Partial Correctness of the Top-Down Solver

Yannick Stade*, Sarah Tilscher*, Helmut Seidl

May 26, 2024

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

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 $\bot$ 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"*
  ⟨*proof*⟩

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

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

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

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

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

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

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

2.3 Functions on the Constraint System

The function *rhs_length* computes the length of a specific path in the strategy tree defined by a value assignment for unknowns 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"
  ⟨proof⟩

termination rhs_length
⟨proof⟩
```

The function *traverse_rhs* traverses a strategy tree and determines the answer when choosing the path through the strategy tree based on a given unknown-value mapping σ

```
function (domintros) traverse_rhs where
  "traverse_rhs (Answer d) _ = d" |
  "traverse_rhs (Query x f) σ = traverse_rhs (f (mlup σ x)) σ"
  ⟨proof⟩

termination traverse_rhs
  ⟨proof⟩
```

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⟩
```

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

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

2.5 Dependencies between Unknowns

The set *dep 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)))"
  ⟨*proof*⟩

termination *dep_aux*
  ⟨*proof*⟩

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

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

lemmas *dep_eq = dep_aux_eq[of 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"
  ⟨*proof*⟩

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

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

2.6 Set *Reach*

Let *reach* be the set of all unknowns contributing to *x* (for a given 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*⟩

lemma *dep_subset_reach_cap_tree:*
  "dep_aux σ' t ⊆ reach_cap_tree σ' c t"
⟨*proof*⟩

lemma *reach_cap_tree_subset:*
  shows *"reach_cap_tree σ c t ⊆ reach_cap_tree σ (c - {x}) t"*
⟨*proof*⟩

lemma *reach_empty_capped:*
  shows *"reach σ x = insert x (reach_cap_tree σ {x} (T x))"*
⟨*proof*⟩

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*⟩

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⟩

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)"
  ⟨proof⟩

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⟩

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⟩

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⟩

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)"
  ⟨proof⟩

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"
  ⟨proof⟩

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}"
  ⟨proof⟩
```

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)"
  ⟨proof⟩

end

end
```

3 The plain Top-Down Solver

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

```
theory TD_plain
  imports Basics
begin

locale TD_plain = Solver D T
  for D :: "'d :: bot"
    and T :: "'x ⇒ ('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))"
  ⟨proof⟩

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

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

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

lemmas dom_defs = query_dom_def iterate_dom_def eval_dom_def
```

3.2 Refinement of Auto-Generated Rules

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

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

```
    ]
    query_iterate_eval.pinduct(2)[
      folded query_dom_def iterate_dom_def eval_dom_def,
      of x c 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⟩
```

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⟩
```

```
lemma iterate_fmlookup:
  assumes "iterate_dom x c σ"
    and "iterate x c σ = (xd, σ')"
    and "x ∈ c"
  shows "mlup σ' x = xd"
  ⟨proof⟩

corollary query_fmlookup:
  assumes "query_dom x y c σ"
    and "query x y c σ = (yd, σ')"
  shows "mlup σ' y = yd"
  ⟨proof⟩

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"
  ⟨proof⟩

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'"
  ⟨proof⟩
```

3.5 Predicate for Valid Input States

We define a predicate for valid input solver states. `c` is the set of called unknowns, i.e., the unknowns currently being evaluated and 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"
  ⟨proof⟩

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⟩
```

3.6 Partial Correctness Proofs

```
lemma x_not_stable:
  assumes "eq x σ ≠ mlup σ x"
    and "part_solution σ s"
  shows "x ∉ s"
  ⟨proof⟩
```

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

```
lemma already_solution:
  shows "query_dom x y c 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*⟩

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*⟩

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*⟩

3.7 Termination of TD_plain for Stable Unknowns

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

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

3.8 Program Refinement for Code Generation

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

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

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

declare solve_rec_c.simps[simp,code]

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

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

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

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

```
lemma query_iterate_eval_solve_rec_c_equiv:
  shows "query_dom x y c 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⟩
```

```
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⟩
```

```
theorem term_equivalence: "solve_dom x ⟷ solve_c_dom x"
  ⟨proof⟩
```

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

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⟩

end
```

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

```
global_interpretation TD_plain_Interp: TD_plain D T for D T
  defines TD_plain_Interp_solve = TD_plain_Interp.solve
  ⟨proof⟩

end
```

4 The Top-Down Solver

In this theory we proof the partial correctness of the original TD by establishing its equivalence with the TD_plain. Compared to the TD_plain, it additionally tracks a set of currently stable unknowns `stabl`, and a map `infl` collecting for each unknown `x` a list of unknowns influenced by it. This allows for the optimization that skips the re-evaluation of unknowns which are already stable. It does, however, also require a destabilization mechanism triggering re-evaluation of all unknowns possibly affected by an unknown whose value has changed.

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

declare fun_upd_apply[simp del]

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

4.1 Definition of Destabilize and Proof of its Termination

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

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

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

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

declare *destab_dom_def[simp]*

lemma *destab_domintros:*
```
  "destab_iter_dom [] infl stabl"
  "destab_dom y infl (stabl - {y}) ⟹
    destab y infl (stabl - {y}) = (infl', stabl') ⟹
    destab_iter_dom ys infl' stabl' ⟹
    destab_iter_dom (y # ys) infl stabl"
  "destab_iter_dom (fmlookup_default infl [] x) (fmdrop x infl) stabl
⟹ destab_dom x infl stabl"
```
⟨*proof*⟩

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

lemma *count_non_empty_dec_fmdrop:*
assumes *"fmlookup_default m [] x ≠ []"*
shows *"Suc (count_non_empty (fmdrop x m)) = count_non_empty m"*
⟨*proof*⟩

lemma *count_non_empty_eq_fmdrop:*
assumes *"fmlookup_default m [] x = []"*
shows *"count_non_empty (fmdrop x m) = count_non_empty m"*
⟨*proof*⟩

termination
⟨*proof*⟩

4.2 Definition of the Solver Algorithm

Apart from passing the additional arguments for the solver state, the *iterate* function contains, compared to the TD_plain, an additional check to skip iteration of already stable unknowns. Furthermore, the helper function *destabilize* is called whenever the newly 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))"
  ⟨proof⟩
```

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⟩
```

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⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩
```

```
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"
  ⟨proof⟩
```

4.6 Description of the Effect of Destabilize

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

```
inductive_set influenced_by for infl x where
  base: "fmlookup infl x = Some ys ⟹ 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))"
⟨proof⟩
```

```
lemma lookup_in_influenced:
  shows "slookup infl x ⊆ influenced_by infl x"
```

⟨*proof*⟩

lemma *influenced_unknowns_fmdrop_set:*
  shows *"influenced_by (fmdrop_set C infl) x = influenced_by_cutoff infl x C"*
⟨*proof*⟩

lemma *influenced_by_transitive:*
  assumes *"y ∈ influenced_by infl x"*
    and *"z ∈ influenced_by infl y"*
  shows *"z ∈ influenced_by infl x"*
  ⟨*proof*⟩

lemma *influenced_cutoff_subset:*
  "influenced_by_cutoff infl x C ⊆ influenced_by infl x"
⟨*proof*⟩

lemma *influenced_cutoff_subset_2:*
  shows *"influenced_by infl x - (⋃y ∈ C. influenced_by infl y) ⊆ influenced_by_cutoff infl x C"*
⟨*proof*⟩

lemma *union_influenced_to_cutoff:*
  shows *"insert y (influenced_by infl y) ∪ influenced_by infl x =*
    insert y (influenced_by infl y) ∪ influenced_by_cutoff infl x (insert y (influenced_by infl y))"
⟨*proof*⟩

lemma *destab_iter_infl_stabl_relation:*
  shows
    "(infl', stabl') = destab_iter xs infl stabl
    ⟹ infl' = fmdrop_set (⋃x ∈ set xs. insert x (influenced_by infl x)) infl
    ∧ stabl' = stabl - (⋃x ∈ set xs. insert x (influenced_by infl x))"
  and *destab_infl_stabl_relation:*
    "(infl', stabl') = destab x infl stabl
    ⟹ infl' = fmdrop_set (insert x (influenced_by infl x)) infl
    ∧ stabl' = stabl - influenced_by infl x"
⟨*proof*⟩

4.7 Predicate for Valid Input States

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

definition *invariant* **where**
  "invariant c σ infl stabl ≡
    c ⊆ stabl
    ∧ part_solution σ (stabl - c)
    ∧ fset (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)"
  ⟨proof⟩
```

4.8 Auxiliary Lemmas for Partial Correctness Proofs

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

lemma *dep_closed_implies_reach_cap_tree_closed:*
```
  assumes "x ∈ stabl'"
    and "∀ξ∈stabl' - (c - {x}). dep σ' ξ ⊆ stabl'"
  shows "reach_cap σ' (c - {x}) x ⊆ stabl'"
⟨proof⟩
```

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)"
  ⟨proof⟩
```

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⟩
```

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⟩
```

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⟩
```

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⟩

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

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

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⟩

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⟩
```

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⟩

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*⟩

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*⟩

```
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*⟩

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⟩
```

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*⟩

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

theorem *term_equivalence: "solve_dom x ⟷ solve_c_dom x"*
⟨*proof*⟩

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

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

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

end

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

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

end

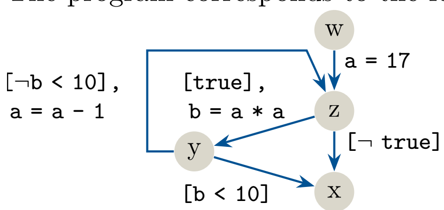
5 Example

theory *Example*
  imports *TD_plain TD_equiv*
begin

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

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

The program corresponds to the following control-flow graph.



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

5.1 Definition of the Domain

datatype *pv* = a | b

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

```
typedef D = "Pow ({a, b})"
  ⟨proof⟩

setup_lifting D.type_definition_D

lift_definition top :: "D" is "{}" ⟨proof⟩
lift_definition bot :: D is "{a, b}" ⟨proof⟩
lift_definition join :: "D ⇒ D ⇒ D" is Set.inter ⟨proof⟩
```

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"
  ⟨proof⟩
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 ⟨proof⟩
end

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

  instance ⟨proof⟩
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.

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

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

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

fun ConstrSys :: "Unknown ⇒ (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.