Partial Correctness of the Top-Down Solver

Yannick Stade*, Sarah Tilscher*, Helmut Seidl

March 17, 2025

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

1 Introduction **3**

2 Preliminaries **4**
- 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 . . . 7
- 2.5 Dependencies between Unknowns . . . 8
- 2.6 Set *Reach* . . . 9
- 2.7 Partial solution . . . 15

3 The plain Top-Down Solver **16**
- 3.1 Definition of the Solver Algorithm . . . 16
- 3.2 Refinement of Auto-Generated Rules . . . 17
- 3.3 Domain Lemmas . . . 18
- 3.4 Case Rules . . . 18
- 3.5 Predicate for Valid Input States . . . 20
- 3.6 Partial Correctness Proofs . . . 21
- 3.7 Termination of TD_plain for Stable Unknowns . . . 25
- 3.8 Program Refinement for Code Generation . . . 27

4 The Top-Down Solver **33**
- 4.1 Definition of Destabilize and Proof of its Termination . . . 33
- 4.2 Definition of the Solver Algorithm . . . 37
- 4.3 Refinement of Auto-Generated Rules . . . 38
- 4.4 Domain Lemmas . . . 39
- 4.5 Case Rules . . . 40
- 4.6 Description of the Effect of Destabilize . . . 42
- 4.7 Predicate for Valid Input States . . . 47
- 4.8 Auxiliary Lemmas for Partial Correctness Proofs . . . 47
- 4.9 Preservation of the Invariant . . . 49
- 4.10 TD_plain and TD Equivalence . . . 52
- 4.11 Partial Correctness of the TD . . . 67
- 4.12 Program Refinement for Code Generation . . . 67

5 Example **75**
- 5.1 Definition of the Domain . . . 76
- 5.2 Definition of the Equation System . . . 77
- 5.3 Solve the Equation System with TD_plain . . . 77
- 5.4 Solve the Equation System with TD . . . 78

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 ⇒ ('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 ⊥ exists among the values.

```
locale Solver =
  fixes D :: "'d :: bot"
    and T :: "'x ⇒ ('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 ⇒ v | None ⇒
d)"

abbreviation slookup where
  "slookup infl x ≡ set (fmlookup_default infl [] x)"

definition mlup where
  "mlup σ x ≡ case σ x of Some v ⇒ v | None ⇒ ⊥"
```

```
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"
  using assms that unfolding fmlookup_default_def
  by (cases "fmlookup infl x"; auto)

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"
  using assms that unfolding fmlookup_default_def
  by (cases "fmlookup infl x"; auto)

lemma slookup_helper[simp]:
  assumes "fmlookup m x = Some ys"
    and "y ∈ set ys"
  shows "y ∈ slookup m x"
  using assms(1,2) notin_fmlookup_default_cases by force

lemma lookup_implies_mlup:
  assumes "σ x = σ' x'"
  shows "mlup σ x = mlup σ' x'"
  using assms
  unfolding mlup_def fmlookup_default_def
  by auto

lemma fmlookup_fminsert:
  assumes "fmlookup_default infl [] x = xs"
  shows "fmlookup (fminsert infl x y) x = Some (y # xs)"
proof(cases "fmlookup infl x")
  case None
  then show ?thesis using assms unfolding fmlookup_default_def fminsert_def
by auto
next
  case (Some a)
  then show ?thesis using assms unfolding fmlookup_default_def fminsert_def
by auto
qed

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"
  using that fmlookup_fminsert
  by fastforce
```

```
lemma fmlookup_default_drop_set:
  "fmlookup_default (fmdrop_set A m) [] x = (if x ∉ A then fmlookup_default
m [] x else [])"
  by (simp add: fmlookup_default_def)

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"
  using assms
  by (simp add: mlup_def)
```

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 σ.

```
function (domintros) rhs_length where
  "rhs_length (Answer d) _ = 0" |
  "rhs_length (Query x f) σ = 1 + rhs_length (f (mlup σ x)) σ"
  by pat_completeness auto

termination rhs_length
proof (rule allI, safe)
  fix t :: "('a, 'b) strategy_tree" and σ :: "('a, 'b) map"
  show "rhs_length_dom (t, σ)"
    by (induction t, auto simp add: rhs_length.domintros)
qed
```

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 σ

```
function (domintros) traverse_rhs where
  "traverse_rhs (Answer d) _ = d" |
  "traverse_rhs (Query x f) σ = traverse_rhs (f (mlup σ x)) σ"
  by pat_completeness auto

termination traverse_rhs
  by (relation "measure (λ(t,σ). rhs_length t σ)") auto
```

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

```
definition eq :: "'x ⇒ ('x, 'd) map ⇒ 'd" where
  "eq x σ = traverse_rhs (T x) σ"
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 σ).

```
inductive_set subt_aux ::
    "('x, 'd) map ⇒ ('x, 'd) strategy_tree ⇒ ('x, 'd) strategy_tree
set" for σ t where
  base: "t ∈ subt_aux σ t"
| step: "t' ∈ subt_aux σ t ⟹ t' = Query y g ⟹ (g (mlup σ y)) ∈ subt_aux
σ t"

definition subt where
  "subt σ x = subt_aux σ (T x)"

lemma subt_of_answer_singleton:
  shows "subt_aux σ (Answer d) = {Answer d}"
proof (intro set_eqI iffI, goal_cases)
  case (1 x)
  then show ?case by (induction rule: subt_aux.induct; simp)
next
  case (2 x)
  then show ?case by (simp add: subt_aux.base)
qed

lemma subt_transitive:
  assumes "t' ∈ subt_aux σ t"
  shows "subt_aux σ t' ⊆ subt_aux σ t"
proof
  fix τ
  assume "τ ∈ subt_aux σ t'"
  then show "τ ∈ subt_aux σ t"
    using assms
    by (induction rule: subt_aux.induct; simp add: subt_aux.step)
qed

lemma subt_unfold:
  shows "subt_aux σ (Query x f) = insert (Query x f) (subt_aux σ (f (mlup
σ x)))"
proof(intro set_eqI iffI, goal_cases)
  case (1 τ)
  then show ?case
    using subt_aux.simps
    by (induction rule: subt_aux.induct; blast)
next
  case (2 τ)
  then show ?case
  proof (elim insertE, goal_cases)
    case 1
    then show ?case
```

```
      using subt_aux.base
      by simp
  next
    case 2
    then show ?case
      using subt_transitive[of "f (mlup σ x)" σ "Query x f"] subt_aux.base
subt_aux.step
      by auto
  qed
qed
```

2.5 Dependencies between Unknowns

The set `dep σ x` collects all unknowns occuring in the right-hand side of `x` when traversing it with σ.

```
function dep_aux where
  "dep_aux σ (Answer d) = {}"
| "dep_aux σ (Query y g) = insert y (dep_aux σ (g (mlup σ y)))"
  by pat_completeness auto

termination dep_aux
  by (relation "measure (λ(σ, t). rhs_length t σ)") auto

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"
  using assms
  by (induction t rule: strategy_tree.induct) auto

lemmas dep_eq = dep_aux_eq[of σ "T x" σ' for σ x σ', folded dep_def]

lemma subt_implies_dep:
  assumes "Query y g ∈ subt_aux σ t"
  shows "y ∈ dep_aux σ t"
  using assms subt_of_answer_singleton subt_unfold
  by (induction t) auto

lemma solution_sufficient:
  assumes "∀y ∈ dep σ x. mlup σ y = mlup σ' y"
  shows "eq x σ = eq x σ'"
proof -
  obtain xd where xd_def: "eq x σ = xd" by simp
  have "traverse_rhs t σ' = xd"
    if "t ∈ subt σ x"
      and "traverse_rhs t σ = xd"
    for t
```

```
    using that
  proof(induction t rule: strategy_tree.induct)
    case (Query y g)
    define t where [simp]: "t = g (mlup σ y)"
    have "traverse_rhs t σ' = xd"
      using subt_aux.step Query.prems Query.IH
      by (simp add: subt_def)
    then show ?case
      using subt_implies_dep[where ?t="T x", folded subt_def dep_def]
Query.prems(1) assms(1)
      by simp
  qed simp
  then show ?thesis
    using assms subt_aux.base xd_def
    unfolding eq_def subt_def
    by simp
qed

corollary eq_mupd_no_dep:
  assumes "x ∉ dep σ y"
  shows "eq y σ = eq y (σ (x ↦ xd))"
  using assms solution_sufficient fmupd_lookup
  unfolding fmlookup_default_def mlup_def
  by simp
```

2.6 Set *Reach*

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

```
inductive_set reach for σ x where
  base: "x ∈ reach σ x"
| step: "y ∈ reach σ x ⟹ z ∈ dep σ y ⟹ z ∈ reach σ 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 σ c t where
  base: "x ∈ dep_aux σ t ⟹ x ∈ reach_cap_tree σ c t"
| step: "y ∈ reach_cap_tree σ c t ⟹ y ∉ c ⟹ z ∈ dep σ y ⟹ z ∈
reach_cap_tree σ c t"

abbreviation "reach_cap σ c x
  ≡ insert x (if x ∈ c then {} else reach_cap_tree σ (insert x c) (T
x))"
```

```
lemma reach_cap_tree_answer_empty[simp]:
  "reach_cap_tree σ c (Answer d) = {}"
proof (intro equals0I, goal_cases)
  case (1 y)
  then show ?case by (induction rule: reach_cap_tree.induct; simp)
qed

lemma dep_subset_reach_cap_tree:
  "dep_aux σ' t ⊆ reach_cap_tree σ' c t"
proof(intro subsetI, goal_cases)
  case (1 x)
  then show ?case using reach_cap_tree.base
    by (induction rule: dep_aux.induct; auto)
qed

lemma reach_cap_tree_subset:
  shows "reach_cap_tree σ c t ⊆ reach_cap_tree σ (c - {x}) t"
proof
  fix xa
  show "xa ∈ reach_cap_tree σ c t ⟹ xa ∈ reach_cap_tree σ (c - {x})
t"
  proof(induction rule: reach_cap_tree.induct)
    case base
    then show ?case
      using reach_cap_tree.base
      by simp
  next
    case (step y' z)
    then show ?case
      using reach_cap_tree.step
      by simp
  qed
qed

lemma reach_empty_capped:
  shows "reach σ x = insert x (reach_cap_tree σ {x} (T x))"
proof(intro equalityI subsetI, goal_cases)
  case (1 y)
  then show ?case
  proof(induction rule: reach.induct)
    case (step y z)
    then show ?case using reach_cap_tree.base[of z σ "T x"] reach_cap_tree.step[of
y σ "{x}"]
      unfolding dep_def by blast
  qed simp
next
  case (2 y)
  then show ?case
```

```
    using reach.base
  proof(cases "y = x")
    case False
    then have "y ∈ reach_cap_tree σ {x} (T x)"
      using 2
      by simp
    then show ?thesis
    proof(induction rule: reach_cap_tree.induct)
      case (base y)
      then show ?case
        using reach.base reach.step[of x]
        unfolding dep_def
        by auto
    next
      case (step y z)
      then show ?case
        using reach.step
        by blast
    qed
  qed simp
qed

lemma dep_aux_implies_reach_cap_tree:
  assumes "y ∉ c"
    and "y ∈ dep_aux σ t"
  shows "reach_cap_tree σ c (T y) ⊆ reach_cap_tree σ c t"
proof
  fix xa
  assume "xa ∈ reach_cap_tree σ c (T y)"
  then show "xa ∈ reach_cap_tree σ c t"
  proof(induction rule: reach_cap_tree.induct)
    case (base x)
    then show ?case
      using assms reach_cap_tree.base reach_cap_tree.step[unfolded dep_def,
of y]
      by simp
  next
    case (step y z)
    then show ?case
      using reach_cap_tree.step
      by simp
  qed
qed

lemma reach_cap_tree_simp:
  shows "reach_cap_tree σ c t
    = dep_aux σ t ∪ (⋃ξ∈dep_aux σ t - c. reach_cap_tree σ (insert ξ
c) (T ξ))"
proof (intro set_eqI iffI, goal_cases)
```

```
  case (1 x)
  then show ?case
  proof (induction rule: reach_cap_tree.induct)
    case (base x)
    then show ?case using reach_cap_tree.step by auto
  next
    case (step y z)
    then show ?case using reach_cap_tree.step[of y σ] reach_cap_tree.base[of
z σ "T y"]
      unfolding dep_def
      by blast
  qed
next
  case (2 x)
  then show ?case
  proof (elim UnE, goal_cases)
    case 1
    then show ?case using reach_cap_tree.base by simp
  next
    case 2
    then obtain y where "x ∈ reach_cap_tree σ (insert y c) (T y)" and
"y ∈ dep_aux σ t - c" by auto
    then show ?case
    using dep_aux_implies_reach_cap_tree[of y c] reach_cap_tree_subset[of
σ "insert y c" "T y" y]
    by auto
  qed
qed

lemma reach_cap_tree_step:
  assumes "mlup σ y = yd"
  shows "reach_cap_tree σ c (Query y g) = insert y (if y ∈ c then {}
    else reach_cap_tree σ (insert y c) (T y)) ∪ reach_cap_tree σ c (g
yd)"
  using assms reach_cap_tree_simp[of σ c]
  by auto

lemma reach_cap_tree_eq:
  assumes "∀x∈reach_cap_tree σ c t. mlup σ x = mlup σ' x"
  shows "reach_cap_tree σ c t = reach_cap_tree σ' c t"
proof(intro equalityI subsetI, goal_cases)
  case (1 x)
  then show ?case
  proof(induction rule: reach_cap_tree.induct)
    case (base x)
    then show ?case
      using assms reach_cap_tree.base[of _ σ t c] dep_aux_eq reach_cap_tree.base[of
x σ' t c]
      by metis
```

```
  next
    case (step y z)
    then show ?case
      using assms reach_cap_tree.step[of y σ c t] dep_eq reach_cap_tree.step[of
y σ' c t z]
      by blast
  qed
next
  case (2 x)
  then show ?case
  proof(induction rule: reach_cap_tree.induct)
    case (base x)
    then show ?case
      using assms reach_cap_tree.base[of _ σ t c] dep_aux_eq reach_cap_tree.base[of
x σ' t c]
      by metis
  next
    case (step y z)
    then show ?case
      using assms reach_cap_tree.step[of y σ c t] dep_eq reach_cap_tree.step[of
y σ' c t z]
      by blast
  qed
qed

lemma reach_cap_tree_simp2:
  shows "insert x (if x ∈ c then {} else reach_cap_tree σ c (T x)) =
        insert x (if x ∈ c then {} else reach_cap_tree σ (insert x c)
(T x))"
proof(cases "x ∈ c" rule: case_split[case_names called not_called])
  case not_called
  moreover have "insert x (reach_cap_tree σ (insert x c) (T x))
    = insert x (reach_cap_tree σ c (T x))"
  proof(intro equalityI subsetI, goal_cases)
    case (1 y)
    then show ?case
    proof(cases "x = y")
      case False
      then show ?thesis
        by (metis "1" Diff_insert_absorb in_mono insert_mono not_called
reach_cap_tree_subset)
    qed auto
  next
    case (2 y)
    then show ?case
    proof(cases "x = y")
      case False
      then show ?thesis
      proof(cases "y ∈ dep σ x" rule: case_split[case_names xdep no_xdep])
```

```
        case xdep
        then show ?thesis using 2 reach_cap_tree.base[of y σ "T x" "insert
x c", folded dep_def]
          by auto
      next
        case no_xdep
        have "y ∈ reach_cap_tree σ c (T x)" using 2 False by auto
        then show ?thesis
        proof (induction rule: reach_cap_tree.induct)
          case (base x)
          then show ?case by (simp add: reach_cap_tree.base)
        next
          case (step y z)
          then show ?case using reach_cap_tree.step reach_cap_tree.base
dep_def by blast
        qed
      qed
    qed auto
  qed
  then show ?thesis by auto
qed auto

lemma dep_closed_implies_reach_cap_tree_closed:
  assumes "x ∈ s"
    and "∀ξ∈s - (c - {x}). dep σ' ξ ⊆ s"
  shows "reach_cap σ' (c - {x}) x ⊆ s"
proof (intro subsetI, goal_cases)
  case (1 y)
  then show ?case using assms
  proof(cases "x = y")
    case False
    then have "y ∈ reach_cap_tree σ' (c - {x}) (T x)"
      using 1 reach_cap_tree_simp2[of x "c - {x}" σ'] by auto
    then show ?thesis using assms
    proof(induction)
      case (base y)
      then show ?case using base.hyps dep_def by auto
    next
      case (step y z)
      then show ?case by (metis (no_types, lifting) Diff_iff insert_subset
mk_disjoint_insert)
    qed
  qed simp
qed

lemma reach_cap_tree_subset2:
  assumes "mlup σ y = yd"
  shows "reach_cap_tree σ c (g yd) ⊆ reach_cap_tree σ c (Query y g)"
  using reach_cap_tree_step[OF assms] by blast
```

```
lemma reach_cap_tree_subset_subt:
  assumes "t' ∈ subt_aux σ t"
  shows "reach_cap_tree σ c t' ⊆ reach_cap_tree σ c t"
  using assms
proof(induction rule: subt_aux.induct)
  case (step t' y g)
  then show ?case using reach_cap_tree_step by simp
qed simp

lemma reach_cap_tree_singleton:
  assumes "reach_cap_tree σ (insert x c) t ⊆ {x}"
  obtains (Answer) d where "t = Answer d"
  | (Query) f where "t = Query x f"
    and "dep_aux σ t = {x}"
  using assms that(1)
proof(cases t)
  case (Query x' f)
  then have "x' ∈ reach_cap_tree σ (insert x c) t"
    using reach_cap_tree.base dep_aux.simps(2) by simp
  then have [simp]: "x' = x" using assms by auto
  then show ?thesis
    using assms that(2) reach_cap_tree.base Query dep_subset_reach_cap_tree
subset_antisym
    by fastforce
qed simp
```

2.7 Partial solution

Finally, we define an unknown-to-value mapping σ to be a partial solution over a set of unknowns `vars` if for every unknown in `vars`, the value obtained from an evaluation of its right-hand side function `eq x` with σ matches the value stored in σ.

```
abbreviation part_solution where
  "part_solution σ vars ≡ (∀x ∈ vars. eq x σ = mlup σ x)"

lemma part_solution_coinciding_sigma_called:
  assumes "part_solution σ (s - c)"
    and "∀x ∈ s. mlup σ x = mlup σ' x"
    and "∀x ∈ s - c. dep σ x ⊆ s"
  shows "part_solution σ' (s - c)"
  using assms
proof(intro ballI, goal_cases)
  case (1 x)
  then have "∀y∈dep σ x. mlup σ y = mlup σ' y" by blast
  then show ?case using 1 solution_sufficient[of σ x σ'] by simp
qed
```

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 ⇒ ('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 σ 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))"
  by pat_completeness auto

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 σ for x c σ
    ]
    query_iterate_eval.pinduct(3)[
      folded query_dom_def iterate_dom_def eval_dom_def,
      of x t c σ for x t c σ
    ]

lemmas iterate_pinduct[consumes 1, case_names Iterate]
```

```
  = query_iterate_eval_pinduct(2)[where ?P="λx y c σ. True" and ?R="λx
t c σ. 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 σ
    ⟹ y ∉ c ⟹ iterate_dom y (insert y c) σ"
  and "iterate_dom x c σ
    ⟹ (eval_dom x (T x) c σ ∧
        (eval x (T x) c σ = (xd_new, σ')
          ⟶ mlup σ' x = xd_old ⟶ xd_new ≠ xd_old ⟶
          iterate_dom x c (σ'(x ↦ xd_new))))"
  and "eval_dom x (Query y g) c σ
    ⟹ (query_dom x y c σ ∧ (query x y c σ = (yd, σ') ⟶ eval_dom x
(g yd) c σ'))"
proof (induction x y c σ and x c σ and x "Query y g" c σ
    arbitrary: and xd_new xd_old σ' and y g yd σ'
    rule: query_iterate_eval_pinduct)
  case (Query x c σ)
  then show ?case
    using query_iterate_eval.domintros(2) by fastforce
next
  case (Iterate x c σ)
  then show ?case
    using query_iterate_eval.domintros(2,3)[folded eval_dom_def iterate_dom_def
query_dom_def]
    by metis
next
  case (Eval c σ)
  then show ?case
    using query_iterate_eval.domintros(1,3) by simp
qed
```

3.4 Case Rules

```
lemma iterate_continue_fixpoint_cases[consumes 3]:
  assumes "iterate_dom x c σ"
    and "iterate x c σ = (xd, σ')"
    and "x ∈ c"
  obtains (Fixpoint) "eval_dom x (T x) c σ"
    and "eval x (T x) c σ = (xd, σ')"
    and "mlup σ' x = xd"
  | (Continue) σ1 xd_new
  where "eval_dom x (T x) c σ"
```

```
    and "eval x (T x) c σ = (xd_new, σ1)"
    and "mlup σ1 x ≠ xd_new"
    and "iterate_dom x c (σ1(x ↦ xd_new))"
    and "iterate x c (σ1(x ↦ xd_new)) = (xd, σ')"
proof -
 obtain xd_new σ1
    where "eval x (T x) c σ = (xd_new, σ1)"
    by (cases "eval x (T x) c σ")
  then show ?thesis
    using assms that dom_backwards_pinduct(2)
    by (cases "mlup σ1 x = xd_new"; simp)
qed

lemma iterate_fmlookup:
  assumes "iterate_dom x c σ"
    and "iterate x c σ = (xd, σ')"
    and "x ∈ c"
  shows "mlup σ' x = xd"
  using assms
proof(induction rule: iterate_pinduct)
  case (Iterate x c σ)
  show ?case
    using Iterate.hyps Iterate.prems
  proof (cases rule: iterate_continue_fixpoint_cases)
    case (Continue σ1 xd_new)
    then show ?thesis
      using Iterate.prems(2) Iterate.IH
      by fastforce
  qed simp
qed

corollary query_fmlookup:
  assumes "query_dom x y c σ"
    and "query x y c σ = (yd, σ')"
  shows "mlup σ' y = yd"
  using assms iterate_fmlookup dom_backwards_pinduct(1)[of x y c σ]
  by (auto split: if_splits)

lemma query_iterate_lookup_cases [consumes 2]:
  assumes "query_dom x y c σ"
    and "query x y c σ = (yd, σ')"
  obtains (Iterate)
        "iterate_dom y (insert y c) σ"
    and "iterate y (insert y c) σ = (yd, σ')"
    and "mlup σ' y = yd"
    and "y ∉ c"
  | (Lookup) "mlup σ y = yd"
    and "σ = σ'"
    and "y ∈ c"
```

```
  using assms that dom_backwards_pinduct(1) query_fmlookup[of x y c σ
yd σ']
  by (cases "y ∈ c"; auto)

lemma eval_query_answer_cases [consumes 2]:
  assumes "eval_dom x t c σ"
    and "eval x t c σ = (d, σ')"
  obtains (Query) y g yd σ1
  where "t = Query y g"
    and "query_dom x y c σ"
    and "query x y c σ = (yd, σ1)"
    and "eval_dom x (g yd) c σ1"
    and "eval x (g yd) c σ1 = (d, σ')"
    and "mlup σ1 y = yd"
  | (Answer) "t = Answer d"
    and "σ = σ'"
  using assms dom_backwards_pinduct(3) that query_fmlookup
  by (cases t; auto split: prod.splits)
```

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 σ 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 σ 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 σ ≡ (∀ξ∈s - c. dep σ ξ ⊆ s) ∧ part_solution σ (s -
c)"

lemma invariant_simp:
  assumes "x ∈ c"
    and "invariant s (c - {x}) σ"
  shows "invariant (insert x s) c σ"
  using assms
proof -
  have "c - {x} ⊆ s ≡ c ⊆ insert x s"
    using assms(1)
    by (simp add: subset_insert_iff)
  moreover have "s - (c - {x}) ⊇ insert x s - c"
```

```
    using assms(1)
    by auto
  ultimately show ?thesis
    using assms(2)
    unfolding invariant_def
    by fastforce
qed

lemma invariant_continue:
  assumes "x ∉ s"
    and "invariant s c σ"
    and "∀y∈s. mlup σ y = mlup σ1 y"
  shows "invariant s c (σ1(x ↦ xd))"
proof -
  show ?thesis
  using assms mlup_eq_mupd_set[OF assms(1,3)] unfolding invariant_def
  proof(intro conjI, goal_cases)
    case 1 then show ?case using dep_eq by blast
  next
    case 2 then show ?case using part_solution_coinciding_sigma_called
      by (metis DiffD1 solution_sufficient subsetD)
  qed
qed
```

3.6 Partial Correctness Proofs

```
lemma x_not_stable:
  assumes "eq x σ ≠ mlup σ x"
    and "part_solution σ s"
  shows "x ∉ s"
  using assms by auto
```

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 σ
    ⟹ query x y c σ = (yd, σ')
    ⟹ y ∈ s
    ⟹ invariant s c σ
    ⟹ σ = σ'"
    and "iterate_dom x c σ
    ⟹ iterate x c σ = (xd, σ')
    ⟹ x ∈ c
    ⟹ x ∈ s
    ⟹ invariant s (c - {x}) σ
    ⟹ σ = σ'"
    and "eval_dom x t c σ
    ⟹ eval x t c σ = (xd, σ')
```

```
    ⟹ dep_aux σ t ⊆ s
    ⟹ invariant s c σ
    ⟹ traverse_rhs t σ' = xd ∧ σ = σ'"
proof(induction arbitrary: yd s σ' and xd s σ' and xd s σ' rule: query_iterate_eval_pindu
  case (Query x y c σ)
  show ?case using Query.IH(1) Query.prems Query.IH(2)
    by (cases rule: query_iterate_lookup_cases; simp)
next
  case (Iterate x c σ)
  show ?case using Iterate.IH(1) Iterate.prems(1,2)
  proof(cases rule: iterate_continue_fixpoint_cases)
    case Fixpoint
    then show ?thesis
      using Iterate.prems(3,4) Iterate.IH(2)[of _ _ "insert x s"]
        invariant_simp[OF Iterate.prems(2,4)]
      unfolding dep_def invariant_def by auto
  next
    case (Continue σ1 xd')
    show ?thesis
    proof(rule ccontr)
      have IH: "eq x σ1 = xd' ∧ σ = σ1"
        using Iterate.prems(2-4) Iterate.IH(2)[OF Continue(2), of s]
          invariant_simp[OF Iterate.prems(2,4)] unfolding dep_def invariant_def
by auto
      then show False
        using Iterate.prems(2-4) Continue(3) unfolding invariant_def by
simp
    qed
  qed
next
  case (Eval x t c σ)
  show ?case using Eval.IH(1) Eval.prems(1)
  proof(cases rule: eval_query_answer_cases)
    case (Query y g yd σ1)
    then show ?thesis using Eval.prems(1-3) Eval.IH(1) Eval.IH(2)[OF
Query(1,3)]
        Eval.IH(3)[OF Query(1) Query(3)[symmetric] _ Query(5)]
      by auto
  qed simp
qed
```

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 σ
    ⟹ query x y c σ = (yd, σ')
    ⟹ invariant s c σ
```

```
    ⟹ invariant (s ∪ reach_cap σ' c y) c σ'
      ∧ (∀ξ ∈ s. mlup σ ξ = mlup σ' ξ)"
    and "iterate_dom x c σ
    ⟹ iterate x c σ = (xd, σ')
    ⟹ x ∈ c
    ⟹ invariant s (c - {x}) σ
    ⟹ invariant (s ∪ (reach_cap σ' (c - {x}) x)) (c - {x}) σ'
      ∧ (∀ξ ∈ s. mlup σ ξ = mlup σ' ξ)"
    and "eval_dom x t c σ
    ⟹ eval x t c σ = (xd, σ')
    ⟹ invariant s c σ
    ⟹ invariant (s ∪ reach_cap_tree σ' c t) c σ'
      ∧ (∀ξ ∈ s. mlup σ ξ = mlup σ' ξ)
      ∧ traverse_rhs t σ' = xd"
proof(induction arbitrary: yd s σ' and xd s σ' and xd s σ' rule: query_iterate_eval_pindu
  case (Query x y c σ)
  show ?case
    using Query.IH(1) Query.prems(1)
  proof (cases rule: query_iterate_lookup_cases)
    case Iterate
    note IH = Query.IH(2)[simplified, OF Iterate(4,2) Query.prems(2)]
    then show ?thesis
      using Iterate(4) by simp
  next
    case Lookup
    then show ?thesis
      using Query.prems(2) unfolding invariant_def by auto
  qed
next
  case (Iterate x c σ)
  show ?case
    using Iterate.IH(1) Iterate.prems(1,2)
  proof(cases rule: iterate_continue_fixpoint_cases)
    case Fixpoint
    note IH = Iterate.IH(2)[OF Fixpoint(2) invariant_simp[OF Iterate.prems(2,3)],
folded eq_def]
    then show ?thesis
      using Fixpoint(3) Iterate.prems(2) reach_cap_tree_simp2[of x "c
- {x}"]
        dep_subset_reach_cap_tree[of σ' "T x", folded dep_def]
      unfolding invariant_def
      by (auto simp add: insert_absorb)
  next
    case (Continue σ1 xd')
    note IH = Iterate.IH(2)[OF Continue(2) invariant_simp[OF Iterate.prems(2,3)]]

    have "part_solution σ1 (s - (c - {x}))"
      using part_solution_coinciding_sigma_called[of s "c - {x}" σ σ1]
IH Iterate.prems(3)
```

```
      unfolding invariant_def
      by simp
    then have x_not_stable: "x ∉ s"
      using x_not_stable[of x σ1 s] IH Continue(3)
      by auto
    then have inv: "invariant s (c - {x}) (σ1(x ↦ xd'))"
      using IH invariant_continue[OF x_not_stable Iterate.prems(3)] by
blast

    note ih = Iterate.IH(3)[OF Continue(2)[symmetric] _ Continue(3)[symmetric]
Continue(5)
        Iterate.prems(2) inv, simplified]
    then show ?thesis
      using IH mlup_eq_mupd_set[OF x_not_stable, of σ]
      unfolding mlup_def
      by auto
  qed
next
  case (Eval x t c σ)
  show ?case using Eval.IH(1) Eval.prems(1)
  proof(cases rule: eval_query_answer_cases)
    case (Query y g yd σ1)
    note IH = Eval.IH(2)[OF Query(1,3) Eval.prems(2)]
    note ih = Eval.IH(3)[OF Query(1) Query(3)[symmetric] _ Query(5) conjunct1[OF
IH], simplified]
    show ?thesis
      using Query IH ih reach_cap_tree_step reach_cap_tree_eq[of σ1 "insert
y c" "T y" σ']
      by (auto simp add: Un_assoc)
  next
    case Answer
    then show ?thesis
      using Eval.prems(2) by simp
  qed
qed
```

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 = σ"
  shows "part_solution σ (reach σ x)"
proof -
  obtain xd where "iterate x {x} Map.empty = (xd, σ)"
    using assms(2) unfolding solve_def by (auto split: prod.splits)
  then show ?thesis
  using assms(1) partial_correctness_ind(2)[of x "{x}" Map.empty xd σ
```

```
"{}"] reach_empty_capped
  unfolding solve_dom_def invariant_def by simp
qed
```

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(goal_cases)
  have "reach_cap σ (c - {x}) x ⊆ s"
    using assms(1,2) dep_closed_implies_reach_cap_tree_closed unfold-
ing invariant_def by simp
  from finite_subset[OF this] have "finite (reach_cap σ (c - {x}) x -
(c - {x}))"
    using assms(4) by simp+
  then have goal: "iterate_dom x c σ ∧ iterate x c σ = (xd, σ)" us-
ing assms(1-3,5)
  proof(induction "reach_cap σ (c - {x}) x - (c - {x})"
      arbitrary: x c xd rule: finite_psubset_induct)
    case psubset
    have "eval_dom x t c σ ∧ (traverse_rhs t σ, σ) = eval x t c σ" if
"t ∈ subt σ x" for t
      using that
    proof(induction t)
      case (Answer _)
      then show ?case
        using query_iterate_eval.domintros(3)[folded query_dom_def iterate_dom_def
eval_dom_def]
        by fastforce
    next
      case (Query y g)
      have "reach_cap_tree σ (insert x (c - {x})) (T x) ⊆ s"
        using dep_closed_implies_reach_cap_tree_closed[OF psubset.prems(1),
of c σ]
          psubset.prems(2)[unfolded invariant_def]
        by auto
      then have y_stable: "y ∈ s"
```

```
        using dep_subset_reach_cap_tree subt_implies_dep[OF Query(2)[unfolded
subt_def]]
        by blast
      show ?case
      proof(cases "y ∈ c" rule: case_split[case_names called not_called])
        case called
        then have dom: "query_dom x y c σ"
          using query_iterate_eval.domintros(1)[folded query_dom_def]
by auto
        moreover have query_val: "(mlup σ y, σ) = query x y c σ"
          using called already_solution(1) partial_correctness_ind(1)
          by (metis query.psimps query_iterate_eval.domintros(1))
        ultimately have "eval_dom x (Query y g) c σ"
          using Query.IH[of "g (mlup σ y)"]
            query_iterate_eval.domintros(3)[folded dom_defs, of "Query
y g" x c σ] Query.prems
            subt_aux.step subt_def
          by fastforce
        have "g (mlup σ y) ∈ subt_aux σ (T x)"
          using Query.prems subt_aux.step subt_def by blast
        then have "eval_dom x (g (mlup σ y)) c σ"
            and "(traverse_rhs (g (mlup σ y)) σ, σ) = eval x (g (mlup
σ y)) c σ"
          using Query.IH unfolding subt_def by auto
        then show ?thesis
          using ‹eval_dom x (Query y g) c σ› query_val
          by (auto split: strategy_tree.split prod.split)
      next
        case not_called
        then obtain yd where lupy: "mlup σ y = yd" and eqy: "eq y σ
= yd"
          using y_stable psubset.prems(2) unfolding invariant_def by auto
        have ih: "eval_dom x (g (mlup σ y)) c σ"
            and "(traverse_rhs (g (mlup σ y)) σ, σ) = eval x (g (mlup
σ y)) c σ"
          using Query.IH[of "g (mlup σ y)"] Query.prems subt_aux.step
subt_def by auto
        moreover have "reach_cap σ c y ⊆ reach_cap σ (c - {x}) x"
          using not_called psubset.prems(4) reach_cap_tree_step[of σ y
yd c g, OF lupy]
            reach_cap_tree_subset_subt[of "Query y g" σ "T x" c, folded
subt_def, OF Query.prems]
          by (simp add: insert_absorb subset_insertI2)
        then have f_def: "reach_cap σ c y - c ⊂ reach_cap σ (c - {x})
x - (c - {x})"
          using psubset.prems(4)
          by blast
        have "invariant s (c - {y}) σ"
          using psubset.prems(2) not_called psubset.prems(1) invariant_simp
```

```
          by (metis Diff_empty Diff_insert0 insert_absorb)
        then have IH: "iterate_dom y (insert y c) σ ∧ iterate y (insert
y c) σ = (yd, σ)"
          using f_def y_stable not_called lupy psubset.hyps(2)[of y "c
- {y}" yd] psubset.hyps(2)
          by (metis Diff_idemp Diff_insert_absorb insertCI )
        then have "query_dom x y c σ ∧ (mlup σ y, σ) = query x y c σ"
          using not_called lupy query_iterate_eval.domintros(1)[folded
dom_defs, of y c σ]
          by simp
        ultimately show ?thesis
          using query_iterate_eval.domintros(3)[folded dom_defs, of "Query
y g" x c σ] by fastforce
      qed
    qed
    note IH = this[of "T x", folded eq_def, OF subt_aux.base[of "T x"
σ, folded subt_def]]
    moreover have "eq x σ = mlup σ x" using psubset.prems(1,2) unfold-
ing invariant_def by auto
    moreover have "iterate_dom x c σ"
      using query_iterate_eval.domintros(2)[folded dom_defs, of x c σ]
IH ‹eq x σ = mlup σ x›
      by (metis Pair_inject)
    ultimately show ?case
      using iterate.psimps[folded dom_defs, of x c σ] psubset.prems(3)
      by (cases "eval x (T x) c σ") auto
  qed
  case 1 show ?case using goal ..
  case 2 show ?case using goal ..
qed
```

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 σ ⟹ solve_rec_c_dom (Q (x,y,c,σ))
    ∧ query x y c σ = the (solve_rec_c (Q (x,y,c,σ)))"
  and "iterate_dom x c σ ⟹ solve_rec_c_dom (I (x,c,σ))
    ∧ iterate x c σ = the (solve_rec_c (I (x,c,σ)))"
  and "eval_dom x t c σ ⟹ solve_rec_c_dom (E (x,t,c,σ))
    ∧ eval x t c σ = the (solve_rec_c (E (x,t,c,σ)))"
proof (induction x y c σ and x c σ and x t c σ rule: query_iterate_eval_pinduct)
  case (Query x y c σ)
  show ?case
  proof (cases "y ∈ c")
    case True
    then have "solve_rec_c (Q (x, y, c, σ)) = Some (mlup σ y, σ)" by simp
    moreover have "query x y c σ = (mlup σ y, σ)"
      using query.psimps[folded dom_defs] Query(1) True by force
    ultimately show ?thesis unfolding solve_rec_c_dom_def by auto
  next
    case False
    then have "query x y c σ = iterate y (insert y c) σ"
      using Query.IH(1) query.pelims[folded dom_defs] by fastforce
    then have "query x y c σ = the (solve_rec_c (Q (x, y, c, σ)))"
      using Query False False by simp
    moreover have "solve_rec_c_dom (Q (x, y, c, σ))"
      using Query(2) False unfolding solve_rec_c_dom_def by simp
```

```
    ultimately show ?thesis using Query unfolding solve_rec_c_dom_def
by auto
  qed
next
  case (Iterate x c σ)
  obtain d1 σ1 where eval: "eval x (T x) c σ = (d1, σ1)"
    and "solve_rec_c (E (x, T x, c, σ)) = Some (d1, σ1)" using Iterate(2)
solve_rec_c_dom_def by force
  show ?case
  proof (cases "d1 = mlup σ1 x")
    case True
    have "iterate x c σ = (d1, σ1)"
      using eval iterate.psimps[folded dom_defs, OF Iterate(1)] True by
simp
    then show ?thesis
      using solve_rec_c_dom_def dom_defs iterate.psimps Iterate by fastforce
  next
    case False
    then have "solve_rec_c_dom (I (x, c, σ1(x ↦ d1)))"
        and "iterate x c (σ1(x ↦ d1)) = the (solve_rec_c (I (x, c, σ1(x
↦ d1))))"
      using Iterate(3)[OF eval[symmetric] _ False] by blast+
    moreover have "iterate x c σ = iterate x c (σ1(x ↦ d1))"
      using eval iterate.psimps[folded dom_defs, OF Iterate(1)] False
by simp
    moreover have "solve_rec_c (I (x, c, σ1(x ↦ d1))) = solve_rec_c
(I (x, c, σ))"
      using False eval Iterate(2) solve_rec_c_dom_def by auto
    ultimately show ?thesis unfolding solve_rec_c_dom_def by auto
  qed
next
  case (Eval x t c σ)
  show ?case
  proof (cases t)
    case (Answer d)
    then have "eval x t c σ = (d, σ)"
      using eval.psimps query_iterate_eval.domintros(3) dom_defs(3)
      by fastforce
    then show ?thesis using Eval Answer unfolding solve_rec_c_dom_def
by simp
  next
    case (Query y g)
    then obtain d1 σ1 where "solve_rec_c (Q (x, y, c, σ)) = Some (d1,
σ1)"
        and "query x y c σ = (d1, σ1)"
      using Query Eval(2) unfolding solve_rec_c_dom_def by auto
    then have "solve_rec_c_dom (E (x, t, c, σ))"
        "eval x (g d1) c σ1 = the (solve_rec_c (E (x, t, c, σ)))"
      using Eval(3) Query unfolding solve_rec_c_dom_def by auto
```

```
    moreover have "eval x t c σ = eval x (g d1) c σ1"
      using Eval.IH(1) Query eval.psimps eval_dom_def
        ‹query x y c σ = (d1, σ1)›
      by fastforce
    ultimately show ?thesis by simp
  qed
qed

lemma solve_rec_c_query_iterate_eval_equiv:
  shows "solve_rec_c s = Some r ⟹ (case s of
        Q (x,y,c,σ) ⇒ query_dom x y c σ ∧ query x y c σ = r
      | I (x,c,σ) ⇒ iterate_dom x c σ ∧ iterate x c σ = r
      | E (x,t,c,σ) ⇒ eval_dom x t c σ ∧ eval x t c σ = r)"
proof (induction arbitrary: s r rule: solve_rec_c.fixp_induct)
  case 1
  then show ?case using option_admissible by fast
next
  case 2
  then show ?case by simp
next
  case (3 S)
  show ?case
  proof (cases s)
    case (Q a)
    obtain x y c σ where "a = (x, y, c, σ)" using prod_cases4 by blast
    have "query_dom x y c σ ∧ query x y c σ = r"
    proof (cases "y ∈ c")
      case True
      then have "Some (mlup σ y, σ) = Some r" using 3(2) Q ‹a = (x,
y, c, σ)› by simp
      then show ?thesis
        by (metis query.psimps query_dom_def
            query_iterate_eval.domintros(1) True option.inject)
    next
      case False
      then have "S (I (y, insert y c, σ)) = Some r"
        using 3(2) Q ‹a = (x, y, c, σ)› by auto
      then have "iterate_dom y (insert y c) σ ∧ iterate y (insert y c)
σ = r"
        using 3(1) unfolding iterate_dom_def by fastforce
      then show ?thesis using False
        by (simp add: query_iterate_eval.domintros(1))
    qed
    then show ?thesis using Q ‹a = (x, y, c, σ)› unfolding query_dom_def
by simp
  next
    case (I a)
    obtain x c σ where "a = (x, c, σ)" using prod_cases3 by blast
    then have IH1: "Option.bind (S (E (x, T x, c, σ)))
```

```
      (λ(d_new, σ).
        if d_new = mlup σ x then Some (d_new, σ)
        else S (I (x, c, σ(x ↦ d_new)))) = Some r"
      using 3(2) I by simp
    then obtain d_new σ1 where eval_some: "S (E (x, T x, c, σ)) = Some
(d_new, σ1)"
      using 3(2) I
      by (cases "S (E (x, T x, c, σ))") auto
    then have eval: "eval_dom x (T x) c σ ∧ eval x (T x) c σ = (d_new,
σ1)"
      using 3(1) unfolding eval_dom_def by force
    have "iterate_dom x c σ ∧ iterate x c σ = r"
    proof (cases "d_new = mlup σ1 x")
      case True
      then show ?thesis
        using eval IH1 dom_defs(2) dom_defs(3) iterate.psimps
          query_iterate_eval.domintros(2) eval_some
        by fastforce
    next
      case False
      then have "S (I (x, c, σ1(x ↦ d_new))) = Some r" using IH1 eval_some
by simp
      then have "iterate_dom x c (σ1(x ↦ d_new))
          ∧ iterate x c (σ1(x ↦ d_new)) = r"
        using 3(1) unfolding iterate_dom_def by fastforce
      then show ?thesis using eval False
        by (smt (verit, best) Pair_inject dom_defs(2) dom_defs(3)
            iterate.psimps query_iterate_eval.domintros(2) case_prod_conv)
    qed
    then show ?thesis using I ‹a = (x, c, σ)› unfolding iterate_dom_def
by simp
  next
    case (E a)
    obtain x t c σ where "a = (x, t, c, σ)" using prod_cases4 by blast
    then have "s = E (x, t, c, σ)" using E by auto
    have "eval_dom x t c σ ∧ eval x t c σ = r"
    proof (cases t)
      case (Answer d)
      then have "eval_dom x t c σ" unfolding eval_dom_def
        using query_iterate_eval.domintros(3) by fastforce
      moreover have "eval x t c σ = (d, σ)"
        by (smt (verit, del_insts) Answer eval_query_answer_cases calculation
            strategy_tree.distinct(1) strategy_tree.simps(1) surj_pair)
      moreover have "(d, σ) = r" using 3(2) ‹s = E (x, t, c, σ)› Answer
by simp
      ultimately show ?thesis by simp
    next
      case (Query y g)
      then have A: "Option.bind (S (Q (x, y, c, σ))) (λ(yd, σ). S (E
```

```
(x, g yd, c, σ)))
        = Some r" using ‹s = E (x, t, c, σ)› 3(2) by simp
      then obtain yd σ1 where S1: "S (Q (x, y, c, σ)) = Some (yd, σ1)"
          and S2: "S (E (x, g yd, c, σ1)) = Some r"
        by (cases "S (Q (x, y, c, σ))") auto
      then have "query_dom x y c σ ∧ query x y c σ = (yd, σ1)"
          and "eval_dom x (g yd) c σ1 ∧ eval x (g yd) c σ1 = r"
        using 3(1)[OF S1] 3(1)[OF S2] unfolding dom_defs by force+
      then show ?thesis
        using query_iterate_eval.domintros(3)[folded dom_defs, of t x
c σ] Query
        by fastforce
    qed
    then show ?thesis using E ‹a = (x, t, c, σ)› unfolding eval_dom_def
by simp
  qed
qed

theorem term_equivalence: "solve_dom x ⟷ solve_c_dom x"
  using query_iterate_eval_solve_rec_c_equiv(2)[of x "{x}" "λx. None"]
    solve_rec_c_query_iterate_eval_equiv[of "I (x, {x}, λx. None)"]
  unfolding solve_dom_def solve_c_dom_def solve_rec_c_dom_def solve_c_def
  by (cases "solve_rec_c (I (x, {x}, λx. None))") force+

theorem value_equivalence:
  "solve_dom x ⟹ ∃σ. solve_c x = Some σ ∧ solve x = σ"
proof goal_cases
  case 1
  then obtain r where "solve_rec_c (I (x, {x}, λx. None)) = Some r
    ∧ iterate x {x} (λx. None) = r"
    using query_iterate_eval_solve_rec_c_equiv(2)
    unfolding solve_rec_c_dom_def solve_dom_def
    by fastforce
  then show ?case unfolding solve_def solve_c_def by (auto split: prod.split)
qed
```

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 ⇒ r
  | None ⇒ Code.abort (String.implode ''Input not in domain'') (λ_. solve
x))"
proof (cases "solve_dom x")
  case True
  then show ?thesis unfolding solve_def solve_c_def
    by (metis solve_def solve_c_def option.simps(5) value_equivalence)
next
  case False
  then have "solve_c x = None" using solve_c_dom_def term_equivalence
```

```
by auto
  then show ?thesis by auto
qed

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
  done

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"
  by pat_completeness auto

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"
  using destab_iter_destab.domintros by auto

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 -
  obtain ys where ys_def: "ys = fmlookup_default m [] x" and ys_non_empty:
"ys ≠ []"
    using assms by simp
  then have in_map: "(x, ys) |∈| fset_of_fmap m"
    unfolding fmlookup_default_def
    by (cases "fmlookup m x"; auto)
  then have eq: "fset_of_fmap (fmdrop x m) = fset_of_fmap m |-| {|(x,
ys)|}"
    by (auto split: if_splits)
  then have "ffilter ((≠) [] ∘ snd) (fset_of_fmap (fmdrop x m))
```

```
      = (ffilter ((≠) [] ∘ snd) (fset_of_fmap m)) |-| {|(x, ys)|}" by
fastforce
  then show ?thesis
    unfolding count_non_empty_def
    using in_map ys_non_empty fcard_Suc_fminus1[of "(x, ys)"]
    by auto
qed

lemma count_non_empty_eq_fmdrop:
  assumes "fmlookup_default m [] x = []"
  shows "count_non_empty (fmdrop x m) = count_non_empty m"
proof -
  have "ffilter ((≠) [] ∘ snd) (fset_of_fmap (fmdrop x m))
      = (ffilter ((≠) [] ∘ snd) (fset_of_fmap m))"
    using assms
    unfolding fmlookup_default_def
    by (auto split: if_splits)
  thus ?thesis unfolding count_non_empty_def by simp
qed

termination
proof -
  {
    fix ys infl stabl
    have "destab_iter_dom ys infl stabl ∧ (destab_iter ys infl stabl
= (infl', stabl')
        ⟶ count_non_empty infl' ≤ count_non_empty infl)"
      for infl' stabl'
    proof(induction "count_non_empty infl" arbitrary: ys infl stabl infl'
stabl'
        rule: full_nat_induct)
      case 1
      then show ?case
      proof(induction ys arbitrary: infl stabl)
        case Nil
        then show ?case
          by (simp add: destab_iter.psimps(1) destab_iter_destab.domintros(1))
      next
        case (Cons y ys)
        have IH: "destab_iter_dom xa x xb ∧
            (destab_iter xa x xb = (xc, xd) ⟶ count_non_empty xc ≤
count_non_empty x)"
          if "Suc m ≤ count_non_empty infl" and "m = count_non_empty
x"
          for m x xa xb xc xd
        using Cons.prems that by blast
        show ?case
        proof(cases "fmlookup_default infl [] y = []")
          case True
```

```
        obtain infl1 stabl1 where inflstabl1: "destab y infl (stabl
- {y}) = (infl1, stabl1)"
          by fastforce
        have y_dom: "destab_dom y infl (stabl - {y})"
          using destab_domintros(1,3) True
          by auto
        have destab_y: "destab y infl (stabl - {y}) = (fmdrop y infl,
stabl - {y})"
          using destab.psimps[folded destab_dom_def, OF y_dom]
            destab_iter.psimps(1)[OF destab_iter_destab.domintros(1)]
True
          by auto
        have count_eq: "count_non_empty (fmdrop y infl) = count_non_empty
infl"
          using count_non_empty_eq_fmdrop[of infl y] True by auto
        then have IH: "destab_iter_dom ys (fmdrop y infl) (stabl -
{y})
            ∧ (destab_iter ys (fmdrop y infl) (stabl - {y}) = (infl',
stabl')
            ⟶ count_non_empty infl' ≤ count_non_empty (fmdrop y infl))"
          using Cons.IH[of "fmdrop y infl" "stabl - {y}"] Cons.prems
          by auto
        then show ?thesis
        proof (intro conjI, goal_cases)
          case 1
          then show dom_ys: ?case using destab_domintros(2)[OF y_dom
destab_y] IH by auto
          case 2
          then show ?case
            using IH count_eq destab_iter.psimps(2) destab_y dom_ys
            by auto
        qed
      next
        case False
        obtain u w where
          prod: "destab_iter (fmlookup_default infl [] y) (fmdrop y
infl) (stabl - {y}) = (u, w)"
          by fastforce

        have eq: "Suc (count_non_empty (fmdrop y infl)) = count_non_empty
infl"
          by (simp add: False count_non_empty_dec_fmdrop)
        then have dom1: "destab_dom y infl (stabl - {y})"
          using IH destab_domintros(3) by auto
        obtain i s where i_s_def: "(i, s) = destab y infl (stabl -
{y})"
          by (metis surj_pair)

        have "count_non_empty u ≤ count_non_empty (fmdrop y infl)"
```

```
            using IH eq prod
            by simp
          then have dom2: "destab_iter_dom ys i s" and dec: "destab_iter
ys u w = (infl', stabl')
              ⟶ count_non_empty infl' ≤ count_non_empty infl"
            using IH[of "count_non_empty u" u ys w infl' stabl'] prod
eq i_s_def destab.psimps dom1
            by auto

          show ?thesis
            using destab_iter.psimps(2) dec destab_iter_destab.domintros(2)
dom1 dom2 prod
            by (simp add: destab.psimps i_s_def)
        qed
      qed
    qed
  }
  then show ?thesis using destab_iter_destab.domintros(3) unfolding destab_iter_dom_def
    by (metis prod.collapse sumE)
qed
```

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 evalauated value of an unknown changed compared to the value tracked in σ. 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 σ = (
    let (xd, infl, stabl, σ) =
      if x ∈ c then
        (mlup σ x, infl, stabl, σ)
      else
        iterate x (insert x c) infl stabl σ
    in (xd, fminsert infl x y, stabl, σ))"
```

```
| "iterate x c infl stabl σ = (
    if x ∉ stabl then
      let (d_new, infl, stabl, σ) = eval x (T x) c infl (insert x stabl)
σ in
      if mlup σ x = d_new then
        (d_new, infl, stabl, σ)
      else
        let (infl, stabl) = destab x infl stabl in
        iterate x c infl stabl (σ(x ↦ d_new))
    else
      (mlup σ x, infl, stabl, σ))"
| "eval x t c infl stabl σ = (case t of
      Answer d ⇒ (d, infl, stabl, σ)
    | Query y g ⇒ (
        let (yd, infl, stabl, σ) = query x y c infl stabl σ in eval x
(g yd) c infl stabl σ))"
  by pat_completeness auto

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

definition query_dom where
  "query_dom x y c infl stabl σ = query_iterate_eval_dom (Inl (x, y, c,
infl, stabl, σ))"
declare query_dom_def [simp]
definition iterate_dom where
  "iterate_dom x c infl stabl σ = query_iterate_eval_dom (Inr (Inl (x,
c, infl, stabl, σ)))"
declare iterate_dom_def [simp]
definition eval_dom where
  "eval_dom x t c infl stabl σ = query_iterate_eval_dom (Inr (Inr (x,
t, c, infl, stabl, σ)))"
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 σ for x y c infl stabl σ
    ]
    query_iterate_eval.pinduct(2)[
      folded query_dom_def iterate_dom_def eval_dom_def,
      of x c infl stabl σ for x c infl stabl σ
    ]
    query_iterate_eval.pinduct(3)[
      folded query_dom_def iterate_dom_def eval_dom_def,
      of x t c infl stabl σ for x t c infl stabl σ
    ]

lemmas iterate_pinduct[consumes 1, case_names Iterate]
  = query_iterate_eval_pinduct(2)[where ?P="λx y c infl stabl σ. True"
    and ?R="λx t c infl stabl σ. 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 σ
    ⟹ y ∉ c ⟹ iterate_dom y (insert y c) infl stabl σ"
  and "iterate_dom x c infl stabl σ
    ⟹ x ∉ stabl ⟹ (eval_dom x (T x) c infl (insert x stabl) σ ∧
        ((xd_new, infl1, stabl1, σ') = eval x (T x) c infl (insert x stabl)
σ
          ⟶ mlup σ' x ≠ xd_new ⟶ (infl2, stabl2) = destab x infl1
stabl1 ⟶
          iterate_dom x c infl2 stabl2 (σ'(x ↦ xd_new))))"
  and "eval_dom x (Query y g) c infl stabl σ
    ⟹ (query_dom x y c infl stabl σ ∧
        ((yd, infl', stabl', σ') = query x y c infl stabl σ ⟶
          eval_dom x (g yd) c infl' stabl' σ'))"
proof (induction x y c infl stabl σ and x c infl stabl σ and x "Query
y g" c infl stabl σ
    arbitrary: and xd_new infl1 stabl1 infl2 stabl2 σ' and y g yd infl'
stabl' σ'
    rule: query_iterate_eval_pinduct)
  case (Query y x c infl stabl σ)
  then show ?case using query_iterate_eval.domintros(2) by fastforce
next
  case (Iterate x c infl stabl σ)
  then show ?case using query_iterate_eval.domintros(2,3) by simp
next
  case (Eval x c infl stabl σ)
  then show ?case using query_iterate_eval.domintros(1,3) by simp
```

qed

4.5 Case Rules

```
lemma iterate_continue_fixpoint_cases[consumes 3]:
  assumes "iterate_dom x c infl stabl σ"
    and "(xd, infl', stabl', σ') = iterate x c infl stabl σ"
    and "x ∈ c"
  obtains (Stable) "infl' = infl"
    and "stabl' = stabl"
    and "σ' = σ"
    and "mlup σ x = xd"
    and "x ∈ stabl"
  | (Fixpoint) "eval_dom x (T x) c infl (insert x stabl) σ"
    and "(xd, infl', stabl', σ') = eval x (T x) c infl (insert x stabl)
σ"
    and "mlup σ' x = xd"
    and "x ∉ stabl"
  | (Continue) stabl1 infl1 σ1 xd_new stabl2 infl2
  where "eval_dom x (T x) c infl (insert x stabl) σ"
    and "(xd_new, infl1, stabl1, σ1) = eval x (T x) c infl (insert x
stabl) σ"
    and "mlup σ1 x ≠ xd_new"
    and "(infl2, stabl2) = destab x infl1 stabl1"
    and "iterate_dom x c infl2 stabl2 (σ1(x ↦ xd_new))"
    and "(xd, infl', stabl', σ') = iterate x c infl2 stabl2 (σ1(x ↦
xd_new))"
    and "x ∉ stabl"
proof(cases "x ∈ stabl" rule: case_split[case_names Stable Unstable])
  case Stable
  then show ?thesis using that(1) assms by auto
next
  case Unstable
  then have sldom: "eval_dom x (T x) c infl (insert x stabl) σ"
    using assms(1) dom_backwards_pinduct(2)
    by simp
  then obtain xd_new infl1 stabl1 σ1
    where slapp: "eval x (T x) c infl (insert x stabl) σ = (xd_new, infl1,
stabl1, σ1)"
    by (cases "eval x (T x) c infl (insert x stabl) σ") auto
  show ?thesis
  proof (cases "mlup σ1 x = xd_new")
    case True
    then show ?thesis
      using Unstable sldom slapp assms that(2)
      by auto
  next
    case False
    then obtain infl2 stabl2 where destab: "destab x infl1 stabl1 = (infl2,
```

```
stabl2)"
      by (cases "destab x infl1 stabl1")
    then have dom: "iterate_dom x c infl2 stabl2 (σ1(x ↦ xd_new))"
      and "iterate x c infl stabl σ
        = iterate x c infl2 stabl2 (σ1(x ↦ xd_new))"
      and app: "iterate x c infl2 stabl2 (σ1(x ↦ xd_new))
        = (xd, infl', stabl', σ')"
      using Unstable False slapp assms(1-3) dom_backwards_pinduct(2)
      by auto
    then show ?thesis
      using sldom slapp Unstable False destab that(3)
      by simp
  qed
qed

lemma iterate_fmlookup:
  assumes "iterate_dom x c infl stabl σ"
    and "(xd, infl', stabl', σ') = iterate x c infl stabl σ"
    and "x ∈ c"
  shows "mlup σ' x = xd"
  using assms
proof(induction rule: iterate_pinduct)
  case (Iterate x c infl stabl σ)
  show ?case
    using Iterate.hyps Iterate.prems
  proof(cases rule: iterate_continue_fixpoint_cases)
    case (Continue σ1 xd_new)
    then show ?thesis
      using Iterate.prems(2) Iterate.IH
      by force
  qed (simp add: Iterate.prems(1))
qed

corollary query_fmlookup:
  assumes "query_dom y x c infl stabl σ"
    and "(xd, infl', stabl', σ') = query y x c infl stabl σ"
  shows "mlup σ' x = xd"
  using assms iterate_fmlookup dom_backwards_pinduct(1)[of y x c infl
stabl σ]
  by (auto split: prod.splits if_splits)

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

```
    and "mlup σ' x = xd"
    and "x ∉ c"
  | (Lookup) "mlup σ x = xd"
    and "infl' = fminsert infl x y"
    and "stabl' = stabl"
    and "σ' = σ"
    and "x ∈ c"
  using assms that dom_backwards_pinduct(1) query_fmlookup[OF assms(1,2)]
  by (cases "x ∈ c"; auto split: prod.splits)

lemma eval_query_answer_cases [consumes 2]:
  assumes "eval_dom x t c infl stabl σ"
    and "(xd, infl', stabl', σ') = eval x t c infl stabl σ"
  obtains (Query) y g yd infl1 stabl1 σ1
  where "t = Query y g"
    and "query_dom x y c infl stabl σ"
    and "(yd, infl1, stabl1, σ1) = query x y c infl stabl σ"
    and "eval_dom x (g yd) c infl1 stabl1 σ1"
    and "(xd, infl', stabl', σ') = eval x (g yd) c infl1 stabl1 σ1"
    and "mlup σ1 y = yd"
  | (Answer) "t = Answer xd"
    and "infl' = infl"
    and "stabl' = stabl"
    and "σ' = σ"
  using assms dom_backwards_pinduct(3) that query_fmlookup
  by (cases t; auto split: prod.splits)
```

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 ⟹ y ∈ set ys ⟹ y ∈ influenced_by
infl x"
| step: "y ∈ influenced_by infl x ⟹ fmlookup infl y = Some zs ⟹ z
∈ set zs
    ⟹ z ∈ influenced_by infl x"
inductive_set influenced_by_cutoff for infl x c where
  base: "x ∉ c ⟹ fmlookup infl x = Some ys ⟹ y ∈ set ys ⟹ y ∈
influenced_by_cutoff infl x c"
| step: "y ∈ influenced_by_cutoff infl x c ⟹ y ∉ c ⟹ fmlookup infl
y = Some zs ⟹ z ∈ set zs
    ⟹ z ∈ influenced_by_cutoff infl x c"

lemma influenced_by_aux:
  shows "influenced_by infl x = (⋃y ∈ slookup infl x. insert y (influenced_by
(fmdrop x infl) y))"
unfolding fmlookup_default_def
```

```
proof(intro equalityI subsetI, goal_cases)
  case (1 u)
  then show ?case
  proof(induction rule: influenced_by.induct)
    case (step y zs z)
    then show ?case
    proof(cases "y ∈ slookup infl x")
      case True
      then show ?thesis
        using step.hyps(2,3) influenced_by.base[of "fmdrop x infl" y]
        by (cases rule: set_fmlookup_default_cases, cases "x = y") auto
    next
      case False
      then show ?thesis
        using step.IH step.hyps(2,3) influenced_by.step[of y "fmdrop x
infl"]
        by (cases rule: notin_fmlookup_default_cases, cases "x = y") auto
    qed
  qed auto
next
  case (2 z)
  then show ?case
  proof(cases "fmlookup infl x")
    case (Some xs)
    then obtain y where z_mem: "z ∈ insert y (influenced_by (fmdrop
x infl) y)"
      and step: "y ∈ set (case fmlookup infl x of None ⇒ [] | Some v
⇒ v)" using 2 by blast
    then show ?thesis using Some influenced_by.base
    proof(cases "z = y")
      case False
      then have "z ∈ influenced_by (fmdrop x infl) y" using z_mem by
auto
      then show ?thesis
      proof(induction rule: influenced_by.induct)
        case (base ys' y')
        then show ?case
          using Some step influenced_by.base[of infl] influenced_by.step[of
y]
          by (auto split: if_splits)
      next
        case (step y' zs z)
        then show ?case using influenced_by.step
          by (auto split: if_splits)
      qed
    qed simp
  qed simp
qed
```

```
lemma lookup_in_influenced:
  shows "slookup infl x ⊆ influenced_by infl x"
proof(intro subsetI, goal_cases)
  case (1 y)
  then show ?case using influenced_by.base[of infl x]
  by (cases rule: set_fmlookup_default_cases) simp
qed

lemma influenced_unknowns_fmdrop_set:
  shows "influenced_by (fmdrop_set C infl) x = influenced_by_cutoff infl
x C"
proof (intro equalityI subsetI, goal_cases)
  case (1 u) then show ?case by (induction rule: influenced_by.induct;
        simp add: influenced_by_cutoff.base influenced_by_cutoff.step
split: if_splits)
next
  case (2 u) then show ?case by (induction rule: influenced_by_cutoff.induct;
        simp add: influenced_by.base influenced_by.step)
qed

lemma influenced_by_transitive:
  assumes "y ∈ influenced_by infl x"
    and "z ∈ influenced_by infl y"
  shows "z ∈ influenced_by infl x"
  using assms
proof (induction rule: influenced_by.induct)
  case (base ys y)
  show ?case using base(3,1,2) influenced_by.step[of _ infl x]
  proof (induction rule: influenced_by.induct)
    case (base us u)
    then show ?case using influenced_by.base[of infl x ys y] by simp
  qed simp
next
  case (step u vs v)
  have "z ∈ influenced_by infl u" using step(5,1-4)
  proof (induction rule: influenced_by.induct)
    case (base ys y)
    then show ?case using influenced_by.base[of infl] influenced_by.step[of
v infl] by auto
  next
    case (step y zs z)
    then show ?case using influenced_by.step[of _ infl] by auto
  qed
  then show ?case using step by auto
qed

lemma influenced_cutoff_subset:
  "influenced_by_cutoff infl x C ⊆ influenced_by infl x"
proof (intro subsetI, goal_cases)
```

```
  case (1 y)
  then show ?case
    by (induction rule: influenced_by_cutoff.induct)
      (auto simp add: influenced_by.base influenced_by.step)
qed

lemma influenced_cutoff_subset_2:
  shows "influenced_by infl x - (⋃y ∈ C. influenced_by infl y) ⊆ influenced_by_cutoff
infl x C"
proof (intro equalityI subsetI, elim DiffE, goal_cases)
  case (1 y)
  then show ?case
  proof (induction rule: influenced_by.induct)
    case (base ys z)
    then show ?case using 1 influenced_by_cutoff.base by fastforce
  next
    case (step y zs z)
    then show ?case
      using influenced_by.base[OF step(2,3)] influenced_by.step[of y infl]
        influenced_by_cutoff.step[of y infl x C zs z]
      by blast
  qed
qed

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 -
  have "u ∈ influenced_by infl y"
    if "u ≠ y" and "u ∉ influenced_by_cutoff infl x (insert y (influenced_by
infl y))"
      and "u ∈ influenced_by infl x" for u
    using that influenced_cutoff_subset_2[of infl x "insert y (influenced_by
infl y)"]
      influenced_by_transitive[of _ infl y] by auto
  moreover have "u ∈ influenced_by infl y"
    if "u ≠ y" and "u ∉ influenced_by infl x"
      and "u ∈ influenced_by_cutoff infl x (insert y (influenced_by infl
y))" for u
    using that(3)
  proof (induction rule: influenced_by_cutoff.induct)
    case (base ys y)
    then show ?case using that(2,3) influenced_cutoff_subset[of infl
x] by auto
  qed simp
  ultimately show ?thesis by auto
qed
```

```
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 (induction xs infl stabl and x infl stabl
    arbitrary: infl' stabl' and infl' stabl' rule: destab_iter_destab.induct)
  case (1 infl stabl)
  then show ?case by simp
next
  case (2 y ys infl stabl)
  then obtain infl'' stabl'' where destab_y: "(infl'', stabl'') = destab
y infl (stabl - {y})"
    and destab_ys: "(infl', stabl') = destab_iter ys infl'' stabl''"
    by (cases "destab y infl (stabl - {y})"; auto)
  note IH1 = "2.IH"(1)[OF destab_y]
  note IH2 = "2.IH"(2)[OF destab_y _ destab_ys, simplified]

  define A where "A x ≡ insert x (influenced_by infl x)" for x
  define B where "B x ≡ insert x (influenced_by_cutoff infl x (insert
y (influenced_by infl y)))"
    for x
  have A_union_B_simp: "A y ∪ (⋃x∈set ys. B x) = (⋃x∈set (y#ys). A
x)"
    using union_influenced_to_cutoff[of y] A_def B_def
    by fastforce

  show ?case
  proof(intro conjI, goal_cases)
    case 1
      have "infl' = fmdrop_set (⋃x∈set ys. B x) (fmdrop_set (A y) infl)"
        using IH1 IH2 influenced_unknowns_fmdrop_set[of "A y"] A_def B_def
by auto
      also have "... = fmdrop_set (A y ∪ (⋃x∈set ys. B x)) infl"
        by (simp add: Un_commute)
      also have "... = fmdrop_set (⋃x∈set (y # ys). A x) infl"
        using A_union_B_simp by auto
      finally show ?case
        using A_def B_def by auto
  next
    case 2
    have "stabl' = stabl - (A y ∪ (⋃x∈set ys. B x))"
      using IH1 IH2 A_def B_def influenced_unknowns_fmdrop_set[of "A y"]
      by auto
```

```
    also have "... = stabl - (⋃x∈set (y#ys). A x)"
      using A_union_B_simp
      by auto
    finally show ?case
      using A_def B_def by auto
  qed
next
  case (3 y infl stabl)
  then have
    destab_y: "destab_iter (fmlookup_default infl [] y) (fmdrop y infl)
stabl = (infl', stabl')"
    by simp
  note IH = "3.IH"[OF destab_y[symmetric]]
  then show ?case using influenced_by_aux[of infl] by simp
qed
```

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 (fmdom infl) ⊆ stabl
    ∧ (∀y∈stabl - c. ∀x ∈ dep σ y. y ∈ 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)"
  using assms
proof -
  have "c - {x} ⊆ stabl ≡ c ⊆ insert x stabl"
    using assms(1)
    by (simp add: subset_insert_iff)
  moreover have "stabl - (c - {x}) ⊇ insert x stabl - c"
    using assms(1)
    by auto
  ultimately show ?thesis
    using assms(2)
    unfolding invariant_def
    by (meson subset_iff subset_insertI2)
qed
```

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 (rule ccontr, goal_cases)
  case 1
  then have "x ∈ fset (fmdom infl)"
    unfolding fmlookup_default_def by force
  then show ?case using assms by blast
qed

lemma dep_closed_implies_reach_cap_tree_closed:
  assumes "x ∈ stabl'"
    and "∀ξ∈stabl' - (c - {x}). dep σ' ξ ⊆ stabl'"
  shows "reach_cap σ' (c - {x}) x ⊆ stabl'"
proof (intro subsetI, goal_cases)
  case (1 y)
  then show ?case using assms
  proof(cases "x = y")
    case False
    then have "y ∈ reach_cap_tree σ' (c - {x}) (T x)"
      using 1 reach_cap_tree_simp2[of x "c - {x}" σ'] by auto
    then show ?thesis using assms
    proof(induction)
      case (base y)
      then show ?case using base.hyps dep_def by auto
    next
      case (step y z)
      then show ?case by (metis (no_types, lifting) Diff_iff insert_subset
mk_disjoint_insert)
    qed
  qed simp
qed

lemma dep_subset_stable:
  assumes "fset (fmdom infl) ⊆ stabl"
    and "(∀y∈stabl - c. ∀x ∈ dep σ y. y ∈ slookup infl x)"
  shows "(∀ξ∈stabl - c. dep σ ξ ⊆ stabl)"
  using assms stabl_infl_empty[of _ stabl infl]
  by (metis DiffD2 Diff_empty subsetI)

lemma new_lookup_to_infl_not_stabl:
  assumes "∀ξ. (slookup infl1 ξ - slookup infl ξ) ∩ stabl = {}"
    and "x ∉ stabl"
    and "fset (fmdom infl) ⊆ stabl"
  shows "influenced_by infl1 x ∩ stabl = {}"
proof -
  have "u ∉ stabl" if "u ∈ influenced_by infl1 x" for u
    using that
  proof (induction rule: influenced_by.induct)
    case (base ys y)
```

```
    have "slookup infl x = {}" using stabl_infl_empty[OF assms(2,3)] by
auto
    then have "y ∈ slookup infl1 x - slookup infl x"
      using base.hyps(1,2) by auto
    then show ?case using base.hyps(1) assms(1,3) by force
  next
    case (step y zs z)
    have "slookup infl y = {}"
      by (meson assms(3) stabl_infl_empty step.IH)
    then have "z ∈ slookup infl1 y - slookup infl y"
      by (simp add: step.hyps(2,3))
    then show ?case using assms(1) stabl_infl_empty[OF _ assms(3)] by
fastforce
  qed
  then show ?thesis by auto
qed

lemma infl_upd_diff:
  assumes "∀ξ. (slookup infl' ξ - slookup infl ξ) ∩ stabl = {}"
  shows "∀ξ. (slookup (fminsert infl' x y) ξ - slookup infl ξ) ∩ (stabl
- {y}) = {}"
proof(intro allI, goal_cases)
  case (1 ξ)
  show ?case using assms unfolding fminsert_def fmlookup_default_def
  by (cases "x = ξ") auto
qed

lemma infl_diff_eval_step:
  assumes "stabl ⊆ stabl1"
    and "∀ξ. (slookup infl' ξ - slookup infl1 ξ) ∩ (stabl1 - {x}) = {}"
    and "∀ξ. (slookup infl1 ξ - slookup infl ξ) ∩ (stabl - {x}) = {}"
  shows "∀ξ. (slookup infl' ξ - slookup infl ξ) ∩ (stabl - {x}) = {}"
proof(intro allI, goal_cases)
  case (1 ξ)
  have "((slookup infl' ξ - slookup infl1 ξ)
         ∪ (slookup infl1 ξ - slookup infl ξ)) ∩ (stabl - {x}) = {}"
    using assms by auto
  then show ?case by blast
qed
```

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 destab_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 (intro ballI, goal_cases)
```

```
  case (1 y)
  show ?case
  proof (rule ccontr, goal_cases)
    case 1
    then have "y ∈ slookup infl1 x"
      using assms ‹y ∈ stabl2 - (c - {x})› by blast
    then have "y ∈ influenced_by infl1 x"
      using lookup_in_influenced by force
    moreover have "y ∉ influenced_by infl1 x"
      using assms(1) ‹y ∈ stabl2 - (c - {x})› by fastforce
    ultimately show ?case by auto
  qed
qed

lemma destab_preserves_c_subset_stabl:
  assumes "c ⊆ stabl"
    and "stabl ⊆ stabl'"
  shows "c ⊆ stabl'"
  using assms by auto

lemma destab_preserves_infl_dom_stabl:
  assumes "(infl', stabl') = destab x infl stabl"
    and "fset (fmdom infl) ⊆ stabl"
  shows "fset (fmdom infl') ⊆ stabl'"
proof -
  have "infl' = fmdrop_set (insert x (influenced_by infl x)) infl"
    and A: "stabl' = stabl - influenced_by infl x"
    using assms(1) destab_infl_stabl_relation by metis+
  then show ?thesis
    using assms(2)
    by (metis Diff_mono fmdom'_alt_def fmdom'_drop_set subset_insertI)
qed

lemma destab_and_upd_preserves_dep_closed_in_infl:
  assumes "(infl2, stabl2) = destab 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 (intro ballI, goal_cases)
  case (1 z y)
  have infl2_def: "infl2 = fmdrop_set (insert x (influenced_by infl1 x))
infl1"
    and stabl2_def: "stabl2 = stabl1 - influenced_by infl1 x"
    using assms(1) destab_infl_stabl_relation by metis+

  have "y ∈ dep σ1 z"
  proof (goal_cases)
    case 1
    have "∀y∈stabl2 - (c - {x}). x ∉ dep σ1 y"
```

```
      using assms(2) stabl2_def destab_x_no_dep by auto
    then have "x ∉ dep σ1 z"
        using ‹z ∈ stabl2 - (c - {x})› by blast
    then have "dep (σ1(x ↦ xd')) z = dep σ1 z"
      using dep_eq[of σ1 z "σ1(x ↦ xd')"] mlup_eq_mupd_set[of x "dep
σ1 z" σ1 σ1 xd']
      by metis
    then show ?case using ‹y ∈ dep (σ1(x ↦ xd')) z› by auto
  qed
  then have z_in_infl1_y: "z ∈ slookup infl1 y"
    using 1(1) stabl2_def assms(2) by fastforce

  have "z ∈ influenced_by infl1 y"
    using lookup_in_influenced[of infl1 y] z_in_infl1_y
    by auto
  then have "y ∉ influenced_by infl1 x" and "y ≠ x"
    using stabl2_def 1(1) influenced_by_transitive[of y _ x z] by auto
  then show ?case
    using z_in_infl1_y fmlookup_drop_set infl2_def
    unfolding fmlookup_default_def
    by fastforce
qed

lemma destab_upd_preserves_part_sol:
  assumes "(infl2, stabl2) = destab 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 (intro ballI, goal_cases)
  case (1 y)
  have stabl2_def: "stabl2 = stabl1 - influenced_by infl1 x"
    using assms(1) destab_infl_stabl_relation by auto
  have x_no_dep: "∀y ∈ stabl2 - (c - {x}). x ∉ dep σ1 y"
    using destab_x_no_dep[OF stabl2_def assms(3)] by simp
  have eq_y_upd: "eq y (σ1(x ↦ xd')) = eq y σ1"
    using 1 eq_mupd_no_dep[of x σ1 y] x_no_dep
    by auto
  show ?case
  proof (cases "y = x")
    case True
    then show ?thesis using assms(4) eq_y_upd unfolding mlup_def by
(simp add: fun_upd_same)
  next
    case False
    then have "y ∈ stabl1 - c"
      using 1 stabl2_def by force
    then have "eq y σ1 = mlup σ1 y"
      using assms(2) by blast
```

then show *?thesis* using *False eq_y_upd* unfolding *mlup_def* by *(simp add: fun_upd_other)*
qed
qed

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' ≡
    stabl ⊆ stabl' ∧
    (∀u ∈ stabl. slookup infl u ⊆ slookup infl' u) ∧
    (∀u. (slookup infl' u - slookup infl u) ∩ (stabl - {x}) = {})"
```

4.10.1 TD_plain → TD

```
lemma TD_plain_TD_equivalence_ind:
  shows "TD_plain.query_dom T x y c σ
    ⟹ TD_plain.query T x y c σ = (yd, σ')
    ⟹ invariant c σ infl stabl
    ⟹ query_dom x y c infl stabl σ
        ∧ (∃infl' stabl'. query x y c infl stabl σ = (yd, infl', stabl',
σ')
        ∧ invariant c σ' infl' stabl'
        ∧ x ∈ slookup infl' y
        ∧ update_rel x infl stabl infl' stabl')"
    and "TD_plain.iterate_dom T x c σ
    ⟹ TD_plain.iterate T x c σ = (xd, σ')
    ⟹ x ∈ c
    ⟹ invariant (c - {x}) σ infl stabl
    ⟹ iterate_dom x c infl stabl σ
        ∧ (∃infl' stabl'. iterate x c infl stabl σ = (xd, infl', stabl',
σ')
        ∧ invariant (c - {x}) σ' infl' stabl'
        ∧ x ∈ stabl'
```

```
        ∧ update_rel x infl stabl infl' stabl')"
    and "TD_plain.eval_dom T x t c σ
    ⟹ TD_plain.eval T x t c σ = (xd, σ')
    ⟹ invariant c σ infl stabl
    ⟹ x ∈ stabl
    ⟹ eval_dom x t c infl stabl σ
        ∧ (∃infl' stabl'. eval x t c infl stabl σ = (xd, infl', stabl',
σ')
        ∧ invariant c σ' infl' stabl'
        ∧ traverse_rhs t σ' = xd
        ∧ (∀y∈dep_aux σ' t. x ∈ slookup infl' y)
        ∧ update_rel x infl stabl infl' stabl')"
proof(induction x y c σ and x c σ and x t c σ
    arbitrary: yd σ' infl stabl and xd σ' infl stabl and xd σ' infl
stabl
    rule: TD_plain.query_iterate_eval_pinduct[of T, consumes 1, case_names
Query Iterate Eval])
  case (Query x y c σ)
  show ?case using Query.IH(1) Query.prems(1)
  proof (cases rule:
      TD_plain.query_iterate_lookup_cases[of T, consumes 2, case_names
Iterate Lookup])
    case Iterate
    moreover obtain infl' stabl' where IH: "iterate_dom y (insert y
c) infl stabl σ ∧
        iterate y (insert y c) infl stabl σ = (yd, infl', stabl', σ')
∧
        invariant c σ' infl' stabl' ∧
        y ∈ stabl' ∧
        update_rel y infl stabl infl' stabl'"
      using Query.IH(2)[simplified, OF Iterate(4,2) Query.prems(2), folded
dom_defs] by auto
    ultimately show ?thesis
    proof (intro conjI, goal_cases)
      case 1 then show dom: ?case using query_iterate_eval.domintros(1)[folded
dom_defs] by auto
      case 2 then show ?case
      proof (intro exI[of _ "fminsert infl' y x"] exI[of _ stabl'], intro
conjI, goal_cases)
        case 1 then show ?case using dom by simp
      next
        case 2 then show ?case
          unfolding invariant_def by (auto simp add: fminsert_def fmlookup_default_def)
      next
        case 6 then have "∀ξ. (slookup infl' ξ - slookup infl ξ) ∩ stabl
= {}"
          by (cases "y ∈ stabl"; auto)
        then show ?case
          using infl_upd_diff[of infl' infl stabl y x] by auto
```

```
      qed (auto simp add: fminsert_def fmlookup_default_def)
    qed
  next
    case Lookup
    then show ?thesis using Query.prems(1,2)
    proof (intro conjI, goal_cases)
      case 1 then show dom: ?case using query_iterate_eval.domintros(1)[of
y c] by auto
      case 2 then show ?case
      proof (intro exI[of _ "fminsert infl y x"] exI[of _ stabl], intro
conjI, goal_cases)
        case 1 then show ?case using dom by simp
      next
        case 2 then show ?case
          unfolding invariant_def by (auto simp add: fminsert_def fmlookup_default_def)
      next
        case 6 then show ?case
          using infl_upd_diff[of infl infl stabl y] by auto
      qed (auto simp add: fminsert_def fmlookup_default_def)
    qed
  qed
next
  case (Iterate x c σ)
  have inv: "invariant c σ infl (insert x stabl)"
    using Iterate.prems(2,3) invariant_simp_c_stabl by auto
  have dep_in_stabl: "∀ξ∈stabl - (c - {x}). dep σ ξ ⊆ stabl"
    using Iterate.prems(3) dep_subset_stable[of infl stabl] unfolding
invariant_def by auto
  show ?case
  proof(cases "x ∈ stabl" rule: case_split[case_names Stable Unstable])
    case Stable
    then show ?thesis
    proof(intro conjI, goal_cases)
      case 1 then show dom: ?case using query_iterate_eval.domintros(2)[of
x stabl] by simp
      case 2 moreover have "σ = σ'"
        using Iterate.prems(3) TD_plain.already_solution(2)[OF Iterate.IH(1)
Iterate.prems(1,2) 2]
          dep_in_stabl unfolding TD_plain.invariant_def invariant_def
by fastforce
      ultimately show ?case
      proof (intro exI[of _ infl] exI[of _ stabl] conjI, goal_cases)
        case 1
          then show ?case using dom TD_plain.iterate_fmlookup[OF Iterate.IH(1)
Iterate.prems(1,2)]
            by auto
        next
          case 2 then show ?case using Iterate.prems(3) by auto
      qed auto
```

```
    qed
  next
    case Unstable
    show ?thesis using Iterate.IH(1) Iterate.prems(1,2)
    proof(cases rule:
        TD_plain.iterate_continue_fixpoint_cases[of T, consumes 3, case_names
Fixpoint Continue])
      case Fixpoint
      moreover obtain infl' stabl' where IH: "eval_dom x (T x) c infl
(insert x stabl) σ ∧
        (xd, infl', stabl', σ') = eval x (T x) c infl (insert x stabl)
σ ∧
        invariant c σ' infl' stabl' ∧
        eq x σ' = xd ∧
        (∀y∈dep σ' x. x ∈ slookup infl' y) ∧
        update_rel x infl (insert x stabl) infl' stabl'"
        using Iterate.IH(2)[OF Fixpoint(2) inv, folded dep_def] by auto
      ultimately show ?thesis using Unstable
      proof(intro conjI, goal_cases)
      case 1 then show dom: ?case using query_iterate_eval.domintros(2)[of
x stabl c infl σ]
        by (cases "eval x (T x) c infl (insert x stabl) σ"; auto)
      case 2 then show ?case
        proof (intro exI[of _ infl'] exI[of _ stabl'] conjI, goal_cases)
          case 1 then show ?case using dom by (auto split: prod.splits)
        next
          case 2 then show ?case unfolding invariant_def by auto
        next
          case 3 then show ?case using Iterate.prems(2) invariant_def
by fastforce
        qed auto
      qed
    next
      case (Continue σ1 xd')
      obtain infl1 stabl1 where IH: "eval_dom x (T x) c infl (insert
x stabl) σ ∧
        (xd', infl1, stabl1, σ1) = eval x (T x) c infl (insert x stabl)
σ ∧
        invariant c σ1 infl1 stabl1 ∧
        eq x σ1 = xd' ∧
        (∀y∈dep σ1 x. x ∈ slookup infl1 y) ∧
        update_rel x infl (insert x stabl) infl1 stabl1"
      using Iterate.IH(2)[OF Continue(2) inv, folded dep_def] by auto
      obtain infl2 stabl2 where destab: "(infl2, stabl2) = destab x infl1
stabl1"
        by (cases "destab x infl1 stabl1"; auto)
      then have infl2_def: "infl2 = fmdrop_set (insert x (influenced_by
infl1 x)) infl1"
        and stabl2_def: "stabl2 = stabl1 - influenced_by infl1 x"
```

```
        using destab_infl_stabl_relation[of infl2 stabl2 x infl1 stabl1]
by auto
      define σ2 where [simp]: "σ2 = σ1(x ↦ xd')"
      have infl_diff: "∀ξ. (slookup infl1 ξ - slookup infl ξ) ∩ stabl
= {}"
        using Unstable Iterate.prems(3) IH
        unfolding invariant_def by auto
      have infl_closed: "∀x∈stabl1 - (c - {x}). ∀y∈dep σ1 x. x ∈ slookup
infl1 y"
        using IH unfolding dep_def invariant_def by auto
      have stabl_inc: "stabl ⊆ stabl2"
        using IH Iterate.prems(3) new_lookup_to_infl_not_stabl[OF infl_diff
Unstable]
        unfolding invariant_def stabl2_def by auto

      have inv2: "invariant (c - {x}) σ2 infl2 stabl2"
        using IH unfolding invariant_def
      proof(elim conjE, intro conjI, goal_cases)
        case 1
        show ?case using destab_preserves_c_subset_stabl stabl_inc Iterate.prems(3)
          unfolding invariant_def by auto
      next
        case 2 then show ?case using destab_upd_preserves_part_sol[OF
destab _ infl_closed] by auto
      next
        case 3 then show ?case using destab_preserves_infl_dom_stabl[OF
destab] by auto
      next
        case 4 show ?case
        proof(intro ballI, goal_cases)
          case (1 y z)
          have x_no_dep: "x ∉ dep σ1 y" if "y ∈ stabl2 - (c - {x})" for
y
            using that destab_infl_stabl_relation[OF destab] infl_closed
destab_x_no_dep by blast
          have "dep σ1 y = dep σ2 y" using x_no_dep[OF 1(1)] dep_eq[of
σ1 _ σ2]
            unfolding mlup_def by (simp add: fun_upd_apply)
          then show ?case using 1 destab_and_upd_preserves_dep_closed_in_infl[OF
destab infl_closed]
            by auto
        qed
      qed
      obtain infl' stabl' where ih: "iterate_dom x c infl2 stabl2 (σ1(x
↦ xd')) ∧
          iterate x c infl2 stabl2 (σ1(x ↦ xd')) = (xd, infl', stabl',
σ') ∧
          invariant (c - {x}) σ' infl' stabl' ∧
          x ∈ stabl' ∧
```

```
          update_rel x infl2 stabl2 infl' stabl'"
        using Iterate.IH(3)[OF Continue(2)[symmetric] _ Continue(3)[symmetric]
Continue(5)
          Iterate.prems(2) inv2[unfolded σ2_def], simplified, folded dom_defs]
          Continue(2,3,5) Iterate.IH(3) Iterate.prems(2) σ2_def inv2
        by fastforce

      show ?thesis using IH ih destab Unstable
      proof(elim conjE, intro conjI, goal_cases)
        case 1 show dom: ?case using query_iterate_eval.domintros(2)[of
x stabl c infl σ]
          using 1(1-2,3-5)
          by (cases "eval x (T x) c infl (insert x stabl) σ"; cases "destab
x infl1 stabl1"; auto)
        case 2 then show ?case
        proof (intro exI[of _ infl'] exI[of _ stabl'] conjI, goal_cases)
          case 1 show ?case using 1(1,5,6) Continue(3) dom Unstable by
(auto split: prod.splits)
        next
          case 4
          show ?case
            using "4"(12) stabl_inc by auto
        next
          case 5  show ?case
          proof(intro ballI subsetI, goal_cases)
            case (1 ξ u)
            have "ξ ∉ insert x (influenced_by infl1 x)"
              using 1(1) stabl2_def stabl_inc Unstable by blast
            then show ?case using stabl_inc infl2_def 1 5(14,16)
                fmlookup_default_drop_set[of "insert x (influenced_by
infl1 x)" infl1 ξ]
              by fastforce
          qed
        next
          case 6 show ?case
          proof(intro allI, goal_cases)
            case (1 ξ)
            have "slookup infl2 ξ ⊆ slookup infl1 ξ" using infl2_def
              unfolding fmlookup_default_def by auto
            moreover have "(slookup infl' ξ - slookup infl2 ξ) ∩ (stabl
- {x}) = {}"
              using stabl_inc ih
              by blast
            moreover have "(slookup infl1 ξ - slookup infl ξ) ∩ (stabl
- {x}) = {}"
              using 6(7)[unfolded invariant_def] infl_diff stabl_infl_empty[of
ξ stabl1 infl1]
              by (cases "ξ ∈ stabl1"; auto)
            ultimately show ?case unfolding stabl2_def by auto
```

```
          qed
        qed auto
      qed
    qed
  qed
next
  case (Eval x t c σ)
  show ?case using Eval.IH(1) Eval.prems(1)
  proof(cases rule: TD_plain.eval_query_answer_cases[of T, consumes 2,
case_names Query Answer])
    case (Query y g yd σ1)
    obtain infl1 stabl1 where IH: "query_dom x y c infl stabl σ ∧
        (yd, infl1, stabl1, σ1) = query x y c infl stabl σ ∧
        invariant c σ1 infl1 stabl1 ∧
        x ∈ slookup infl1 y ∧
        update_rel x infl stabl infl1 stabl1"
      using Eval.IH(2)[OF Query(1,3) Eval.prems(2)] by metis
    then obtain infl' stabl' where ih: "eval_dom x (g yd) c infl1 stabl1
σ1 ∧
      (xd, infl', stabl', σ') = eval x (g yd) c infl1 stabl1 σ1 ∧
      invariant c σ' infl' stabl' ∧
      traverse_rhs (g yd) σ' = xd ∧
      (∀y∈dep_aux σ' (g yd). x ∈ slookup infl' y) ∧
      update_rel x infl1 stabl1 infl' stabl'"
      using Eval.prems(3) Eval.IH(3)[OF Query(1) Query(3)[symmetric] _
Query(5), of infl1 stabl1]
      by fastforce
    have td1_inv: "TD_plain.invariant T stabl c σ"
      using Eval.prems(2) dep_subset_stable unfolding TD_plain.invariant_def
invariant_def by blast
    have td1_inv2: "TD_plain.invariant T (stabl ∪ reach_cap σ1 c y) c
σ1"
      using TD_plain.partial_correctness_ind(1)[OF Query(2,3) td1_inv]
by auto
    have mlup: "mlup σ' y = yd"
      using TD_plain.partial_correctness_ind(3)[OF Query(4,5) td1_inv2]
Query(6) by auto

    show ?thesis using IH ih
    proof (elim conjE, intro conjI, goal_cases)
      case 1
      show dom: ?case
        using 1(1-3) Query(1) query_iterate_eval.domintros(3)[of t x c
infl stabl σ]
        by (cases "query x y c infl stabl σ"; fastforce)
      case 2
      then show ?case
      proof (intro exI[of _ infl'] exI[of _ stabl'] conjI, goal_cases)
        case 1 show ?case using 1(3,4) dom Query(1) by (auto split:prod.splits)
```

```
      next
        case 3 then show ?case using Query(1) mlup by auto
      next
        case 4 show ?case using 4(5,7,10,14) Query(1) mlup stabl_infl_empty[of
y stabl1 infl1]
          unfolding invariant_def by auto
      next
        case 6 then show ?case by blast
      next
        case 7 show ?case
          using 7(9,12,15) infl_diff_eval_step[of stabl stabl1 infl' infl1
x infl]
          by auto
      qed auto
    qed
  next
    case Answer
    then show ?thesis using Eval.prems(2)
    proof (intro conjI, goal_cases)
      case 1 then show dom: ?case using query_iterate_eval.domintros(3)[of
t] by auto
      case 2 then show ?case
      proof (intro exI[of _ infl] exI[of _ stabl] conjI, goal_cases)
        case 1 then show ?case using dom by auto
      qed auto
    qed
  qed
qed

corollary TD_plain_TD_equivalence:
  assumes "TD_plain.solve_dom T x"
    and "TD_plain.solve T x = σ"
  shows "∃stabl. solve_dom x ∧ solve x = (stabl, σ)"
proof -
  obtain xd where iter: "TD_plain.iterate T x {x} Map.empty = (xd, σ)"
    using assms(2) unfolding TD_plain.solve_def by (auto split: prod.splits)
  have inv: "invariant ({x} - {x}) Map.empty fmempty {}" unfolding invariant_def
by fastforce
  obtain infl stabl where "iterate_dom x {x} fmempty {} (λx. None)"
      and "iterate x {x} fmempty {} (λx. None) = (xd, infl, stabl, σ)"
    using TD_plain_TD_equivalence_ind(2)[OF assms(1)[unfolded TD_plain.solve_dom_def]
iter _ inv]
    by auto
  then show ?thesis unfolding solve_dom_def solve_def by (auto split:
prod.splits)
qed
```

4.10.2 TD → 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 σ
    ⟹ (yd, infl', stabl', σ') = query x y c infl stabl σ
    ⟹ invariant c σ infl stabl
    ⟹ finite stabl
    ⟹ invariant c σ' infl' stabl'
      ∧ TD_plain.query_dom T x y c σ
      ∧ (yd, σ') = TD_plain.query T x y c σ
      ∧ finite stabl'
      ∧ x ∈ slookup infl' y
      ∧ update_rel x infl stabl infl' stabl'"
    and "iterate_dom x c infl stabl σ
    ⟹ (xd, infl', stabl', σ') = iterate x c infl stabl σ
    ⟹ x ∈ c
    ⟹ invariant (c - {x}) σ infl stabl
    ⟹ finite stabl
    ⟹ invariant (c - {x}) σ' infl' stabl'
      ∧ TD_plain.iterate_dom T x c σ
      ∧ (xd, σ') = TD_plain.iterate T x c σ
      ∧ finite stabl'
      ∧ x ∈ stabl'
      ∧ update_rel x infl stabl infl' stabl'"
    and "eval_dom x t c infl stabl σ
    ⟹ (xd, infl', stabl', σ') = eval x t c infl stabl σ
    ⟹ invariant c σ infl stabl
    ⟹ x ∈ stabl
    ⟹ finite stabl
    ⟹ invariant c σ' infl' stabl'
      ∧ TD_plain.eval_dom T x t c σ
      ∧ (xd, σ') = TD_plain.eval T x t c σ
      ∧ finite stabl'
      ∧ traverse_rhs t σ' = xd
      ∧ (∀y∈dep_aux σ' t. x ∈ slookup infl' y)
      ∧ update_rel x infl stabl infl' stabl'"
proof(induction x y c infl stabl σ and y c infl stabl σ and x t c infl
stabl σ
    arbitrary: yd infl' stabl' σ' and xd infl' stabl' σ' and xd infl'
stabl' σ'
    rule: query_iterate_eval_pinduct)
  case (Query y x c infl stabl σ)
  show ?case using Query.IH(1) Query.prems(1)
  proof(cases rule: query_iterate_lookup_cases)
    case (Iterate infl1)
```

```
    moreover
    note IH = Query.IH(2)[simplified, folded TD_plain_dom_defs, OF Iterate(5,2)
Query.prems(2,3)]
    ultimately show ?thesis
    proof(intro conjI, goal_cases)
      case 1 then show ?case unfolding invariant_def
        by (auto simp add: fminsert_def fmlookup_default_def)
    next
      case 2 then show dom: ?case using TD_plain.query_iterate_eval.domintros(1)[of
x c] by auto
      case 3 then show ?case using dom by auto
    next case 8 then have "∀ξ. (slookup infl1 ξ - slookup infl ξ) ∩
stabl = {}"
        using Query.prems(3)[unfolded invariant_def]
        by (cases "x ∈ stabl"; simp)
      then show ?case
        using 8 infl_upd_diff[of infl1 infl stabl x] Query.prems(2) by
auto
    qed (auto simp add: fminsert_def fmlookup_default_def)
  next
    case Lookup
    then show ?thesis using Query.prems(2,3)
    proof(intro conjI, goal_cases)
      case 1 then show ?case unfolding invariant_def
        by (auto simp add: fminsert_def fmlookup_default_def)
    next
      case 2 then show dom: ?case using TD_plain.query_iterate_eval.domintros(1)[of
x c] by auto
      case 3 then show ?case using dom by auto
    next case 8 then show ?case
      using infl_upd_diff[of infl infl stabl x] Query.prems(2) by auto
    qed (auto simp add: fminsert_def fmlookup_default_def)
  qed
next
  case (Iterate x c infl stabl σ)
  then have inv: "invariant c σ infl (insert x stabl)" using invariant_simp_c_stabl
by metis
  have xstabl: "x ∈ insert x stabl" by simp
  have stablfinite: "finite (insert x stabl)" using Iterate.prems(4) by
auto
  show ?case using Iterate.IH(1) Iterate.prems(1-2)
  proof(cases rule: iterate_continue_fixpoint_cases)
    case Stable
    have "TD_plain.invariant T stabl (c - {x}) σ"
      using Iterate.prems(3) dep_subset_stable[of infl stabl]
      unfolding invariant_def TD_plain.invariant_def[of T]
      by auto
    then have "TD_plain.iterate_dom T x c σ" and "TD_plain.iterate T
x c σ = (xd, σ)"
```

```
      using Stable(5,4) Iterate.prems(2,4) TD_plain.td1_terminates_for_stabl[of
x stabl T] by auto
    then show ?thesis using Stable(2,3,5) Iterate.prems(1,3,4) Iterate.IH(1)
by auto
  next
    case Fixpoint
    note IH = Iterate.IH(2)[OF Fixpoint(4,2) inv xstabl stablfinite, folded
eq_def dep_def]
    then show ?thesis
    proof(intro conjI, goal_cases)
      case 1 then show ?case unfolding invariant_def
      proof(intro conjI, goal_cases)
        case 1 then have "part_solution σ' (stabl' - (c - {x}))"
          using Fixpoint(3) unfolding eq_def invariant_def by auto
        then show ?case using IH invariant_def by auto
      next
        case 2
        then show ?case using Fixpoint(3) by auto
      next
        case 3 then show ?case using Iterate.prems(2) by (simp add:
insert_absorb)
      qed auto
    next
      case 2 then show dom: ?case
        using Fixpoint(3) TD_plain.query_iterate_eval.domintros(2)[of
T, folded TD_plain_dom_defs]
        by (metis prod.inject)
      case 3 then show ?case using dom Fixpoint(3) by (auto split: prod.splits)
    next
      case 6 then show ?case
        using Fixpoint(4) by blast
    next case 8
      have "x ∉ fset (fmdom infl)"
        using Iterate.prems(3) Fixpoint(4)
        unfolding invariant_def
        by auto
      then have "slookup infl x = {}"
        unfolding fmlookup_default_def
        by (simp add: fmdom_notD)
      then show ?case
        using Fixpoint(4) IH lookup_in_influenced
        by auto
    qed auto
  next
    case (Continue stabl1 infl1 σ1 xd' stabl2 infl2)
    have infl2_def: "infl2 = fmdrop_set (insert x (influenced_by infl1
x)) infl1"
      and stabl2_def: "stabl2 = stabl1 - influenced_by infl1 x"
      using destab_infl_stabl_relation[of infl2 stabl2 x infl1 stabl1]
```

```
Continue(4) by auto
    note IH = Iterate.IH(2)[OF Continue(7,2) inv xstabl stablfinite]

    have "(slookup infl1 ξ - slookup infl ξ) ∩ stabl = {}" for ξ
      using Iterate.prems(3) Continue(7) IH
      unfolding invariant_def
      by auto
    then have stabl_inc: "stabl ⊆ stabl2"
      using Iterate.prems(3) Continue(4,7) new_lookup_to_infl_not_stabl[of
infl1 infl stabl x]
        destab_infl_stabl_relation[of infl2 stabl2] IH
      unfolding invariant_def
      by auto

    have infl_closed: "(∀x∈stabl1 - (c - {x}). ∀y∈dep σ1 x. x ∈ slookup
infl1 y)"
      using IH[unfolded invariant_def, folded dep_def] by auto

    have x_no_dep: "x ∉ dep σ1 y" if "y ∈ stabl2 - (c - {x})" for y
      using that Continue(4) destab_infl_stabl_relation destab_x_no_dep[OF
_ infl_closed]
        by fastforce

    have "invariant (c - {x}) (σ1(x ↦ xd')) infl2 stabl2"
      using IH Iterate.prems(2,3) Continue(4,7)
      unfolding invariant_def
    proof(elim conjE, intro conjI, goal_cases)
      case 1
      define σ2 where [simp]: "σ2 = σ1(x ↦ xd')"
      show ?case using 1(4) stabl_inc by auto
      case 2
      show ?case
        using 2(2,8,15) destab_upd_preserves_part_sol infl_closed
        by auto
      case 3
      show ?case using 3(2,12) destab_preserves_infl_dom_stabl by auto
      case 4
      show ?case
      proof(intro ballI, goal_cases)
        case (1 y z)
        have "dep σ1 y = dep σ2 y" using x_no_dep[OF 1(1)] dep_eq[of
σ1 _ σ2] σ2_def fun_upd_apply
          unfolding mlup_def by metis
        then show ?case using 1 4(2) destab_and_upd_preserves_dep_closed_in_infl
infl_closed by auto
      qed
    qed
    then have "invariant (c - {x}) (σ1(x ↦ xd')) infl2 stabl2" by simp+
    note inv = this
```

```
    have B: "finite stabl2"
      by (metis Continue(4) Diff_subset IH destab_infl_stabl_relation
infinite_super)
    note ih = Iterate.IH(3)[OF Continue(7,2) _ _ _ Continue(3,4) _ Continue(6)
Iterate.prems(2) inv
        B, of "(infl1, stabl1, σ1)" "(stabl1, σ1)", simplified, folded
TD_plain_dom_defs]
    then show ?thesis
    proof(intro conjI, goal_cases)
      case 2 show dom: ?case
        using IH TD_plain.query_iterate_eval.domintros(2)[of T x c σ,
folded TD_plain_dom_defs] ih
        by (metis Pair_inject)
      case 3 then show ?case using dom Continue(3) IH ih
        by (auto split: prod.split)
    next case 6 then show ?case
        using stabl_inc by auto
    next case 7
      then show ?case unfolding invariant_def
      proof(elim conjE, intro ballI subsetI, goal_cases)
        case (1 ξ u)
        have "ξ ∉ insert x (influenced_by infl1 x)"
          using 1(13) Continue(7) stabl2_def stabl_inc by blast
        then show ?case
          using stabl_inc infl2_def 1(10,13,14) IH
            fmlookup_default_drop_set[of "insert x (influenced_by infl1
x)" infl1 ξ]
          by fastforce
      qed
    next case 8
      then show ?case unfolding invariant_def
      proof(intro allI, goal_cases)
        case (1 ξ)
        have "slookup infl2 ξ ⊆ slookup infl1 ξ"
          using infl2_def unfolding fmlookup_default_def by auto
        moreover have "(slookup infl' ξ - slookup infl2 ξ) ∩ stabl =
{}"
        proof (cases "x ∈ stabl2")
          case True
          then show ?thesis using Continue(5,6) by auto
        next
          case False
          then show ?thesis
            using 1(1) inv[unfolded invariant_def] stabl_inc
            by fastforce
        qed
        moreover have "(slookup infl1 ξ - slookup infl ξ) ∩ stabl =
{}"
          using Continue(7) Iterate.prems(3) IH stabl_infl_empty[of x
```

```
stabl infl]
          unfolding invariant_def by auto
        ultimately show ?case using infl2_def stabl2_def by blast
      qed
    qed auto
  qed
next
  case (Eval x t c infl stabl σ)
  show ?case using Eval.IH(1) Eval.prems(1)
  proof(cases rule: eval_query_answer_cases)
    case (Query y g yd infl1 stabl1 σ1)
    note IH = Eval.IH(2)[OF Query(1,3) Eval.prems(2,4)]
    then have "invariant c σ1 infl1 stabl1
        ∧ TD_plain.invariant T
          stabl1 c σ1"
      using Eval.prems(3)
      unfolding invariant_def
    proof(elim conjE, intro conjI, goal_cases)
      case 1 show ?case using 1(2) .
    next
      case 2 show ?case using 2(4) .
    next
      case 3 show ?case using 3(6) .
    next
      case 4 show ?case using 4(7) .
    next
      case 5 show ?case using Eval.prems(3) IH
        reach_cap_tree_simp2 dep_eq unfolding TD_plain.invariant_def
        by (meson "5"(13) dep_subset_stable)
    qed
    then have "invariant c σ1 infl1 stabl1"
      and "TD_plain.invariant T stabl1 c σ1"
      by simp+
    note inv = this
    have B: "finite stabl1" using IH by simp
    have C: "x ∈ stabl1" using IH Eval.prems(3) by blast
    note ih = Eval.IH(3)[OF Query(1,3) _ _ _ Query(5) inv(1) C B,
        of "(infl1, stabl1, σ1)" "(stabl1, σ1)", simplified, folded TD_plain_dom_defs]

    have "y ∈ stabl1"
      using IH stabl_infl_empty[of y stabl1 infl1]
      unfolding invariant_def
      by fastforce
    then have "mlup σ1 y = mlup σ' y"
      using TD_plain.partial_correctness_ind(3)[of T x "g yd" c σ1 xd
σ' stabl1] inv ih by auto
    then have mlup: "mlup σ' y = yd"
      using Query(6) by auto
```

```
    show ?thesis using ih
    proof(intro conjI, goal_cases)
      case 2
      then show dom: ?case
        using IH Query(1) TD_plain.query_iterate_eval.domintros(3)[of
t T, folded TD_plain_dom_defs]
        by (cases "TD_plain.query T x y c σ") fastforce
      case 3
      then show ?case
        using dom IH Query(1)
          TD_plain.query_iterate_eval.domintros(3)[of t T, folded TD_plain_dom_defs]
        by (auto split: prod.splits)
    next
      case 5
      then show ?case using Query IH mlup unfolding invariant_def by
auto
    next
      case 6
      then show ?case using 6 Query IH mlup ‹y ∈ stabl1› unfolding invariant_def
by auto
    next
      case 7
      then show ?case using IH by auto
    next
      case 8
      then show ?case using IH by blast
    next
      case 9
      then show ?case
        using infl_diff_eval_step[of stabl stabl1 infl' infl1 x] IH ih
Eval.prems(2,3) by auto
    qed auto
  next
    case Answer
    then show ?thesis using Answer TD_plain.query_iterate_eval.domintros(3)
Eval.prems(2-3,4)
      by fastforce
  qed
qed

corollary TD_TD_plain_equivalence:
  assumes "solve_dom x"
    and "solve x = (stabl, σ)"
  shows "TD_plain.solve_dom T x ∧ TD_plain.solve T x = σ"
proof -
  obtain xd infl where iter: "(xd, infl, stabl, σ) = iterate x {x} fmempty
{} Map.empty"
    using assms(2) unfolding solve_def by (auto split: prod.splits)
  have inv: "invariant ({x} - {x}) Map.empty fmempty {}" unfolding invariant_def
```

```
by fastforce
  have "TD_plain.iterate_dom T x {x} (λx. None) ∧ (xd, σ) = TD_plain.iterate
T x {x} (λx. None)"
    using TD_TD_plain_equivalence_ind(2)[OF assms(1)[unfolded solve_dom_def]
iter _ inv, simplified]
    by auto
  then show ?thesis unfolding TD_plain.solve_dom_def TD_plain.solve_def
by (auto split: prod.splits)
qed
```

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, σ)"
  shows "part_solution σ stabl" and "reach σ x ⊆ stabl"
proof(goal_cases)
  note dom = assms(1)[unfolded solve_dom_def]
  obtain infl xd where app: "(xd, infl, stabl, σ) = iterate x {x} fmempty
{} Map.empty"
    using assms unfolding solve_def by (cases "iterate x {x} fmempty
{} Map.empty") auto
  case 1 show ?case using TD_TD_plain_equivalence_ind(2)[OF dom app,
unfolded invariant_def] by auto
  case 2 show ?case
    using TD_TD_plain_equivalence_ind(2)[OF dom app, unfolded invariant_def]
      reach_empty_capped dep_closed_implies_reach_cap_tree_closed
      dep_subset_stable[of infl stabl "{}"] by auto
qed
```

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 × 'a × 'a set × ('a, 'a list) fmap ×
'a set × ('a, 'b) map"
  | I "'a × 'a set × ('a, 'a list) fmap × 'a set × ('a, 'b) map"
  | E "'a × ('a,'b) strategy_tree × 'a set × ('a, 'a list) fmap × 'a
set × ('a, 'b) map"

partial_function (option) solve_rec_c ::
  "('x, 'd) state ⇒ ('d × ('x, 'x list) fmap × 'x set × ('x, 'd) map)
option"
  where
```

```
  "solve_rec_c s = (case s of Q (y,x,c,infl,stabl,σ) ⇒ Option.bind
      (if x ∈ c then
        Some (mlup σ x, infl, stabl, σ)
      else
        solve_rec_c (I (x, (insert x c), infl, stabl, σ)))
      (λ (xd, infl, stabl, σ). Some (xd, fminsert infl x y, stabl, σ))
  | I (x,c,infl,stabl,σ) ⇒
      if x ∉ stabl then Option.bind (
        solve_rec_c (E (x, (T x), c, infl, insert x stabl, σ))) (λ(d_new,
infl, stabl, σ).
        if mlup σ x = d_new then
          Some (d_new, infl, stabl, σ)
        else
          let (infl, stabl) = destab x infl stabl in
          solve_rec_c (I (x, c, infl, stabl, σ(x ↦ d_new))))
      else
        Some (mlup σ x, infl, stabl, σ)
  | E (x,t,c,infl,stabl,σ) ⇒ (case t of
        Answer d ⇒ Some (d, infl, stabl, σ)
      | 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 *(induction x y c infl stabl σ* **and** *x c infl stabl σ* **and** *x t c infl stabl σ*

```
    rule: query_iterate_eval_pinduct)
  case (Query x y c infl stabl σ)
  show ?case
  proof (cases "y ∈ c")
    case True
    then have "solve_rec_c (Q (x, y, c, infl, stabl, σ))
        = Some (mlup σ y, fminsert infl y x, stabl, σ)"
      by simp
    moreover have "query x y c infl stabl σ = (mlup σ y, fminsert infl
y x, stabl, σ)"
      using query.psimps[folded dom_defs] Query(1) True by force
    ultimately show ?thesis unfolding solve_rec_c_dom_def by auto
  next
    case False
    obtain d1 infl1 stabl1 σ1 where
        I: "iterate y (insert y c) infl stabl σ = (d1, infl1, stabl1,
σ1)"
      using prod_cases4 by blast
    then have J: "query x y c infl stabl σ = (d1, fminsert infl1 y x,
stabl1, σ1)"
      using False Query.IH(1) query.pelims[folded dom_defs] by fastforce
    then have "solve_rec_c (I (y, insert y c, infl, stabl, σ)) = Some
(d1, infl1, stabl1, σ1)"
      using Query(2) False I by (simp add: solve_rec_c_dom_def)
    then have "solve_rec_c (Q (x, y, c, infl, stabl, σ)) = Some (d1,
fminsert infl1 y x, stabl1, σ1)"
      using False by simp
    moreover have "solve_rec_c_dom (Q (x, y, c, infl, stabl, σ))"
      using Query(2) False unfolding solve_rec_c_dom_def by fastforce
    ultimately show ?thesis using Query J unfolding solve_rec_c_dom_def
by auto
  qed
next
  case (Iterate x c infl stabl σ)
  show ?case
  proof (cases "x ∈ stabl")
    case True
    have "iterate_dom x c infl stabl σ ∧
        iterate x c infl stabl σ = (mlup σ x, infl, stabl, σ)"
      using True iterate.psimps query_iterate_eval.domintros(2)
      unfolding iterate_dom_def
      by fastforce
    then show ?thesis using True unfolding solve_rec_c_dom_def by auto
  next
    case False
    obtain d1 infl1 stabl1 σ1 where
        eval: "eval x (T x) c infl (insert x stabl) σ = (d1, infl1, stabl1,
σ1)"
        "solve_rec_c (E (x, T x, c, infl, insert x stabl, σ)) = Some (d1,
```

```
infl1, stabl1, σ1)"
      using Iterate(2) solve_rec_c_dom_def False by force
    show ?thesis
    proof (cases "mlup σ1 x = d1")
      case True
      have "iterate x c infl stabl σ = (d1, infl1, stabl1, σ1)"
        using eval iterate.psimps[folded dom_defs, OF Iterate(1)] True
False by simp
      moreover have "solve_rec_c (I (x, c, infl, stabl, σ)) = Some (d1,
infl1, stabl1, σ1)"
        using eval False True by simp
      ultimately show ?thesis unfolding solve_rec_c_dom_def by simp
    next
      case False
      obtain infl2 stabl2 where destab: "(infl2, stabl2) = destab x infl1
stabl1"
        by (cases "destab x infl1 stabl1") auto
      have "solve_rec_c_dom (I (x, c, infl2, stabl2, σ1(x ↦ d1)))"
        and "iterate x c infl2 stabl2 (σ1(x ↦ d1)) =
          the (solve_rec_c (I (x, c, infl2, stabl2, σ1(x ↦ d1))))"
        using Iterate(3)[OF ‹x ∉ stabl› eval(1)[symmetric] _ _ _ False
destab] by blast+
      moreover have "iterate x c infl stabl σ = iterate x c infl2 stabl2
(σ1(x ↦ d1))"
        using eval iterate.psimps[folded dom_defs, OF Iterate(1)] False
‹x ∉ stabl› destab
        by (smt (verit) case_prod_conv)
      moreover have "solve_rec_c (I (x, c, infl, stabl, σ))
          = solve_rec_c (I (x, c, infl2, stabl2, σ1(x ↦ d1)))"
        using ‹x ∉ stabl› False eval(2) destab[symmetric] by simp
      ultimately show ?thesis unfolding solve_rec_c_dom_def by auto
    qed
  qed
next
  case (Eval x t c infl stabl σ)
  show ?case
  proof (cases t)
    case (Answer d)
    then have "eval x t c infl stabl σ = (d, infl, stabl, σ)"
      using eval.psimps query_iterate_eval.domintros(3) dom_defs(3) by
fastforce
    then show ?thesis using Eval Answer unfolding solve_rec_c_dom_def
by simp
  next
    case (Query y g)
    then obtain d1 infl1 stabl1 σ1 where
        query: "solve_rec_c (Q (x, y, c, infl, stabl, σ)) = Some (d1,
infl1, stabl1, σ1)"
          "query x y c infl stabl σ = (d1, infl1, stabl1, σ1)"
```

```
      using Query Eval(2) unfolding solve_rec_c_dom_def by auto
    then have "solve_rec_c_dom (E (x, g d1, c, infl1, stabl1, σ1))"
        "eval x (g d1) c infl1 stabl1 σ1 = the (solve_rec_c (E (x, g d1,
c, infl1, stabl1, σ1)))"
      using Eval(3)[OF Query] by auto
    moreover have "eval x t c infl stabl σ = eval x (g d1) c infl1 stabl1
σ1"
      using Eval.IH(1) Query eval.psimps eval_dom_def query
      by fastforce
    moreover have "solve_rec_c (E (x, t, c, infl, stabl, σ))
        = solve_rec_c (E (x, g d1, c, infl1, stabl1, σ1))"
      using Query query solve_rec_c.simps[of "E (x,t,c,infl,stabl,σ)"]
      by (simp del: solve_rec_c.simps)
    ultimately show ?thesis using solve_rec_c_dom_def by force
  qed
qed

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 (induction arbitrary: s r rule: solve_rec_c.fixp_induct)
  case 1
  then show ?case using option_admissible by fast
next
  case 2
  then show ?case by simp
next
  case (3 S)
  show ?case
  proof (cases s)
    case (Q a)
    obtain x y c infl stabl σ where "a = (x, y, c, infl, stabl, σ)" us-
ing prod_cases6 by blast
    have "query_dom x y c infl stabl σ ∧ query x y c infl stabl σ = r"
    proof (cases "y ∈ c")
      case True
      then have "Some (mlup σ y, fminsert infl y x, stabl, σ) = Some
r"
        using 3(2) Q ‹a = (x, y, c, infl, stabl, σ)› by simp
      then show ?thesis using query.psimps[folded query_dom_def, of x
y c infl stabl σ]
            query_iterate_eval.domintros(1)[folded query_dom_def, of y
c infl] True by simp
    next
```

```
      case False
      then have "Option.bind (S (I (y, insert y c, infl, stabl, σ)))
(λ(d,infl,stabl,σ).
          Some (d, fminsert infl y x, stabl, σ)) = Some r"
        using 3(2) Q ‹a = (x, y, c, infl, stabl, σ)› by simp
      then obtain d1 infl1 stabl1 σ1
        where "S (I (y, insert y c, infl, stabl, σ)) = Some (d1 ,infl1,
stabl1, σ1)"
        and "(d1, fminsert infl1 y x, stabl1, σ1) = r"
        by (cases "S (I (y, insert y c, infl, stabl, σ))") auto
      then have "iterate_dom y (insert y c) infl stabl σ
          ∧ iterate y (insert y c) infl stabl σ = (d1, infl1, stabl1,
σ1)"
        using 3(1) unfolding iterate_dom_def by fastforce
      then show ?thesis using False ‹(d1, fminsert infl1 y x, stabl1,
σ1) = r›
        by (simp add: query_iterate_eval.domintros(1) False)
    qed
    then show ?thesis using Q ‹a = (x, y, c, infl, stabl, σ)› by simp
  next
    case (I a)
    obtain x c infl stabl σ where "a = (x, c, infl, stabl, σ)" using
prod_cases5 by blast
    show ?thesis
    proof(cases "x ∈ stabl")
      case True
      then have "(mlup σ x, infl, stabl, σ) = r" using I ‹a = (x, c,
infl, stabl, σ)› 3(2) by simp
      moreover have "iterate_dom x c infl stabl σ
          ∧ iterate x c infl stabl σ = (mlup σ x, infl, stabl, σ)"
        using True query_iterate_eval.domintros(2) iterate.psimps dom_defs
by fastforce
      ultimately show ?thesis using I ‹a = (x, c, infl, stabl, σ)› by
simp
    next
      case False
      then have IH1: "Option.bind (S (E (x, T x, c, infl, insert x stabl,
σ)))
         (λ(d_new, infl, stabl, σ).
           if mlup σ x = d_new then Some (d_new, infl, stabl, σ)
           else let (infl, stabl) = destab x infl stabl in
            S (I (x, c, infl, stabl, σ(x ↦ d_new)))) = Some r"
        using 3(2) I ‹a = (x, c, infl, stabl, σ)› by simp
      then obtain d_new infl1 stabl1 σ1
        where eval_some: "S (E (x, T x, c, infl, insert x stabl, σ))
          = Some (d_new, infl1, stabl1, σ1)"
        using 3(2) I
        by (cases "S (E (x, T x, c, infl, insert x stabl, σ))") auto
      then have eval: "eval_dom x (T x) c infl (insert x stabl) σ
```

```
          ∧ eval x (T x) c infl (insert x stabl) σ = (d_new, infl1, stabl1,
σ1)"
        using 3(1) unfolding TD_plain.eval_dom_def by force
      have "iterate_dom x c infl stabl σ ∧ iterate x c infl stabl σ =
r"
      proof (cases "mlup σ1 x = d_new")
        case True
        then have "(d_new, infl1, stabl1, σ1) = r" using IH1 eval_some
by simp
        moreover have "iterate_dom x c infl stabl σ"
          using query_iterate_eval.domintros(2)[folded dom_defs] False
True eval by fastforce
        ultimately show ?thesis
          using iterate.psimps[folded dom_defs] False True eval by fastforce
      next
        case False
        obtain infl2 stabl2 where destab: "(infl2, stabl2) = destab x
infl1 stabl1"
          by (cases "destab x infl1 stabl1") auto
        then have "S (I (x, c, infl2, stabl2, σ1(x ↦ d_new))) = Some
r"
          using IH1 False eval_some by (smt (verit, best) bind.bind_lunit
case_prod_conv)
        then have iter_cont: "iterate_dom x c infl2 stabl2 (σ1(x ↦ d_new))
            ∧ iterate x c infl2 stabl2 (σ1(x ↦ d_new)) = r"
          using 3(1) unfolding iterate_dom_def by fastforce
        then have "iterate_dom x c infl stabl σ"
          using query_iterate_eval.domintros(2)[folded dom_defs destab.simps,
              of x stabl c infl σ] eval ‹x ∉ stabl› False destab
          by (cases "destab x infl1 stabl1") auto
        then show ?thesis
          using iterate.psimps[folded dom_defs, of x c infl stabl σ] ‹x
∉ stabl› destab eval
            False iter_cont
          by (cases "destab x infl1 stabl1") auto
      qed
      then show ?thesis
        using I ‹a = (x, c, infl, stabl,  σ)› by simp
    qed
  next
    case (E a)
    obtain x t c infl stabl σ where "a = (x, t, c, infl, stabl, σ)" us-
ing prod_cases6 by blast
    then have "s = E (x, t, c, infl, stabl, σ)" using E by auto
    have "eval_dom x t c infl stabl σ ∧ eval x t c infl stabl σ = r"
    proof (cases t)
      case (Answer d)
      then have "eval_dom x t c infl stabl σ"
        unfolding eval_dom_def
```

```
        using query_iterate_eval.domintros(3)
        by fastforce
      moreover have "eval x t c infl stabl σ = (d, infl, stabl, σ)"
        using Answer eval.psimps[folded dom_defs, OF calculation] by auto
      moreover have "(d, infl, stabl, σ) = r"
        using 3(2) ‹s = E (x, t, c, infl, stabl, σ)› Answer by simp
      ultimately show ?thesis by simp
    next
      case (Query y g)
      then have A: "Option.bind (S (Q (x, y, c, infl, stabl, σ))) (λ(yd,
infl, stabl, σ).
        S (E (x, g yd, c, infl, stabl, σ))) = Some r" using ‹s = E (x,
t, c, infl, stabl, σ)› 3(2)
        by simp
      then obtain yd infl1 stabl1 σ1
          where S1: "S (Q (x, y, c, infl, stabl, σ)) = Some (yd, infl1,
stabl1, σ1)"
          and S2: "S (E (x, g yd, c, infl1, stabl1, σ1)) = Some r"
        by (cases "S (Q (x, y, c, infl, stabl, σ))") auto
      then have "query_dom x y c infl stabl σ
          ∧ query x y c infl stabl σ = (yd, infl1, stabl1, σ1)"
          and "eval_dom x (g yd) c infl1 stabl1 σ1 ∧ eval x (g yd) c
infl1 stabl1 σ1 = r"
        using 3(1)[OF S1] 3(1)[OF S2] unfolding TD_plain.dom_defs by force+
      then show ?thesis
        using query_iterate_eval.domintros(3)[folded dom_defs] eval.psimps[folded
dom_defs] Query
        by fastforce
    qed
    then show ?thesis
      using E ‹a = (x, t, c, infl, stabl, σ)› by simp
  qed
qed

theorem term_equivalence: "solve_dom x ⟷ solve_c_dom x"
  using solve_rec_c_query_iterate_eval_equiv[of "I (x, {x}, fmempty, {},
λx. None)"]
    query_iterate_eval_solve_rec_c_equiv(2)[of x "{x}" fmempty "{}" "λx.
None"]
  unfolding solve_dom_def solve_c_dom_def solve_rec_c_dom_def solve_c_def
  by (cases "solve_rec_c (I (x, {x}, fmempty, {}, λx. None))") fastforce+

theorem value_equivalence: "solve_dom x ⟹ ∃σ. solve_c x = Some σ ∧
solve x = σ"
proof goal_cases
  case 1
  then obtain r where "solve_rec_c (I (x, {x}, fmempty, {}, λx. None))
= Some r
    ∧ iterate x {x} fmempty {} (λx. None) = r"
```

```
    using query_iterate_eval_solve_rec_c_equiv(2)[OF 1[unfolded solve_dom_def]]
    unfolding solve_rec_c_dom_def solve_dom_def
    by fastforce
  then show ?case unfolding solve_c_def solve_def by (auto split: prod.split)
qed
```

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 (cases "solve_dom x")
  case True
  then show ?thesis
    using solve_c_def solve_def value_equivalence by fastforce
next
  case False
  then have "solve_c x = None" using solve_c_dom_def term_equivalence
by (meson option.exhaust)
  then show ?thesis by auto
qed

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
  done

end
```

5 Example

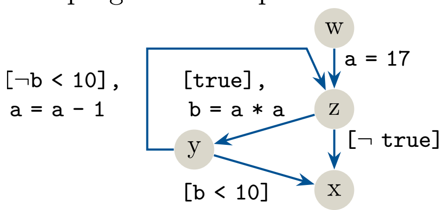
```
theory Example
  imports TD_plain TD_equiv
begin
```

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})"
  by auto

setup_lifting D.type_definition_D

lift_definition top :: "D" is "{}" by simp
lift_definition bot :: D is "{a, b}" by simp
lift_definition join :: "D ⇒ D ⇒ D" is Set.inter by blast
```

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

```
lift_definition insert :: "pv ⇒ D ⇒ D"
  is "λe d. if e ∈ {a, b} then Set.insert e d else d"
  by auto
definition set_to_D :: "pv set ⇒ D" where
  "set_to_D = (λs. fold (λe acc. if e ∈ 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 ..
end

instantiation D :: equal
begin
  definition equal_D :: "D ⇒ D ⇒ bool"
  where "equal_D d1 d2 = ((Rep_D d1) = (Rep_D d2))"

  instance by standard (simp add: equal_D_def Rep_D_inject)
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.

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

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

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

fun ConstrSys :: "Unknown ⇒ (Unknown, D) strategy_tree" where
  "ConstrSys X = Query Y (λd1. if d1 = top then Answer top
    else Query Z (λd2. Answer (join d1 d2)))"
| "ConstrSys Y = Query Z (λd. if d ∈ {top, set_to_D {b}}
    then Answer (set_to_D {b}) else Answer bot)"
| "ConstrSys Z = Query Y (λd1. if d1 ∈ {top, set_to_D {a}}
    then Answer (set_to_D {a})
    else Query W (λd2. if d2 ∈ {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^2I. 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.