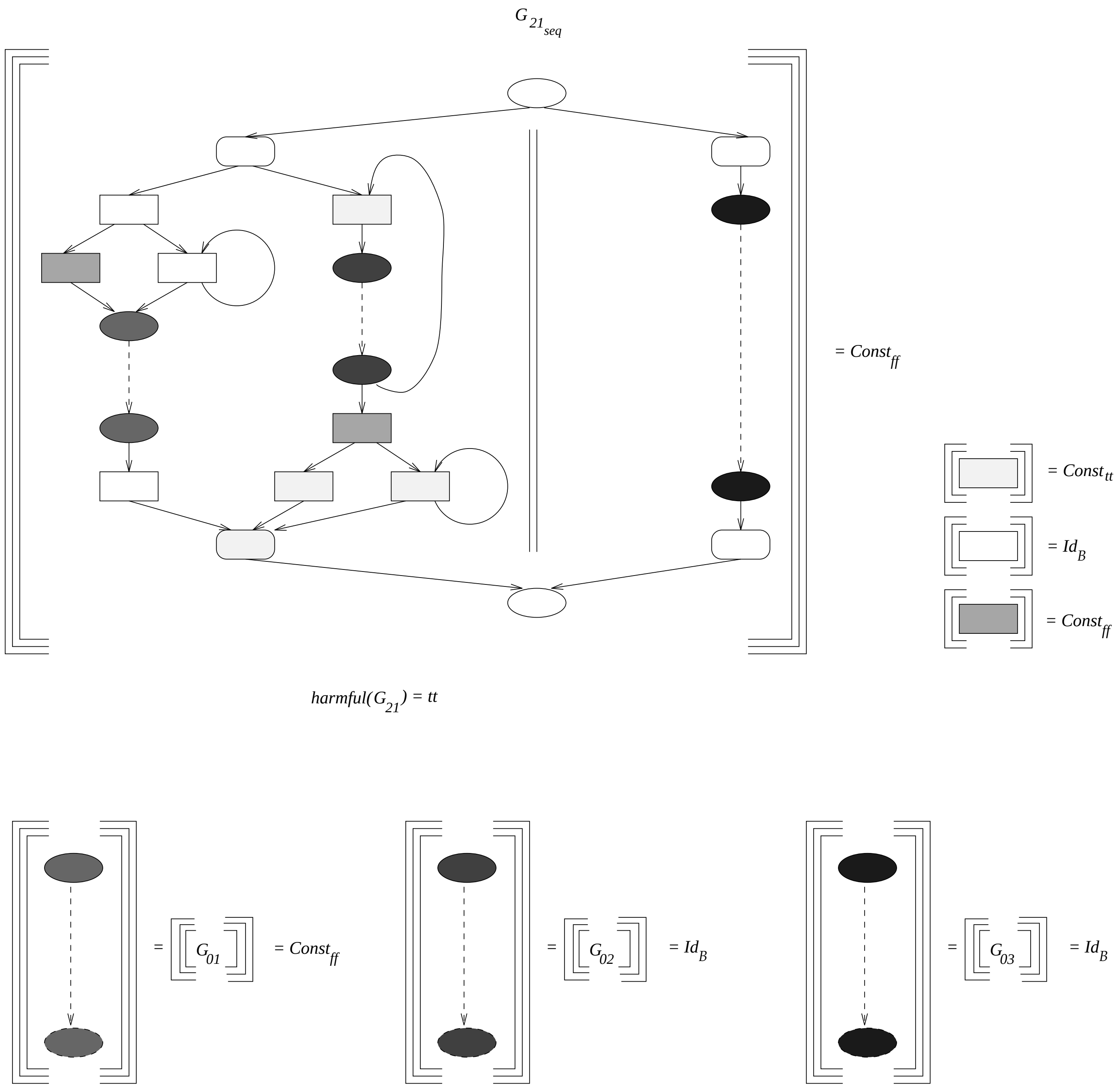

Figure 13: After the Third Iteration of the Outermost For-Loop of Procedure GLOBEFF

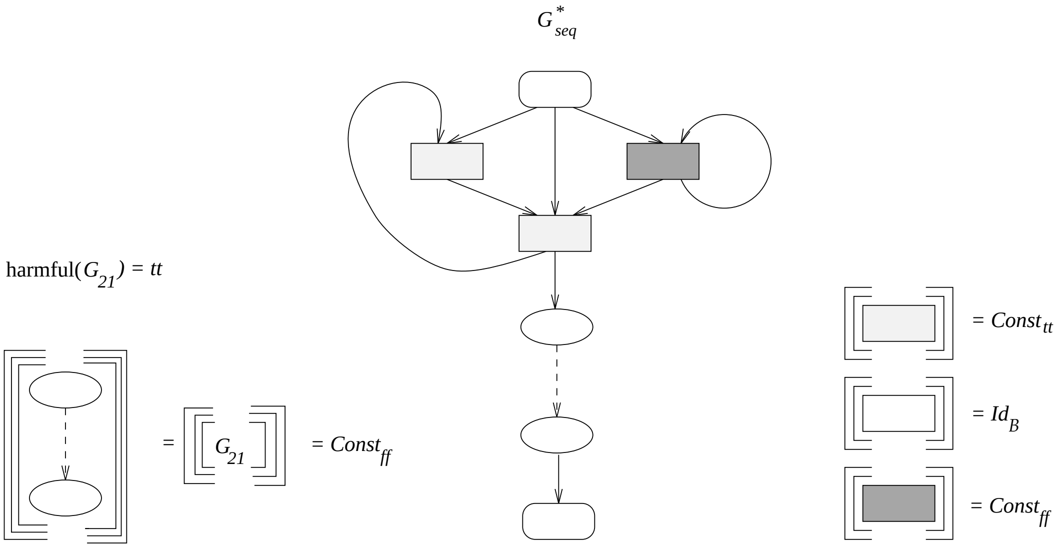

Figure 14: The Flow Graph G^*_{seq} with the adapted Semantic Functional