Non-monotone Fixpoint Iterations to Resolve Second Order Effects

Alfons Geser,[1] Jens Knoop,[1] Gerald Lüttgen,[2] Oliver Rüthing,[3] Bernhard Steffen[1]

Technical Report TR-96-01
Department of Computer Science
North Carolina State University
March 1996

---

[1]Fakultät für Mathematik und Informatik, Universität Passau, Innstraße 33, D–94032 Passau, Germany.

[2]Department of Computer Science, N.C. State University, Raleigh, NC 27695-8206, USA. The author is supported by the German Academic Exchange Service under grant D/95/09026 (Doktorandenstipendium HSP II / AUFE).

[3]Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Preußerstraße 1–9, D–24105 Kiel, Germany.

This paper appears in *Proceedings of the 6th International Symposium on Compiler Construction (CC'96)*, Linköping, Sweden, April 24 - 26, 1996.

ABSTRACT We present a new *fixpoint theorem* which guarantees the existence and the finite computability of the least common solution of a countable system of recursive equations over a *wellfounded* domain. The functions are only required to be *increasing* and *delay-monotone*, the latter being a property much weaker than monotonicity. We hold that wellfoundedness is a natural condition as it guarantees termination of every fixpoint computation algorithm. Our fixpoint theorem covers, under the wellfoundedness condition, all the known 'synchronous' versions of fixpoint theorems. To demonstrate its power and versatility we contrast an application in *data flow analysis*, where known versions are applicable as well, to a practically relevant application in *program optimization*, which due to its second order effects, requires the full strength of our new theorem. In fact, the new theorem is central for establishing the optimality of the *partial dead code elimination* algorithm considered, which is implemented in the new release of the Sun SPARCompiler[4] language systems.

Keywords: Data Flow Analysis, Program Optimization, Fixpoint Computation, Second Order Effect, Non-monotone Function, Chaotic Iteration, Partial Dead Code Elimination, Code Motion, Workset Algorithm.

---

[4]SPARCompiler is a registered trademark of SPARC International, Inc., and is licensed exclusively to Sun Microsystems, Inc.

1 Motivation and Related Work

Many practically relevant problems in computer science can be characterized by means of the least common solution of a system of *recursive equations*

$$\begin{aligned} x &= f_1(x) \\ &\vdots \\ x &= f_n(x) \end{aligned}$$

where $\mathcal{F} =_{df} \{f_k : D \to D \mid 1 \leq k \leq n\}$ is a family of *monotone* functions on a *wellfounded partial order* $\langle D; \sqsubseteq \rangle$. Solving this system of equations is equivalent to the computation of a fixpoint of $\mathcal{F}$, i.e. a *common fixpoint* $x = f_k(x)$ of all f_k. A typical *iteration* algorithm starts with the initial value $\bot$, the smallest element of D, and successively updates the value of x applying the functions f_k in an arbitrary order, so as to approximate the least fixpoint of $\mathcal{F}$. People speak of a *chaotic iteration*.

The origin of fixpoint theorems in computer science dates back to the fundamental work of Tarski [20]. Tarski's theorem considers a monotone function and guarantees the existence of its least fixpoint with respect to a complete partial order. This setup, however, turned out to be too restrictive for a lot of practically relevant applications which led to a number of generalizations. See [14] for a survey of the history of fixpoint theory.

Vector iteration [18] provides such a generalization, where one computes the least fixpoint $\vec{x} = (x^1, \ldots, x^m) \in D^m$ of a monotone vector function $\vec{f} = (f^1, \ldots, f^m)$. Liberalizing Tarski's iteration $\vec{x}_0 = \vec{\bot}, \vec{x}_1 = \vec{f}(\vec{x}_0), \vec{x}_2 = \vec{f}(\vec{x}_1), \ldots$, where $\vec{x}_i$ denotes the value of $\vec{x}$ after the i-th iteration, one may choose $\vec{x}_0 = \vec{\bot}, \vec{x}_1 = \vec{f}_{J_0}(\vec{x}_0), \vec{x}_2 = \vec{f}_{J_1}(\vec{x}_1), \ldots$, where $J_i \subseteq \{1, \ldots, n\}$ and the k-th component $\vec{f}_{J_i}(\vec{x}_i)^k$ of $\vec{f}_{J_i}(\vec{x}_i)$ is $f^k(\vec{x}_i)$ if $k \in J_i$ and $\vec{x}_i^k$ otherwise. Intuitively, at each step i the set J_i denotes the indices k of the components which are updated. It is known that a *fairness condition* for the J_i, which guarantess that every function f^k is considered sufficiently often, is mandatory. Considering the vectors $\vec{x}$ as objects and the update operations $\vec{f}_J$ as functions, we have a clear instance of the chaotic iteration above (see also Section 3). Recent contributions to fixpoint theory provide efficient strategies for vector iteration, e.g. by using demand driven evaluation strategies (cf. [22, 8]).

The vector approach has been further generalized towards *asynchronous iterations* [1, 3, 21, 23], where $\vec{f}_{J_i}$ may use components of a choice of earlier vectors $\vec{x}_j$, where $j \leq i$, of the iteration.

Despite its power the vector iteration approach turns out to be too restrictive in two aspects. First, the functions involved in the fixpoint iteration may be such that they cannot be regarded as components of a single func-

tion f. To our knowledge, the only serious attack to this problem has been made by P. and R. Cousot [5]. The common fixpoints of a family $\mathcal{F} =_{df} (f_k)_{k \in \mathbb{N}}$ of monotonic functions are described by iterations, given that each pair f_k, $f_{k'}$ commutes: $f_k(f_{k'}(x)) = f_{k'}(f_k(x))$ for all $x \in D$.

Second, and even worse, program transformations may have *second order effects* (cf. Rosen, Wegman and Zadeck [19]). Typically, program transformations are not idempotent; one transformation may have a strong impact on the profitability of another transformation; often the transformation functions involved are no longer monotone. Then none of the known fixpoint theorems apply.

In this paper we offer a new fixpoint theorem which does not require monotonicity. Given $d \sqsubseteq d'$, monotonicity amounts to show $f_k(d) \sqsubseteq f_k(d')$. Instead, we allow that the expression $f_k(d')$ may be replaced by $f_j(d')$ for any j, and even by some *arbitrary composition* of functions applied to d'. If the functions are increasing, i.e. $x \sqsubseteq f_k(x)$ holds for all $k \in \mathbb{N}$, this task becomes the easier to solve, the longer the compositions are.

We require only two very weak conditions for technical convenience. First, we require that the underlying domain is *wellfounded*, a condition which is reasonable in practice, because it means termination of the iteration. In fact we are confident that wellfoundedness is not essential if one can afford nontermination. Second, we require that all functions in $\mathcal{F}$ are *increasing*. This requirement is not really restrictive, as we show in Sections 2 and 3.

Our fixpoint theorem is applicable to arbitrary countable *families of functions* $\mathcal{F} =_{df} \{f_k : D \to D \mid k \in \mathbb{N}\}$ on wellfounded partial orders $\langle D; \sqsubseteq \rangle$: Under the above mentioned premises our theorem guarantees the existence of a least common fixpoint of $\mathcal{F}$, which is reached eventually by any *fair* chaotic iteration (cf. Definition 2.1).

The remainder of the paper is structured as follows. We present the new fixpoint theorem in Section 2. In Section 3 we show that *vector iterations* are a special case of chaotic iterations. Section 4 demonstrates the power of our theorem by applying it to a classical data flow analysis algorithm, and by treating a problem beyond the scope of classical fixpoint theorems: the proof of the optimality of a program optimization for *partial dead code elimination* [12], which is composed of program transformations having second order effects. This algorithm is implemented in Version 4.0 of the Sun SPARCompiler language systems, which underlines the practical relevance of the new fixpoint theorem. Section 5 contains our conclusions and directions to future work. Finally, the appendix contains all technical proofs of the paper.

2 The Fixpoint Theorem

In this section, we present our new fixpoint theorem, which guarantess under certain premises that a family of functions, $\mathcal{F} =_{df} (f_k)_{k \in \mathbb{N}}$, has a least common fixpoint $\mu\mathcal{F}$, together with a corresponding 'generic' terminating algorithm. This requires the following basic notions.

A *partial order* $\langle D; \sqsubseteq \rangle$ is a set D together with a reflexive, antisymmetric, and transitive binary relation $\sqsubseteq \subseteq D \times D$. A sequence $(d_i)_{i \in \mathbb{N}}$ of elements $d_i \in D$ is called an *(ascending) chain* if $\forall i \in \mathbb{N}.\ d_i \sqsubseteq d_{i+1}$. A chain $T =_{df} (d_i)_{i \in \mathbb{N}}$ is *stationary* if $\{d_i \mid i \in \mathbb{N}\}$ is finite. The partial order relation $\sqsubseteq$ is called *wellfounded* if every chain is stationary. A function $f : D \to D$ on D is *increasing* if $d \sqsubseteq f(d)$ for all $d \in D$, and *monotone* if $\forall d, d' \in D.\ d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$. If $\mathcal{F} =_{df} (f_k)_{k \in \mathbb{N}}$ is a family of functions and $s = (s_1, \ldots, s_n) \in \mathbb{N}^*$ then f_s is defined by the composition $f_s =_{df} f_{s_n} \circ \cdots \circ f_{s_1}$.

The following notions are central for dealing with fixpoint iterations of a family of functions.

Definition 2.1 (Strategy, Chaotic Iteration Sequence, Fairness) *Let $\langle D; \sqsubseteq \rangle$ be a partial order and $\mathcal{F} =_{df} (f_k)_{k \in \mathbb{N}}$ be a family of increasing functions $f_k : D \to D$. A* strategy *is any function $\gamma : \mathbb{N} \to \mathbb{N}$. A strategy γ and an element $d \in D$ induce a* chaotic iteration *$f_\gamma(d) = (d_i)_{i \in \mathbb{N}}$ of elements $d_i \in D$ inductively defined by $d_0 = d$ and $d_{i+1} = f_{\gamma(i)}(d_i)$. A strategy γ is called* fair *iff*

$$\forall i, k \in \mathbb{N}.\ (f_k(d_i) \neq d_i \ \textit{implies}\ \exists j > i.\ d_j \neq d_i)$$

Fixpoint theorems usually require that the considered functions are monotone. In practice, however, functions are often *not* monotone, but satisfy the following weaker notion.

Definition 2.2 (Delay-Monotonicity) *Let $\langle D; \sqsubseteq \rangle$ be a partial order and $\mathcal{F} =_{df} (f_k)_{k \in \mathbb{N}}$ be a family of functions $f_k : D \to D$. Then $\mathcal{F}$ is called* delay-monotone*, if for all $k \in \mathbb{N}$:*

$$d \sqsubseteq d' \ \textit{implies}\ \exists s \in \mathbb{N}^*.\ f_k(d) \sqsubseteq f_s(d')$$

If every f_k is a *monotone* function in the usual sense, then $\mathcal{F}$ is delay-monotone. Note that delay-monotonicity does in general *not* carry over to proper subsets of $\mathcal{F}$.

Now, we are prepared for our main result, which, in particular, yields that $\bigsqcup f_\gamma(\bot)$ is independent of the choice of γ.

Theorem 2.3 (Chaotic Fixpoint Iterations) *Let $\langle D; \sqsubseteq \rangle$ be a wellfounded partial order with least element $\bot$, let $\mathcal{F}$ be a delay-monotone family $(f_k)_{k \in \mathbb{N}}$ of increasing functions, and let $\gamma : \mathbb{N} \to \mathbb{N}$*

```
d := ⊥;
while ∃ k ∈ ℕ. d ≠ f_k(d) do
    choose k ∈ ℕ where d ⊏ f_k(d) in
      d := f_k(d)
    ni
od
```

FIGURE 1. The Nondeterministic Skeleton Algorithm

be a fair strategy. Then the least common fixpoint $\mu\mathcal{F}$ of $\mathcal{F}$ exists and is given by $\bigsqcup f_\gamma(\bot)$. In particular, $\mu\mathcal{F}$ is always reached within a finite number of iteration steps.

Note that the following counterexample shows that "increasing" is essential. Let $\bot \sqsubset a$, and $f_1(\bot) = f_1(a) = \bot$, $f_2(\bot) = f_2(a) = a$, i.e. both functions are monotone, but f_1 is not increasing. Indeed, f_1 and f_2 have no common fixpoints.

Theorem 2.3 suggests an iterative strategy for computing the least fixpoint of $\mathcal{F}$. One defines $\gamma(i)$ at step i during the run of the algorithm. Whenever d_i is not yet a fixpoint of $\mathcal{F}$, i.e. there is some $k \in \mathbb{N}$ where $f_k(d_i)$ is strictly greater than d_i, one chooses $\gamma(i) = k$ for an arbitrary such k. This idea is illustrated in the nondeterministic skeleton algorithm presented in Figure 1.

3 Special Case: Vector Iterations

Let $\langle C; \sqsubseteq_C \rangle$ be a wellfounded partial order and $D = C^n$ for some $n \in \mathbb{N}$, ordered by the pointwise extension $\sqsubseteq$ of $\sqsubseteq_C$. Now let $f : D \to D$ be a monotone function. Instead of iterating $d_1 = f(\bot), d_2 = f(d_1), \ldots$ according to Tarski's theorem, one may pass over to a dissection of f to its components, f^k, i.e. $f(d) = (f^1(d), \ldots, f^n(d))$ and perform selective updates. Here and in the sequel we use an upper index i at a vector of length n to select its i-th component. A *vector iteration* is an iteration of the form $d_1 = f_{J_0}(\bot), d_2 = f_{J_1}(d_1), \ldots$, where $J_i \subseteq \{1, \ldots, n\}$ and

$$f_J(d)^i =_{df} \begin{cases} f^i(d) & \text{if } i \in J \\ d^i & \text{otherwise} \end{cases}$$

performs a selective update of the components specified by J. The set of common fixpoints of the function family $\mathcal{F} =_{df} \{f_J \mid J \subseteq \{1, \ldots, n\}\}$ is equal to the set of fixpoints of f. Note that each f_J is monotone since f is monotone.

Now let us demonstrate that the vector approach is modelled conveniently in our setting. To this end, we generalize the notion of a *strategy* to that of a *set strategy*. A *set strategy* is any function $\gamma : \mathbb{N} \to \mathfrak{P}(\{1, \ldots, n\})$. The intended meaning being that $\gamma(i)$ yields a set J_i of indices in $\{1, \ldots, n\}$ of components to be updated at step i. A set strategy is called *fair*, iff

$$\forall i \in \mathbb{N},\, J \subseteq \mathbb{N}.(f_J(d_i) \neq d_i \text{ implies } \exists j > i.\, d_j \neq d_i)$$

The following result shows that for a *monotone* vector function f, every chaotic iteration sequence is a chain, i.e. a totally ordered subset of D.

Lemma 3.1 (Vector Iterations)
Let $\langle C; \sqsubseteq_C \rangle$ be a wellfounded partial order with least element $\bot_C$, let $n \in \mathbb{N}$, and let $D = C^n$ be ordered by the pointwise extension $\sqsubseteq$ of $\sqsubseteq_C$. Let $f = (f^1, \ldots, f^n)$ be a monotone function on D, and let $\mathcal{F} =_{df} \{f_J \mid J \subseteq \{1, \ldots, n\}\}$, where the functions $f_J : D \to D$ are defined as above, and let $\gamma : \mathbb{N} \to \mathfrak{P}(\{1, \ldots, n\})$ be a set strategy. Then every chaotic iteration $f_\gamma(\bot)$ is a chain.

Without loss of generality we may assume that D is the smallest set that contains $\bot$ and is closed under $\mathcal{F}$ and $\bigsqcup$. Then increasingness means exactly that every iteration yields a chain. In other words, for vector iterations the increasingness property is no real restriction.

The following corollary is a special case of Theorem 2.3 for vector iterations and a consequence of Lemma 3.1. In particular, if $|\mathcal{F}| = 1$ our corollary reduces to Tarski's theorem [20] in the case of wellfounded partial orders.

Corollary 3.2 (Chaotic Vector Iterations)
Let $\langle C; \sqsubseteq_C \rangle$ be a wellfounded partial order with least element $\bot_C$, let $n \in \mathbb{N}$, and let $D = C^n$ be ordered by the pointwise extension $\sqsubseteq$ of $\sqsubseteq_C$. Let $f = (f^1, \ldots, f^n)$ be a monotone function on D, let $\mathcal{F} =_{df} \{f_J \mid J \subseteq \{1, \ldots, n\}\}$, and let γ be a fair set strategy. Then $\bigsqcup f_\gamma(\bot)$ is the least fixpoint $\mu\mathcal{F}$ of $\mathcal{F}$. In particular, $\mu\mathcal{F} = \mu f$, and $\mu\mathcal{F}$ is always reached within a finite number of iteration steps.

4 Applications

In this section, we demonstrate our Fixpoint Theorem 2.3 by proving the correctness and termination of a workset algorithm for *data flow analysis*, and by establishing terminating optimal *program optimization* on the basis of program transformations with second order effects. Whereas the first application can already be handled by Corollary 3.2, which reflects the scope of classical vector iteration approaches as they are common in practice, the second application requires the full strength of our main Theorem 2.3

since the component transformations of the optimization, the algorithm for *partial dead code elimination* of [12], are not even monotone on the relevant domain. Here, the new theorem is central for establishing the optimality of this algorithm, which is implemented in Version 4.0 of the Sun SPARCompiler language systems.

4.1 Data Flow Analysis: Workset Algorithms

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. [7, 16]). For imperative languages, it provides information about the program states that may occur at a given program point during execution. Usually, this information is computed by means of some iterative workset algorithm, which can elegantly be modelled by the vector iteration approach.

In DFA and program optimization (cf. Section 4.2) it is common to represent programs as *directed flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent the statements, edges $(n, m) \in E$ the nondeterministic branching structure of the program under consideration, and $\mathbf{s}$ and $\mathbf{e}$ the unique *start node* and *end node* of G, which are assumed to possess no predecessors and successors, respectively. Moreover, $pred_G(n) =_{df} \{ m \mid (m, n) \in E \}$ 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. Finally, every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$, i.e. every node $n \in N$ is reachable from $\mathbf{s}$, and $\mathbf{e}$ is reachable from every node $n \in N$.

Theoretically wellfounded are DFAs that are based on *abstract interpretation* (cf. [4, 15]). The point of this approach is to replace the "full" semantics of a program by a simpler more abstract version, which is tailored to deal with a specific problem. Usually, the abstract semantics is specified by means of a *local semantic functional*

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

which gives abstract meaning to every program statement in terms of a monotone (or even continuous) transformation function on a wellfounded partial order $\langle \mathcal{C}; \sqsubseteq \rangle$ with least element $\bot$, whose elements express the DFA-information of interest.

Given a program G and a local abstract semantics $[\![\ ]\!]$, the goal of DFA is to annotate the program points of G with DFA-information that properly reflect the run-time behaviour of G with respect to the problem under consideration. Formally, this annotation is defined by the least solution of Equation System 4.1 which specifies the consistency between pre-conditions of the statements of G expressed in terms of $\mathcal{C}$ with respect to some start information $c_0 \in \mathcal{C}$. This annotation is known as the solution of the *minimal fixpoint (MFP)* approach in the sense of Kam and Ullman [9].

```
pre[s] := c_0;
forall n ∈ N\{s} do pre[n] := ⊥ od;
workset := N;
while workset ≠ ∅ do
    choose n ∈ workset in
        workset := workset\{ n };
        new := pre[n] ⊔ ⨆{[[ m ]](pre[m]) | m ∈ pred_G(n)};
        if new ⊐ pre[n] then
            pre[n] := new;
            workset := workset ∪ succ_G(n)
        fi
    ni
od
```

FIGURE 2. A Workset Algorithm

Equation System 4.1

$$\mathbf{pre}(n) \quad = \quad \begin{cases} c_0 & \text{if } n = \mathbf{s} \\ \bigsqcup \{ [\![m]\!](\mathbf{pre}(m)) \mid m \in pred_G(n) \} & \text{otherwise} \end{cases}$$

In practice, the *MFP*-solution, which we denote by $\mathbf{pre}_{c_0}$, is computed by means of some iterative workset algorithm (see Figure 2).

We show that termination and correctness in this approach are a consequence of Corollary 3.2. To begin with, let $G = (N, E, \mathbf{s}, \mathbf{e})$ be the flow graph under consideration, and let $[\![\ ]\!] : N \to (\mathcal{C} \to \mathcal{C})$ be a local abstract semantics, such that all semantic functions are monotone. Without loss of generality we identify in the following the set of nodes of N with the set of natural numbers $\{1, \ldots, \mathbf{n}\}$, where $\mathbf{n}$ denotes the number of nodes of N.

Now let us define $D =_{df} \mathcal{C}^{\mathbf{n}}$ equipped with the pointwise extension of $\sqsubseteq$. One easily verifies that D is a wellfounded partial order. A value $d = (d^1, \ldots, d^{\mathbf{n}})$ represents an annotation of the flow graph where the value d^k is assigned to node k.

For every node k of the flow graph we define a function $f^k : D \to C$ by

$$f^k(d^1, \ldots, d^{\mathbf{n}}) =_{df} d'^k$$

where

$$d'^k = d^k \sqcup \bigsqcup \{ [\![m]\!](d^m) \mid m \in pred_G(k) \}$$

Intuitively, f^k describes the effect of a computation of the local semantics at node k. The following lemma states that the DFA problem is modelled correctly.

Lemma 4.2 *For all $d \in D$ we have: d is a solution of Equation System 4.1 if and only if d is a fixpoint of $f_{=_{df}}(f^1, \ldots, f^n)$.*

The workset algorithm of Figure 2 follows the general pattern of the nondeterministic skeleton algorithm of Figure 1 with $\mathcal{F} = \{f_{\{k\}} \mid 1 \leq k \leq n\}$. It profits from a set *workset* of indices which satisfies the invariant: $workset \supseteq \{k \mid f_{\{k\}}(d) \neq d\}$. One easily verifies that f is monotone. Hence, the premises of Corollary 3.2 are satisfied and we obtain the following theorem.

Theorem 4.3 (Correctness and Termination)
Every run of the workset algorithm terminates with the MFP-solution $\mathbf{pre}_{c_0}$.

4.2 Program Optimization: Partial Dead Code Elimination

In this section, we demonstrate an application of the Chaotic Fixpoint Iteration Theorem 2.3 in program optimization by proving the optimality of the *partial dead code elimination* algorithm of [12]. Intuitively, an assignment in a program is *dead* if its left hand side variable is dead immediately after its execution, i.e. if on every program continuation reaching the end of the program the first use of this variable is preceded by a redefinition of it. Correspondingly, an assignment is *partially dead*, if it is dead along some program paths reaching the end of the program.

Conceptually, the elimination of partially dead occurrences of an assignment pattern α (for short: partially dead α-occurrences) can be decomposed into two steps. First, moving them as far as possible in the direction of the control flow, and second, removing all dead α-occurrences. In order to preserve the program semantics, both the sinking and the elimination steps must be *admissible*. This is defined in full detail in [12]. Therefore, we restrict the presentation here to those parts that are essential for Theorem 2.3.

The relevance of Theorem 2.3 for partial dead code elimination stems from the fact that assignment sinking and elimination steps in general have *second order effects*, i.e. they usually enable assignment sinking and elimination steps for other assignment patterns. For example, eliminating the partially dead occurrences of some assignment pattern is often the premise that occurrences of other assignment patterns can be eliminated at all. In [12] this is taken care of by repeatedly applying admissible assignment sinking and elimination steps to the assignment patterns of the argument program until the program stabilizes, i.e. until a fixpoint is reached. The correctness of this iterative approach is a consequence of Theorem 2.3, as we show in the remainder of this section, where we consider an arbitrary, but fixed program G.

For a program G', we write $G' \vdash_{se} G''$ if the flow graph G'' results from

G' by applying an admissible assignment sinking or elimination transformation. We denote the set of all admissible assignment sinking and dead code elimination functions by $\mathcal{S}$ and $\mathcal{E}$, respectively. Additionally, we abbreviate $\mathcal{S} \cup \mathcal{E}$ by $\mathcal{T}$. It consists of all functions $f_{G_1,G_2} : \mathcal{G} \to \mathcal{G}$ defined by

$$\forall\, G' \in \mathcal{G}.\ f_{G_1,G_2}(G') =_{df} \begin{cases} G_2 & \text{if } G' = G_1 \\ G' & \text{otherwise} \end{cases}$$

where $G_1, G_2 \in \mathcal{G}$ and $G_1 \vdash_{se} G_2$. We also write $G' \vdash^{f}_{se} G''$ for $f(G') = G''$. Then,

$$\mathcal{G} =_{df} \{\, G' \mid G \vdash^{*}_{se} G' \,\}$$

denotes the *universe* of programs resulting from G by partial dead code elimination.

In order to compare the quality of different programs in $\mathcal{G}$, we introduce the relation "better" between programs of $\mathcal{G}$. Note that this relation is reflexive. In fact, *at least as good* would be the more precise but uglier notion.

Definition 4.4 (Optimality)

1. *Let $G', G'' \in \mathcal{G}$. Then G' is* better *than G'', in signs $G'' \sqsubseteq\!\!\!\!\sim\ G'$, if and only if for every assignment pattern α and every program path p leading from the start node to the end node of the argument program there are at most as many occurrences of α in G' as in G''.*[1]

2. *$G^* \in \mathcal{G}$ is* optimal *if and only if G^* is better than any other program in $\mathcal{G}$.*

It is easy to check that the relation $\sqsubseteq\!\!\!\!\sim$ is reflexive, transitive, and well-founded. Unfortunately, it is not antisymmetric. Hence, there may be several programs being optimal in the sense of Definition 4.4. In order to apply Theorem 2.3, we thus consider the partial order $\vdash^{*}_{se}$ instead of $\sqsubseteq\!\!\!\!\sim$, but we reconsider the relation $\sqsubseteq\!\!\!\!\sim$, subsequently (cf. Theorem 4.13).

In addition to $\mathcal{S}$ and $\mathcal{E}$, we define the set of maximal assignment sinkings and eliminations, which are the functions involved in the partial dead code elimination algorithm of [12]. A function $f_{G_1,G_2} \in \mathcal{S}\,(\mathcal{E})$ is called *maximal*, if for all functions $f_{G_1,G_3} \in \mathcal{S}\,(\mathcal{E})$ there is a function $f_{G_3,G_2} \in \mathcal{T}$ satisfying $f_{G_1,G_2} = f_{G_3,G_2} \circ f_{G_1,G_3}$. The set of all maximal sinking and elimination functions are denoted by $\mathcal{S}^{max}$ and $\mathcal{E}^{max}$, respectively, and $\mathcal{T}^{max} \subseteq \mathcal{T}$ denotes the union of $\mathcal{S}^{max}$ and $\mathcal{E}^{max}$. Finally, we denote the set of (maximal) admissible assignment sinkings and eliminations with respect to an assignment pattern α by $\mathcal{T}_\alpha$ and $\mathcal{T}^{max}_\alpha$. As a first result we obtain the

[1] Partial dead code elimination preserves the branching structure of the argument program. Hence, starting from a path in G, we can easily identify corresponding paths in G' and G''.

Dominance Lemma 4.5, which follows immediately from the definitions of $\mathcal{T}_\alpha^{max}$ and $\mathcal{T}_\alpha$.

Lemma 4.5 (Dominance)
Let $G_1 \in \mathcal{G}$, let $f \in \mathcal{T}_\alpha^{max}$ and $g \in \mathcal{T}_\alpha$ be corresponding functions, i.e. both sinking or both elimination functions, let $G_1 \vdash_{se}^{g} G_2$, and let $G_1 \vdash_{se}^{f} G_3$. Then we have: $G_2 \vdash_{se} G_3$. In particular: $G_3 \neq G_1$ if $G_2 \neq G_1$.

The next lemma can be proven by a straightforward induction on the length of a derivation sequence. The point for proving the induction step is that a program resulting from a transformation of $\mathcal{T}$ is at least as good as its argument with respect to $\sqsubseteq_{\approx}$. It is in the same equivalence class after sinking and trivial elimination steps, i.e. elimination steps where no assignment occurrence has been eliminated; and it is better otherwise. This follows immediately from the constraints that are satisfied by admissible assignment sinkings and eliminations.

Lemma 4.6 *We have: $G' \vdash_{se}^{*} G'' \Rightarrow G' \sqsubseteq_{\approx} G''$*

In other words, Lemma 4.6 says $\vdash_{se}^{*} \subseteq \sqsubseteq_{\approx}$. From the wellfoundedness of $\sqsubseteq_{\approx}$ and the definitions of $\vdash_{se}$ and $\mathcal{T}^{max}$ we immediately conclude:

Lemma 4.7 (Wellfoundedness and Increasingness)

1. *The relation $\vdash_{se}^{*}$ is wellfounded.*
2. *All functions $f \in \mathcal{T}^{max}$ are increasing.*

Next, we show that $\mathcal{T}$ is delay-monotone. This proof is supported by the following lemma, whose first part is a consequence of the fact that eliminating dead assignment occurrences does not reanimate other dead assignment occurrences, and whose second part is a consequence of the admissibility of g and a simple program transformation supposed in [12] which is typical for code motion transformations (cf. [6, 10, 11, 19]), namely to insert in every edge leading from a node with more than one successor to a node with more than one predecessor a new 'synthetic' node.

Lemma 4.8 *Let $G_1, G_2, G_3 \in \mathcal{G}$, and $g, h \in \mathcal{T}$ with $G_1 \vdash_{se}^{g} G_2$ and $G_1 \vdash_{se}^{h} G_3$.*

1. *If $g \in \mathcal{E}_\alpha$, and occ an α-occurrence occurring both in G_1 and G_2, then we have: If occ is dead in G_1, then it is dead in G_2.*
2. *If $g, h \in \mathcal{S}_\alpha$, occ an α-occurrence that has been moved by g into a node n of G_2 with more than one predecessor, and occ$'$ an α-occurrence that has been moved by h into a predecessor m of n, then we have: occ is dead in n iff occ$'$ is dead in m.*

Additionally, we have:

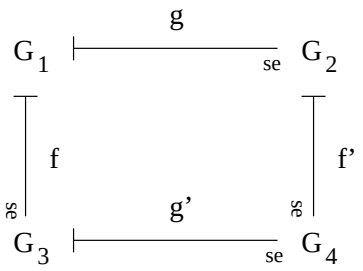

Fig. 3. Commuting Diagram

Lemma 4.9 *Let $G_1, G_2 \in \mathcal{G}$, let $g \in \mathcal{T}$, $f \in \mathcal{T}^{max}$, and let $\alpha, \beta \in \mathcal{AP}$ be two different assignment patterns. Then we have:*

1. *If $f, g \in \mathcal{E}$, then there are transformations $f', g' \in \mathcal{E}$ such that the diagram in Figure 3 commutes.*
2. *If $f, g \in \mathcal{S}$, then there are transformations $f', g' \in \mathcal{S}$ such that the diagram in Figure 3 commutes.*
3. *If $g \in \mathcal{E}_\alpha$ and $f \in \mathcal{S}_\beta$, then there are transformations $g' \in \mathcal{E}_\alpha$ and $f' \in \mathcal{S}_\beta$ such that the diagram in Figure 3 commutes.*
4. *If $g \in \mathcal{S}_\alpha$ and $f \in \mathcal{E}_\beta$, then there are transformations $g' \in \mathcal{S}_\alpha$ and $f' \in \mathcal{E}_\beta$ such that the diagram in Figure 3 commutes.*

Lemma 4.8 and Lemma 4.9 allow us to establish the following lemma, which is the key for proving the delay-monotonicity of $\mathcal{T}$.

Lemma 4.10 (Main Lemma)
$\forall\, g \in \mathcal{T}.\, G_1 \vdash^{g}_{se} G_2 \Rightarrow \forall\, f \in \mathcal{T}\ \exists\, f_1, \ldots, f_n \in \mathcal{T}.\, f(G_1) \vdash^{*}_{se} f_n \circ \ldots \circ f_1(G_2)$

The following theorem states the desired delay-monotonicity result. The reasoning closely resembles the classical Newman Lemma [17], saying that confluence follows from local confluence if the given relation is wellfounded. Note that monotonicity does not hold.

Lemma 4.11 (Delay-Monotonicity)
$\mathcal{T}$ is delay-monotone, i.e.

$$\forall\, f \in \mathcal{T}.\, G' \vdash^{*}_{se} G'' \Rightarrow \exists\, f_1, \ldots, f_n \in \mathcal{T}.\, f(G') \vdash^{*}_{se} f_n \circ \ldots \circ f_1(G'')\ .$$

Finally, we have to show that the set of common fixpoints of $\mathcal{T}^{max}$ and $\mathcal{T}$ coincide. Central for proving this result is the Dominance Lemma 4.5. Moreover, we have to check that the fixpoints of $\mathcal{T}$ are maximal in $\mathcal{G}$.

Theorem 4.12 (Fixpoint Characterization)

1. *$G' \in \mathcal{G}$ is a fixpoint of the functions in $\mathcal{T}$ if and only if G' is a fixpoint of the functions in $\mathcal{T}^{max}$.*

2. *$G' \in \mathcal{G}$ is a fixpoint of the functions in $\mathcal{T}$ if and only if G' is maximal in $\mathcal{G}$.*

Collecting our results we have: $\vdash^*_{se}$ is a wellfounded (Lemma 4.7(1)) complete partial order on $\mathcal{G}$, whose least element is G itself; all functions $f \in \mathcal{T}^{max}$ are increasing (Lemma 4.7(2)) and $\mathcal{T}$ is delay-monotone with respect to $\vdash^*_{se}$ (Lemma 4.11). Hence, Theorem 2.3 is applicable.

Moreover, the function families $\mathcal{T}^{max}$ and $\mathcal{T}$ have the same common fixpoints (Theorem 4.12(1)), and all of their fixpoints are maximal in $\vdash^*_{se}$ (Theorem 4.12(2)).

Combining these results and applying Lemma 4.6 we obtain that there exists a terminating optimal program transformation [12]:

Theorem 4.13 (Optimal Partial Dead Code Elimination)
$\mathcal{G}$ has (up to local reorderings in basic blocks) a unique optimal element (with respect to $\sqsubseteq\!\!\!\approx$) which can be computed by any fair sequence of function applications from $\mathcal{T}^{max}$.

We remark that the optimality of the partial faint code elimination algorithm which is also introduced in [12] as well as the optimality of the algorithm for the uniform elimination of partially redundant expressions and assignments in [13] can be proven in exactly the same fashion.

5 Conclusions

We have presented a new fixpoint theorem, which gives a sufficient condition for the existence and computability of the least common fixpoint of a family of functions on a wellfounded partial order. The point of this theorem is that for wellfounded partial orders the usual monotonicity condition can be substantially weakened. This allows us to capture a new and interesting class of practically relevant applications. To characterize this class, we discussed applications in *data flow analysis* and *program optimization*. Whereas the first application could still be treated by the known fixpoint theorems, the second application requires the generalization developed in this paper. Our new theorem is the key for proving the optimality of the partial dead code elimimation algorithm of [12], which is implemented in the new release of the Sun SPARCompiler language systems. Moreover, as our theorem only requires delay-monotonicity, a property being weaker than monoticity, algorithm designers gain greater flexibility in the construction process than in the classical setup.

6 References

[1] G. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the Association for Computing Machinery*, 25(2):226–244, April 1978.

[2] B. Le Charlier, editor. *SAS '94*, volume 864 of *Lecture Notes in Computer Science*, Namur, Belgium, 1994. Springer-Verlag.

[3] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical Report 88, Laboratoire d'Informatique, U.S.M.G., Grenoble, France, September 1977.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977.

[5] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–87, 1979.

[6] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation '92*, volume *27*,7 of *ACM SIGPLAN Notices*, pages 212–223, San Francisco, CA, June 1992.

[7] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[8] N. Jørgensen. Finding fixpoints in finite function spaces using needeness analysis and chaotic iteration. In Charlier [2], pages 329–345.

[9] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[10] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation '92*, volume *27*,7 of *ACM SIGPLAN Notices*, pages 224–234, San Francisco, CA, June 1992.

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

[12] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation '94*, volume *29*,6 of *ACM SIGPLAN Notices*, pages 147–158, Orlando, FL, June 1994.

[13] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation '95*, volume *30*,6 of *ACM SIGPLAN Notices*, pages 233–245, La Jolla, CA, June 1995.

[14] J.-L. Lassez, V.L. Nguyen, and E.A. Sonnenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[15] K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103–129, 1993.

[16] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[17] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43,2:223–243, 1942.

[18] F. Robert. Convergence locale d'itérations chaotiques non linéaires. Technical Report 58, Laboratoire d'Informatique, U.S.M.G., Grenoble, France, December 1976.

[19] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 12–27, San Diego, January 1988. IEEE Computer Society Press.

[20] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 25(2):285–309, 1955.

[21] A. Üresin and M. Dubois. Sufficient conditions for the convergence of asynchronous iterations. *Parallel Computing*, 10:83–92, 1989.

[22] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In Charlier [2], pages 314–328.

[23] J. Wei. Parallel asynchronous iterations of least fixed points. *Parallel Computing*, 19:887–895, 1993.

Appendix

Proof of the New Fixpoint Theorem 2.3

Proof: The *wellfoundedness* of $\sqsubseteq$, the *increasingness* of the functions of $\mathcal{F}$, and the *fairness* of the strategy directly imply that $\bigsqcup f_\gamma(\bot)$ is a fixpoint and that it is reached in a finite number of iteration steps. Thus, we are left with showing that $\bigsqcup f_\gamma(\bot)$ is a lower bound for every common fixpoint of $\mathcal{F}$. Let δ be an arbitrary fixed point of $\mathcal{F}$, i.e. $f_k(\delta) = \delta$ for all $f_k \in \mathcal{F}$ and, as a consequence, $f_s(\delta) = \delta$ for all $s \in \mathbb{N}^*$. For $(d_i)_{i\in\mathbb{N}} = f_\gamma(\bot)$ we show $d_i \sqsubseteq \delta$ for all $i \in \mathbb{N}$ by induction on i. For $i = 0$, we have $d_0 = \bot \sqsubseteq \delta$. The induction step, $d_{i+1} = f_{\gamma(i)}(d_i) \sqsubseteq f_s(\delta) = \delta$ for some $s \in \mathbb{N}^*$, then follows from the induction hypothesis $d_i \sqsubseteq \delta$ and delay-monotonicity. Hence $d_i \sqsubseteq \delta$ for all i and so $\bigsqcup f_\gamma(\bot) \sqsubseteq \delta$ by definition of $\bigsqcup$. □

Proof of Lemma 3.1

Proof: Let γ be an arbitrary set strategy on $\{1, \ldots, n\}$, and $(d_i)_{i\in\mathbb{N}}$ be its induced chaotic iteration starting from $\bot =_{df} (\bot_C, \ldots, \bot_C)$. We have to show

$$\forall i \in \mathbb{N}.\, d_i \sqsubseteq d_{i+1} \ .$$

The proof is by induction on i. Let $J =_{df} \gamma(i)$. By definition of f_J the property

$$\forall\, k \in \{1, \ldots, n\} \setminus J \ \forall\, d \in D.\ f_J(d)^k = d^k$$

holds. Therefore, it suffices to show $d_i^k \sqsubseteq_C f_k(d_i)^k$ for all $k \in J$. The case $d_i^k = \bot_C$ is trivial. Otherwise, d_i^k must have been updated in an earlier step. More precisely, $d_i^k = f_{\gamma(j)}(d_j)^k$ where j is the greatest index $j < i$ such that $k \in \gamma(j)$. By induction hypothesis for each $j' = j, \ldots, i-1$, we obtain $d_{j'} \sqsubseteq d_{j'+1}$, from which $d_j \sqsubseteq d_i$ follows by transitivity of $\sqsubseteq$. By monotonicity of $f_{\gamma(j)}$, we have $d_i^k = f_{\gamma(j)}(d_j)^k \sqsubseteq_C f_{\gamma(j)}(d_i)^k = f_k(d_i) = f_J(d_i)^k = d_{i+1}^k$, and the proof is done. □

Proof of Lemma 4.9

For $f \in \mathcal{E}_\alpha$, let $elim_\alpha(f, G')$ denote the set of α-occurrences in G' that are eliminated by f.

Proof: The first part of Lemma 4.9 is proven by investigating two cases: (1a) $g, f \in \mathcal{E}_\alpha$, (1b) $g \in \mathcal{E}_\alpha$, $f \in \mathcal{E}_\beta$

In case (1a) the maximality of f guarantees $elim_\alpha(g, G_1) \subseteq elim_\alpha(f, G_1)$. We obtain that all α-occurrences in $elim_\alpha(f, G_1) \setminus elim_\alpha(g, G_1)$ are dead in G_2 by Lemma 4.8(1). Hence, there is a transformation in $\mathcal{E}$, which eliminates all α-occurrences in $elim_\alpha(f, G_1) \setminus elim_\alpha(g, G_1)$ in G_2. Choosing this transformation as f', and an arbitrary function of $\mathcal{T}^{max}$ leaving G_3 invariant as g', we get

$$G_2 \vdash_{se}^{f'} G_4$$

and, therefore, as desired

$$G_3 \vdash_{se}^{g'} G_3 = G_4 \quad .$$

In case (1b) Lemma 4.8(1) yields that $elim_\alpha(g, G_1)$ and $elim_\beta(f, G_1)$ are subsets of the sets of dead α- and β-occurrences in G_3 and G_2, respectively. Hence, there are transformations in $\mathcal{E}$ which eliminate all α-occurrences of $elim_\alpha(g, G_1)$ in G_3 and all β-occurrences of $elim_\beta(f, G_1)$ in G_2. Choosing these transformations as g' and f', respectively, we obtain as desired

$$G_2 \vdash_{se}^{f'} G_4 \quad \text{and} \quad G_3 \vdash_{se}^{g'} G_4$$

Similarly to the proof of the first part of Lemma 4.9, two cases have also to be considered in the proof of its second part: (2a) $g, f \in \mathcal{S}_\alpha$, (2b) $g \in \mathcal{S}_\alpha$, $f \in \mathcal{S}_\beta$

In case (2a) the Dominance Lemma 4.5 yields the existence of an admissible assignment sinking $f' \in \mathcal{S}$, which directly transforms G_2 into G_3. Thus, by choosing an arbitrary function of $\mathcal{T}^{max}$ leaving G_3 invariant as g', we succeed in this case.

In order to prove case (2b) consider the program G_2', which results from G_2 by reinserting all α-occurrences that have been moved by g. Let G_4' be the program which results from the maximal β-sinking to G_2', i.e. $G_2' \vdash_{se}^{f} G_4'$, and let G_4 result from G_4' by eliminating the reinserted α-occurrences. Obviously, there is a transformation $f' \in \mathcal{S}_\beta$, which directly transforms G_2 into G_4. The admissibility of g implies that G_2 and G_4 are identical except for β-occurrences. Thus, interchanging the roles of α and β and applying the same construction to G_3, we get the existence of a transformation $g' \in \mathcal{S}_\alpha$, which transforms G_3 into G_4. This completes the proof of case (2b).

The remaining two parts of Lemma 4.9 can now be proven along the proof lines of part (2) by additionally applying Lemma 4.8(1). □

Proof of the Main Lemma 4.10

Proof: Let $\alpha, \beta \in \mathcal{AP}$ be different assignment patterns. Then the Main Lemma 4.10 is proven by investigating the following cases:

1. $g, f \in \mathcal{E}$
2. $g, f \in \mathcal{S}$
3. $g \in \mathcal{E}_\alpha\,,\ f \in \mathcal{S}_\beta$
4. $g \in \mathcal{S}_\alpha\,,\ f \in \mathcal{E}_\beta$
5. $g \in \mathcal{E}_\alpha\,,\ f \in \mathcal{S}_\alpha$
6. $g \in \mathcal{S}_\alpha\,,\ f \in \mathcal{E}_\alpha$

Applying the Dominance Lemma 4.5 we can assume without loss of generality that f is maximal, i.e. $f \in \mathcal{T}^{max}$. The first four cases are then immediate consequences of the corresponding parts of Lemma 4.9. Thus, we are left with the cases (5) and (6), which both can be proven in the same fashion. Thus, we only present the proof of case (5).

In the situation of case (5) let $G_2^{'}$ be the program, which results from G_2 by reinserting a labelled version of all α-occurrences that have been eliminated by g. Due to the labelling the reinserted α-occurrences can be distinguished from the remaining ones. In $G_2^{'}$, we assume that only unlabelled α-occurrences can be subject to assignment sinkings; however, all α-occurrences, i.e. labelled or not, are considered to block the sinking of α-occurrences, i.e. no α-occurrence can sink across a labelled or unlabelled α-occurrence in $G_2^{'}$. Now we choose the uniquely determined maximal α-sinking and α-elimination as f_1 and f_2, respectively, and denote the program resulting from the subsequent application of f_1 and f_2 to $G_2^{'}$ by $G_4^{'}$. By eliminating all labelled α-occurrences in $G_4^{'}$ we obtain the program G_4. Of course, f_1 and f_2 have corresponding functions in $\mathcal{T}$ which directly transform G_2 into G_4. Thus, in order to complete the proof of case (5), it is sufficient to show that a maximal α-elimination transforms G_3 into G_4 as well. The point here is that due to the reinsertion of α-occurrences eliminated by g, $G_2^{'}$ has precisely the same 'α-blockades' as G_1. Hence, on join-free paths, i.e. on paths where no node has more than one predecessor, (unlabelled) α-occurrences in $G_2^{'}$ have precisely the same sinking potential as their corresponding occurrences in G_1. Only on paths containing join-nodes the sinking potential can be different: In $G_2^{'}$ an α-occurrence occ can be blocked in a predecessor n of a join-node j, because there is a brother m of n,[2] to which no α-occurrence is sinkable; in G_1, however, the same α-occurrence can successfully be sunk into j, because some of the α-occurrences eliminated by g in G_1 are sinkable to m. It is worth noting that the α-occurrence is dead in j and never becomes live again. Hence, it is eliminated by the subsequent application of f_2. This, however, holds for the α-occurrence blocked in the predecessor n of j as well, since it is dead according to Lemma 4.8(2). Combining these results we obtain as desired that maximal α-elimination transforms G_3 into G_4. □

[2]The set of brothers of a node n is given by $\bigcup \{pred(m) \mid m \in succ(n)\}$.

Proof of Theorem 4.12

Proof: Since (2) holds trivially, we only prove (1). The first implication, "$\Rightarrow$", is a simple consequence of $\mathcal{T}^{max} \subseteq \mathcal{T}$. The second implication, "$\Leftarrow$", is proven by showing the contrapositive. Without loss of generality, we can assume $g \in \mathcal{T}_\alpha \backslash \mathcal{T}_\alpha^{max}$ and $G' \vdash_{se}^{g} G''$ with $G'' \neq G'$. Now, let $f \in \mathcal{T}_\alpha^{max}$ be the uniquely determined function f of $\mathcal{T}_\alpha^{max}$ corresponding to g. Then the Dominance Lemma 4.5 yields as desired that the program resulting from the application of f to G' is different from G'. $\square$