# Verifying a Local Generic Solver in Coq

Martin Hofmann[1], Aleksandr Karbyshev[2], and Helmut Seidl[2]

[1] Institut für Informatik, Universität München,
hofmann@ifi.lmu.de
[2] Fakultät für Informatik, Technische Universität München,
{aleksandr.karbyshev,seidl}@in.tum.de

**Abstract.** Fixpoint engines are the core components of program analysis tools and compilers. If these tools are to be trusted, special attention should be paid also to the correctness of such solvers. In this paper we consider the local generic fixpoint solver **RLD** which can be applied to constraint systems $\mathbf{x} \sqsupseteq f_{\mathbf{x}}, \mathbf{x} \in V$, over some lattice $\mathbb{D}$ where the right-hand sides $f_{\mathbf{x}}$ are given as arbitrary functions implemented in some specification language. The verification of this algorithm is challenging, because it uses higher-order functions and relies on side effects to track variable dependences as they are encountered dynamically during fixpoint iterations. Here, we present a correctness proof of this algorithm which has been formalized by means of the interactive proof assistant Coq.

## 1 Introduction

A *generic* solver computes a solution of a constraint system $\mathbf{x} \sqsupseteq f_{\mathbf{x}}, \mathbf{x} \in V$, over some lattice $\mathbb{D}$, where the right-hand side $f_{\mathbf{x}}$ of each variable $\mathbf{x}$ is given as a function of type $(V \to D) \to D$ implemented in some programming language. A *local* generic solver, when started with a set $X \subseteq V$ of *interesting* variables, tries to determine the values for the $X$ of a solution of the constraint system by touching as few variables as possible.

Local generic solvers are a convenient tool for the implementation of efficient frameworks for program analyses. They have first been proposed for the analysis of logic programs [3, 5–7] and model-checking [10], but recently have also attracted attention in interprocedural analyzers of imperative programs [1, 14]. One particularly simple instance **RLD** of a local generic solver has been included into the textbook on *Program Analysis and Optimization* [15], although without any proof of correctness of the algorithm.

Efficient solvers for constraint systems exploit that often right-hand side functions query the current variable assignment only for few variables. A generic solver, however, must consider right-hand sides as *black boxes* which cannot be preprocessed for variable dependences before-hand. Therefore, efficient generic solvers rely on *self-observation* to detect and record variable dependences on-the-fly during evaluation of right-hand sides. The local generic solver **TD** by van Hentenryck [3] as well as the solver **RLD** add a recursive descent into solving variables before reporting their values. Both self-observation through side-effects and the recursive evaluation make these solvers intricate in their operational behavior and therefore their design and implementation are error-prone.

In fact, during experimentation with tiny variations of the solver **RLD** we found that many seemingly correct algorithms and implementations are bogus. In view of the application of such solvers in tools for deriving correctness properties, possibly of safety critical systems, it seems mandatory to us to have full confidence into the applied software.

The first issue in proving any generic solver correct is which kind of functions safely may be applied as right-hand sides of constraints. In the companion paper [8] we therefore have presented a semantical property of *purity.* The notion of purity is general enough to allow any function expressed in a pure functional language without recursion, but also allows certain forms of (well-behaved) stateful computation. Purity of a function $f$ allows $f$ to be represented as a *strategy tree.* This means that the evaluation of $f$ on a variable assignment $\sigma$ can be considered as a sequence of variable look-ups followed by local computations and ending in an answer value.

It is w.r.t. this representation that we prove the local generic solver **RLD** correct. Related formal correctness proofs have been provided for variants of Kildall's algorithm for dataflow analysis [2, 4, 11, 13] This fixpoint algorithm is neither generic nor local. It also exploits variable dependences which, however, are explicitly given through the control-flow graph.

All theorems and proofs are formalized by means of the interactive theorem prover Coq [12].

## 2 The local generic solver RLD

One basic idea of the algorithm **RLD** is that, as soon as the value of variable $\mathbf{y}$ is requested during reevaluation of the right-hand side $f_{\mathbf{x}}$, the algorithm does not naively return the current value for $\mathbf{y}$. Instead, it first tries to get a better approximation of it, thus reducing the overall number of iterations and computations performed. This idea is similar to that of the algorithm **TD**.

Both algorithms also record the variable dependencies $(\mathbf{x}, \mathbf{y})$ (w.r.t. the current variable assignment) as they are encountered during evaluation of the right-hand side $f_{\mathbf{x}}$ as a *side-effect.* The main difference between the two algorithms lies in how they behave when a variable $\mathbf{x}$ changes its value. While the algorithm **TD** recursively *destabilizes* all variables which also indirectly depend on $\mathbf{x}$, the algorithm **RLD** only destabilizes the variables which immediately (locally) are influenced by $\mathbf{x}$, and triggers the reevaluation of these variables at once.

The algorithm **RLD** maintains the following data structures.

1. Finite map $\sigma$, storing current values of variables. We track only finite number of observed variables, since the overall size of set $V$ can be tremendously

large. We define the auxiliary function

$$\sigma_\perp\,\mathbf{x} = \begin{cases} \sigma\,\mathbf{x} & \text{if } \mathbf{x} \in \mathrm{dom}(\sigma), \\ \perp & \text{otherwise} \end{cases}$$

that returns a current value of $\sigma\,\mathbf{x}$ if it is defined; otherwise, it returns $\perp$.

2. Finite set *stable* $\subseteq V$. Intuitively, if variable $\mathbf{x}$ is marked as stable then either $\mathbf{x}$ is already *solved*, i.e., a computation for $\mathbf{x}$ has completed and $\sigma$ gives a solution for $\mathbf{x}$ and all those variables $\mathbf{x}$ transitively depends on, or $\mathbf{x}$ is *called* and it is in the call stack of `solve` function and its value is being processed.
3. Finite map *infl*, where dependencies between variables are stored. More exactly, *infl* $\mathbf{x}$ returns an overapproximation of a set of variables $\mathbf{y}$, for which evaluation of $f_\mathbf{y}$ on the current $\sigma_\perp$ depends on $\mathbf{x}$. Again, we track only finite number of observed variables and define the auxiliary function

$$\mathit{infl}_{[\,]}\,\mathbf{x} = \begin{cases} \mathit{infl}\,\mathbf{x} & \text{if } \mathbf{x} \in \mathrm{dom}(\mathit{infl}), \\ [\,] & \text{otherwise.} \end{cases}$$

The structures have initial values: $\sigma = \emptyset$, $\mathit{stable} = \emptyset$, $\mathit{infl} = \emptyset$.

The algorithm **RLD** proceeds as follows (see Fig. 1). The function `solve_all` is called for a list $X$ of interesting variables from the initial state (with $\sigma = \emptyset$, $\mathit{stable} = \emptyset$, $\mathit{infl} = \emptyset$). The function `solve_all` calls recursively `solve` $\mathbf{x}$ for every $\mathbf{x} \in X$.

The function `solve` when called for some variable $\mathbf{x}$ first checks whether $\mathbf{x}$ is already in the set *stable*. If so, the function returns; otherwise, the algorithm marks $\mathbf{x}$ as being stable and tries to satisfy a constraint $\sigma\,\mathbf{x} \sqsupseteq f_\mathbf{x}\,\sigma$. For that, it reevaluates a value of the right-hand side $f_\mathbf{x}$, and calculates the least upper bound *new* of the result together with the old value of $\sigma\,\mathbf{x}$. If the value of *new* is strictly larger than the old value, the function `solve` updates the value of $\sigma$ for $\mathbf{x}$. Since the value of $\sigma\,\mathbf{x}$ has changed, all constraints of variables $\mathbf{y}$ dependent on $\mathbf{x}$ may not be satisfied anymore. Hence the function `solve` *destabilizes* all the variables from $\mathit{work} = \mathit{infl}_{[\,]}\,\mathbf{x}$, i.e., subtracts *work* from the set *stable*. Then value $\mathit{infl}\,\mathbf{x}$ is reset to empty and `solve_all` *work* is recursively called.

We mention, that the right-hand side $f_\mathbf{x}$ is not evaluated *directly* on the function $\sigma$, but by using an auxiliary stateful function $\lambda y.$`eval(`$\mathbf{x}$`,`$y$`)`, allowing firstly to get better values for variables the variable $\mathbf{x}$ depends on. Once `eval(x,y)` is called, it first calls `solve` $\mathbf{y}$ and then adds $\mathbf{x}$ to $\mathit{infl}\,\mathbf{y}$. The latter reflects the fact that the value of $\mathbf{x}$ possibly depends on the value of $\mathbf{y}$. Only after recording the variable dependence $(\mathbf{x}, \mathbf{y})$, the current value of $\mathbf{y}$ is returned.

Our goal is to prove that the algorithm **RLD** is a local generic solver for any (possibly infinite) constraint system $\mathcal{S} = (V, f)$ where right-hand sides $f_\mathbf{x}$ are *pure*.

## 3 Systems of constraints

Instead of reasoning about an algorithm which modifies a global state by side-effecting functions as in Fig. 1, we prefer to reason about the *denotational se-*

```
function eval($\mathbf{x} : V,\ \mathbf{y} : V$) =
    solve($\mathbf{y}$);
    $\mathit{infl}\,\mathbf{y} \leftarrow \mathit{infl}\,\mathbf{y} \cup \{\mathbf{x}\}$;
    $\sigma_\bot\,\mathbf{y}$
function eval_rhs($\mathbf{x} : V$) =
    $f_{\mathbf{x}}(\lambda y.$eval($\mathbf{x},\ y$))

function extract_work($\mathbf{x} : V$) =
    let $\mathit{work} = \mathit{infl}_{[\,]}\,\mathbf{x}$ in
    $\mathit{stable} \leftarrow \mathit{stable} \setminus \mathit{work}$; $\mathit{infl}\,\mathbf{x} \leftarrow [\,]$;
    $\mathit{work}$
function solve($\mathbf{x} : V$) =
    if $\mathbf{x} \in \mathit{stable}$ then ()
    else
        $\mathit{stable} \leftarrow \mathit{stable} \cup \{\mathbf{x}\}$;
        let $\mathit{cur}$ = eval_rhs($\mathbf{x}$) in
        let $\mathit{new} = \sigma_\bot\,\mathbf{x} \sqcup \mathit{cur}$ in
        if $\mathit{new} \sqsubseteq \sigma_\bot\,\mathbf{x}$ then ()
        else
            $\sigma\,\mathbf{x} \leftarrow \mathit{new}$;
            let $\mathit{work}$ = extract_work($\mathbf{x}$) in
            solve_all($\mathit{work}$)
        end
    end
function solve_all($\mathit{work} : 2^V$) =
    foreach $\mathbf{x} \in \mathit{work}$ do solve($\mathbf{x}$)

begin
    $\sigma = \emptyset$; $\mathit{stable} = \emptyset$; $\mathit{infl} = \emptyset$;
    solve_all($X$);
    $(\sigma_\bot, \mathit{stable})$
end
```

**Fig. 1.** The recursive solver tracking local dependencies (**RLD**)

*mantics* of such an algorithm, i.e., about the corresponding purely functional program where the global state is explicitly threaded through the program.

Assume $\mathbb{D} = (D, \sqcup, \sqsubseteq)$ is a lattice consisting of the carrier $D$ equipped with the partial ordering $\sqsubseteq$ and the least upper bound operation $\sqcup$. A pair $(V, f)$ is a *constraint system*, where $V$ is a set of variables and $f$ is a functional of type

$$f : V \to (V \to \mathcal{M}D) \to \mathcal{M}D,$$

that for every $\mathbf{x} \in V$ returns a corresponding *right-hand side* $f_{\mathbf{x}} : (V \to \mathcal{M}D) \to \mathcal{M}D$. Here, the monad constructor $\mathcal{M}$ when applied to a set $D$, returns a computation resulting in a value from $D$. In our application, we assume $\mathcal{M}D$ to be a *state transformer* monad defined by $S \to (D \times S)$ for some set $S$ of states where $f$ is assumed to be *polymorphic* in $S$.

This means that right-hand sides may have side effects onto the global state and that they can be applied to variable assignments whose evaluation themselves may have side effects. What we assume, however, is that the side effects of the evaluation of a call $f_{\mathbf{x}}\,\sigma$ only are attributed to side-effects incurred by the evaluation of the function $\sigma$. This property is *not* captured by polymorphism in a state alone [8]. It is guaranteed, however, by the notion of *purity* introduced in [8]. If the function $f_{\mathbf{x}}$ is pure in the sense of [8], then $f_{\mathbf{x}}$ is representable by means of a *strategy tree*. This means that the evaluation of $f_{\mathbf{x}}$ on a variable assignment consists of a sequence of variable look-ups followed by some local computation leading to further look-ups and so on until eventually a result is produced.

### 3.1 Strategy trees

**Definition 1.** *For a given set of values $D$ and a set of variables $V$ we define the set $\mathcal{T}(V,D)$ of* strategy trees *inductively by:*

- *if $a \in D$ then $\mathit{Answ}(a) \in \mathcal{T}(V,D)$;*
- *if $\mathbf{x} \in V$ and $c : D \to \mathcal{T}(V,D)$ is a total function then $\mathit{Quest}(\mathbf{x},c) \in \mathcal{T}(V,D)$.*

*Let $\tau$ be a mapping from $V \to \mathcal{M}D$. By means of the monad operations* **return** $:$ *$D \to \mathcal{M}D$ and* **bind** $: \mathcal{M}D \to (D \to \mathcal{M}D) \to \mathcal{M}D$ *we define the function*

$$[\![\cdot]\!] : \mathcal{T}(V,D) \to (V \to \mathcal{M}D) \to \mathcal{M}D$$

*recursively by:*

$$\begin{aligned} [\![\mathit{Answ}(a)]\!]\,\tau &= \mathbf{return}\;a\,, \\ [\![\mathit{Quest}(\mathbf{x},c)]\!]\,\tau &= \mathbf{bind}\;(\tau\,\mathbf{x})\;(\mathbf{fun}\;a \to [\![c\;a]\!]\,\tau)\,. \end{aligned}$$

Recall that for state transformer monads, the monad operations **return** $: D \to \mathcal{M}D$ and **bind** $: \mathcal{M}D \to (D \to \mathcal{M}D) \to \mathcal{M}D$ are defined by:

$$\begin{aligned} \mathbf{return}\;a &= \mathbf{fun}\;s \to (a,s)\,, \\ \mathbf{bind}\;m\,f &= \mathbf{fun}\;s \to \mathbf{let}\;(a,s_1) = m\;s\;\mathbf{in}\;f\,a\,s_1\,. \end{aligned}$$

Therefore, the function $[\![\cdot]\!]$ is given by:

$$\begin{aligned} [\![\mathsf{Answ}(a)]\!]\,\tau &= \mathbf{fun}\;s \to (a,s)\,, \\ [\![\mathsf{Quest}(\mathbf{x},c)]\!]\,\tau &= \mathbf{fun}\;s \to \mathbf{let}\;(a,s_1) = \tau\,\mathbf{x}\,s\;\mathbf{in}\;[\![c\;a]\!]\,\tau\,s_1\,. \end{aligned}$$

The evaluation of a strategy tree thus formalizes the stateful evaluation of the pure function represented by this tree.

Moreover, if $\tau$ does not depend on the state and has no effect on the state, i.e., is of the form

$$\tau = \mathbf{return} \circ \sigma = \mathbf{fun}\;\mathbf{x} \to \mathbf{return}\;(\sigma\,\mathbf{x})$$

for some function $\sigma : V \to D$, then for all states $s$ and trees $r \in \mathcal{T}(V, D)$

$$[\![r]\!]\,\tau\, s = (a, s)$$

holds, for some $a \in D$. Therefore, we define the function

$$[\![\cdot]\!]^* : \mathcal{T}(V, D) \to (V \to D) \to D$$

by:

$$[\![r]\!]^*\,\sigma = \mathit{fst}([\![r]\!]\,(\mathbf{return} \circ \sigma)\,())\,.$$

In our application, the solver not only evaluates pure functions, i.e., strategy trees, but also records the variables accessed during this evaluation. In order to reason about the sequence of accessed variables together with their values, we *instrument* the evaluation of strategy trees by additionally taking a list of already visited variables together with their values and returning updated list for the rest computations. For the state transformer monad this instrumented evaluation is defined by:

$$\begin{aligned}
[\![\mathsf{Answ}(a)]\!]'\,\tau\, l &= \mathbf{return}\ (a, l)\,,\\
[\![\mathsf{Quest}(\mathbf{x}, c)]\!]'\,\tau\, l &= \mathbf{bind}\ (\tau\,\mathbf{x})\ (\mathbf{fun}\ a \to [\![c\ a]\!]'\,\tau\,(l \mathbin{@} [(\mathbf{x}, a)]))\,,
\end{aligned}$$

or, again instantiated for state transformer monads,

$$\begin{aligned}
[\![\mathsf{Answ}(a)]\!]'\,\tau\, l &= \mathbf{fun}\ s \to ((a, l), s)\,,\\
[\![\mathsf{Quest}(\mathbf{x}, c)]\!]'\,\tau\, l &= \mathbf{fun}\ s \to \mathbf{let}\ (a, s_1) = \tau\,\mathbf{x}\, s\ \mathbf{in}\ [\![c\ a]\!]'\,\tau\,(l \mathbin{@} [(\mathbf{x}, a)])\, s_1\,,
\end{aligned}$$

where $l : (V \times D)$ *list*.

Then for every strategy tree $r$, mapping $\tau : V \to \mathcal{M}D$ and list $l_1 : (V \times D)$ *list*

$$[\![r]\!]\,\tau\, s = (a, s') \qquad \text{iff} \quad [\![r]\!]'\,\tau\, l_1\, s = ((a, l_2), s')\,,$$

for some $a \in D$ and $l_2 : (V \times D)$ *list*. Moreover, if $\tau = \mathbf{return} \circ \sigma$ for some $\sigma : V \to D$, then for all states $s$

$$[\![r]\!]'\,\tau\,[\,]\, s = ((a, l), s)$$

holds, for some $a \in D$ and $l : (V \times D)$ *list*.

Now assume that we are given a mapping $t : V \to \mathcal{T}(V, D)$. Relative to this mapping and an assignment $\sigma : V \to D$ we define

$$\begin{aligned}
&\mathsf{trace}_\sigma\, r = l\,, \quad \text{where} \quad [\![r]\!]'\,(\mathbf{return} \circ \sigma)\,[\,]\,() = ((\_, l), \_),\ r \in \mathcal{T}(V, D)\,,\\
&\mathsf{dep}_{t,\sigma}\,\mathbf{x} = \{\mathbf{y} \mid (\mathbf{y}, \_) \in \mathsf{trace}_\sigma(t\,\mathbf{x})\}\,.
\end{aligned}$$

Moreover, we define $\mathsf{dep}_{t,\sigma}(X) = \bigcup_{\mathbf{x} \in X} \mathsf{dep}_{t,\sigma}\,\mathbf{x}$. Intuitively, the function $\mathsf{dep}_{t,\sigma}$ applied to a variable $\mathbf{x}$ and a variable assignment $\sigma$ returns a set of variables that $\mathbf{x}$ *directly depends on relative to* $\sigma$, i.e., a set of those variables which values are required to evaluate the strategy tree for the right-hand side of $\mathbf{x}$. The relation

$$\mathsf{Dep}_{t,\sigma} = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in \mathsf{dep}_{t,\sigma}\,\mathbf{x}\}$$

is also called a *dependence graph* for the variable assignment $\sigma$. Let $\mathsf{Dep}^+_{t,\sigma}$ be a transitive closure of the relation $\mathsf{Dep}_{t,\sigma}$ and $\mathsf{Dep}^*_{t,\sigma} = \mathsf{Dep}^+_{t,\sigma} \cup \{(\mathbf{x}, \mathbf{x}) \mid \mathbf{x} \in V\}$ be a reflexive and transitive closure of $\mathsf{Dep}_{t,\sigma}$ and denote $\mathsf{dep}^*_{t,\sigma}\,\mathbf{x} = \{\mathbf{y} \mid \mathsf{Dep}^*_{t,\sigma}(\mathbf{x}, \mathbf{y})\}$ and $\mathsf{dep}^*_{t,\sigma}(X) = \bigcup_{\mathbf{x} \in X} \mathsf{dep}^*_{t,\sigma}\,\mathbf{x}$.

### 3.2 Solutions

**Definition 2.** *Let $\mathcal{S} = (V, f)$ be a constraint system over the lattice $\mathbb{D}$ and $X \subseteq V$. We say that a variable assignment $\sigma : V \to D$ is a* solution *of the constraint system $\mathcal{S}$, if for every $\mathbf{x} \in V$, $\sigma\,\mathbf{x} \sqsupseteq d$ whenever $(d, ()) = f_{\mathbf{x}}(\mathbf{return} \circ \sigma)\,()$ holds. For the latter statement, we also write $\sigma\,\mathbf{x} \sqsupseteq f_{\mathbf{x}}\,\sigma$.*

**Definition 3.** *A partial function*

$$\mathcal{A} : (V \to \mathcal{T}(V, D)) \times 2^V \to (V \to D) \times 2^V$$

*is (the denotational semantics of) a* local solver *if it takes as input a pair $(t, X)$ of a strategy function $t$ and a set $X \subseteq V$ of interesting variables and, whenever it terminates, returns a pair $(\sigma, X')$ consisting of a variable assignment $\sigma : V \to D$ together with a set $X' \subseteq V$ such that the following holds:*

1. *$X \subseteq X'$ and $\mathsf{dep}^*_{t,\sigma}(X') \subseteq X'$;*
2. *$\sigma\,\mathbf{x} \sqsupseteq [\![t\ \mathbf{x}]\!]^*\,\sigma$ holds for every $\mathbf{x} \in X'$.*

*In particular, this means that $\sigma$ restricted to $X'$ is a solution of the constraint system $(X', f\mid_{X'})$.*

## 4 Functional implementation with explicit state passing

In the functional implementation of algorithm **RLD**, the global state is made explicit, and passed into function calls by means of a separate parameter. Accordingly, the modified state together with the computed value (if there is any) are jointly returned. The type of a state is

$$\textbf{type}\ \mathsf{state} = 2^V \times (V \rightharpoonup D) \times (V \rightharpoonup V\ \mathit{list})\,.$$

The three components correspond to the set *stable*, the finite (partial) map $\sigma$, and the finite (partial) map *infl*, respectively.

To facilitate the handling of the state we introduce the following auxiliary functions:

- The function `get` : $\mathsf{state} \to V \to D$ implements the function $\sigma_\bot$;
- The function `set` : $V \to D \to \mathsf{state} \to \mathsf{state}$ when applied to $\mathbf{x}$ updates the current value of $\sigma\,\mathbf{x}$;
- The function `get_stable` : $\mathsf{state} \to 2^V$ extracts the set *stable* from the current state;
- The function `is_stable` : $V \to \mathsf{state} \to \mathbf{bool}$ checks whether a given variable $\mathbf{x}$ is in the set *stable*;
- The function `add_stable` : $V \to \mathsf{state} \to \mathsf{state}$ adds a given variable to the set *stable*;
- The function `rem_stable` : $V \to \mathsf{state} \to \mathsf{state}$ removes a given variable from the set *stable*;
- The function `get_infl` : $V \to \mathsf{state} \to V\ \mathit{list}$ implements the function $\mathit{infl}_{[\,]}$;

```
let rec eval x y = fun $s \to$
    let $s$ = solve y $s$ in
    let $s$ = add_infl y x in
    (get y $s, s$)
and eval_rhs x = fun $s \to$
    $[\![t\,\mathbf{x}]\!]$ (eval x) $s$

and solve x = fun $s \to$
    if is_stable x $s$ then $s$
    else
        let $s$ = add_stable x $s$ in
        let ($new\_val$, $s$) = eval_rhs x $s$ in
        let $cur\_val$ = get $s$ x in
        let $new\_val = cur\_val \sqcup new\_val$ in
        if $new\_val \sqsubseteq cur\_val$ then $s$
        else
            let $s$ = set x $new\_val$ $s$ in
            let ($work, s$) = extract_work x $s$ in
            solve_all $work$ $s$
and solve_all $work$ = fun $s \to$
    match $work$ with
    | [] $\to s$
    | x :: xs $\to$ solve_all xs (solve x $s$) in
let s_init = $(\emptyset, \emptyset, \emptyset)$ in
let $s$ = solve_all $X$ s_init in
(get $s$, get_stable $s$)
```

**Fig. 2.** Functional implementation of **RLD**

- The function add_infl : $V \to V \to$ state $\to$ state applied to variables $\mathbf{x}$ and $\mathbf{y}$ adds the pair $(\mathbf{y}, \mathbf{x})$ to *infl*;
- The function rem_infl : $V \to$ state $\to$ state applied to the variable $\mathbf{x}$ sets the list $infl_{[]}$ $\mathbf{x}$ in the current state to [].

The auxiliary function extract_work : $V \to$ state $\to$ ($V$ *list* $\times$ state) applied to a variable $\mathbf{x}$ determines the list $w$ of variables immediately influenced by $\mathbf{x}$, resets *infl* $\mathbf{x}$ to [], and subtracts $w$ from the set *stable* as follows:

```
let extract_work x = fun $s \to$
    let $w$ = get_infl x $s$ in
    let $s$ = rem_infl x $s$ in
    let $s$ = fold_left (fun $s\ y \to$ rem_stable $y\ s$) $s\ w$ in
    $(w, s)$
```

Using the auxiliary functions $[\![\cdot]\!]$ for strategy trees, the mutually recursive functions eval, eval_rhs, solve and solve_all of the algorithm are then given in Fig. 2.

Given a list of interesting variables $X \subseteq V$ the algorithm calls the function solve_all from the initial state s_init $= (\emptyset, \emptyset, \emptyset)$.

From now on, **RLD** refers to this functional implementation. We prove:

**Theorem 4.** *The algorithm* **RLD** *is a local generic solver.*

## 5 Proof of Theorem 4

The proof consists of four main steps:

1. We instrument the functional program, introducing auxiliary data structures — ghost variables.
2. We implement the instrumented program in Coq.
3. We provide invariants for the instrumented program.
4. We prove these invariants jointly by induction on number of recursive calls.

### 5.1 Instrumentation

In order to express the invariants necessary to prove the correctness of the algorithm, we introduce additional components into the state which do not affect the operational behavior of the algorithm but record auxiliary information. The auxiliary data structures appear in the program as *ghost variables*, i.e., variables which are not allowed to appear in case distinctions and may not be written into ordinary structures. Thus, they do not influence the "control flow" of the program. We distinguish:

- the set *called* of variables which are currently processed;
- the set *queued* of variables which have been *destabilized*, i.e., removed from the set *stable* by the algorithm and not yet been reevaluated.

Accordingly, the type `state` in the instrumented program is given by:

$$\textbf{type}\ \mathsf{state} = 2^V \times 2^V \times (V \rightharpoonup D) \times (V \rightharpoonup V\ \mathit{list}) \times 2^V\ .$$

The five components correspond to the sets *stable* and *called*, the finite (partial) map $\sigma$, the finite (partial) map *infl*, and the set *queued*, respectively.

Also, we require the following auxiliary functions:

- The function `add_called` : $V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ adds a given variable to the set *called*;
- The function `rem_called` : $V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ removes a given variable from the set *called*;
- The function `add_queued` : $V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ adds a given variable to the set *queued*;
- The function `rem_queued` : $V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ removes a given variable from the set *queued*.

```
(*...*)
and eval_rhs x = fun s →
    $[\![t\,\mathbf{x}]\!]'$ (eval x) [ ] s

and solve x = fun s →
    if is_stable x s then s
    else
        let s = rem_queued x s in
        let s = add_stable x s in
        let s = add_called x s in
        let ((new_val, _), s) = eval_rhs x s in
        let s = rem_called x s in
        let cur_val = get s x in
        let new_val = cur_val $\sqcup$ new_val in
        if new_val $\sqsubseteq$ cur_val then s
        else
            let s = set x new_val s in
            let (work, s) = extract_work x s in
            solve_all work s
```

**Fig. 3.** Instrumented implementation of the functions `eval_rhs` and `solve`

In the instrumented implementation, we also replace the evaluation $[\![\cdot]\!]$ for strategy trees with $[\![\cdot]\!]'$ which additionally returns the list of accessed variables together with their respective values. Also, the function `extract_work` for a given $\mathbf{x}$ additionally removes the list $w$ of variables influenced by $\mathbf{x}$ from the set *called* and adds it to the set *queued* of the current state.

The instrumented functions `eval_rhs` and `solve` are given in Fig. 3. The functions `eval` and `solve_all` remain unchanged.

It is intuitively clear that the instrumentation does not alter the relevant behavior of the algorithm and that therefore the subsequent verification of the instrumented version also establishes the correctness of the original one. We now sketch two ways for making this rigorous; neither of them is part of the formal verification, though, which operates entirely on the instrumented version. For the rest of this section let us used primed notation, e.g. $\mathsf{state}'$, $\mathtt{solve}'$ etc. for the instrumented versions, leaving the unprimed ones for the original version.

We can define a simulation relation $\sim\subseteq \mathsf{state}\times\mathsf{state}'$ as the graph of the projection from $\mathsf{state}'$ to $\mathsf{state}$. We define a lifted relation $\mathcal{M}(\sim)\subseteq\mathcal{M}X\times\mathcal{M}'X$ for any $X$ by

$$f\ \mathcal{M}(\sim)\ f' \equiv \forall s, s', s_1, s_1', x, x'.\, f(s)=(x,s_1)\wedge f'(s')=(x',s_1')\wedge \\ s\sim s' \implies s_1\sim s_1'\wedge x=x'\,.$$

Two functions $f: X\to\mathcal{M}Y$ and $f': X\to\mathcal{M}'Y$ are related if $f(x)\ \mathcal{M}(\sim)\ f'(x)$ holds for all $x\in X$. It is then a straightforward consequence from the definitions that each component of the algorithm is related to its primed (instrumented) version and thus that they yield equal results when started in related states and after discarding the instrumentation.

Alternatively, we can modify the verification of the instrumented version to yield a direct verification of the original version by existentially quantifying the instrumentation components in all invariants. When showing that such existentially quantified invariants are indeed preserved, one opens the existentials in the assumption yielding a fixed but arbitrary instrumentation of the starting state; one then updates this instrumentation using the above updating functions `rem_queued`, `add_stable` etc. and uses the resulting instrumentation as existential witness for the conclusion. The remaining proof obligation then follows step by step the verification of the instrumented version. See [9] for a formal account of this proof-transforming procedure in the context of Hoare logic.

### 5.2 Implementation in Coq

Coq accepts the definition of a recursive function only if it is provably terminating. Since the algorithm **RLD** is generic, we neither make any assumptions concerning the lattice $\mathbb{D}$ (e.g., w.r.t. finiteness of ascending chains), nor assume finiteness of the set of variables $V$. Accordingly, termination of the algorithm cannot be guaranteed. Therefore, our formalization of the algorithm in Coq relies on the representation of partial functions through their function graphs. The mutual recursive definition of these *relations* exactly mimics the functional implementation of the algorithm.

We define the following relations (see appendix):

- for every $\mathbf{x}, \mathbf{y} \in V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $d \in D$, $\texttt{EvalRel}(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{s}', d)$ holds iff the call $\texttt{eval}\ \mathbf{x}\ \mathbf{y}\ \mathbf{s}$ terminates and returns the value $(d, \mathbf{s}')$;
- for every $\mathbf{x} \in V$, $t \in \mathcal{T}(D, T)$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $d \in D$, $l, l' : (V \times D)\,\mathit{list}$, $\texttt{Wrap\_Eval\_x}(\mathbf{x}, t, \mathbf{s}, \mathbf{s}', d, l, l')$ holds iff the call $[\![t]\!]'\,(\texttt{eval}\ \mathbf{x})\ l\ \mathbf{s}$ terminates and returns the value $((d, l'), \mathbf{s}')$;
- for every $\mathbf{x} \in V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $d \in D$, $l' : (V \times D)\,\mathit{list}$, $\texttt{Eval\_rhs}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l')$ holds iff the call $\texttt{eval\_rhs}\ \mathbf{x}\ \mathbf{s}$ terminates and returns the value $((d, l'), \mathbf{s}')$;
- for every $\mathbf{x} \in V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $\texttt{Solve}(\mathbf{x}, \mathbf{s}, \mathbf{s}')$ holds iff the call $\texttt{solve}\ \mathbf{x}\ \mathbf{s}$ terminates and returns the value $\mathbf{s}'$;
- for every $\mathit{work} \subseteq V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $\texttt{SolveAll}(\mathit{work}, \mathbf{s}, \mathbf{s}')$ holds iff the call $\texttt{solve\_all}\ \mathit{work}\,\mathbf{s}$ terminates and returns the value $\mathbf{s}'$.

The defined predicates relate states before the call and after termination of the corresponding functions. Therefore, they can be used to reason about properties of the algorithm, even if its termination is not guaranteed.

### 5.3 Invariants

Given a variable assignment $\sigma$ we inductively define relation $\mathsf{valid} \subseteq (V \times D)\ \mathit{list} \times (V \to D)$ as follows:

- $\mathsf{valid}([\,], \sigma)$;

- for any $\mathbf{x} \in V$, $d \in D$ and $l : (V \times D)\,\mathit{list}$, if $\mathsf{valid}(l, \sigma)$ and $d = \sigma\,\mathbf{x}$ then $\mathsf{valid}((\mathbf{x}, d){::}l, \sigma)$;

and relation $\mathsf{legal} \subseteq (V \times D)\,\mathit{list} \times \mathcal{T}(V, D)$ inductively by:

- $\mathsf{legal}([\,], r)$ for any $r \in \mathcal{T}(V, D)$;
- for any $\mathbf{x} \in V$, $d \in D$, $l : (V \times D)\,\mathit{list}$ and $c : D \to \mathcal{T}(V, D)$, if $\mathsf{legal}(l, c(d))$ then $\mathsf{legal}((\mathbf{x}, d){::}l, \mathsf{Quest}(\mathbf{x}, c))$.

Intuitively, $\mathsf{valid}(l, \sigma)$ holds iff the path $l$ agrees with the variable assignment $\sigma$, and $\mathsf{legal}(l, r)$ means that one can walk along the path $l$ in the tree $r$, for every $(\mathbf{x}, d)$ from $l$ using a value $d$ as an argument of a corresponding continuation function. For example, one can show by induction that $\mathsf{trace}_\sigma\, r$ is valid for $\sigma$ and is legal in $r$, i.e., $\mathsf{valid}(\mathsf{trace}_\sigma\, r, \sigma)$ and $\mathsf{legal}(\mathsf{trace}_\sigma\, r, r)$ hold for any $r \in \mathcal{T}(V, D)$ and given variable assignment $\sigma$.

Given a strategy tree $r$ and a path $l$ legal in $r$ we can define a function $\mathsf{subtree}(l, r)$ recursively as follows:

- if $l = [\,]$ then $\mathsf{subtree}(l, r) = r$;
- if $l = (\mathbf{x}, d){::}vs$ and $r = \mathsf{Quest}(\mathbf{x}, c)$ then $\mathsf{subtree}(l, r) = \mathsf{subtree}(vs, c(d))$.

We have that $\mathsf{subtree}(\mathsf{trace}_\sigma\, r, r) = \mathsf{Answ}(a)$ holds for every tree $r \in \mathcal{T}(V, D)$ and variable assignment $\sigma$.

We prove by induction on length of a path the following lemma.

**Lemma 5.** *For any given $r \in \mathcal{T}(V, D)$, a path $l : (V \times D)\,\mathit{list}$ and a variable assignment $\sigma : V \to D$, the following is equivalent:*

- *$l = \mathsf{trace}_\sigma\, r$;*
- *$\mathsf{valid}(l, \sigma)$, $\mathsf{legal}(l, r)$, $\mathsf{subtree}(l, r) = \mathsf{Answ}(a)$, for some $a \in D$, hold.* □

From now on, for simplicity, we denote `get_infl` as $\mathit{infl}_{[\,]}$ and `get` as $\sigma_\perp$. States $\mathbf{s}$ and $\mathbf{s}'$ denote a state before a call of some function and a state after the call terminates, respectively. Structures *stable*, *called*, *queued* and *infl* are components of the state $\mathbf{s}$, primed structures are components of the state $\mathbf{s}'$. Let $t : V \to \mathcal{T}(V, D)$ be a given strategy function. We denote a tree $t\,\mathbf{x}$ by $t_\mathbf{x}$. We say that variable $\mathbf{x}$ is *solved* in the state $\mathbf{s}$ if $\mathbf{x} \in \mathit{stable} \setminus \mathit{called}$. We treat lists as sets in the formulae below.

We define:

$$\begin{aligned} \mathcal{I}_0(\mathbf{s}) &\equiv \mathit{called} \subseteq \mathit{stable} \wedge \mathit{queued} \cap \mathit{stable} = \emptyset\,, \\ \mathcal{I}_1(\mathbf{s}, \mathbf{s}') &\equiv \mathit{stable}' \supseteq \mathit{stable} \wedge \mathit{called}' \subseteq \mathit{called} \wedge \mathit{queued}' \subseteq \mathit{queued}\,. \end{aligned}$$

We call a state $\mathbf{s}$ (a transition from $\mathbf{s}$ to $\mathbf{s}'$) *consistent* if $\mathcal{I}_0(\mathbf{s})$ (respectively, $\mathcal{I}_1(\mathbf{s}, \mathbf{s}')$) holds. The formula

$$\mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \equiv \forall \mathbf{z} \in V.\ \sigma_\perp\, \mathbf{s}'\, \mathbf{z} \sqsupseteq \sigma_\perp\, \mathbf{s}\, \mathbf{z}$$

expresses that the variable assignment in the state $\mathbf{s}'$ returns larger values than that in the state $\mathbf{s}$. The formula

$$\begin{aligned}\mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s},\mathbf{s}') \equiv \forall \mathbf{z} \in V.\, &(\sigma_\perp\,\mathbf{s}'\,\mathbf{z} \sqsubseteq \sigma_\perp\,\mathbf{s}\,\mathbf{z} \implies \mathit{infl}_{[\,]}\,\mathbf{z}\,\mathbf{s} \subseteq \mathit{infl}_{[\,]}\,\mathbf{z}\,\mathbf{s}') \land \\ &(\sigma_\perp\,\mathbf{s}'\,\mathbf{z} \not\sqsubseteq \sigma_\perp\,\mathbf{s}\,\mathbf{z} \implies \mathit{infl}_{[\,]}\,\mathbf{z}\,\mathbf{s} \subseteq \mathit{stable}' \setminus \mathit{called}')\end{aligned}$$

relates structures $\sigma$ and *infl*. It expresses for every variable $\mathbf{z}$ the following. If the value of $\mathbf{z}$ did not increase, then *infl*′ contains more dependencies; otherwise, all the variables influenced by $\mathbf{z}$ in $\mathbf{s}$ are solved in $\mathbf{s}'$. The formula

$$\mathcal{I}_{\mathrm{dep}}(\mathbf{x},\mathbf{s}) \equiv \forall \mathbf{z} \in \mathsf{dep}_{t,(\sigma_\perp\,\mathbf{s})}\,\mathbf{x}.\, \mathbf{z} \in \mathit{stable} \cup \mathit{queued} \land \mathbf{x} \in \mathit{infl}_{[\,]}\,\mathbf{z}\,\mathbf{s}\,.$$

expresses that for every variable $\mathbf{z}$ influencing $\mathbf{x}$, this dependency is stored in the state $\mathbf{s}$. The formula

$$\mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \equiv \forall \mathbf{x} \in \mathit{stable} \setminus \mathit{called}.\, \sigma_\perp\,\mathbf{s}\,\mathbf{x} \sqsupseteq [\![ t_{\mathbf{x}} ]\!]^*(\sigma_\perp\,\mathbf{s}) \land \mathcal{I}_{\mathrm{dep}}(\mathbf{x},\mathbf{s})$$

defines the *correctness* of the state $\mathbf{s}$. This means that for every variable $\mathbf{x}$ which is solved in $\mathbf{s}$, the constraint $\sigma\,\mathbf{x} \sqsupseteq f_{\mathbf{x}}\,\sigma$ is satisfied for $\mathbf{x}$ and dependencies of $\mathbf{x}$ are treated correctly. The most difficult part of the proof was to determine invariants for the main functions of the algorithm which are sufficiently strong to prove its correctness. The most complicated invariant refers to the function $[\![\cdot]\!]'(\mathtt{eval}\,\mathbf{x})$. The formula

$$\begin{aligned}&\mathcal{I}_{[\![\cdot]\!]'(\mathtt{eval}\,\mathbf{x})}(\mathbf{x}, r, \mathbf{s}, \mathbf{s}', d, \mathit{vlist}, \mathit{vlist}') \equiv \\ &\quad \mathbf{x} \in \mathit{stable} \land \mathcal{I}_0(\mathbf{s}) \land \mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \land (\forall (\mathbf{y}, v) \in \mathit{vlist}.\, \mathbf{y} \in \mathit{stable}) \implies \\ &\quad \mathcal{I}_0(\mathbf{s}') \land \mathcal{I}_1(\mathbf{s},\mathbf{s}') \land \mathit{vlist} \subseteq \mathit{vlist}' \land (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{y} \in \mathit{stable}) \land \\ &\quad \mathcal{I}_\sigma(\mathbf{s},\mathbf{s}') \land \mathcal{I}_{\sigma,\mathrm{infl}}(\mathbf{s},\mathbf{s}') \land \mathcal{I}_{\mathrm{corr}}(\mathbf{s}') \land \\ &\quad \big[\mathbf{x} \in \mathit{called} \land (\forall (\mathbf{y}, v) \in \mathit{vlist}.\, \mathbf{x} \in \mathit{infl}_{[\,]}\,\mathbf{y}\,\mathbf{s}) \land \\ &\quad \mathsf{valid}(\mathit{vlist}, \sigma_\perp\,\mathbf{s}) \land \mathsf{legal}(\mathit{vlist}, t_{\mathbf{x}}) \land \mathsf{subtree}(\mathit{vlist}, t_{\mathbf{x}}) = r \implies \\ &\qquad \big(\mathbf{x} \in \mathit{called}' \implies \\ &\qquad\quad \mathsf{valid}(\mathit{vlist}', \sigma_\perp\,\mathbf{s}') \land \mathsf{legal}(\mathit{vlist}', t_{\mathbf{x}}) \land \mathsf{subtree}(\mathit{vlist}', t_{\mathbf{x}}) = \mathsf{Answ}(d) \land \\ &\qquad\quad (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{x} \in \mathit{infl}_{[\,]}\,\mathbf{y}\,\mathbf{s}') \land \mathcal{I}_{\mathrm{dep}}(\mathbf{x},\mathbf{s}')\big) \land \\ &\qquad \big(\mathbf{x} \notin \mathit{called}' \implies \mathbf{x} \in \mathit{stable}' \setminus \mathit{called}'\big)\big]\end{aligned}$$

relates the arguments *vlist* and $\mathbf{s}$ with the result value $((d, \mathit{vlist}'), \mathbf{s}')$ of the call whenever it terminates. It proceeds recursively on the tree $r$, taking as a parameter a list *vlist* of already visited variables together with their new values. The function $[\![\cdot]\!]'\,(\mathtt{eval}\,\mathbf{x})$ is called for a stable variable $\mathbf{x}$ and applied to a partial path *vlist* of stable variables and an initial consistent correct state $\mathbf{s}$. As a result it returns a value $d$ and a longer path *vlist*′, which extends *vlist*, of stable visited variables, together with a consistent correct state $\mathbf{s}'$. The formula states that values $\sigma\,\mathbf{x}$ of all variables $\mathbf{x}$ grew, and *infl* changes according to changes in $\sigma$. It distinguishes the case where $\mathbf{x} \in \mathit{called}$. Then if *vlist* is a valid and legal path in

$t_{\mathbf{x}}$ leading to the subtree $r$ and if $\mathbf{x} \in \mathit{called}'$ then the result path $\mathit{vlist}'$ is again valid and legal in $t_{\mathbf{x}}$ and leads to an answer $d$ and all the dependencies of $\mathbf{x}$ are recorded. Note that by lemma 5 this implies that $\mathit{vlist}'$ is a trace in $t_{\mathbf{x}}$ by $\sigma'$. If $\mathbf{x} \in \mathit{called}$ and $\mathbf{x} \notin \mathit{called}'$ then it was reevaluated and solved during a recursive call for some variable $\mathbf{y}$ of $r$. It does not matter which value $d$ is returned in this case since $\mathbf{x}$ is solved in $\mathbf{s}'$ and the corresponding constraint is satisfied and will be preserved after the sequent update of $\sigma\,\mathbf{x}$. In the case $\mathbf{x} \notin \mathit{called}$ we can deduce that $\mathbf{x}$ is solved in $\mathbf{s}'$ using $\mathcal{I}_1(\mathbf{s}, \mathbf{s}')$. The formula

$$\begin{aligned}
&\mathcal{I}_{\texttt{eval\_rhs}}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l') \equiv \\
&\qquad \mathbf{x} \in \mathit{called} \wedge \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \implies \\
&\qquad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}') \wedge \\
&\qquad \big[\mathbf{x} \in \mathit{called}' \implies d = [\![t_{\mathbf{x}}]\!]^*(\sigma_\perp\, \mathbf{s}') \wedge l' = \mathsf{trace}_\sigma\, t_{\mathbf{x}} \wedge \\
&\qquad\quad (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{x} \in \mathit{infl}_{[\,]}\, \mathbf{y}\ \mathbf{s}') \wedge \mathcal{I}_{\mathrm{dep}}(\mathbf{x}, \mathbf{s}')\big]
\end{aligned}$$

relates the arguments $\mathbf{x}$ and $\mathbf{s}$ of the call of $\texttt{eval\_rhs}\ \mathbf{x}\ \mathbf{s}$ with the result state $\mathbf{s}'$ whenever it terminates. If the input state $\mathbf{s}$ is consistent and correct then so is the state $\mathbf{s}'$. In the case when $\mathbf{x}$ stays called we have that $d$ is a value of the right-hand side of $\mathbf{x}$ on $\sigma'$ and $l'$ is a trace in $t_{\mathbf{x}}$ by $\sigma'$. In the case $\mathbf{x} \notin \mathit{called}'$ the variable $\mathbf{x}$ is processed during some intermediate recursive call and is solved in $\mathbf{s}'$. The formula

$$\begin{aligned}
&\mathcal{I}_{\texttt{solve}}(\mathbf{x}, \mathbf{s}, \mathbf{s}') \equiv \\
&\qquad \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \implies \\
&\qquad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}') \wedge \\
&\qquad \big[\mathbf{x} \in \mathit{stable} \implies \mathbf{s} = \mathbf{s}'\big] \wedge \\
&\qquad \big[\mathbf{x} \notin \mathit{stable} \implies \mathit{stable}' \supseteq \mathit{stable} \cup \{\mathbf{x}\} \wedge \mathit{queued}' \subseteq \mathit{queued} \setminus \{\mathbf{x}\}\big]
\end{aligned}$$

relates arguments $\mathbf{x}$ and $\mathbf{s}$ with the result state $\mathbf{s}'$ of the call of $\texttt{solve}\ \mathbf{x}\ \mathbf{s}$ whenever it terminates. If the state $\mathbf{s}$ is consistent and correct then so is $\mathbf{s}'$. In the case $\mathbf{x} \in \mathit{stable}$ the state does not change. If $\mathbf{x} \notin \mathit{stable}$ then eventually $\mathbf{x}$ is solved in $\mathbf{s}'$ and is removed from the set *queued*. The formula

$$\begin{aligned}
&\mathcal{I}_{\texttt{solve\_all}}(w, \mathbf{s}, \mathbf{s}') \equiv \\
&\qquad \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \implies \\
&\qquad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}') \wedge \\
&\qquad (w \cup \mathit{stable} \setminus \mathit{called} \subseteq \mathit{stable}' \setminus \mathit{called}') \wedge (\mathit{queued}' \subseteq \mathit{queued} \setminus w)
\end{aligned}$$

relates the arguments $w$ and $\mathbf{s}$ with the result state $\mathbf{s}'$ of the call $\texttt{solve\_all}\ w\ \mathbf{s}$ whenever it terminates. It states that all the variables solved in $\mathbf{s}$ together with the variables from $w$ are solved in $\mathbf{s}'$ and none of the variables from $w$ is in $\mathit{queued}'$. We note that although $w = \mathit{infl}\,\mathbf{x}$ (for a corresponding $\mathbf{x}$) may contain invalid dependencies, i.e., variables not dependent on $\mathbf{x}$ on the current $\sigma$, $\mathcal{I}_{\mathrm{corr}}(\mathbf{s}')$ states that $\mathit{infl}\,\mathbf{x}$ is appropriately recomputed.

By induction on number of unfoldings of definitions we prove in Coq that the formulae $\mathcal{I}_{\texttt{eval}}$, $\mathcal{I}_{[\![\cdot]\!]'(\texttt{eval}\,\mathbf{x})}$, $\mathcal{I}_{\texttt{eval\_rhs}}$, $\mathcal{I}_{\texttt{solve}}$ and $\mathcal{I}_{\texttt{solve\_all}}$ are invariants of corresponding functions in the following sense.

**Theorem 6.** *For all states* $\mathbf{s}, \mathbf{s}' : \texttt{state}$ *the following is true:*

- *for every* $\mathbf{x}, \mathbf{y} \in V$*,* $d \in D$*,* $\textit{EvalRel}(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{s}', d)$ *implies* $\mathcal{I}_{\texttt{eval}}(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{s}', d)$*;*
- *for every* $\mathbf{x} \in V$*,* $r \in \mathcal{T}(D, T)$*,* $d \in D$*,* $l, l' : (V \times D)$ *list,* $\textit{Wrap\_Eval\_x}(\mathbf{x}, r, \mathbf{s}, \mathbf{s}', d, l, l')$ *implies* $\mathcal{I}_{[\![\cdot]\!]'(\texttt{eval}\,\mathbf{x})}(\mathbf{x}, r, \mathbf{s}, \mathbf{s}', d, l, l')$*;*
- *for every* $\mathbf{x} \in V$*,* $d \in D$*,* $l' : (V \times D)$ *list,* $\textit{Eval\_rhs}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l')$ *implies* $\mathcal{I}_{\texttt{eval\_rhs}}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l')$*;*
- *for every* $\mathbf{x} \in V$*,* $\textit{Solve}(\mathbf{x}, \mathbf{s}, \mathbf{s}')$ *implies* $\mathcal{I}_{\texttt{solve}}(\mathbf{x}, \mathbf{s}, \mathbf{s}')$*;*
- *for every* $w \in V$ *list,* $\textit{SolveAll}(w, \mathbf{s}, \mathbf{s}')$ *implies* $\mathcal{I}_{\texttt{solve\_all}}(w, \mathbf{s}, \mathbf{s}')$*.* □

### 5.4 Putting things together

Having verified the invariants, we now prove that theorem 4 holds, i.e., that **RLD** is a local solver. Let **s_init** be an initial state with $\textit{stable} = \textit{called} = \textit{queued} = \sigma = \textit{infl} = \emptyset$. Assume that **RLD** applied to $(t, X)$ terminates and let $\mathbf{s}'$ be the state returned by the call $\texttt{solve\_all}\, X\, \mathbf{s\_init}$. According to the definition 3, we have to show that:

1. $X \subseteq \textit{stable}'$ and $\mathsf{dep}^*_{t,(\sigma_\bot\, \mathbf{s}')}(\textit{stable}') \subseteq \textit{stable}'$;
2. $\sigma_\bot\, \mathbf{s}'\, \mathbf{x} \sqsupseteq [\![t_\mathbf{x}]\!]^*(\sigma_\bot\, \mathbf{s}')$ holds for every $\mathbf{x} \in \textit{stable}'$.

By theorem 6, implication $\mathcal{I}_{\text{solve\_all}}(X, \mathbf{s\_init}, \mathbf{s}')$ holds; and its premise is true, inasmuch as both $\mathcal{I}_0(\mathbf{s\_init})$ and $\mathcal{I}_{\text{corr}}(\mathbf{s\_init})$ hold. Therefore, we have $\mathcal{I}_1(\mathbf{s\_init}, \mathbf{s}')$, and hence $\textit{called}' = \textit{queued}' = \emptyset$. From $(X \cup \textit{stable} \setminus \textit{called} \subseteq \textit{stable}' \setminus \textit{called}')$ we conclude, that $X \subseteq \textit{stable}'$. From $\mathcal{I}_{\text{corr}}(\mathbf{s}')$ it follows, that $\forall \mathbf{x} \in \textit{stable}'.\ \sigma_\bot\, \mathbf{s}'\, \mathbf{x} \sqsupseteq [\![t_\mathbf{x}]\!]^*(\sigma_\bot\, \mathbf{s}')$ and $\mathsf{dep}_{t,(\sigma_\bot\, \mathbf{s}')}(\textit{stable}') \subseteq \textit{stable}'$ hold. Hence we have $\mathsf{dep}^*_{t,(\sigma_\bot\, \mathbf{s}')}(\textit{stable}') \subseteq \textit{stable}'$ and the statement of theorem 4 follows. □

## 6 Conclusion

We have presented the outline of a proof that the algorithm **RLD** is a local generic solver. By that, we enabled the inclusion of this algorithm into the trusted code base of a verified program analyzer. Since the solver can be applied to constraint systems where right hand sides of variables are arbitrary *pure* functions, this enables the design and implementation of flexible and general verified analyzer frameworks.

The extended version of this paper will provide further verified properties of the algorithm **RLD**, such as sufficient conditions for its termination as well as sufficient conditions for returning fragments not of any but of the least solution of the given constraint system. In practical applications such as the analyzer Goblint it is often convenient to allow more than one constraint for a variable. Therefore, it would be also interesting to provide formalized correctness proofs also for corresponding extension of **RLD**.

## References

1. Michael Backes and Peeter Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *ACM Conference on Computer and Communications Security*, pages 370–379, 2006. 1
2. David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Programming Languages and Systems, 13th European Symp. on Programming (ESOP)*, pages 385–400. Springer, LNCS 2986, 2004. 1
3. Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992. 1
4. Solange Coupet-Grimal and William Delobel. A uniform and certified approach for two static analyses. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2004. 1
5. Christian Fecht. Gena - a tool for generating prolog analyzers from specifications. In *2nd Static Analysis Symposium (SAS)*, pages 418–419. LNCS 983, 1995. 1
6. Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *European Symposium on Programming (ESOP)*, pages 90–104. LNCS 1381, Springer Verlag, 1998. Long version in *Northern Journal of Computing 5, 304-329,1998*. 1
7. Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999. 1
8. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. What is a pure functional? In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *ICALP (2)*, volume 6199 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2010. 1, 3
9. Martin Hofmann and Mariela Pavlova. Elimination of ghost variables in program logics. In Gilles Barthe and Cédric Fournet, editors, *Proc. Trustworthy Global Computing, LNCS 4912*, volume 4912 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007. 5.1
10. Niels Jorgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In *1st Static Analysis Symposium (SAS)*, pages 329–345. LNCS 864, Springer Verlag, 1994. 1
11. Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 3(298):583–626, 2003. 1
12. The Coq development team. *The Coq proof assistant reference manual*. TypiCal Project (formerly LogiCal), 2009. Version 8.2-bugfix. 1
13. Tobias Nipkow. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures, 4th Int. Conf. (FoSSaCS)*, pages 347–363. Springer, LNCS 2030, 2001. 1
14. Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In *Static Analysis, 16th Int. Symposium, (SAS)*, pages 171–187. LNCS 5673, Springer Verlag, 2009. 1
15. Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Übersetzerbau: Analyse und Transformation*. Springer Verlag, 2010. 1