

# Partial Correctness of the Top-Down Solver

Yannick Stade\*, Sarah Tilscher\*, Helmut Seidl

February 6, 2026

## Abstract

The top-down solver (TD) is a local and generic fixpoint algorithm used for abstract interpretation. Being local means it only evaluates equations required for the computation of the value of some initially queried unknown, while being generic means that it is applicable for arbitrary equation systems where right-hand sides are considered as black-box functions. To avoid unnecessary evaluations of right-hand sides, the TD collects stable unknowns that need not be re-evaluated. This optimization requires the additional tracking of dependencies between unknowns and a non-local destabilization mechanism to assure the re-evaluation of previously stable unknowns that were affected by a changed value.

Due to the recursive evaluation strategy and the non-local destabilization mechanism of the TD, its correctness is non-obvious. To provide a formal proof of its partial correctness, we employ the insight that the TD can be considered an optimized version of a considerably simpler recursive fixpoint algorithm. Following this insight, we first prove the partial correctness of the simpler recursive fixpoint algorithm, the plain TD. Then, we transfer the statement of partial correctness to the TD by establishing the equivalence of both algorithms concerning both their termination behavior and their computed result.

---

\*The first two authors contributed equally to this research and are ordered alphabetically.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Preliminaries</b>	<b>4</b>
2.1	Strategy Trees	4
2.2	Auxiliary Lemmas for Default Maps	4
2.3	Functions on the Constraint System	6
2.4	Subtrees of Strategy Trees	6
2.5	Dependencies between Unknowns	7
2.6	Set <i>Reach</i>	8
2.7	Partial solution	9
<b>3</b>	<b>The plain Top-Down Solver</b>	<b>10</b>
3.1	Definition of the Solver Algorithm	10
3.2	Refinement of Auto-Generated Rules	11
3.3	Domain Lemmas	12
3.4	Case Rules	12
3.5	Predicate for Valid Input States	13
3.6	Partial Correctness Proofs	14
3.7	Termination of TD_plain for Stable Unknowns	15
3.8	Program Refinement for Code Generation	16
<b>4</b>	<b>The Top-Down Solver</b>	<b>17</b>
4.1	Definition of Destabilize and Proof of its Termination	18
4.2	Definition of the Solver Algorithm	19
4.3	Refinement of Auto-Generated Rules	21
4.4	Domain Lemmas	21
4.5	Case Rules	22
4.6	Description of the Effect of Destabilize	23
4.7	Predicate for Valid Input States	24
4.8	Auxiliary Lemmas for Partial Correctness Proofs	25
4.9	Preservation of the Invariant	26
4.10	TD_plain and TD Equivalence	26
4.11	Partial Correctness of the TD	29
4.12	Program Refinement for Code Generation	29
<b>5</b>	<b>Example</b>	<b>31</b>
5.1	Definition of the Domain	32
5.2	Definition of the Equation System	33
5.3	Solve the Equation System with TD_plain	33
5.4	Solve the Equation System with TD	34

# 1 Introduction

Static analysis of programs based on abstract interpretation requires efficient and reliable fixpoint engines [1]. In this work, we focus on the top-down solver (TD) [3]—a generic fixpoint algorithm that can handle arbitrary equation systems, even those with infinitely many equations. The latter is achieved by a property called *local*: When the TD is invoked to compute the value of some unknown, it recursively descends only into those unknowns on which the initially queried unknown depends. In order to avoid redundant re-evaluations of equations, the TD maintains a set of stable unknowns whose re-evaluation can be replaced by a simple lookup. Removing unknowns from the set of stable unknowns when they are possibly affected by changes to other unknowns, requires information about dependencies between unknowns. These dependencies need not be provided beforehand but are detected through self-observation on the fly. This makes the TD suitable also for equation systems where dependencies change dynamically during the solver’s computation.

By removing the collecting of stable unknowns and dependency tracking, we obtain a stripped version of the TD, which we call the *plain TD*. The plain TD is capable of solving the same equation systems as the original TD and also shares the same termination behavior, but also re-evaluates those unknowns that have already been evaluated and whose value could just be looked up. In the first part of this work, we show the partial correctness of the plain TD. We use a mutual induction following its computation trace to establish invariants describing a valid solver state. From this, the partial correctness of the solver’s result can be derived. The proof is described in [Section 3](#).

We then recover the original TD from the plain TD and prove the equivalence between the two, i. e., that they share the same termination behavior and return the same result whenever they terminate. This way, the partial correctness statement from the plain TD is shown to carry over to the original TD. The essential part of this proof is twofold: First, we extend the invariants to describe the additional data structures for collecting stable unknowns and the dependencies between unknowns. Second, we show that the destabilization of an unknown preserves those invariants. The corresponding proofs are outlined in [Section 4](#).

We conclude this work with an example in [Section 5](#) showing the application of the TD to a simple equation system derived from a program for the analysis of must-be initialized variables.

## 2 Preliminaries

Before we define the TD in Isabelle/HOL and start with its partial correctness proof, we define all required data structures, formalize definitions and prove auxiliary lemmas.

```
theory Basics
  imports Main "HOL-Library.Finite_Map"
begin
```

```
unbundle lattice_syntax
```

### 2.1 Strategy Trees

The constraint system is a function mapping each unknown to a right-hand side to compute its value. We require the right-hand sides to be pure functionals [2]. This means that they may query the values of other unknowns and perform additional computations based on those, but they may, e.g., not spy on the solver's data structures. Such pure functions can be expressed as strategy trees.

```
datatype ('a, 'b) strategy_tree = Answer 'b |
  Query 'a "'b  $\Rightarrow$  ('a, 'b) strategy_tree"
```

The solver is defined based on a black-box function  $T$  describing the constraint system and under the assumption that the special element  $\perp$  exists among the values.

```
locale Solver =
  fixes D :: "'d :: bot"
  and T :: "'x  $\Rightarrow$  ('x, 'd) strategy_tree"
begin
```

### 2.2 Auxiliary Lemmas for Default Maps

The solver maintains a solver state to implement optimizations based on self-observation. Among the data structures for the solver state are maps that return a default value for non-existing keys. In the following, we define some helper functions and lemmas for these.

```
definition fmlookup_default where
  "fmlookup_default m d x = (case fmlookup m x of Some v  $\Rightarrow$  v | None  $\Rightarrow$ 
d)"
```

```
abbreviation slookup where
  "slookup infl x  $\equiv$  set (fmlookup_default infl [] x)"
```

```
definition mlup where
  "mlup  $\sigma$  x  $\equiv$  case  $\sigma$  x of Some v  $\Rightarrow$  v | None  $\Rightarrow$   $\perp$ "
```

```

definition fminsert where
  "fminsert infl x y = fmupd x (y # (fmlookup_default infl [] x)) infl"

lemma set_fmlookup_default_cases:
  assumes "y ∈ slookup infl x"
  obtains (1) xs where "fmlookup infl x = Some xs" and "y ∈ set xs"
  ⟨proof⟩

lemma notin_fmlookup_default_cases:
  assumes "y ∉ slookup infl x"
  obtains (1) xs where "fmlookup infl x = Some xs" and "y ∉ set xs"
  | (2) "fmlookup infl x = None"
  ⟨proof⟩

lemma slookup_helper[simp]:
  assumes "fmlookup m x = Some ys"
  and "y ∈ set ys"
  shows "y ∈ slookup m x"
  ⟨proof⟩

lemma lookup_implies_mlup:
  assumes "σ x = σ' x'"
  shows "mlup σ x = mlup σ' x'"
  ⟨proof⟩

lemma fmlookup_fminsert:
  assumes "fmlookup_default infl [] x = xs"
  shows "fmlookup (fminsert infl x y) x = Some (y # xs)"
  ⟨proof⟩

lemma fmlookup_fminsert':
  obtains xs ys
  where "fmlookup (fminsert infl x y) x = Some xs"
  and "fmlookup_default infl [] x = ys" and "xs = y # ys"
  ⟨proof⟩

lemma fmlookup_default_drop_set:
  "fmlookup_default (fmdrop_set A m) [] x = (if x ∉ A then fmlookup_default
m [] x else [])"
  ⟨proof⟩

lemma mlup_eq_mupd_set:
  assumes "x ∉ s"
  and "∀y∈s. mlup σ y = mlup σ' y"
  shows "∀y∈s. mlup σ y = mlup (σ'(x ↦ xd)) y"
  ⟨proof⟩

```

## 2.3 Functions on the Constraint System

The function `rhs_length` computes the length of a specific path in the strategy tree defined by a value assignment for unknowns  $\sigma$ .

```
function (domintros) rhs_length where
  "rhs_length (Answer d) _ = 0" |
  "rhs_length (Query x f)  $\sigma$  = 1 + rhs_length (f (mlup  $\sigma$  x))  $\sigma$ "
  <proof>
```

```
termination rhs_length
  <proof>
```

The function `traverse_rhs` traverses a strategy tree and determines the answer when choosing the path through the strategy tree based on a given unknown-value mapping  $\sigma$

```
function (domintros) traverse_rhs where
  "traverse_rhs (Answer d) _ = d" |
  "traverse_rhs (Query x f)  $\sigma$  = traverse_rhs (f (mlup  $\sigma$  x))  $\sigma$ "
  <proof>
```

```
termination traverse_rhs
  <proof>
```

The function `eq` evaluates the right-hand side of an unknown  $x$  with an unknown-value mapping  $\sigma$ .

```
definition eq :: "'x  $\Rightarrow$  ('x, 'd) map  $\Rightarrow$  'd" where
  "eq x  $\sigma$  = traverse_rhs (T x)  $\sigma$ "
declare eq_def[simp]
```

## 2.4 Subtrees of Strategy Trees

We define the set of subtrees of a strategy tree for a specific path (defined through  $\sigma$ ).

```
inductive_set subt_aux ::
  "('x, 'd) map  $\Rightarrow$  ('x, 'd) strategy_tree  $\Rightarrow$  ('x, 'd) strategy_tree
  set" for  $\sigma$  t where
  base: "t  $\in$  subt_aux  $\sigma$  t"
  | step: "t'  $\in$  subt_aux  $\sigma$  t  $\implies$  t' = Query y g  $\implies$  (g (mlup  $\sigma$  y))  $\in$  subt_aux
   $\sigma$  t"
```

```
definition subt where
  "subt  $\sigma$  x = subt_aux  $\sigma$  (T x)"
```

```
lemma subt_of_answer_singleton:
  shows "subt_aux  $\sigma$  (Answer d) = {Answer d}"
  <proof>
```

```

lemma subt_transitive:
  assumes "t' ∈ subt_aux σ t"
  shows "subt_aux σ t' ⊆ subt_aux σ t"
  ⟨proof⟩

```

```

lemma subt_unfold:
  shows "subt_aux σ (Query x f) = insert (Query x f) (subt_aux σ (f (mlup
σ x)))"
  ⟨proof⟩

```

## 2.5 Dependencies between Unknowns

The set  $dep\ \sigma\ x$  collects all unknowns occurring in the right-hand side of  $x$  when traversing it with  $\sigma$ .

```

function dep_aux where
  "dep_aux σ (Answer d) = {}"
| "dep_aux σ (Query y g) = insert y (dep_aux σ (g (mlup σ y)))"
  ⟨proof⟩

```

```

termination dep_aux
  ⟨proof⟩

```

```

definition dep where
  "dep σ x = dep_aux σ (T x)"

```

```

lemma dep_aux_eq:
  assumes "∀y ∈ dep_aux σ t. mlup σ y = mlup σ' y"
  shows "dep_aux σ t = dep_aux σ' t"
  ⟨proof⟩

```

**lemmas**  $dep\_eq = dep\_aux\_eq[of\ \sigma\ "T\ x"\ \sigma']$  for  $\sigma\ x\ \sigma'$ , folded *dep\_def*

```

lemma subt_implies_dep:
  assumes "Query y g ∈ subt_aux σ t"
  shows "y ∈ dep_aux σ t"
  ⟨proof⟩

```

```

lemma solution_sufficient:
  assumes "∀y ∈ dep σ x. mlup σ y = mlup σ' y"
  shows "eq x σ = eq x σ'"
  ⟨proof⟩

```

```

corollary eq_mupd_no_dep:
  assumes "x ∉ dep σ y"
  shows "eq y σ = eq y (σ (x ↦ xd))"
  ⟨proof⟩

```

## 2.6 Set Reach

Let *reach* be the set of all unknowns contributing to  $x$  (for a given  $\sigma$ ). This corresponds to the set of all unknowns on which  $x$  transitively depends on when evaluating the necessary right-hand sides with  $\sigma$ .

**inductive\_set reach for  $\sigma$   $x$  where**

*base:* " $x \in \text{reach } \sigma \ x$ "

*| step:* " $y \in \text{reach } \sigma \ x \implies z \in \text{dep } \sigma \ y \implies z \in \text{reach } \sigma \ x$ "

The solver stops descending when it encounters an unknown whose evaluation it has already started (i.e. an unknown in  $c$ ). Therefore, *reach* might collect contributing unknowns which the solver did not descend into. For a predicate, that relates more closely to the solver's history, we define the set *reach\_cap*. Similarly to *reach* it collects the unknowns on which an unknown transitively depends, but only until an unknown in  $c$  is reached.

**inductive\_set reach\_cap\_tree for  $\sigma$   $c$   $t$  where**

*base:* " $x \in \text{dep\_aux } \sigma \ t \implies x \in \text{reach\_cap\_tree } \sigma \ c \ t$ "

*| step:* " $y \in \text{reach\_cap\_tree } \sigma \ c \ t \implies y \notin c \implies z \in \text{dep } \sigma \ y \implies z \in \text{reach\_cap\_tree } \sigma \ c \ t$ "

**abbreviation "reach\_cap  $\sigma$   $c$   $x$**

$\equiv \text{insert } x \ (\text{if } x \in c \ \text{then } \{\} \ \text{else } \text{reach\_cap\_tree } \sigma \ (\text{insert } x \ c) \ (T \ x))$ "

**lemma reach\_cap\_tree\_answer\_empty[simp]:**

"reach\_cap\_tree  $\sigma$   $c$  (Answer  $d$ ) = {}"

*<proof>*

**lemma dep\_subset\_reach\_cap\_tree:**

"dep\_aux  $\sigma'$   $t \subseteq \text{reach\_cap\_tree } \sigma' \ c \ t$ "

*<proof>*

**lemma reach\_cap\_tree\_subset:**

shows "reach\_cap\_tree  $\sigma$   $c \ t \subseteq \text{reach\_cap\_tree } \sigma \ (c - \{x\}) \ t$ "

*<proof>*

**lemma reach\_empty\_capped:**

shows "reach  $\sigma \ x = \text{insert } x \ (\text{reach\_cap\_tree } \sigma \ \{x\} \ (T \ x))$ "

*<proof>*

**lemma dep\_aux\_implies\_reach\_cap\_tree:**

assumes " $y \notin c$ "

and " $y \in \text{dep\_aux } \sigma \ t$ "

shows "reach\_cap\_tree  $\sigma \ c \ (T \ y) \subseteq \text{reach\_cap\_tree } \sigma \ c \ t$ "

*<proof>*

**lemma reach\_cap\_tree\_simp:**

shows "reach\_cap\_tree  $\sigma \ c \ t$ "

$= \text{dep\_aux } \sigma \ t \cup (\bigcup \xi \in \text{dep\_aux } \sigma \ t - c. \text{reach\_cap\_tree } \sigma \ (\text{insert } \xi \ c) \ (T \ \xi))"$   
 $\langle \text{proof} \rangle$

**lemma reach\_cap\_tree\_step:**  
 assumes "mlup  $\sigma \ y = yd$ "  
 shows "reach\_cap\_tree  $\sigma \ c \ (\text{Query } y \ g) = \text{insert } y \ (\text{if } y \in c \ \text{then } \{\} \ \text{else } \text{reach\_cap\_tree } \sigma \ (\text{insert } y \ c) \ (T \ y)) \cup \text{reach\_cap\_tree } \sigma \ c \ (g \ yd)"$   
 $\langle \text{proof} \rangle$

**lemma reach\_cap\_tree\_eq:**  
 assumes " $\forall x \in \text{reach\_cap\_tree } \sigma \ c \ t. \text{mlup } \sigma \ x = \text{mlup } \sigma' \ x$ "  
 shows "reach\_cap\_tree  $\sigma \ c \ t = \text{reach\_cap\_tree } \sigma' \ c \ t$ "  
 $\langle \text{proof} \rangle$

**lemma reach\_cap\_tree\_simp2:**  
 shows "insert  $x \ (\text{if } x \in c \ \text{then } \{\} \ \text{else } \text{reach\_cap\_tree } \sigma \ c \ (T \ x)) = \text{insert } x \ (\text{if } x \in c \ \text{then } \{\} \ \text{else } \text{reach\_cap\_tree } \sigma \ (\text{insert } x \ c) \ (T \ x))"$   
 $\langle \text{proof} \rangle$

**lemma dep\_closed\_implies\_reach\_cap\_tree\_closed:**  
 assumes " $x \in s$ "  
 and " $\forall \xi \in s - (c - \{x\}). \text{dep } \sigma' \ \xi \subseteq s$ "  
 shows "reach\_cap  $\sigma' \ (c - \{x\}) \ x \subseteq s$ "  
 $\langle \text{proof} \rangle$

**lemma reach\_cap\_tree\_subset2:**  
 assumes "mlup  $\sigma \ y = yd$ "  
 shows "reach\_cap\_tree  $\sigma \ c \ (g \ yd) \subseteq \text{reach\_cap\_tree } \sigma \ c \ (\text{Query } y \ g)"$   
 $\langle \text{proof} \rangle$

**lemma reach\_cap\_tree\_subset\_subt:**  
 assumes " $t' \in \text{subt\_aux } \sigma \ t$ "  
 shows "reach\_cap\_tree  $\sigma \ c \ t' \subseteq \text{reach\_cap\_tree } \sigma \ c \ t$ "  
 $\langle \text{proof} \rangle$

**lemma reach\_cap\_tree\_singleton:**  
 assumes "reach\_cap\_tree  $\sigma \ (\text{insert } x \ c) \ t \subseteq \{x\}"$   
 obtains (Answer)  $d$  where " $t = \text{Answer } d$ "  
 | (Query)  $f$  where " $t = \text{Query } x \ f$ "  
 and "dep\_aux  $\sigma \ t = \{x\}"$   
 $\langle \text{proof} \rangle$

## 2.7 Partial solution

Finally, we define an unknown-to-value mapping  $\sigma$  to be a partial solution over a set of unknowns  $\text{vars}$  if for every unknown in  $\text{vars}$ , the value obtained

from an evaluation of its right-hand side function `eq x` with  $\sigma$  matches the value stored in  $\sigma$ .

**abbreviation** `part_solution where`

`"part_solution  $\sigma$  vars  $\equiv (\forall x \in vars. eq\ x\ \sigma = mlup\ \sigma\ x)"$`

**lemma** `part_solution_coinciding_sigma_called:`

`assumes "part_solution  $\sigma$  (s - c)"`  
`and " $\forall x \in s. mlup\ \sigma\ x = mlup\ \sigma'\ x"$`   
`and " $\forall x \in s - c. dep\ \sigma\ x \subseteq s"$`   
`shows "part_solution  $\sigma'$  (s - c)"`  
`<proof>`

**end**

**end**

### 3 The plain Top-Down Solver

`TD_plain` is a simplified version of the original TD which only keeps track of already called unknowns to avoid infinite descend in case of recursive dependencies. In contrast to the TD, it does, however, not track stable unknowns and the dependencies between unknowns. Instead, it re-iterates every unknown when queried again.

**theory** `TD_plain`  
**imports** `Basics`  
**begin**

**locale** `TD_plain = Solver D T`  
`for D :: "'d :: bot"`  
`and T :: "'x  $\Rightarrow$  ('x, 'd) strategy_tree"`  
**begin**

#### 3.1 Definition of the Solver Algorithm

The recursively descending solver algorithm is defined with three mutual recursive functions. Initially, the function `iterate` is called from the top-level `solve` function for the requested unknown. `iterate` keeps evaluating the right-hand side by calling the function `eval` and updates the value mapping  $\sigma$  until the value stabilizes. The function `eval` walks through a strategy tree and chooses the path based on the result for queried unknowns. These queries are delegated to the third mutual recursive function `query` which checks that the unknown is not already being evaluated and iterates it otherwise. The function keyword is used for the definition, since, without further assumptions, the solver may not terminate.

**function** `(domintros)`

```

    query :: "'x ⇒ 'x ⇒ 'x set ⇒ ('x, 'd) map ⇒ 'd × ('x, 'd) map"
and
    iterate :: "'x ⇒ 'x set ⇒ ('x, 'd) map ⇒ 'd × ('x, 'd) map" and
    eval :: "'x ⇒ ('x, 'd) strategy_tree ⇒ 'x set ⇒ ('x, 'd) map ⇒
'd × ('x, 'd) map" where
    "query x y c σ = (
      if y ∈ c then
        (mlup σ y, σ)
      else
        iterate y (insert y c) σ)"
| "iterate x c σ = (
  let (d_new, σ) = eval x (T x) c σ in
  if d_new = mlup σ x then
    (d_new, σ)
  else
    iterate x c (σ(x ↦ d_new)))"
| "eval x t c σ = (case t of
  Answer d ⇒ (d, σ)
  | Query y g ⇒ (let (yd, σ) = query x y c σ in eval x (g yd) c σ))"
⟨proof⟩

```

```

definition solve :: "'x ⇒ ('x, 'd) map" where
  "solve x = (let (_, σ) = iterate x {x} Map.empty in σ)"

```

```

definition query_dom where
  "query_dom x y c σ = query_iterate_eval_dom (Inl (x, y, c, σ))"
declare query_dom_def [simp]
definition iterate_dom where
  "iterate_dom x c σ = query_iterate_eval_dom (Inr (Inl (x, c, σ)))"
declare iterate_dom_def [simp]
definition eval_dom where
  "eval_dom x t c σ = query_iterate_eval_dom (Inr (Inr (x, t, c, σ)))"
declare eval_dom_def [simp]

```

```

definition solve_dom where
  "solve_dom x = iterate_dom x {x} Map.empty"

```

```

lemmas dom_defs = query_dom_def iterate_dom_def eval_dom_def

```

### 3.2 Refinement of Auto-Generated Rules

The auto-generated `pinduct` rule contains a redundant assumption. This lemma removes this redundant assumption for easier instantiation and assigns each case a comprehensible name.

```

lemmas query_iterate_eval_pinduct[consumes 1, case_names Query Iterate Eval]
= query_iterate_eval.pinduct(1)[
  folded query_dom_def iterate_dom_def eval_dom_def,
  of x y c σ for x y c σ

```

```

]
query_iterate_eval.pinduct(2)[
  folded query_dom_def iterate_dom_def eval_dom_def,
  of x c  $\sigma$  for x c  $\sigma$ 
]
query_iterate_eval.pinduct(3)[
  folded query_dom_def iterate_dom_def eval_dom_def,
  of x t c  $\sigma$  for x t c  $\sigma$ 
]

lemmas iterate_pinduct[consumes 1, case_names Iterate]
= query_iterate_eval_pinduct(2)[where ?P="λx y c  $\sigma$ . True" and ?R="λx
t c  $\sigma$ . True",
  simplified (no_asm_use), folded query_dom_def iterate_dom_def eval_dom_def]

declare query.psimps [simp]
declare iterate.psimps [simp]
declare eval.psimps [simp]

```

### 3.3 Domain Lemmas

```

lemma dom_backwards_pinduct:
  shows "query_dom x y c  $\sigma$ 
 $\implies$  y  $\notin$  c  $\implies$  iterate_dom y (insert y c)  $\sigma$ "
  and "iterate_dom x c  $\sigma$ 
 $\implies$  (eval_dom x (T x) c  $\sigma$   $\wedge$ 
  (eval x (T x) c  $\sigma$  = (xd_new,  $\sigma'$ )
 $\longrightarrow$  mlup  $\sigma'$  x = xd_old  $\longrightarrow$  xd_new  $\neq$  xd_old  $\longrightarrow$ 
  iterate_dom x c ( $\sigma'(x \mapsto xd\_new)$ )))"
  and "eval_dom x (Query y g) c  $\sigma$ 
 $\implies$  (query_dom x y c  $\sigma$   $\wedge$  (query x y c  $\sigma$  = (yd,  $\sigma'$ )  $\longrightarrow$  eval_dom x
(g yd) c  $\sigma'$ ))"
<proof>

```

### 3.4 Case Rules

```

lemma iterate_continue_fixpoint_cases[consumes 3]:
  assumes "iterate_dom x c  $\sigma$ "
  and "iterate x c  $\sigma$  = (xd,  $\sigma'$ )"
  and "x  $\in$  c"
  obtains (Fixpoint) "eval_dom x (T x) c  $\sigma$ "
  and "eval x (T x) c  $\sigma$  = (xd,  $\sigma'$ )"
  and "mlup  $\sigma'$  x = xd"
  | (Continue)  $\sigma_1$  xd_new
  where "eval_dom x (T x) c  $\sigma$ "
  and "eval x (T x) c  $\sigma$  = (xd_new,  $\sigma_1$ )"
  and "mlup  $\sigma_1$  x  $\neq$  xd_new"
  and "iterate_dom x c ( $\sigma_1(x \mapsto xd\_new)$ )"
  and "iterate x c ( $\sigma_1(x \mapsto xd\_new)$ ) = (xd,  $\sigma'$ )"
<proof>

```

```

lemma iterate_fmlookup:
  assumes "iterate_dom x c  $\sigma$ "
    and "iterate x c  $\sigma = (xd, \sigma')$ "
    and " $x \in c$ "
  shows "mlup  $\sigma' x = xd$ "
  <proof>

corollary query_fmlookup:
  assumes "query_dom x y c  $\sigma$ "
    and "query x y c  $\sigma = (yd, \sigma')$ "
  shows "mlup  $\sigma' y = yd$ "
  <proof>

lemma query_iterate_lookup_cases [consumes 2]:
  assumes "query_dom x y c  $\sigma$ "
    and "query x y c  $\sigma = (yd, \sigma')$ "
  obtains (Iterate)
    "iterate_dom y (insert y c)  $\sigma$ "
    and "iterate y (insert y c)  $\sigma = (yd, \sigma')$ "
    and "mlup  $\sigma' y = yd$ "
    and " $y \notin c$ "
  | (Lookup) "mlup  $\sigma y = yd$ "
    and " $\sigma = \sigma'$ "
    and " $y \in c$ "
  <proof>

lemma eval_query_answer_cases [consumes 2]:
  assumes "eval_dom x t c  $\sigma$ "
    and "eval x t c  $\sigma = (d, \sigma')$ "
  obtains (Query) y g yd  $\sigma1$ 
  where "t = Query y g"
    and "query_dom x y c  $\sigma$ "
    and "query x y c  $\sigma = (yd, \sigma1)$ "
    and "eval_dom x (g yd) c  $\sigma1$ "
    and "eval x (g yd) c  $\sigma1 = (d, \sigma')$ "
    and "mlup  $\sigma1 y = yd$ "
  | (Answer) "t = Answer d"
    and " $\sigma = \sigma'$ "
  <proof>

```

### 3.5 Predicate for Valid Input States

We define a predicate for valid input solver states.  $c$  is the set of called unknowns, i.e., the unknowns currently being evaluated and  $\sigma$  is an unknown-to-value mapping. Both are data structures maintained by the solver. In contrast, the parameter  $s$  describing a set of unknowns, for which a partial solution has already been computed or which are currently being evaluated,

is introduced for the proof. Although it is similar to the set `stabl` maintained by the original TD, it is only an under-approximation of it. A valid solver state is one, where  $\sigma$  is a partial solution for all truly stable unknowns, i.e., unknowns in  $s - c$ , and where these truly stable unknowns only depend on unknowns which are also truly stable or currently being evaluated. A substantial part of the partial correctness proof is to show that this property about the solver's state is preserved during a solver's run.

**definition invariant where**

```
"invariant s c  $\sigma \equiv (\forall \xi \in s - c. \text{dep } \sigma \xi \subseteq s) \wedge \text{part\_solution } \sigma (s - c)"$ 
```

**lemma invariant\_simp:**

```
assumes "x  $\in c$ "
  and "invariant s (c - {x})  $\sigma$ "
shows "invariant (insert x s) c  $\sigma$ "
<proof>
```

**lemma invariant\_continue:**

```
assumes "x  $\notin s$ "
  and "invariant s c  $\sigma$ "
  and " $\forall y \in s. \text{mlup } \sigma y = \text{mlup } \sigma 1 y$ "
shows "invariant s c ( $\sigma 1(x \mapsto xd)$ )"
<proof>
```

### 3.6 Partial Correctness Proofs

**lemma x\_not\_stable:**

```
assumes "eq x  $\sigma \neq \text{mlup } \sigma x$ "
  and "part_solution  $\sigma s$ "
shows "x  $\notin s$ "
<proof>
```

With the following lemma we establish, that whenever the solver is called for an unknown in  $s$  and where the solver state and  $s$  fulfill the invariant, the output value mapping is unchanged compared to the input value mapping.

**lemma already\_solution:**

```
shows "query_dom x y c  $\sigma$ 
 $\implies \text{query } x y c \sigma = (yd, \sigma')$ 
 $\implies y \in s$ 
 $\implies \text{invariant } s c \sigma$ 
 $\implies \sigma = \sigma'$ "
and "iterate_dom x c  $\sigma$ 
 $\implies \text{iterate } x c \sigma = (xd, \sigma')$ 
 $\implies x \in c$ 
 $\implies x \in s$ 
 $\implies \text{invariant } s (c - \{x\}) \sigma$ 
 $\implies \sigma = \sigma'$ "
and "eval_dom x t c  $\sigma$ 
```

```

 $\Rightarrow$  eval x t c  $\sigma$  = (xd,  $\sigma'$ )
 $\Rightarrow$  dep_aux  $\sigma$  t  $\subseteq$  s
 $\Rightarrow$  invariant s c  $\sigma$ 
 $\Rightarrow$  traverse_rhs t  $\sigma'$  = xd  $\wedge$   $\sigma$  =  $\sigma'$ "

```

*<proof>*

Furthermore, we show that whenever the solver is called with a valid solver state, the valid solver state invariant also holds for its output state and the set of stable unknowns increases by the set `reach_cap` of the current unknown.

**lemma partial\_correctness\_ind:**

```

shows "query_dom x y c  $\sigma$ 
 $\Rightarrow$  query x y c  $\sigma$  = (yd,  $\sigma'$ )
 $\Rightarrow$  invariant s c  $\sigma$ 
 $\Rightarrow$  invariant (s  $\cup$  reach_cap  $\sigma'$  c y) c  $\sigma'$ 
 $\wedge$  ( $\forall \xi \in s$ . mlup  $\sigma$   $\xi$  = mlup  $\sigma'$   $\xi$ )"
and "iterate_dom x c  $\sigma$ 
 $\Rightarrow$  iterate x c  $\sigma$  = (xd,  $\sigma'$ )
 $\Rightarrow$  x  $\in$  c
 $\Rightarrow$  invariant s (c - {x})  $\sigma$ 
 $\Rightarrow$  invariant (s  $\cup$  (reach_cap  $\sigma'$  (c - {x}) x)) (c - {x})  $\sigma'$ 
 $\wedge$  ( $\forall \xi \in s$ . mlup  $\sigma$   $\xi$  = mlup  $\sigma'$   $\xi$ )"
and "eval_dom x t c  $\sigma$ 
 $\Rightarrow$  eval x t c  $\sigma$  = (xd,  $\sigma'$ )
 $\Rightarrow$  invariant s c  $\sigma$ 
 $\Rightarrow$  invariant (s  $\cup$  reach_cap_tree  $\sigma'$  c t) c  $\sigma'$ 
 $\wedge$  ( $\forall \xi \in s$ . mlup  $\sigma$   $\xi$  = mlup  $\sigma'$   $\xi$ )
 $\wedge$  traverse_rhs t  $\sigma'$  = xd"

```

*<proof>*

Since the initial solver state fulfills the valid solver state predicate, we can conclude from the above lemma, that the solve function returns a partial solution for the queried unknown `x` and all unknowns on which it transitively depends.

**corollary partial\_correctness:**

```

assumes "solve_dom x"
and "solve x =  $\sigma$ "
shows "part_solution  $\sigma$  (reach  $\sigma$  x)"

```

*<proof>*

### 3.7 Termination of TD\_plain for Stable Unknowns

In the equivalence proof of the TD and the TD\_plain, we need to show that when the TD trivially terminates because the queried unknown is already stable and its value is only looked up, the evaluation of this unknown `x` with TD\_plain also terminates. For this, we exploit that the set of stable unknowns is always finite during a terminating solver's run and provide the following lemma:

```

lemma td1_terminates_for_stabl:
  assumes "x ∈ s"
    and "invariant s (c - {x}) σ"
    and "mlup σ x = xd"
    and "finite s"
    and "x ∈ c"
  shows "iterate_dom x c σ" and "iterate x c σ = (xd, σ)"
⟨proof⟩

```

### 3.8 Program Refinement for Code Generation

For code generation, we define a refined version of the solver function using the `partial_function` keyword with the `option` attribute.

```

datatype ('a,'b) state = Q "'a × 'a × 'a set × ('a, 'b) map"
  | I "'a × 'a set × ('a, 'b) map" | E "'a × ('a,'b) strategy_tree
  × 'a set × ('a, 'b) map"

```

```

partial_function (option)
  solve_rec_c :: "('x, 'd) state ⇒ ('d × ('x, 'd) map) option"
  where
    "solve_rec_c s = (case s of Q (x, y, c, σ) ⇒
      if y ∈ c then
        Some (mlup σ y, σ)
      else
        solve_rec_c (I (y, (insert y c), σ))
    | I (x, c, σ) ⇒
      Option.bind (solve_rec_c (E (x, (T x), c, σ))) (λ(d_new, σ).
        if d_new = mlup σ x then
          Some (d_new, σ)
        else
          solve_rec_c (I (x, c, (σ(x ↦ d_new))))))
    | E (x, t, c, σ) ⇒
      (case t of
        Answer d ⇒ Some (d, σ)
      | Query y g ⇒ Option.bind (solve_rec_c (Q (x, y, c, σ)))
        (λ(yd, σ). solve_rec_c (E (x, (g yd), c, σ)))))"

```

```

declare solve_rec_c.simps[simp,code]

```

```

definition solve_rec_c_dom where "solve_rec_c_dom p ≡ ∃σ. solve_rec_c
p = Some σ"

```

```

definition solve_c :: "'x ⇒ (('x, 'd) map) option" where
  "solve_c x = Option.bind (solve_rec_c (I (x, {x}, Map.empty))) (λ(_,
σ). Some σ)"

```

```

definition solve_c_dom :: "'x ⇒ bool" where "solve_c_dom x ≡ ∃σ. solve_c
x = Some σ"

```

We proof the equivalence between the refined solver function for code generation and the initial version used for the partial correctness proof.

```

lemma query_iterate_eval_solve_rec_c_equiv:
  shows "query_dom x y c  $\sigma$   $\implies$  solve_rec_c_dom (Q (x,y,c, $\sigma$ ))
     $\wedge$  query x y c  $\sigma$  = the (solve_rec_c (Q (x,y,c, $\sigma$ )))"
  and "iterate_dom x c  $\sigma$   $\implies$  solve_rec_c_dom (I (x,c, $\sigma$ ))
     $\wedge$  iterate x c  $\sigma$  = the (solve_rec_c (I (x,c, $\sigma$ )))"
  and "eval_dom x t c  $\sigma$   $\implies$  solve_rec_c_dom (E (x,t,c, $\sigma$ ))
     $\wedge$  eval x t c  $\sigma$  = the (solve_rec_c (E (x,t,c, $\sigma$ )))"
<proof>

```

```

lemma solve_rec_c_query_iterate_eval_equiv:
  shows "solve_rec_c s = Some r  $\implies$  (case s of
    Q (x,y,c, $\sigma$ )  $\Rightarrow$  query_dom x y c  $\sigma$   $\wedge$  query x y c  $\sigma$  = r
  | I (x,c, $\sigma$ )  $\Rightarrow$  iterate_dom x c  $\sigma$   $\wedge$  iterate x c  $\sigma$  = r
  | E (x,t,c, $\sigma$ )  $\Rightarrow$  eval_dom x t c  $\sigma$   $\wedge$  eval x t c  $\sigma$  = r)"
<proof>

```

```

theorem term_equivalence: "solve_dom x  $\longleftrightarrow$  solve_c_dom x"
<proof>

```

```

theorem value_equivalence:
  "solve_dom x  $\implies$   $\exists$   $\sigma$ . solve_c x = Some  $\sigma$   $\wedge$  solve x =  $\sigma$ "
<proof>

```

Then, we can define the code equation for `solve` based on the refined solver program `solve_c`.

```

lemma solve_code_equation [code]:
  "solve x = (case solve_c x of Some r  $\Rightarrow$  r
  | None  $\Rightarrow$  Code.abort (String.implode ''Input not in domain'') ( $\lambda$ _. solve
x))"
<proof>

```

**end**

To setup the code generation for the solver locale we use a dedicated rewrite definition.

```

global_interpretation TD_plain_Interp: TD_plain D T for D T
  defines TD_plain_Interp_solve = TD_plain_Interp.solve
<proof>

```

**end**

## 4 The Top-Down Solver

In this theory we proof the partial correctness of the original TD by establishing its equivalence with the TD\_plain. Compared to the TD\_plain, it

additionally tracks a set of currently stable unknowns *stabl*, and a map *infl* collecting for each unknown *x* a list of unknowns influenced by it. This allows for the optimization that skips the re-evaluation of unknowns which are already stable. It does, however, also require a destabilization mechanism triggering re-evaluation of all unknowns possibly affected by an unknown whose value has changed.

```

theory TD_equiv
  imports Main "HOL-Library.Finite_Map" Basics TD_plain
begin

declare fun_upd_apply[simp del]

locale TD = Solver D T
  for D :: "'d::bot"
    and T :: "'x ⇒ ('x, 'd) strategy_tree"
begin

```

#### 4.1 Definition of Destabilize and Proof of its Termination

The destabilization function is called by the solver before continuing iteration because the value of an unknown changed. In this case, also the values of unknowns whose last evaluation was based on the outdated value, need to be re-evaluated again. This re-evaluation of influenced unknowns is enforced by following the entries for directly influenced unknowns in the map *infl* and removing all transitively influenced unknowns from *stabl*. This way, influenced unknowns are not re-evaluated immediately, but instead will be re-evaluated whenever they are queried again.

```

function (domintros)
destab_iter :: "'x list ⇒ ('x, 'x list) fmap ⇒ 'x set ⇒ ('x, 'x list)
fmap × 'x set"
and destab :: "'x ⇒ ('x, 'x list) fmap ⇒ 'x set ⇒ ('x, 'x list) fmap
× 'x set" where
  "destab_iter [] infl stabl = (infl, stabl)"
| "destab_iter (y # ys) infl stabl = (
  let (infl, stabl) = destab y infl (stabl - {y}) in
  destab_iter ys infl stabl)"
| "destab x infl stabl = destab_iter (fmlookup_default infl [] x) (fmdrop
x infl) stabl"
  ⟨proof⟩

```

```

definition destab_iter_dom where
  "destab_iter_dom ls infl stabl = destab_iter_destab_dom (Inl (ls, infl,
stabl))"
declare destab_iter_dom_def[simp]

```

```

definition destab_dom where
  "destab_dom y infl stabl = destab_iter_destab_dom (Inr (y, infl, stabl))"

```

```
declare destab_dom_def[simp]
```

```
lemma destab_domintros:
```

```
"destab_iter_dom [] infl stabl"
"destab_dom y infl (stabl - {y}) ==>
  destab y infl (stabl - {y}) = (infl', stabl') ==>
  destab_iter_dom ys infl' stabl' ==>
  destab_iter_dom (y # ys) infl stabl"
"destab_iter_dom (fmlookup_default infl [] x) (fmdrop x infl) stabl
==> destab_dom x infl stabl"
<proof>
```

```
definition count_non_empty :: "('a, 'b list) fmap => nat" where
  "count_non_empty m = fcard (ffilter ((≠) [] ∘ snd) (fset_of_fmap m))"
```

```
lemma count_non_empty_dec_fmdrop:
```

```
  assumes "fmlookup_default m [] x ≠ []"
  shows "Suc (count_non_empty (fmdrop x m)) = count_non_empty m"
<proof>
```

```
lemma count_non_empty_eq_fmdrop:
```

```
  assumes "fmlookup_default m [] x = []"
  shows "count_non_empty (fmdrop x m) = count_non_empty m"
<proof>
```

```
termination
```

```
<proof>
```

## 4.2 Definition of the Solver Algorithm

Apart from passing the additional arguments for the solver state, the *iterate* function contains, compared to the *TD\_plain*, an additional check to skip iteration of already stable unknowns. Furthermore, the helper function *destabilize* is called whenever the newly evaluated value of an unknown changed compared to the value tracked in  $\sigma$ . Lastly, a dependency is recorded whenever returning from a *query* call for unknown  $x$  within the evaluation of right-hand side of unknown  $y$ .

```
function (domintros)
```

```
  query :: "'x => 'x => 'x set => ('x, 'x list) fmap => 'x set => ('x,
'd) map
          => 'd × ('x, 'x list) fmap × 'x set × ('x, 'd) map" and
  iterate :: "'x => 'x set => ('x, 'x list) fmap => 'x set => ('x, 'd)
map
          => 'd × ('x, 'x list) fmap × 'x set × ('x, 'd) map" and
  eval :: "'x => ('x, 'd) strategy_tree => 'x set => ('x, 'x list)
fmap => 'x set
          => ('x, 'd) map => 'd × ('x, 'x list) fmap × 'x set ×
('x, 'd) map" where
```

```

"query y x c infl stabl  $\sigma$  = (
  let (xd, infl, stabl,  $\sigma$ ) =
    if  $x \in c$  then
      (mlup  $\sigma$  x, infl, stabl,  $\sigma$ )
    else
      iterate x (insert x c) infl stabl  $\sigma$ 
  in (xd, fminsert infl x y, stabl,  $\sigma$ ))"
| "iterate x c infl stabl  $\sigma$  = (
  if  $x \notin$  stabl then
    let (d_new, infl, stabl,  $\sigma$ ) = eval x (T x) c infl (insert x stabl)
 $\sigma$  in
    if mlup  $\sigma$  x = d_new then
      (d_new, infl, stabl,  $\sigma$ )
    else
      let (infl, stabl) = destab x infl stabl in
        iterate x c infl stabl ( $\sigma(x \mapsto d\_new)$ )
    else
      (mlup  $\sigma$  x, infl, stabl,  $\sigma$ ))"
| "eval x t c infl stabl  $\sigma$  = (case t of
  Answer d  $\Rightarrow$  (d, infl, stabl,  $\sigma$ )
| Query y g  $\Rightarrow$  (
  let (yd, infl, stabl,  $\sigma$ ) = query x y c infl stabl  $\sigma$  in eval x
(g yd) c infl stabl  $\sigma$ ))"
<proof>

```

**definition solve** :: "'x  $\Rightarrow$  'x set  $\times$  ('x, 'd) map" where  
 "solve x = (let (\_, \_, stabl,  $\sigma$ ) = iterate x {x} fmempty {} Map.empty  
 in (stabl,  $\sigma$ ))"

**definition query\_dom** where  
 "query\_dom x y c infl stabl  $\sigma$  = query\_iterate\_eval\_dom (Inl (x, y, c,  
 infl, stabl,  $\sigma$ ))"  
**declare query\_dom\_def** [simp]  
**definition iterate\_dom** where  
 "iterate\_dom x c infl stabl  $\sigma$  = query\_iterate\_eval\_dom (Inr (Inl (x,  
 c, infl, stabl,  $\sigma$ )))"  
**declare iterate\_dom\_def** [simp]  
**definition eval\_dom** where  
 "eval\_dom x t c infl stabl  $\sigma$  = query\_iterate\_eval\_dom (Inr (Inr (x,  
 t, c, infl, stabl,  $\sigma$ )))"  
**declare eval\_dom\_def** [simp]

**definition solve\_dom** where  
 "solve\_dom x = iterate\_dom x {x} fmempty {} Map.empty"

**lemmas dom\_defs** = query\_dom\_def iterate\_dom\_def eval\_dom\_def

### 4.3 Refinement of Auto-Generated Rules

The auto-generated `pinduct` rule contains a redundant assumption. This lemma removes this redundant assumption such that the rule is easier to instantiate and gives comprehensible names to the cases.

```
lemmas query_iterate_eval_pinduct[consumes 1, case_names Query Iterate Eval]
```

```
= query_iterate_eval.pinduct(1)[
  folded query_dom_def iterate_dom_def eval_dom_def,
  of x y c infl stabl  $\sigma$  for x y c infl stabl  $\sigma$ 
]
query_iterate_eval.pinduct(2)[
  folded query_dom_def iterate_dom_def eval_dom_def,
  of x c infl stabl  $\sigma$  for x c infl stabl  $\sigma$ 
]
query_iterate_eval.pinduct(3)[
  folded query_dom_def iterate_dom_def eval_dom_def,
  of x t c infl stabl  $\sigma$  for x t c infl stabl  $\sigma$ 
]
```

```
lemmas iterate_pinduct[consumes 1, case_names Iterate]
= query_iterate_eval_pinduct(2)[where ?P="λx y c infl stabl  $\sigma$ . True"
and ?R="λx t c infl stabl  $\sigma$ . True", simplified (no_asm_use),
folded query_dom_def iterate_dom_def eval_dom_def]
```

```
declare query.psimps [simp]
declare iterate.psimps [simp]
declare eval.psimps [simp]
```

### 4.4 Domain Lemmas

```
lemma dom_backwards_pinduct:
  shows "query_dom x y c infl stabl  $\sigma$ 
     $\implies$  y  $\notin$  c  $\implies$  iterate_dom y (insert y c) infl stabl  $\sigma$ "
  and "iterate_dom x c infl stabl  $\sigma$ 
     $\implies$  x  $\notin$  stabl  $\implies$  (eval_dom x (T x) c infl (insert x stabl)  $\sigma \wedge$ 
      ((xd_new, infl1, stabl1,  $\sigma'$ ) = eval x (T x) c infl (insert x stabl)
 $\sigma$ 
       $\implies$  mlup  $\sigma'$  x  $\neq$  xd_new  $\implies$  (infl2, stabl2) = destab x infl1
stabl1  $\implies$ 
      iterate_dom x c infl2 stabl2 ( $\sigma'(x \mapsto xd\_new)$ )))"
  and "eval_dom x (Query y g) c infl stabl  $\sigma$ 
     $\implies$  (query_dom x y c infl stabl  $\sigma \wedge$ 
      ((yd, infl', stabl',  $\sigma'$ ) = query x y c infl stabl  $\sigma \implies$ 
      eval_dom x (g yd) c infl' stabl'  $\sigma'$ ))"
<proof>
```

## 4.5 Case Rules

```

lemma iterate_continue_fixpoint_cases[consumes 3]:
  assumes "iterate_dom x c infl stabl  $\sigma$ "
    and "(xd, infl', stabl',  $\sigma'$ ) = iterate x c infl stabl  $\sigma$ "
    and " $x \in c$ "
  obtains (Stable) "infl' = infl"
    and "stabl' = stabl"
    and " $\sigma' = \sigma$ "
    and "mlup  $\sigma$  x = xd"
    and " $x \in stabl$ "
  | (Fixpoint) "eval_dom x (T x) c infl (insert x stabl)  $\sigma$ "
    and "(xd, infl', stabl',  $\sigma'$ ) = eval x (T x) c infl (insert x stabl)
 $\sigma$ "
    and "mlup  $\sigma'$  x = xd"
    and " $x \notin stabl$ "
  | (Continue) stabl1 infl1  $\sigma$ 1 xd_new stabl2 infl2
  where "eval_dom x (T x) c infl (insert x stabl)  $\sigma$ "
    and "(xd_new, infl1, stabl1,  $\sigma$ 1) = eval x (T x) c infl (insert x
stabl)  $\sigma$ "
    and "mlup  $\sigma$ 1 x  $\neq$  xd_new"
    and "(infl2, stabl2) = destab x infl1 stabl1"
    and "iterate_dom x c infl2 stabl2 ( $\sigma$ 1(x  $\mapsto$  xd_new))"
    and "(xd, infl', stabl',  $\sigma'$ ) = iterate x c infl2 stabl2 ( $\sigma$ 1(x  $\mapsto$ 
xd_new))"
    and " $x \notin stabl$ "
  <proof>

```

```

lemma iterate_fmlookup:
  assumes "iterate_dom x c infl stabl  $\sigma$ "
    and "(xd, infl', stabl',  $\sigma'$ ) = iterate x c infl stabl  $\sigma$ "
    and " $x \in c$ "
  shows "mlup  $\sigma'$  x = xd"
  <proof>

```

```

corollary query_fmlookup:
  assumes "query_dom y x c infl stabl  $\sigma$ "
    and "(xd, infl', stabl',  $\sigma'$ ) = query y x c infl stabl  $\sigma$ "
  shows "mlup  $\sigma'$  x = xd"
  <proof>

```

```

lemma query_iterate_lookup_cases [consumes 2]:
  assumes "query_dom y x c infl stabl  $\sigma$ "
    and "(xd, infl', stabl',  $\sigma'$ ) = query y x c infl stabl  $\sigma$ "
  obtains (Iterate) infl1
  where "iterate_dom x (insert x c) infl stabl  $\sigma$ "
    and "(xd, infl1, stabl',  $\sigma'$ ) = iterate x (insert x c) infl stabl
 $\sigma$ "
    and "infl' = fminsert infl1 x y"
    and "mlup  $\sigma'$  x = xd"

```

```

    and "x  $\notin$  c"
  | (Lookup) "mlup  $\sigma$  x = xd"
    and "infl' = fminsert infl x y"
    and "stabl' = stabl"
    and " $\sigma'$  =  $\sigma$ "
    and "x  $\in$  c"
  <proof>

```

```

lemma eval_query_answer_cases [consumes 2]:
  assumes "eval_dom x t c infl stabl  $\sigma$ "
    and "(xd, infl', stabl',  $\sigma'$ ) = eval x t c infl stabl  $\sigma$ "
  obtains (Query) y g yd infl1 stabl1  $\sigma$ 1
  where "t = Query y g"
    and "query_dom x y c infl stabl  $\sigma$ "
    and "(yd, infl1, stabl1,  $\sigma$ 1) = query x y c infl stabl  $\sigma$ "
    and "eval_dom x (g yd) c infl1 stabl1  $\sigma$ 1"
    and "(xd, infl', stabl',  $\sigma'$ ) = eval x (g yd) c infl1 stabl1  $\sigma$ 1"
    and "mlup  $\sigma$ 1 y = yd"
  | (Answer) "t = Answer xd"
    and "infl' = infl"
    and "stabl' = stabl"
    and " $\sigma'$  =  $\sigma$ "
  <proof>

```

## 4.6 Description of the Effect of Destabilize

To describe the effect of a call to the function `destab`, we define an inductive set that, based on some `infl` map, collects all unknowns transitively influenced by some unknown `x`.

**inductive\_set influenced\_by** for `infl x` where

```

  base: "fmlookup infl x = Some ys  $\implies$  y  $\in$  set ys  $\implies$  y  $\in$  influenced_by infl x"
  | step: "y  $\in$  influenced_by infl x  $\implies$  fmlookup infl y = Some zs  $\implies$  z  $\in$  set zs
     $\implies$  z  $\in$  influenced_by infl x"

```

**inductive\_set influenced\_by\_cutoff** for `infl x c` where

```

  base: "x  $\notin$  c  $\implies$  fmlookup infl x = Some ys  $\implies$  y  $\in$  set ys  $\implies$  y  $\in$  influenced_by_cutoff infl x c"
  | step: "y  $\in$  influenced_by_cutoff infl x c  $\implies$  y  $\notin$  c  $\implies$  fmlookup infl y = Some zs  $\implies$  z  $\in$  set zs
     $\implies$  z  $\in$  influenced_by_cutoff infl x c"

```

**lemma influenced\_by\_aux:**

```

  shows "influenced_by infl x = ( $\bigcup$  y  $\in$  slookup infl x. insert y (influenced_by (fmdrop x infl) y))"
  <proof>

```

**lemma lookup\_in\_influenced:**

```

  shows "slookup infl x  $\subseteq$  influenced_by infl x"

```

*<proof>*

**lemma** *influenced\_unknowns\_fmdrop\_set:*  
  **shows** "influenced\_by (fmdrop\_set C infl) x = influenced\_by\_cutoff infl x C"  
*<proof>*

**lemma** *influenced\_by\_transitive:*  
  **assumes** "y ∈ influenced\_by infl x"  
  **and** "z ∈ influenced\_by infl y"  
  **shows** "z ∈ influenced\_by infl x"  
*<proof>*

**lemma** *influenced\_cutoff\_subset:*  
  "influenced\_by\_cutoff infl x C ⊆ influenced\_by infl x"  
*<proof>*

**lemma** *influenced\_cutoff\_subset\_2:*  
  **shows** "influenced\_by infl x - (⋃y ∈ C. influenced\_by infl y) ⊆ influenced\_by\_cutoff infl x C"  
*<proof>*

**lemma** *union\_influenced\_to\_cutoff:*  
  **shows** "insert y (influenced\_by infl y) ∪ influenced\_by infl x =  
  insert y (influenced\_by infl y) ∪ influenced\_by\_cutoff infl x (insert y (influenced\_by infl y))"  
*<proof>*

**lemma** *destab\_iter\_infl\_stabl\_relation:*  
  **shows**  
    "(infl', stabl') = destab\_iter xs infl stabl  
    ⇒ infl' = fmdrop\_set (⋃x ∈ set xs. insert x (influenced\_by infl x)) infl  
    ∧ stabl' = stabl - (⋃x ∈ set xs. insert x (influenced\_by infl x))"  
  **and** *destab\_infl\_stabl\_relation:*  
    "(infl', stabl') = destab x infl stabl  
    ⇒ infl' = fmdrop\_set (insert x (influenced\_by infl x)) infl  
    ∧ stabl' = stabl - influenced\_by infl x"  
*<proof>*

## 4.7 Predicate for Valid Input States

For the TD, we extend the predicate of valid solver states of the TD\_plain, to also covers the additional data structures *stabl* and *infl*:

**definition** *invariant where*  
  "invariant c σ infl stabl ≡  
  c ⊆ stabl  
  ∧ part\_solution σ (stabl - c)  
  ∧ fset (fdom infl) ⊆ stabl"

$\wedge (\forall y \in \text{stabl} - c. \forall x \in \text{dep } \sigma y. y \in \text{slookup infl } x)$ "

**lemma invariant\_simp\_c\_stabl:**  
 assumes "x ∈ c"  
 and "invariant (c - {x}) σ infl stabl"  
 shows "invariant c σ infl (insert x stabl)"  
 ⟨proof⟩

## 4.8 Auxiliary Lemmas for Partial Correctness Proofs

**lemma stabl\_infl\_empty:**  
 assumes "x ∉ stabl"  
 and "fset (fmdom infl) ⊆ stabl"  
 shows "slookup infl x = {}"  
 ⟨proof⟩

**lemma dep\_closed\_implies\_reach\_cap\_tree\_closed:**  
 assumes "x ∈ stabl'"  
 and " $\forall \xi \in \text{stabl}' - (c - \{x\}). \text{dep } \sigma' \xi \subseteq \text{stabl}'$ "  
 shows " $\text{reach\_cap } \sigma' (c - \{x\}) x \subseteq \text{stabl}'$ "  
 ⟨proof⟩

**lemma dep\_subset\_stable:**  
 assumes "fset (fmdom infl) ⊆ stabl"  
 and " $(\forall y \in \text{stabl} - c. \forall x \in \text{dep } \sigma y. y \in \text{slookup infl } x)$ "  
 shows " $(\forall \xi \in \text{stabl} - c. \text{dep } \sigma \xi \subseteq \text{stabl})$ "  
 ⟨proof⟩

**lemma new\_lookup\_to\_infl\_not\_stabl:**  
 assumes " $\forall \xi. (\text{slookup infl1 } \xi - \text{slookup infl } \xi) \cap \text{stabl} = \{\}$ "  
 and "x ∉ stabl"  
 and "fset (fmdom infl) ⊆ stabl"  
 shows " $\text{influenced\_by infl1 } x \cap \text{stabl} = \{\}$ "  
 ⟨proof⟩

**lemma infl\_upd\_diff:**  
 assumes " $\forall \xi. (\text{slookup infl}' \xi - \text{slookup infl } \xi) \cap \text{stabl} = \{\}$ "  
 shows " $\forall \xi. (\text{slookup (fminsert infl}' x y) \xi - \text{slookup infl } \xi) \cap (\text{stabl} - \{y\}) = \{\}$ "  
 ⟨proof⟩

**lemma infl\_diff\_eval\_step:**  
 assumes "stabl ⊆ stabl1"  
 and " $\forall \xi. (\text{slookup infl}' \xi - \text{slookup infl1 } \xi) \cap (\text{stabl1} - \{x\}) = \{\}$ "  
 and " $\forall \xi. (\text{slookup infl1 } \xi - \text{slookup infl } \xi) \cap (\text{stabl} - \{x\}) = \{\}$ "  
 shows " $\forall \xi. (\text{slookup infl}' \xi - \text{slookup infl } \xi) \cap (\text{stabl} - \{x\}) = \{\}$ "  
 ⟨proof⟩

## 4.9 Preservation of the Invariant

In this section, we prove that the destabilization of some unknown that is currently being iterated, will preserve the valid solver state invariant.

```

lemma destabil_x_no_dep:
  assumes "stabl2 = stabl1 - influenced_by infl1 x"
    and "∀y∈stabl1 - (c - {x}). ∀z∈dep σ1 y. y ∈ slookup infl1 z"
  shows "∀y ∈ stabl2 - (c - {x}). x ∉ dep σ1 y"
⟨proof⟩

lemma destabil_preserves_c_subset_stabl:
  assumes "c ⊆ stabl"
    and "stabl ⊆ stabl'"
  shows "c ⊆ stabl'"
⟨proof⟩

lemma destabil_preserves_infl_dom_stabl:
  assumes "(infl', stabl') = destabil x infl stabl"
    and "fset (fmdom infl) ⊆ stabl"
  shows "fset (fmdom infl') ⊆ stabl'"
⟨proof⟩

lemma destabil_and_upd_preserves_dep_closed_in_infl:
  assumes "(infl2, stabl2) = destabil x infl1 stabl1"
    and "(∀y∈stabl1 - (c - {x}). ∀z∈dep σ1 y. y ∈ slookup infl1 z)"
  shows "(∀y∈stabl2 - (c - {x}). ∀z∈dep (σ1(x ↦ xd'))) y. y ∈ slookup
infl2 z)"
⟨proof⟩

lemma destabil_upd_preserves_part_sol:
  assumes "(infl2, stabl2) = destabil x infl1 stabl1"
    and "part_solution σ1 (stabl1 - c)"
    and "∀y∈stabl1 - (c - {x}). ∀x∈dep σ1 y. y ∈ slookup infl1 x"
    and "traverse_rhs (T x) σ1 = xd'"
  shows "part_solution (σ1(x ↦ xd')) (stabl2 - (c - {x}))"
⟨proof⟩

```

## 4.10 TD\_plain and TD Equivalence

Finally, we can prove the equivalence of TD and TD\_plain. We split this proof into two parts: first we show that whenever the TD\_plain terminates the TD terminates as well and returns the same result, and second we show the other direction, i.e., whenever the TD terminates, the TD\_plain terminates as well and returns the same result.

```

declare TD_plain.query_dom_def[of T, simp]
declare TD_plain.eval_dom_def[of T, simp]
declare TD_plain.iterate_dom_def[of T, simp]
declare TD_plain.query.psimps[of T, simp]

```

```

declare TD_plain.iterate.psimps[of T,simp]
declare TD_plain.eval.psimps[of T,simp]

```

To carry out the induction proof, we complement the valid solver state invariant, with a second predicate `update_rel`, that describes the relation between output and input solver states.

```

abbreviation "update_rel x infl stabl infl' stabl'  $\equiv$ 
  stabl  $\subseteq$  stabl'  $\wedge$ 
  ( $\forall u \in$  stabl. slookup infl u  $\subseteq$  slookup infl' u)  $\wedge$ 
  ( $\forall u$ . (slookup infl' u - slookup infl u)  $\cap$  (stabl - {x}) = {})"

```

#### 4.10.1 TD\_plain $\rightarrow$ TD

```

lemma TD_plain_TD_equivalence_ind:
  shows "TD_plain.query_dom T x y c  $\sigma$ 
 $\implies$  TD_plain.query T x y c  $\sigma = (yd, \sigma')$ 
 $\implies$  invariant c  $\sigma$  infl stabl
 $\implies$  query_dom x y c infl stabl  $\sigma$ 
 $\wedge$  ( $\exists$  infl' stabl'. query x y c infl stabl  $\sigma = (yd, infl', stabl',$ 
 $\sigma')$ 
 $\wedge$  invariant c  $\sigma'$  infl' stabl'
 $\wedge$  x  $\in$  slookup infl' y
 $\wedge$  update_rel x infl stabl infl' stabl'"
  and "TD_plain.iterate_dom T x c  $\sigma$ 
 $\implies$  TD_plain.iterate T x c  $\sigma = (xd, \sigma')$ 
 $\implies$  x  $\in$  c
 $\implies$  invariant (c - {x})  $\sigma$  infl stabl
 $\implies$  iterate_dom x c infl stabl  $\sigma$ 
 $\wedge$  ( $\exists$  infl' stabl'. iterate x c infl stabl  $\sigma = (xd, infl', stabl',$ 
 $\sigma')$ 
 $\wedge$  invariant (c - {x})  $\sigma'$  infl' stabl'
 $\wedge$  x  $\in$  stabl'
 $\wedge$  update_rel x infl stabl infl' stabl'"
  and "TD_plain.eval_dom T x t c  $\sigma$ 
 $\implies$  TD_plain.eval T x t c  $\sigma = (xd, \sigma')$ 
 $\implies$  invariant c  $\sigma$  infl stabl
 $\implies$  x  $\in$  stabl
 $\implies$  eval_dom x t c infl stabl  $\sigma$ 
 $\wedge$  ( $\exists$  infl' stabl'. eval x t c infl stabl  $\sigma = (xd, infl', stabl',$ 
 $\sigma')$ 
 $\wedge$  invariant c  $\sigma'$  infl' stabl'
 $\wedge$  traverse_rhs t  $\sigma' = xd$ 
 $\wedge$  ( $\forall y \in$  dep_aux  $\sigma'$  t. x  $\in$  slookup infl' y)
 $\wedge$  update_rel x infl stabl infl' stabl'"
  <proof>

corollary TD_plain_TD_equivalence:
  assumes "TD_plain.solve_dom T x"
  and "TD_plain.solve T x =  $\sigma$ "

```

shows " $\exists \text{stabl. solve\_dom } x \wedge \text{solve } x = (\text{stabl}, \sigma)$ "  
 <proof>

#### 4.10.2 TD $\rightarrow$ TD\_plain

lemmas TD\_plain\_dom\_defs =  
 TD\_plain.query\_dom\_def[of T]  
 TD\_plain.iterate\_dom\_def[of T]  
 TD\_plain.eval\_dom\_def[of T]

lemma TD\_TD\_plain\_equivalence\_ind:  
 shows "query\_dom x y c infl stabl  $\sigma$   
 $\implies$  (yd, infl', stabl',  $\sigma'$ ) = query x y c infl stabl  $\sigma$   
 $\implies$  invariant c  $\sigma$  infl stabl  
 $\implies$  finite stabl  
 $\implies$  invariant c  $\sigma'$  infl' stabl'  
 $\wedge$  TD\_plain.query\_dom T x y c  $\sigma$   
 $\wedge$  (yd,  $\sigma'$ ) = TD\_plain.query T x y c  $\sigma$   
 $\wedge$  finite stabl'  
 $\wedge$  x  $\in$  slookup infl' y  
 $\wedge$  update\_rel x infl stabl infl' stabl'"  
 and "iterate\_dom x c infl stabl  $\sigma$   
 $\implies$  (xd, infl', stabl',  $\sigma'$ ) = iterate x c infl stabl  $\sigma$   
 $\implies$  x  $\in$  c  
 $\implies$  invariant (c - {x})  $\sigma$  infl stabl  
 $\implies$  finite stabl  
 $\implies$  invariant (c - {x})  $\sigma'$  infl' stabl'  
 $\wedge$  TD\_plain.iterate\_dom T x c  $\sigma$   
 $\wedge$  (xd,  $\sigma'$ ) = TD\_plain.iterate T x c  $\sigma$   
 $\wedge$  finite stabl'  
 $\wedge$  x  $\in$  stabl'  
 $\wedge$  update\_rel x infl stabl infl' stabl'"  
 and "eval\_dom x t c infl stabl  $\sigma$   
 $\implies$  (xd, infl', stabl',  $\sigma'$ ) = eval x t c infl stabl  $\sigma$   
 $\implies$  invariant c  $\sigma$  infl stabl  
 $\implies$  x  $\in$  stabl  
 $\implies$  finite stabl  
 $\implies$  invariant c  $\sigma'$  infl' stabl'  
 $\wedge$  TD\_plain.eval\_dom T x t c  $\sigma$   
 $\wedge$  (xd,  $\sigma'$ ) = TD\_plain.eval T x t c  $\sigma$   
 $\wedge$  finite stabl'  
 $\wedge$  traverse\_rhs t  $\sigma'$  = xd  
 $\wedge$  ( $\forall y \in \text{dep\_aux } \sigma' t. x \in \text{slookup infl' } y$ )  
 $\wedge$  update\_rel x infl stabl infl' stabl'"  
 <proof>

corollary TD\_TD\_plain\_equivalence:  
 assumes "solve\_dom x"  
 and "solve x = (stabl,  $\sigma$ )"

```

  shows "TD_plain.solve_dom T x  $\wedge$  TD_plain.solve T x =  $\sigma$ "
  <proof>

```

#### 4.11 Partial Correctness of the TD

From the equivalence of the TD and TD\_plain and the partial correctness proof of the TD\_plain we can now conclude partial correctness also for the TD.

```

corollary partial_correctness:
  assumes "solve_dom x"
    and "solve x = (stabl,  $\sigma$ )"
  shows "part_solution  $\sigma$  stabl" and "reach  $\sigma$  x  $\subseteq$  stabl"
  <proof>

```

#### 4.12 Program Refinement for Code Generation

To derive executable code for the TD, we do a program refinement and define an equivalent solve function based on partial\_function with options that can be used for the code generation.

```

datatype ('a,'b) state = Q "'a  $\times$  'a  $\times$  'a set  $\times$  ('a, 'a list) fmap  $\times$ 
'a set  $\times$  ('a, 'b) map"
  | I "'a  $\times$  'a set  $\times$  ('a, 'a list) fmap  $\times$  'a set  $\times$  ('a, 'b) map"
  | E "'a  $\times$  ('a,'b) strategy_tree  $\times$  'a set  $\times$  ('a, 'a list) fmap  $\times$  'a
set  $\times$  ('a, 'b) map"

```

```

partial_function (option) solve_rec_c ::
  "('x, 'd) state  $\Rightarrow$  ('d  $\times$  ('x, 'x list) fmap  $\times$  'x set  $\times$  ('x, 'd) map)
option"

```

```

where
  "solve_rec_c s = (case s of Q (y,x,c,infl,stabl, $\sigma$ )  $\Rightarrow$  Option.bind
    (if x  $\in$  c then
      Some (mlup  $\sigma$  x, infl, stabl,  $\sigma$ )
    else
      solve_rec_c (I (x, (insert x c), infl, stabl,  $\sigma$ )))
    (\(xd, infl, stabl,  $\sigma$ ). Some (xd, fminsert infl x y, stabl,  $\sigma$ ))
  | I (x,c,infl,stabl, $\sigma$ )  $\Rightarrow$ 
    if x  $\notin$  stabl then Option.bind (
      solve_rec_c (E (x, (T x), c, infl, insert x stabl,  $\sigma$ ))) (\(d_new,
infl, stabl,  $\sigma$ ).
      if mlup  $\sigma$  x = d_new then
        Some (d_new, infl, stabl,  $\sigma$ )
      else
        let (infl, stabl) = destab x infl stabl in
          solve_rec_c (I (x, c, infl, stabl,  $\sigma$ (x  $\mapsto$  d_new))))
    else
      Some (mlup  $\sigma$  x, infl, stabl,  $\sigma$ )
  | E (x,t,c,infl,stabl, $\sigma$ )  $\Rightarrow$  (case t of
    Answer d  $\Rightarrow$  Some (d, infl, stabl,  $\sigma$ )

```

```

      | Query y g ⇒ (
        Option.bind (solve_rec_c (Q (x, y, c, infl, stabl, σ))) (λ(yd,
infl, stabl, σ).
        solve_rec_c (E (x, g yd, c, infl, stabl, σ))))))"

```

```

definition solve_rec_c_dom where "solve_rec_c_dom p ≡ ∃σ. solve_rec_c
p = Some σ"

```

```

declare destab.simps[code]
declare destab_iter.simps[code]
declare solve_rec_c.simps[simp,code]

```

```

definition solve_c :: "'x ⇒ ('x set × (('x, 'd) map)) option" where
"solve_c x = Option.bind (solve_rec_c (I (x, {x}, fmempty, {}, Map.empty)))
(λ(., _, stabl, σ). Some (stabl,σ))"

```

```

definition solve_c_dom :: "'x ⇒ bool" where "solve_c_dom x ≡ ∃σ. solve_c
x = Some σ"

```

We prove the equivalence of the refined solver function for code generation and the initial version used for the partial correctness proof.

```

lemma query_iterate_eval_solve_rec_c_equiv:
  shows "query_dom x y c infl stabl σ ⇒ solve_rec_c_dom (Q (x,y,c,infl,stabl,σ))
    ∧ query x y c infl stabl σ = the (solve_rec_c (Q (x,y,c,infl,stabl,σ)))"
  and "iterate_dom x c infl stabl σ ⇒ solve_rec_c_dom (I (x,c,infl,stabl,σ))
    ∧ iterate x c infl stabl σ = the (solve_rec_c (I (x,c,infl,stabl,σ)))"
  and "eval_dom x t c infl stabl σ ⇒ solve_rec_c_dom (E (x,t,c,infl,stabl,σ))
    ∧ eval x t c infl stabl σ = the (solve_rec_c (E (x,t,c,infl,stabl,σ)))"
⟨proof⟩

```

```

lemma solve_rec_c_query_iterate_eval_equiv:
  shows "solve_rec_c s = Some r ⇒ (case s of
    Q (x,y,c,infl,stabl,σ) ⇒ query_dom x y c infl stabl σ
    ∧ query x y c infl stabl σ = r
  | I (x,c,infl,stabl,σ) ⇒ iterate_dom x c infl stabl σ
    ∧ iterate x c infl stabl σ = r
  | E (x,t,c,infl,stabl,σ) ⇒ eval_dom x t c infl stabl σ
    ∧ eval x t c infl stabl σ = r)"
⟨proof⟩

```

```

theorem term_equivalence: "solve_dom x ↔ solve_c_dom x"
⟨proof⟩

```

```

theorem value_equivalence: "solve_dom x ⇒ ∃σ. solve_c x = Some σ ∧
solve x = σ"
⟨proof⟩

```

With the equivalence of the refined version and the initial version proven, we can specify a the code equation.

```

lemma solve_code_equation [code]:
  "solve x = (case solve_c x of Some r ⇒ r
    | None ⇒ Code.abort (String.implode ''Input not in domain'') (λ_. solve
x))"
  ⟨proof⟩

```

**end**

Finally, we use a dedicated rewrite rule for the code generation of the solver locale.

```

global_interpretation TD_Interp: TD D T for D T
  defines
    TD_Interp_solve = TD_Interp.solve
  ⟨proof⟩

```

**end**

## 5 Example

```

theory Example
  imports TD_plain TD_equiv
begin

```

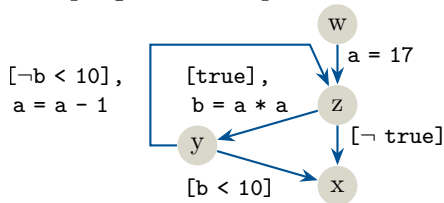
As an example, let us consider a program analysis, namely the analysis of must-be initialized program variables for the following program:

```

a = 17
while true:
  b = a * a
  if b < 10: break
  a = a - 1

```

The program corresponds to the following control-flow graph.



From the control-flow graph of the program, we generate the equation system to be solved by the TD. The left-hand side of an equation consists of an unknown which represents a program point. The right-hand side for some unknown describes how the set of must-be initialized variables at the corresponding program point can be computed from the sets of must-be initialized variables at the predecessors.

## 5.1 Definition of the Domain

```
datatype pv = a | b
```

A fitting domain to describe possible values for the must-be initialized analysis, is an inverse power set lattice of the set of all program variables. The least informative value which is always a true over-approximation for the must-be initialized analysis is the empty set (called top), whereas the initial value to start fixpoint iteration from is the set  $\{a, b\}$  (called bot). The join operation, which is used to combine the values of several incoming edges to obtain a sound over-approximation over all paths, corresponds to the intersection of sets.

```
typedef D = "Pow  $\{a, b\}$ "  
   $\langle proof \rangle$ 
```

```
setup_lifting D.type_definition_D
```

```
lift_definition top :: "D" is "{}"  $\langle proof \rangle$   
lift_definition bot :: D is "{a, b}"  $\langle proof \rangle$   
lift_definition join :: "D  $\Rightarrow$  D  $\Rightarrow$  D" is Set.inter  $\langle proof \rangle$ 
```

Additionally, we define some helper functions to create values of type D.

```
lift_definition insert :: "pv  $\Rightarrow$  D  $\Rightarrow$  D"  
  is " $\lambda e d$ . if  $e \in \{a, b\}$  then Set.insert e d else d"  
   $\langle proof \rangle$   
definition set_to_D :: "pv set  $\Rightarrow$  D" where  
  "set_to_D = ( $\lambda s$ . fold ( $\lambda e acc$ . if  $e \in s$  then insert e acc else acc)  
  [a, b] top)"
```

We show that the considered domain fulfills the sort constraints bot and equal as expected by the solver.

```
instantiation D :: bot  
begin  
  definition bot_D :: D  
  where "bot_D = bot"  
  
  instance  $\langle proof \rangle$   
end  
  
instantiation D :: equal  
begin  
  definition equal_D :: "D  $\Rightarrow$  D  $\Rightarrow$  bool"  
  where "equal_D d1 d2 = ((Rep_D d1) = (Rep_D d2))"  
  
  instance  $\langle proof \rangle$   
end
```

## 5.2 Definition of the Equation System

The following equation system can be generated for the must-be initialized analysis and the program from above.

$$\mathcal{T} : \begin{aligned} w &= \emptyset \\ z &= (y \cup \{a\}) \cap (w \cup \{a\}) \\ y &= z \cup \{b\} \\ x &= y \cap z \end{aligned}$$

Below we define this equation system and express the right-hand sides with strategy trees.

**datatype** *Unknown* = *X* | *Y* | *Z* | *W*

```
fun ConstrSys :: "Unknown  $\Rightarrow$  (Unknown, D) strategy_tree" where
  "ConstrSys X = Query Y ( $\lambda$ d1. if d1 = top then Answer top
    else Query Z ( $\lambda$ d2. Answer (join d1 d2)))"
| "ConstrSys Y = Query Z ( $\lambda$ d. if d  $\in$  {top, set_to_D {b}}
  then Answer (set_to_D {b}) else Answer bot)"
| "ConstrSys Z = Query Y ( $\lambda$ d1. if d1  $\in$  {top, set_to_D {a}}
  then Answer (set_to_D {a})
  else Query W ( $\lambda$ d2. if d2  $\in$  {top, set_to_D {a}}
    then Answer (set_to_D {a}) else Answer bot))"
| "ConstrSys W = Answer top"
```

## 5.3 Solve the Equation System with TD\_plain

We solve the equation system for each unknown, first with the TD\_plain and in the following also with the TD. Note, that we use a finite map that defaults to bot for keys that are not contained in the map. This can happen in two cases: (1) when the value computed for that unknown is equal to bot, or (2) if the unknown was not queried during the solving and therefore no value was stored in the finite map for it.

**definition** *solution\_plain\_X* **where**

```
"solution_plain_X = TD_plain_Interp_solve ConstrSys X"
value "(solution_plain_X X, solution_plain_X Y, solution_plain_X Z, solution_plain_X W)"
```

**definition** *solution\_plain\_Y* **where**

```
"solution_plain_Y = TD_plain_Interp_solve ConstrSys Y"
value "(solution_plain_Y X, solution_plain_Y Y, solution_plain_Y Z, solution_plain_Y W)"
```

**definition** *solution\_plain\_Z* **where**

```
"solution_plain_Z = TD_plain_Interp_solve ConstrSys Z"
value "(solution_plain_Z X, solution_plain_Z Y, solution_plain_Z Z, solution_plain_Z W)"
```

```

definition solution_plain_W where
  "solution_plain_W = TD_plain_Interp_solve ConstrSys W"
value "(solution_plain_W X, solution_plain_W Y, solution_plain_W Z, solution_plain_W
W)"

```

## 5.4 Solve the Equation System with TD

```

definition solutionX where "solutionX = TD_Interp_solve ConstrSys X"
value "((snd solutionX) X, (snd solutionX) Y, (snd solutionX) Z, (snd
solutionX) W)"

```

```

definition solutionY where "solutionY = TD_Interp_solve ConstrSys Y"
value "((snd solutionY) X, (snd solutionY) Y, (snd solutionY) Z, (snd
solutionY) W)"

```

```

definition solutionZ where "solutionZ = TD_Interp_solve ConstrSys Z"
value "((snd solutionZ) X, (snd solutionZ) Y, (snd solutionZ) Z, (snd
solutionZ) W)"

```

```

definition solutionW where "solutionW = TD_Interp_solve ConstrSys W"
value "((snd solutionW) X, (snd solutionW) Y, (snd solutionW) Z, (snd
solutionW) W)"

```

**end**

## References

- [1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.
- [2] M. Hofmann, A. Karbyshev, and H. Seidl. What is a pure functional? In S. Abramsky, C. Gavoille, C. Kirchner, F. M. auf der Heide, and P. G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2010.
- [3] S. Tilscher, Y. Stade, M. Schwarz, R. Vogler, and H. Seidl. The Top-Down Solver—An Exercise in A<sup>2</sup>I. In V. Arceri, A. Cortesi, P. Ferrara, and M. Olliaro, editors, *Challenges of Software Verification*, volume 238, pages 157–179. Springer Nature Singapore, Singapore, 2023.