A Universal Top-Down Fixpoint Algorithm

Baudouin Le Charlier[1]
Pascal Van Hentenryck[2]

Technical Report No. CS-92-25

May 1992

[1]University of Namur 21 rue Grandgagnage, B-5000 Namur Belgium
[2]Brown University Computer Science Box 1910, Providence, RI 02912

A Universal Top-Down Fixpoint Algorithm

Baudouin Le Charlier
University of Namur,
21 rue Grandgagnage, B-5000 Namur (Belgium)
Email: ble@info.fundp.ac.be

Pascal Van Hentenryck
Brown University,
Box 1910, Providence, RI 02912 (USA)
Email: pvh@cs.brown.edu

Abstract

Computing fixpoints and postfixpoints of transformations has numerous applications in computer science. In this paper, we present a universal top-down fixpoint algorithm based on the weak assumption that the transformation is defined by an effective procedure. Given the procedure and a value α, the universal algorithm computes the value of α in the least fixpoint of the transformation and has the property of focusing on the subset of elements necessary to compute the value of α. The total correctness of the algorithm is proven and requires some weak (but rather technical) assumptions. These conditions can be relaxed further if postfixpoints suffice for the application. As a consequence, the algorithm generalizes and abstracts numerous algorithms and provides a versatile alternative to bottom-up algorithms. It has been used to derive specific abstract interpretation algorithms for Prolog, which are practical both in terms of efficiency and accuracy and are, to our knowledge, the fastest algorithms available.

1 Introduction

The computation of least fixpoints and/or postfixpoints of transformations (or functionals) has numerous applications in computer science, including programming languages (e.g. abstract interpretation), data bases (e.g. dependencies), and artificial intelligence (e.g. expert systems). Many fixpoint algorithms were proposed in the past; most of them, however, are either specific to a given problem, restricted to a syntactic class of transformations (e.g. equations), or they impose some strong restrictions on the domains of the transformations (e.g. finiteness). Moreover, many algorithms are bottom-up, which complicates them substantially for infinite domains and/or when the application only requires a subset of the fixpoint or postfixpoint.

In this paper, we propose a universal top-down fixpoint algorithm, based upon the weak assumption that the transformation, say τ, is defined by an effective procedure `tau` (e.g. a procedure in a programming language), which, given a functional parameter f and a value α, returns the value $\tau f \alpha$. The algorithm, given `tau` and α, returns the value of α in the least fixpoint of τ. It has the property of focusing on the subset of elements necessary to compute the value of α and explores few, if any, other elements in the practical applications we have considered [15]. Its partial correctness and termination are proven and require some additional weak (but rather technical) assumptions. These assumptions are mostly specific monotonicity conditions and they can be relaxed further when the application only requires a postfixpoint. In particular, widening techniques [7] allow termination even on infinite domains and the monotonicity conditions imposed for partial correctness can be dropped at the cost of a stronger assumption for termination. As a consequence, the algorithm generalizes and abstracts numerous existing algorithms, removes many restrictions, and provides a versatile alternative to bottom-up algorithms. The algorithm has proven useful to derive several abstract interpretation algorithms for Prolog, based on various semantics [14, 15, 16, 9]. These algorithms (to our knowledge, the fastest available) are practical both in terms of efficiency and accuracy.

The rest of the paper is organized as follows. Section 2 presents some mathematical preliminaries. Section 3 presents the universal top-down fixpoint algorithm. Section 4 outlines the proofs of partial correctness and termination and formulates the necessary assumptions. Section 5 indicates how to weaken further the conditions when only a postfixpoint is required. Section 6 discusses how several optimizations can be included in the algorithm. Section 7 discusses related works on fixpoint algorithms, abstract interpretation, and deductive data bases, and suggests several possible improvements for specific application areas.

2 Preliminaries

The basic computation domains considered in this paper are *complete partial orders* (cpos). Recall that a cpo is *noetherian* if it contains no infinite strictly increasing chains. Cpos are denoted by A, A' in the following. We use $A \mapsto A'$ to represent the set of *monotone functions* from A to A', where A, A' are cpos and A' is endowed with a `lub` operation. Similarly, $(A \mapsto A') \mapsto (A \mapsto A')$ is the set of monotonic functionals (or transformations). Transformations are denoted by the letter τ in the following. The *least fixpoint* of a transformation is denoted by $\mu(\tau)$ or $\mu\tau$ when there is no ambiguity. A *postfixpoint* of a transformation τ is a function f such that $f \geq \tau(f)$.

We also consider partial functions and use $A \not\to A'$ to represent the set of partial functions from A to A'. We write $f(x) = \textbf{undef}$ if the partial function f is undefined for some value $x \in A$. The domain of a partial function f, dom(f), is the set of $\alpha \in A$ such that $f(\alpha) \neq \textbf{undef}$. $A \not\to A'$ is endowed with a structure of cpo by the traditional inclusion relation on partial functions:

$$f \subseteq g \text{ iff } \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \;\&\; f(\alpha) = g(\alpha) \text{ for all } \alpha \in \mathrm{dom}(f).$$

Let $S \subseteq A$. The *restriction* of a partial function f to S, say $f_{|S}$, is the partial function satisfying $\mathrm{dom}(f_{|S}) = \mathrm{dom}(f) \cap S$ and $f_{|S}(\alpha) = f(\alpha)\ \forall \alpha \in \mathrm{dom}(f_{|S})$. Let $\tau^+ : (A \not\to A') \to (A \not\to A')$, $f \in A \not\to A'$, and $\alpha \in A$. We say that α is based in f (wrt τ^+) iff $\tau^+ f\alpha \neq \textbf{undef}$. $f \in A \not\to A'$ is a *partial fixpoint (resp. postfixpoint)* iff $\forall \alpha \in \mathrm{dom}(f)$: α is based in f & $f\alpha = \tau^+ f\alpha$ (resp. $f\alpha \geq \tau^+ f\alpha$).

The set of monotone partial functions is denoted $A \mapsto\!\!\!\!\not\;\; A'$. We also define a partial order $\leq$ on monotone partial functions:

$$f \leq g \text{ iff } \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \;\&\; f(\alpha) \leq g(\alpha) \text{ for all } \alpha \in \mathrm{dom}(f).$$

3 The Problem

We consider a monotone transformation $\tau : (A \mapsto A') \mapsto (A \mapsto A')$ described by a computable procedure `tau(f:A→A';α:A):A'` written in any usual programming language (e.g. Pascal). Procedure `tau` computes the transformation τ (i.e. `tau(f,`α`)`) returns $\tau f \alpha$, calls f finitely often, and terminates when the functional parameter computes a (total) monotone function. The second requirement can be formalized, using denotational semantics [25], by requiring that procedure `tau` computes a continuous transformation $\tau^+ : (A \not\rightarrow A') \rightarrow (A \not\rightarrow A')$, i.e. $\forall f \in (A \mapsto A') : \tau^+ f = \tau f$. In the following, we overload τ and τ^+ and use τ only.

The purpose of this paper is to present an algorithm to compute $\mu(\tau)\alpha$, given procedure `tau` and α. The algorithm computes a subset of the least fixpoint for practical (A may be infinite) and efficiency reasons. As can be noticed, the algorithm is very general since it only assumes the existence of procedure `tau`. It is fact necessary to add some weak (but rather technical) assumptions to prove partial correctness and termination. They are discussed later in the paper.

4 The Algorithm

4.1 Informal Presentation

The goal of our algorithm is to avoid computing the entire table of $\mu(\tau)$ for practical and efficiency reasons, since, on the one hand, A may be infinite, and, on the other hand, only a small subset of $\mu(\tau)$ is sufficient in general to compute $\mu(\tau)\alpha$. As a consequence, the result of our algorithm is the table of a partial function f satisfying $f(\alpha) = \mu(\tau)\alpha$. Of course, the elements necessary to compute $\mu(\tau)\alpha$ are not known in advance and it is the purpose of the top-down strategy to avoid considering non-relevant elements. How the algorithm computes this table is the key issue. Recall that the only tool at our disposal is the procedure `tau(f:A`$\rightarrow$`A';`α`:A):A'`.

Our algorithm builds the partial table by simulating the execution of procedure `tau` on α. The simulation is straightforward as long as function f is not used. When function f is applied to a parameter β, the algorithm suspends its simulation (we say that α is suspended) and initiates a sub-simulation on β. On return of the sub-simulation, the algorithm updates the partial table and resumes the initial simulation, using, in both cases, the sub-simulation result. The sub-simulation itself proceeds as the initial simulation except when function f is applied to α. In this case, no sub-simulation is initiated and the current value of α in the partial table is used instead. Since the value of α may be approximated, the algorithm needs to redo the sub-simulation on β if the value of α is refined.

The table built by the algorithm corresponds in fact to a sequence of partial functions $f_0, \ldots, f_n$ satisfying $f_0 \leq f_1 \leq \ldots \leq \mu(\tau)$. f_0 is the function undefined everywhere and the algorithm moves from a function f_i to the next by considering a new element β (i.e. $\beta \notin \mathrm{dom}(f_i)$ & $\beta \in \mathrm{dom}(f_{i+1})$) or refining an existing element β (i.e. $f_i(\beta) < f_{i+1}(\beta')$).

4.2 Operations on Partial Tables

The algorithm can be viewed as building a partial function represented by its table. Tables are denoted by `pt` in the following. Also, because of the correspondence between tables and partial functions, we do not distinguish the two when convenient. In particular, when we write about the domain of a table, or the application of a table to a value, we mean the appropriate operations of the partial function.

The algorithm manipulates partial tables through two operations which are used respectively to introduce a new element in `pt` and to update `pt` with the result β for α. Their specification are as follows (in the postconditions, a parameter subscripted by 0 denotes the value of the parameter at call time):

`Extend_pt(pt,`α`)`

Pre: $\alpha \notin \mathrm{dom}(\mathtt{pt})$.

Post: $\mathtt{pt} = \mathtt{pt}_0 \cup \{(\alpha, \beta)\}$ where $\beta = lub\{\mathrm{pt}_0(\alpha') : \alpha' \leq \alpha \text{ and } \alpha' \in \mathrm{dom}(\mathrm{pt}_0)\}$

`Adjust_pt(pt,`α, β`)`

Pre: $\alpha \in \mathrm{dom}(\mathtt{pt}_0)$

Post: $\mathtt{pt} = \mathrm{pt}_1 \cup \mathrm{pt}_2$ where $\mathrm{pt}_1 = \mathrm{pt}_0 \setminus \{(\alpha', \beta') \in \mathrm{pt}_0 \mid \alpha' \geq \alpha\}$ and
$\mathtt{pt}_2 = \{(\alpha', lub\{\mathrm{pt}_0(\alpha'), \beta\} \mid \alpha' \geq \alpha \;\&\; \alpha' \in \mathrm{dom}(\mathrm{pt}_0)\}$.

In the algorithm, we use in fact a slightly more general version that returns also the set of $\alpha' \in \mathrm{dom}(\mathtt{pt})$ such that $\mathtt{pt}(\alpha')$ has been modified by the operation.

4.3 The Dependency Graph

Our algorithm includes a dependency graph to avoid redundant computations. The main idea is to avoid restarting simulations of procedure `tau` that would not produce any information. Dependencies enable to improve the worst case complexity of the algorithm for many instantiations (see [14]).

Definition 1 A dependency graph dg is a set of elements $(\alpha, \{\beta_1, \ldots, \beta_n\})$ such that

$$(\alpha, S) \in dg \;\&\; (\alpha, S') \in dg \;\Rightarrow\; S = S'.$$

The domain of dg, $\mathrm{dom}(dg)$, is the set of α such that $(\alpha, S) \in dg$ for some S. The codomain of dg, $\mathrm{codom}(dg)$, is the set of all β such that there exists $(\alpha, S) \in dg$ and $\beta \in S$. If $\alpha \in \mathrm{dom}(dg)$, $dg(\alpha)$ denotes the set S such that $(\alpha, S) \in dg$.

Intuitively, $dg(\alpha)$ represents all elements which α depends upon. The value of α needs to be reconsidered if any of these elements are refined. More precisely, two situations must be distinguished:

- If α is not suspended, $\alpha \in \mathrm{dom}(dg)$ means that $\mathtt{pt}(\alpha)$ cannot be refined by a new simulation.
- If α is suspended, $\alpha \in \mathrm{dom}(dg)$ means that, so far, the computation of α need no further simulation. In addition, $dg(\alpha)$ contains the set of β called directly by the simulation of α.

Definition 2 Let dg be a dependency graph such that $\alpha \in \mathrm{dom}(dg)$. The set $dg^+(\alpha)$ is the smallest subset of $\mathrm{codom}(dg)$ closed by the following two rules:

1. $\beta \in dg(\alpha) \;\Rightarrow\; \beta \in dg^+(\alpha)$.
2. $\beta \in dg(\alpha) \;\&\; \beta \in \mathrm{dom}(dg) \;\&\; \gamma \in dg^+(\beta) \;\Rightarrow\; \gamma \in dg^+(\alpha)$.

The reflexive and transitive closure of dg, noted $dg^*(\alpha)$, is simply $dg^+(\alpha) \cup \{\alpha\}$.

The operations on dependency graphs (to remove, extend, and add dependencies) can be specified as follows:

$\texttt{Remove\_dg(dg,}\{\beta_1, \ldots, \beta_n\}\texttt{)}$

Post: $\texttt{dg} = \texttt{dg}_0 \setminus \{(\alpha, S) \in \texttt{dg}_0 \mid \{\beta_1, \ldots, \beta_n\} \cap \texttt{dg}_0^+(\alpha) \neq \emptyset\}$.

$\texttt{Extend\_dg(dg,}\alpha\texttt{)}$

Pre: $\alpha \notin \mathrm{dom}(\texttt{dg})$;

Post: $\texttt{dg} = \texttt{dg}_0 \cup \{(\alpha, \emptyset)\}$.

$\texttt{Add\_dg(dg,}\alpha\texttt{,}\beta\texttt{)}$

Pre: $\alpha \in \mathrm{dom}(\texttt{dg})$;

Post: $\texttt{dg} = \texttt{dg}_0 \setminus \{(\alpha, \texttt{dg}_0(\alpha))\} \cup \{(\alpha, \texttt{dg}_0(\alpha) \cup \{\beta\})\}$.

Note also that if $\texttt{dg}^+(\alpha)$ does not contain any suspended element, then $\texttt{pt}(\alpha) = \mu(\tau)\alpha$.

4.4 The Algorithm

The algorithm is depicted in Figure 1. The top-level procedure is `compute_fixpoint` which receives two inputs, the functional parameter `tau` and an element α of A, and returns two outputs, the partial table `pt` and the dependency graph `dg`. The procedure expected as functional parameter is a syntactic variant of the procedure `tau` given by the user. The modification, which can be automated directly, amounts to replacing each call $\texttt{f(}\beta\texttt{)}$ in $\texttt{tau(f,}\alpha\texttt{)}$ by $\texttt{f(}\beta,\alpha\texttt{)}$. This simple change allows us to simulate procedure `tau` on an unknown function. The partial table returned as result satisfies $\texttt{pt}(\alpha) = \mu(\tau)\alpha$ for all $\alpha \in \mathrm{dom}(\texttt{dg})$, where τ is the transformation described by `tau`. In addition to the parameters, `compute_fixpoint` makes use of a local set `ics` representing the set of initiated simulations (or computations), a local procedure, and a local function. Its body initializes `pt, dg, ics` to empty sets and calls procedure `repeat_computation`.

Procedure `repeat_computation` starts by testing if its parameter α is in the domain of the dependency graph or in the set of initiated computations. In the first case, no information can be gained by executing the remaining instructions, since no element which α depends upon has been refined. In the second case, a sub-computation has already been started for α. In all other cases, the procedure extends the partial table if necessary and updates `ics` to include α. It then enters a repeat loop which, informally speaking, computes a local fixpoint for α given the values of the suspended elements in `ics`. This local fixpoint is obtained by successive executions of procedure `tau` and terminates when α is in the domain of the dependency graph (which means that an additional iteration would produce no new information). Each iteration computes a value β (possibly) used to update the partial table. The dependency graph is also extended with α before the call and is possibly updated after the call to remove elements depending upon elements in `modified`. Note that procedure `tau` has the local function `pretended_f` as argument.

When procedure `tau` applies its functional argument, function `pretended_f` is used. It receives two parameters: β is the argument to which the function should be applied while α is the value which required the value of β. To obtain the value for β, the function calls recursively procedure `repeat_computation` on β. It then updates the dependency graph if α is still in its domain; otherwise α needs an additional iteration and no update takes place. Finally, the procedure returns the value $\texttt{pt}(\beta)$.

```
procedure compute_fixpoint(in tau(f:A→A';α:A):A',α:A; out pt,dg);
   var ics: set of A;
   procedure repeat_computation(in α:A);
      var β:A';
      begin
         if α ∉ dom(dg) ∪ ics then
                begin
                if α ∉ dom(pt) then Extend_pt(pt,α);
                ics := ics ∪ {α};
                repeat
                   Extend_dg(dg,α);
                   β := tau(pretended_f,α);
                   Adjust_pt(pt,α,β,modified);
                   Remove_dg(dg,modified)
                until α ∈ dom(dg);
                ics := ics − {α}
             end
      end;
   function pretended_f(in β,α,:A):A';
      begin
         repeat_computation(β);
         if α ∈ dom(dg) then Add_dg(dg,α,β);
         pretended_f := pt(β)
      end;
   begin
         pt := {}; dg := {}; ics := {}; repeat_computation(α)
   end
```

Figure 1: The Universal Top-down Fixpoint Algorithm

5 Correctness

5.1 Partial Correctness

Partial correctness mainly amounts to proving the condition $\mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})} = (\mu\tau)_{|\mathrm{dom}(\mathtt{dg})}$ whenever the algorithm terminates. We outline the main ideas here; most of the details are in Appendix 1. The proof is split into two steps:

Condition 1: $\forall x \in \mathrm{dom}(\mathtt{dg}) : \mathtt{pt}x \geq \mu\tau x$.

Condition 2: $\forall x \in \mathrm{dom}(\mathtt{dg}) : \mathtt{pt}x \leq \mu\tau x$.

Condition 1 is proven by showing the property

$$\forall x \in \mathrm{dom}(\mathtt{dg}) : \begin{cases} \mathtt{pt}x \geq \tau\mathtt{pt}x, \\ x \text{ is based in } \mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})}. \end{cases}$$

The condition means that $\mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})}$ is a partial postfixpoint for τ. The property follows from operations `Adjust_pt` and `Remove_dg` which ensure $\mathtt{pt}\alpha \geq \tau\mathtt{pt}\alpha$ on termination of the repeat loop of procedure `repeat_computation`. Moreover, conditions α is based in $\mathtt{pt}_{|\mathtt{dg}(\alpha)}$ and $\mathtt{dg}(\alpha) \subseteq \mathrm{codom}(\mathtt{dg}) \subseteq \mathrm{dom}(\mathtt{dg})$ are also fulfilled.

The proof of condition 2 requires some additional hypotheses about procedure `tau`. Informally speaking, *the information produced by a call $f(\alpha_i)$ should be "recorded monotonically" inside the internal store of* `tau`. To express this condition mathematically, we assume, without loss of generality, that the execution of `tau` depends on some internal state σ. In other words, the execution from any computation point is completely determined by the current internal state σ and the actual functional parameter f. This implies the existence of a continuous function $\theta : (A \not\rightarrow A') \rightarrow (\Sigma \not\rightarrow A')$, where Σ is the set of internal states, satisfying $\theta f\sigma = \tau f\alpha$ for any possible intermediate internal state σ occurring during execution of $\mathtt{tau}(f, \alpha)$. Clearly, θ is a *continuation* in the sense of denotational semantics.

Σ is endowed with an order relation (denoted $\leq$) which is intended to reflecting, in some sense, the orderings on A and A'. Intuitively, if $\alpha_1 \leq \alpha_2$ and σ_1 and σ_2 are the internal states at corresponding execution points for two calls $\mathtt{tau}(f, \alpha_1)$ and $\mathtt{tau}(f, \alpha_2)$, then $\sigma_1 \leq \sigma_2$. In other words, the internal state contains a store of the information received in the past and the greater the value of α, the greater the value of the store. This can be formalized as follows:

Hypothesis 1: θf is a monotone function of σ (when execution terminates) or, in symbols:

$$\forall f \in (A \mapsto A') : \forall \sigma_1, \sigma_2 \in \Sigma : \sigma_1 \leq \sigma_2 \Rightarrow \theta f\sigma_1 \leq \theta f\sigma_2.$$

For partial correctness, we assume, for convenience, that $\theta f\sigma_1 \leq \theta f\sigma_2$ holds whenever $\theta f\sigma_1$ or $\theta f\sigma_2$ is not defined. Finally, we need a "recording" function $r : \Sigma \rightarrow A' \mapsto \Sigma$ such that $r\sigma\alpha'$ defines the new internal state resulting from the internal state σ before the call and the value α' returned by the call. Mathematically,

$$\theta f\sigma = \theta f(r\sigma(f\alpha))$$

holds for any partial function f such that $f(\alpha)$ is defined and any internal state τ before a call `f(`α`)`. The recording function should satisfy the following property:

Hypothesis 2: The value returned by f is recorded monotonically in the store or, in symbols,

$$\forall \alpha'_1, \alpha'_2 \in A' : \forall \sigma \in \Sigma : \alpha'_1 \leq \alpha'_2 \Rightarrow r\sigma\alpha'_1 \leq r\sigma\alpha'_2.$$

Note also that the monotonicity of θ wrt f can be proven from the hypotheses on θ and r.

Now the proof that condition 2 holds on termination follows from the invariant $\texttt{pt} \leq \mu\tau$ which always holds during the computation. The invariant imposes that the result β of any call to procedure `tau(pretended_f,`α`)` satisfies $\beta \leq \mu\tau\alpha$, which, in turn, follows from the property that $\theta(\mu\tau)\sigma \leq \mu\tau\alpha$ holds for any state σ occurring during the call. The proof of the last property uses the hypotheses on θ and r. More precisely,

- $\theta(\mu\tau)\sigma = \mu\tau\alpha$ holds before the first call to `pretended_f` by definition of θ and $\tau(\mu\tau) = \mu\tau$.
- Let σ, σ' denote respectively internal states after and before a call `pretended_f(`α, α_i`)`. $\theta(\mu\tau)\sigma' \leq \mu\tau\alpha$ holds by hypothesis. Let β_i be the value returned by `pretended_f(`α,α_i`)`. By definition of r, $\sigma = r\sigma'\beta_i$. But $\beta_i \leq \mu\tau\alpha_i$ because $\beta_i = \texttt{pt}\alpha_i$ and $\texttt{pt} \leq \mu\tau$ by hypothesis. Therefore

$$\begin{array}{rcll} \theta(\mu\tau)\sigma & = & \theta(\mu\tau)(r\sigma'\beta_i) & \text{(definition of } r) \\ & \leq & \theta(\mu\tau)(r\sigma'(\mu\tau\alpha_i)) & \text{(monotonicity of } r, \mu\tau \text{ and } \theta(\mu\tau)) \\ & = & \theta(\mu\tau)\sigma' & \text{(property of } r \text{ and } \theta) \\ & \leq & \mu\tau\alpha & \text{(hypothesis)} \end{array}$$

5.2 Termination

We give sufficient conditions for the termination of the algorithm. Other (weaker) conditions are possible for particular transformation classes. Moreover, these conditions are relaxed further for upper approximations of least fixpoints in the next section. Proving termination amounts to showing the absence of following three situations:

1. There is an infinite sequence of imbricated calls to procedure `repeat_computation`.
2. The **repeat until** loop in procedure `repeat_computation` is executed infinitely often.
3. Execution of β := `tau(pretended_f,`α`)` contains an internal loop or calls `pretended_f` infinitely often. This situation may arise since the algorithm does not use `tau` with a constant (functional) argument and is the most difficult one to preclude.

5.2.1 Sufficient Conditions for Termination

Hypothesis 3: *A is finite.*

Hypothesis 3 precludes situation 1 since the test $\alpha \notin \text{dom}(\texttt{dg}) \cup \texttt{ics}$ ensures that imbricated calls have different input values and thus bounds the number of imbricated calls with $\#A$.

Hypothesis 4: *$A \not\mapsto A'$ is a noetherian cpo.*

Hypothesis 4 precludes situation 2, since operations `Extend_pt` and `Adjust_pt` return a partial table greater or equal to `pt` and hence ensures that the partial table `pt` is increasing.

Situation 3 requires some additional assumptions about `tau`. An obvious condition is to require `tau` to terminate for any terminating functional parameter when triggered at any point in procedure for any internal state σ. This condition is satisfied for many applications but is still too strong a restriction as it would reject the procedure:

```
function tau(f:A→A; α:A):A;
   var σ:  A;
   begin
      σ := ⊥;
      while f(σ) <> σ do σ := f(σ);
      tau := σ
   end.
```

If A is noetherian, the procedure terminates for any monotone function f but may loop on non monotone functions. An example is $A = \{\bot, \top\}$, $f(\bot) = \top$, $f(\top) = \bot$. Moreover, the procedure may even loop for some monotone f if triggered in some "bad" states. For instance, assuming $A = \{\bot, 0, 1\}$ where $\bot \leq 0, \bot \leq 1, 0 \not\leq 1, 1 \not\leq 0$ and $f(\bot) = \bot$, $f(0) = 1$, $f(1) = 0$, procedure `tau` loops if triggered at the beginning of the **while** loop with $\sigma = 0$.

A weaker condition can in fact be imposed.

Hypothesis 5.a: Procedure `tau` terminates when started in a state "below the least fixpoint" or, in symbols,

$$\forall f \in (A \mapsto A') : \forall \sigma, \sigma' \in \Sigma : f \leq \mu\tau \;\&\; \sigma \leq \sigma' \;\&\; \exists \alpha \in A : \theta(\mu\tau)\sigma' = \mu\tau\alpha \;\Rightarrow\; \theta f \sigma \neq \textbf{undef}.$$

This somewhat unnatural hypothesis is implied by the more natural

Hypothesis 5.b:

$$\forall f_1, f_2 \in (A \mapsto A') : \forall \sigma_1, \sigma_2 \in \Sigma : (f_1 \leq f_2 \;\&\; \sigma_1 \leq \sigma_2) \Rightarrow (\theta f_2 \sigma_2 \neq \textbf{undef} \Rightarrow \theta f_1 \sigma_1 \neq \textbf{undef}).$$

Note the relationship of this condition with the monotonicity condition for partial correctness. Note also that, in practical applications, it is often possible to modify the code of procedure `tau` to ensure termination for arbitrary states and arbitrary total functional parameters without changing the value of the least fixpoint. Consider our example again and assume that A is endowed with a *lub* operation. The previous procedure can be modified to give:

```
function tau(f:A→A; α:A):A;
   var σ:  A;
   begin
      σ := ⊥;
      while f(σ) <> σ > do σ := lub(σ,f(σ));
      tau := σ
   end.
```

We now prove that hypothesis 5.a precludes situation 3. Let σ be an internal state during a call tau(pretended_f,α). The corresponding partial table pt can be embedded in a monotone function f ($\mathtt{pt} \subseteq f$) such that $f \leq \mu\tau$. Hence $\theta f \sigma \neq$ **undef**. It follows that no internal loop may occur in tau until the next call to pretended_f. By induction on execution length, no internal loop occurs. It remains to prove that no execution of tau calls pretended_f infinitely often. Consider the execution point where pt has reached its maximal value. Let f be a total monotonic function such that ($\mathtt{pt} \subseteq f$) and $f \leq \mu\tau$. All further calls pretended_f(α, α_i) return the value pt(α_i), i.e. $f(\alpha_i)$. Therefore, for any further state σ reached in a call to tau, $\theta f \sigma \neq$ **undef** holds, implying termination.

6 Extensions

In many applications such as abstract interpretation, it is acceptable to compute an upper approximation of the least fixpoint. We now show how to broaden the applicability of the algorithm for these problems. In particular, A can now be infinite and some of the monotonicity conditions can be relaxed.

6.1 Infinite Domains

Definition 3 [7] Let A be a poset. A widening operator on A is a function: $\nabla : A \times A \rightarrow A$ satisfying

1. for each infinite sequence $x_0, x_1, \ldots, x_i, \ldots$ ($x_k \in A$), the sequence $y_0, y_1, \ldots, y_i, \ldots$ is increasing and stationary, where $y_0 = x_0$ & $y_{i+1} = y_i \nabla x_{i+1}$ ($i \geq 0$).

2. $\forall x, y \in A : x, y \leq x \nabla y$.

Widening operators can be used to prevent the set ics of initiated computations from growing infinitely when A is an infinite set. It suffices to modify the universal algorithm to work on a finite number of increasing sequences, each of which constructed with a widening operator. When repeat_computation(α) is called, the first instruction of procedure repeat_computation now becomes α := y ∇ α where y is the last element in one of the sequences in the set ics. The rest of the algorithm is left unchanged. Now the condition $\mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})} \geq (\mu\tau)_{|\mathrm{dom}(\mathtt{dg})}$ holds when the modified algorithm terminates. Alternatively, widening techniques are also useful to speed up the algorithm when convergence is slow.

6.2 Non Monotone Transformations

In abstract interpretation, it is sometimes difficult, if not almost impossible, to prove monotonicity of transformation τ. (See [11, 21].) Fortunately, monotonicity can be dropped since the first part of the partial correctness proof does not use the hypotheses on τ, θ and r. The algorithm then computes a partial postfixpoint of τ (assuming termination), which is still useful in abstract interpretation, since it is an upper approximation of the least fixpoint in the concrete domain. However, stronger conditions are needed for termination: procedure tau must terminate for any internal state σ and total function parameter f.

Widening techniques can be applied in the case of non monotone transformation to broaden further the applicability. As suggested by the previous example, they can be also applied "inside" procedure `tau` to ensure its termination.

7 Instantiations and Optimizations

7.1 Existing Instantiations

The universal top-down fixpoint algorithm is a generalization of several specific algorithms that we developed for the abstract interpretation of Prolog.

The generic algorithm described in [15, 14, 21] is an instantiation of the universal algorithm where the transformation τ is a simple input/output abstract semantics based on SLD-resolution, the standard operational semantics for Prolog (see [17]). In this instantiation, τ is specific to a given program and the cpos A and A' are the same. The worst case complexity (i.e. the number of iterations in the repeat loop of procedure `repeat_computation`) was analyzed for several interesting program classes and behavioral assumptions. The worst case complexity is $O(n^2hs^2)$ where n is the size of the analyzed program, and h and s are the height and the size of the cpo A. However, under reasonable assumptions satisfied by many programs or subprograms, the complexity reduces to $O(nhs^2)$, $O(nhs)$, and $O(nh)$ for several classes of recursive programs. The algorithm was evaluated experimentally on real programs using a complex infinite abstract domain containing modes, sharing, and structural information. The number of iterations is $h'n$ in the average and bounded by $3h'n$ (where h' is the length of the longest explored increasing sequence), showing the practicability of the approach. In addition, the algorithm explores at most 15% of unnecessary elements and none for most programs.

The reexecution algorithm described in [16] is also an instantiation of the universal algorithm for a much more complex semantics exploiting referential transparency to improve accuracy of the analysis. Although the semantics was much more complex, the derivation algorithm and its correctness proof were substantially simplified because of the availability of the universal algorithm.

7.2 Optimizations

In [16], we have described two general optimizations for fixpoint algorithms. Both of them produced about 30% improvement on the first instantiation. The second optimization was also integrated in the second instantiations and its effect should be even more important (although no explicit comparison was made).

The first optimization (the so-called prefix optimization) applies when procedure `tau` is reexecuted on an input value $\alpha \in A$. Rather than restarting the execution of `tau` from scratch, it is more efficient to restart the execution from the first call to the functional parameter which gives a value different from the previous execution. This idea can be implemented with a data structure keeping, for each value α, the internal states $\sigma_1, \ldots, \sigma_n$ of procedure `tau` before the calls to the functional parameter with input values $\alpha_1, \ldots, \alpha_n$. This data structure, which can be seen as a dependency graph with a finer granularity, needs to be updated each time `pt(`α_j`)` is refined to remove $\sigma_{j+1}, \ldots, \sigma_n$. The recomputation starts from the internal state with the largest k still in the data structure at reexecution time. More sophisticated optimizations are possible when procedure `tau` can be decomposed in several independent subparts, avoiding reexecution of non-affected subparts even when an earlier call has an improved value. Deriving the algorithm automatically is however more complicated in this case since it is necessary to consider the internal structure of `tau`.

The second optimization is based on the fact that reexecution on the same value redoes many internal operations with the same input arguments. These operations can in fact be cached so that they are only computed once. Although this optimization is rather systematic, it also requires considering the internal structure of `tau`.

Finally, a last optimization that may turn useful amounts to detecting when an element α has reached its final value, i.e. $\mathtt{pt}\alpha = \mu(\tau)\alpha$. This is the case when procedure `tau` on α does not make use (directly or indirectly) of any element in the set `ics`. All subsequent calls to `tau` with α can thus be avoided. This optimization does not require considering the internal structure of `tau`.

8 Related Work and Discussion

8.1 Bottom-up Fixpoint Algorithms

In this paper, we view the computation of the least fixpoint of a monotone transformation as a generalization of computing the values of a function from its recursive definition. This leads to a top-down approach in contrast to the more usual approach inspired by the Kleene's sequence. We now compare our algorithm with a representative bottom-up algorithm proposed by O'Keefe [24].

O'Keefe's algorithm solves finite sets of equations of the form $x_i = expr_i \ \ (1 \leq i \leq n)$ where $x_1, \ldots, x_n$ are distinct variables, ranging on lattices of finite depth $T_1, \ldots, T_n$, and $expr_1, \ldots, expr_n$ are well-typed monotone expressions possibly involving $x_1, \ldots, x_n$. The algorithm computes the least fixpoint of the transformation

$$\begin{array}{rl} \tau : & T_1 \times \ldots \times T_n \mapsto T_1 \times \ldots \times T_n \\ & \langle x_1, \ldots, x_n \rangle \rightsquigarrow \langle expr_1, \ldots, expr_n \rangle \end{array}$$

It proceeds as follows. Variables x_1 to x_n are initialized to $\perp$ and pushed onto a stack. Then variables are popped from the stack until it becomes empty. Each time a variable x_i is popped its value is recomputed. If the new value is greater than the previous one, all variables that depends on x_i and are not on the stack are pushed on it.

Our algorithm is obviously applicable to the same class of problems (with $A = \{1, \ldots, n\}$ and $A' = T_1 + \ldots + T_n$). It can also be shown that both algorithms have the same worst case complexity on this class of problems. Moreover, none of the algorithms outperforms the other. For example, our algorithm is optimal for equations of the form:

$$x_{j_i} = expr_{j_i} \langle x_{j_1}, \ldots, x_{j_{i-1}} \rangle \ \ (1 \leq i \leq n)$$

where $j_1, \ldots, j_n$ is a permutation of $1, \ldots, n$, since each expression is computed at most once while O'Keefe's algorithm is not optimal.

Example 4 Consider the following system:

$$\begin{array}{c} x_2 = x_1 \overline{+} 1 \\ \vdots \\ x_n = x_{n-1} \overline{+} 1 \\ x_1 = 1 \end{array}$$

where $T_1 = \ldots = T_n = \{0, \ldots, n\}$ and $x \overline{+} y = min(n, x+y)$. O'Keefe's algorithm requires at least $2(n-1)$ additions and $(n-1)(n-2)/2$ in the worst case depending on how $x_2, \ldots, x_n$ are put onto the stack.

Conversely, there are also situations where O'Keefe's algorithm gives better results.

Example 5 Consider the following system ($n \geq 3$):

$$\begin{aligned}
x_1 &= max(x_2, x_{n-2}) \\
&\vdots \\
x_{n-3} &= max(x_{n-2}, x_{n-2}) \\
x_{n-2} &= \textbf{ if } x_{n-1} = n \textbf{ then } n \textbf{ else } 0 \\
x_{n-1} &= x_n \overline{+} 1 \\
x_n &= max(x_1, x_{n-1})
\end{aligned}$$

Computation of x_n with our algorithm requires $n(n+1)$ operations while O'Keefe's algorithm only requires $7n - 11$.

As far as applicability is concerned, it is clear that O'Keefe's algorithm addresses a considerably more restrictive class than ours. The two main restrictions are the following:

1. A reduces to the finite set $\{1, \ldots, n\}$, ordered pointwise. (A' is the disjoint product $T_1 + \ldots + T_n$.)
2. The functional transformation τ is not defined by an arbitrary algorithm but is constrained to be defined by a system of n equations.

It is nevertheless possible to adapt O'Keefe's algorithm to deal with more general problems. However, O'Keefe's algorithm becomes substantially more complicated compared to ours. As an example, consider the problem of solving recursive equations of the form

$$f(x) = expr\langle f, x\rangle$$

on a finite lattice T, where *expr* is an expression built from f, x and constant symbols denoting monotone functions. To compute $f(v)$ where $v \in T$, we can partially evaluate $expr\langle f, v\rangle$ and identify other subexpressions, say $f(w)$ for which we iterate the process. We obtain a finite set of equations and O'Keefe's algorithm is applicable. The problem is more difficult when f appears in subexpressions of expressions of the form $f(expr_1, \ldots, expr_n)$. In order to evaluate the subexpressions we can initialize $f(v)$ to the bottom element of T for each $v \in T$. So a first set of equations can be derived and solved. The solution is used to derive a new set of equations which is then solved, and so on until no greater values are obtained for f. This complicated process should be contrasted with the application of our algorithm to a procedure `tau` defined as follows:

```
function tau(f:T→T; α:T):T;
   begin  tau:=expr⟨f,α⟩   end.
```

The previous discussion can also be related to the *minimal function graph semantics* discussed in the paper and to bottom-up computation of recursive functions (see [3]).

8.2 Chaotic Iteration Algorithms

Chaotic iteration was introduced by P. Cousot in [7] as a very general class of methods for finding the least solution of a set of equations of the form $x_i = expr_i \ \ (1 \leq i \leq n)$. A chaotic iteration strategy for such a set of equations defines a finite sequence $i_1, \ldots, i_m$ such that the "loop":

$(x_1, \ldots, x_n)$`:=`$(\perp, \ldots, \perp)$`;`
for `j:= 1` **to** `m` **do** x_{i_j} `:= expr`$_{i_j}$`<`$x_1, \ldots, x_n$`>`

compute the least solution of the system. The best strategy minimizes the amount of computation (roughly speaking, the number m of iterations). Clearly, this is very problem dependent. O'Keefe's algorithm provides a standard (bottom-up) chaotic iteration strategy while ours provides a top-down one. However, chaotic iteration seems difficult to generalize to the class of problems considered in this paper.

8.3 Deductive Data Bases

Fixpoint computation algorithms have been extensively studied to solve efficiently queries in deductive data bases (e.g. [5, 27, 28]). Bottom-up and top-down approaches are both used and have been related by means of rewriting techniques (e.g. magic sets [1, 27]). Our algorithm should be related to OLDT-resolution [26]. However, our algorithm is inefficient in this context because elements of A and A' denote sets of values (facts) instead of "individual" values. As a consequence, each iteration redoes all computations performed by the previous ones. Our algorithm can be adapted to this type of problems to derive an efficient and general version of OLDT. The idea is to replace "single" valued functions by "multi" valued ones, delivering values one at a time, and to use the prefix optimization cleverly. We will investigate this idea in future research.

8.4 Abstract Interpretation

Abstract interpretation is a general methodology for building static analyses of programs. It was introduced by P. and R. Cousot in [7]. The original idea was subsequently adapted, reformulated, and applied to many programming paradigms. (Approximate) computation of fixpoints is a main issue in abstract interpretation. In the following we relate our algorithm to several aspects and approaches in this field. We stress its usefulness in such a context both from a practical and a methodological point of view.

Operational Frameworks The operational approach to abstract interpretation reduces to the idea of executing the program on a non standard (abstract) domain. This leads either to specific algorithms (e.g. [8, 2, 10]) or to the so-called operational frameworks [4, 6]. Those approaches are difficult to prove correct, to implement, and to modify, since they mix semantic and algorithmic aspects. A more systematic approach consists in separating both issues. Defining a new class of static analyses then reduces to the definition of a non standard fixpoint semantics with respect to which the universal algorithm can be instantiated (the semantics maps each program to a procedure `tau`). Hence, a general algorithm parametrized on the abstract domain is obtained. The advantages of the approach are twofold. Most optimizations can be handled at the algorithmic level and the semantics can be changed without requiring any substantial modification of the algorithm.

Denotational Frameworks The denotational approach to abstract interpretation was first introduced by F. Nielson [22]. It was then further developed by many authors including [13, 19, 29]. In this approach, the elegant notations of denotational semantics and the mathematical framework of the Cousot's are put together to provide high level descriptions of static analyses. Authors relying on this approach are mostly concerned with semantic issues. They assume the existence of well-accepted and efficient algorithms for fixpoint computation. O'Keefe algorithm is often referred to as well as algorithms designed for deductive data bases and OLDT-resolution. Our universal algorithm provides a versatile alternative to O'Keefe's algorithm to implement the denotational approach given its wide applicability. Moreover, OLDT-resolution is not especially adequate for abstract interpretation, since many results are produced for each single input leading to an exponential algorithm.

Minimal Function Graph and Query Directed Semantics To overcome the applicability problem of O'Keefe's algorithm (and more generally of any bottom-up evaluation), many authors design "instrumental" semantics introducing supplementary results needed only by the algorithm for fixpoint computation. They are called *minimal function graph semantics* in [12, 29] and *query directed semantics* in [18]. The basic idea is that each iteration provides the set of input values for the next iteration in addition to the output values corresponding to the previous inputs. Such an instrumental semantics is no longer needed with our approach (see the comparison with O'Keefe's algorithm). However, the additional results provided by those derived semantics are sometimes useful for applications. For example, the minimal function graph semantics is useful for program specialization [29]. The same information is obtained by instantiating our algorithm to a simple input/output semantics, also called *total function graph* semantics. Note that $\mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})}$ is the required minimal function graph.

Collecting Semantics Instrumental semantics designed for bottom-up computation purposes must be distinguished from *collecting semantics*. Collecting semantics characterizes *explicitly* properties that are present only implicitly in a simpler abstract semantics. An obvious example is given by an abstract semantics collecting information at each program point as opposed to a simpler input/output semantics (for loops and procedures). Oddly enough, it happens sometimes that the minimal function graph and collecting semantics collapse. In these cases, our universal algorithm can be instantiated to the simple input/output semantics and provides for free the results of the collecting one. In the general case (the collecting semantics does not reduce to the minimal function graph), it is more efficient to compute the needed information in two steps. First, the universal algorithm is applied to the simple semantics. Then, the equations defining the collecting semantics are executed once, using the results of the first step.

Granularity *Granularity* is a useful criterium to compare Abstract Interpretation algorithms (and frameworks). It is specified by giving the "program points" where information about all possible executions is considered. It is *finer* if more program points are considered, *coarser* otherwise. It is *static* if the pieces of information corresponding to all execution paths leading to the same point are lumped together. Static granularity has the advantage that a finite set of equations can be associated with the analyzed program. *Dynamic* granularity is achieved when different pieces of information can be associated with the same program point, possibly corresponding to different execution paths. For a fixed abstract domain, a finer granularity is likely to give more precise results at the price of a higher computation cost. Similarly, dynamic granularity is a priori more accurate and costly than static granularity. Usefulness of granularity to compare abstract interpretation frameworks can be illustrated by the works of C. Mellish [20], U. Nilsson [23] and M. Bruynooghe [4], which proposed frameworks for the abstract interpretation of (pure) Prolog.

The earlier work is due to Mellish. It has static granularity and only two program points are considered for each procedure: procedure entry and procedure exit. The abstract semantics can be expressed as system of $2n$ equations with $2n$ variables where n is the number of procedures. Since this system is small, the naive bottom-up method based on the Kleene's sequence can be used as done in Mellish's algorithm. O' Keefe's paper [24] was motivated by Mellish's work and provides a faster method.

The framework of Nilsson has static but finer granularity: all program points are considered (clause entry and exit, and any point between two calls). This results in a system of m equations with m variables where m is the number of program points. The algorithm of is essentially O'Keefe's one and it is easy to exhibit programs where it gives more precise results than Mellish's algorithm.

Bruynooghe's framework uses the same program points as Nilsson but with dynamic granularity. We have presented in [14] a generic abstract interpretation algorithm for Prolog wich can be seen as a precise implementation of Bruynooghe's framework. Alternatively it can be seen as an instantiation of the universal algorithm of this paper to a simple input/output semantics for Prolog.

It is possible to derive abstract interpretation algorithms with the same granularity as Mellish and Nilsson by instantiating the universal algorithm to appropriate (but more complicated) instrumental semantics. An other approach to investigate in the future is the definition of universal algorithms that should automatically provide a coarser granularity when instantiated to a simple (input/output) abstract semantics. Those algorithms should use widening techniques and/or introduce "hooks" in the `tau` procedure to control the granularity level. Hence, we conclude that semantic and algorithmic issues in abstract interpretation are in some sense dual. Tuning the granularity can be done at both levels just as a matter of convenience.

Acknowledgements

Henry Leroy pointed out that our previous algorithms would work in a much more general setting. This research was partly supported by the Belgian National Incentive-Program for fundamental Research in Artificial Intelligence (Baudouin Le Charlier) and by the National Science Foundation under grant number CCR-9108032 and by the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-91-J-4052 (Pascal Van Hentenryck).

References

[1] F. et al. Bancilhon. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of Fifth ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, 1986.

[2] A. Bansam and L. Sterling. An abstract interpretation scheme for logic programs based on type expression. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 422–429, Tokyo, 1988.

[3] G. Berry. Bottom-up Computation of Recursive Programs. In *RAIRO-rouge 10(3)*, pages 47–82. March 1976.

[4] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[5] F. Bry. Query Evaluation in Recursive Databases. In *Proc. First Int. Conf. on Deductive and Object-Oriented Databases*, pages 20–39, Kyoto, Japan, 1989.

[6] P. Codognet and G. Filè. Computations, Abstractions and Constraints in Logic Programs. In *Proceedings of the fourth International Conference on Programming languages (ICCL'92)*, Oakland, U.S.A., April 1992.

[7] 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 Fourth ACM Symposium on POPL*, pages 238–252, Los Angeles, CA, 1977.

[8] S. Debray. Efficient Dataflow Analysis of Logic Programs. In *Proc. of 15th Annual Symposium on POPL*, pages 260–273, San Diego, CA, 1988.

[9] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. In *Fourth International Symposium on Programming Language Implementation and Logic Programming (PLILP-92)*, Leuven (Belgium), August 1992.

[10] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 1991. (To appear).

[11] G Janssens. *Deriving Run Time Properties Of Logic Programs By Means of Abstract Interpretation.* PhD thesis, Katholieke Universiteit Leuven, Department Computerwetenschappen, Leuven (Belgium), 1990.

[12] N.D. Jones and A. Mycroft. Dataflow Analysis of Applicative Programs using Minimal Function Graphs. In *Proceedings of 13th ACM symposium on Principles of Programming Languages*, pages 123–142, St. Petersburg, Florida, 1986.

[13] N.D. Jones and H. Sondergaard. *A Semantics-Based Framework for the Abstract Interpretation of Prolog*, volume Abstract Interpretation of Declarative Languages, pages 123–142. Ellis Horwood, 1987.

[14] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis (Extended Abstract). In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[15] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Fransisco, CA, April 1992.

[16] B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. Technical Report CS-92-12, CS Department, Brown University, 1992. (44 pages).

[17] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 1984.

[18] K. Marriott and H. Sondergaard. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, pages 601–606, San Fransisco, CA, 1989.

[19] K. Marriott and H. Sondergaard. Abstract Interpretation of Logic Programs: the Denotational Approach, June 1990. To appear in ACM Transaction on Programming Languages.

[20] C. Mellish. *Abstract Interpretation of Prolog Programs*, volume Abstract Interpretation of Declarative Languages, pages 181–198. Ellis Horwood, 1987.

[21] K. Musumbu. *Interpretation Abstraite de Programmes Prolog.* PhD thesis, University of Namur (Belgium), September 1990.

[22] F. Nielson. A Denotational Framework for Data Flow Analysis. *Acta Informatica*, 18:265–287, 1982.

[23] U. Nilsson. Systematic Semantic Approximations of Logic Programs. In *Proceedings of PLILP 90*, pages 293–306, Linkoeping, Sweeden, August 1990.

[24] R.A. O'Keefe. Finite Fixed-Point Problems. In J-L. Lassez, editor, *Fourth International Conference on Logic Programming*, pages 729–743, Melbourne, Australia, 1987.

[25] J. Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory* . MIT Press, Cambridge Mass., 1977.

[26] H Tamaki and T. Sato. OLD-resolution with Tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, July 1986.

[27] J.D. Ullman. Bottom-up Beats Top-Down for Datalog. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 140–149, 1989.

[28] L. Vieille. Recursive Axioms in Deductive Databases : the Query/Subquery Approach. In *Proceedings of the First International Conference on Expert Databases Systems*, pages 179–193, Charleston, South Carolina, April 1986.

[29] W.H. Winsborough. A minimal function graph semantics for logic programs. Technical Report TR-711, Computer-Science Department, University of Wisconsin at Madison, August 1987.

9 Appendix 1: Proof of Partial Correctness

9.1 Invariant Conditions

An Invariant Condition is a condition that hold "almost" always during the execution of the algorithm. To reuse them extensively, invariant conditions are parametrized on the values of the used objects: `dg`, `ics`, `pt`. When used without parameters, their original textual form is intended. We use the following invariant conditions:

$$\begin{array}{ll}
\mathrm{Inv}_1\langle \mathtt{dg}, \mathtt{ics}\rangle : & \mathrm{codom}(\mathtt{dg}) \subseteq \mathrm{dom}(\mathtt{dg}) \cup \mathtt{ics} \\
\mathrm{Inv}_2\langle \mathtt{dg}, \mathtt{ics}, \mathtt{pt}\rangle : & \mathrm{dom}(\mathtt{dg}), \mathtt{ics} \subseteq \mathrm{dom}(\mathtt{pt}) \\
\mathrm{Inv}_3\langle \mathtt{dg}, \mathtt{ics}, \mathtt{pt}\rangle : & \forall x \in \mathrm{dom}(\mathtt{dg}) - \mathtt{ics} : \left\{ \begin{array}{l} x \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(x)}, \\ \mathtt{pt}x \geq \tau \mathtt{pt} x \end{array} \right. \\
\mathrm{Inv}_4\langle \mathtt{pt}\rangle : & \mathtt{pt} \leq \mu(\tau) \\
\mathrm{Inv}\langle \mathtt{dg}, \mathtt{ics}, \mathtt{pt}\rangle : & \bigwedge_{i=1}^{4} \mathrm{Inv}_i
\end{array}$$

9.2 General Post Conditions

General Post Conditions relates objects values at two different execution times called initial and final. Those conditions are intended to be used frequently in postconditions of the procedures. We use the following general post conditions:

$$\begin{array}{ll}
\mathrm{GP}_1\langle \mathrm{ics}_0, \mathtt{ics}\rangle : & \mathtt{ics} = \mathtt{ics}_0 \\
\mathrm{GP}_2\langle \mathrm{dg}_0, \mathtt{dg}, \mathrm{ics}_0\rangle : & \mathrm{dom}(\mathtt{dg}) \cap \mathrm{ics}_0 \subseteq \mathrm{dom}(\mathrm{dg}_0) \\
\mathrm{GP}_3\langle \mathrm{pt}_0, \mathtt{pt}\rangle : & \mathrm{pt}_0 \leq \mathtt{pt} \\
\mathrm{GP}_4\langle \mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg}\rangle : & U(\mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg}) \subseteq \mathrm{dom}(\mathtt{dg}) \\
\mathrm{GP}_5\langle \mathrm{dg}_0, \mathrm{ics}_0, \mathrm{pt}_0, \mathtt{dg}, \mathtt{pt}\rangle : & \forall x \in U(\mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg}) : \left\{ \begin{array}{l} \overline{\mathtt{dg}}(x) = \overline{\mathrm{dg}}_0(x), \\ \forall y \in \mathtt{dg}^+(x) : \mathtt{pt}y = \mathrm{pt}_0 y \end{array} \right. \\
\mathrm{GP}\langle \mathrm{dg}_0, \mathrm{ics}_0, \mathrm{pt}_0 \mathtt{dg}, \mathtt{ics}, \mathtt{pt}\rangle : & \bigwedge_{i=1}^{5} \mathrm{GP}_i
\end{array}$$

The new notations used by the post conditions are defined as follows. Let `dg` be a dependency graph and $\alpha \in \mathrm{dom}(\mathtt{dg})$. $\overline{\mathtt{dg}}(\alpha)$ is the subset of `dg` "reachable" from α, i.e.

$$\overline{\mathtt{dg}}(\alpha) = \{(\beta, S) \in \mathtt{dg} \mid \beta \in \mathtt{dg}^*(\alpha)\}.$$

Intuitively, $U(\mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg})$ contains the values whose elements they depend upon are left unchanged. Formally, it is defined by

$$U(\mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg}) \;=\; \bigcup_{x \in \mathrm{dom}(\mathtt{dg}) \,\cap\, \mathrm{ics}_0} \mathtt{dg}_0^*(x).$$

9.3 Specifications of the Procedures

We now turn to the specifications of the procedures.

```
procedure repeat_computation(in α:A);
   var β:A';
   begin
      if α ∉ dom(dg) ∪ ics then
      begin
             if α ∉ dom(pt) then Extend_pt(pt,α);
             ics := ics ∪ {α};
             repeat {A1}
                Extend_dg(dg,α) {A2};
                β := tau(pretended_f,α) {A3};
                Adjust_pt(pt,α,β,modified) {A4};
                Remove_dg(dg,modified) {A5}
             until α ∈ dom(dg);
             ics := ics − {α}
          end
   end
```

Figure 2: The annotated `repeat_computation` Procedure

Specification 6 $\{$`repeat_computation(`α`)`$\}$

Pre : Inv
Post : Inv & GP & $\alpha \in \mathrm{dom}(\mathtt{dg})$

Specification 7 $\{\beta$ `:= tau(pretended_f,`α`)`$\}$

Pre : Inv & $\alpha \in \mathtt{ics}$ & $\overline{\mathtt{dg}}(\alpha) = \{<\alpha, \{\}>\}$
Post : Inv & GP & $\beta \leq \mu\tau\alpha$ &

$$\alpha \in \mathrm{dom}(\mathtt{dg}) \Rightarrow \left\{ \begin{array}{l} \alpha \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(\alpha)} \\ \beta = \tau\mathtt{pt}\alpha \end{array} \right\}$$

Specification 8 $\{$`compute_fixpoint(`α`,pt,dg)`$\}$

Pre : **true**
Post : $\alpha \in \mathrm{dom}(\mathtt{dg})$ & $\mathrm{codom}(\mathtt{dg}) \subseteq \mathrm{dom}(\mathtt{dg})$ &
$\forall x \in \mathrm{dom}(\mathtt{dg}) : x$ is based in $\mathtt{pt}_{|\mathtt{dg}(x)}$ &
$\mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})} = \mu(\tau)_{|\mathrm{dom}(\mathtt{dg})}$

9.4 Partial Correctness of `repeat_computation(`α`)`

Partial correctness of Procedure `repeat_computation` relies on five intermediate assertions A_1, $\ldots$,A_5 that hold where indicated in the annotated version of the procedure (see Figure 2). The assertions are depicted in Figure 3.

We prove statements of the form $\{A_i\}\ S\ \{A_{i+1}\}$. In the proofs, dg_i, ics_i and pt_i denote the values of `dg`, `ics` and `pt` before execution of S. To denote the final values, the symbols `dg`, `ics` and

$$\mathrm{A}_1 : \left\{ \begin{array}{l} \mathrm{Inv} \ \& \ \alpha \notin \mathrm{ics}_0 \ \& \ \mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\} \ \& \\ \mathrm{GP}_{2..5} \ \& \ \alpha \notin \mathrm{dom}(\mathtt{dg}) \end{array} \right.$$

$$\mathrm{A}_2 : \left\{ \begin{array}{l} \mathrm{Inv} \ \& \ \alpha \notin \mathrm{ics}_0 \ \& \ \mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\} \ \& \\ \mathrm{GP}_{2..5} \ \& \ \alpha \in \mathrm{dom}(\mathtt{dg}) \ \& \ \mathtt{dg}(\alpha) = \{\} \end{array} \right.$$

$$\mathrm{A}_3 : \left\{ \begin{array}{l} \mathrm{Inv} \ \& \ \alpha \notin \mathrm{ics}_0 \ \& \ \mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\} \ \& \\ \mathrm{GP}_{2..5} \ \& \\ \alpha \in \mathrm{dom}(\mathtt{dg}) \Rightarrow \left\{ \begin{array}{l} \alpha \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(\alpha)} \\ \beta = \tau \mathtt{pt} \alpha \end{array} \right\} \ \& \\ \beta \leq \mu \tau \alpha \end{array} \right.$$

$$\mathrm{A}_4 : \left\{ \begin{array}{l} \mathrm{Inv}_{1..2,4} \ \& \ \alpha \notin \mathrm{ics}_0 \ \& \ \mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\} \ \& \\ \forall x \in \mathrm{dom}(\mathtt{dg}) - \mathtt{ics} : (\mathtt{dg}^+(x) \cap \mathtt{modified} = \{\}) \Rightarrow \left\{ \begin{array}{l} x \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(x)} \\ \mathtt{pt} x \geq \tau \mathtt{pt} x \end{array} \right\} \ \& \\ \alpha \in \mathrm{dom}(\mathtt{dg}) \Rightarrow \left\{ \begin{array}{l} \alpha \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(\alpha)} \\ \mathtt{pt} \alpha \geq \tau \mathtt{pt} \alpha \end{array} \right\} \ \& \\ \mathrm{GP}_{2..4} \ \& \\ \forall x \in U(\mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg}) : \left\{ \begin{array}{l} \overline{\mathtt{dg}}(x) = \overline{\mathrm{dg}}_0(x), \\ \forall y \in \mathtt{dg}^+(x) - \mathtt{modified} : \mathtt{pt} y = \mathrm{pt}_0 y \end{array} \right\} \end{array} \right.$$

$$\mathrm{A}_5 : \left\{ \begin{array}{l} \mathrm{Inv} \ \& \ \alpha \notin \mathrm{ics}_0 \ \& \ \mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\} \ \& \\ \alpha \in \mathrm{dom}(\mathtt{dg}) \Rightarrow \left\{ \begin{array}{l} \alpha \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(\alpha)} \\ \mathtt{pt} \alpha \geq \tau \mathtt{pt} \alpha \end{array} \right\} \ \& \\ \mathrm{GP}_{2..5} \end{array} \right.$$

Figure 3: Assertions for `repeat_computation`

pt are used. To simplify notation, we introduce some notational conventions for assertions. Let As be an assertion containing possibly occurrences of the symbols dg, ics, pt, dg_0, ics_0 and pt_0. We denote $\mathrm{As}^{(i)}$ the assertion As where dg_i, ics_i and pt_i have been substituted to dg, ics and pt. Similarly we denote $\mathrm{As}^{[i]}$ the assertion As where dg_i, ics_i and pt_i have been substituted to dg_0, ics_0 and pt_0.

The proof proceeds by symbolic execution, using the specification of `β := tau(pretended_f,α)`. Since it is rather tedious, we focus on the less obvious parts.

9.4.1 $\{A_1\}$ `Extend_dg(dg,α)` $\{A_2\}$

Straightforward consequence of the specification of `Extend_dg`.

9.4.2 $\{A_2\}$ `β := tau(pretended_f,α)` $\{A_3\}$

Due to the specification of `β := tau(pretended_f,α)`, conditions Inv, $\alpha \notin \mathrm{ics}_0$, $\mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\}$ and $\mathrm{GP}_{2..5}$ are maintained. This is obvious except for GP_3, GP_4 and GP_5. The proof of GP_3 uses the fact that $\mathrm{GP}_3^{(2)}$ and $\mathrm{GP}_3^{[2]}$ hold due to A_2 and the specification of `β := tau(pretended_f,α)`. Proofs of GP_4 and GP_5 are similar.

9.4.3 $\{A_3\}$ `Adjust_pt(pt,α,β,modified)` $\{A_4\}$

Conditions $\mathrm{Inv}_{1..2}$ are maintained because operation `Adjust_pt` does not use dg nor ics, and does not change dom(pt). Condition Inv_3 is replaced by a weaker one, i.e.

$$\forall x \in \mathrm{dom}(\mathtt{dg}) - \mathtt{ics} : (\mathtt{dg}^+(x) \cap \mathtt{modified} = \{\}) \Rightarrow \left\{ \begin{array}{l} x \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(x)} \\ \mathtt{pt}x \geq \tau\mathtt{pt}x \end{array} \right\},$$

due to the fact that $\tau\mathtt{pt}x$ can have been increased if $\mathtt{dg}^+(x)\cap\mathtt{modified} \neq \{\}$. Otherwise $\mathtt{pt}x \geq \mathrm{pt}_3 x$ due to the specification of `Adjust_pt`. Therefore: $\mathtt{pt}x \geq \mathrm{pt}_3 x \geq \tau\mathtt{pt}_3 x = \tau\mathtt{pt}x$.
Condition

$$\alpha \in \mathrm{dom}(\mathtt{dg}) \Rightarrow \left\{ \begin{array}{l} \alpha \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(\alpha)} \\ \mathtt{pt}\alpha \geq \tau\mathtt{pt}\alpha \end{array} \right\}$$

holds because of the third subcondition of $\mathrm{A}_3^{(3)}$ and because $\mathtt{pt}\alpha \geq \beta$ due to the specification of `Adjust_pt`. Conditions $\alpha \notin \mathrm{ics}_0$ & $\mathtt{ics} = \mathrm{ics}_0 \cup \{\alpha\}$, GP_2 and GP_4 are maintained since `Adjust_pt(pt,α,β,modified)` does not use ics nor dg.
Condition Inv_4 holds due to $\mathrm{Inv}_4^{(3)}$, $\beta \leq \mu\tau\alpha$ and because

$$\forall f, g \in (A \not\mapsto A') : \forall < x, y > \in A \times A' : f \leq g \;\&\; y \leq gx \quad \Rightarrow \quad \mathrm{Adjust\_pt}(f, x, y) \leq gx.$$

Finally the condition

$$\forall x \in U(\mathrm{dg}_0, \mathrm{ics}_0, \mathtt{dg}) : \left\{ \begin{array}{l} \overline{\mathtt{dg}}(x) = \overline{\mathrm{dg}}_0(x), \\ \forall y \in \mathtt{dg}^+(x) - \mathtt{modified} : \mathtt{pt}y = \mathrm{pt}_0 y \end{array} \right\}$$

is a consequence of $\mathrm{GP}_5^{(3)}$, $\mathtt{dg} = \mathrm{dg}_3$ and of the fact that $\mathtt{pt}y = \mathrm{pt}_3 y$ if $y \notin \mathtt{modified}$.

9.4.4 $\{A_4\}$ `Remove_dg(dg,modified)` $\{A_5\}$

The proof uses condition $A_4^{(4)}$, the fact that `Remove_dg(dg,modified)` does not use `ics` nor `pt` and the following consequences of the specification of `Remove_dg`.

$$\begin{array}{l} \mathrm{dom}(\mathtt{dg}) \subseteq \mathrm{dom}(\mathrm{dg}_4)\ , \\ \forall x \in \mathrm{dom}(\mathtt{dg}) : \mathtt{dg}^+(x) = \mathrm{dg}_4^+(x). \end{array}$$

9.4.5 Proof of Partial Correctness

Let dg_0, ics_0 and pt_0 denote the values of `dg`, `ics` and `pt` at procedure entry. Correctness is straightforward if $\alpha \in \mathrm{dom}(\mathrm{dg}_0) \cup \mathrm{ics}_0$. Otherwise it is easy to see that the body of the **repeat until** instruction is executed under the precondition A_1. (Only pt will be possibly increased by the call `Extend_pt(pt,`α`).`) A_5 holds after each execution of the body. Reexecution of the body takes place when $\alpha \notin \mathrm{dom}(\mathtt{dg})$. Then A_1 is true once again since $(A_5\ \&\ \alpha \notin \mathrm{dom}(\mathtt{dg})) \Rightarrow A_1$. Finally termination of the **repeat until** instruction happens when $(A_5\ \&\ \alpha \in \mathrm{dom}(\mathtt{dg}))$ holds. This clearly implies that the postcondition of the specification holds at procedure exit.

9.5 Partial Correctness of β `:= tau(pretended_f,`α`)`

Assuming termination, procedure `tau` calls `pretended_f` finitely often, say $n(\geq 0)$ times. Let α_i denote the first actual parameter of the i-th call $(1 \leq i \leq n)$. We prove that assertion $\mathrm{Ip}\langle i\rangle$ below holds before the first call $(i = 0)$, between the calls $i-1$ and i $(1 \leq i < n)$, and after the last call if any $(i = n)$. Assertion $\mathrm{Ip}\langle i\rangle$ is given together with four auxiliary assertions $\mathrm{Ip}_1, \ldots, \mathrm{Ip}_4$. Recall that σ denotes the internal state of `tau` and θ its associated continuation function.

$$\begin{array}{ll}
\mathrm{Ip}_1 : & \mathtt{dg}(\alpha) = \{\alpha_1, \ldots, \alpha_i\} \\[1ex]
\mathrm{Ip}_2 : & \overline{\mathtt{dg}}(\alpha_j) = \overline{\mathrm{dg}}_j(\alpha_j) \ \ (\forall j : 1 \leq j \leq i) \\[1ex]
\mathrm{Ip}_3 : & \forall x \in \mathtt{dg}^+(\alpha_j) : \mathtt{pt}x = \mathrm{pt}_j x \ \ (\forall j : 1 \leq j \leq i) \\[1ex]
\mathrm{Ip}_4 : & \forall j : 1 \leq j \leq i : \mathtt{pt}\alpha_j = \mathrm{pt}_j\alpha_j \\[1ex]
\mathrm{Ip}\langle i\rangle : & \left\{ \begin{array}{l} \mathrm{Inv}\ \&\ \mathrm{GP}\ \&\ \theta(\mu\tau)\sigma \leq \mu\tau\alpha\ \& \\ \alpha \in \mathrm{dom}(\mathtt{dg}) \Rightarrow \bigwedge_{k=1}^{4} \mathrm{Ip}_k \end{array} \right.
\end{array}$$

The proof is by induction on i.

Base case $(i = 0)$ Execution of `tau` does not use `dg`, `ics` nor `pt` (except through calls to `pretended_f`). So all conditions hold trivially because they hold initially. Condition

$$\theta(\mu\tau)\sigma \leq \mu\tau\alpha$$

holds because `pretended_f` was not used. Therefore the current state is the same as for a call `tau(f,`α`)` where `f` computes $\mu\tau$.

Induction step $(1 \leq i \leq n)$ Assume that $\mathrm{Ip}\langle i-1 \rangle$ holds before the i-th call to `pretended_f`. The call `repeat_computation(`α_i`)` is valid, since Inv holds. Therefore it establishes the following condition:

$$\mathrm{Inv} \ \& \ \mathrm{GP}^{[i-1]} \ \& \ \alpha_i \in \mathrm{dom}(\mathtt{dg}).$$

Moreover conditions $\mathrm{Inv}^{(i-1)}$ and $\mathrm{GP}^{(i-1)}$ also hold. Putting all conditions together allows to deduce that Inv & GP holds after the call `pretended_f(`α`,`α_i`)`.
Now we prove that condition

$$\theta(\mu\tau)\sigma \leq \mu\tau\alpha$$

holds. Let σ' denote the internal state before the call. $\theta(\mu\tau)\sigma' \leq \mu\tau\alpha$ holds by hypothesis. Let β_i be the value returned by the call `pretended_f(`α`,`α_i`)`. By definition of r, $\sigma = r\sigma'\beta_i$. But $\beta_i \leq \mu\tau\alpha_i$ because $\beta_i = \mathtt{pt}\alpha_i$ and $\mathtt{pt} \leq \mu\tau$ due to the specification of `repeat_computation`. Therefore

$$\begin{array}{rcll} \theta(\mu\tau)\sigma & = & \theta(\mu\tau)(r\sigma'\beta_i) & \text{(definition of } \tau) \\ & \leq & \theta(\mu\tau)(r\sigma'(\mu\tau\alpha_i)) & \text{(monotonicity of } r, \mu\tau \text{ and } \theta) \\ & = & \theta(\mu\tau)\sigma' & \text{(property of } r \text{ and } \theta) \\ & \leq & \mu\tau\alpha & \text{(hypothesis)} \end{array}$$

Partial correctness As $\mathrm{Ip}\langle n \rangle$ holds at completion, Inv & GP holds. To show $\beta \leq \mu\tau\alpha$, consider the final state σ. Then:

$$\begin{array}{rcll} \beta & = & \theta(\mu\tau)\sigma & (\theta f \sigma \text{ depends on } \sigma, \text{ not on } f, \text{ for final states.}) \\ & \leq & \mu\tau\alpha & (\mathrm{Ip}\langle n \rangle) \end{array}$$

For the last condition, suppose $\alpha \in \mathrm{dom}(\mathtt{dg})$. Conditions Ip_1 and Ip_3 imply that any call `tau(`f,α`)` where $f(\alpha_j) = \mathtt{pt}(\alpha_j)$ $(\forall j : 1 \leq j \leq n)$ has the same sequence of internal states as the completed call to `tau`. Hence it returns β. Therefore

$$\alpha \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(\alpha)} \ \& \ \beta = \tau\mathtt{pt}\alpha.$$

9.6 Partial Correctness of `compute_fixpoint(`α`,pt,dg)`

Applying Specification 4 with $\mathrm{dg}_0 = \mathrm{ics}_0 = \mathrm{pt}_0 = \{\}$ results in the postcondition:

$$\begin{array}{l} \mathrm{codom}(\mathtt{dg}) \subseteq \mathrm{dom}(\mathtt{dg}) \subseteq \mathrm{dom}(\mathtt{pt}), \\ \forall x \in \mathrm{dom}(\mathtt{dg}) : \left\{ \begin{array}{l} x \text{ is based in } \mathtt{pt}_{|\mathtt{dg}(x)}, \\ \mathtt{pt}x \geq \tau\mathtt{pt}x, \end{array} \right. \\ \mathtt{pt} \leq \mu(\tau). \end{array}$$

which obviously implies:

$$\forall x \in \mathrm{dom}(\mathtt{dg}) : \mathtt{pt}x \leq \mu\tau x,$$

To show the converse, consider the transfinite Kleene's sequence $(f_\imath)_{\imath \leq \epsilon}$ defined as follows:

$$\begin{array}{rcll} f_0 & = & \perp & \\ f_\imath & = & \tau f_{\imath-1} & \text{if } \imath \text{ is not a limit ordinal} \\ & = & \tau\left(\bigsqcup_{\imath' < \imath} f_{\imath'}\right) & \text{if } \imath \text{ is a limit ordinal} \end{array}$$

with ϵ being the smallest ordinal such that $f_\epsilon = \tau f_\epsilon = \mu\tau$. Condition

$$\forall x \in \mathrm{dom}(\mathtt{dg}) : \mathtt{pt}x \geq f_\imath x$$

is proven by transfinite induction on $\imath$:

$$\mathtt{pt}x \;\geq\; \bot = f_0 x$$

$$\begin{aligned} \mathtt{pt}x &\geq \tau\mathtt{pt}x \\ &\geq \tau f_{\imath-1}x \qquad \text{(induction hypothesis, monotonicity of } \tau \text{ and } x \text{ is based in } \mathtt{pt}_{|\mathrm{dom}(\mathtt{dg})}) \\ &= f_\imath x \end{aligned}$$

when $\imath$ is a non limit ordinal. The proof is similar for limit ordinals. It follows that

$$\forall x \in \mathrm{dom}(\mathtt{dg}) : \mathtt{pt}x \geq f_\epsilon x = \mu\tau x.$$

Note Using transfinite induction seems unnatural here but is technically simpler because

1. τ is only required to be monotone wrt $\leq$ and not continuous;
2. Only partial correctness is of interest to us.

Of course, termination is possible only for values reachable in a finite number of steps (except when widening operations are used).