On the Verification of Local Generic Solvers

Martin Hofmann, Aleksandr Karbyshev and Helmut Seidl

TUM-I1323
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 $x \sqsubseteq f_x, x \in V$, over some (semi-)lattice $D$ 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 $x \sqsubseteq f_x, x \in V$, over some (semi-)lattice $D$, 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 \textit{strategy tree}. This means that any evaluation of \( f \) on a variable assignment \( \sigma \) 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 \textbf{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 \textit{monotonic}. In that case, the Knaster-Tarski theorem for complete lattices guarantees the existence of the unique \textit{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 \textit{exact}.

The fixpoint algorithm \textbf{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, \textbf{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 \textit{arbitrary} monotonic constraint systems, we enhance the algorithm \textbf{RLD} to the algorithm \textbf{RLDE} which still allows interference of different evaluations but identifies potentially superfluous evaluations and ignores their contributions. Thus, algorithm \textbf{RLDE} is exact.

The paper is structured as follows. In section 2 we describe the fixpoint algorithm \textbf{RLD}. In section 3 we provide an example of a small, seemingly natural, but erroneous optimization of \textbf{RLD}. In sections 4, precise definitions of \textit{constraint system} and \textit{solver} are given. Further, in section 5 we give a purely functional implementation of \textbf{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 \textbf{RLD} when applied to monotonic constraint systems. We show that \textbf{RLD} is \textit{not} an exact solver and therefore, in section 8, we present the enhanced version \textbf{RLDE}. In section 9 we sketch a proof of termination of \textbf{RLD} (\textbf{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 \url{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 performs on a constraint system \( x \ni f_x, x \in V \), where \( V \) is a set of variables, defined over some bounded join-semilattice \( D = (\mathbb{D}, \sqcap, \sqcup, \bot) \) consisting of a carrier \( \mathbb{D} \) 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 \( \sigma \), storing current values (from \( \mathbb{D} \)) 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

\[
\sigma_\bot x = \begin{cases} \sigma x & \text{if } x \in \text{dom}(\sigma) \\ \bot & \text{otherwise} \end{cases}
\]

that returns a current value of \( \sigma x \) if it is defined; otherwise, it returns \( \bot \).

2. Finite set \( \text{stable} \subseteq V \). Intuitively, if variable \( x \) is marked as stable then either \( x \) is already solved, i.e., a computation for \( x \) has completed and \( \sigma \) 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 \( \text{infl} \), which stores dependencies between variables. More exactly, \( \text{infl} x \) 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 \( \text{infl} \) as a (finite) mapping from \( V \) to lists of \( V \). We define the auxiliary function

\[
\text{infl}_\bot x = \begin{cases} \text{infl} x & \text{if } x \in \text{dom}(\text{infl}) \\ \emptyset & \text{otherwise} \end{cases}
\]

The structures have initial values \( \sigma_{\text{init}} = \emptyset, \text{stable}_{\text{init}} = \emptyset, \text{infl}_{\text{init}} = \emptyset \).

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 \( x \in X \) 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 \( \sigma x \ni f_x \sigma \). For that, it evaluates a value \( \text{rhs} \) of the right-hand side \( f_x \) by invoking eval_rhs \( x \). After eval_rhs returns, solve calculates a least upper bound of \( \text{rhs} \) with \( \text{cur} \), the current value of \( \sigma x \), and stores the produced value in \( \text{new} \). Then it compares values of \( \text{new} \) and \( \text{cur} \). If the current value is larger than a new one, the constraint for \( x \) is satisfied, and solve returns. Otherwise, the value of \( \sigma \) for \( x \) gets updated with \( \text{new} \). Since the value of \( \sigma x \) 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 \( \text{work} = \text{infl}_\bot x \) 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 = 0; stable = 0; infl = 0;
  solve_all(X);
  (σ ⊥ , stable)
end

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

Our goal is to prove that the algorithm RLD is a local generic solver for any (possibly infinite) constraint system $S = (V, f)$ where right-hand sides $f_x$ are pure functions in the sense of [Hofmann et al. 2010b].

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 ⊆ 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 implemented efficiently.

We mention that the right-hand side $f_x$ is not evaluated directly on $\sigma$, but by using an auxiliary stateful function $\lambda y.\text{eval(x,y)}$, which allows firstly to get better values for variables the variable $x$ depends on. Thus, once eval(x,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 $S = (V, f)$ 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
      called ← called \ {x};
      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 processed, 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 = ∅). It tracks a subset of stable variables currently being processed, i.e., called ⊆ stable is invariant for every program point. We say that the variable x is solved if x ∈ stable \ called. Variable x is added to called just before a reevaluation of fx 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 D = (⊥, a, ⊤), ⊑, ⊔) with ⊥ ⊏ a ⊏ ⊤. Figure 3 represents right-hand sides of the con-
constraint 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 \( D \) 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:

\[
\begin{align*}
  f_s &= \text{fun } \sigma \rightarrow \text{let } v_1 = \sigma \, v \text{ in let } x_1 = \sigma \, x \text{ in } v_1 \cup x_1, \\
  f_u &= \text{fun } \sigma \rightarrow \text{let } v_1 = \sigma \, v \text{ in if } (v_1 \subseteq \bot) \text{ then } a \text{ else } \top.
\end{align*}
\]

Functions \( f_s, f_x \) and \( f_v \) can be represented similarly.

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

1. \( \text{solve } x \) is called, and the variable \( x \) is added again to the sets \( \text{stable} \) and \( \text{called} \).

The state just prior to reevaluation of the right-hand side \( f_x \) (call of \( \text{eval_rhs } x \)) is:

\[
\begin{align*}
  \text{stable} &= \{ s, t, u, x \}, & \text{called} &= \{ t, x \}, \\
  \sigma(s) &= \sigma(u) = \sigma(x) = a, & \sigma(t) &= \sigma(v) = \bot, \\
  \text{infl } t &= \text{infl } s = [\ ], & \text{infl } u &= [x], & \text{infl } v &= [x; u; s], & \text{infl } x &= [s].
\end{align*}
\]

During the invocation of \( \text{eval_rhs } x \) the algorithm traverses the tree representing \( f_x \) and tries to recompute variables \( s, u \) and \( v \) in turn.

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

— The algorithm recomputes \( v \) (call to \( \text{solve } v \)), which gets a large value \( a \) (as \( \sigma(s) = a \)). Since \( \sigma(v) \) has changed, variables influenced by \( v \) might need to be recomputed, these are variables from \( \text{infl } v = [x; u; s] \). However, the algorithm does not destabilize \( x \), as \( x \in \text{called} \). Thus, the variable \( u \) is destabilized and recomputed (since \( u \in \text{stable} \setminus \text{called} \)), and gets a greater value \( \top \). At this point, although \( \text{infl } u = [x; x] \), the variable \( x \) is again not recomputed, since \( x \in \text{called} \). Thus, the value of \( x \) remains as before, \( \sigma(x) = a \). Then \( s \) gets recomputed, but this does not lead to a change in the state. Finally, \( \text{solve } s \) returns.
2. solve \( v \) is called, but \( v \) is solved at this moment.
Finally, the algorithm returns the variable assignment
\[
\sigma_1(t) = \sigma_1(s) = \sigma_1(x) = \sigma_1(v) = a, \quad \sigma_1(u) = \top
\]
that is not a solution, since the constraint for \( x \) is not satisfied:
\[
\sigma_1(x) = a \sqsubseteq \top = f_x \sigma_1.
\]
We conclude that this “natural” optimization is wrong. The counterexample was our guiding motivation for a rigorous verification of the fixpoint algorithm \( \text{RLD} \).

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 \( D = (D, \sqcup, \sqsubseteq, \bot) \) 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 \( x \in V \) returns a corresponding right-hand side
\[
f_x : (V \to TD) \to TD,
\]
where \( T \) is a monad constructor with associated monad operations \( \text{val}^X : X \to TX \) and \( \text{bind}^{X,Y} : TX \to (X \to TY) \to TY \). 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
\[
T_S X = S \to X \times S,
\]
\[
\text{val}^X_{T_S} x = \text{fun} s \to (x, s),
\]
\[
\text{bind}^{X,Y}_{T_S} t g = \text{fun} s \to \text{let} (x_1, s_1) = t s \text{ in } g x_1 s_1.
\]
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 \( f_x \sigma \) 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 [Hofmann et al. 2010b]. It is guaranteed, however, by the notion of purity introduced in [Hofmann et al. 2010b]. If the function \( f_x \) is pure in the sense of [Hofmann et al. 2010b], then \( f_x \) is representable by means of a strategy tree. This means that any evaluation of \( f_x \) 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 \( \text{Tree}_{V,D} \) of strategy trees inductively by:
- for \( a \in D \), \( \text{Ans}(a) \in \text{Tree}_{V,D} \);
- for \( x \in V \) and \( c : D \to \text{Tree}_{V,D} \), \( \text{Que}(x, c) \in \text{Tree}_{V,D} \).

Fix a monad \( T \). We define the function
\[
\llbracket \cdot \rrbracket_T : \text{Tree}_{V,D} \to (V \to TD) \to TD
\]
recursively by:
\[
\begin{align*}
\lbrack \text{Ans}(a) \rbrack_T h &= \text{val}_T a, \\
\lbrack \text{Que}(x, c) \rbrack_T h &= \text{bind}_T(h(x))(\text{fun } a \rightarrow \lbrack c \ a \rbrack_T h).
\end{align*}
\]

For the state monad \(T_S\), we have
\[
\begin{align*}
\lbrack \text{Ans}(a) \rbrack_S h &= \text{fun } s \rightarrow (a, s), \\
\lbrack \text{Que}(x, c) \rbrack_S h &= \text{fun } s \rightarrow a_1, s_1 = h(x) \text{in } (\lbrack c(a_1) \rbrack_S h) s_1.
\end{align*}
\]

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
\[
h = \text{val}_{T_S} \circ \sigma = \text{fun } x \rightarrow \text{val}_{T_S}(\sigma x),
\]
then for all states \(s\) and trees \(t \in \text{Tree}_{V,D}\)
\[
\lbrack t \rbrack h s = (a, s), \ a \in D.
\]

Therefore, we define the function
\[
\lbrack \cdot \rbrack^* : \text{Tree}_{V,D} \rightarrow (V \rightarrow D) \rightarrow D
\]
by
\[
\lbrack t \rbrack^* \sigma = \text{fst}(\lbrack t \rbrack_{\text{unit}} (\text{val}_{\text{unit}} \circ \sigma)(*)], \ \text{where unit} = \{\ast\}.
\]

In our application, the solver not only evaluates pure functions, i.e., strategy trees, but also records the values accessed during the evaluation. In order to reason about the sequence of accessed variables together with their values, we \textit{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 \(T\),
\[
\lbrack \cdot \rbrack_T^* : \text{Tree}_{V,D} \rightarrow (V \times D) \text{list} \rightarrow (V \rightarrow TD) \rightarrow T(D \times (V \times D) \text{list})
\]
is defined by
\[
\begin{align*}
\lbrack \text{Ans}(a) \rbrack_T^* \ l h &= \text{val}_T(a, l), \\
\lbrack \text{Que}(x, c) \rbrack_T^* \ l h &= \text{bind}_T(h(x))(\text{fun } a \rightarrow \lbrack c \ a \rbrack_T^* ((l \ @ \ ((x, a)))) h),
\end{align*}
\]
and, again instantiated for a state monad \(T_S\),
\[
\begin{align*}
\lbrack \text{Ans}(a) \rbrack_S^* \ l h &= \text{fun } s \rightarrow ((a, l), s), \\
\lbrack \text{Que}(x, c) \rbrack_S^* \ l h &= \text{fun } s \rightarrow \text{let } (a_1, s_1) = h(x) \text{in } (\lbrack c(a_1) \rbrack_S^* (l \ @ \ ((x, a_1)))) s_1.
\end{align*}
\]
where \(\oplus\) is a list append function. Then for every strategy tree \(t \in \text{Tree}_{V,D}, h : V \rightarrow TD, l_1 \in (V \times D) \text{list}, a \in D\)
\[
\lbrack t \rbrack_S^* h s = (a, s') \iff \lbrack t \rbrack^*_S l_1 h s = ((a, l_2), s') \text{ for some } l_2.
\]
Thus, instrumentation does not influence the state. Moreover, if \(h = \text{val}_{T_S} \circ \sigma\) for some \(\sigma : V \rightarrow D\) then for every state \(s\) we have
\[
\lbrack t \rbrack^*_S h s = ((a, l), s) \quad \text{for some } a \in D, \ l \in (V \times D) \text{list}.
\]
In this case, the list \(l\) represents a trace of variables together with their respective values that are accessed within evaluation of \(\lbrack t \rbrack^*_S h\) for any \(S\), in particular, for \(\lbrack t \rbrack^* \sigma\). Note that if, for some variable \(x, (x, a_1)\) and \((x, a_2)\) occur in \(l\) then \(a_1 = a_2\) necessarily. Given \(f : V \rightarrow \text{Tree}_{V,D}\) and \(\sigma : V \rightarrow D\), we define
\[
\text{dep}_{f, \sigma} x = \{y \ | \ (y, \_ ) \in \text{trace}_\sigma(f x)\}.
\]
Intuitively, the function $\text{dep}_{\bar{f},\sigma}$ applied to a variable $x$ returns a set of variables that $x$ directly depends on relative to $\sigma$, i.e., a set of those variables values of which are required to evaluate the strategy tree $\bar{f}_x$. For $X \subseteq V$, we define $\text{dep}_{\bar{f},\sigma}(X) = \bigcup_{x \in X} \text{dep}_{\bar{f},\sigma}(x)$.

4.2. Solutions

Definition 4.2. Let $S = (V, f)$ be a constraint system over a (semi-)lattice $D$. We say that the variable assignment $\sigma : V \rightarrow D$ is a solution of the constraint system $S$ if for every $x \in V$, $\sigma x \sqsupseteq d$ whenever $(d, \ast) = f_x(\text{val}_{T_{\text{unit}}} \circ \sigma)(\ast)$ holds. For the latter, we also write $\sigma x \sqsupseteq f_x \sigma$.

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

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

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

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

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

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

Definition 4.4. The partial function $A_{V,D} : (V \rightarrow \text{Tree}_{V, D}) \times 2^V \rightarrow (V \rightarrow D) \times 2^V$ polymorphic in $V$ and $D$ is a generic local solver if given a constraint system $S = (V, f)$ over some bounded joint-semilattice $D = (D, \sqsubseteq, \sqcup, \sqcap, \bot)$ with pure $f, A$ 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 $(\sigma, X')$ of $S$ relative to $X$, whenever it terminates.

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

Definition 4.5. We say that the generic local solver $A_{V,D}$ is exact if, for any constraint system $(V, f)$ over a complete lattice $D$ with pure and monotonic $f, A$ 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 $(\sigma, X')$ of $S$ relative to $X$ such that for the least solution $\mu$ of $S$, $\mu \mid_{X'} = \sigma \mid_{X'}$ 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 →
[f]_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

\[
\text{type state} = (V \rightarrow D) \times (V \rightarrow V\text{ list}) \times 2^V.
\]

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

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

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

\[
\begin{align*}
\text{let } & \text{extract\_work } x = \text{fun } s \rightarrow \\
& \text{let } w = \text{get\_infl } x s \text{ in} \\
& \text{let } s = \text{rem\_infl } x s \text{ in} \\
& \text{let } s = \text{fold\_left } (\text{fun } y \rightarrow \text{rem\_stable } y s) s w \text{ in} \\
(\text{w, s})
\end{align*}
\]

Using the evaluation function $[\cdot]_{state}$ for strategy trees specialized for the state monad $T_{state}$, the mutually recursive functions $\text{eval, eval\_rhs, solve and solve\_all}$ of the algorithm are then given in Fig. 4. Provided a list of interesting variables $X \subseteq V$, the algorithm calls the function $\text{solve\_all}$ from the initial state $s_{init} = (\emptyset, \emptyset, \emptyset)$.

From now on, $\text{RLD}$ refers to this functional implementation. We prove:

**Theorem 5.1.** The algorithm $\text{RLD}$ 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 $\text{called}$ of variables which are being processed;
— the set $\text{queued}$ of variables which have been *destabilized*, i.e., removed from the set $\text{stable}$ by the algorithm, and have not yet been reevaluated by $\text{solve}$.

Accordingly, the type $\text{state}$ in the instrumented program is given by:

\[
\text{type state} = (V \rightarrow D) \times (V \rightarrow V \text{ list}) \times 2^V \times 2^V \times 2^V.
\]

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

- $\text{add\_called} : V \rightarrow \text{state} \rightarrow \text{state}$ adds a given variable to $\text{called}$;
- $\text{rem\_called} : V \rightarrow \text{state} \rightarrow \text{state}$ removes a given variable from $\text{called}$;
- $\text{add\_queued} : V \rightarrow \text{state} \rightarrow \text{state}$ adds a given variable to $\text{queued}$;
- $\text{rem\_queued} : V \rightarrow \text{state} \rightarrow \text{state}$ removes a given variable from $\text{queued}$.

In the instrumented implementation, we also replace the evaluation $[\cdot]_{\text{state}}$ for strategy trees with $[\cdot]_{\text{state}}'$ which additionally returns the list of accessed variables together with their respective values. Also, the function $\text{extract\_work}$ for a given $x$ additionally removes all the variables influenced by $x$ from the set $\text{called}$ and adds them to the set $\text{queued}$ in the current state. The instrumented functions $\text{eval\_rhs}$ and $\text{solve}$ are given in Fig. 5. The functions $\text{eval}$ and $\text{solve\_all}$ remain unchanged.
and \texttt{eval\_rhs x = fun s -> [f']_{\text{state}} (eval x) [] s}
and \texttt{solve x = fun s -> if is\_stable x s then s
else}
let \(s = \text{rem\_queued x s in}
\) let \(s = \text{add\_stable x s in}
\) let \((\text{rhs, _}) = \text{eval\_rhs x s in}
\) let \(s = \text{rem\_called x s in}
\) let \(\text{cur} = \text{get s x in}
\) let \(\text{new} = \text{cur} \cup \text{rhs in}
\) if \(\text{new} \subseteq \text{cur} \) then \(s\)
else
let \(s = \text{set x new s in}
\) let \((\text{work}, s) = \text{extract\_work x s in}
\) solve\_all work s

Fig. 5. Instrumented implementation of the functions \texttt{eval\_rhs} and \texttt{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., \(\text{state}', \text{solve}'\) etc. for the instrumented versions, leaving the unprimed ones for the original version.

We can define a simulation relation \(\sim \subseteq \text{state} \times \text{state}'\) as the graph of the projection from \(\text{state}'\) to \(\text{state}\). We define a lifted relation \(T(\sim) \subseteq T_{\text{state}} X \times T_{\text{state}'} X\) for any \(X\) by
\[
f T(\sim) f' \equiv \forall s, s', s_1, x, x'. f(s) = (x, s_1) \land f'(s') = (x', s_1') \land (s \sim s' \implies s_1 \sim s_1' \land x = x').
\]
Functions \(f : X \rightarrow T_{\text{state}} Y\) and \(f' : X \rightarrow T_{\text{state}'} Y\) 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 \text{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 \texttt{rem\_queued}, \texttt{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 \text{Hoare} logic.

6.2. Implementation in \text{CoQ}

\text{CoQ} accepts the definition of a recursive function only if it is provably terminating. Since the algorithm \textbf{RLD} is generic, we neither make any assumptions concerning the
(semi-)lattice \( D \) (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 \( x, y : V, s, s' : \text{state}, d : D \), \( \text{Eval}(x, y, s, s', d) \) holds iff the call \( \text{eval} x \ y \ s \) terminates and returns the value \((d, s')\);
- For every \( x : V, f : V \rightarrow T^{\text{state}}_{D}, \text{Eval}_x(x, f) \) holds iff \( f \) is a total function extending the partial function \( \text{eval} x \), i.e., for every \( x, y : V, s, s' : \text{state}, d : D \), \( \text{Eval}(x, y, s, s', d) \) holds iff \( f(x) = (d, s') \);
- For every \( x : V, s, s' : \text{state}, d : D, l, l' : (V \times D)\text{list}, \text{wrap}_x(x, t, s, s', d, l, l') \) holds iff the call \( \text{eval}_x \) s terminates and returns the value \((s, d, l')\);
- For every \( x : V, s, s' : \text{state}, \text{Solve}(x, s, s') \) holds iff the call \( \text{solve} \) s terminates and returns the value \( s' \);
- For every \( X : V\text{list}, s, s' : \text{state}, \text{SolveAll}(X, s, s') \) holds iff \( \text{solve_all} X \) 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 \( \sigma : V \rightarrow D \) and traces generated by \( \sigma \) in a tree \( t \in T_{V,D} \). Given \( \sigma \), the relation \( \text{valid} \subseteq (V \times D)\text{list} \times (V \rightarrow D) \) is inductively defined by:

- \( \text{valid}([], \sigma) \);
- For any \( x \in V, d \in D \) and \( l : (V \times D)\text{list} \), if \( \text{valid}(l, \sigma) \) and \( d = \sigma x \) then \( \text{valid}(l, \sigma) \);

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

- \( \text{legal}([], l) \) for any \( l \in \text{Tree}_{V,D} \);
- For any \( x \in V, d \in D, l : (V \times D)\text{list} \) and \( c : D \rightarrow \text{Tree}_{V,D} \), if \( \text{legal}(l, c(d)) \) then \( \text{legal}(x, l, \text{Que}(x, c)) \).

Intuitively, \( \text{valid}(l, \sigma) \) holds iff the sequence \( l \) agrees with the variable assignment \( \sigma \), and \( \text{legal}(l, t) \) means that one can walk along the path \( l \) in the tree \( t \), for every \( (x, d) \) 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 \( \text{trace}_\sigma t \) is valid for \( \sigma \) and is legal in \( t \), i.e., \( \text{valid}(\text{trace}_\sigma t, \sigma) \) and \( \text{legal}(\text{trace}_\sigma t, t) \) hold for any \( t \in \text{Tree}_{V,D} \) and given \( \sigma : V \rightarrow D \).

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

- If \( l = [] \) then \( \text{subtree}(l, t) = t \);
- If \( l = (x, d) :: s \) and \( t = \text{Que}(x, c) \) then \( \text{subtree}(l, t) = \text{subtree}(s, c(d)) \).

It is not difficult to show that \( \text{subtree}(\text{trace}_\sigma t, t) = \text{Ans}(a) \) holds for every tree \( t \in \text{Tree}_{V,D} \) and variable assignment \( \sigma \). By induction on length of a path we prove the following lemma.
**Lemma 6.1.** For any given \( t \in Tree_{V,D} \), a path \( l : (V \times D) \) list and a variable assignment \( \sigma : V \rightarrow D \), the following is equivalent:

\[ l = trace_l(t) \]
\[ valid(l, \sigma), legal(l, t), subtree(l, t) = Ans(a), \text{ for some } a \in D, \text{ hold}. \]

From now on, for brevity, we denote \( \text{get\_infl} \) as \( \text{infl} \) and \( \text{get as } \sigma \). States \( s \) and \( s' \) will denote a state before a call of a function and a state after the call terminates, respectively. Structures \( \text{stable}, \text{called}, \text{queued}, \text{infl} \) and \( \sigma \) are components of the state \( s \), primed structures are components of the state \( s' \). Let \( f : V \rightarrow Tree_{V,D} \) be a given strategy function. We say that variable \( x \) is \textit{solved} in state \( s \) if \( x \in \text{stable} \setminus \text{called} \). We treat lists as sets in the formulae below.

We define

\[
I_0(s) \equiv \text{called} \subseteq \text{stable} \land \text{queued} \land \text{stable} = \emptyset,
\]
\[
I_1(s, s') \equiv \text{stable}' \supseteq \text{stable} \land \text{called}' \subseteq \text{called} \land \text{queued}' \subseteq \text{queued}.
\]

We call a state \( s \) (a transition from \( s \) to \( s' \)) \textit{consistent} if \( I_0(s) \) (respectively, \( I_1(s, s') \)) holds. Thus, \( I_0(s) \) tells that the set of \textit{called} variables is always a subset of \textit{stable} variables, and those variables that are \textit{queued} for reevaluation cannot be in \textit{stable}. \( I_1(s, s') \) asserts that for a consistent transition the set of \textit{stable} variables may only increase and the sets of \textit{called} and \textit{queued} may only shrink. For example, we will show below that \( \text{solve}(x, s, s') \) relates \( s \) and \( s' \) consistently. The formula

\[
\mathcal{I}_\sigma(s, s') \equiv \forall z \in V. \sigma_\perp s' z \supseteq \sigma_\perp s z
\]

expresses that current values of variables are large in the state \( s' \) than those in the state \( s \). The formula

\[
\mathcal{I}_{\sigma, \text{infl}}(s, s') \equiv \forall z \in V. (\sigma_\perp s' z \subseteq \sigma_\perp s z \implies \text{infl} z s \subseteq \text{infl} z s') \land
(\sigma_\perp s' z \nsubseteq \sigma_\perp s z \implies \text{infl} z s \subseteq \text{infl} z s' \setminus \text{called}')
\]

relates structures \( \sigma \) and \( \text{infl} \). For every variable \( z \), it asserts the following. If the value of \( z \) did not increase, then \( \text{infl} \) 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

\[
\mathcal{I}_{\text{dep}}(x, s) \equiv \forall z \in \text{dep}_{f,(\sigma_\perp s)} x. z \in \text{stable} \cup \text{queued} \land x \in \text{infl} z s .
\]

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

\[
\mathcal{I}_{\text{corr}}(s) \equiv \forall x \in \text{stable} \setminus \text{called}. \sigma_\perp s x \supseteq [f_x]^\ast(\sigma_\perp s) \land \mathcal{I}_{\text{dep}}(x, s)
\]

defines the correctness of the state \( s \). This means that for every variable \( x \) which is solved in \( s \), the constraint \( \sigma x \supseteq f_x \sigma \) is satisfied for \( x \) and dependencies of \( x \) are treated correctly. Namely, as implied by the \( \mathcal{I}_{\text{dep}}(x, s) \) conjunct, \( \text{infl} x \) overapproximates the set of actual dependencies of \( x \) returned by \( \text{dep}_{f,(\sigma_\perp s)} x \), 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 \( [\cdot]^\prime_{\text{state}}(\text{eval}\ x) \). The formula

\[
I_{\cdot}[x, t, s, s', d, \text{vlist}, \text{vlist}'] \equiv \\
x \in \text{stable} \land I_0(s) \land I_{\text{corr}}(s) \land (\forall (y, v) \in \text{vlist}. y \in \text{stable}) \implies \\
I_0(s') \land I_1(s, s') \land \text{vlist} \subseteq \text{vlist}' \land (\forall (y, v) \in \text{vlist}'. y \in \text{stable}) \land \\
I_{\sigma}(s, s') \land I_{\sigma, \text{infl}}(s, s') \land I_{\text{corr}}(s') \land \\
[x \in \text{called} \land (\forall (y, v) \in \text{vlist}. x \in \text{infl}(y)) \land \\
\text{valid}(\text{vlist}, \sigma) \land \text{legal}(\text{vlist}, f_x) \land \text{subtree}(\text{vlist}, f_x) = t \implies \\
(x \in \text{called}' \implies \\
\text{valid}(\text{vlist}', \sigma) \land \text{legal}(\text{vlist}', f_x) \land \text{subtree}(\text{vlist}', f_x) = \text{Ans}(d) \land \\
(\forall (y, v) \in \text{vlist}'. x \in \text{infl}(y)) \land \\
(x \notin \text{called}' \implies x \in \text{stable}' \setminus \text{called}')]
\]

relates the arguments \text{vlist} and \( s \) with the result value \(((d, \text{vlist}'), s')\) of the call \( [\cdot]^\prime_{\text{state}}(\text{eval}\ x) \) \text{vlist}\ s, if it terminates. Remind that the function proceeds recursively on the tree \( t \), taking as a parameter a list \text{vlist} of already visited variables along with their received values.

\( I_{\cdot}[x, t, s, s', d, \text{vlist}, \text{vlist}'] \) asserts that if the function \( [\cdot]^\prime_{\text{state}}(\text{eval}\ x) \) \text{vlist}\ s is invoked for a stable variable \( x \) and a partial path \text{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 \text{vlist}' that extends \text{vlist} of stable visited variables, together with a consistent correct state \( s' \). The \( I_{\sigma}(s, s') \) part tells that in result values \( \sigma x \) of all variables \( x \) grew, and \( I_{\sigma, \text{infl}}(s, s') \) guarantees that \( \text{infl} \) alters according to changes in \( \sigma \).

The invariant distinguishes the case when \( x \in \text{called} \) initially. Then if \text{vlist} is a valid and legal path in \( f_x \) leading to the subtree \( t \) and if \( x \in \text{called}' \) then the result path \text{vlist}' is again valid and legal in \( f_x \) and leads to an answer \( d \) and all the dependencies of \( x \) are recorded. Note that by lemma 6.1 this implies that \text{vlist}' is a trace in \( f_x \) by \( \sigma' \). If \( x \in \text{called} \) and \( x \notin \text{called}' \) 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 \( \sigma x \). In the case \( x \notin \text{called} \), \( x \) was solved in \( s \), and thus we deduce that \( x \) is solved in \( s' \) using \( I_1(s, s') \).

The formula

\[
I_{\text{eval}, \text{rhs}}(x, s, s', d, l') \equiv \\
x \in \text{called} \land I_0(s) \land I_{\text{corr}}(s) \implies \\
I_0(s') \land I_1(s, s') \land I_{\sigma}(s, s') \land I_{\sigma, \text{infl}}(s, s') \land I_{\text{corr}}(s') \land \\
[x \in \text{called}' \implies d = [\tilde{f}_x]''(\sigma) \land l' = \text{trace}_{\sigma'} f_x \land \\
(\forall (y, v) \in \text{vlist}'. x \in \text{infl}(y)) \land I_{\text{dep}}(x, s')]
\]

relates the arguments \( x \) and \( s \) of the call of \( \text{eval}, \text{rhs} \) \( x \ s \) 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 \( f_x \) by \( \sigma' \) and \( d \) is a value of the right-hand side of \( x \) on \( \sigma' \). The latter can be shown by lemma 6.1. In the case \( x \notin \text{called}' \), the variable \( x \) is processed during some intermediate recursive call.
and is solved in $s'$. The formula
\[
\mathcal{I}_{\text{solve}}(x, s, s') \equiv \\
\mathcal{I}_0(s) \land \mathcal{I}_{\text{corr}}(s) \implies \\
\mathcal{I}_0(s') \land \mathcal{I}_1(s, s') \land \mathcal{I}_\sigma(s, s') \land \mathcal{I}_{\text{infl}}(s, s') \land \mathcal{I}_{\text{corr}}(s') \land \\
[x \in \text{stable} \implies s = s'] \land \\
[x \not\in \text{stable} \implies \text{stable}' \supseteq \text{stable} \cup \{x\} \land \text{queued}' \subseteq \text{queued} \setminus \{x\}\]
relates arguments $x$ and $s$ with the result state $s'$ of the call of $\text{solve}$ $x$ $s$ whenever it terminates. If the state $s$ is consistent and correct then so is $s'$. In the case $x \in \text{stable}$ the state does not change. If $x \not\in \text{stable}$ then eventually $x$ is solved in $s'$ and is removed from the set $\text{queued}$. The formula
\[
\mathcal{I}_{\text{solve,all}}(w, s, s') \equiv \\
\mathcal{I}_0(s) \land \mathcal{I}_{\text{corr}}(s) \implies \\
\mathcal{I}_0(s') \land \mathcal{I}_1(s, s') \land \mathcal{I}_\sigma(s, s') \land \mathcal{I}_{\text{infl}}(s, s') \land \mathcal{I}_{\text{corr}}(s') \land \\
(w \cup \text{stable} \setminus \text{called} \subseteq \text{stable} \setminus \text{called}) \land (\text{queued}' \subseteq \text{queued} \setminus w)
\]
relates the arguments $w$ and $s$ with the result state $s'$ of the call $\text{solve,all}$ $w$ $s$ 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 $\text{queued}'$. We note that although $w = \text{infl}x$ (for corresponding $x$) may contain invalid dependencies, i.e., variables not dependent on $x$ on the current $\sigma$, $\mathcal{I}_{\text{corr}}(s')$ states that $\text{infl}x$ is appropriately recomputed.

By induction on number of unfoldings of definitions we prove in Coq that the formulae $\mathcal{I}_{\text{eval}}, \mathcal{I}_{\text{eval,x}}, \mathcal{I}_{\text{eval,rhs}}, \mathcal{I}_{\text{solve}}$ and $\mathcal{I}_{\text{solve,all}}$ are invariants of corresponding functions in the following sense.

**Theorem 6.2.** For all states $s, s'$: state the following is true:

- for every $x, y : V, d : D$, $\mathcal{E}_{\text{eval}}(x, y, s, s', d)$ implies $\mathcal{I}_{\text{eval}}(x, y, s, s', d)$;
- for every $x : V, t : \text{Tree}V, d : D, l, l' : (V \times D)\text{ list}$, $\mathcal{E}_{\text{eval-rhs}}(x, t, s, s', d, l, l')$ implies $\mathcal{I}_{\text{eval-rhs}}(x, t, s, s', d, l, l')$;
- for every $x \in V, d : D$, $\mathcal{E}_{\text{solve}}(x, s, s')$ implies $\mathcal{I}_{\text{solve}}(x, s, s')$;
- for every $w : V\text{ list}$, $\mathcal{E}_{\text{solve-all}}(w, s, s')$ implies $\mathcal{I}_{\text{solve-all}}(w, s, s')$. □

### 6.4. Putting things together

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

1. $X \subseteq \text{stable}'$ and $\text{dep}_{f, (\sigma \bot, w)}(\text{stable}') \subseteq \text{stable}'$;
2. $\sigma \bot s' \subseteq [f_X]^*(\sigma \bot s')$ holds for every $x \in \text{stable}'$.

By theorem 6.2, the invariant $\mathcal{I}_{\text{solve,all}}(X, s_{\text{init}}, s')$ holds; and its premise is true, inasmuch as both $\mathcal{I}_0(s_{\text{init}})$ and $\mathcal{I}_{\text{corr}}(s_{\text{init}})$ hold. Therefore, we have $\mathcal{I}_1(s_{\text{init}}, s')$, and hence called' = queued' = $\bot$. From $(X \cup \text{stable} \setminus \text{called} \subseteq \text{stable}' \setminus \text{called}')$ we conclude, that $X \subseteq \text{stable}'$. From $\mathcal{I}_{\text{corr}}(s')$ it follows, that $\forall x \in \text{stable}'$. $\sigma \bot s' \subseteq [f_X]^*(\sigma \bot s')$ and $\text{dep}_{f, (\sigma \bot, s')}(\text{stable}') \subseteq \text{stable}'$ 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, \textbf{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 \( \bot \sqsubset a, b \sqsubset \top \), and \( a \) and \( b \) are incomparable. For example, the third tree on the figure may represent a right-hand side

\[
\begin{aligned}
f_x = \text{fun } \sigma \rightarrow \\
\text{let } s_1 = \sigma \text{ s in} \\
\text{let } u_1 = \sigma \text{ u in} \\
\text{if } u_1 \sqsubseteq \bot \text{ then} \\
\text{let } v_1 = \sigma \text{ v in} \\
\text{let } u_2 = \sigma \text{ u in} \\
\text{if } u_2 \sqsubseteq \bot \text{ then } v_1 \sqcup a \text{ else } b \\
\text{else} \\
\text{let } v_1 = \sigma \text{ v in } v_1 \sqcup a
\end{aligned}
\]

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

\[
\sigma_0(x) = a, \quad x \in \{ t, s, x, u, v \}
\]

Will \textbf{RLD} return an exact solution?

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

1. For recomputation of \( x \), \texttt{solve} \( x \) is called, and variable \( x \) is put back into \texttt{stable} and \texttt{called}. The state just prior to reevaluation of the right-hand side \( f_x \) (the call of \texttt{eval\_rhs} \( x \)) is as follows:

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

During the run of \texttt{eval\_rhs} \( x \) the algorithm traverses the tree \( f_x \) and recomputes necessary variables as described below.

- Since \( s, u \in \texttt{stable} \), the algorithm does not descend into solving them. The structures \( \sigma, \texttt{stable} \) and \texttt{called} are not changed, but the solver records that \( x \) depends on \( s \) and \( u \), i.e., \( x \) is added to \texttt{infl} \( s \) and \texttt{infl} \( u \).

- Since \( \sigma(u) = \bot \), the algorithm takes the upper branch of (1) in \( f_x \). Thus, it recomputes \( v \), which gets a larger value \( a \) (as \( \sigma(s) = a \)). Since the value of \( v \) has changed, variables influenced by \( v \) might need to be recomputed before \texttt{solve} \( v \) returns. At this point, these are variables from \texttt{infl} \( v = [x; u; s] \). Those are removed from the sets \texttt{stable} and \texttt{called}, and get recomputed by calling consequently \texttt{solve} \( x \), \texttt{solve} \( u \), and...
solve $s$. At the end end of their run, the altered state components are (infl structure is omitted)

$$
stable = \{s, t, u, v, x\}, \quad called = \{t\},
$$

$$
\sigma(s) = \sigma(u) = \sigma(v) = \sigma(x) = a, \quad \sigma(t) = \bot.
$$

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

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

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

Finally, the algorithm terminates and returns a solution $\sigma_1$ with $\sigma_1(x) = \top$, and thus $\sigma_1$ is not minimal. The reason why the “bad” value $b$ is reached during the run of the solver is that $u$ changes its value between branchings (1) and (2). Note that the leaf $b$ is not reachable in $[\bar{f}_x]_{\text{unit}}(\text{val}_{\text{unit}} \circ \sigma)$, for any variable assignment $\sigma$. 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 $v$ is queried at most once. The property does not hold for the tree $\bar{f}_x$ as on Fig. 6, since there exists a path on which the variable $u$ is queried twice. We prove:

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

**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.

$$
I'_{\text{solve}\ all}(x, s, s') \equiv \text{UniqueLookup}(\bar{f}) \land \text{Monotone}(\bar{f}) \implies \forall \mu : V \rightarrow D. \mu x \sqsubseteq [\bar{f}_x]^{\mu} s \sqsubseteq \mu \implies \sigma_+ s' \sqsubseteq \mu,
$$

where $\sigma_+ s$ denotes the function $get\ s$. The invariant states that, given any solution $\mu$, if $\sigma_+$ is (pointwise) less than $\mu$ 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 $s_{\text{init}}$ be an initial state with $\text{stable} = \text{called} = \text{queued} = \emptyset$, $\sigma = \text{infl} = \emptyset$, and let $\mu$ be the least solution of $S$. Assume that $\text{RLD}$ applied to $(f, X)$ terminates and let $s'$ be the state returned by the call $\text{solve}_{\text{all}} X s_{\text{init}}$. Since $\sigma_{\|} s_{\text{init}} X = \bot \sqsubseteq \mu X$, for all $x$, we have $\sigma_{\|} s' \sqsubseteq \mu$. Let $\sigma'_{X'}$ be an extension of $\sigma_{\|} s'$ to a global solution as defined in Section 4.2. Since $X'$ is $\text{dep}$-closed, we have

$$\lbrack f_x \rbrack (\sigma_{\|} s') = \lbrack f_x \rbrack \sigma'_{X'}, \quad \text{for all } x \in X'.$$

Using monotonicity of $f$, for every $x \in X'$ we get $\mu x \sqsupseteq \sigma_{\|} s' x \sqsupseteq \lbrack f_x \rbrack (\sigma_{\|} s') = \lbrack f_x \rbrack \sigma'_{X'}, \quad \lbrack f_x \rbrack \mu = \mu x$. Therefore, $\mu x = \lbrack f_x \rbrack (\sigma_{\|} s')$ holds, for all $x \in X'$. This proves the theorem. $\square$

8. THE SOLVER $\text{RLDE}$

In this section, we present a modification of the solver $\text{RLD}$ which is exact.

The idea of improvement comes from a careful inspection of behavior of the instrumented $\text{RLD}$. In the previous example, we observed that, for variable $x$, during a call to $\text{solve} x$ after $\text{eval}_x x$ returns, $x$ may not belong to $\text{called}$. This may happen in the presence of cyclic variable dependencies — for example, when $x$ depends on some $v$ (met in $f_x$), which gets a strictly larger value while evaluating $\text{eval}_x x$. In this case, a recursive call to $\text{solve} x$ is triggered in order to recompute $x$. The invariant $I_{\text{solve}}$ guarantees that $x$ is solved after the recursive call to $\text{solve}$ returns, i.e., $x \in \text{stable} \setminus \text{called}$ 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 $f_x$ by any stateless variable assignment $\sigma$.

The idea, therefore, is to check whether $x$ is still in $\text{called}$ before updating it. To implement this idea, we adjust the original $\text{RLD}$ in the following way (Fig. 7). The structure $\text{called}$ 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 $\text{RLDE}$ 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 $\text{RLDE}$ 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. $\square$

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 $\text{RLD}$ ($\text{RLDE}$) when called with $(f, X)$, 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 $x \in \text{infl} z$. That contradicts the ascending chain condition of $\mathbb{D}$.

Suppose, that the algorithm does not terminate for $(f, X)$. Since strategy trees are well-founded (by definition), function $\text{solve}$ must be called infinitely many times. Therefore, there exist a variable $x \in V$ such that $\text{solve} x$ is triggered infinitely often. Note that $\text{solve} x$ is executed only either when evaluating right-hand side $\text{eval}_y y$, for some variable $y$ (such that $x$ is queried in $f_y$), or when $x$ is destabilized. Since $x$ can be destabilized at most finitely many times, there exists a variable $y$ such that $\text{solve} x$
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 in-
finity often, since solve y when called with y ∈ 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 finite-
ness of an equation system, the solvers may not terminate in general. However, as
we demonstrate, one can guarantee termination of RLD (RLDE) under certain condi-
tions.

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
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
On the Verification of Local Generic Solvers


