TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

On the Verification of Local Generic Solvers

Martin Hofmann, Aleksandr Karbyshev and Helmut Seidl

TUM-I1323

Technischer Bericht
Technische Universität München
Institut für Informatik

On the Verification of Local Generic Solvers

MARTIN HOFMANN, Ludwig-Maximilians-Universität München
ALEKSANDR KARBYSHEV and HELMUT SEIDL, Technische Universität München

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 two local generic fixpoint solvers **RLD** and **RLDE**, which can be applied to constraint systems $\mathrm{x} \sqsupseteq f_{\mathrm{x}}$, $\mathrm{x} \in V$, over some (semi-)lattice $\mathbb{D}$ where the right-hand sides f_{x} are given as arbitrary functions implemented in some specification language. Verification of these algorithms is challenging, because they use higher-order functions and rely on side effects to track variable dependences as they are encountered dynamically during fixpoint iterations. Here, we present a correctness proof of these algorithms and show that **RLDE** is an *exact* solver while **RLD** is not. Proofs are formalized by means of the interactive proof assistant COQ.

This paper is an extended version of [Hofmann et al. 2010a].

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—Formal methods; D.2.4 [**Software Engineering**]: Software/Program Verification—Correctness proofs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—Invariants; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: fixpoint algorithm, static analysis, COQ

1. INTRODUCTION

A *generic* solver computes a solution of a constraint system $\mathrm{x} \sqsupseteq f_{\mathrm{x}}, \mathrm{x} \in V$, over some (semi-)lattice $\mathbb{D}$, where the right-hand side f_{x} of each variable 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 values for X of a solution of the constraint system by touching as few variables as possible.

Local generic solvers are a convenient tool for implementation of efficient frameworks for program analyses. They have first been proposed for the analysis of logic programs [Charlier and Hentenryck 1992; Fecht 1995; Fecht and Seidl 1998; 1999] and model-checking [Jorgensen 1994], but recently have also attracted attention in interprocedural analyzers of imperative programs [Backes and Laud 2006; Seidl and Vojdani 2009]. One particularly simple instance **RLD** of a local generic solver has been included into the textbook on *Program Analysis and Optimization* [Seidl et al. 2012], although without any proof of correctness of the algorithm. The proof of partial correctness of **RLD** was first presented in [Hofmann et al. 2010a].

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 dependencies beforehand. Therefore, efficient generic solvers rely on *self-observation* to detect and record variable dependencies on-the-fly during evaluation of right-hand sides. The local generic solver **TD** by van Hentenryck [Charlier and Hentenryck 1992] 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 in the applied software.

Providing a modular implementation of a generic fixpoint algorithm [Pottier 2009], Pottier poses as open problems

(1) whether a purely functional implementation of such an algorithm exists, and
(2) whether a formal verification of the algorithm is possible.

The first issue in proving any generic solver correct is which kind of functions may be safely applied as right-hand sides of constraints. In the companion paper [Hofmann et al. 2010b], we have presented a semantic property of *purity* for second-order functionals. 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 any evaluation of f on a variable assignment σ can be considered as a sequence of variable lookups followed by local computations and ending in an answer value.

Based on the notion of purity, we tackle and solve the problems posed by Pottier. Indeed, we show the local generic solver **RLD** admits a purely functional implementation and is correct. Related formal correctness proofs have been provided for variants of Kildall's algorithm for dataflow analysis [Klein and Nipkow 2003; Cachera et al. 2004; Coupet-Grimal and Delobel 2004]. However, this fixpoint algorithm is neither generic nor local. It also exploits variable dependencies which, nevertheless, are explicitly given through the control-flow graph.

In many applications constraint systems are not arbitrary but have good properties such as providing right-hand sides which are *monotonic*. In that case, the Knaster-Tarski theorem for complete lattices guarantees the existence of the unique *least* solution. A good fixpoint algorithm, therefore, when applied to such a system should return this least solution (or parts of it) whenever it terminates. Such kind of solvers we call *exact*.

The fixpoint algorithm **RLD**, however, may not return (parts of) the least solution, even if all right-hand sides of constraints are monotonic. The reason is that evaluations of right-hand sides are not executed atomically, implying that different evaluations may interfere, and thus values of variables may change during the evaluation. Therefore, if a variable is queried more than once, leaves in the strategy trees may become reachable, which could not have been reached by atomic evaluation. Accordingly, **RLD** is guaranteed to return parts of the least solution only for constraint systems where on every branch of a right-hand side every variable occurs at most once.

In order to deal with *arbitrary* monotonic constraint systems, we enhance the algorithm **RLD** to the algorithm **RLDE** which still allows interference of different evaluations but identifies potentially superfluous evaluations and ignores their contributions. Thus, algorithm **RLDE** is exact.

The paper is structured as follows. In section 2 we describe the fixpoint algorithm **RLD**. In section 3 we provide an example of a small, seemingly natural, but erroneous optimization of **RLD**. In sections 4, precise definitions of *constraint system* and *solver* are given. Further, in section 5 we give a purely functional implementation of **RLD** (in a pure ML-like language) with explicit state passing, which is proven to be a local solver in section 6. In section 7 we analyse behaviour of **RLD** when applied to monotonic constraint systems. We show that **RLD** is *not* an exact solver and therefore, in section 8, we present the enhanced version **RLDE**. In section 9 we sketch a proof of termination of **RLD** (**RLDE**) under certain conditions. All theorems and proofs (besides termination proof) are formalized by means of the interactive theorem prover COQ [The Coq Development Team 2012]. The source codes are available online on the second author's homepage `http://www2.in.tum.de/~karbyshev/`.

This paper is an extended version of [Hofmann et al. 2010a].

2. THE LOCAL GENERIC SOLVER RLD

The algorithm **RLD** performes on a constraint system $\mathrm{x} \sqsupseteq f_{\mathrm{x}}$, $\mathrm{x} \in V$, where V is a set of variables, defined over some *bounded join-semilattice* $\mathbb{D} = (D, \sqcup, \sqsubseteq, \bot)$ consisting of a carrier D equipped with the partial ordering $\sqsubseteq$ and the least upper bound operation $\sqcup$, and having the distinguished least element $\bot$. Generally, we do *not* require completeness or the ascending chain condition of $\mathbb{D}$.

One basic idea of the algorithm **RLD** is that, as soon as the value of variable y is requested during reevaluation of the right-hand side f_{x}, the algorithm does not naively return the current value for y. Instead, it first tries to get a better approximation of it, thus reducing the overall number of performed iterations. This idea is similar to that of the algorithm **TD** [Charlier and Hentenryck 1992].

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

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

(1) Finite map σ, storing current values (from D) of variables. We track only finite number of observed variables, since the overall size of set V can be extremely large. We define the auxiliary function

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

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

(2) Finite set $\mathit{stable} \subseteq V$. Intuitively, if variable x is marked as stable then either x is already *solved*, i.e., a computation for x has completed and σ gives a solution for x and all those variables x transitively depends on, or x is *called* and it is in the call stack of `solve` function and its value is being processed.

(3) Finite map *infl*, which stores dependencies between variables. More exactly, *infl* x returns an overapproximation of a set of variables y, for which evaluation of f_{y} on the current $\sigma_\bot$ depends on x. For the sake of efficiency, we implement *infl* as a (finite) mapping from V to lists of V. We define the auxiliary function

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

The structures have initial values $\sigma_{init} = \emptyset$, $\mathit{stable}_{init} = \emptyset$, $\mathit{infl}_{init} = \emptyset$.

The algorithm **RLD** proceeds as follows (see Fig. 1). The function `solve_all` is invoked for a list $X \subseteq V$ of interesting variables from the initial state. The function `solve_all` calls recursively `solve` x for every $\mathrm{x} \in X$ in turn.

The function `solve` when called for some variable x first checks whether x is already in the set *stable*. If so, the function returns; otherwise, the algorithm marks x as being stable and tries to satisfy a constraint $\sigma\,\mathrm{x} \sqsupseteq f_{\mathrm{x}}\,\sigma$. For that, it evaluates a value rhs of the right-hand side f_{x} by invoking `eval_rhs` x. After `eval_rhs` returns, `solve` calculates a least upper bound of rhs with cur, the current value of $\sigma\,\mathrm{x}$, and stores the produced value in new. Then it compares values of new and cur. If the current value is larger than a new one, the constraint for x is satisfied, and `solve` returns. Otherwise, the value of σ for x gets updated with new. Since the value of $\sigma\,\mathrm{x}$ has changed, all constraints of variables y dependent on x may not be satisfied anymore. Hence, the function `solve` *destabilizes* all the variables accumulated in $\mathit{work} = \mathit{infl}_{[\,]}\,\mathrm{x}$ by invok-

```
function eval(x : V, y : V) =
    solve(y); infl y ← x::infl y; σ⊥ y

function eval_rhs(x : V) =
    f_x(λy.eval(x, y))

function extract_work(x : V) =
    let work = infl_[] x in
    stable ← stable \ work; infl x ← []; work

function solve(x : V) =
    if x ∈ stable then ()
    else
        stable ← stable ∪ {x};
        let rhs = eval_rhs(x) in
        let cur = σ⊥ x in
        let new = cur ⊔ rhs in
        if new ⊑ cur then ()
        else
            σ x ← new;
            let work = extract_work(x) in
            solve_all(work)
        end
    end

function solve_all(work : 2^V) =
    foreach x ∈ work do solve(x)

begin
    σ = ∅; stable = ∅; infl = ∅;
    solve_all(X);
    (σ⊥, stable)
end
```

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

ing `extract_work`, i.e., those are subtracted from the set *stable*. Then *infl* x is reset to empty and `solve_all` *work* is recursively called. Note that for efficiency reasons the comparison $new \sqsubseteq cur$ can be replaced by an equality check. This may be beneficial if every element of the (semi-)lattice $\mathbb{D}$ has a unique representation and the equality operation is implemeted efficiently.

We mention that the right-hand side f_{x} is not evaluated directly on σ, but by using an auxiliary stateful function $\lambda y.$`eval`$(\mathbf{x}, y)$, which allows firstly to get better values for variables the variable x depends on. Thus, once `eval`$(\mathbf{x}, \mathbf{y})$ is invoked, it first calls `solve` y and then adds x to *infl* y. The latter reflects the fact that the value of x possibly depends on the value of y. Only after recording the variable dependence (x, y) the current value of 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_{x} are *pure* functions in the sense of [Hofmann et al. 2010b].

```
(*...*)
function extract_work(x : V) =
    let work = infl_[] x in
    stable ← stable \ (work\called); infl x ← []; work

function solve(x : V) =
    if x ∈ stable then ()
    else
        stable ← stable ∪ {x};
        called ← called ∪ {x};
        let rhs = eval_rhs(x) in
        called ← called \ {x};
        let cur = σ_⊥ x in
        let new = cur ⊔ rhs in
        if new ⊑ cur then ()
        else
            σ x ← new;
            let work = extract_work(x) in
            solve_all(work)
        end
    end

(*...*)
begin
    σ = ∅; stable = ∅; called = ∅; infl = ∅;
    solve_all(X);
    (σ_⊥, stable)
end
```

Fig. 2. The erroneous optimization of **RLD**

3. AN ERRONEOUS OPTIMIZATION

When starting to reason about the algorithm **RLD** from Figure 1, one might feel tempted to avoid to call `solve` for variables whose evaluation has already been triggered, but has not yet been completed.

We note that the meaning of the set *stable* is twofold. First, it contains all the variables x for which a call `solve` x already has terminated. Those variables are solved, i.e., the corresponding constraints are satisfied. Second, it contains variables being proccesed, for which reevaluation of right-hand sides is triggered but not yet finished, and corresponding constraints may not be satisfied. Therefore, it seems reasonable to distinguish this kind of *called* variables and prevent them from redundant destabilization since their recomputation is pending. This idea would lead to the following optimization.

We introduce the set of variables *called* (initially, $called = \emptyset$). It tracks a subset of stable variables currently being processed, i.e., $called \subseteq stable$ is invariant for every program point. We say that the variable x is *solved* if $\mathrm{x} \in stable \setminus called$. Variable x is added to *called* just before a reevaluation of f_{x} starts and is removed from it right after the reevaluation returns. The function `extract_work` does not destabilize variables from the set *called*, i.e., the algorithm does not trigger recomputation of variables from *infl* x that currently belong to *called* and keep them in the sets *stable* and *called* (see Fig.2).

This optimization appears to be wrong as shown by the counterexample which appears in Fig.3. The constraint system is defined over the three-element lattice $\mathbb{D} = (\{\bot, a, \top\}, \sqsubseteq, \sqcup)$ with $\bot \sqsubset a \sqsubset \top$. Figure 3 represents right-hand sides of the con-

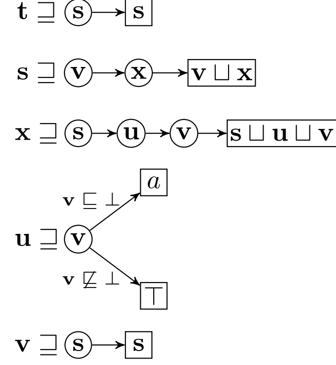

Fig. 3. Counterexample for erroneous optimization The constraint system is defined over the lattice $\mathbb{D} = (\{\bot, a, \top\}, \sqsubseteq, \sqcup)$ with $\bot \sqsubset a \sqsubset \top$.

straint system by means of strategy trees (precise definition of a strategy tree follows in Section 4). Here, round nodes denote queries to variables while rectangle boxes denote answers expressed in terms of constants and received values of queried variables. Conceptually, query nodes have one outgoing edge for every value of $\mathbb{D}$ corresponding to every possible received value for the variable. In the figure, however, we merge edges with equal subtrees for the sake of simplicity. In the example, these right-hand sides could be represented in a pure ML-like language by:

$$\begin{aligned} f_{\mathbf{s}} &= \textbf{fun } \sigma \to \textbf{let } v_1 = \sigma\,\mathbf{v} \textbf{ in let } x_1 = \sigma\,\mathbf{x} \textbf{ in } v_1 \sqcup x_1 \ , \\ f_{\mathbf{u}} &= \textbf{fun } \sigma \to \textbf{let } v_1 = \sigma\,\mathbf{v} \textbf{ in if } (v_1 \sqsubseteq \bot) \textbf{ then } a \textbf{ else } \top \ . \end{aligned}$$

Functions $f_{\mathbf{t}}$, $f_{\mathbf{x}}$ and $f_{\mathbf{v}}$ can be represented similarly.

We want to compute a local solution for the variable $\mathbf{t}$. Let us trace computations done by the solver from the initial state. We call `solve` $\mathbf{t}$, which in turn recursively calls `solve` $\mathbf{s}$. During the invocation of `solve` $\mathbf{s}$, the algorithm recursively computes new values of variables $\mathbf{v}$, then $\mathbf{x}$ and $\mathbf{u}$. For the sake of brevity, we skip a description of those steps, but note that they lead to change in $\sigma\,\mathbf{x}$. Before an altered value of $\mathbf{s}$ (which equals a) is returned, the algorithm recomputes all the variables dependent on $\mathbf{s}$ (these are variables from *infl* $\mathbf{s} = [\mathbf{x};\mathbf{v}]$), and resets *infl* $\mathbf{s}$ to $[\,]$.

1. `solve` $\mathbf{x}$ is called, and the variable $\mathbf{x}$ is added again to the sets *stable* and *called*. The state just prior to reevaluation of the right-hand side $f_{\mathbf{x}}$ (call of `eval_rhs` $\mathbf{x}$) is:

$$\begin{aligned} &\textit{stable} = \{\mathbf{s},\mathbf{t},\mathbf{u},\mathbf{x}\}, \quad \textit{called} = \{\mathbf{t},\mathbf{x}\}, \\ &\sigma(\mathbf{s}) = \sigma(\mathbf{u}) = \sigma(\mathbf{x}) = a, \quad \sigma(\mathbf{t}) = \sigma(\mathbf{v}) = \bot, \\ &\textit{infl}\ \mathbf{t} = \textit{infl}\ \mathbf{s} = [\,], \quad \textit{infl}\ \mathbf{u} = [\mathbf{x}], \quad \textit{infl}\ \mathbf{v} = [\mathbf{x};\mathbf{u};\mathbf{s}], \quad \textit{infl}\ \mathbf{x} = [\mathbf{s}]\,. \end{aligned}$$

During the invocation of `eval_rhs` $\mathbf{x}$ the algorithm traverses the tree representing $f_{\mathbf{x}}$ and tries to recompute variables $\mathbf{s}$, $\mathbf{u}$ and $\mathbf{v}$ in turn.

— Since $\mathbf{s}, \mathbf{u} \in \textit{stable}$, the algorithm does not descend into solving them. The structures σ, *stable* and *called* are not changed, but the solver records that $\mathbf{x}$ depends on $\mathbf{s}$ and $\mathbf{u}$, in particular, $\mathbf{x}$ is added to *infl* $\mathbf{s}$ and *infl* $\mathbf{u}$.

— The algorithm recomputes $\mathbf{v}$ (call to `solve` $\mathbf{v}$), which gets a large value a (as $\sigma(\mathbf{s}) = a$). Since $\sigma(\mathbf{v})$ has changed, variables influenced by $\mathbf{v}$ might need to be recomputed, these are variables from *infl* $\mathbf{v} = [\mathbf{x};\mathbf{u};\mathbf{s}]$. However, the algorithm does *not* destabilize $\mathbf{x}$, as $\mathbf{x} \in \textit{called}$. Thus, the variable $\mathbf{u}$ is destabilized and recomputed (since $\mathbf{u} \in \textit{stable} \setminus \textit{called}$), and gets a greater value $\top$. At this point, although *infl* $\mathbf{u} = [\mathbf{x};\mathbf{x}]$, the variable $\mathbf{x}$ is again *not* recomputed, since $\mathbf{x} \in \textit{called}$. Thus, the value of $\mathbf{x}$ remains as before, $\sigma(\mathbf{x}) = a$. Then $\mathbf{s}$ gets recomputed, but this does not lead to a change in the state. Finally, `solve` $\mathbf{s}$ returns.

2. `solve` $\mathbf{v}$ is called, but $\mathbf{v}$ is solved at this moment.

Finally, the algorithm returns the variable assignment

$$\sigma_1(\mathbf{t}) = \sigma_1(\mathbf{s}) = \sigma_1(\mathbf{x}) = \sigma_1(\mathbf{v}) = a, \quad \sigma_1(\mathbf{u}) = \top$$

that is not a solution, since the constraint for $\mathbf{x}$ is not satisfied:

$$\sigma_1(\mathbf{x}) = a \sqsubset \top = f_{\mathbf{x}}\,\sigma_1 \quad .$$

We conclude that this "natural" optimization is wrong. The counterexample was our guiding motivation for a rigorous verification of the fixpoint algorithm **RLD**.

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

Assume $\mathbb{D} = (D, \sqcup, \sqsubseteq, \bot)$ is a *bounded join-semilattice*. A pair (V, f) is a *constraint system*, where V is a set of variables and f is a functional of type

$$f : V \to (V \to TD) \to TD$$

that for every $\mathbf{x} \in V$ returns a corresponding *right-hand side*

$$f_{\mathbf{x}} : (V \to TD) \to TD,$$

where T is a monad constructor with associated monad operations $\mathsf{val}_T^X : X \to TX$ and $\mathsf{bind}_T^{X,Y} : TX \to (X \to TY) \to TY$. We will omit indices when it is clear from context which monad and sets are meant. Our application is formulated in terms of a *state monad* T_S, that is, given a set S of states, we define

$$\begin{aligned} T_S X &= S \to X \times S \;\;, \\ \mathsf{val}_{T_S}^X\, x &= \mathbf{fun}\; s \to (x, s) \;\;, \\ \mathsf{bind}_{T_S}^{X,Y}\, t\, g &= \mathbf{fun}\; s \to \mathbf{let}\; (x_1, s_1) = t\; s\; \mathbf{in}\; g\, x_1\, s_1 \;\;. \end{aligned}$$

The constraint function 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 σ. This property is *not* captured by polymorphism in a state alone [Hofmann et al. 2010b]. It is guaranteed, however, by the notion of *purity* introduced in [Hofmann et al. 2010b]. If the function $f_{\mathbf{x}}$ is pure in the sense of [Hofmann et al. 2010b], then $f_{\mathbf{x}}$ is representable by means of a *strategy tree*. This means that any 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.

4.1. Strategy trees

Definition 4.1. For a given set of values D and a set of variables V we define the set $\mathit{Tree}_{V,D}$ of *strategy trees* inductively by:

— for $a \in D$, $\mathsf{Ans}(a) \in \mathit{Tree}_{V,D}$;
— for $\mathbf{x} \in V$ and $c : D \to \mathit{Tree}_{V,D}$, $\mathsf{Que}(\mathbf{x}, c) \in \mathit{Tree}_{V,D}$.

Fix a monad T. We define the function

$$[\![\cdot]\!]_T : \mathit{Tree}_{V,D} \to (V \to TD) \to TD$$

recursively by:

$$\begin{aligned}
[\![\mathsf{Ans}(a)]\!]_T\, h &= \mathsf{val}_T\, a \ \ , \\
[\![\mathsf{Que}(\mathbf{x}, c)]\!]_T\, h &= \mathsf{bind}_T (h\, \mathbf{x})\, (\mathbf{fun}\ a \to [\![c\ a]\!]_T\, h) \ \ .
\end{aligned}$$

For the state monad T_S, we have

$$\begin{aligned}
[\![\mathsf{Ans}(a)]\!]_S\, h &= \mathbf{fun}\ s \to (a, s) \ \ , \\
[\![\mathsf{Que}(\mathbf{x}, c)]\!]_S\, h &= \mathbf{fun}\ s \to a_1, s_1) = h\, \mathbf{x}\, s(\mathbf{in}([\![c(a_1]\!]_S\, h\, s_1 \ \ .
\end{aligned}$$

Evaluation of a strategy tree thus formalizes a stateful evaluation of the pure function represented by this tree. Moreover, if the argument h does not depend on state and has no effect on state, i.e., is of the form

$$h = \mathsf{val}_{T_S} \circ \sigma = \mathbf{fun}\ \mathbf{x} \to \mathsf{val}_{T_S}\ (\sigma\, \mathbf{x}), \qquad \text{for some } \sigma : V \to D,$$

then for all states s and trees $t \in \mathit{Tree}_{V,D}$

$$[\![t]\!]\, h\, s = (a, s), \quad a \in D \ \ .$$

Therefore, we define the function

$$[\![\cdot]\!]^\star : \mathit{Tree}_{V,D} \to (V \to D) \to D$$

by

$$[\![t]\!]^\star\, \sigma = \mathit{fst}([\![t]\!]_{\mathit{unit}}\, (\mathsf{val}_{T_{\mathit{unit}}} \circ \sigma)\, (\star)), \quad \text{where } \mathit{unit} = \{\star\}.$$

In our application, the solver not only evaluates pure functions, i.e., strategy trees, but also records the variables accessed during the 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. That is, for a monad T,

$$[\![\cdot]\!]'_T : \mathit{Tree}_{V,D} \to (V \times D)\, \mathbf{\mathit{list}} \to (V \to TD) \to T(D \times (V \times D)\, \mathbf{\mathit{list}})$$

is defined by

$$\begin{aligned}
[\![\mathsf{Ans}(a)]\!]'_T\, l\, h &= \mathsf{val}_T\ (a, l) \ \ , \\
[\![\mathsf{Que}(\mathbf{x}, c)]\!]'_T\, l\, h &= \mathsf{bind}_T\ (h\, \mathbf{x})\ (\mathbf{fun}\ a \to [\![c\ a]\!]'_T\, (l\, @\, [(\mathbf{x}, a)])\, h) \ \ ,
\end{aligned}$$

and, again instantiated for a state monad T_S,

$$\begin{aligned}
[\![\mathsf{Ans}(a)]\!]'_S\, l\, h &= \mathbf{fun}\ s \to ((a, l), s) \ \ , \\
[\![\mathsf{Que}(\mathbf{x}, c)]\!]'_S\, l\, h &= \mathbf{fun}\ s \to \mathbf{let}\ (a_1, s_1) = h\, \mathbf{x}\, s\ \mathbf{in}\ [\![c\ a_1]\!]'_S\, (l\, @\, [(\mathbf{x}, a_1)])\, h\, s_1 \ \ ,
\end{aligned}$$

where $@$ is a list append function. Then for every strategy tree $t \in \mathit{Tree}_{V,D}$, $h : V \to TD$, $l_1 \in (V \times D)\, \mathbf{\mathit{list}}$, $a \in D$

$$[\![t]\!]_S\, h\, s = (a, s') \quad \text{iff} \quad [\![t]\!]'_S\, l_1\, h\, s = ((a, l_2), s') \text{ for some } l_2 \ \ .$$

Thus, instrumentation does not influence the state. Moreover, if $h = \mathsf{val}_{T_S} \circ \sigma$ for some $\sigma : V \to D$ then for every state s we have

$$[\![t]\!]'_S\, [\,]\, h\, s = ((a, l), s) \qquad \text{for some } a \in D,\ l \in (V \times D)\, \mathbf{\mathit{list}} \ \ .$$

In this case, the list l represents a trace of variables together with their respective values that are accessed within evaluation of $[\![t]\!]_S\, h$ for any S, in particular, for $[\![t]\!]^\star\, \sigma$. Note that if, for some variable $\mathbf{x}$, $(\mathbf{x}, a_1)$ and $(\mathbf{x}, a_2)$ occur in l then $a_1 = a_2$ necessarily. Given $\bar{f} : V \to \mathit{Tree}_{V,D}$ and $\sigma : V \to D$, we define

$$\begin{aligned}
\mathsf{trace}_\sigma\, t &= l\,, \quad \text{where} \quad [\![t]\!]'_{\mathit{unit}}\, (\mathsf{val}_{T_{\mathit{unit}}} \circ \sigma)\, [\,]\, (\star) = ((_, l), _),\ t \in \mathit{Tree}_{V,D} \ \ , \\
\mathsf{dep}_{\bar{f},\sigma}\, \mathbf{x} &= \{\mathbf{y} \mid (\mathbf{y}, _) \in \mathsf{trace}_\sigma(\bar{f}_\mathbf{x})\} \ \ .
\end{aligned}$$

Intuitively, the function $\mathsf{dep}_{f,\sigma}$ applied to a variable x returns a set of variables that x *directly depends on relative to* σ, i.e., a set of those variables values of which are required to evaluate the strategy tree $\bar{f}_{\mathbf{x}}$. For $X \subseteq V$, we define $\mathsf{dep}_{\bar{f},\sigma}(X) = \bigcup_{\mathbf{x}\in X} \mathsf{dep}_{\bar{f},\sigma}\,\mathbf{x}$.

4.2. Solutions

Definition 4.2. Let $\mathcal{S} = (V, f)$ be a constraint system over a (semi-)lattice $\mathbb{D}$. We say that the 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 $(d, \star) = f_{\mathbf{x}}(\mathsf{val}_{T_{unit}} \circ \sigma)\,(\star)$ holds. For the latter, we also write $\sigma\,\mathbf{x} \sqsupseteq f_{\mathbf{x}}\,\sigma$.

Let $\mathcal{S} = (V, f)$ be a constraint system over a (semi-)lattice $\mathbb{D}$ with a *pure* state polymorphic $f : V \to \Lambda S.\,(V \to T_S D) \to T_S D$. As shown in [Hofmann et al. 2010b], we can construct a strategy function $\bar{f} : V \to \mathit{Tree}_{V,D}$ such that $f_{\mathbf{x}}\,h\,s = [\![\bar{f}_{\mathbf{x}}]\!]_S\,h\,s$ for all $\mathbf{x} \in V$, sets of states S, functions $h : V \to T_S D$, and $s \in S$.

Definition 4.3. Let $X \subseteq V$. We say that the pair (σ, X') is a *local solution* of $\mathcal{S}$ relative to X if

(1) $X \subseteq X'$ and $\mathsf{dep}_{\bar{f},\sigma}(X') \subseteq X'$;
(2) $\sigma\,\mathbf{x} \sqsupseteq [\![\bar{f}_{\mathbf{x}}]\!]^{\star}\,\sigma$ holds for every $\mathbf{x} \in X'$.

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

Note that if $\mathbb{D}$ has a greatest element $\top_D$, we can trivially extend any local solution (σ, X') to a global one by

$$\sigma_{X'}\,\mathbf{x} = \begin{cases} \sigma\,\mathbf{x} & \mathbf{x} \in X' \\ \top_D & \text{otherwise.} \end{cases}$$

Definition 4.4. The partial function

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

polymorphic in V and D is (a denotational semantics of) a *generic local solver* if given a constraint system $\mathcal{S} = (V, f)$ over some bounded join-semilattice $\mathbb{D} = (D, \sqsubseteq, \sqcup, \bot)$ with pure f, $\mathcal{A}$ when applied to a pair $(\bar{f}, X)$ of a strategy function $\bar{f}$ (for f) and a set $X \subseteq V$ of interesting variables returns a local solution (σ, X') of $\mathcal{S}$ relative to X, whenever it terminates.

We say that the function $f : (V \to D) \to D$ is *monotonic* if $\sigma_1 \sqsubseteq \sigma_2$ implies $f\,\sigma_1 \sqsubseteq f\,\sigma_2$. We say that the stateful function $f : \Lambda S.\,(V \to T_S D) \to T_S D$ is *monotonic* if it is monotonic for T_{unit}. Recall that by theorem of Knaster-Tarski for any complete lattice $\mathbb{D}$ and a constraint system $\mathcal{S} = (V, f)$ over $\mathbb{D}$ with monotonic f there exists the least solution $\mu : V \to D$ of $\mathcal{S}$, and $\mu\,\mathbf{x} = f_{\mathbf{x}}\,\mu$, for all $\mathbf{x} \in V$.

Definition 4.5. We say that the generic local solver $\mathcal{A}_{V,D}$ is *exact* if, for any constraint system (V, f) over a complete lattice $\mathbb{D}$ with pure and monotonic f, $\mathcal{A}$ when applied to a pair $(\bar{f}, X)$ of a strategy function $\bar{f}$ (for f) and a set $X \subseteq V$ of interesting variables — if it terminates — returns a local solution (σ, X') of $\mathcal{S}$ relative to X such that for the least solution μ of $\mathcal{S}$, $\mu\restriction_{X'} = \sigma\restriction_{X'}$ holds.

5. 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

```
let rec eval x y = fun s →
    let s = solve y s in
    let s = add_infl y x in
    (get y s, s)

and eval_rhs x = fun s →
    ⟦f̄_x⟧_state (eval x) s

and solve x = fun s →
    if is_stable x s then s
    else
        let s = add_stable x s in
        let (rhs, s) = eval_rhs x s in
        let cur = get s x in
        let new = cur ⊔ rhs in
        if new ⊑ cur then s
        else
            let s = set x new s in
            let (work, s) = extract_work x s in
            solve_all work s

and solve_all work = fun s →
    match work with
    | [] → s
    | x :: xs → solve_all xs (solve x s) in

let s_init = (∅, ∅, ∅) in
let s = solve_all X s_init in
(get s, get_stable s)
```

Fig. 4. Functional implementation of **RLD**

type of a state is

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

The three components correspond to the finite (partial) map σ, the finite (partial) map *infl*, and the set *stable* in imperative implementation, respectively. To facilitate the handling of the state we introduce the following auxiliary functions:

— `get` : $\mathsf{state} \to V \to D$ implements the function $\sigma_\perp$;
— `set` : $V \to D \to \mathsf{state} \to \mathsf{state}$ when applied to x and d updates the current value of $\sigma\,\mathrm{x}$ with d;
— `get_stable` : $\mathsf{state} \to 2^V$ extracts the component *stable* from a given state;
— `is_stable` : $V \to \mathsf{state} \to \textbf{bool}$ checks if a given variable x is in *stable*;
— `add_stable` : $V \to \mathsf{state} \to \mathsf{state}$ adds a given variable to *stable*;
— `rem_stable` : $V \to \mathsf{state} \to \mathsf{state}$ removes a given variable from *stable*;
— `get_infl` : $V \to \mathsf{state} \to V\,\mathit{list}$ implements the function $\mathit{infl}_{[\,]}$;
— `add_infl` : $V \to V \to \mathsf{state} \to \mathsf{state}$ applied to variables x and y adds the pair (y, x) to *infl*;
— `rem_infl` : $V \to \mathsf{state} \to \mathsf{state}$ applied to a variable x resets the list $\mathit{infl}_{[\,]}\,\mathrm{x}$ in a given state to $[\,]$.

The auxiliary function `extract_work` : $V \to \mathsf{state} \to (V\,\mathit{list} \times \mathsf{state})$ applied to a variable x determines the list w of variables immediately influenced by x, resets $\mathit{infl}\,\mathrm{x}$ to $[\,]$, and

subtracts w from the component *stable* of a given state as follows:

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

Using the evaluation function $[\![\cdot]\!]_{\mathsf{state}}$ for strategy trees specialized for the state monad T_{state}, the mutually recursive functions `eval`, `eval_rhs`, `solve` and `solve_all` of the algorithm are then given in Fig. 4. Provided 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 5.1. *The algorithm* **RLD** *is a local generic solver.*

6. PROOF OF THEOREM 5.1

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.

6.1. 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 being processed;
— the set *queued* of variables which have been *destabilized*, i.e., removed from the set *stable* by the algorithm, and have not yet been reevaluated by `solve`.

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

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

The five components correspond to the finite (partial) map σ, the finite (partial) map *infl*, and the sets *stable*, *called*, and *queued*, respectively. In what follows, we will refer to the components of state by these names. We introduce the following auxiliary functions:

— `add_called` : $V \to \mathsf{state} \to \mathsf{state}$ adds a given variable to *called*;
— `rem_called` : $V \to \mathsf{state} \to \mathsf{state}$ removes a given variable from *called*;
— `add_queued` : $V \to \mathsf{state} \to \mathsf{state}$ adds a given variable to *queued*;
— `rem_queued` : $V \to \mathsf{state} \to \mathsf{state}$ removes a given variable from *queued*.

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 `x` additionally removes all the variables influenced by `x` from the set *called* and adds them to the set *queued* in the current state. The instrumented functions `eval_rhs` and `solve` are given in Fig. 5. The functions `eval` and `solve_all` remain unchanged.

```
(*...*)
and eval_rhs x = fun s →
    ⟦f̄_x⟧'_state (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 ((rhs, _), s) = eval_rhs x s in
        let s = rem_called x s in
        let cur = get s x in
        let new = cur ⊔ rhs in
        if new ⊑ cur then s
        else
            let s = set x new s in
            let (work, s) = extract_work x s in
            solve_all work s
```

Fig. 5. Instrumented implementation of the functions `eval_rhs` and `solve`

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. For the rest of this section let us used primed notation, e.g., state', `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 state' to state. We define a lifted relation $T(\sim) \subseteq T_{\mathsf{state}}X \times T_{\mathsf{state}'}X$ for any X by

$$f\ T(\sim)\ f' \equiv \forall \mathrm{s}, \mathrm{s}', \mathrm{s}_1, \mathrm{s}_1', x, x'.\, f(\mathrm{s}) = (x, \mathrm{s}_1) \wedge f'(\mathrm{s}') = (x', \mathrm{s}_1') \wedge \\ (\mathrm{s} \sim \mathrm{s}' \implies \mathrm{s}_1 \sim \mathrm{s}_1' \wedge x = x')\ .$$

Functions $f : X \to T_{\mathsf{state}}Y$ and $f' : X \to T_{\mathsf{state}'}Y$ are related if $f(x)\ T(\sim)\ f'(x)$ holds for all $x \in X$. One can show then inductively 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 after discarding the instrumentation. We provide a formalization of this approach in COQ.

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 [Hofmann and Pavlova 2007] for a formal account of this proof-transforming procedure in the context of Hoare logic.

6.2. 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

(semi-)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 in general. 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. The definitions are as follows (for more details see Appendix or refer to the source code).

— for every $\mathbf{x}, \mathbf{y} : V$, s, s' : state, $d : D$, $\texttt{Eval}(\mathbf{x}, \mathbf{y}, \mathrm{s}, \mathrm{s}', d)$ holds iff the call $\texttt{eval x y s}$ terminates and returns the value (d, s');
— for every $\mathbf{x} : V$, $f : V \to T_{\mathsf{state}} D$, $\texttt{Eval}_{\mathbf{x}}(\mathbf{x}, f)$ holds iff f is a *total* function extending the partial function $\texttt{eval x}$, i.e., for every $\mathbf{x}, \mathbf{y} : V$, s, s' : state, $d : D$, $\texttt{Eval}(\mathbf{x}, \mathbf{y}, \mathrm{s}, \mathrm{s}', d)$ holds iff $f(\mathbf{x})(\mathrm{s}) = (d, \mathrm{s}')$;
— for every $\mathbf{x} : V$, $t : \mathit{Tree}_{V,D}$, s, s' : state, $d : D$, $l, l' : (V \times D)\,\textit{list}$, $\texttt{Wrap_Eval_x}(\mathbf{x}, t, \mathrm{s}, \mathrm{s}', d, l, l')$ holds iff the call $[\![t]\!]'_{\mathsf{state}}\,(\texttt{eval x})\ l\ \mathrm{s}$ terminates and returns the value $((d, l'), \mathrm{s}')$;
— for every $\mathbf{x} : V$, s, s' : state, $d : D$, $l' : (V \times D)\,\textit{list}$, $\texttt{Eval_rhs}(\mathbf{x}, \mathrm{s}, \mathrm{s}', d, l')$ holds iff the call $\texttt{eval_rhs x s}$ terminates and returns the value $((d, l'), \mathrm{s}')$;
— for every $\mathbf{x} : V$, s, s' : state, $\texttt{Solve}(\mathbf{x}, \mathrm{s}, \mathrm{s}')$ holds iff the call $\texttt{solve x s}$ terminates and returns the value s';
— for every $X : V\,\textit{list}$, s, s' : state, $\texttt{SolveAll}(X, \mathrm{s}, \mathrm{s}')$ holds iff $\texttt{solve_all}\ X\ \mathrm{s}$ terminates and returns the value 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, such as partial correctness.

6.3. Invariants

To formulate invariants, we first provide several definitions that relate finite sequences of pairs of variables and values from D with variable assignments $\sigma : V \to D$ and traces generated by σ in a tree $t \in \mathit{Tree}_{V,D}$. Given σ, the relation valid $\subseteq (V \times D)\,\textit{list} \times (V \to D)$ is inductively defined by:

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

and the relation legal $\subseteq (V \times D)\,\textit{list} \times \mathit{Tree}_{V,D}$ by:

— $\mathsf{legal}([\,], t)$ for any $t \in \mathit{Tree}_{V,D}$;
— for any $\mathbf{x} \in V$, $d \in D$, $l : (V \times D)\,\textit{list}$ and $c : D \to \mathit{Tree}_{V,D}$, if $\mathsf{legal}(l, c(d))$ then $\mathsf{legal}((\mathbf{x}, d){::}l, \mathsf{Que}(\mathbf{x}, c))$.

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

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

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

It is not difficult to show that $\mathsf{subtree}(\mathsf{trace}_\sigma\,t, t) = \mathsf{Ans}(a)$ holds for every tree $t \in \mathit{Tree}_{V,D}$ and variable assignment σ. By induction on length of a path we prove the following lemma.

LEMMA 6.1. *For any given $t \in \mathit{Tree}_{V,D}$, a path $l : (V \times D)$ list and a variable assignment $\sigma : V \to D$, the following is equivalent:*

— $l =$ *trace*$_\sigma\, t$;
— *valid*(l, σ), *legal*(l, t), *subtree*$(l, t) =$ *Ans*(a), *for some $a \in D$, hold.* □

From now on, for brevity, we denote `get_infl` as $\mathit{infl}_{[]}$ and `get` as $\sigma_\perp$. States s and s′ will denote a state before a call of a function and a state after the call terminates, respectively. Structures *stable*, *called*, *queued*, *infl* and σ are components of the state s, primed structures are components of the state s′. Let $\bar{f} : V \to \mathit{Tree}_{V,D}$ be a given strategy function. We say that variable x is *solved* in the state s if $\mathbf{x} \in \mathit{stable} \setminus \mathit{called}$. We treat lists as sets in the formulae below.

We define

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

We call a state s (a transition from s to s′) *consistent* if $\mathcal{I}_0(\mathrm{s})$ (respectively, $\mathcal{I}_1(\mathrm{s}, \mathrm{s}')$) holds. Thus, $\mathcal{I}_0(\mathrm{s})$ tells that the set of *called* variables is always a subset of *stable* variables, and those variables that are *queued* for reevaluation cannot be in *stable*. $\mathcal{I}_1(\mathrm{s}, \mathrm{s}')$ asserts that for a consistent transition the set of *stable* variables may only increase and the sets of *called* and *queued* may only shrink. For example, we will show below that `Solve`$(\mathbf{x}, \mathrm{s}, \mathrm{s}')$ relates s and s′ consistently. The formula

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

expresses that current values of variables are large in the state s′ than those in the state s. The formula

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

relates structures σ and *infl*. For every variable z, it asserts the following. If the value of z did not increase, then *infl*′ contains more dependencies; otherwise, if the value of z is altered in s′, all the variables influenced by z in s are solved in s′. The formula

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

asserts that for every variable z influencing x, this dependency is stored in the state s, and z is either stable or queued. The formula

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

defines the *correctness* of the state s. This means that for every variable x which is solved in s, the constraint $\sigma\, \mathbf{x} \sqsupseteq f_\mathbf{x}\, \sigma$ is satisfied for x and dependencies of x are treated correctly. Namely, as implied by the $\mathcal{I}_{\mathrm{dep}}(\mathbf{x}, \mathrm{s})$ conjunct, $\mathit{infl}\, \mathbf{x}$ overapproximates the set of actual dependencies of x returned by $\mathsf{dep}_{\bar{f},(\sigma_\perp\, \mathrm{s})}\, \mathbf{x}$, for every solved variable x in s. 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 intricate invariant refers to the function $[\![\cdot]\!]'_{\mathsf{state}}\,(\mathtt{eval}\,\mathbf{x})$. The formula

$$
\begin{array}{l}
\mathcal{I}_{[\![\cdot]\!]'(\mathtt{eval}\,\mathbf{x})}(\mathbf{x}, t, \mathbf{s}, \mathbf{s}', d, \mathit{vlist}, \mathit{vlist}') \equiv \\
\quad \mathbf{x} \in \mathit{stable} \wedge \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \wedge (\forall (\mathbf{y}, v) \in \mathit{vlist}.\, \mathbf{y} \in \mathit{stable}) \implies \\
\quad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathit{vlist} \subseteq \mathit{vlist}' \wedge (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{y} \in \mathit{stable}) \wedge \\
\quad \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathrm{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}') \wedge \\
\quad \big[\mathbf{x} \in \mathit{called} \wedge (\forall (\mathbf{y}, v) \in \mathit{vlist}.\, \mathbf{x} \in \mathit{infl}_{[\,]}\, \mathbf{y}\ \mathbf{s}) \wedge \\
\quad \mathsf{valid}(\mathit{vlist}, \sigma_\perp\, \mathbf{s}) \wedge \mathsf{legal}(\mathit{vlist}, \bar{f}_\mathbf{x}) \wedge \mathsf{subtree}(\mathit{vlist}, \bar{f}_\mathbf{x}) = t \implies \\
\qquad \big(\mathbf{x} \in \mathit{called}' \implies \\
\qquad\quad \mathsf{valid}(\mathit{vlist}', \sigma_\perp\, \mathbf{s}') \wedge \mathsf{legal}(\mathit{vlist}', \bar{f}_\mathbf{x}) \wedge \mathsf{subtree}(\mathit{vlist}', \bar{f}_\mathbf{x}) = \mathsf{Ans}(d) \wedge \\
\qquad\quad (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{x} \in \mathit{infl}_{[\,]}\, \mathbf{y}\ \mathbf{s}') \wedge \mathcal{I}_{\mathrm{dep}}(\mathbf{x}, \mathbf{s}')\big) \wedge \\
\qquad \big(\mathbf{x} \notin \mathit{called}' \implies \mathbf{x} \in \mathit{stable}' \setminus \mathit{called}'\big)\big]
\end{array}
$$

relates the arguments *vlist* and s with the result value $((d, \mathit{vlist}'), \mathrm{s}')$ of the call $[\![t]\!]'_{\mathsf{state}}\,(\mathtt{eval}\,\mathbf{x})$ *vlist* s, if it terminates. Remind that the function proceeds recursively on the tree t, taking as a parameter a list *vlist* of already visited variables along with their received values.

$\mathcal{I}_{[\![\cdot]\!]'(\mathtt{eval}\,\mathbf{x})}(\mathbf{x}, t, \mathbf{s}, \mathbf{s}', d, \mathit{vlist}, \mathit{vlist}')$ asserts that if the function $[\![t]\!]'_{\mathsf{state}}\,(\mathtt{eval}\,\mathbf{x})$ *vlist* s is invoked for a stable variable x and a partial path *vlist* of stable variables starting from an initial consistent correct state s then — if it terminates — it returns a value d and a longer path vlist' (that extends *vlist*) of stable visited variables, together with a consistent correct state s′. The $\mathcal{I}_\sigma(\mathrm{s}, \mathrm{s}')$ part tells that in result values $\sigma\,\mathrm{x}$ of all variables x grew, and $\mathcal{I}_{\sigma,\mathit{infl}}(\mathrm{s}, \mathrm{s}')$ guarantees that *infl* alters according to changes in σ.

The invariant distinguishes the case when $\mathrm{x} \in \mathit{called}$ initially. Then if *vlist* is a valid and legal path in $\bar{f}_\mathbf{x}$ leading to the subtree t and if $\mathrm{x} \in \mathit{called}'$ then the result path vlist' is again valid and legal in $\bar{f}_\mathbf{x}$ and leads to an answer d and all the dependencies of x are recorded. Note that by lemma 6.1 this implies that vlist' is a trace in $\bar{f}_\mathbf{x}$ by σ'. If $\mathrm{x} \in \mathit{called}$ and $\mathrm{x} \notin \mathit{called}'$ then x was reevaluated and solved during a recursive call for some variable y of t. It does not matter which value d is returned in this case since x is solved in s′ and the corresponding constraint is satisfied and will be preserved after a consecutive update of $\sigma\,\mathrm{x}$. In the case $\mathrm{x} \notin \mathit{called}$, x was solved in s, and thus we deduce that x is solved in s′ using $\mathcal{I}_1(\mathrm{s}, \mathrm{s}')$.

The formula

$$
\begin{array}{l}
\mathcal{I}_{\mathtt{eval_rhs}}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l') \equiv \\
\qquad \mathbf{x} \in \mathit{called} \wedge \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}) \implies \\
\qquad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\mathrm{corr}}(\mathbf{s}') \wedge \\
\qquad \big[\mathbf{x} \in \mathit{called}' \implies d = [\![\bar{f}_\mathbf{x}]\!]^\star(\sigma_\perp\, \mathbf{s}') \wedge l' = \mathsf{trace}_{\sigma'}\, \bar{f}_\mathbf{x} \wedge \\
\qquad\quad (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{x} \in \mathit{infl}_{[\,]}\, \mathbf{y}\ \mathbf{s}') \wedge \mathcal{I}_{\mathrm{dep}}(\mathbf{x}, \mathbf{s}')\big]
\end{array}
$$

relates the arguments x and s of the call of `eval_rhs` x s with the result state s′ whenever it terminates. If the input state s is consistent and correct then so is the state s′. In the case when x is still called in s′, we have that l' is a trace in $\bar{f}_\mathbf{x}$ by σ' and d is a value of the right-hand side of x on σ'. The latter can be shown by lemma 6.1. In the case $\mathrm{x} \notin \mathit{called}'$, the variable x is processed during some intermediate recursive call

and is solved in s′. The formula

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

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

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

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

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

THEOREM 6.2. *For all states* $\mathrm{s},\mathrm{s}' : \mathtt{state}$ *the following is true:*

—*for every* $\mathrm{x},\mathrm{y} : V$, $d : D$, $\mathit{EvalRel}(\mathrm{x},\mathrm{y},\mathrm{s},\mathrm{s}',d)$ *implies* $\mathcal{I}_{\texttt{eval}}(\mathrm{x},\mathrm{y},\mathrm{s},\mathrm{s}',d)$;
—*for every* $\mathrm{x} : V$, $t : \mathit{Tree}_{V,D}$, $d : D$, $l, l' : (V \times D)$ *list*, $\mathit{Wrap_Eval_x}(\mathrm{x},t,\mathrm{s},\mathrm{s}',d,l,l')$ *implies* $\mathcal{I}_{[\![\cdot]\!]'(\texttt{eval x})}(\mathrm{x},t,\mathrm{s},\mathrm{s}',d,l,l')$;
—*for every* $\mathrm{x} \in V$, $d : D$, $l' : (V \times D)$ *list*, $\mathit{Eval_rhs}(\mathrm{x},\mathrm{s},\mathrm{s}',d,l')$ *implies* $\mathcal{I}_{\texttt{eval_rhs}}(\mathrm{x},\mathrm{s},\mathrm{s}',d,l')$;
—*for every* $\mathrm{x} : V$, $\mathit{Solve}(\mathrm{x},\mathrm{s},\mathrm{s}')$ *implies* $\mathcal{I}_{\texttt{solve}}(\mathrm{x},\mathrm{s},\mathrm{s}')$;
—*for every* $w : V$ *list*, $\mathit{SolveAll}(w,\mathrm{s},\mathrm{s}')$ *implies* $\mathcal{I}_{\texttt{solve_all}}(w,\mathrm{s},\mathrm{s}')$. □

6.4. Putting things together

Having verified the invariants, we now prove that theorem 5.1 holds, i.e., that **RLD** is a local solver. Let $\mathrm{s}_{\mathit{init}}$ be the initial state with $\mathit{stable} = \mathit{called} = \mathit{queued} = \emptyset$, $\sigma = \mathit{infl} = \emptyset$. Assume that **RLD** applied to $(\bar{f}, X)$ terminates and let s′ be the state returned by the call `solve_all` X $\mathrm{s}_{\mathit{init}}$. According to the definition 4.4, we have to show that:

(1) $X \subseteq \mathit{stable}'$ and $\mathsf{dep}_{\bar{f},(\sigma_\bot\,\mathrm{s}')}(\mathit{stable}') \subseteq \mathit{stable}'$;
(2) $\sigma_\bot\,\mathrm{s}'\,\mathrm{x} \sqsupseteq [\![\bar{f}_{\mathrm{x}}]\!]^\star(\sigma_\bot\,\mathrm{s}')$ holds for every $\mathrm{x} \in \mathit{stable}'$.

By theorem 6.2, the invariant $\mathcal{I}_{\mathrm{solve_all}}(X,\mathrm{s}_{\mathit{init}},\mathrm{s}')$ holds; and its premise is true, inasmuch as both $\mathcal{I}_0(\mathrm{s}_{\mathit{init}})$ and $\mathcal{I}_{\mathrm{corr}}(\mathrm{s}_{\mathit{init}})$ hold. Therefore, we have $\mathcal{I}_1(\mathrm{s}_{\mathit{init}},\mathrm{s}')$, and hence $\mathit{called}' = \mathit{queued}' = \emptyset$. From $(X \cup \mathit{stable} \setminus \mathit{called} \subseteq \mathit{stable}' \setminus \mathit{called}')$ we conclude, that $X \subseteq \mathit{stable}'$. From $\mathcal{I}_{\mathrm{corr}}(\mathrm{s}')$ it follows, that $\forall \mathrm{x} \in \mathit{stable}'.\ \sigma_\bot\,\mathrm{s}'\ \mathrm{x} \sqsupseteq [\![\bar{f}_{\mathrm{x}}]\!]^\star(\sigma_\bot\,\mathrm{s}')$ and $\mathsf{dep}_{\bar{f},(\sigma_\bot\,\mathrm{s}')}(\mathit{stable}') \subseteq \mathit{stable}'$ hold, and the statement of theorem 5.1 follows. □

7. MONOTONIC CASE

In many applications, right-hand sides f arise as monotonic functions, i.e., $\sigma_1 \sqsubseteq \sigma_2$ implies $f\,\sigma_1 \sqsubseteq f\,\sigma_2$. In this case, one would expect that a "good" solver produces more precise results. As shown by the following example, **RLD** is *not* an exact solver, i.e., it may not return a precise solution even if right-hand sides are monotonic functions.

Consider the constraint system with right-hand sides as on Figure 6 over the lattice $\mathbb{D} = (\{\bot, a, b, \top\}, \sqsubseteq, \sqcup)$, where $\bot \sqsubset a, b \sqsubset \top$, a and b are incomparable. For example, the third tree on the figure may represent a right-hand side

```
f_x = fun σ →
    let s_1 = σ s in
    let u_1 = σ u in
    if u_1 ⊑ ⊥ then
        let v_1 = σ v in
        let u_2 = σ u in
        if u_2 ⊑ ⊥ then v_1 ⊔ a else b
    else
        let v_1 = σ v in v_1 ⊔ a
```

Note that a result of the query to s is never used during computations of $f_{\mathbf{x}}\,\sigma$, but this dependency allows to trigger recomputation of x once a value of s changes. Also, there is a path in $\bar{f}_{\mathbf{x}}$ where variable $\mathbf{u}$ is queried twice. Those query nodes are marked by (1) and (2). We want to compute a local solution for the variable $\mathbf{t}$. It is not difficult to check that all the right-hand sides are monotonic functions and the least solution of the system is

$$\sigma_0(x) = a, \quad x \in \{\mathbf{t}, \mathrm{s}, \mathrm{x}, \mathbf{u}, \mathrm{v}\}\ .$$

Will **RLD** return an *exact* solution?

To figure out the problem, let us trace the computations done by the instrumented **RLD**. From initial state, we call `solve` $\mathbf{t}$, which in turn recursively invokes `solve` s. During the invocation of `solve` s, the algorithm recursively computes new values of variables x, $\mathbf{u}$, and v. We skip a description of these computation steps. Before a new (altered) value of s (for which $\sigma(\mathrm{s}) = a$) is returned, the algorithm needs to recompute all the variables dependent on s (these are variables from *infl* s $= [\mathrm{x}; \mathrm{v}]$). Thus, *infl* s is reset to $[\,]$, and x and v are removed from the sets *stable* and *called*.

1. For recomputation of x, `solve` x is called, and variable x is put back into *stable* and *called*. The state just prior to reevaluation of the right-hand side $\bar{f}_{\mathbf{x}}$ (the call of `eval_rhs` x) is as follows:

$$\begin{array}{l} \textit{stable} = \{\mathrm{s}, \mathbf{t}, \mathbf{u}, \mathrm{x}\}, \quad \textit{called} = \{\mathbf{t}, \mathrm{x}\}, \\ \sigma(\mathrm{s}) = \sigma(\mathrm{x}) = a, \quad \sigma(\mathbf{u}) = \sigma(\mathbf{t}) = \sigma(\mathrm{v}) = \bot, \\ \textit{infl}\ \mathbf{t} = \textit{infl}\ \mathrm{s} = [\,], \quad \textit{infl}\ \mathbf{u} = [\mathrm{x}; \mathrm{x}], \quad \textit{infl}\ \mathrm{v} = [\mathrm{x}; \mathbf{u}; \mathrm{s}], \quad \textit{infl}\ \mathrm{x} = [\mathrm{s}]. \end{array}$$

During the run of `eval_rhs` x the algorithm traverses the tree $\bar{f}_{\mathbf{x}}$ and recomputes necessary variables as described below.

— Since $\mathrm{s}, \mathbf{u} \in$ *stable*, the algorithm does not descend into solving them. The structures σ, *stable* and *called* are not changed, but the solver records that x depends on s and $\mathbf{u}$, i.e., x is added to *infl* s and *infl* $\mathbf{u}$.

— Since $\sigma(\mathbf{u}) = \bot$, the algorithm takes the upper branch of (1) in $\bar{f}_{\mathbf{x}}$. Thus, it recomputes v, which gets a larger value a (as $\sigma(\mathrm{s}) = a$). Since the value of v has changed, variables influenced by v might need to be recomputed before `solve` v returns. At this point, these are variables from *infl* v $= [\mathrm{x}; \mathbf{u}; \mathrm{s}]$. Those are removed from the sets *stable* and *called*, and get recomputed by calling consequently `solve` x, `solve` $\mathbf{u}$, and

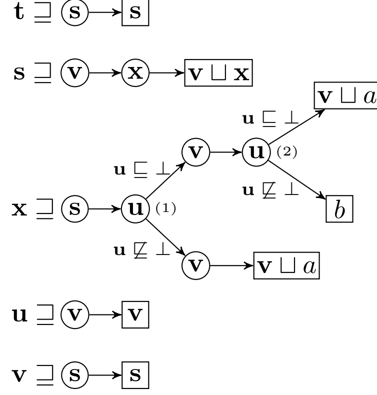

Fig. 6. Counterexample for the monotonic case. The constraint system is defined over the lattice $\mathbb{D} = (\{\bot, a, b, \top\}, \sqsubseteq, \sqcup)$ with $\bot \sqsubset a, b \sqsubset \top$, a and b are incomparable.

solve s. At the end end of their run, the altered state components are (*infl* structure is omitted)

$$\begin{array}{l} \mathit{stable} = \{\mathbf{s}, \mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{x}\}, \quad \mathit{called} = \{\mathbf{t}\}, \\ \sigma(\mathbf{s}) = \sigma(\mathbf{u}) = \sigma(\mathbf{v}) = \sigma(\mathbf{x}) = a, \quad \sigma(\mathbf{t}) = \bot. \end{array}$$

Thus, the value $\sigma(\mathbf{u})$ is altered and equals a. The algorithm returns from solve $\mathbf{v}$ and meets the branching (2) in $\bar{f}_{\mathbf{x}}$. This time, the algorithm must follow the branch $\mathbf{u} \not\sqsubseteq \bot$, as $\sigma(\mathbf{u}) = a$ and returns the "bad" value b.

Then, evaluation of eval_rhs $\mathbf{x}$ finishes and $\sigma(\mathbf{x})$ receives a value $a \sqcup b = \top$. Thus, more recomputations triggered which we omit here.

2. For recomputation of $\mathbf{v}$, solve $\mathbf{v}$ is called, but $\mathbf{v}$ is solved at this moment.

Finally, the algorithm terminates and returns a solution σ_1 with $\sigma_1(\mathbf{x}) = \top$, and thus σ_1 is not minimal. The reason why the "bad" value b is reached during the run of the solver is that $\mathbf{u}$ changes its value between branchings (1) and (2). Note that the leaf b is *not* reachable in $[\![\bar{f}_{\mathbf{x}}]\!]_{unit}(\mathsf{val}_{T_{unit}} \circ \sigma)$, for any variable assignment σ. We will discuss this issue in more detail in the next section.

Although **RLD** does not return a minimal solution generally, it is exact for a subclass of monotonic strategy functions. We say that a strategy tree t has an *unique-lookup* property if for any path through the tree any variable $\mathbf{v}$ is queried at most once. The property does not hold for the tree $\bar{f}_{\mathbf{x}}$ as on Fig. 6, since there exists a path on which the variable $\mathbf{u}$ is queried twice. We prove:

THEOREM 7.1. *Given a constraint system $\mathcal{S} = (V, f)$ over the complete lattice $\mathbb{D}$ with a pure and monotonic f such that $\bar{f}_{\mathbf{x}}$ enjoys the unique-lookup property for every $\mathbf{x} \in V$.* **RLD** *when applied to $(\bar{f}, X)$ — if it terminates — returns a local solution (σ, X') such that $\sigma \restriction_{X'} = \mu \restriction_{X'}$, where μ is a least solution of $\mathcal{S}$.*

PROOF. To show this, we need to establish extra invariants for every function of the algorithm. The one for the function solve_all is as below.

$$\begin{array}{l} \mathcal{I}'_{\texttt{solve_all}}(\mathbf{x}, \mathrm{s}, \mathrm{s}') \equiv \mathit{UniqueLookup}(\bar{f}) \wedge \mathit{Monotone}(\bar{f}) \implies \\ \qquad \forall \mu : V \to D.\, \mu\, \mathbf{x} \sqsupseteq [\![\bar{f}_{\mathbf{x}}]\!]^{\star} \mu \wedge \sigma_{\bot}\, \mathrm{s} \sqsubseteq \mu \implies \sigma_{\bot}\, \mathrm{s}' \sqsubseteq \mu \ , \end{array}$$

where $\sigma_{\bot}$ s denotes the function get s. The invariant states that, given any solution μ, if $\sigma_{\bot}$ is (pointwise) less than μ in the state s then it stays so in the state s'. We provide similar invariants for the rest of functions. The proof is by induction on number of unfoldings of definitions.

Let s_{init} be an initial state with $stable = called = queued = \emptyset$, $\sigma = infl = \emptyset$, and let μ be the least solution of $\mathcal{S}$. Assume that **RLD** applied to $(\bar{f}, X)$ terminates and let s′ be the state returned by the call `solve_all` X s_{init}. Since $\sigma_\perp\, s_{init}\, x = \perp \sqsubseteq \mu\, x$, for all x, we have $\sigma_\perp\, s' \sqsubseteq \mu$. Let $\sigma'_{X'}$ be an extension of $\sigma_\perp\, s'$ to a global solution as defined in Section 4.2. Since X' is dep-closed, we have

$$[\![\bar{f}_x]\!]\,(\sigma_\perp\, s') = [\![\bar{f}_x]\!]\;\sigma'_{X'} \qquad \text{for all } x \in X' \ .$$

Using monotonicity of f, for every $x \in X'$ we get $\mu\, x \sqsupseteq \sigma_\perp\, s'\, x \sqsupseteq [\![\bar{f}_x]\!]\,(\sigma_\perp\, s') = [\![\bar{f}_x]\!]\;\sigma'_{X'} \sqsupseteq [\![\bar{f}_x]\!]\,\mu = \mu\, x$. Therefore, $\mu\, x = [\![\bar{f}_x]\!]\,(\sigma_\perp\, s')$ holds, for all $x \in X'$. This proves the theorem. □

8. THE SOLVER RLDE

In this section, we present a modification of the solver **RLD** which is exact.

The idea of improvement comes from a careful inspection of behavior of the instrumented **RLD**. In the previous example, we observed that, for variable x, during a call to `solve x` after `eval_rhs x` returns, x may *not* belong to *called*. This may happen in the presence of cyclic variable dependencies — for example, when x depends on some v (met in $\bar{f}_x$), which gets a strictly larger value while evaluating `eval_rhs x`. In this case, a recursive call to `solve x` is triggered in order to recompute x. The invariant $\mathcal{I}_{\texttt{solve}}$ guarantees that x is *solved* after the recursive call to `solve` returns, i.e., $x \in stable \setminus called$ and the constraint for x is satisfied. The variable x still stays *solved* after an answer-value, say b, is reached, and thus there is no need to update it with join with b (although, it is always safe to do it). In the previous example, b spoils the result solution, although it is not reachable in $\bar{f}_x$ by any stateless variable assignment σ.

The idea, therefore, is to check whether x is still in *called* before updating it. To implement this idea, we adjust the original RLD in the following way (Fig. 7). The structure *called* is maintained in states explicitly, as in the instrumented version. This information can then be used to avoid unnecessary updates.

We prove the modified algorithm **RLDE** correct. Moreover, whenever it terminates it returns an exact solution if right-hand sides are monotonic functions over a complete lattice. We have:

THEOREM 8.1. *The algorithm* **RLDE** *is an* exact *generic local solver.*

A proof is similar to the proofs of theorems 5.1 and 7.1. The corresponding invariants can be reused with small changes. □

9. TERMINATION

THEOREM 9.1. *For any finite constraint system $S = (V, f)$ with pure f over a (semi-)lattice $\mathbb{D}$ that has no infinite strictly ascending chains, the algorithm* **RLD** *(***RLDE***) when called with $(\bar{f}, X)$, for a finite $X \subseteq V$, terminates.*

PROOF (SKETCH). First, we show that each variable can be destabilized at most finitely many times. Suppose, by contradiction, that there exists a variable x destabilized infinitely often. Since V is finite, there exists z such that $\sigma(z)$ increases infinitely often while $x \in infl\ z$. That contradicts to the ascending chain condition of $\mathbb{D}$.

Suppose, that the algorithm does not terminate for $(\bar{f}, X)$. Since strategy trees are well-founded (by definition), function `solve` must be called infinitely many times. Therefore, there exist a variable $x \in V$ such that `solve x` is triggered infinitely often. Note that `solve x` is executed only either when evaluating right-hand side `eval_rhs y`, for some variable y (such that x is queried in $\bar{f}_y$), or when x is destabilized. Since x can be destabilized at most finitely many times, there exists a variable y such that `solve x`

```
function extract_work(x : V) =
    let work = infl_[] x in
    called ← called \ work; stable ← stable \ work; infl x ← []; work

function solve(x : V) =
    if x ∈ stable then ()
    else
        stable ← stable ∪ {x};
        called ← called ∪ {x};
        let rhs = eval_rhs(x) in
        if x ∈ called then
            let cur = σ_⊥ x in
            let new = cur ⊔ rhs in
            called ← called \ {x};
            if new ⊑ cur then ()
            else
                σ x ← new;
                let work = extract_work(x) in
                solve_all(work)
            end
        end
    end
```

Fig. 7. Main functions of the optimized solver (**RLDE**)

is called infinitely often from `eval_rhs` y. For that, y itself has to be destabilized infinitely often, since `solve` y when called with $y \in stable$ does not launch `eval_rhs` y and simply returns a current state. Contradiction. □

10. CONCLUSION

In the present paper, we introduce two general-purpose local solvers **RLD** and **RLDE**. As we have shown, the solvers admit a purely functional implementation. Based on the semantical notion of *purity* [Hofmann et al. 2010b], we have presented a rigourous proof of partial correctness of the algorithms. Moreover, we provide further verified properties of the solvers, such as sufficient conditions for returning fragments of the least solution of a given constraint system by **RLD**, and exactness of **RLDE**. Since we require neither the ascending chain property of an abstract value domain nor finiteness of an equation system, the solvers may not terminate in general. However, as we demonstrate, one can guarantee termination of **RLD** (**RLDE**) under certain conditions.

By all 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. A modification of the solver **RLD** (for systems with multiple constraints) has been implemented in the static analyser for concurrent C programs GOBLINT [Seidl and Vojdani 2009].

References

BACKES, M. AND LAUD, P. 2006. Computationally sound secrecy proofs by mechanized flow analysis. In *ACM Conference on Computer and Communications Security*. 370–379.

CACHERA, D., JENSEN, T. P., PICHARDIE, D., AND RUSU, V. 2004. Extracting a data flow analyser in constructive logic. In *Programming Languages and Systems, 13th European Symp. on Programming (ESOP)*. Springer, LNCS 2986, 385–400.

CHARLIER, B. L. AND HENTENRYCK, P. V. 1992. A universal top-down fixpoint algorithm. Tech. Rep. CS-92-25, Brown University, Providence, RI 02912.

COUPET-GRIMAL, S. AND DELOBEL, W. 2004. A uniform and certified approach for two static analyses. In *TYPES*, J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds. Lecture Notes in Computer Science Series, vol. 3839. Springer, 115–137.

FECHT, C. 1995. Gena — a tool for generating prolog analyzers from specifications. In *2nd Static Analysis Symposium (SAS)*. LNCS 983, 418–419.

FECHT, C. AND SEIDL, H. 1998. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *European Symposium on Programming (ESOP)*. LNCS 1381, Springer Verlag, 90–104. Long version in *Northern Journal of Computing 5, 304–329, 1998*.

FECHT, C. AND SEIDL, H. 1999. A faster solver for general systems of equations. *Sci. Comput. Program. 35,* 2, 137–161.

HOFMANN, M., KARBYSHEV, A., AND SEIDL, H. 2010a. Verifying a local generic solver in Coq. In *SAS*, R. Cousot and M. Martel, Eds. Lecture Notes in Computer Science Series, vol. 6337. Springer, 340–355.

HOFMANN, M., KARBYSHEV, A., AND SEIDL, H. 2010b. What is a pure functional? In *ICALP (2)*, S. Abramsky, C. Gavoille, C. Kirchner, F. M. auf der Heide, and P. G. Spirakis, Eds. Lecture Notes in Computer Science Series, vol. 6199. Springer, 199–210.

HOFMANN, M. AND PAVLOVA, M. 2007. Elimination of ghost variables in program logics. In *Proc. Trustworthy Global Computing, LNCS 4912*, G. Barthe and C. Fournet, Eds. Lecture Notes in Computer Science Series, vol. 4912. Springer, 1–20.

JORGENSEN, N. 1994. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In *1st Static Analysis Symposium (SAS)*. LNCS 864, Springer Verlag, 329–345.

KLEIN, G. AND NIPKOW, T. 2003. Verified bytecode verifiers. *Theor. Comput. Sci. 3,* 298, 583–626.

POTTIER, F. 2009. Lazy least fixed points in ML. Unpublished.

SEIDL, H. AND VOJDANI, V. 2009. Region analysis for race detection. In *Static Analysis, 16th Int. Symposium, (SAS)*. LNCS 5673, Springer Verlag, 171–187.

SEIDL, H., WILHELM, R., AND HACK, S. 2012. *Compiler Design: Analysis and Transformation*. SpringerLink : Bücher. Springer.

THE COQ DEVELOPMENT TEAM. 2012. *The Coq proof assistant reference manual*. TypiCal Project (formerly LogiCal). Version 8.4.