Stratified Datalog and Program Analysis

Anders Schlichtkrull, René Rydhof Hansen, Flemming Nielson

Abstract

In this entry we formalize stratified Datalog, the first such formalization in Isabelle to the best of our knowledge. Next we formally establish the existence of least solutions for any stratified Datalog program, essential for reasoning about negations in clauses. Lastly we illustrate the usefulness of our Datalog formalization by further formalizing the general Bit-Vector Framework and formalize and prove correct five analyses in this framework namely liveness, reaching definitions, available expressions, very busy expressions and reachability. Many of our definitions follow Nielson and Nielson's textbook [NN20]. The formalization is described in our SAC 2024 paper [SHN24].

Contents

1 Introduction

In the following we develop the first, to the best of our knowledge, Isabelle formalization of Datalog in general and stratified Datalog in particular. We use the formalization to prove the existence of least solutions (to Datalog programs) with respect to a simple ordering on predicate valuations. The ordering is from Nielson and Nielson's textbook [NNH99], and appears also in a paper by Przymusinski [Prz88] in a different formulation. We prove the two formulations equivalent. Furthermore, we apply our formalization to also formalize so-called *bit-vector analyses* [NNH99] as Datalog programs covering all the usual variants [NN20]: forward may analyses, backward may analyses, forward must analyses, and backward must analyses. The analyses are proved to correctly summarize the paths in the program, and we additionally show that each variant of our Bit-Vector Framework has an instance by formalizing four classic analyses: reaching definitions, live variables, available expressions, and very busy expressions [NNH99]. We use the labeled transition systems formalization by Schlichtkrull et al. [SSST23, SSST22] to represent program graphs.

A Rocq formalization of stratified Datalog, based on the thesis by Dumbrava [Dum16], has been developed by Benzaken et al. [BCD17], giving a comprehensive formalization focusing on correctness of evaluation strategies for (stratified) Datalog programs.

Non-stratified Datalog has been independently formalized in Rocq for use as a stepping stone for formalizing further logics (Binder, Horn clauses, BCiC) used to model and reason about authentication [Whi06] and a subset called Regular Datalog has also been formalized in Rocq [BDA18]. Rocq has also been used to formalize various semantics for Prolog (with corresponding proofs of equivalence), with the goal of proving correctness of static analyses of Prolog programs [KKB13]. Non-stratified Datalog has also been formalized in Lean with the goal of certifying results from Datalog reasoners [TGMK25].

Formalizations of analysis frameworks and the underlying theory have mainly focused on the *abstract interpretation* [CC77] approach to static analysis and primarily using Rocq, with impressive results [CP10, JLB+15]. Additionally an Isabelle formalization of abstract interpretation of a small imperative language occurs in the "fully mechanized" semantics textbook of Nipkow [NK14, Nip12].

Concrete program analyses have been formalized and proven correct in Isabelle, as witnessed in the Archive of Formal Proofs[1], including slicing, control flow analysis, conflict analysis, call arity analysis etc. [LMO07, Was08, Bre16, Was09, Bre10, LM08, Imm11, Bre15, Jia21, WL08, WLS09]. These do not use Datalog.

[1] https://www.isa-afp.org/topics/computer-science/programming-languages/static-analysis/

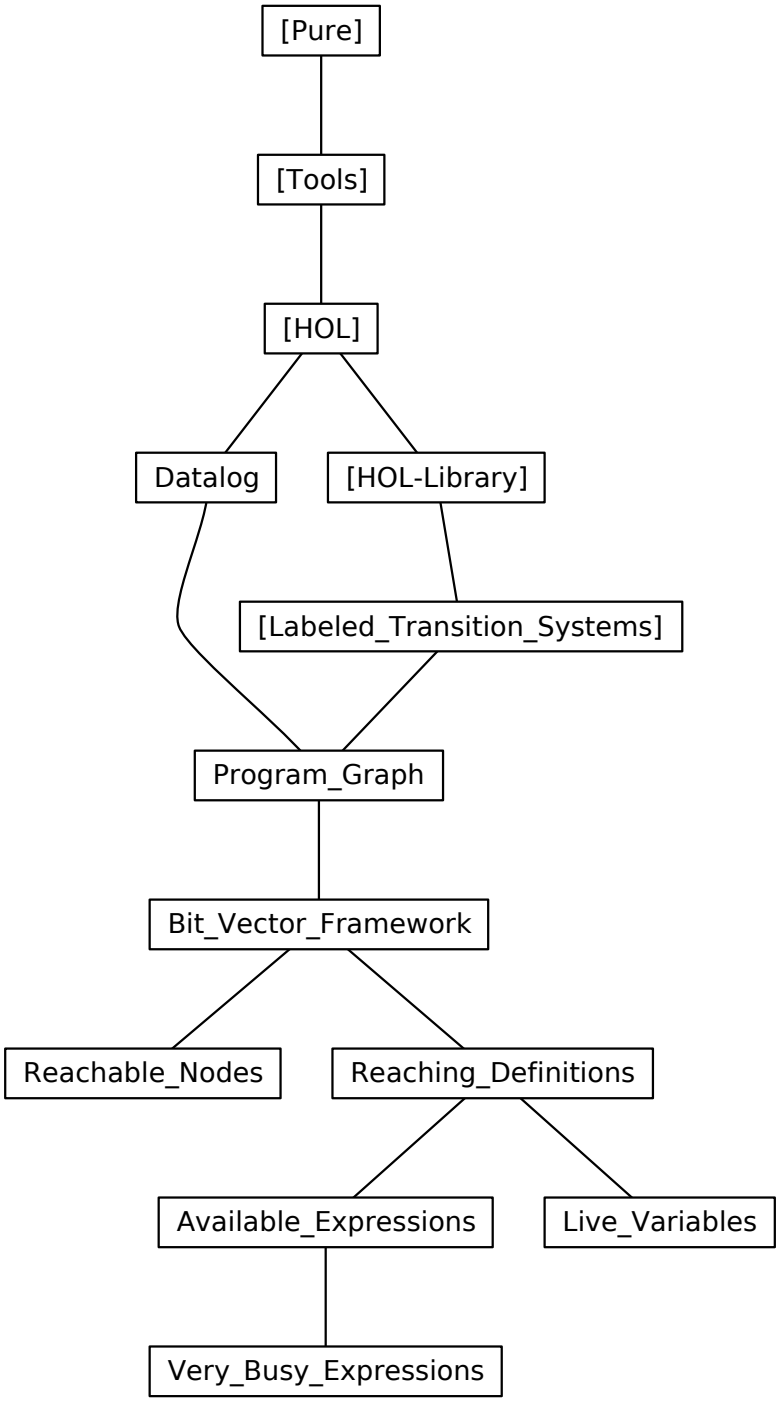

Figure 1: Theory dependency graph

theory Datalog **imports** Main **begin**

2 Datalog programs and their solutions

```
datatype (vars_id: 'x,'c) id =
  is_Var: Var 'x
  | is_Cst: Cst (the_Cst: 'c)

datatype (preds_rh: 'p,'x,'c) rh =
  Eql "('x,'c) id" "('x,'c) id" ("_ = _" [61, 61] 61)
  | Neql "('x,'c) id" "('x,'c) id" ("_ ≠ _" [61, 61] 61)
  | PosLit 'p "('x,'c) id list" ("+ _ _" [61, 61] 61)
  | NegLit 'p "('x,'c) id list" ("¬ _ _" [61, 61] 61)

type_synonym ('p,'x,'c) lh = "'p × ('x,'c) id list"

fun preds_lh :: "('p,'x,'c) lh ⇒ 'p set" where
  "preds_lh (p,ids) = {p}"

datatype (preds_cls: 'p, 'x,'c) clause = Cls 'p "('x,'c) id list" (the_rhs: "('p,'x,'c) rh list")

fun the_lh where
  "the_lh (Cls p ids rhs) = (p,ids)"

type_synonym ('p,'x,'c) dl_program = "('p,'x,'c) clause set"

definition "preds_dl dl = ⋃{preds_cls c| c. c ∈ dl}"

lemma preds_dl_union[simp]: "preds_dl (dl1 ∪ dl2) = preds_dl dl1 ∪ preds_dl dl2"
  unfolding preds_dl_def by auto

type_synonym ('x,'c) var_val = "'x ⇒ 'c"

type_synonym ('p,'c) pred_val = "'p ⇒ 'c list set"


fun eval_id :: "('x,'c) id ⇒ ('x,'c) var_val ⇒ 'c" ("⟦_⟧id") where
  "⟦Var x⟧id σ = σ x"
| "⟦Cst c⟧id σ = c"

fun eval_ids :: "('x,'c) id list ⇒ ('x,'c) var_val ⇒ 'c list" ("⟦_⟧ids") where
  "⟦ids⟧ids σ = map (λa. ⟦a⟧id σ) ids"

fun meaning_rh :: "('p,'x,'c) rh ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧rh") where
  "⟦a = a'⟧rh ϱ σ ⟷ ⟦a⟧id σ = ⟦a'⟧id σ"
| "⟦a ≠ a'⟧rh ϱ σ ⟷ ⟦a⟧id σ  ≠ ⟦a'⟧id σ"
| "⟦+ p ids⟧rh ϱ σ ⟷ ⟦ids⟧ids σ ∈ ϱ p"
| "⟦¬ p ids⟧rh ϱ σ ⟷ ¬ ⟦ids⟧ids σ ∈ ϱ p"

fun meaning_rhs :: "('p,'x,'c) rh list ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧rhs") where
  "⟦rhs⟧rhs ϱ σ ⟷ (∀rh ∈ set rhs. ⟦rh⟧rh ϱ σ)"

fun meaning_lh :: "('p,'x,'c) lh ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧lh") where
  "⟦(p,ids)⟧lh ϱ σ ⟷ ⟦ids⟧ids σ ∈ ϱ p"

fun meaning_cls :: "('p,'x,'c) clause ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧cls") where
  "⟦Cls p ids rhs⟧cls ϱ σ ⟷ (⟦rhs⟧rhs ϱ σ ⟶ ⟦(p,ids)⟧lh ϱ σ)"

fun solves_lh :: "('p,'c) pred_val ⇒ ('p,'x,'c) lh ⇒ bool" (infix "⊨lh" 91) where
  "ϱ ⊨lh (p,ids) ⟷ (∀σ. ⟦(p,ids)⟧lh ϱ σ)"

fun solves_rh :: "('p,'c) pred_val ⇒ ('p,'x,'c) rh ⇒ bool" (infix "⊨rh" 91) where
  "ϱ ⊨rh rh ⟷ (∀σ. ⟦rh⟧rh ϱ σ)"
```

definition solves_cls :: "('p,'c) pred_val ⇒ ('p,'x,'c) clause ⇒ bool" (**infix** "$\models_{cls}$" 91) **where**
"$\varrho \models_{cls}$ c ⟷ (∀σ. ⟦c⟧$_{cls}$ ϱ σ)"

definition solves_program :: "('p,'c) pred_val ⇒ ('p,'x,'c) dl_program ⇒ bool" (**infix** "$\models_{dl}$" 91) **where**
"$\varrho \models_{dl}$ dl ⟷ (∀c ∈ dl. $\varrho \models_{cls}$ c)"

3 Substitutions

type_synonym ('x,'c) subst = "'x ⇒ ('x,'c) id"

fun subst_id :: "('x,'c) id ⇒ ('x,'c) subst ⇒ ('x,'c) id" (**infix** "$\cdot_{id}$" 70) **where**
"(Var x) $\cdot_{id}$ η = η x"
| "(Cst e) $\cdot_{id}$ η = (Cst e)"

fun subst_ids :: "('x,'c) id list ⇒ ('x,'c) subst ⇒ ('x,'c) id list" (**infix** "$\cdot_{ids}$" 50) **where**
"ids $\cdot_{ids}$ η = map (λa. a $\cdot_{id}$ η) ids"

fun subst_rh :: "('p,'x,'c) rh ⇒ ('x,'c) subst ⇒ ('p,'x,'c) rh" (**infix** "$\cdot_{rh}$" 50) **where**
"(a = a') $\cdot_{rh}$ η = (a $\cdot_{id}$ η = a' $\cdot_{id}$ η)"
| "(a ≠ a') $\cdot_{rh}$ η = (a $\cdot_{id}$ η ≠ a' $\cdot_{id}$ η)"
| "(+ p ids) $\cdot_{rh}$ η = (+ p (ids $\cdot_{ids}$ η))"
| "(¬ p ids) $\cdot_{rh}$ η = (¬ p (ids $\cdot_{ids}$ η))"

fun subst_rhs :: "('p,'x,'c) rh list ⇒ ('x,'c) subst ⇒ ('p,'x,'c) rh list" (**infix** "$\cdot_{rhs}$" 50) **where**
"rhs $\cdot_{rhs}$ η = map (λa. a $\cdot_{rh}$ η) rhs"

fun subst_lh :: "('p,'x,'c) lh ⇒ ('x,'c) subst ⇒ ('p,'x,'c) lh" (**infix** "$\cdot_{lh}$" 50) **where**
"(p,ids) $\cdot_{lh}$ η = (p, ids $\cdot_{ids}$ η)"

fun subst_cls :: "('p,'x,'c) clause ⇒ ('x,'c) subst ⇒ ('p,'x,'c) clause" (**infix** "$\cdot_{cls}$" 50) **where**
"(Cls p ids rhs) $\cdot_{cls}$ η = Cls p (ids $\cdot_{ids}$ η) (rhs $\cdot_{rhs}$ η)"

definition compose :: "('x,'c) subst ⇒ ('x,'c) var_val ⇒ ('x,'c) var_val" (**infix** "$\circ_{sv}$" 50) **where**
"(η $\circ_{sv}$ σ) x = ⟦(η x)⟧$_{id}$ σ"

4 Substituting variable valuations

fun substv_id :: "('x,'c) id ⇒ ('x,'c) var_val ⇒ ('x,'c) id" (**infix** "$\cdot_{vid}$" 70) **where**
"(Var x) $\cdot_{vid}$ σ = Cst (σ x)"
| "(Cst e) $\cdot_{vid}$ σ = (Cst e)"

fun substv_ids :: "('x,'c) id list ⇒ ('x,'c) var_val ⇒ ('x,'c) id list" (**infix** "$\cdot_{vids}$" 50) **where**
"ids $\cdot_{vids}$ σ = map (λa. a $\cdot_{vid}$ σ) ids"

fun substv_rh :: "('p,'x,'c) rh ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) rh" (**infix** "$\cdot_{vrh}$" 50) **where**
"(a = a') $\cdot_{vrh}$ σ = (a $\cdot_{vid}$ σ = a' $\cdot_{vid}$ σ)"
| "(a ≠ a') $\cdot_{vrh}$ σ = (a $\cdot_{vid}$ σ ≠ a' $\cdot_{vid}$ σ)"
| "(+ p ids) $\cdot_{vrh}$ σ = (+ p (ids $\cdot_{vids}$ σ))"
| "(¬ p ids) $\cdot_{vrh}$ σ = (¬ p (ids $\cdot_{vids}$ σ))"

definition substv_rhs :: "('p,'x,'c) rh list ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) rh list" (**infix** "$\cdot_{vrhs}$" 50) **where**
"rhs $\cdot_{vrhs}$ σ = map (λa. a $\cdot_{vrh}$ σ) rhs"

fun substv_lh :: "('p,'x,'c) lh ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) lh" (**infix** "$\cdot_{vlh}$" 50) **where**
"(p,ids) $\cdot_{vlh}$ σ = (p, ids $\cdot_{vids}$ σ)"

fun substv_cls :: "('p,'x,'c) clause ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) clause" (**infix** "$\cdot_{vcls}$" 50) **where**
"(Cls p ids rhs) $\cdot_{vcls}$ σ = Cls p (ids $\cdot_{vids}$ σ) (rhs $\cdot_{vrhs}$ σ)"

5 Datalog lemmas

5.1 Variable valuations

```
lemma substv_id_is_Cst_eval_id:
  "a' ·vid σ' = Cst (⟦a'⟧id σ')"
  by (cases a') auto

lemma eval_id_is_substv_id:
  "⟦a'⟧id σ' = ⟦a⟧id σ ⟷ (a' ·vid σ') = (a ·vid σ)"
  by (cases a'; cases a) auto

lemma eval_ids_is_substv_ids:
  "⟦ids'⟧ids σ' = ⟦ids⟧ids σ ⟷ (ids' ·vids σ') = (ids ·vids σ)"
proof (induction ids' arbitrary: ids)
  case Nil
  then show ?case
    by auto
next
  case (Cons a ids')
  note Cons_outer = Cons
  show ?case
  proof (cases ids)
    case Nil
    then show ?thesis
      using Cons_outer by auto
  next
    case (Cons a list)
    then show ?thesis
      using eval_id_is_substv_id Cons_outer by force
  qed
qed

lemma solves_rh_substv_rh_if_meaning_rh:
  assumes "⟦a⟧rh ϱ σ"
  shows "ϱ ⊨rh (a ·vrh σ)"
using assms proof (induction a)
  case (Eql a a')
  then show ?case
    by (auto simp add: eval_id_is_substv_id)
next
  case (Neql a a')
  then show ?case
    using eval_id_is_substv_id by (auto simp add: substv_id_is_Cst_eval_id)
next
  case (PosLit p ids)
  then show ?case
    by (auto simp add: substv_id_is_Cst_eval_id comp_def eval_id_is_substv_id)
next
  case (NegLit p ids)
  then show ?case
    by (auto simp add: substv_id_is_Cst_eval_id comp_def eval_id_is_substv_id)
qed

lemma solves_lh_substv_lh_if_meaning_lh:
  assumes "⟦a⟧lh ϱ σ"
  shows "ϱ ⊨lh (a ·vlh σ)"
proof -
  obtain p ids where a_split: "a = (p, ids)"
    by (cases a)
  show ?thesis
    using assms unfolding a_split
    by (auto simp add: substv_id_is_Cst_eval_id comp_def eval_id_is_substv_id)
qed
```

5.2 Solve lhs

```
lemma solves_lh_iff_solves_lh: "ϱ ⊨cls Cls p ids [] ⟷ ϱ ⊨rh (+ p ids)"
  using solves_cls_def by force

lemma solves_lh_lh:
  assumes "ϱ ⊨cls Cls p args []"
  shows "ϱ ⊨lh (p, args)"
  using assms unfolding solves_cls_def by auto

lemmas solve_lhs = solves_lh_iff_solves_lh solves_lh_lh
```

5.3 Propositional inferences

5.3.1 Of last right hand

```
lemma prop_inf_last_from_cls_rh_to_cls:
  assumes "ϱ ⊨cls Cls p ids (rhs @ [rh])"
  assumes "ϱ ⊨rh rh"
  shows "ϱ ⊨cls Cls p ids rhs"
  using assms unfolding solves_cls_def by auto

lemma prop_inf_last_from_cls_lh_to_cls:
  assumes "ϱ ⊨cls Cls p ids (rhs @ [+ p ids])"
  assumes "ϱ ⊨lh (p, ids)"
  shows "ϱ ⊨cls Cls p ids rhs"
  using assms by (force simp add: solves_cls_def)

lemmas prop_inf_last = prop_inf_last_from_cls_rh_to_cls prop_inf_last_from_cls_lh_to_cls
```

5.3.2 Of only right hand

```
lemma modus_ponens_rh:
  assumes "ϱ ⊨cls Cls p ids [+ p' ids']"
  assumes "ϱ ⊨lh (p', ids')"
  shows "ϱ ⊨lh (p, ids)"
  using assms
proof -
  from assms(2) have "∀σ. ⟦+ p' ids'⟧rh ϱ σ"
    by fastforce
  then have "ϱ ⊨cls Cls p ids []"
    using assms(1) self_append_conv2 solves_rh.elims(3) prop_inf_last_from_cls_rh_to_cls by metis
  then show "ϱ ⊨lh (p, ids)"
    by (meson solves_lh_lh)
qed

lemma prop_inf_only_from_cls_cls_to_cls:
  assumes "ϱ ⊨cls Cls p ids [+ p' ids']"
  assumes "ϱ ⊨cls Cls p' ids' []"
  shows "ϱ ⊨cls Cls p ids []"
  by (metis append_self_conv2 assms prop_inf_last_from_cls_rh_to_cls solves_lh_iff_solves_lh)

lemmas prop_inf_only = modus_ponens_rh prop_inf_only_from_cls_cls_to_cls
```

5.3.3 Of all right hands

```
lemma modus_ponens:
  assumes "ϱ ⊨cls Cls p ids rhs"
  assumes "∀rh∈set rhs. ϱ ⊨rh rh"
  shows "ϱ ⊨lh (p, ids)"
  using assms unfolding solves_cls_def meaning_lh.simps by force

lemmas prop_inf_all = modus_ponens

lemmas prop_inf = prop_inf_last prop_inf_only prop_inf_all
```

5.4 Substitution

```
lemma substitution_lemma_id: "⟦a⟧id (η ∘sv σ) = ⟦a ·id η⟧id σ"
  by (cases a) (auto simp add: compose_def)

lemma substitution_lemma_ids: "⟦ids⟧ids (η ∘sv σ) = ⟦ids ·ids η⟧ids σ"
  using substitution_lemma_id by auto

lemma substitution_lemma_lh: "⟦(p, ids)⟧lh ϱ (η ∘sv σ) ⟷ ⟦(p, ids ·ids η )⟧lh ϱ σ"
  by (metis meaning_lh.simps substitution_lemma_ids)

lemma substitution_lemma_rh:"⟦rh⟧rh ϱ (η ∘sv σ) ⟷ ⟦rh ·rh η⟧rh ϱ σ"
proof (induction rh)
  case (Eql a a')
  then show ?case
    by (simp add: substitution_lemma_id)
next
  case (Neql a a')
  then show ?case
    by (simp add: substitution_lemma_id)
next
  case (PosLit p ids)
  then show ?case
    using substitution_lemma_lh by fastforce
next
  case (NegLit p ids)
  then show ?case
    using substitution_lemma_lh by fastforce
qed

lemma substitution_lemma_rhs: "⟦rhs⟧rhs ϱ (η ∘sv σ) ⟷ ⟦rhs ·rhs η⟧rhs ϱ σ"
  by (simp add: substitution_lemma_rh)

lemma substitution_lemma_cls:
  "⟦c⟧cls ϱ (η ∘sv σ) = ⟦c ·cls η⟧cls ϱ σ"
proof (induction c)
  case (Cls p ids rhs)
  have "⟦rhs⟧rhs ϱ (η ∘sv σ) = ⟦rhs ·rhs η⟧rhs ϱ σ"
    using substitution_lemma_rhs by blast
  moreover
  have  "⟦(p, ids)⟧lh ϱ (η ∘sv σ) = ⟦(p, ids ·ids η)⟧lh ϱ σ"
    using substitution_lemma_lh by metis
  ultimately
  show ?case
    unfolding meaning_cls.simps by auto
qed

lemma substitution_lemma:
  assumes "ϱ ⊨cls c"
  shows "ϱ ⊨cls (c ·cls (η::('x,'c) subst))"
proof -
  show ?thesis
    unfolding solves_cls_def
  proof
    fix σ :: "'x ⇒ 'c"
    from assms have "⟦c⟧cls ϱ (η ∘sv σ)"
      using solves_cls_def by auto
    then show "⟦c ·cls η ⟧cls ϱ σ"
      using substitution_lemma_cls by blast
  qed
qed
```

6 Stratification and solutions to stratified datalog programs

```
type_synonym 'p strat = "'p ⇒ nat"

fun rnk :: "'p strat ⇒ ('p,'x,'c) rh ⇒ nat" where
  "rnk s (a = a') = 0"
| "rnk s (a ≠ a') = 0"
| "rnk s (+ p ids) = s p"
| "rnk s (¬ p ids) = 1 + s p"

fun strat_wf_cls :: "'p strat ⇒ ('p,'x,'c) clause ⇒ bool" where
  "strat_wf_cls s (Cls p ids rhs) ⟷ (∀rh ∈ set rhs. s p ≥ rnk s rh)"

definition strat_wf :: "'p strat ⇒ ('p,'x,'c) dl_program ⇒ bool" where
  "strat_wf s dl ⟷ (∀c ∈ dl. strat_wf_cls s c)"

definition max_stratum :: "'p strat ⇒ ('p,'x,'c) dl_program ⇒ nat" where
  "max_stratum s dl = Max {s p | p ids rhs. Cls p ids rhs ∈ dl}"

fun pred_val_lte_stratum :: "('p,'c) pred_val ⇒ 'p strat ⇒ nat ⇒ ('p,'c) pred_val"
  ("_ ≤≤≤_≤≤≤ _" 0) where
  "(ϱ ≤≤≤s≤≤≤ n) p = (if s p ≤ n then ϱ p else {})"

fun dl_program_lte_stratum :: "('p,'x,'c) dl_program ⇒ 'p strat ⇒ nat ⇒ ('p,'x,'c) dl_program"
  ("_ ≤≤_≤≤ _" 0) where
  "(dl ≤≤s≤≤ n) = {(Cls p ids rhs)| p ids rhs . (Cls p ids rhs) ∈ dl ∧ s p ≤ n}"

fun dl_program_on_stratum :: "('p,'x,'c) dl_program ⇒ 'p strat ⇒ nat ⇒ ('p,'x,'c) dl_program"
  ("_ ==_== _" 0) where
  "(dl ==s== n) = {(Cls p ids rhs)| p ids rhs . (Cls p ids rhs) ∈ dl ∧ s p = n}"
```

— The ordering on predicate valuations from Flemming Nielson and Hanne Riis Nielson. Program analysis (an appetizer). CoRR,abs/2012.10086, 2020.

```
definition lt :: "('p,'c) pred_val ⇒ 'p strat ⇒ ('p,'c) pred_val ⇒ bool" ("_ ⊏_⊏ _") where
  "(ϱ ⊏s⊏ ϱ') ⟷ (∃p. ϱ p ⊂ ϱ' p ∧
                     (∀p'. s p' = s p ⟶ ϱ p' ⊆ ϱ' p') ∧
                     (∀p'. s p' < s p ⟶ ϱ p' = ϱ' p'))"
```

— The ordering on predicate valuations from Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, Foundations of Deductive Databases and Logic Programming, pages 193216. Morgan Kaufmann, 1988.

```
definition lt_prz :: "('p,'c) pred_val ⇒ 'p strat ⇒ ('p,'c) pred_val ⇒ bool" ("_ ⊏_⊏_prz _") where
  "(ϱM ⊏s⊏prz ϱN) ⟷ ϱN ≠ ϱM ∧
                     (∀pA cA. cA ∈ ϱN pA - ϱM pA ⟶ (∃pB cB. cB ∈ ϱM pB - ϱN pB ∧ s pA > s pB))"

definition lte :: "('p,'c) pred_val ⇒ 'p strat ⇒ ('p,'c) pred_val ⇒ bool" ("_ ⊑_⊑ _") where
  "(ϱ ⊑s⊑ ϱ') ⟷ ϱ = ϱ' ∨ (ϱ ⊏s⊏ ϱ')"

definition least_solution :: "('p,'c) pred_val ⇒ ('p,'x,'c) dl_program ⇒ 'p strat ⇒ bool"
  ("_ ⊨lst") where
  "ϱ ⊨lst dl s ⟷ ϱ ⊨dl dl ∧ (∀ϱ'. ϱ' ⊨dl dl ⟶ ϱ ⊑s⊑ ϱ')"

definition minimal_solution :: "('p,'c) pred_val ⇒ ('p,'x,'c) dl_program ⇒ 'p strat ⇒ bool"
  ("_ ⊨min") where
  "ϱ ⊨min dl s ⟷ ϱ ⊨dl dl ∧ (∄ϱ'. ϱ' ⊨dl dl ∧ ϱ' ⊏s⊏ ϱ)"

lemma lte_def2:
  "(ϱ ⊑s⊑ ϱ') ⟷ ϱ ≠ ϱ' ⟶ (ϱ ⊏s⊏ ϱ')"
  unfolding lte_def by auto
```

6.1 Solving lower strata

```
lemma strat_wf_lte_if_strat_wf_lte:
  assumes "n > m"
```

```
  assumes "strat_wf s (dl ≤≤s≤≤ n)"
  shows "strat_wf s (dl ≤≤s≤≤ m)"
  using assms unfolding strat_wf_def by fastforce

lemma strat_wf_lte_if_strat_wf:
  assumes "strat_wf s dl"
  shows "strat_wf s (dl ≤≤s≤≤ n)"
  using assms unfolding strat_wf_def by auto

lemma meaning_lte_m_iff_meaning_rh:
  assumes "rnk s rh ≤ n"
  shows "⟦rh⟧_rh (ϱ ≤≤≤s≤≤≤ n) σ ⟷ ⟦rh⟧_rh ϱ σ"
  using assms equals0D meaning_rh.elims(3) pred_val_lte_stratum.simps by fastforce

lemma meaning_lte_m_iff_meaning_lh:
  assumes "s p ≤ m"
  shows "⟦(p, ids)⟧_lh (ϱ ≤≤≤s≤≤≤ m) σ ⟷ ⟦(p, ids)⟧_lh ϱ σ"
  using assms by auto

lemma meaning_lte_m_iff_meaning_cls:
  assumes "strat_wf_cls s (Cls p ids rhs)"
  assumes "s p ≤ m"
  shows "⟦Cls p ids rhs⟧_cls (ϱ ≤≤≤s≤≤≤ m) σ ⟷ ⟦Cls p ids rhs⟧_cls ϱ σ"
proof -
  have p_leq_m: "s p ≤ m"
    using assms by fastforce
  have rh_leq_m: "∀rh ∈ set rhs. rnk s rh ≤ m"
    using assms assms(2) dual_order.trans by (metis (no_types, lifting) p_leq_m strat_wf_cls.simps)

  show "⟦Cls p ids rhs⟧_cls (ϱ ≤≤≤s≤≤≤ m) σ ⟷ ⟦Cls p ids rhs⟧_cls ϱ σ"
    using meaning_lte_m_iff_meaning_rh[of s _ m ϱ σ] p_leq_m rh_leq_m assms(2) by force
qed

lemma solves_lte_m_iff_solves_cls:
  assumes "strat_wf_cls s (Cls p ids rhs)"
  assumes "s p ≤ m"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨_cls Cls p ids rhs ⟷ ϱ ⊨_cls Cls p ids rhs"
  by (meson assms meaning_lte_m_iff_meaning_cls solves_cls_def)

lemma downward_lte_solves:
  assumes "n > m"
  assumes "ϱ ⊨_dl (dl ≤≤s≤≤ n)"
  assumes "strat_wf s dl"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨_dl (dl ≤≤s≤≤ m)"
  unfolding solves_program_def
proof
  fix c
  assume a: "c ∈ (dl ≤≤s≤≤ m)"
  obtain p ids rhs where c_split: "c = Cls p ids rhs"
    by (cases c) auto

  have "m < n"
    using assms(1) by blast
  moreover
  have "strat_wf_cls s (Cls p ids rhs)"
    using a assms(3) c_split strat_wf_lte_if_strat_wf strat_wf_def by blast
  moreover
  have "s p ≤ m"
    using a c_split by force
  moreover
  have "ϱ ⊨_cls Cls p ids rhs"
    using c_split assms a unfolding solves_program_def by force
  ultimately
```

```
  show "(ϱ ≤≤≤s≤≤≤ m) ⊨cls c"
    using c_split by (simp add: solves_lte_m_iff_solves_cls)
qed

lemma downward_solves:
  assumes "ϱ ⊨dl dl"
  assumes "strat_wf s dl"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨dl (dl ≤≤s≤≤ m)"
  unfolding solves_program_def
proof
  fix c
  assume a: "c ∈ (dl ≤≤s≤≤ m)"
  then obtain p ids rhs where c_def: "c = Cls p ids rhs"
    by (cases c) auto

  have "ϱ ⊨cls c"
    using ‹c ∈ (dl ≤≤s≤≤ m)› assms(1) solves_program_def by auto

  have "(ϱ ≤≤≤s≤≤≤ m) ⊨cls Cls p ids rhs"
    using ‹ϱ ⊨cls c› a assms(2) c_def solves_lte_m_iff_solves_cls strat_wf_def by fastforce
  then show "(ϱ ≤≤≤s≤≤≤ m) ⊨cls c"
    using c_def by auto
qed
```

6.2 Least solutions

6.2.1 Existence of least solutions

```
definition Inter' :: "('a ⇒ 'b set) set ⇒ 'a ⇒ 'b set" ("⋂") where
  "(⋂ ϱs) = (λp. ⋂{ϱ p | ϱ. ϱ ∈ ϱs})"

lemma Inter'_def2: "(⋂ ϱs) = (λp. ⋂{m. ∃ϱ ∈ ϱs. m = ϱ p})"
  unfolding Inter'_def by auto

lemma member_Inter':
  assumes "∀p ∈ ps. y ∈ p x"
  shows "y ∈ (⋂ ps) x"
  using assms unfolding Inter'_def by auto

lemma member_if_member_Inter':
  assumes "y ∈ (⋂ ps) x"
  assumes "p ∈ ps"
  shows "y ∈ p x"
  using assms unfolding Inter'_def by auto

lemma member_Inter'_iff:
  "(∀p ∈ ps. y ∈ p x) ⟷ y ∈ (⋂ ps) x"
  unfolding Inter'_def by auto

lemma intersection_valuation_subset_valuation:
  assumes "P ϱ"
  shows "⋂ {ϱ'. P  ϱ'} p ⊆ ϱ p"
  using assms unfolding Inter'_def by auto

fun solve_pg where
  "solve_pg s dl 0 = (⋂ {ϱ. ϱ ⊨dl (dl ==s== 0)})"
| "solve_pg s dl (Suc n) = (⋂ {ϱ. ϱ ⊨dl (dl ==s== Suc n) ∧ (ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n})"

definition least_rank_p_st :: "('p ⇒ bool) ⇒ 'p ⇒ ('p ⇒ nat) ⇒ bool" where
  "least_rank_p_st P p s ⟷ P p ∧ (∀p'. P p' ⟶ s p ≤ s p')"

lemma least_rank_p_st_exists:
  assumes "P p"
  shows "∃p''. least_rank_p_st P p'' s"
```

```
  using assms
proof (induction "s p" arbitrary: p rule: less_induct)
  case less
  then show ?case
  proof (cases "∃p''. s p'' < s p ∧ P p''")
    case True
    then show ?thesis
      using less by auto
  next
    case False
    then show ?thesis
      by (metis least_rank_p_st_def less.prems linorder_not_le)
  qed
qed

lemma below_least_rank_p_st:
  assumes "least_rank_p_st P p'' s"
  assumes "s p < s p'"
  shows "¬P p"
  using assms
proof (induction "s p'")
  case 0
  then show ?case by auto
next
  case (Suc n)
  then show ?case
    by (metis least_rank_p_st_def linorder_not_le)
qed

lemma lt_if_lt_prz:
  assumes "ϱ_M ⊏s⊏_prz ϱ_N"
  shows "ϱ_N ⊏s⊏ ϱ_M"
proof -
  from assms have unf:
    "ϱ_N ≠ ϱ_M"
    "∀p_A c_A. c_A ∈ ϱ_N p_A - ϱ_M p_A ⟶ (∃p_B c_B. c_B ∈ ϱ_M p_B - ϱ_N p_B ∧ s p_B < s p_A)"
    unfolding lt_prz_def by auto
  then have "∃p. ϱ_N p ≠ ϱ_M p"
    by auto
  then have "∃p. least_rank_p_st (λp. ϱ_N p ≠ ϱ_M p) p s"
    using least_rank_p_st_exists by smt
  then obtain p where "least_rank_p_st (λp. ϱ_N p ≠ ϱ_M p) p s"
    by auto
  have "ϱ_N p ≠ ϱ_M p"
    by (metis (mono_tags, lifting) ‹least_rank_p_st (λp. ϱ_N p ≠ ϱ_M p) p s› least_rank_p_st_def)
  then have "(∃c. c ∈ ϱ_M p ∧ c ∉ ϱ_N p) ∨ (∃c. c ∈ ϱ_N p ∧ c ∉ ϱ_M p)"
    by auto
  then show "ϱ_N ⊏s⊏ ϱ_M"
  proof
    assume "∃c. c ∈ ϱ_M p ∧ c ∉ ϱ_N p"
    then obtain c where "c ∈ ϱ_M p" "c ∉ ϱ_N p"
      by auto
    have "(∀p'. s p' < s p ⟶ ϱ_N p' = ϱ_M p')"
      using ‹least_rank_p_st (λp. ϱ_N p ≠ ϱ_M p) p s› below_least_rank_p_st by fastforce
    have "(∀p'. s p' = s p ⟶ ϱ_N p' ⊆ ϱ_M p')"
      using ‹∀p'. s p' < s p ⟶ ϱ_N p' = ϱ_M p'› unf by auto
    then show ?thesis
      unfolding lt_def using ‹∀p'. s p' < s p ⟶ ϱ_N p' = ϱ_M p'› ‹ϱ_N p ≠ ϱ_M p› by blast
  next
    assume "∃c. c ∈ ϱ_N p ∧ c ∉ ϱ_M p"
    then obtain c where "c ∈ ϱ_N p ∧ c ∉ ϱ_M p"
      by auto
    then have "∃p_B c_B. c_B ∈ ϱ_M p_B - ϱ_N p_B ∧ s p_B < s p"
```

```
    using unf by auto
  then obtain p_B c_B where "c_B ∈ ϱ_M p_B - ϱ_N p_B" "s p_B < s p"
    by auto
  then have "False"
    using ‹least_rank_p_st (λp. ϱ_N p ≠ ϱ_M p) p s› below_least_rank_p_st by fastforce
  then show ?thesis
    by auto
 qed
qed

lemma lt_prz_if_lt_if:
 assumes "ϱ_M ⊏s⊏ ϱ_N"
 shows "ϱ_N ⊏s⊏_prz ϱ_M"
proof -
 have "ϱ_M ≠ ϱ_N"
   using assms lt_def by fastforce
 moreover
 have "(∀p_A c_A. c_A ∈ ϱ_M p_A - ϱ_N p_A ⟶ (∃p_B c_B. c_B ∈ ϱ_N p_B - ϱ_M p_B ∧ s p_B < s p_A))"
 proof (rule, rule, rule)
   fix p_A c_A
   assume "c_A ∈ ϱ_M p_A - ϱ_N p_A"
   then show "∃p_B c_B. c_B ∈ ϱ_N p_B - ϱ_M p_B ∧ s p_B < s p_A"
     by (smt (verit) DiffD1 DiffD2 antisym_conv3 assms lt_def psubset_imp_ex_mem subsetD)
 qed
 ultimately
 show "ϱ_N ⊏s⊏_prz ϱ_M"
   unfolding lt_prz_def by auto
qed

lemma lt_prz_iff_lt:
 "ϱ_M ⊏s⊏ ϱ_N ⟷ ϱ_N ⊏s⊏_prz ϱ_M"
 using lt_if_lt_prz lt_prz_if_lt_if by blast

lemma solves_leq:
 assumes "ϱ ⊨_dl (dl ≤≤s≤≤ m)"
 assumes "n ≤ m"
 shows "ϱ ⊨_dl (dl ≤≤s≤≤ n)"
 using assms unfolding solves_program_def by auto

lemma solves_Suc:
 assumes "ϱ ⊨_dl (dl ≤≤s≤≤ Suc n)"
 shows "ϱ ⊨_dl (dl ≤≤s≤≤ n)"
 by (meson assms lessI less_imp_le_nat solves_leq)

lemma solve_pg_0_subset:
 assumes "ϱ ⊨_dl (dl ≤≤s≤≤ 0)"
 shows "(solve_pg s dl 0) p ⊆ ϱ p"
 using assms by (auto simp add: Inter'_def)

lemma solve_pg_Suc_subset:
 assumes "ϱ ⊨_dl (dl ==s== Suc n)"
 assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
 shows "(solve_pg s dl (Suc n)) p ⊆ ϱ p"
 using assms by (force simp add: Inter'_def2)

lemma solve_pg_0_empty:
 assumes "s p > 0"
 shows "(solve_pg s dl 0) p = {}"
proof -
 define ϱ'' :: "'a ⇒ 'b list set" where "ϱ'' = (solve_pg s dl 0)"
 define ϱ' :: "'a ⇒ 'b list set" where "ϱ' = (λp. if s p = 0 then UNIV else {})"
 have "ϱ' ⊨_dl (dl ==s== 0)"
   unfolding solves_program_def solves_cls_def
```

```
  proof (rule, rule)
    fix c σ
    assume c_dl0: "c ∈ (dl ==s== 0)"
    obtain p' ids rhs where c_split: "c = Cls p' ids rhs"
      by (cases c) auto
    then have sp'0: "s p' = 0"
      using c_dl0 by auto
    have "⟦Cls p' ids rhs⟧cls ϱ' σ"
      unfolding meaning_cls.simps
    proof (rule)
      assume "⟦rhs⟧rhs ϱ' σ"
      show "⟦(p', ids)⟧lh ϱ' σ"
        using ϱ'_def sp'0 by force
    qed
    then show "⟦c⟧cls ϱ' σ"
      unfolding c_split by auto
  qed
  have "ϱ'' p ⊆ ϱ' p"
    using ‹ϱ' ⊨dl (dl ==s== 0)› ϱ''_def solve_pg_0_subset by fastforce
  then have "ϱ'' p = {}"
    unfolding ϱ'_def using assms(1) by simp
  then show ?thesis
    unfolding ϱ''_def by auto
qed

lemma solve_pg_above_empty:
  assumes "s p > n"
  shows "(solve_pg s dl n) p = {}"
  using assms proof (induction n arbitrary: p)
  case 0
  then show ?case
    using solve_pg_0_empty by metis
next
  case (Suc n)
  define ϱ'' :: "'a ⇒ 'b list set" where
    "ϱ'' = (solve_pg s dl (Suc n))"
  define ϱ' :: "'a ⇒ 'b list set" where
    "ϱ' = (λp. if s p < Suc n then (solve_pg s dl n) p else if s p = Suc n then UNIV else {})"

  have ϱ'_solves: "ϱ' ⊨dl (dl ==s== Suc n)"
    unfolding solves_program_def solves_cls_def
  proof (rule, rule)
    fix c σ
    assume c_dlSucn: "c ∈ (dl ==s== Suc n)"
    obtain p' ids rhs where c_split: "c = Cls p' ids rhs"
      by (cases c) auto
    then have sp'Sucn: "s p' = Suc n"
      using c_dlSucn by auto
    have "⟦Cls p' ids rhs⟧cls ϱ' σ"
      unfolding meaning_cls.simps
    proof (rule)
      assume "⟦rhs⟧rhs ϱ' σ"
      show "⟦(p', ids)⟧lh ϱ' σ"
        using ϱ'_def sp'Sucn by auto[]
    qed
    then show "⟦c⟧cls ϱ' σ"
      unfolding c_split by auto
  qed
  have "∀p. (ϱ' ≤≤≤s≤≤≤ n) p = (solve_pg s dl n) p"
    using Suc by (auto simp add: ϱ'_def)
  then have "ϱ'' p ⊆ ϱ' p"
    using solve_pg_Suc_subset[of ϱ' dl s n  p] ϱ'_solves ϱ''_def by force
  then have "ϱ'' p = {}"
```

```
    unfolding ϱ'_def using assms(1) Suc by simp
  then show ?case
    unfolding ϱ''_def by auto
qed

lemma exi_sol_n:
  "∃ϱ'. ϱ' ⊨dl (dl ==s== Suc m) ∧ (ϱ' ≤≤≤s≤≤≤ m) = solve_pg s dl m"
proof -
  define ϱ' where
    "ϱ' = (λp. (if s p < Suc m then (solve_pg s dl m) p else if s p = Suc m then UNIV else {}))"

  have "ϱ' ⊨dl (dl ==s== Suc m)"
    unfolding ϱ'_def solves_cls_def solves_program_def by fastforce
  moreover
  have "∀p. (ϱ' ≤≤≤s≤≤≤ m) p = solve_pg s dl m p"
    unfolding ϱ'_def using solve_pg_above_empty[of m s _ dl] by auto
  ultimately
  show ?thesis
    by force
qed

lemma solve_pg_agree_above:
  assumes "s p ≤ m"
  shows "(solve_pg s dl m) p = (solve_pg s dl (s p)) p"
  using assms
proof (induction m)
  case 0
  then show ?case
    by force
next
  case (Suc m)
  show ?case
  proof (cases "s p = Suc m")
    case True
    then show ?thesis by auto
  next
    case False
    then have s_p: "s p ≤ m"
      using Suc by auto
    then have solve_pg_sp_m: "(solve_pg s dl (s p)) p = (solve_pg s dl m) p"
      using Suc by auto
    have ϱ'_solve_pg: "∀ϱ'. ϱ' ⊨dl (dl ==s== Suc m) ⟶
                           (ϱ' ≤≤≤s≤≤≤ m) = solve_pg s dl m ⟶
                           ϱ' p = solve_pg s dl m p"
      by (metis pred_val_lte_stratum.simps s_p)
    have "⋂ {ϱ'. ϱ' ⊨dl (dl ==s== Suc m) ∧ (ϱ' ≤≤≤s≤≤≤ m) = solve_pg s dl m} p =
          solve_pg s dl (s p) p"
    proof (rule; rule)
      fix x
      assume "x ∈ ⋂ {ϱ'. ϱ' ⊨dl (dl ==s== Suc m) ∧ (ϱ' ≤≤≤s≤≤≤ m) = solve_pg s dl m} p"
      then show "x ∈ solve_pg s dl (s p) p"
        by (metis (mono_tags) CollectI ϱ'_solve_pg exi_sol_n member_if_member_Inter' solve_pg_sp_m)
    next
      fix x
      assume "x ∈ solve_pg s dl (s p) p"
      then show "x ∈ ⋂ {ϱ'. ϱ' ⊨dl (dl ==s== Suc m) ∧ (ϱ' ≤≤≤s≤≤≤ m) = solve_pg s dl m} p"
        by (simp add: ϱ'_solve_pg member_Inter' solve_pg_sp_m)
    qed
    then show ?thesis
      by simp
  qed
qed
```

```
lemma solve_pg_two_agree_above:
  assumes "s p ≤ n"
  assumes "s p ≤ m"
  shows "(solve_pg s dl m) p = (solve_pg s dl n) p"
  using assms solve_pg_agree_above by metis

lemma pos_rhs_stratum_leq_clause_stratum:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "+ p' ids' ∈ set rhs"
  shows "s p' ≤ s p"
  using assms unfolding strat_wf_def by fastforce

lemma neg_rhs_stratum_less_clause_stratum:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "¬ p' ids' ∈ set rhs"
  shows "s p' < s p"
  using assms unfolding strat_wf_def by fastforce

lemma solve_pg_two_agree_above_on_rh:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "s p ≤ n"
  assumes "s p ≤ m"
  assumes "rh ∈ set rhs"
  shows "⟦rh⟧_rh (solve_pg s dl m) σ ⟷ ⟦rh⟧_rh (solve_pg s dl n) σ"
proof (cases rh)
  case (Eql a a')
  then show ?thesis
    using assms by auto
next
  case (Neql a a')
  then show ?thesis
    using assms by auto
next
  case (PosLit p' ids')
  then have "s p' ≤ m"
    using assms pos_rhs_stratum_leq_clause_stratum by fastforce
  moreover
  from PosLit have "s p' ≤ n"
    using assms pos_rhs_stratum_leq_clause_stratum by fastforce
  ultimately
  have "solve_pg s dl m p' = solve_pg s dl n p'"
    using solve_pg_two_agree_above[of s p' n m dl] by force
  then show ?thesis
    by (simp add: PosLit)
next
  case (NegLit p' ids)
  then have "s p' < m"
    using assms neg_rhs_stratum_less_clause_stratum by fastforce
  moreover
  from NegLit have "s p' < n"
    using assms neg_rhs_stratum_less_clause_stratum by fastforce
  ultimately
  have "solve_pg s dl m p' = solve_pg s dl n p'"
    using solve_pg_two_agree_above[of s p' n m dl] by force
  then show ?thesis
    by (simp add: NegLit)
qed

lemma solve_pg_two_agree_above_on_lh:
  assumes "s p ≤ n"
```

```
  assumes "s p ≤ m"
  shows "⟦(p,ids)⟧lh (solve_pg s dl m) σ ⟷ ⟦(p,ids)⟧lh (solve_pg s dl n) σ"
  by (metis assms meaning_lh.simps solve_pg_two_agree_above)

lemma solve_pg_two_agree_above_on_cls:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "s p ≤ n"
  assumes "s p ≤ m"
  shows "⟦Cls p ids rhs⟧cls (solve_pg s dl n) σ ⟷ ⟦Cls p ids rhs⟧cls (solve_pg s dl m) σ"
  using assms solve_pg_two_agree_above_on_rh solve_pg_two_agree_above_on_lh
  by (meson meaning_rhs.simps meaning_cls.simps)

lemma solve_pg_two_agree_above_on_cls_Suc:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "s p ≤ n"
  shows "⟦Cls p ids rhs⟧cls (solve_pg s dl (Suc n)) σ ⟷ ⟦Cls p ids rhs⟧cls (solve_pg s dl n) σ"
  using solve_pg_two_agree_above_on_cls[OF assms(1,2,3), of "Suc n" σ] assms(3) by auto

lemma stratum0_no_neg':
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "s p = 0"
  assumes "rh ∈ set rhs"
  shows "∄p' ids. rh = ¬ p' ids"
  by (metis One_nat_def add_eq_0_iff_both_eq_0 assms bot_nat_0.extremum_uniqueI
      bot_nat_0.not_eq_extremum rnk.simps(4) strat_wf_cls.simps strat_wf_def zero_less_Suc)

lemma stratumSuc_less_neg':
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ dl"
  assumes "s p = Suc n"
  assumes "¬ p' ids' ∈ set rhs"
  shows "s p' ≤ n"
  using assms unfolding strat_wf_def by force

lemma stratum0_no_neg:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ 0)"
  assumes "rh ∈ set rhs"
  shows "∄p' ids. rh = ¬ p ids"
  using assms stratum0_no_neg' by fastforce

lemma stratumSuc_less_neg:
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ Suc n)"
  assumes "¬ p' ids' ∈ set rhs"
  shows "s p' ≤ n"
  using assms neg_rhs_stratum_less_clause_stratum by fastforce

lemma all_meaning_rh_if_solve_pg_0:
  assumes "strat_wf s dl"
  assumes "⟦rh⟧rh (solve_pg s dl 0) σ"
  assumes "ϱ ⊨dl (dl ≤≤s≤≤ 0)"
  assumes "rh ∈ set rhs"
  assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ 0)"
  shows "⟦rh⟧rh ϱ σ"
proof (cases rh)
  case (Eql a1 a2)
  then show ?thesis
    using assms by auto
next
```

```
  case (Neql a1 a2)
  then show ?thesis
    using assms by auto
next
  case (PosLit p ids)
  then show ?thesis
    using assms meaning_rh.simps(3) solve_pg_0_subset by fastforce
next
  case (NegLit p ids)
  then show ?thesis
    using assms stratum0_no_neg' by fastforce
qed

lemma all_meaning_rh_if_solve_pg_Suc:
  assumes "strat_wf s dl"
  assumes "⟦rh⟧rh (solve_pg s dl (Suc n)) σ"
  assumes "ϱ ⊨dl (dl ==s== Suc n)"
  assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
  assumes "rh ∈ set rhs"
  assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ Suc n)"
  shows "⟦rh⟧rh ϱ σ"
proof (cases rh)
  case (Eql a a')
  then show ?thesis
    using assms(2) by auto
next
  case (Neql a a')
  then show ?thesis
    using assms(2) by force
next
  case (PosLit p' ids')
  then show ?thesis
    using assms solve_pg_Suc_subset by fastforce
next
  case (NegLit p' ids')
  then have "s p' ≤ n"
    using stratumSuc_less_neg[OF assms(1) assms(6), of p'] assms(5) by auto
  then have "⟦rh⟧rh (solve_pg s dl n) σ"
    by (metis NegLit assms(2) le_imp_less_Suc less_imp_le_nat meaning_rh.simps(4)
        solve_pg_two_agree_above)
  then have "⟦rh⟧rh (ϱ ≤≤≤s≤≤≤ n) σ"
    using assms(4) by presburger
  then show ?thesis
    by (simp add: NegLit ‹s p' ≤ n›)
qed

lemma solve_pg_0_if_all_meaning_lh:
  assumes "∀ϱ'. ϱ' ⊨dl (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧lh ϱ' σ"
  shows "⟦(p, ids)⟧lh (solve_pg s dl 0) σ"
  using assms by (auto simp add: Inter'_def)

lemma all_meaning_lh_if_solve_pg_0:
  assumes "⟦(p, ids)⟧lh (solve_pg s dl 0) σ"
  shows "∀ϱ'. ϱ' ⊨dl (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧lh ϱ' σ"
  using assms by (auto simp add: Inter'_def)

lemma solve_pg_0_iff_all_meaning_lh:
  "⟦(p, ids)⟧lh (solve_pg s dl 0) σ ⟷ (∀ϱ'. ϱ' ⊨dl (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧lh ϱ' σ)"
  using solve_pg_0_if_all_meaning_lh all_meaning_lh_if_solve_pg_0 by metis

lemma solve_pg_Suc_if_all_meaning_lh:
  assumes "∀ϱ'. ϱ' ⊨dl (dl ==s== Suc n) ⟶ (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶ ⟦(p, ids)⟧lh ϱ'
σ"
```

```
  shows "⟦(p, ids)⟧lh (solve_pg s dl (Suc n)) σ"
  using assms by (auto simp add: Inter'_def)

lemma all_meaning_if_solve_pg_Suc_lh:
  assumes "⟦(p, ids)⟧lh (solve_pg s dl (Suc n)) σ"
  shows "∀ϱ'. ϱ' ⊨dl (dl ==s== Suc n) ⟶ (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶ ⟦(p, ids)⟧lh ϱ' σ"
  using assms by (auto simp add: Inter'_def)

lemma solve_pg_Suc_iff_all_meaning_lh:
  "(⟦(p, ids)⟧lh (solve_pg s dl (Suc n)) σ) ⟷
   (∀ϱ'. ϱ' ⊨dl (dl ==s== Suc n) ⟶ (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶ ⟦(p, ids)⟧lh ϱ' σ)"
  by (auto simp add: Inter'_def)

lemma solve_pg_0_meaning_cls':
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ 0)"
  shows "⟦Cls p ids rhs⟧cls (solve_pg s dl 0) σ"
  unfolding meaning_cls.simps
proof
  define ϱ'' :: "'a ⇒ 'c list set" where "ϱ'' = (solve_pg s dl 0)"
  assume "⟦rhs⟧rhs (solve_pg s dl 0) σ"
  then have "∀ϱ'. ϱ' ⊨dl (dl ≤≤s≤≤ 0) ⟶ (∀rh∈set rhs. ⟦rh⟧rh ϱ' σ)"
    using all_meaning_rh_if_solve_pg_0[OF assms(1), of _ σ _ rhs p ids, OF _ _ _ assms(2)]
    by auto
  then have "∀ϱ'. ϱ' ⊨dl (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧lh  ϱ' σ"
    by (metis assms(2) meaning_cls.simps solves_cls_def solves_program_def meaning_rhs.simps)
  then show "⟦(p, ids)⟧lh (solve_pg s dl 0) σ"
    using solve_pg_0_if_all_meaning_lh by auto
qed

lemma solve_pg_meaning_cls':
  assumes "strat_wf s dl"
  assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ n)"
  shows "⟦Cls p ids rhs⟧cls (solve_pg s dl n) σ"
  unfolding meaning_cls.simps
  using assms
proof (induction n)
  case 0
  then show ?case
    using solve_pg_0_meaning_cls' unfolding meaning_cls.simps by metis
next
  case (Suc n)
  have leq_n: "s p ≤ Suc n"
    using Suc.prems(2) by auto

  have cls_in: "Cls p ids rhs ∈ dl"
    using assms(2) by force

  show ?case
  proof (cases "s p = Suc n")
    case True
    have cls_in_Suc: "Cls p ids rhs ∈ (dl ==s== Suc n)"
      by (simp add: True cls_in)
    show ?thesis
    proof
      define ϱ'' :: "'a ⇒ 'c list set" where "ϱ'' = (solve_pg s dl (Suc n))"
      assume "⟦rhs⟧rhs (solve_pg s dl (Suc n)) σ"
      then have
        "∀ϱ'. ϱ' ⊨dl (dl ==s== Suc n) ⟶
              (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶
              (∀rh∈set rhs. ⟦rh⟧rh ϱ' σ)"
      using all_meaning_rh_if_solve_pg_Suc[OF assms(1) _ _ _ _ Suc(3), of _ σ]
      by auto
```

```
      then have "∀ϱ'. ϱ' ⊨dl (dl ==s== Suc n) ⟶
                 (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶
                 ⟦(p, ids)⟧lh ϱ' σ"
        using meaning_cls.simps solves_cls_def solves_program_def cls_in_Suc meaning_rhs.simps by metis
      then show "⟦(p, ids)⟧lh (solve_pg s dl (Suc n)) σ"
        using solve_pg_Suc_if_all_meaning_lh[of dl s n p ids σ] by auto
    qed
  next
    case False
    then have False': "s p < Suc n"
      using leq_n by auto
    then have s_p_n: "s p ≤ n"
      by auto
    then have "Cls p ids rhs ∈ (dl ≤≤s≤≤ n)"
      by (simp add: cls_in)
    then have "(∀rh∈set rhs. ⟦rh⟧rh (solve_pg s dl n) σ) ⟶ ⟦(p, ids)⟧lh (solve_pg s dl n) σ"
      using Suc by auto
    then show ?thesis
      using False' meaning_cls.simps solve_pg_two_agree_above_on_cls_Suc assms cls_in s_p_n
        meaning_rhs.simps by metis
  qed
qed

lemma solve_pg_meaning_cls:
  assumes "strat_wf s dl"
  assumes "c ∈ (dl ≤≤s≤≤ n)"
  shows "⟦c⟧cls (solve_pg s dl n) σ"
  using assms solve_pg_meaning_cls'[of s dl] by (cases c) metis

lemma solve_pg_solves_cls:
  assumes "strat_wf s dl"
  assumes "c ∈ (dl ≤≤s≤≤ n)"
  shows "solve_pg s dl n ⊨cls c"
  unfolding solves_cls_def using solve_pg_meaning_cls assms by blast

lemma solve_pg_solves_dl:
  assumes "strat_wf s dl"
  shows "solve_pg s dl n ⊨dl (dl ≤≤s≤≤ n)"
proof -
  have "∀c ∈ (dl ≤≤s≤≤ n). (solve_pg s dl n) ⊨cls c"
    using assms solve_pg_solves_cls[of s dl] by auto
  then show "solve_pg s dl n ⊨dl (dl ≤≤s≤≤ n)"
    using solves_program_def by blast
qed

lemma disjE3:
  assumes major: "P ∨ Q ∨ Z"
    and minorP: "P ⟹ R"
    and minorQ: "Q ⟹ R"
    and minorZ: "Z ⟹ R"
  shows R
  using assms by auto

lemma solve_pg_0_below_solution:
  assumes "ϱ ⊨dl (dl ≤≤s≤≤ 0)"
  shows "(solve_pg s dl 0) ⊑s⊑ ϱ"
proof -
  define ϱ'' :: "'a ⇒ 'b list set" where "ϱ'' = solve_pg s dl 0"

  have "ϱ'' ≠ ϱ ⟶ ϱ'' ⊏s⊏ ϱ"
  proof
    assume "ϱ'' ≠ ϱ"
    have ϱ''_subs_ϱ: "∀p. ϱ'' p ⊆ ϱ p"
```

```
    using solve_pg_0_subset unfolding ϱ''_def using assms(1) by force
  have "∃p. least_rank_p_st (λp. ϱ'' p ≠ ϱ p) p s"
    by (meson ‹ϱ'' ≠ ϱ› ext least_rank_p_st_exists)
  then obtain p where p_p: "least_rank_p_st (λp. ϱ'' p ≠ ϱ p) p s"
    by auto
  then have "ϱ'' p ⊂ ϱ p"
    by (metis (mono_tags, lifting) ϱ''_subs_ϱ least_rank_p_st_def psubsetI)
  moreover
  have "∀p'. s p' = s p ⟶ ϱ'' p' ⊆ ϱ p'"
    using ϱ''_subs_ϱ by auto
  moreover
  have "∀p'. s p' < s p ⟶ ϱ'' p' = ϱ p'"
    using below_least_rank_p_st[OF p_p] by auto
  ultimately
  show "ϱ'' ⊏s⊏ ϱ"
    unfolding lt_def by auto
qed
then show ?thesis
  using ϱ''_def lte_def by auto
qed

lemma least_disagreement_proper_subset:
  assumes "ϱ''n ⊏s⊏ ϱ"
  assumes "least_rank_p_st (λp. ϱ''n p ≠ ϱ p) p s"
  shows "ϱ''n p ⊂ ϱ p"
proof -
  from assms obtain p'' where p''_p:
    "ϱ''n p'' ⊂ ϱ p''"
    "(∀p'. s p' = s p'' ⟶ ϱ''n p' ⊆ ϱ p')"
    "(∀p'. s p' < s p'' ⟶ ϱ''n p' = ϱ p')"
    unfolding lt_def by auto
  moreover
  from p''_p have "s p'' = s p"
    by (metis (mono_tags, lifting) antisym_conv2 assms(2) least_rank_p_st_def)
  ultimately
  show ?thesis
    by (metis (mono_tags, lifting) assms(2) least_rank_p_st_def psubsetI)
qed

lemma subset_on_least_disagreement:
  assumes "ϱ''n ⊏s⊏ ϱ"
  assumes "least_rank_p_st (λp. ϱ''n p ≠ ϱ p) p s"
  assumes "s p' = s p"
  shows "ϱ''n p' ⊆ ϱ p'"
proof -
  from assms obtain p'' where p''_p:
    "ϱ''n p'' ⊂ ϱ p''"
    "(∀p'. s p' = s p'' ⟶ ϱ''n p' ⊆ ϱ p')"
    "(∀p'. s p' < s p'' ⟶ ϱ''n p' = ϱ p')"
    unfolding lt_def by auto
  moreover
  from p''_p have "s p'' = s p"
    by (metis (mono_tags, lifting) antisym_conv2 assms(2) least_rank_p_st_def)
  ultimately
  show ?thesis
    using assms by metis
qed

lemma equal_below_least_disagreement:
  assumes "ϱ''n ⊏s⊏ ϱ"
  assumes "least_rank_p_st (λp. ϱ''n p ≠ ϱ p) p s"
  assumes "s p' < s p"
  shows "ϱ''n p' = ϱ p'"
```

```
proof -
  from assms obtain p'' where p''_p:
    "ϱ''n p'' ⊂ ϱ p''"
    "(∀p'. s p' = s p'' ⟶ ϱ''n p' ⊆ ϱ p')"
    "(∀p'. s p' < s p'' ⟶ ϱ''n p' = ϱ p')"
    unfolding lt_def by auto
  moreover
  from p''_p have "s p'' = s p"
    by (metis (mono_tags, lifting) antisym_conv2 assms(2) least_rank_p_st_def)
  ultimately
  show ?thesis
    using assms by metis
qed

lemma solution_on_subset_solution_below:
  "(dl ==s== n) ⊆ (dl ≤≤s≤≤ n)"
  by fastforce

lemma solves_program_mono:
  assumes "dl ⊆ dl'"
  assumes "ϱ ⊨dl dl'"
  shows "ϱ ⊨dl dl"
  by (meson assms in_mono solves_program_def)

lemma solution_on_if_solution_below:
  assumes "ϱ ⊨dl (dl ≤≤s≤≤ n)"
  shows  "ϱ ⊨dl (dl ==s== n)"
  by (meson assms solves_program_mono solution_on_subset_solution_below)

lemma solve_pg_Suc_subset_solution:
  assumes "ϱ ⊨dl (dl ≤≤s≤≤ Suc n)"
  assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
  shows "solve_pg s dl (Suc n) p ⊆ ϱ p"
  by (meson assms solution_on_if_solution_below solve_pg_Suc_subset)

lemma solve_pg_subset_solution:
  assumes "m > n"
  assumes "ϱ ⊨dl (dl ≤≤s≤≤ m)"
  assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
  shows "solve_pg s dl (Suc n) p ⊆ ϱ p"
  by (meson Suc_leI assms solve_pg_Suc_subset_solution solves_leq)

lemma below_least_disagreement:
  assumes "least_rank_p_st (λp. ϱ' p ≠ ϱ p) p s"
  assumes "s p' < s p"
  shows "ϱ' p' = ϱ p'"
  using assms below_least_rank_p_st by fastforce

definition agree_below_eq :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool"  where
  "agree_below_eq ϱ ϱ' n s ⟷ (∀p. s p ≤ n ⟶ ϱ p = ϱ' p)"

definition agree_below :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool"  where
  "agree_below ϱ ϱ' n s ⟷ (∀p. s p < n ⟶ ϱ p = ϱ' p)"

definition agree_above :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool"  where
  "agree_above ϱ ϱ' n s ⟷ (∀p. s p > n ⟶ ϱ p = ϱ' p)"

definition agree_above_eq :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool" where
  "agree_above_eq ϱ ϱ' n s ⟷ (∀p. s p ≥ n ⟶ ϱ p = ϱ' p)"

lemma agree_below_trans:
  assumes "agree_below_eq ϱ ϱ' n s"
  assumes "agree_below_eq ϱ' ϱ'' n s"
```

```
  shows "agree_below_eq ϱ ϱ'' n s"
  using assms unfolding agree_below_eq_def by auto

lemma agree_below_eq_less_eq:
  assumes "l ≤ n"
  assumes "agree_below_eq ϱ ϱ' n s"
  shows "agree_below_eq ϱ ϱ' l s"
  using assms unfolding agree_below_eq_def by auto

lemma agree_below_trans':
  assumes "agree_below_eq ϱ ϱ' n s"
  assumes "agree_below_eq ϱ' ϱ'' m s"
  assumes "l ≤ n"
  assumes "l ≤ m"
  shows "agree_below_eq ϱ ϱ'' l s"
  using assms unfolding agree_below_eq_def by auto

lemma agree_below_eq_least_disagreement:
  assumes "least_rank_p_st (λp. ϱ' p ≠ ϱ p) p s"
  assumes "n < s p"
  shows "agree_below_eq ϱ' ϱ n s"
  using agree_below_eq_def assms(1) assms(2) below_least_rank_p_st by fastforce

lemma agree_below_least_disagreement:
  assumes "least_rank_p_st (λp. ϱ' p ≠ ϱ p) p s"
  shows "agree_below ϱ' ϱ (s p) s"
  using agree_below_def assms below_least_rank_p_st by fastforce

lemma eq_if_agree_below_eq_agree_above:
  assumes "agree_below_eq ϱ ϱ' n s"
  assumes "agree_above ϱ ϱ' n s"
  shows "ϱ = ϱ'"
  by (metis agree_above_def agree_below_eq_def assms ext le_eq_less_or_eq nat_le_linear)

lemma eq_if_agree_below_agree_above_eq:
  assumes "agree_below ϱ ϱ' n s"
  assumes "agree_above_eq ϱ ϱ' n s"
  shows "ϱ = ϱ'"
  by (metis agree_above_eq_def agree_below_def assms ext le_eq_less_or_eq nat_le_linear)


lemma eq_if_agree_below_eq_agree_above_eq:
  assumes "agree_below_eq ϱ ϱ' n s"
  assumes "agree_above_eq ϱ ϱ' n s"
  shows "ϱ = ϱ'"
  by (meson agree_above_def agree_above_eq_def eq_if_agree_below_eq_agree_above assms
      less_imp_le_nat)

lemma agree_below_eq_pred_val_lte_stratum:
  "agree_below_eq ϱ (ϱ ≤≤≤s≤≤≤ n) n s"
  by (simp add: agree_below_eq_def)

lemma agree_below_eq_pred_val_lte_stratum_less_eq:
  assumes "m ≤ n"
  shows "agree_below_eq ϱ (ϱ ≤≤≤s≤≤≤ n) m s"
  using agree_below_eq_less_eq agree_below_eq_pred_val_lte_stratum assms by blast

lemma agree_below_eq_solve_pg:
  assumes "l ≤ m"
  assumes "l ≤ n"
  shows "agree_below_eq (solve_pg s dl n) (solve_pg s dl m) l s"
  unfolding agree_below_eq_def by (meson assms dual_order.trans solve_pg_two_agree_above)
```

```
lemma solve_pg_below_solution:
  assumes "ϱ ⊨dl (dl ≤≤s≤≤ n)"
  shows "solve_pg s dl n ⊑s⊑ ϱ"
  using assms
proof (induction n arbitrary: ϱ)
  case 0
  then show ?case
    using solve_pg_0_below_solution by blast
next
  case (Suc n)
  define ϱ''n :: "'a ⇒ 'b list set" where "ϱ''n = solve_pg s dl n"
  define ϱ''n1 :: "'a ⇒ 'b list set" where "ϱ''n1 = solve_pg s dl (Suc n)"

  have ϱ''n_below_ϱ: "ϱ''n ⊑s⊑ ϱ"
    using Suc.IH Suc.prems ϱ''n_def solves_Suc by blast

  have agree_ϱ''n1_ϱ''n: "agree_below_eq ϱ''n1 ϱ''n n s"
    unfolding ϱ''n_def ϱ''n1_def using agree_below_eq_solve_pg using le_Suc_eq by blast

  have "ϱ''n1 ⊑s⊑ ϱ"
    unfolding lte_def2
  proof
    assume "ϱ''n1 ≠ ϱ"
    then have "∃p. least_rank_p_st (λp. ϱ''n1 p ≠ ϱ p) p s"
      using least_rank_p_st_exists[of "(λp. ϱ''n1 p ≠ ϱ p)"] by force
    then obtain p where p_p: "least_rank_p_st (λp. ϱ''n1 p ≠ ϱ p) p s"
      by blast
    then have dis: "ϱ''n1 p ≠ ϱ p"
      unfolding least_rank_p_st_def by auto
    from p_p have agg: "agree_below ϱ''n1 ϱ (s p) s"
      by (simp add: agree_below_least_disagreement)

    define i where "i = s p"
    have "i < Suc n ∨ Suc n ≤ i"
      by auto
    then show "ϱ''n1 ⊏s⊏ ϱ"
    proof (rule disjE)
      assume "i < Suc n"
      then have "s p ≤ n"
        unfolding i_def by auto
      then have "ϱ''n p ≠ ϱ p"
        by (metis agree_ϱ''n1_ϱ''n agree_below_eq_def dis)

      have "ϱ''n ⊏s⊏ ϱ"
        by (metis ϱ''n_below_ϱ ‹ϱ''n p ≠ ϱ p› lte_def)
      moreover
      have "∀p'. ϱ''n p' ≠ ϱ p' ⟶ s p ≤ s p'"
        by (metis agree_ϱ''n1_ϱ''n ‹s p ≤ n› agg agree_below_def agree_below_eq_def le_trans
            linorder_le_cases linorder_le_less_linear)
      then have "least_rank_p_st (λp. ϱ''n p ≠ ϱ p) p s"
          using ‹ϱ''n p ≠ ϱ p› unfolding least_rank_p_st_def by auto
      ultimately
      have "ϱ''n p ⊂ ϱ p ∧
           (∀p'. s p' = s p ⟶ ϱ''n p' ⊆ ϱ p') ∧
           (∀p'. s p' < s p ⟶ ϱ''n p' = ϱ p')"
        using least_disagreement_proper_subset[of ϱ''n s ϱ p]
          subset_on_least_disagreement[of ϱ''n s ϱ p] equal_below_least_disagreement[of ϱ''n s ϱ p]
        by metis
      then have "ϱ''n1 p ⊂ ϱ p ∧
           (∀p'. s p' = s p ⟶ ϱ''n1 p' ⊆ ϱ p') ∧
           (∀p'. s p' < s p ⟶ ϱ''n1 p' = ϱ p')"
        using agree_ϱ''n1_ϱ''n ‹s p ≤ n› by (simp add: agree_below_eq_def)
      then show "ϱ''n1 ⊏s⊏ ϱ"
```

```
      unfolding lt_def by auto
  next
    assume "Suc n ≤ i"
    have "ϱ ⊨dl (dl ≤≤s≤≤ Suc n)"
      using Suc.prems by auto
    moreover
    have "(ϱ ≤≤≤s≤≤≤ n) = (solve_pg s dl n)"
    proof -
      have "agree_below_eq ϱ''n ϱ''n1 n s"
        by (metis agree_ϱ''n1_ϱ''n agree_below_eq_def)
      moreover
      have "agree_below_eq ϱ''n1 ϱ n s"
        using ‹Suc n ≤ i› agree_below_eq_least_disagreement i_def p_p by fastforce
      moreover
      have "agree_below_eq ϱ (ϱ ≤≤≤s≤≤≤ n) n s"
        by (simp add: agree_below_eq_pred_val_lte_stratum)
      ultimately
      have "agree_below_eq ϱ''n (ϱ ≤≤≤s≤≤≤ n) n s"
        using agree_below_trans by metis
      moreover
      have "agree_above ϱ''n (ϱ ≤≤≤s≤≤≤ n) n s"
        using ϱ''n_def by (simp add: agree_above_def solve_pg_above_empty)
      ultimately
      have "(ϱ ≤≤≤s≤≤≤ n) = ϱ''n"
         using eq_if_agree_below_eq_agree_above by blast
      then show "(ϱ ≤≤≤s≤≤≤ n) = (solve_pg s dl n)"
        using ϱ''n_def by metis
    qed
    ultimately
    have "ϱ ⊨dl (dl ≤≤s≤≤ Suc n) ∧ (ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
      by auto
    then have "ϱ''n1 p ⊆ ϱ p"
      using solve_pg_subset_solution[of n "Suc n" ϱ dl s p]
      using ϱ''n1_def by fastforce
    then have "ϱ''n1 p ⊂ ϱ p"
      using dis by auto
    moreover
    have "∀p'. s p' = s p ⟶ ϱ''n1 p' ⊆ ϱ p'"
      using ‹ϱ ⊨dl (dl ≤≤s≤≤ Suc n) ∧ (ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n› ϱ''n1_def
        solve_pg_subset_solution by fastforce
    moreover
    have "∀p'. s p' < s p ⟶ ϱ''n1 p' = ϱ p'"
      using below_least_rank_p_st p_p by fastforce
    ultimately
    show "ϱ''n1 ⊏s⊏ ϱ"
      unfolding lt_def by auto
  qed
 qed
 then show ?case
   unfolding ϱ''n1_def by auto
qed

lemma solve_pg_least_solution':
 assumes "strat_wf s dl"
 shows "solve_pg s dl n ⊨lst (dl ≤≤s≤≤ n) s"
 using assms least_solution_def solve_pg_below_solution solve_pg_solves_dl by blast

lemma stratum_less_eq_max_stratum:
 assumes "finite dl"
 assumes "Cls p ids rhs ∈ dl"
 shows "s p ≤ max_stratum s dl"
proof -
 have "s p ∈ {s p | p ids rhs. Cls p ids rhs ∈ dl}"
```

```
    using assms(2) by auto
  moreover
  have "{s p | p ids rhs. Cls p ids rhs ∈ dl} = (λc. (case c of Cls p ids rhs ⇒ s p)) ‘ dl"
    unfolding image_def by (metis (mono_tags, lifting) clause.case the_lh.cases)
  then have "finite {s p | p ids rhs. Cls p ids rhs ∈ dl}"
    by (simp add: assms(1))
  ultimately
  show ?thesis
    unfolding max_stratum_def using Max.coboundedI by auto
qed

lemma finite_above_max_stratum:
  assumes "finite dl"
  assumes "max_stratum s dl ≤ n"
  shows "(dl ≤≤s≤≤ n) = dl"
proof (rule; rule)
  fix c
  assume "c ∈ (dl ≤≤s≤≤ n)"
  then show "c ∈ dl"
    by auto
next
  fix c
  assume c_in_dl: "c ∈ dl"
  then obtain p ids rhs where c_split: "c = Cls p ids rhs"
    by (cases c) auto
  then have c_in_dl': "Cls p ids rhs ∈ dl"
    using c_in_dl by auto
  then have "s p ≤ max_stratum s dl"
    using stratum_less_eq_max_stratum assms by metis
  then have "Cls p ids rhs ∈ (dl ≤≤s≤≤ n)"
    using c_in_dl' assms(2) by auto
  then show "c ∈ (dl ≤≤s≤≤ n)"
    unfolding c_split by auto
qed

lemma finite_max_stratum:
  assumes "finite dl"
  shows "(dl ≤≤s≤≤ max_stratum s dl) = dl"
  using assms finite_above_max_stratum[of dl s "max_stratum s dl"] by auto

lemma solve_pg_least_solution:
  assumes "finite dl"
  assumes "strat_wf s dl"
  shows "solve_pg s dl (max_stratum s dl) ⊨_lst dl s"
proof -
  have "solve_pg s dl (max_stratum s dl) ⊨_lst (dl ≤≤s≤≤ (max_stratum s dl)) s"
    using solve_pg_least_solution' assms by auto
  then show ?thesis
    using finite_max_stratum assms by metis
qed

lemma exi_least_solution:
  assumes "finite dl"
  assumes "strat_wf s dl"
  shows "∃ϱ. ϱ ⊨_lst dl s"
  using assms solve_pg_least_solution by metis
```

6.2.2 Equality of least and minimal solution

```
lemma least_iff_minimal:
  assumes "finite dl"
  assumes "strat_wf s dl"
  shows "ϱ ⊨_lst dl s ⟷ ϱ ⊨_min dl s"
proof
```

```
  assume "ϱ ⊨_lst dl s"
  then have ϱ_least: "ϱ ⊨_dl dl" "(∀σ'. σ' ⊨_dl dl ⟶ ϱ ⊑s⊑ σ')"
    unfolding least_solution_def by auto
  then have "(∄ϱ'. ϱ' ⊨_dl dl ∧ ϱ' ⊏s⊏ ϱ)"
    by (metis (full_types) ‹∀ϱ'. ϱ' ⊨_dl dl ⟶ ϱ ⊑s⊑ ϱ'› lt_def lte_def nat_neq_iff psubsetE)
  then show "ϱ ⊨_min dl s"
    unfolding minimal_solution_def using ϱ_least by metis
next
  assume min: "ϱ ⊨_min dl s"
  have "∃ϱ'. ϱ' ⊨_lst dl s"
    using solve_pg_least_solution assms by metis
  then show "ϱ ⊨_lst dl s"
    by (metis min least_solution_def lte_def minimal_solution_def)
qed
```

6.2.3 Least solution on lower strata

```
lemma below_subset:
  "(dl ≤≤s≤≤ n) ⊆ dl"
  by auto

lemma finite_below_finite:
  assumes "finite dl"
  shows "finite (dl ≤≤s≤≤ n)"
  using assms finite_subset below_subset by metis

lemma downward_least_solution:
  assumes "finite dl"
  assumes "n > m"
  assumes "strat_wf s dl"
  assumes "ϱ ⊨_lst (dl ≤≤s≤≤ n) s"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨_lst (dl ≤≤s≤≤ m) s"
proof (rule ccontr)
  assume a: "¬ (ϱ ≤≤≤s≤≤≤ m) ⊨_lst (dl ≤≤s≤≤ m) s"
  have s_dl_m: "strat_wf s (dl ≤≤s≤≤ m)"
    using assms strat_wf_lte_if_strat_wf by auto
  have strat_wf_n: "strat_wf s (dl ≤≤s≤≤ n)"
    using assms strat_wf_lte_if_strat_wf by auto
  from a have "¬ (ϱ ≤≤≤s≤≤≤ m) ⊨_min (dl ≤≤s≤≤ m) s"
    using least_iff_minimal s_dl_m assms(1) finite_below_finite by metis
  moreover
  have "(ϱ ≤≤≤s≤≤≤ m) ⊨_dl (dl ≤≤s≤≤ m)"
    using assms downward_lte_solves least_solution_def by blast
  ultimately
  have "(∃σ'. σ' ⊨_dl (dl ≤≤s≤≤ m) ∧ (σ' ⊏s⊏ (ϱ ≤≤≤s≤≤≤ m)))"
    unfolding minimal_solution_def by auto
  then obtain ϱ' where ϱ'_p1: "ϱ' ⊨_dl (dl ≤≤s≤≤ m)" and ϱ'_p2: "(ϱ' ⊏s⊏ (ϱ ≤≤≤s≤≤≤ m))"
    by auto
  then have "∃p. ϱ' p ⊂ (ϱ ≤≤≤s≤≤≤ m) p ∧
                  (∀p'. s p' = s p ⟶ ϱ' p' ⊆ (ϱ ≤≤≤s≤≤≤ m) p') ∧
                  (∀p'. s p' < s p ⟶ ϱ' p' = (ϱ ≤≤≤s≤≤≤ m) p')"
    unfolding lt_def by auto
  then obtain p where p_p1: "ϱ' p ⊂ (ϱ ≤≤≤s≤≤≤ m) p" and
    p_p2: "∀p'. s p' = s p ⟶ ϱ' p' ⊆ (ϱ ≤≤≤s≤≤≤ m) p'" and
    p_p3: "∀p'. s p' < s p ⟶ ϱ' p' = (ϱ ≤≤≤s≤≤≤ m) p'"
    by auto
  define ϱ'' where "ϱ'' == λp. if s p ≤ m then ϱ' p else UNIV"

  have "s p ≤ m"
    using p_p1 by auto
  then have "ϱ'' p ⊂ ϱ p"
    unfolding ϱ''_def using p_p1 by auto
  moreover
  have "∀p'. s p' = s p ⟶ ϱ'' p' ⊆ ϱ p'"
```

```
  using p_p2
  by (metis ϱ''_def calculation pred_val_lte_stratum.simps top.extremum_strict)
moreover
have "∀p'. s p' < s p ⟶ ϱ'' p' = ϱ p'"
  using ϱ''_def p_p3 calculation(1) by force
ultimately
have "(ϱ'' ⊏s⊏ ϱ)"
  by (metis lt_def)
moreover
have "ϱ'' ⊨dl (dl ≤≤s≤≤ n)"
  unfolding solves_program_def
proof
  fix c
  assume c_in_dl_n: "c ∈ (dl ≤≤s≤≤ n)"
  then obtain p ids rhs where c_def: "c = Cls p ids rhs"
    by (cases c) auto

  have "ϱ'' ⊨cls Cls p ids rhs"
    unfolding solves_cls_def
  proof
    fix σ
    show "⟦Cls p ids rhs⟧cls ϱ'' σ"
    proof (cases "s p ≤ m")
      case True
      then have "c ∈ (dl ≤≤s≤≤ m)"
        using c_in_dl_n c_def by auto
      then have "⟦Cls p ids rhs⟧cls ϱ' σ"
        using ϱ'_p1 c_def solves_cls_def solves_program_def by blast

      show ?thesis
        unfolding meaning_cls.simps
      proof
        assume "⟦rhs⟧rhs ϱ'' σ"
        have "⟦rhs⟧rhs ϱ' σ"
          unfolding meaning_rhs.simps
        proof
          fix rh
          assume "rh ∈ set rhs"
          then have "⟦rh⟧rh ϱ'' σ"
            using ‹⟦rhs⟧rhs ϱ'' σ› by force
          then show "⟦rh⟧rh ϱ' σ"
          proof (cases rh)
            case (Eql a a')
            then show ?thesis
              using ‹⟦rh⟧rh ϱ'' σ› by auto
          next
            case (Neql a a')
            then show ?thesis
              using ‹⟦rh⟧rh ϱ'' σ› by auto
          next
            case (PosLit p ids)
            then show ?thesis
              using ‹⟦rh⟧rh ϱ'' σ› ‹c ∈ (dl ≤≤s≤≤ m)› ‹rh ∈ set rhs› ϱ''_def assms(3) c_def
                pos_rhs_stratum_leq_clause_stratum by fastforce
          next
            case (NegLit p ids)
            then show ?thesis
              using ‹⟦rh⟧rh ϱ'' σ› ‹c ∈ (dl ≤≤s≤≤ m)› ‹rh ∈ set rhs› ϱ''_def c_def
                neg_rhs_stratum_less_clause_stratum s_dl_m by fastforce
          qed
        qed
        then have "⟦(p, ids)⟧lh ϱ' σ"
          using ‹⟦Cls p ids rhs⟧cls ϱ' σ› by force
```

```
        then show  "⟦(p, ids)⟧lh ϱ'' σ"
          using ϱ''_def by force
      qed
    next
      case False
      then show ?thesis
        by (simp add: ϱ''_def)
    qed
  qed
  then show "ϱ'' ⊨cls c"
    using c_def by blast
qed
ultimately
have "¬ϱ ⊨min (dl ≤≤s≤≤ n) s"
  unfolding minimal_solution_def by auto
then have "¬ϱ ⊨lst (dl ≤≤s≤≤ n) s"
  using least_iff_minimal strat_wf_n finite_below_finite assms by metis
then show "False"
  using assms by auto
qed

lemma downward_least_solution_same_stratum:
  assumes "finite dl"
  assumes "strat_wf s dl"
  assumes "ϱ ⊨lst dl s"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨lst (dl ≤≤s≤≤ m) s"
proof (cases "m < max_stratum s dl")
  case True
  have "(dl ≤≤s≤≤ max_stratum s dl) = dl"
    using assms(1) finite_max_stratum by blast
  with True show ?thesis
    using downward_least_solution[of dl m "max_stratum s dl" s ϱ]
      assms by auto
next
  case False
  then have "max_stratum s dl ≤ m"
    by auto
  moreover
  have "(dl ≤≤s≤≤ m) = dl"
    using assms(1) calculation finite_above_max_stratum by blast
  then
  show ?thesis
    using assms below_subset downward_least_solution least_solution_def lessI less_imp_le_nat
      solves_leq[of _ dl s _ m] solves_program_mono[of _ dl ϱ ] by metis
qed
```

6.3 Negation

```
definition agree_var_val :: "'x set ⇒ ('x, 'c) var_val ⇒ ('x, 'c) var_val ⇒ bool " where
  "agree_var_val xs σ σ' ⟷ (∀x ∈ xs. σ x = σ' x)"

fun vars_ids :: "('a, 'b) id list ⇒ 'a set" where
  "vars_ids ids = ⋃(vars_id ‘ set ids)"

fun vars_lh :: "('p,'x,'c) lh ⇒ 'x set" where
  "vars_lh (p,ids) = vars_ids ids"

definition lh_consequence :: "('p, 'c) pred_val ⇒ ('p, 'x, 'c) clause ⇒ ('p, 'x, 'c) lh ⇒ bool"
  where
  "lh_consequence ϱ c lh ⟷ (∃σ'. ((the_lh c) ·vlh σ') = lh ∧ ⟦the_rhs c⟧rhs ϱ σ')"

lemma meaning_rh_iff_meaning_rh_pred_val_lte_stratum:
  assumes "c ∈ (dl ≤≤s≤≤ s p)"
  assumes "strat_wf s dl"
```

```
  assumes "rh ∈ set (the_rhs c)"
  shows "⟦rh⟧_rh ϱ σ' ⟷ ⟦rh⟧_rh (ϱ ≤≤≤s≤≤≤ s p) σ'"
proof (cases rh)
  case (Eql a a')
  then show ?thesis
    by auto
next
  case (Neql a a')
  then show ?thesis
    by auto
next
  case (PosLit p' ids)
  show ?thesis
  proof
    assume "⟦rh⟧_rh ϱ σ'"
    then show "⟦rh⟧_rh (ϱ ≤≤≤s≤≤≤ s p) σ'"
      using PosLit assms pos_rhs_stratum_leq_clause_stratum by fastforce
  next
    assume "⟦rh⟧_rh (ϱ ≤≤≤s≤≤≤ s p) σ'"
    then show "⟦rh⟧_rh ϱ σ'"
      by (metis PosLit equals0D meaning_rh.simps(3) pred_val_lte_stratum.simps)
  qed
next
  case (NegLit p' ids)
  show ?thesis
  proof
    assume "⟦rh⟧_rh ϱ σ'"
    then show "⟦rh⟧_rh (ϱ ≤≤≤s≤≤≤ s p) σ'"
      using NegLit assms by fastforce
  next
    assume "⟦rh⟧_rh (ϱ ≤≤≤s≤≤≤ s p) σ'"
    then show "⟦rh⟧_rh ϱ σ'"
      using NegLit assms(1) assms(2) assms(3) neg_rhs_stratum_less_clause_stratum by fastforce
  qed
qed

lemma meaning_rhs_iff_meaning_rhs_pred_val_lte_stratum:
  assumes "c ∈ (dl ≤≤s≤≤ s p)"
  assumes "strat_wf s dl"
  shows "⟦the_rhs c⟧_rhs ϱ σ' ⟷ ⟦the_rhs c⟧_rhs (ϱ ≤≤≤s≤≤≤ s p) σ'"
  by (meson assms(1) assms(2) meaning_rh_iff_meaning_rh_pred_val_lte_stratum meaning_rhs.simps)

lemma meaning_rhs_if_meaning_rhs_with_removed_top_strata:
  assumes "⟦rhs⟧_rhs (ϱ'(p := ϱ' p - {⟦ids⟧_ids σ})) σ'"
  assumes "strat_wf s dl"
  assumes c_dl': "Cls p' ids' rhs ∈ (dl ≤≤s≤≤ s p)"
  shows "⟦rhs⟧_rhs ϱ' σ'"
proof -
  have "∀rh' ∈ set rhs. rnk s rh' ≤ s p'"
    using assms c_dl' in_mono strat_wf_cls.simps strat_wf_def by fastforce

  show ?thesis
    unfolding meaning_rhs.simps
  proof
    fix rh
    assume "rh ∈ set rhs"
    show "⟦rh⟧_rh ϱ' σ'"
    proof (cases rh)
      case (Eql a a')
      then show ?thesis
        using ‹rh ∈ set rhs› assms by auto
    next
      case (Neql a a')
```

```
      then show ?thesis
        using ‹rh ∈ set rhs› assms by auto
    next
      case (PosLit p'' ids'')
      then show ?thesis
        by (metis Diff_empty Diff_insert0 ‹rh ∈ set rhs› assms fun_upd_other fun_upd_same insertCI
            insert_Diff meaning_rh.simps(3) meaning_rhs.elims(2))
    next
      case (NegLit p'' ids'')
      have "p'' ≠ p"
        using NegLit One_nat_def ‹∀rh'∈set rhs. rnk s rh' ≤ s p'› ‹rh ∈ set rhs› c_dl' by fastforce
      then show ?thesis
        using NegLit  ‹rh ∈ set rhs› assms by auto
    qed
  qed
qed

lemma meaning_PosLit_least':
  assumes "finite dl"
  assumes "ϱ ⊨_lst dl s"
  assumes "strat_wf s dl"
  assumes "⟦+ p ids⟧_rh ϱ σ"
  shows "∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·_vlh σ)"
proof (rule ccontr)
  assume "¬(∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·_vlh σ))"
  then have "¬(∃c ∈ dl. ∃σ'. ((the_lh c) ·_vlh σ') = ((p,ids) ·_vlh σ) ∧ (⟦the_rhs c⟧_rhs ϱ σ'))"
    unfolding lh_consequence_def by auto
  then have a: "∀c ∈ dl. ∀σ'. ((the_lh c) ·_vlh σ') = ((p,ids) ·_vlh σ) ⟶ ¬(⟦the_rhs c⟧_rhs ϱ σ')"
    by metis

  define ϱ' where "ϱ' = (ϱ ≤≤≤s≤≤≤ s p)"
  define dl' where "dl' = (dl ≤≤s≤≤ s p)"

  have ϱ'_least: "ϱ' ⊨_lst dl' s"
    using downward_solves[of ϱ dl s] assms downward_least_solution_same_stratum
    unfolding ϱ'_def dl'_def by blast
  moreover
  have no_match: "∀c ∈ dl'. ∀σ'. ((the_lh c) ·_vlh σ') = ((p,ids) ·_vlh σ) ⟶ ¬(⟦the_rhs c⟧_rhs ϱ' σ')"
    using a
    unfolding dl'_def ϱ'_def
    by (meson assms(3) below_subset meaning_rhs_iff_meaning_rhs_pred_val_lte_stratum in_mono)

  define ϱ'' where "ϱ'' = ϱ'(p := ϱ' p - {⟦ids⟧_ids σ})"


  have "ϱ'' ⊨_dl dl'"
    unfolding solves_program_def
  proof
    fix c
    assume c_dl': "c ∈ dl'"
    obtain p' ids' rhs' where c_split: "c = Cls p' ids' rhs'"
      by (cases c)
    show "ϱ'' ⊨_cls c"
      unfolding solves_cls_def
    proof
      fix σ'
      have "⟦Cls p' ids' rhs'⟧_cls ϱ'' σ'"
        unfolding meaning_cls.simps
      proof
        assume "⟦rhs'⟧_rhs ϱ'' σ'"
        then have rhs'_true: "⟦rhs'⟧_rhs ϱ' σ'"
          using meaning_rhs_if_meaning_rhs_with_removed_top_strata[of rhs' ϱ' p ids σ σ']
            ϱ''_def assms(3) c_dl' c_split dl'_def using ϱ''_def by force
```

```
        have "⟦(p',ids')⟧lh ϱ' σ'"
          by (metis rhs'_true c_split c_dl' ϱ'_least clause.inject least_solution_def
              meaning_cls.elims(2) solves_cls_def solves_program_def)
        moreover
        have "((p', ids') ·vlh σ') ≠ ((p, ids) ·vlh σ)"
          using no_match rhs'_true c_split c_dl' by fastforce
        ultimately
        show "⟦(p', ids')⟧lh ϱ'' σ'"
          using  ϱ''_def eval_ids_is_substv_ids by auto
      qed
      then show "⟦c⟧cls ϱ'' σ'"
        unfolding c_split by auto
    qed
  qed
  moreover
  have "ϱ'' ⊏s⊏ ϱ'"
  proof -
    have "ϱ'' p ⊂ ϱ' p"
      unfolding ϱ'_def
      using DiffD2 ‹⟦+ p ids⟧rh ϱ σ› ϱ''_def ϱ'_def by auto
    moreover
    have "∀p'. s p' = s p ⟶ ϱ'' p' ⊆ ϱ' p'"
      unfolding ϱ'_def
      by (simp add: ϱ''_def ϱ'_def)
    moreover
    have "∀p'. s p' < s p ⟶ ϱ'' p' = ϱ' p'"
      using ϱ''_def by force
    ultimately
    show "ϱ'' ⊏s⊏ ϱ'"
      unfolding lt_def by auto
  qed
  ultimately
  show "False"
    by (metis assms(1,3) dl'_def finite_below_finite least_iff_minimal minimal_solution_def
        strat_wf_lte_if_strat_wf)
qed

lemma meaning_lh_least':
  assumes "finite dl"
  assumes "ϱ ⊨lst dl s"
  assumes "strat_wf s dl"
  assumes "⟦(p, ids)⟧lh ϱ σ"
  shows "∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·vlh σ)"
  using assms meaning_PosLit_least' by fastforce

lemma meaning_lh_least:
  assumes "finite dl"
  assumes "ϱ ⊨lst dl s"
  assumes "strat_wf s dl"
  shows "⟦(p, ids)⟧lh ϱ σ ⟷ (∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·vlh σ))"
proof
  assume "⟦(p, ids)⟧lh ϱ σ"
  then show "∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·vlh σ)"
    by (meson assms meaning_lh_least')
next
  assume "∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·vlh σ)"
  then show "⟦(p, ids)⟧lh ϱ σ"
    unfolding lh_consequence_def
    using assms(2) eval_ids_is_substv_ids least_solution_def meaning_cls.simps meaning_lh.simps
      solves_cls_def solves_program_def clause.exhaust clause.sel(3) prod.inject substv_lh.simps
      the_lh.simps by metis
qed
```

```
lemma meaning_PosLit_least:
  assumes "finite dl"
  assumes "ϱ ⊨_lst dl s"
  assumes "strat_wf s dl"
  shows "⟦+ p ids⟧_rh ϱ σ ⟷ (∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·_vlh σ))"
proof
  assume "⟦+ p ids⟧_rh ϱ σ"
  then show "∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·_vlh σ)"
    by (meson assms(1,2,3) meaning_PosLit_least')
next
  assume "∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·_vlh σ)"
  then show "⟦+ p ids⟧_rh ϱ σ"
    unfolding lh_consequence_def
    using assms(2) eval_ids_is_substv_ids least_solution_def meaning_cls.simps meaning_lh.simps
      solves_cls_def solves_program_def clause.exhaust clause.sel(3) meaning_rh.simps(3) prod.inject

      substv_lh.simps the_lh.simps by metis
qed

lemma meaning_NegLit_least:
  assumes "finite dl"
  assumes "ϱ ⊨_lst dl s"
  assumes "strat_wf s dl"
  shows "⟦¬ p ids⟧_rh ϱ σ ⟷ (¬(∃c ∈ dl. lh_consequence ϱ c ((p,ids) ·_vlh σ)))"
  by (metis assms(1,2,3) meaning_PosLit_least meaning_rh.simps(3) meaning_rh.simps(4))

lemma solves_PosLit_least:
  assumes "finite dl"
  assumes "ϱ ⊨_lst dl s"
  assumes "strat_wf s dl"
  assumes "∀a ∈ set ids. is_Cst a"
  shows "ϱ ⊨_rh (+ p ids) ⟷ (∃c ∈ dl. lh_consequence ϱ c (p,ids))"
proof -
  have "∀σ. ((p, ids) ·_vlh σ) = (p, ids)"
    using assms(4) by (induction ids) (auto simp add: is_Cst_def)
  then show ?thesis
    by (metis assms(1,2,3) meaning_PosLit_least solves_rh.simps)
qed

lemma solves_lh_least:
  assumes "finite dl"
  assumes "ϱ ⊨_lst dl s"
  assumes "strat_wf s dl"
  assumes "∀a ∈ set ids. is_Cst a"
  shows "ϱ ⊨_lh (p, ids) ⟷ (∃c ∈ dl. lh_consequence ϱ c (p,ids))"
proof -
  have "∀σ. ((p, ids) ·_vlh σ) = (p, ids)"
    using assms(4) by (induction ids) (auto simp add: is_Cst_def)
  then show ?thesis
    by (metis assms(1,2,3) meaning_lh_least solves_lh.simps)
qed

lemma solves_NegLit_least:
  assumes "finite dl"
  assumes "ϱ ⊨_lst dl s"
  assumes "strat_wf s dl"
  assumes "∀a ∈ set ids. is_Cst a"
  shows "ϱ ⊨_rh (¬ p ids) ⟷ ¬(∃c ∈ dl. lh_consequence ϱ c (p,ids))"
proof -
  have "∀σ. ((p, ids) ·_vlh σ) = (p, ids)"
    using assms(4) by (induction ids) (auto simp add: is_Cst_def)
  then show ?thesis
    by (metis assms(1,2,3) meaning_NegLit_least solves_rh.simps)
```

qed

end
theory Program_Graph **imports** Labeled_Transition_Systems.LTS Datalog **begin**

7 Actions

```
datatype (fv_arith: 'v) arith =
  Integer int
  | Arith_Var 'v
  | Arith_Op "'v arith" "int ⇒ int ⇒ int" "'v arith"
  | Minus "'v arith"

datatype (fv_boolean: 'v) boolean =
  true
  | false
  | Rel_Op "'v arith" "int ⇒ int ⇒ bool" "'v arith"
  | Bool_Op "'v boolean" "bool ⇒ bool ⇒ bool" "'v boolean"
  | Neg "'v boolean"

datatype 'v action =
  Asg 'v "'v arith" ("_ ::= _" [1000, 61] 61)
  | Bool "'v boolean"
  | Skip
```

8 Memories

```
type_synonym 'v memory = "'v ⇒ int"
```

9 Semantics

```
fun sem_arith :: "'v arith ⇒ 'v memory ⇒ int" where
  "sem_arith (Integer n) σ = n"
| "sem_arith (Arith_Var x) σ = σ x"
| "sem_arith (Arith_Op a1 op a2) σ = op (sem_arith a1 σ) (sem_arith a2 σ)"
| "sem_arith (Minus a) σ = - (sem_arith a σ)"

fun sem_bool :: "'v boolean ⇒ 'v memory ⇒ bool" where
  "sem_bool true σ = True"
| "sem_bool false σ = False"
| "sem_bool (Rel_Op a1 op a2) σ = op (sem_arith a1 σ) (sem_arith a2 σ)"
| "sem_bool (Bool_Op b1 op b2) σ = op (sem_bool b1 σ) (sem_bool b2 σ)"
| "sem_bool (Neg b) σ = (¬(sem_bool b σ))"

fun sem_action :: "'v action ⇒ 'v memory ⇀ 'v memory" where
  "sem_action (x ::= a) σ = Some (σ(x := sem_arith a σ))"
| "sem_action (Bool b) σ = (if sem_bool b σ then Some σ else None)"
| "sem_action Skip σ = Some σ"
```

10 Program Graphs

10.1 Types

```
type_synonym ('n,'a) edge = "'n × 'a × 'n"

type_synonym ('n,'a) program_graph = "('n,'a) edge set × 'n × 'n"

type_synonym ('n,'v) config = "'n * 'v memory"
```

10.2 Program Graph Locale

```
locale program_graph =
  fixes pg :: "('n,'a) program_graph"
begin

definition edges where
  "edges = fst pg"

definition start where
  "start = fst (snd pg)"

definition "end" where
  "end = snd (snd pg)"

definition pg_rev :: "('n,'a) program_graph" where
  "pg_rev = (rev_edge ‘ edges, end, start)"

end
```

10.3 Finite Program Graph Locale

```
locale finite_program_graph = program_graph pg
  for pg :: "('n::finite,'v) program_graph" +
  assumes "finite edges"
begin

lemma finite_pg_rev: "finite (fst pg_rev)"
  by (metis finite_program_graph_axioms finite_program_graph_def finite_imageI fst_conv pg_rev_def)

end

locale finite_action_program_graph =
  fixes pg :: "('n,'v action) program_graph"
begin

interpretation program_graph pg .
```

— Execution Sequences:

```
fun initial_config_of :: "('n,'v) config ⇒ bool" where
  "initial_config_of (q,σ) ⟷ q = start"

fun final_config_of :: "('n,'v) config ⇒ bool" where
  "final_config_of (q,σ) ⟷ q = end"

inductive exe_step :: "('n,'v) config ⇒ 'v action ⇒ ('n,'v) config ⇒ bool" where
  "(q1, α, q2) ∈ edges ⟹ sem_action α σ = Some σ' ⟹ exe_step (q1,σ) α (q2,σ')"

end


end
theory Bit_Vector_Framework imports Program_Graph begin
```

— We encode the Bit Vector Framework into Datalog.

11 Bit-Vector Framework

11.1 Definitions

```
datatype pred =
  the_may
  | the_must
```

```
  | the_kill
  | the_gen
  | the_init
  | the_anadom

datatype var =
  the_u

abbreviation "may == PosLit the_may"
abbreviation "must == PosLit the_must"
abbreviation NegLit_BV ("¬may") where
  "¬may ≡ NegLit the_may"
abbreviation "kill == PosLit the_kill"
abbreviation NegLit_kill ("¬kill") where
  "¬kill ≡ NegLit the_kill"
abbreviation "gen == PosLit the_gen"
abbreviation "init == PosLit the_init"
abbreviation "anadom == PosLit the_anadom"

fun s_BV :: "pred ⇒ nat" where
  "s_BV the_kill = 0"
| "s_BV the_gen = 0"
| "s_BV the_init = 0"
| "s_BV the_anadom = 0"
| "s_BV the_may = 1"
| "s_BV the_must = 2"

datatype ('n,'a,'d) cst =
  is_Node: Node (the_Node: 'n)
  | is_Elem: Elem (the_Elem: 'd)
  | is_Action: Action (the_Action: "'a")

abbreviation may_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("may⟨_⟩ :- _ .") where
   "may⟨ids⟩ :- ls. ≡ Cls the_may ids ls"

abbreviation must_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("must⟨_⟩ :- _ .") where
  "must⟨ids⟩ :- ls. ≡ Cls the_must ids ls"

abbreviation init_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("init⟨_⟩ :- _ .") where
  "init⟨ids⟩ :- ls. ≡ Cls the_init ids ls"

abbreviation anadom_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("anadom⟨_⟩ :- _ .") where
  "anadom⟨ids⟩ :- ls. ≡ Cls the_anadom ids ls"

abbreviation kill_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("kill⟨_⟩ :- _ .") where
  "kill⟨ids⟩ :- ls. ≡ Cls the_kill ids ls"

abbreviation gen_Cls
```

```
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("gen⟨_⟩ :- _ .") where
  "gen⟨ids⟩ :- ls. ≡ Cls the_gen ids ls"

abbreviation BV_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("may⟨_⟩.") where
  "may⟨ids⟩. ≡ (the_may, ids)"

abbreviation must_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("must⟨_⟩.") where
  "must⟨ids⟩. ≡ (the_must, ids)"

abbreviation init_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("init⟨_⟩.") where
  "init⟨ids⟩. ≡ (the_init, ids)"

abbreviation dom_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("anadom⟨_⟩.") where
  "anadom⟨ids⟩. ≡ (the_anadom, ids)"

abbreviation u :: "(var, 'a) id" where
  "u == Var the_u"

abbreviation Cst_N :: "'n ⇒ (var, ('n, 'a, 'd) cst) id" where
  "Cst_N q == Cst (Node q)"

abbreviation Cst_E :: "'d ⇒ (var, ('n, 'a, 'd) cst) id" where
  "Cst_E e == Cst (Elem e)"

abbreviation Cst_A :: "'a ⇒ (var, ('n, 'a, 'd) cst) id" where
  "Cst_A α == Cst (Action α)"

abbreviation the_Node_id :: "(var, ('n, 'a, 'd) cst) id ⇒ 'n" where
  "the_Node_id ident == the_Node (the_Cst ident)"

abbreviation the_Elem_id :: "(var, ('n, 'a, 'd) cst) id ⇒ 'd" where
  "the_Elem_id ident == the_Elem (the_Cst ident)"

abbreviation the_Action_id :: "(var, ('n, 'a, 'd) cst) id ⇒ 'a" where
  "the_Action_id ident == the_Action (the_Cst ident)"

abbreviation is_Elem_id :: "(var, ('n, 'a, 'd) cst) id ⇒ bool" where
  "is_Elem_id ident == is_Cst ident ∧ is_Elem (the_Cst ident)"

abbreviation is_Node_id :: "(var, ('n, 'a, 'd) cst) id ⇒ bool" where
  "is_Node_id ident == is_Cst ident ∧ is_Node (the_Cst ident)"

abbreviation is_Action_id :: "(var, ('n, 'a, 'd) cst) id ⇒ bool" where
  "is_Action_id ident == is_Cst ident ∧ is_Action (the_Cst ident)"
```

11.2 Forward may-analysis

```
locale analysis_BV_forward_may = finite_program_graph pg
  for pg :: "('n::finite,'a) program_graph" +
  fixes analysis_dom :: "'d set"
  fixes kill_set :: "('n,'a) edge ⇒ 'd set"
  fixes gen_set :: "('n,'a) edge ⇒ 'd set"
  fixes d_init :: "'d set"
  assumes "finite analysis_dom"
  assumes "d_init ⊆ analysis_dom"
  assumes "∀e. gen_set e ⊆ analysis_dom"
  assumes "∀e. kill_set e ⊆ analysis_dom"
begin
```

```
lemma finite_d_init: "finite d_init"
  by (meson analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def
      analysis_BV_forward_may_def rev_finite_subset)

interpretation LTS edges .

definition "S_hat" :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^_E⟦_⟧ _") where
  "S^_E⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
  assumes "d1 ⊆ d2"
  shows "S^_E⟦e⟧ d1 ⊆ S^_E⟦e⟧ d2"
  using assms unfolding S_hat_def by auto

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^_Es⟦_⟧ _") where
  "S^_Es⟦[]⟧ R = R" |
  "S^_Es⟦e # π⟧ R = S^_Es⟦π⟧ (S^_E⟦e⟧ R)"

lemma S_hat_edge_list_def2:
  "S^_Es⟦π⟧ R = foldl (λa b. S^_E⟦b⟧ a) R π"
proof (induction π arbitrary: R)
  case Nil
  then show ?case
    by simp
next
  case (Cons a π)
  then show ?case
    by simp
qed

lemma S_hat_edge_list_append[simp]:
  "S^_Es⟦xs @ ys⟧ R = S^_Es⟦ys⟧ (S^_Es⟦xs⟧ R)"
  unfolding S_hat_edge_list_def2 foldl_append by auto

lemma S_hat_edge_list_mono:
  assumes "R1 ⊆ R2"
  shows "S^_Es⟦π⟧ R1 ⊆ S^_Es⟦π⟧ R2"
proof(induction π rule: List.rev_induct)
  case Nil
  then show ?case
    using assms by auto
next
  case (snoc x xs)
  then show ?case
    using assms by (simp add: S_hat_mono)
qed

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^_P⟦_⟧ _") where
  "S^_P⟦π⟧ R = S^_Es⟦LTS.transition_list π⟧ R"

lemma S_hat_path_mono:
  assumes "R1 ⊆ R2"
  shows "S^_P⟦π⟧ R1 ⊆ S^_P⟦π⟧ R2"
  unfolding S_hat_path_def using assms S_hat_edge_list_mono by auto

fun ana_kill_edge_d :: "('n, 'a) edge ⇒ 'd ⇒ (pred, var, ('n, 'a, 'd) cst) clause" where
  "ana_kill_edge_d (q_o, α, q_s) d = kill⟨[Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]⟩ :- []."

definition ana_kill_edge :: "('n, 'a) edge ⇒ (pred, var, ('n, 'a, 'd) cst) clause set" where
  "ana_kill_edge e = ana_kill_edge_d e ` (kill_set e)"

lemma kill_set_eq_kill_set_inter_analysis_dom: "kill_set e = kill_set e ∩ analysis_dom"
```

```
  by (meson analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def
      analysis_BV_forward_may_def inf.orderE)

fun ana_gen_edge_d :: "('n, 'a) edge ⇒ 'd ⇒ (pred, var, ('n, 'a, 'd) cst) clause" where
  "ana_gen_edge_d (q_o, α, q_s) d = gen⟨[Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]⟩ :- []."

definition ana_gen_edge :: "('n, 'a) edge ⇒ (pred, var, ('n, 'a, 'd) cst) clause set" where
  "ana_gen_edge e = ana_gen_edge_d e ‘ (gen_set e)"

lemma gen_set_eq_gen_set_inter_analysis_dom: "gen_set e = gen_set e ∩ analysis_dom"
  by (meson analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def
      analysis_BV_forward_may_def inf.orderE)

definition ana_init :: "'d ⇒ (pred, var, ('n, 'a, 'd) cst) clause" where
  "ana_init d = init⟨[Cst_E d]⟩ :- []."

definition ana_anadom :: "'d ⇒ (pred, var, ('n, 'a, 'd) cst) clause" where
  "ana_anadom d = anadom⟨[Cst_E d]⟩ :- []."

definition ana_entry_node :: "(pred, var, ('n, 'a, 'd) cst) clause" where
  "ana_entry_node = may⟨[Cst_N start,u]⟩ :- [init[u]]."

fun ana_edge :: "('n, 'a) edge ⇒ (pred, var, ('n, 'a, 'd) cst) clause set" where
  "ana_edge (q_o, α, q_s) =
     {
        may⟨[Cst_N q_s, u]⟩ :-
          [
            may[Cst_N q_o, u],
            ¬kill[Cst_N q_o, Cst_A α, Cst_N q_s, u]
          ].
        ,
        may⟨[Cst_N q_s, u]⟩ :- [gen[Cst_N q_o, Cst_A α, Cst_N q_s, u]].
     }"

definition ana_must :: "'n ⇒ (pred, var, ('n, 'a, 'd) cst) clause" where
  "ana_must q = must⟨[Cst_N q,u]⟩ :- [¬may[Cst_N q,u], anadom[u]]."

lemma ana_must_meta_var:
  assumes "ϱ ⊨_cls must⟨[Cst_N q,u]⟩ :- [¬may[Cst_N q,u], anadom[u]]."
  shows "ϱ ⊨_cls must⟨[Cst_N q,v]⟩ :- [¬may[Cst_N q,v], anadom[v]]."
proof -
  define μ where "μ = Var(the_u := v)"
  have "ϱ ⊨_cls ((must⟨[Cst_N q,u]⟩ :- [¬may[Cst_N q,u], anadom[u]].) ·_cls μ)"
    using assms substitution_lemma by blast
  then show ?thesis
    unfolding μ_def by auto
qed

definition ana_pg_fw_may :: "(pred, var, ('n, 'a, 'd) cst) clause set" where
  "ana_pg_fw_may = ⋃(ana_edge ‘ edges)
              ∪ ana_init ‘ d_init
              ∪ ana_anadom ‘ analysis_dom
              ∪ ⋃(ana_kill_edge ‘ edges)
              ∪ ⋃(ana_gen_edge ‘ edges)
              ∪ ana_must ‘ UNIV
              ∪ {ana_entry_node}"

lemma ana_entry_node_meta_var:
  assumes "ϱ ⊨_cls may⟨[Cst_N start,u]⟩ :- [init[u]]."
  shows "ϱ ⊨_cls may⟨[Cst_N start,u]⟩ :- [init[u]]."
proof -
  define μ where "μ = Var(the_u := u)"
  have "ϱ ⊨_cls ((may⟨[Cst_N start,u]⟩ :- [init[u]].) ·_cls μ)"
```

```
    using assms substitution_lemma by blast
  then show ?thesis
    unfolding μ_def by auto
qed

definition summarizes_fw_may :: "(pred, ('n, 'a, 'd) cst) pred_val ⇒ bool" where
  "summarizes_fw_may ϱ ⟷
     (∀π d. π ∈ path_with_word_from start ⟶ d ∈ S^_P⟦π⟧ d_init ⟶
        ϱ ⊨_lh (may⟨[Cst_N (LTS.end_of π), Cst_E d]⟩.))"

lemma S_hat_path_append:
  assumes "length qs = length w"
  shows "S^_P⟦(qs @ [qnminus1, qn], w @ [α])⟧ d_init =
    S^_E⟦(qnminus1, α, qn)⟧ (S^_P⟦(qs @ [qnminus1], w)⟧ d_init)"
proof -
  have "S^_P⟦ (qs @ [qnminus1, qn], w @ [α])⟧ d_init =
        S^_Es⟦(transition_list (qs @ [qnminus1, qn], w @ [α]))⟧ d_init"
    unfolding S_hat_path_def by auto
  moreover
  have "S^_Es⟦(transition_list (qs @ [qnminus1, qn], w @ [α]))⟧ d_init =
        S^_Es⟦(transition_list (qs @ [qnminus1], w) @ [(qnminus1, α, qn)])⟧ d_init"
    using transition_list_reversed_simp[of qs w] assms
    by auto
  moreover
  have
    "... = S^_Es⟦[(qnminus1, α, qn)]⟧ (S_hat_edge_list (transition_list (qs @ [qnminus1], w)) d_init)"
    using S_hat_edge_list_append[of "transition_list (qs @ [qnminus1], w)" "[(qnminus1, α, qn)]" d_init]
    by auto
  moreover
  have "... = S^_E⟦(qnminus1, α, qn)⟧ (S^_P⟦ (qs @ [qnminus1], w)⟧ d_init)"
    unfolding S_hat_path_def by auto
  ultimately show ?thesis
    by blast
qed

lemma ana_pg_fw_may_stratified: "strat_wf s_BV ana_pg_fw_may"
  unfolding ana_pg_fw_may_def strat_wf_def ana_init_def ana_anadom_def ana_gen_edge_def
    ana_must_def ana_entry_node_def  ana_kill_edge_def by auto

lemma finite_ana_edge_edgeset: "finite (⋃ (ana_edge ‘ edges))"
proof -
  have "finite edges"
    using finite_program_graph_axioms finite_program_graph_def by blast
  moreover
  have "∀e ∈ edges. finite (ana_edge e)"
    by force
  ultimately
  show ?thesis
    by blast
qed

lemma finite_ana_kill_edgeset: "finite (⋃ (ana_kill_edge ‘ edges))"
proof -
  have "finite edges"
    using finite_program_graph_axioms finite_program_graph_def by blast
  moreover
  have "∀e ∈ edges. finite (ana_kill_edge e)"
    by (metis ana_kill_edge_def analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def
        analysis_BV_forward_may_def finite_Int finite_imageI kill_set_eq_kill_set_inter_analysis_dom)
  ultimately
  show ?thesis
    by blast
qed
```

```
lemma finite_ana_gen_edgeset: "finite (⋃ (ana_gen_edge ‘ edges))"
proof -
  have "finite edges"
    using finite_program_graph_axioms finite_program_graph_def by blast
  moreover
  have "∀e ∈ edges. finite (ana_gen_edge e)"
    by (metis ana_gen_edge_def analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def
        analysis_BV_forward_may_def finite_imageI rev_finite_subset)
  ultimately
  show ?thesis
    by blast
qed

lemma finite_ana_anadom_edgeset: "finite (ana_anadom ‘ analysis_dom)"
  by (meson analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def
      analysis_BV_forward_may_def finite_imageI)

lemma ana_pg_fw_may_finite: "finite ana_pg_fw_may"
  unfolding ana_pg_fw_may_def using finite_ana_edge_edgeset finite_d_init
    finite_ana_anadom_edgeset finite_ana_kill_edgeset finite_ana_gen_edgeset by auto

fun vars_lh :: "('p,'x,'e) lh ⇒ 'x set" where
  "vars_lh (p,ids) = vars_ids ids"

lemma not_kill:
  fixes ϱ :: "(pred, ('n, 'a, 'd) cst) pred_val"
  assumes "d ∉ kill_set(q_o, α, q_s)"
  assumes "ϱ ⊨_lst ana_pg_fw_may s_BV"
  shows "ϱ ⊨_rh ¬kill[Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]"
proof -
  have "finite ana_pg_fw_may"
    by (simp add: ana_pg_fw_may_finite)
  moreover
  have "ϱ ⊨_lst ana_pg_fw_may s_BV"
    using assms(2) by blast
  moreover
  have "strat_wf s_BV ana_pg_fw_may"
    by (simp add: ana_pg_fw_may_stratified)
  moreover
  have "∀c∈ana_pg_fw_may.
          ∀σ'.
            (the_lh c ·_vlh σ') = ((the_kill, [Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]))
              ⟶ ¬ ⟦the_rhs c⟧_rhs ϱ σ'"
  proof (rule, rule, rule)
    fix c σ'
    assume c_dl: "c ∈ (ana_pg_fw_may)"
    assume "(the_lh c ·_vlh σ') = ((the_kill, [Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]))"
    moreover
    from c_dl have "c ∈ (ana_pg_fw_may)"
      by auto
    ultimately
    show "¬ ⟦the_rhs c⟧_rhs ϱ σ'"
      unfolding ana_pg_fw_may_def ana_init_def ana_anadom_def ana_kill_edge_def
        ana_gen_edge_def ana_must_def ana_entry_node_def using assms(1) by fastforce
  qed
  then have "¬ (∃c∈ana_pg_fw_may.
                  lh_consequence ϱ c (the_kill, [Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]))"
    using lh_consequence_def by metis
  ultimately
  show "ϱ ⊨_rh ¬kill [Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]"
    using solves_NegLit_least[of ana_pg_fw_may ϱ s_BV "[Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]" the_kill]
```

```
    by auto
qed

lemma S_hat_edge_list_subset_analysis_dom:
  assumes "d ∈ Ŝ_Es⟦π⟧ d_init"
  shows "d ∈ analysis_dom"
  using assms
proof(induction π rule: List.rev_induct)
  case Nil
  then show ?case
    by (metis S_hat_edge_list.simps(1) analysis_BV_forward_may.axioms(2)
        analysis_BV_forward_may_axioms analysis_BV_forward_may_axioms_def subsetD)

next
  case (snoc e π)
  have "gen_set e ∩ analysis_dom ⊆ analysis_dom"
    by fastforce
  then show ?case
    using S_hat_def gen_set_eq_gen_set_inter_analysis_dom snoc by auto
qed

lemma S_hat_path_subset_analysis_dom:
  assumes "d ∈ Ŝ_P⟦(ss,w)⟧ d_init"
  shows "d ∈ analysis_dom"
  using assms S_hat_path_def S_hat_edge_list_subset_analysis_dom by auto

lemma S_hat_path_last:
  assumes "length qs = length w"
  shows "Ŝ_P⟦(qs @ [qnminus1, qn], w @ [α])⟧ d_init =
         Ŝ_E⟦(qnminus1, α, qn)⟧ (Ŝ_P⟦(qs @ [qnminus1], w)⟧ d_init)"
  using assms transition_list_reversed_simp unfolding S_hat_path_def by force

lemma gen_sound:
  assumes "d ∈ gen_set (q, α, q')"
  assumes "(q, α, q') ∈ edges"
  assumes "ϱ ⊨_lst ana_pg_fw_may s_BV"
  shows "ϱ ⊨_cls gen⟨[Cst_N q, Cst_A α, Cst_N q', Cst_E d]⟩ :- [] ."
proof -
  have "gen⟨[Cst_N q, Cst_A α, Cst_N q', Cst_E d]⟩ :- [] . ∈ ana_pg_fw_may"
    using assms(1,2) unfolding ana_pg_fw_may_def ana_gen_edge_def by fastforce
  then show "?thesis"
    using ‹gen⟨[Cst_N q, Cst_A α, Cst_N q', Cst_E d]⟩ :- [] . ∈ ana_pg_fw_may› assms(3)
      least_solution_def solves_program_def by blast
qed

lemma sound_ana_pg_fw_may':
  assumes "(ss,w) ∈ path_with_word_from start"
  assumes "d ∈ Ŝ_P⟦(ss,w)⟧ d_init"
  assumes "ϱ ⊨_lst ana_pg_fw_may s_BV"
  shows "ϱ ⊨_lh may⟨[Cst_N (LTS.end_of (ss, w)), Cst_E d]⟩."
  using assms
proof (induction rule: LTS.path_with_word_from_induct_reverse[OF assms(1)])
  case (1 s)
  have ϱ_soultion: "ϱ ⊨_dl ana_pg_fw_may"
    using assms(3) unfolding least_solution_def by auto

  from 1(1) have start_end: "LTS.end_of ([s], []) = start"
    by (simp add: LTS.end_of_def LTS.start_of_def)

  from 1 have "Ŝ_P⟦([s], [])⟧ d_init = d_init"
    unfolding S_hat_path_def by auto
  then have "d ∈ d_init"
    using 1(2) by auto
```

```
  then have "ϱ ⊨cls init⟨[Cst_E d]⟩ :- [] ."
    using ϱ_soultion unfolding ana_pg_fw_may_def ana_init_def solves_program_def by auto
  moreover
  have "ϱ ⊨cls may⟨[Cst_N start, u]⟩ :- [init[u]]."
    by (metis Un_insert_right ana_entry_node_def analysis_BV_forward_may.ana_pg_fw_may_def
        analysis_BV_forward_may_axioms ϱ_soultion insertI1 solves_program_def)
  then have "ϱ ⊨cls may⟨[Cst_N start, Cst_E d]⟩ :- [init[Cst_E d]]."
    using ana_entry_node_meta_var by blast
  ultimately have "ϱ ⊨cls may⟨[Cst_N start, Cst_E d]⟩ :- [] ."
    using prop_inf_only_from_cls_cls_to_cls by metis
  then show ?case
    using start_end solves_lh_lh by metis
next
  case (2 qs qnminus1 w α qn)
  then have len: "length qs = length w"
    using path_with_word_lengths by blast

  have "d ∈ S^_E⟦(qnminus1, α, qn)⟧ (S^_P⟦(qs @ [qnminus1], w)⟧ d_init)"
    using "2.prems"(2) S_hat_path_last len by blast
  then have "d ∈ (S^_P⟦(qs @ [qnminus1], w)⟧ d_init) - kill_set (qnminus1, α, qn) ∨
             d ∈ gen_set (qnminus1, α, qn)"
    unfolding S_hat_def by simp
  then show ?case
  proof
    assume dSnotkill: "d ∈ (S^_P⟦(qs @ [qnminus1], w)⟧ d_init) - kill_set (qnminus1, α, qn)"
    then have "ϱ ⊨lh may⟨[Cst_N qnminus1, Cst_E d]⟩."
      using 2(1,3,6) by (auto simp add: LTS.end_of_def)
    moreover
    from dSnotkill have "ϱ ⊨rh ¬kill [Cst_N qnminus1, Cst_A α, Cst_N qn, Cst_E d]"
      using not_kill[of d qnminus1 α qn ϱ] 2(6) by auto
    moreover
    have "ϱ ⊨cls may⟨[Cst_N qn, u]⟩ :- [may [Cst_N qnminus1, u],
                                        ¬kill [Cst_N qnminus1, Cst_A α, Cst_N qn, u]]."
      using 2(6) "2.hyps"(2)
      by (force simp add: ana_pg_fw_may_def solves_program_def least_solution_def)
    then have "ϱ ⊨cls may⟨[Cst_N qn, Cst_E d]⟩ :- [may [Cst_N qnminus1, Cst_E d],
                                                  ¬kill [Cst_N qnminus1, Cst_A α, Cst_N qn, Cst_E d]]."
      using substitution_lemma[of ϱ _ "λu. Cst_E d"] by force
    ultimately
    have "ϱ ⊨lh may⟨[Cst_N qn, Cst_E d]⟩."
      by (metis (no_types, lifting) Cons_eq_appendI prop_inf_last_from_cls_rh_to_cls modus_ponens_rh

          self_append_conv2)
    then show "?case"
      by (auto simp add: LTS.end_of_def)
  next
    assume d_gen: "d ∈ gen_set (qnminus1, α, qn)"

    have "∀c∈ana_edge (qnminus1, α, qn). ϱ ⊨cls c"
      using 2(6) "2.hyps"(2) unfolding ana_pg_fw_may_def solves_program_def least_solution_def
      by blast
    then have "ϱ ⊨cls may⟨[Cst_N qn, u]⟩ :- [gen [Cst_N qnminus1, Cst_A α, Cst_N qn, u]] ."
      by auto
    then have "ϱ ⊨cls may⟨[Cst_N qn, Cst_E d]⟩ :- [gen [Cst_N qnminus1, Cst_A α, Cst_N qn, Cst_E d]] ."
      using substitution_lemma[of ϱ _ "λu. Cst_E d" ]
      by force
    moreover
    have "ϱ ⊨cls gen⟨[Cst_N qnminus1, Cst_A α, Cst_N qn, Cst_E d]⟩ :- [] ."
      using d_gen "2.hyps"(2) 2(6) gen_sound by auto
    ultimately
    have "ϱ ⊨lh may⟨[Cst_N qn, Cst_E d]⟩."
      by (meson modus_ponens_rh solves_lh_lh)
    then show ?case
```

```
    by (auto simp add: LTS.end_of_def)
  qed
qed

theorem sound_ana_pg_fw_may:
  assumes "ϱ ⊨_lst ana_pg_fw_may s_BV"
  shows "summarizes_fw_may ϱ"
  using sound_ana_pg_fw_may' assms unfolding summarizes_fw_may_def by (cases pg) fastforce

end
```

11.3 Backward may-analysis

```
locale analysis_BV_backward_may = finite_program_graph pg
  for pg :: "('n::finite,'a) program_graph" +
  fixes analysis_dom :: "'d set"
  fixes kill_set :: "('n,'a) edge ⇒ 'd set"
  fixes gen_set :: "('n,'a) edge ⇒ 'd set"
  fixes d_init :: "'d set"
  assumes "finite edges"
  assumes "finite analysis_dom"
  assumes "d_init ⊆ analysis_dom"
  assumes "∀e. gen_set e ⊆ analysis_dom"
  assumes "∀e. kill_set e ⊆ analysis_dom"
begin



interpretation LTS edges .

definition S_hat :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^_E⟦_⟧ _") where
  "S^_E⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
  assumes "R1 ⊆ R2"
  shows "S^_E⟦e⟧ R1 ⊆ S^_E⟦e⟧ R2"
  using assms unfolding S_hat_def by auto

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^_Es⟦_⟧ _") where
  "S^_Es⟦[]⟧ R = R" |
  "S^_Es⟦(e # π)⟧ R = S^_E⟦e⟧ (S^_Es⟦π⟧ R)"

lemma S_hat_edge_list_def2:
  "S^_Es⟦π⟧ R = foldr S_hat π R"
proof (induction π arbitrary: R)
  case Nil
  then show ?case
    by simp
next
  case (Cons a π)
  then show ?case
    by simp
qed

lemma S_hat_edge_list_append[simp]:
  "S^_Es⟦(xs @ ys)⟧ R = S^_Es⟦xs⟧ (S^_Es⟦ys⟧ R)"
  unfolding S_hat_edge_list_def2 foldl_append by auto

lemma S_hat_edge_list_mono:
  assumes "d1 ⊆ d2"
  shows "S^_Es⟦π⟧ d1 ⊆ S^_Es⟦π⟧ d2"
proof(induction π)
  case Nil
  then show ?case
```

```
    using assms by auto
next
  case (Cons x xs)
  then show ?case
    using assms by (simp add: S_hat_mono)
qed

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^_P⟦_⟧ _") where
  "S^_P⟦π⟧ R = S^_Es⟦(transition_list π)⟧ R"

definition summarizes_bw_may :: "(pred, ('n, 'a, 'd) cst) pred_val ⇒ bool" where
  "summarizes_bw_may ϱ ⟷ (∀π d. π ∈ path_with_word_to end ⟶ d ∈ S^_P⟦π⟧ d_init ⟶
                              ϱ ⊨_lh may⟨[Cst_N (start_of π), Cst_E d]⟩.)"

lemma kill_subs_analysis_dom: "(kill_set (rev_edge e)) ⊆ analysis_dom"
  by (meson analysis_BV_backward_may_axioms analysis_BV_backward_may_axioms_def
      analysis_BV_backward_may_def)

lemma gen_subs_analysis_dom: "(gen_set (rev_edge e)) ⊆ analysis_dom"
  by (meson analysis_BV_backward_may.axioms(2) analysis_BV_backward_may_axioms
      analysis_BV_backward_may_axioms_def)

interpretation fw_may: analysis_BV_forward_may
  pg_rev analysis_dom "λe. (kill_set (rev_edge e))" "(λe. gen_set (rev_edge e))" d_init
  using analysis_BV_forward_may_def finite_pg_rev analysis_BV_backward_may_axioms
    analysis_BV_backward_may_def
  by (metis (no_types, lifting) analysis_BV_backward_may_axioms_def
      analysis_BV_forward_may_axioms_def finite_program_graph_def program_graph.edges_def)

abbreviation ana_pg_bw_may where
  "ana_pg_bw_may == fw_may.ana_pg_fw_may"

lemma rev_end_is_start:
  assumes "ss ≠ []"
  assumes "end_of (ss, w) = end"
  shows "start_of (rev ss, rev w) = fw_may.start"
  using assms
  unfolding end_of_def LTS.start_of_def fw_may.start_def pg_rev_def fw_may.start_def
  using hd_rev by (metis fw_may.start_def fst_conv pg_rev_def snd_conv)

lemma S_hat_edge_list_forward_backward:
  "S^_Es⟦ss⟧ d_init = fw_may.S_hat_edge_list (rev_edge_list ss) d_init"
proof (induction ss)
  case Nil
  then show ?case
    unfolding rev_edge_list_def by auto
next
  case (Cons e es)
  have "S^_Es⟦e # es⟧ d_init = S^_E⟦e⟧ S^_Es⟦es⟧ d_init"
    by simp
  also
  have "... = fw_may.S_hat (rev_edge e) (foldr fw_may.S_hat (map rev_edge es) d_init)"
    unfolding foldr_conv_foldl fw_may.S_hat_edge_list_def2[symmetric]
    unfolding rev_edge_list_def[symmetric] fw_may.S_hat_def S_hat_def
    using Cons by simp
  also
  have "... = fw_may.S_hat_edge_list (rev_edge_list (e # es)) d_init"
    unfolding rev_edge_list_def fw_may.S_hat_edge_list_def2 foldl_conv_foldr
    by simp
  finally
  show ?case
    by metis
qed
```

```
lemma S_hat_path_forward_backward:
  assumes "(ss,w) ∈ path_with_word"
  shows "S^_P⟦(ss, w)⟧ d_init = fw_may.S_hat_path (rev ss, rev w) d_init"
  using S_hat_edge_list_forward_backward unfolding S_hat_path_def fw_may.S_hat_path_def
  by (metis transition_list_rev_edge_list assms)

lemma summarizes_bw_may_forward_backward':
  assumes "(ss,w) ∈ path_with_word"
  assumes "end_of (ss,w) = end"
  assumes "d ∈ S^_P⟦(ss,w)⟧ d_init"
  assumes "fw_may.summarizes_fw_may ϱ"
  shows "ϱ ⊨_lh may⟨[Cst_N (start_of (ss, w)), Cst_E d]⟩."
proof -
  have rev_in_edges: "(rev ss, rev w) ∈ LTS.path_with_word fw_may.edges"
    using assms(1) rev_path_in_rev_pg[of ss w] fw_may.edges_def pg_rev_def by auto
  moreover
  have "LTS.start_of (rev ss, rev w) = fw_may.start"
    using assms(1,2) rev_end_is_start by (metis LTS.path_with_word_not_empty)
  moreover
  have "d ∈ fw_may.S_hat_path (rev ss, rev w) d_init"
    using assms(3)
    using assms(1) S_hat_path_forward_backward by auto
  ultimately
  have "ϱ ⊨_lh may⟨[Cst_N (LTS.end_of (rev ss, rev w)), Cst_E d]⟩."
    using assms(4) fw_may.summarizes_fw_may_def by blast
  then show ?thesis
    by (metis LTS.end_of_def LTS.start_of_def fst_conv hd_rev rev_rev_ident)
qed

lemma summarizes_dl_BV_forward_backward:
  assumes "fw_may.summarizes_fw_may ϱ"
  shows "summarizes_bw_may ϱ"
  unfolding summarizes_bw_may_def
proof(rule; rule ; rule; rule)
  fix π d
  assume "π ∈ {π ∈ path_with_word. LTS.end_of π = end}"
  moreover
  assume "d ∈ S^_P⟦π⟧ d_init"
  ultimately
  show "ϱ ⊨_lh may⟨[Cst_N (LTS.start_of π), Cst_E d]⟩."
    using summarizes_bw_may_forward_backward'[of "fst π" "snd π" d ϱ] using assms by auto
qed

theorem sound_ana_pg_bw_may:
  assumes "ϱ ⊨_lst ana_pg_bw_may s_BV"
  shows "summarizes_bw_may ϱ"
  using assms fw_may.sound_ana_pg_fw_may[of ϱ] summarizes_dl_BV_forward_backward by metis

end
```

11.4 Forward must-analysis

```
locale analysis_BV_forward_must = finite_program_graph pg
  for pg :: "('n::finite,'a) program_graph" +
  fixes analysis_dom :: "'d set"
  fixes kill_set :: "('n,'a) edge ⇒ 'd set"
  fixes gen_set :: "('n,'a) edge ⇒ 'd set"
  fixes d_init :: "'d set"
  assumes "finite analysis_dom"
  assumes "d_init ⊆ analysis_dom"
begin

lemma finite_d_init: "finite d_init"
```

```
  by (meson analysis_BV_forward_must.axioms(2) analysis_BV_forward_must_axioms
      analysis_BV_forward_must_axioms_def rev_finite_subset)

interpretation LTS edges .

definition S_hat :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^_E⟦_⟧ _") where
  "S^_E⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
  assumes "R1 ⊆ R2"
  shows "S^_E⟦e⟧ R1 ⊆ S^_E⟦e⟧ R2"
  using assms unfolding S_hat_def by auto

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^_Es⟦_⟧ _") where
  "S^_Es⟦[]⟧ R = R" |
  "S^_Es⟦(e # π)⟧ R = S^_Es⟦π⟧ (S^_E⟦e⟧ R)"

lemma S_hat_edge_list_def2:
  "S^_Es⟦π⟧ R = foldl (λa b. S^_E⟦b⟧ a) R π"
proof (induction π arbitrary: R)
  case Nil
  then show ?case
    by simp
next
  case (Cons a π)
  then show ?case
    by simp
qed

lemma S_hat_edge_list_append[simp]:
  "S^_Es⟦(xs @ ys)⟧ R = S^_Es⟦ys⟧ (S^_Es⟦xs⟧ R)"
  unfolding S_hat_edge_list_def2 foldl_append by auto

lemma S_hat_edge_list_mono:
  assumes "R1 ⊆ R2"
  shows "S^_Es⟦π⟧ R1 ⊆ S^_Es⟦π⟧ R2"
proof(induction π rule: List.rev_induct)
  case Nil
  then show ?case
    using assms by auto
next
  case (snoc x xs)
  then show ?case
    using assms by (simp add: S_hat_mono)
qed

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^_P⟦_⟧ _") where
  "S^_P⟦π⟧ R = S^_Es⟦LTS.transition_list π⟧ R"

lemma S_hat_path_mono:
  assumes "R1 ⊆ R2"
  shows "S^_P⟦π⟧ R1 ⊆ S^_P⟦π⟧ R2"
  unfolding S_hat_path_def using assms S_hat_edge_list_mono by auto

definition summarizes_fw_must :: "(pred, ('n, 'a, 'd) cst) pred_val ⇒ bool" where
   "summarizes_fw_must ϱ ⟷
     (∀q d.
         ϱ ⊨_lh must⟨[q, d]⟩. ⟶
           (∀π. π ∈ path_with_word ⟶
                start_of π = start ⟶
                end_of π = the_Node_id q ⟶
                (the_Elem_id d) ∈ S^_P⟦π⟧ d_init))"
```

```
interpretation fw_may: analysis_BV_forward_may
  pg analysis_dom "λe. analysis_dom - (kill_set e)" "(λe. analysis_dom - gen_set e)"
  "analysis_dom - d_init"
  using analysis_BV_forward_may.intro[of pg] analysis_BV_forward_must_def[of pg]
    analysis_BV_forward_may_axioms_def analysis_BV_forward_must_axioms
    analysis_BV_forward_must_axioms_def by (metis Diff_subset)

abbreviation ana_pg_fw_must where
  "ana_pg_fw_must == fw_may.ana_pg_fw_may"

lemma opposite_lemma:
  assumes "d ∈ analysis_dom"
  assumes "¬d ∈ fw_may.S_hat_edge_list π (analysis_dom - d_init)"
  shows "d ∈ S^Es⟦π⟧ d_init"
  using assms proof (induction π rule: List.rev_induct)
  case Nil
  then show ?case
    by auto
next
  case (snoc x xs)
  then show ?case
    unfolding fw_may.S_hat_edge_list_def2
    by (simp add: S_hat_def fw_may.S_hat_def)
qed

lemma opposite_lemma_path:
  assumes "d ∈ analysis_dom"
  assumes "¬d ∈ fw_may.S_hat_path π (analysis_dom - d_init)"
  shows "d ∈ S^P⟦π⟧ d_init"
  using S_hat_path_def fw_may.S_hat_path_def assms opposite_lemma by metis

lemma the_must_only_ana_must: "the_must ∉ preds_dl (ana_pg_fw_must - (fw_may.ana_must ‘ UNIV))"
  unfolding fw_may.ana_pg_fw_may_def preds_dl_def preds_dl_def fw_may.ana_init_def
    preds_dl_def fw_may.ana_anadom_def preds_dl_def fw_may.ana_kill_edge_def preds_dl_def
    fw_may.ana_gen_edge_def fw_may.ana_entry_node_def by auto

lemma agree_off_rh:
  assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
  assumes "p' ∉ preds_rh rh"
  shows "⟦rh⟧rh ϱ' σ ⟷ ⟦rh⟧rh ϱ σ"
  using assms proof (cases rh)
  case (Eql a a')
  then show ?thesis
    by auto
next
  case (Neql a a')
  then show ?thesis
    by auto
next
  case (PosLit p ids)
  then show ?thesis
    using assms by auto
next
  case (NegLit p ids)
  then show ?thesis
    using assms by auto
qed

definition preds_rhs where
  "preds_rhs rhs = ⋃(preds_rh ‘ set rhs)"

lemma preds_cls_preds_rhs: "preds_cls (Cls p ids rhs) = {p} ∪ preds_rhs rhs"
  by (simp add: preds_rhs_def)
```

```
lemma agree_off_rhs:
  assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
  assumes "p' ∉ preds_rhs rhs"
  shows "⟦rhs⟧rhs ϱ' σ ⟷ ⟦rhs⟧rhs ϱ σ"
  using assms agree_off_rh[of p' ϱ' ϱ _ σ] unfolding preds_rhs_def by auto

lemma agree_off_lh:
  assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
  assumes "p' ∉ preds_lh lh"
  shows "⟦lh⟧lh ϱ' σ ⟷ ⟦lh⟧lh ϱ σ"
proof (cases lh)
  case (Pair p ids)
  then show ?thesis
    using assms by auto
qed

lemma agree_off_cls:
  assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
  assumes "p' ∉ preds_cls c"
  shows "⟦c⟧cls ϱ' σ ⟷ ⟦c⟧cls ϱ σ"
proof (cases c)
  case (Cls p ids rhs)
  show ?thesis
  proof
    assume "⟦c⟧cls ϱ' σ"
    show "⟦c⟧cls ϱ σ"
      unfolding Cls meaning_cls.simps
    proof
      assume "⟦rhs⟧rhs ϱ σ"
      then have "⟦rhs⟧rhs ϱ' σ"
        using agree_off_rhs
        by (metis Cls assms(1) assms(2) clause.simps(6) insert_iff preds_rhs_def)
      then show"⟦(p, ids)⟧lh ϱ σ"
        using Cls ‹⟦c⟧cls ϱ' σ› assms(1) assms(2) by auto
    qed
  next
    assume "⟦c⟧cls ϱ σ"
    show "⟦c⟧cls ϱ' σ"
    unfolding Cls meaning_cls.simps
    proof
      assume "⟦rhs⟧rhs ϱ' σ"
      then have "⟦rhs⟧rhs ϱ σ"
        using agree_off_rhs
        by (metis Cls assms(1) assms(2) clause.simps(6) insert_iff preds_rhs_def)
      then show"⟦(p, ids)⟧lh ϱ' σ"
        using Cls ‹⟦c⟧cls ϱ σ› assms(1) assms(2) by auto
    qed
  qed
qed

lemma agree_off_solves_cls:
  assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
  assumes "p' ∉ preds_cls c"
  shows "ϱ' ⊨cls c ⟷ ϱ ⊨cls c"
proof (cases c)
  case (Cls p ids rhs)
  then show ?thesis
    by (metis (mono_tags, opaque_lifting) agree_off_cls assms solves_cls_def)
qed

lemma agree_off_dl:
  assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
```

```
  assumes "p' ∉ preds_dl dl"
  shows "ϱ' ⊨dl dl ⟷ ϱ ⊨dl dl"
proof
  assume "ϱ' ⊨dl dl"
  show "ϱ ⊨dl dl"
    unfolding solves_program_def
  proof
    fix c
    assume "c ∈ dl"
    have "p' ∉ preds_cls c"
      using ‹c ∈ dl› assms(2) preds_dl_def by fastforce
    show "ϱ ⊨cls c"
      by (metis ‹ϱ' ⊨dl dl› ‹c ∈ dl› ‹p' ∉ preds_cls c› agree_off_solves_cls assms(1)
          solves_program_def)
  qed
next
  assume "ϱ ⊨dl dl"
  show "ϱ' ⊨dl dl"
    unfolding solves_program_def
  proof
    fix c
    assume "c ∈ dl"
    have "p' ∉ preds_cls c"
      using ‹c ∈ dl› assms(2) preds_dl_def by fastforce
    show "ϱ' ⊨cls c"
      by (metis ‹ϱ ⊨dl dl› ‹c ∈ dl› ‹p' ∉ preds_cls c› agree_off_solves_cls assms(1)
          solves_program_def)
  qed
qed

lemma is_Cst_if_init:
  assumes "ϱ ⊨lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨lh init⟨[d]⟩."
  shows "is_Cst d"
proof (rule ccontr)
  assume "¬ is_Cst d"
  then have du: "d = u"
    by (metis (full_types) id.disc(1) id.exhaust_disc id.expand var.exhaust)
  then have "⟦init⟨[d]⟩.⟧lh ϱ (λx. Action undefined)"
    using assms
    by auto
  then have "ϱ ⊨lh init⟨[d ·vid (λx. Action undefined)]⟩."
    using solves_lh_substv_lh_if_meaning_lh[of "init⟨[d]⟩." ϱ "(λx. Action undefined)"] du by auto
  moreover
  have "is_Cst (Cst_A undefined)"
    by auto
  moreover
  have "is_Cst (d ·vid (λx. Action undefined))"
    by (metis id.disc(4) substv_id.elims)
  ultimately
  have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (init⟨[Cst_A undefined]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV] du
    by (simp add: assms(1) fw_may.ana_pg_fw_may_finite fw_may.ana_pg_fw_may_stratified)
  then show False
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_init_def fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
      fw_may.ana_must_def by auto
qed

lemma is_Cst_if_anadom:
  assumes "ϱ ⊨lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨lh anadom⟨[d]⟩."
  shows "is_Cst d"
```

```
proof (rule ccontr)
  assume "¬ is_Cst d"
  then have du: "d = u"
    by (metis (full_types) id.disc(1) id.exhaust_disc id.expand var.exhaust)
  then have "⟦anadom⟨[d]⟩.⟧lh ϱ (λx. Action undefined)"
    using assms
    by auto
  then have "ϱ ⊨lh anadom⟨[d ·vid (λx. Action undefined)]⟩."
    using solves_lh_substv_lh_if_meaning_lh[of "anadom⟨[d]⟩." ϱ "(λx. Action undefined)"] du by auto
  moreover
  have "is_Cst (CstA undefined)"
    by auto
  moreover
  have "is_Cst (d ·vid (λx. Action undefined))"
    by (metis id.disc(4) substv_id.elims)
  ultimately
  have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (anadom⟨[CstA undefined]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV] du
    by (simp add: assms(1) fw_may.ana_pg_fw_may_finite fw_may.ana_pg_fw_may_stratified)
  then show False
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_init_def fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
      fw_may.ana_must_def by auto
qed

lemma if_init:
  assumes "ϱ ⊨lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨lh init⟨[d]⟩."
  shows "is_Elemid d ∧ the_Elemid d ∈ (analysis_dom - d_init)"
proof -
  have d_Cst: "is_Cst d"
    using assms(1) assms(2) is_Cst_if_init by blast

  from assms(1,2) d_Cst have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (init⟨[d]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV "[d]"] fw_may.ana_pg_fw_may_finite
      fw_may.ana_pg_fw_may_stratified by fastforce

  then obtain c where
    "c ∈ ana_pg_fw_must"
    "lh_consequence ϱ c (init⟨[d]⟩.)"
    by auto
  from this have "∃d' ∈ (analysis_dom - d_init). c = init⟨[CstE d']⟩ :- []."
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
      fw_may.ana_must_def fw_may.ana_init_def by auto
  then obtain d' where "d' ∈ analysis_dom - d_init ∧ c = init⟨[CstE d']⟩ :- []."
    by auto
  have "lh_consequence ϱ (init⟨[CstE d']⟩ :- [].) (init⟨[d]⟩.)"
    using ‹d' ∈ analysis_dom - d_init ∧ c = init⟨[CstE d']⟩ :- [] .›
      ‹lh_consequence ϱ c init⟨[d]⟩.› by fastforce
  then have "∃σ. (init⟨[CstE d']⟩. ·vlh σ) = init⟨[d]⟩."
    unfolding lh_consequence_def by auto
  then obtain σ where σ_p:
    "(init⟨[CstE d']⟩. ·vlh σ) = init⟨[d]⟩."
    by auto
  then have "d = CstE d'"
    by auto
  then have "the_Elemid d ∈ analysis_dom - d_init ∧ is_Elemid d"
    using ‹d' ∈ analysis_dom - d_init ∧ c = init⟨[CstE d']⟩ :- [] .› by auto
  then show ?thesis
    by auto
qed
```

```
lemma if_anadom:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh anadom⟨[d]⟩."
  shows "is_Elem_id d ∧ the_Elem_id d ∈ analysis_dom"
proof -
  have d_Cst: "is_Cst d"
    using assms(1) assms(2) is_Cst_if_anadom by blast

  from assms(1,2) d_Cst have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (anadom⟨[d]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV "[d]"] fw_may.ana_pg_fw_may_finite
      fw_may.ana_pg_fw_may_stratified by fastforce
  then obtain c where
    "c ∈ ana_pg_fw_must"
    "lh_consequence ϱ c (anadom⟨[d]⟩.)"
    by auto
  from this have "∃d' ∈ analysis_dom. c = anadom⟨[Cst_E d']⟩ :- []."
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
      fw_may.ana_must_def fw_may.ana_init_def by auto
  then obtain d' where "d' ∈ analysis_dom ∧ c = anadom⟨[Cst_E d']⟩ :- []."
    by auto
  have "lh_consequence ϱ (anadom⟨[Cst_E d']⟩ :- [].) (anadom⟨[d]⟩.)"
    using ‹d' ∈ analysis_dom ∧ c = anadom⟨[Cst_E d']⟩ :- [] .›
      ‹lh_consequence ϱ c anadom⟨[d]⟩.› by fastforce
  then have "∃σ. (anadom⟨[Cst_E d']⟩. ·_vlh σ) = anadom⟨[d]⟩."
    unfolding lh_consequence_def by auto
  then obtain σ where σ_p:
    "(anadom⟨[Cst_E d']⟩. ·_vlh σ) = anadom⟨[d]⟩."
    by auto
  then have "d = Cst_E d'"
    by auto
  then have "the_Elem_id d ∈ analysis_dom ∧ is_Elem_id d"
    using ‹d' ∈ analysis_dom ∧ c = anadom⟨[Cst_E d']⟩ :- [] .› by auto
  then show ?thesis
    by auto
qed

lemma is_elem_if_init:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh init⟨[Cst d]⟩."
  shows "is_Elem d"
  by (metis if_init assms id.sel(2))

lemma not_init_node:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "¬ϱ ⊨_lh init⟨[Cst_N q]⟩."
  by (metis if_init assms cst.collapse(2) cst.disc(1) cst.disc(2) id.sel(2))

lemma not_init_action:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "¬ϱ ⊨_lh init⟨[Cst_A q]⟩."
  by (metis assms cst.collapse(2) cst.simps(9) id.sel(2) if_init)

lemma in_analysis_dom_if_init':
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh init⟨[Cst_E d]⟩."
  shows "d ∈ analysis_dom"
  using assms if_init by force

lemma in_analysis_dom_if_init:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh init⟨[d]⟩."
  shows "the_Elem_id d ∈ analysis_dom"
```

```
  using assms if_init by force

lemma not_anadom_node:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "¬ϱ ⊨_lh anadom⟨[Cst_N q]⟩."
  by (metis assms cst.collapse(2) cst.disc(1) cst.disc(2) id.sel(2) if_anadom)

lemma not_anadom_action:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "¬ϱ ⊨_lh anadom⟨[Cst_A q]⟩."
  by (metis assms cst.collapse(2) cst.simps(9) id.sel(2) if_anadom)

lemma in_analysis_dom_if_anadom':
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh anadom⟨[Cst_E d]⟩."
  shows "d ∈ analysis_dom"
  using assms if_anadom by force

lemma in_analysis_dom_if_anadom:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh anadom⟨[d]⟩."
  shows "the_Elem_id d ∈ analysis_dom ∧ is_Elem_id d"
  using assms if_anadom by force

lemma must_fst_id_is_Cst:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q,d]⟩."
  shows "is_Cst q"
proof (rule ccontr)
  assume "¬ is_Cst q"
  then have qu: "q = u"
    by (metis (full_types) id.disc(1) id.exhaust_disc id.expand var.exhaust)
  then have "⟦must⟨[q,d]⟩.⟧_lh ϱ (λx. Action undefined)"
    using assms
    by auto
  then have "ϱ ⊨_lh must⟨[Cst_A undefined, d ·_vid (λx. Action undefined)]⟩."
    using solves_lh_substv_lh_if_meaning_lh[of "must⟨[q, d]⟩." ϱ "(λx. Action undefined)"] qu by auto
  moreover
  have "is_Cst (Cst_A undefined)"
    by auto
  moreover
  have "is_Cst (d ·_vid (λx. Action undefined))"
    by (metis id.disc(4) substv_id.elims)
  ultimately
  have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (must⟨[Cst_A undefined, d ·_vid (λx. Action undefined)]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV "[Cst_A undefined, d ·_vid (λx. Action undefined)]"
        the_must]
    by (simp add: assms(1) fw_may.ana_pg_fw_may_finite fw_may.ana_pg_fw_may_stratified)
  then show False
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_init_def fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
      fw_may.ana_must_def by auto
qed

lemma must_snd_id_is_Cst:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q,d]⟩."
  shows "is_Cst d"
proof (rule ccontr)
  assume "¬ is_Cst d"
  then have du: "d = u"
    by (metis (full_types) id.disc(1) id.exhaust_disc id.expand var.exhaust)
  then have "⟦must⟨[q,d]⟩.⟧_lh ϱ (λx. Action undefined)"
```

```
    using assms
    by auto
  then have "ϱ ⊨lh must⟨[q ·vid (λx. Action undefined), Cst_A undefined]⟩."
    using solves_lh_substv_lh_if_meaning_lh[of "must⟨[q, d]⟩." ϱ "(λx. Action undefined)"] du by auto
  moreover
  have "is_Cst (Cst_A undefined)"
    by auto
  moreover
  have "is_Cst (q ·vid (λx. Action undefined))"
    by (metis id.disc(4) substv_id.elims)
  ultimately
  have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (must⟨[q ·vid (λx. Action undefined), Cst_A undefined]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV "[q ·vid (λx. Action undefined), Cst_A undefined]"
        the_must]
    by (simp add: assms(1) fw_may.ana_pg_fw_may_finite fw_may.ana_pg_fw_may_stratified)
    then obtain c where c_p:
    "c ∈ ana_pg_fw_must"
    "lh_consequence ϱ c (must⟨[q ·vid (λx. Action undefined), Cst_A undefined]⟩.)"
    by auto
  from this have "∃q'. c = must⟨[Cst_N q',u]⟩ :- [¬may[Cst_N q',u], anadom[u]]."
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_init_def fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
      fw_may.ana_must_def by auto
  then obtain q' where "c = must⟨[Cst_N q',u]⟩ :- [¬may[Cst_N q',u], anadom[u]]."
    by auto
  then have "lh_consequence ϱ (must⟨[Cst_N q',u]⟩ :- [¬may[Cst_N q',u], anadom[u]].)
                            (must⟨[q ·vid (λx. Action undefined), Cst_A undefined]⟩.)"
    using c_p(2) by auto
  then have "∃σ'. (must⟨[Cst_N q', u]⟩. ·vlh σ') = must⟨[q ·vid (λx. Action undefined), Cst_A undefined]⟩.
                ∧ ⟦[¬may[Cst_N q', u], anadom[u]]⟧rhs ϱ σ'"
    unfolding lh_consequence_def using the_lh.simps clause.sel(3) by metis
  then obtain σ' where σ'_p:
    "(must⟨[Cst_N q', u]⟩. ·vlh σ') = must⟨[q ·vid (λx. Action undefined), Cst_A undefined]⟩."
    "⟦[¬may[Cst_N q', u], anadom[u]]⟧rhs ϱ σ'"
    by metis
  then have "σ' the_u = Action undefined"
    by auto
  then have "ϱ ⊨rh anadom[(Cst_A undefined)]"
    using σ'_p(2) solves_rh_substv_rh_if_meaning_rh  by auto
  then show False
    using assms(1) not_anadom_action by auto
qed

lemma if_must:
  assumes "ϱ ⊨lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨lh must⟨[q,d]⟩."
  shows
    "ϱ ⊨rh ¬may[q, d] ∧ ϱ ⊨lh anadom⟨[d]⟩. ∧ is_Node_id q ∧is_Elem_id d ∧ the_Elem_id d ∈ analysis_dom"
proof -
  have Csts: "is_Cst q" "is_Cst d"
    using must_fst_id_is_Cst must_snd_id_is_Cst using assms by auto

  from assms(1,2) Csts have "∃c ∈ ana_pg_fw_must. lh_consequence ϱ c (must⟨[q,d]⟩.)"
    using solves_lh_least[of ana_pg_fw_must ϱ s_BV "[q,d]" the_must] fw_may.ana_pg_fw_may_finite
      fw_may.ana_pg_fw_may_stratified by fastforce

  then obtain c where
    "c ∈ ana_pg_fw_must"
    "lh_consequence ϱ c (must⟨[q,d]⟩.)"
    by auto
  from this have "∃q'. c = must⟨[Cst_N q',u]⟩ :- [¬may[Cst_N q',u], anadom[u]]."
    unfolding fw_may.ana_pg_fw_may_def fw_may.ana_entry_node_def lh_consequence_def
      fw_may.ana_init_def fw_may.ana_anadom_def fw_may.ana_kill_edge_def fw_may.ana_gen_edge_def
```

```
      fw_may.ana_must_def by auto

  then obtain q' where "c = must⟨[Cst_N q',u]⟩ :- [¬may[Cst_N q',u], anadom[u]]."
    by auto
  have "lh_consequence ϱ (must⟨[Cst_N q',u]⟩ :- [¬may[Cst_N q',u], anadom[u]].) (must⟨[q,d]⟩.)"
    using ‹c = must⟨[Cst_N q', u]⟩ :- [¬may [Cst_N q', u], anadom [u]] .›
      ‹lh_consequence ϱ c must⟨[q, d]⟩.› by fastforce
  then have "∃σ. (must⟨[Cst_N q', u]⟩. ·_vlh σ) = must⟨[q, d]⟩. ∧ ⟦[¬may[Cst_N q', u], anadom[u]]⟧_rhs ϱ
σ"
    unfolding lh_consequence_def by auto
  then obtain σ where σ_p:
    "(must⟨[Cst_N q', u]⟩. ·_vlh σ) = must⟨[q, d]⟩."
    "⟦[¬may[Cst_N q', u], anadom[u]]⟧_rhs ϱ σ"
    by auto
  then have "q = Cst_N q'"
    by auto
  from σ_p have "∃d'. σ the_u = d' ∧ d = Cst d'"
    by auto
  then obtain d' where
    "σ the_u = d'"
    "d = Cst d'"
    by auto
  from σ_p(2) have "⟦¬may[Cst_N q', u]⟧_rh ϱ σ"
    by auto
  then have "ϱ ⊨_rh ¬may[q, d]"
    using solves_rh_substv_rh_if_meaning_rh ‹σ the_u = d'› ‹d = Cst d'› ‹q = Cst_N q'› by force
  have "⟦anadom[u]⟧_rh ϱ σ"
    using σ_p(2) by auto
  then have "ϱ ⊨_rh anadom[d]"
    using solves_rh_substv_rh_if_meaning_rh ‹σ the_u = d'› ‹d = Cst d'› ‹q = Cst_N q'› by force
  then have "the_Elem_id d ∈ analysis_dom ∧ is_Elem_id d"
    using in_analysis_dom_if_anadom[of ϱ d] assms by fastforce
  show ?thesis
    using ‹ϱ ⊨_rh ¬may [q, d]› ‹ϱ ⊨_rh anadom [d]› ‹q = Cst_N q'›
      ‹the_Elem_id d ∈ analysis_dom ∧ is_Elem_id d› by auto
qed

lemma not_must_and_may:
  assumes "[Node q, Elem d] ∈ ϱ the_must"
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "[Node q, Elem d] ∈ ϱ the_may"
  shows False
proof -
  have "ϱ ⊨_lh must⟨[Cst_N q, Cst_E d]⟩."
    using assms(1) by auto
  then have "ϱ ⊨_rh ¬may [Cst_N q, Cst_E d]"
    using if_must assms(2) by metis
  then show False
    using assms(3) by auto
qed

lemma not_solves_must_and_may:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[Cst_N q, Cst_E d]⟩."
  assumes "ϱ ⊨_lh may⟨[Cst_N q, Cst_E d]⟩."
  shows "False"
proof -
  have "[Node q, Elem d] ∈ ϱ the_must"
    using assms(2)
    unfolding solves_lh.simps
    unfolding meaning_lh.simps
    by auto
  moreover
```

```
  have "[Node q, Elem d] ∈ ϱ the_may"
    using assms(3)
    unfolding solves_lh.simps
    unfolding meaning_lh.simps
    by auto
  ultimately
  show "False"
    using not_must_and_may[of q d ϱ] assms(1) by auto
qed

lemma anadom_if_must:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q, d]⟩."
  shows "ϱ ⊨_lh anadom⟨[d]⟩."
  using assms(1) assms(2) if_must by blast

lemma not_must_action:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "¬ϱ ⊨_lh must⟨[Cst_A q,d]⟩."
  by (metis assms cst.disc(3) id.sel(2) if_must)

lemma is_encode_elem_if_must_right_arg:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q, d]⟩."
  shows "∃d'. d = Cst_E d'"
  by (metis assms(1) assms(2) cst.collapse(2) id.collapse(2) if_must)

lemma not_must_element:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "¬ϱ ⊨_lh must⟨[Cst_E q,d]⟩."
  by (metis assms cst.disc(2) id.sel(2) if_must)

lemma is_encode_node_if_must_left_arg:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q, d]⟩."
  shows "∃q'. q = Cst_N q'"
  by (metis assms(1) assms(2) cst.collapse(1) id.collapse(2) if_must)

lemma in_analysis_dom_if_must:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q, d]⟩."
  shows "the_Elem_id d ∈ analysis_dom"
  using assms(1) assms(2) if_must by blast

lemma sound_ana_pg_fw_must':
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  assumes "ϱ ⊨_lh must⟨[q, d]⟩."
  assumes "π ∈ path_with_word_from_to start (the_Node_id q)"
  shows "the_Elem_id d ∈ S^_P⟦π⟧ d_init"
proof -
  have d_ana: "the_Elem_id d ∈ analysis_dom"
    using assms(1) assms(2) in_analysis_dom_if_must by auto

  have πe: "q = Cst_N (end_of π)"
    using assms(1) assms(2) assms(3) is_encode_node_if_must_left_arg by fastforce

  have d_encdec: "d = Cst_E (the_Elem_id d)"
    by (metis cst.sel(2) assms(1) assms(2) id.sel(2) is_encode_elem_if_must_right_arg)

  have not_may: "¬ ϱ ⊨_lh may⟨[Cst_N (end_of π), d]⟩."
    using not_solves_must_and_may[OF assms(1), of "(end_of π)" "the_Elem_id d"] assms(2) πe d_encdec
    by force
  have "¬the_Elem_id d ∈ fw_may.S_hat_path π (analysis_dom - d_init)"
```

```
    using fw_may.sound_ana_pg_fw_may assms(1)
    unfolding fw_may.summarizes_fw_may_def
     edges_def start_def assms(2) edges_def start_def
    using assms(3)  d_encdec edges_def not_may start_def by (metis (mono_tags) mem_Collect_eq)
  then show "the_Elem_id d ∈ S^_P⟦π⟧ d_init"
    using opposite_lemma_path
    using assms(1)
    using d_ana by blast
qed

theorem sound_ana_pg_fw_must:
  assumes "ϱ ⊨_lst ana_pg_fw_must s_BV"
  shows "summarizes_fw_must ϱ"
  using assms unfolding summarizes_fw_must_def using sound_ana_pg_fw_must' by auto

end
```

11.5 Backward must-analysis

```
locale analysis_BV_backward_must = finite_program_graph pg
  for pg :: "('n::finite,'a) program_graph" +
  fixes analysis_dom :: "'d set"
  fixes kill_set :: "('n,'a) edge ⇒ 'd set"
  fixes gen_set :: "('n,'a) edge ⇒ 'd set"
  fixes d_init :: "'d set"
  assumes "finite analysis_dom"
  assumes "d_init ⊆ analysis_dom"
begin

lemma finite_d_init: "finite d_init"
  by (meson analysis_BV_backward_must.axioms(2) analysis_BV_backward_must_axioms
      analysis_BV_backward_must_axioms_def rev_finite_subset)

interpretation LTS edges .

definition S_hat :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^_E⟦_⟧ _") where
  "S^_E⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
  assumes "R1 ⊆ R2"
  shows "S^_E⟦e⟧ R1 ⊆ S^_E⟦e⟧ R2"
  using assms unfolding S_hat_def by auto

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^_Es⟦_⟧ _") where
  "S^_Es⟦[]⟧ R = R" |
  "S^_Es⟦(e # π)⟧ R = S^_E⟦e⟧ (S^_Es⟦π⟧ R)"

lemma S_hat_edge_list_def2:
  "S^_Es⟦π⟧ R = foldr S_hat π R"
proof (induction π arbitrary: R)
  case Nil
  then show ?case
    by simp
next
  case (Cons a π)
  then show ?case
    by simp
qed

lemma S_hat_edge_list_append[simp]:
  "S^_Es⟦xs @ ys⟧ R = S^_Es⟦xs⟧ (S^_Es⟦ys⟧ R)"
  unfolding S_hat_edge_list_def2 foldl_append by auto

lemma S_hat_edge_list_mono:
```

```
  assumes "R1 ⊆ R2"
  shows "S^_Es⟦π⟧ R1 ⊆ S^_Es⟦π⟧ R2"
proof(induction π)
  case Nil
  then show ?case
    using assms by auto
next
  case (Cons x xs)
  then show ?case
    using assms by (simp add: S_hat_mono)
qed

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^_P⟦_⟧ _") where
  "S^_P⟦π⟧ R = S^_Es⟦LTS.transition_list π⟧ R"

definition summarizes_bw_must :: "(pred, ('n, 'v, 'd) cst) pred_val ⇒ bool" where
   "summarizes_bw_must ϱ ⟷
     (∀q d.
         ϱ ⊨_lh must⟨[q, d]⟩. ⟶
          (∀π. π ∈ path_with_word_from_to (the_Node_id q) end ⟶ the_Elem_id d ∈ S^_P⟦π⟧ d_init))"

interpretation fw_must: analysis_BV_forward_must
  pg_rev analysis_dom "λe. (kill_set (rev_edge e))" "(λe. gen_set (rev_edge e))" d_init
  using analysis_BV_forward_must_def finite_pg_rev analysis_BV_backward_must_axioms
    analysis_BV_backward_must_def analysis_BV_backward_must_axioms_def
    analysis_BV_forward_must_axioms.intro finite_program_graph.intro
    program_graph.edges_def by metis


abbreviation ana_pg_bw_must where
  "ana_pg_bw_must == fw_must.ana_pg_fw_must"

lemma rev_end_is_start:
  assumes "ss ≠ []"
  assumes "end_of (ss, w) = end"
  shows "start_of (rev ss, rev w) = fw_must.start"
  using assms
  unfolding LTS.end_of_def LTS.start_of_def fw_must.start_def pg_rev_def fw_must.start_def
  using hd_rev by (metis fw_must.start_def fst_conv pg_rev_def snd_conv)

lemma S_hat_edge_list_forward_backward:
  "S^_Es⟦ss⟧ d_init = fw_must.S_hat_edge_list (rev_edge_list ss) d_init"
proof (induction ss)
  case Nil
  then show ?case
    unfolding rev_edge_list_def by auto
next
  case (Cons a ss)
  have "S^_Es⟦a # ss⟧ d_init = S^_E⟦a⟧ S^_Es⟦ss⟧ d_init"
    by simp
  also
  have "... = (((S^_Es⟦ss⟧ d_init) - kill_set a) ∪ gen_set a)"
    using S_hat_def by auto
  also
  have "... = fw_must.S_hat_edge_list (rev_edge_list ss) d_init - kill_set a ∪ gen_set a"
    using Cons by auto
  also
  have "... = fw_must.S_hat_edge_list (rev_edge_list ss) d_init - kill_set (rev_edge (rev_edge a))
               ∪ gen_set (rev_edge (rev_edge a))"
    by simp
  also
  have "... = fw_must.S_hat (rev_edge a) (fw_must.S_hat_edge_list (rev_edge_list ss) d_init)"
    using fw_must.S_hat_def by auto
```

```
  also
  have "... = fw_must.S_hat (rev_edge a) (fw_must.S_hat_edge_list (rev (map rev_edge ss)) d_init)"
    by (simp add: rev_edge_list_def)
  also
  have "... = fw_must.S_hat (rev_edge a) (foldl (λx y. fw_must.S_hat y x) d_init (rev (map rev_edge
ss)))"
    using fw_must.S_hat_edge_list_def2 by force
  also
  have "... = fw_must.S_hat (rev_edge a) (foldr fw_must.S_hat (map rev_edge ss) d_init)"
    by (simp add: foldr_conv_foldl)
  also
  have "... = foldr fw_must.S_hat (rev (rev (map rev_edge (a # ss)))) d_init"
    by force
  also
  have "... = foldl (λa b. fw_must.S_hat b a) d_init (rev (map rev_edge (a # ss)))"
    by (simp add: foldr_conv_foldl)
  also
  have "... = fw_must.S_hat_edge_list (rev (map rev_edge (a # ss))) d_init"
    using fw_must.S_hat_edge_list_def2 by auto
  also
  have "... = fw_must.S_hat_edge_list (rev_edge_list (a # ss)) d_init"
    by (simp add: rev_edge_list_def)
  finally
  show ?case
    by auto
qed

lemma S_hat_path_forward_backward:
  assumes "(ss,w) ∈ path_with_word"
  shows "S^_P⟦(ss, w)⟧ d_init = fw_must.S_hat_path (rev ss, rev w) d_init"
  using S_hat_edge_list_forward_backward unfolding S_hat_path_def fw_must.S_hat_path_def
  by (metis transition_list_rev_edge_list assms)

lemma summarizes_fw_must_forward_backward':
  assumes "fw_must.summarizes_fw_must ϱ"
  assumes "ϱ ⊨_lh must⟨[q, d]⟩."
  assumes "π ∈ path_with_word_from_to (the_Node_id q) end"
  shows "the_Elem_id d ∈ S^_P⟦π⟧ d_init"
proof -
  define rev_π where "rev_π = (rev (fst π), rev (snd π))"
  have rev_π_path: "rev_π ∈ LTS.path_with_word fw_must.edges"
    using rev_π_def assms(3) fw_must.edges_def pg_rev_def rev_path_in_rev_pg
    by (metis (no_types, lifting) fst_conv mem_Collect_eq  prod.collapse)
  have rev_π_start: "start_of rev_π = fw_must.start"
    using  rev_π_def analysis_BV_backward_must_axioms
      assms(3) pg_rev_def start_of_def edges_def end_of_def hd_rev
      by (metis (mono_tags, lifting) fw_must.start_def mem_Collect_eq prod.sel)
  have rev_π_start_end: "end_of rev_π = the_Node_id q"
    using assms(3) rev_π_def end_of_def last_rev start_of_def
    by (metis (mono_tags, lifting) mem_Collect_eq prod.sel(1))
  have "the_Elem_id d ∈ fw_must.S_hat_path (rev (fst π), rev (snd π)) d_init"
    using rev_π_def rev_π_path rev_π_start_end rev_π_start assms(1) assms(2)
      fw_must.summarizes_fw_must_def by blast
  then show ?thesis
    by (metis (no_types, lifting) S_hat_path_forward_backward assms(3) mem_Collect_eq prod.collapse)
qed


lemma summarizes_bw_must_forward_backward:
  assumes "fw_must.summarizes_fw_must ϱ"
  shows "summarizes_bw_must ϱ"
  unfolding summarizes_bw_must_def
  using assms summarizes_fw_must_forward_backward' by auto
```

```
theorem sound_ana_pg_bw_must:
  assumes "ϱ ⊨_lst ana_pg_bw_must s_BV"
  shows "summarizes_bw_must ϱ"
  using assms fw_must.sound_ana_pg_fw_must[of ϱ] summarizes_bw_must_forward_backward by metis

end

end
theory Reaching_Definitions imports Bit_Vector_Framework begin
```

— We encode the Reaching Definitions analysis into Datalog. First we define the analysis, then we encode the analysis directly into Datalog and prove the encoding correct. Hereafter we encode it into Datalog again, but this time using our Bit-Vector Framework locale. We also prove this encoding correct. This latter encoding is described in our SAC 2024 paper.

12 Reaching Definitions

```
type_synonym ('n,'v) def = "'v * 'n option * 'n"

type_synonym ('n,'v) analysis_assignment = "'n ⇒ ('n,'v) def set"
```

12.1 What is defined on a path

```
fun def_action :: "'v action ⇒ 'v set" where
  "def_action (x ::= a) = {x}"
| "def_action (Bool b) = {}"
| "def_action Skip = {}"

abbreviation def_edge :: "('n,'v action) edge ⇒ 'v set" where
  "def_edge == λ(q1, α, q2). def_action α"

definition def_of :: "'v ⇒ ('n,'v action) edge ⇒ ('n,'v) def" where
  "def_of == (λx (q1, α, q2). (x, Some q1, q2))"

definition def_var :: "('n,'v action) edge list ⇒ 'v ⇒ 'n ⇒ ('n,'v) def" where
  "def_var π x start = (if (∃e ∈ set π. x ∈ def_edge e)
                        then (def_of x (last (filter (λe. x ∈ def_edge e) π)))
                        else (x, None, start))"

definition def_path :: "('n list × 'v action list) ⇒ 'n ⇒ ('n,'v) def set" where
  "def_path π start = ((λx. def_var (LTS.transition_list π) x start) ‘ UNIV)"
```

12.2 Reaching Definitions in Datalog

```
datatype ('n,'v) RD_elem =
  RD_Node 'n
  | RD_Var 'v
  | Questionmark

datatype RD_var =
  the_u
  | the_v
  | the_w

datatype RD_pred =
  the_RD
  | the_VAR

abbreviation Cst_RDN :: "'n ⇒ (RD_var, ('n, 'v) RD_elem) id" where
  "Cst_RDN q == Cst (RD_Node q)"

fun Cst_RDN_Q :: "'n option ⇒ (RD_var, ('n, 'v) RD_elem) id" where
```

```
  "Cst_RDN_Q (Some q) = Cst (RD_Node q)"
| "Cst_RDN_Q None = Cst Questionmark"

abbreviation Cst_RDV :: "'v ⇒ (RD_var, ('n, 'v) RD_elem) id" where
  "Cst_RDV v == Cst (RD_Var v)"

abbreviation RD_Cls :: "(RD_var, ('n, 'v) RD_elem) id list ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) rh
list ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) clause" ("RD⟨_⟩ :- _ .") where
  "RD⟨args⟩ :- ls. ≡ Cls the_RD args ls"

abbreviation VAR_Cls :: "'v ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) clause" ("VAR⟨_⟩ :-.") where
  "VAR⟨x⟩ :-. == Cls the_VAR [Cst_RDV x] []"

abbreviation RD_lh :: "(RD_var, ('n, 'v) RD_elem) id list ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) lh"
("RD⟨_⟩.") where
  "RD⟨args⟩. ≡ (the_RD, args)"

abbreviation VAR_lh :: "'v ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) lh" ("VAR⟨_⟩.") where
  "VAR⟨x⟩. ≡ (the_VAR, [Cst_RDV x])"


abbreviation "RD == PosLit the_RD"
abbreviation "VAR == PosLit the_VAR"

abbreviation u :: "(RD_var, 'aa) id" where
  "u == Var the_u"

abbreviation v :: "(RD_var, 'aa) id" where
  "v == Var the_v"

abbreviation w :: "(RD_var, 'aa) id" where
  "w == Var the_w"

fun ana_edge :: "('n, 'v action) edge ⇒ (RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "ana_edge (q_o, x ::= a, q_s) =
     {
        RD⟨[Cst_RDN q_s, u, v, w]⟩ :-
          [
            RD[Cst_RDN q_o, u, v, w],
            u ≠ (Cst_RDV x)
          ].
        ,
        RD⟨[Cst_RDN q_s, Cst_RDV x, Cst_RDN q_o, Cst_RDN q_s]⟩ :- [].
     }"
| "ana_edge (q_o, Bool b, q_s) =
     {
       RD⟨[Cst_RDN q_s, u, v, w]⟩ :-
         [
           RD[Cst_RDN q_o, u, v, w]
         ].
     }"
| "ana_edge (q_o, Skip, q_s) =
     {
       RD⟨[Cst_RDN q_s, u, v, w]⟩ :-
         [
           RD[Cst_RDN q_o, u, v, w]
         ].
     }"

definition ana_entry_node :: "'n ⇒ (RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "ana_entry_node start =
     {
       RD⟨[Cst_RDN start, u, Cst Questionmark, Cst_RDN start]⟩ :-
```

```
        [
          VAR[u]
        ].
    }"


fun ana_RD :: "('n, 'v action) program_graph ⇒ (RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "ana_RD (es,start,end) = ⋃(ana_edge ‘ es) ∪ ana_entry_node start"

definition var_contraints :: "(RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "var_contraints = VAR_Cls ‘ UNIV"

type_synonym ('n,'v) quadruple = "'n *'v * 'n option * 'n"

fun summarizes_RD :: "(RD_pred,('n,'v) RD_elem) pred_val ⇒ ('n,'v action) program_graph ⇒ bool" where
  "summarizes_RD ϱ (es, start, end) =
    (∀π x q1 q2.
       π ∈ LTS.path_with_word_from es start ⟶
       (x, q1, q2) ∈ def_path π start ⟶
       ϱ ⊨_lh RD⟨[Cst_RDN (LTS.end_of π), Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩.)"

lemma def_var_x: "fst (def_var ts x start) = x"
  unfolding def_var_def by (simp add: case_prod_beta def_of_def)

lemma last_def_transition:
  assumes "length ss = length w"
  assumes "x ∈ def_action α"
  assumes "(x, q1, q2) ∈ def_path (ss @ [s, s'], w @ [α]) start"
  shows "Some s = q1 ∧ s' = q2"
proof -
  obtain y where y_p: "(x, q1, q2) = def_var (transition_list (ss @ [s], w) @ [(s, α, s')]) y start"
    by (metis (no_types, lifting) assms(1) assms(3) def_path_def imageE transition_list_reversed_simp)
  show ?thesis
  proof (cases "y = x")
    case True
    then show ?thesis
      using assms y_p unfolding def_var_def def_of_def by auto
  next
    case False
    then show ?thesis
      by (metis y_p def_var_x fst_conv)
  qed
qed

lemma not_last_def_transition:
  assumes "length ss = length w"
  assumes "x ∉ def_action α"
  assumes "(x, q1, q2) ∈ def_path (ss @ [s, s'], w @ [α]) start"
  shows "(x, q1, q2) ∈ def_path (ss @ [s], w) start"
proof -
  obtain y where y_p: "(x, q1, q2) = def_var (transition_list (ss @ [s], w) @ [(s, α, s')]) y start"
    by (metis (no_types, lifting) assms(1) assms(3) def_path_def imageE transition_list_reversed_simp)
  have "(x, q1, q2) ∈ range (λx. def_var (transition_list (ss @ [s], w)) x start)"
  proof (cases "y = x")
    case True
    then show ?thesis
      using assms y_p unfolding def_var_def def_of_def by auto
  next
    case False
    then show ?thesis
      by (metis y_p def_var_x fst_conv)
  qed
  then show ?thesis
```

```
    by (simp add: def_path_def)
qed

theorem RD_sound':
  assumes "(ss,w) ∈ LTS.path_with_word_from es start"
  assumes "(x,q1,q2) ∈ def_path (ss,w) start"
  assumes "ϱ ⊨dl (var_contraints ∪ ana_RD (es, start, end))"
  shows "ϱ ⊨lh RD⟨[Cst_RDN (LTS.end_of (ss, w)), Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
  using assms
proof (induction rule: LTS.path_with_word_from_induct_reverse[OF assms(1)])
  case (1 s)
  have "VAR⟨x⟩ :-. ∈ var_contraints"
    unfolding var_contraints_def by auto
  from assms(3) this have "ϱ ⊨cls VAR⟨x⟩ :-."
    unfolding solves_program_def by auto
  then have "∀y. ⟦(VAR⟨x⟩ :-.)⟧cls ϱ y"
    unfolding solves_cls_def by auto
  then have x_sat: "[RD_Var x] ∈ ϱ the_VAR"
    by auto

  have "RD⟨[Cst_RDN start, u, Cst Questionmark, Cst_RDN start]⟩ :-
         [
           VAR[u]
         ]. ∈ ana_RD (es, start, end)"
    by (simp add: ana_entry_node_def)
  then have "ϱ ⊨cls RD⟨[Cst_RDN start, u, Cst Questionmark, Cst_RDN start]⟩ :- [VAR [u]] ."
    using assms(3) unfolding solves_program_def by auto
  then have "∀σ. ⟦RD⟨[Cst_RDN start, u, Cst Questionmark, Cst_RDN start]⟩ :- [VAR [u]] .⟧cls ϱ σ"
    unfolding solves_cls_def by metis
  then have "⟦RD⟨[Cst_RDN start, u, Cst Questionmark, Cst_RDN start]⟩ :- [VAR [u]] .⟧cls ϱ (λv. RD_Var
x)"
    by presburger
  then have "[RD_Var x] ∈ ϱ the_VAR ⟶ [RD_Node start, RD_Var x, Questionmark, RD_Node start] ∈ ϱ
the_RD"
    by simp
  then have "[RD_Node start, RD_Var x, Questionmark, RD_Node start] ∈ ϱ the_RD"
    using x_sat by auto

  from this 1 show ?case
    unfolding LTS.LTS.end_of_def def_path_def def_var_def LTS.start_of_def by auto
next
  case (2 ss s w α s')
  from 2(1) have len: "length ss = length w"
    using LTS.path_with_word_length by force
  show ?case
  proof(cases "x ∈ def_action α")
    case True
    then have sq: "Some s = q1 ∧ s' = q2" using 2(5)
      using last_def_transition[of ss w x α q1 q2 s s'] len by auto
    from True have "∃e. (s,x ::= e,s') ∈ es"
      using "2.hyps"(2) by (cases α) auto
    then have "RD⟨[Cst_RDN q2, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩ :- []. ∈ ana_RD (es, start, end)"
      using True ana_RD.simps sq by fastforce
    then have "ϱ ⊨cls RD⟨[Cst_RDN q2, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩ :- [] ."
      using 2(6) unfolding solves_program_def by auto
    then have "ϱ ⊨lh RD⟨[Cst_RDN q2, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
      using solves_lh_lh by metis
    then show ?thesis
      by (simp add: LTS.end_of_def sq)
  next
    case False
    then have x_is_def: "(x, q1, q2) ∈ def_path (ss @ [s], w) start" using 2(5)
      using not_last_def_transition len by force
```

```
    then have "ϱ ⊨lh RD⟨[Cst_RDN (LTS.end_of (ss @ [s], w)), Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
    proof -
      have "(ss @ [s], w) ∈ LTS.path_with_word es"
        using 2(1) by auto
      moreover
      have "ϱ ⊨dl (var_contraints ∪ ana_RD (es, start, end))"
        using 2(6) by auto
      moreover
      have "LTS.start_of (ss @ [s], w) = start"
        using "2.hyps"(1) by auto
      moreover
      have "(x, q1, q2) ∈ def_path (ss @ [s], w) start"
        using x_is_def by auto
      ultimately
      show "ϱ ⊨lh RD⟨[Cst_RDN (LTS.end_of (ss @ [s], w)), Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
        using 2(3) by auto
    qed
    then have ind: "ϱ ⊨lh RD⟨[Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
      by (simp add: LTS.end_of_def)
    define μ where "μ = undefined(the_u := Cst_RDV x, the_v := Cst_RDN_Q q1, the_w := Cst_RDN q2)"
    show ?thesis
    proof (cases α)
      case (Asg y e)
      have xy: "x ≠ y"
        using False Asg by auto
      then have xy': "ϱ ⊨rh (Cst_RDV x ≠ Cst_RDV y)"
        by auto
      have "(s, y ::= e, s') ∈ es"
        using "2.hyps"(2) Asg by auto
      then have "RD⟨[Cst_RDN s', u, v, w]⟩ :-
          [
            RD[Cst_RDN s, u, v, w],
            u ≠ (Cst_RDV y)
          ]. ∈ ana_RD (es,start,end)"
        unfolding ana_RD.simps by force
      from this False have "ϱ ⊨cls RD⟨[Cst_RDN s', u, v, w]⟩ :- [RD [Cst_RDN s, u, v, w], u ≠ Cst_RDV
y] ."
        by (meson "2.prems"(3) UnCI solves_program_def)
      moreover have
        "(RD⟨[Cst_RDN s', u, v, w]⟩ :-
          [
            RD[Cst_RDN s, u, v, w],
            u ≠ (Cst_RDV y)
          ].) ·cls μ =
          RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩ :-
          [
            RD [Cst_RDN s,  Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2],
            Cst_RDV x ≠ Cst_RDV y] ."
        unfolding μ_def by auto
      ultimately
      have "ϱ ⊨cls RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩
                  :- [RD [Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2], Cst_RDV x ≠ Cst_RDV y]
."
        unfolding solves_cls_def by (metis substitution_lemma_cls)
      then have "ϱ ⊨cls RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩
                       :- [RD [Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]] ."
        using xy' by (simp add: prop_inf_last_from_cls_rh_to_cls)
      then have "ϱ ⊨cls RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩ :- [] ."
        using ind by (metis meaning_cls.simps modus_ponens_rh solves_cls_def solves_lh.simps)
      then have "ϱ ⊨lh RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
        using solves_lh_lh by metis
      then show ?thesis
        by (simp add: LTS.end_of_def)
```

```
    next
      case (Bool b)
      have "(s, Bool b, s') ∈ es"
        using "2.hyps"(2) Bool by auto
      then have "RD⟨[Cst_RDN s', u, v, w]⟩ :-
         [
          RD[Cst_RDN s, u, v, w]
         ]. ∈ ana_RD (es,start,end)"
        unfolding ana_RD.simps by force
      then have "ϱ ⊨_cls RD⟨[Cst_RDN s', u, v, w]⟩ :- [RD [Cst_RDN s, u, v, w]] ."
        by (meson "2.prems"(3) UnCI solves_program_def)
      moreover
      have "(RD⟨[Cst_RDN s', u, v, w]⟩ :- [RD[Cst_RDN s, u, v, w]].) ·_cls μ =
            RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩ :-
              [RD[Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]]."
        unfolding μ_def by auto
      ultimately have "ϱ ⊨_cls RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩
                               :- [RD [Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]] ."
        by (metis substitution_lemma)
      then have "ϱ ⊨_lh RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩."
        using ind
        by (meson prop_inf_only)
      then show ?thesis
        by (simp add: LTS.end_of_def)
    next
      case Skip
      have "(s, Skip, s') ∈ es"
        using "2.hyps"(2) Skip by auto
      then have "RD⟨[Cst_RDN s', u, v, w]⟩ :-
         [
          RD[Cst_RDN s, u, v, w]
         ]. ∈ ana_RD (es,start,end)"
        unfolding ana_RD.simps by force
      then have "ϱ ⊨_cls RD⟨[Cst_RDN s', u, v, w]⟩ :- [RD [Cst_RDN s, u, v, w]] ."
        by (meson "2.prems"(3) UnCI solves_program_def)
      moreover
      have "(RD⟨[Cst_RDN s', u, v, w]⟩ :- [RD [Cst_RDN s, u, v, w]] .) ·_cls μ =
            RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩ :-
              [RD [Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]]."
        unfolding μ_def by auto
      ultimately
      have "ϱ ⊨_cls RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩
                    :- [RD [Cst_RDN s, Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]] ."
        by (metis substitution_lemma)
      from modus_ponens_rh[OF this ind] have "ϱ ⊨_lh RD⟨[Cst_RDN s', Cst_RDV x, Cst_RDN_Q q1, Cst_RDN
q2]⟩."
        .
      then show ?thesis
        by (simp add: LTS.end_of_def)
    qed
  qed
qed

theorem RD_sound:
  assumes "ϱ ⊨_dl (var_contraints ∪ ana_RD pg)"
  shows "summarizes_RD ϱ pg"
  using assms RD_sound' by (cases pg) force
```

12.3 Reaching Definitions as Bit-Vector Framework analysis

```
locale analysis_RD = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph" +
  assumes "finite edges"
begin
```

```
interpretation LTS edges .

definition analysis_dom_RD :: "('n,'v) def set" where
  "analysis_dom_RD = UNIV × UNIV × UNIV"

fun kill_set_RD :: "('n,'v action) edge ⇒ ('n,'v) def set" where
  "kill_set_RD (q_o, x ::= a, q_s) = {x} × UNIV × UNIV"
| "kill_set_RD (q_o, Bool b, q_s) = {}"
| "kill_set_RD (v, Skip, vc) = {}"

fun gen_set_RD :: "('n,'v action) edge ⇒ ('n,'v) def set" where
  "gen_set_RD (q_o, x ::= a, q_s) = {x} × {Some q_o} × {q_s}"
| "gen_set_RD (q_o, Bool b, q_s) = {}"
| "gen_set_RD (v, Skip, vc) = {} "

definition d_init_RD :: " ('n,'v) def set" where
  "d_init_RD = (UNIV × {None} × {start})"


lemma finite_analysis_dom_RD: "finite analysis_dom_RD"
  by auto

lemma d_init_RD_subset_analysis_dom_RD:
  "d_init_RD ⊆ analysis_dom_RD"
  unfolding d_init_RD_def analysis_dom_RD_def by auto

lemma gen_RD_subset_analysis_dom: "gen_set_RD e ⊆ analysis_dom_RD"
  unfolding analysis_dom_RD_def by auto

lemma kill_RD_subset_analysis_dom: "kill_set_RD e ⊆ analysis_dom_RD"
  unfolding analysis_dom_RD_def by auto

interpretation fw_may: analysis_BV_forward_may pg analysis_dom_RD kill_set_RD gen_set_RD d_init_RD
  using analysis_BV_forward_may_def analysis_RD_axioms analysis_RD_def
    d_init_RD_subset_analysis_dom_RD finite_analysis_dom_RD gen_RD_subset_analysis_dom
    kill_RD_subset_analysis_dom analysis_BV_forward_may_axioms.intro by metis

lemma def_var_def_edge_S_hat:
  assumes "def_var π x start ∈ R"
  assumes "x ∉ def_edge t"
  shows "def_var π x start ∈ fw_may.S_hat t R"
proof -
  define q1 where "q1 = fst t"
  define α where "α = fst (snd t)"
  define q2 where "q2 = snd (snd t)"
  have t_def: "t = (q1, α, q2)"
    by (simp add: α_def q1_def q2_def)

  from assms(2) have x_not_def: "x ∉ def_edge (q1, α, q2)"
    unfolding t_def by auto

  have "def_var π x start ∈ fw_may.S_hat (q1, α, q2) R"
  proof (cases α)
    case (Asg y exp)
    then show ?thesis
      by (metis (no_types, lifting) DiffI Un_iff assms(1) x_not_def def_action.simps(1) def_var_x fw_may.S_hat_de
kill_set_RD.simps(1) mem_Sigma_iff old.prod.case prod.collapse)
  next
    case (Bool b)
    then show ?thesis
      by (simp add: fw_may.S_hat_def assms(1))
  next
```

```
    case Skip
    then show ?thesis
      by (simp add: fw_may.S_hat_def assms(1))
  qed
  then show ?thesis
    unfolding t_def by auto
qed

lemma def_var_S_hat_edge_list: "(def_var π) x start ∈ fw_may.S_hat_edge_list π d_init_RD"
proof (induction π rule: rev_induct)
  case Nil
  then show ?case
    unfolding def_var_def d_init_RD_def by auto
next
  case (snoc t π)
  then show ?case
  proof (cases "x ∈ def_edge t")
    case True
    then have "def_var (π @[t]) x start = def_var [t] x start"
      by (simp add: def_var_def)
    moreover
    have "fw_may.S_hat_edge_list (π @ [t]) d_init_RD = fw_may.S_hat t (fw_may.S_hat_edge_list π d_init_RD)"
      unfolding fw_may.S_hat_edge_list_def2 by simp
    moreover
    obtain q1 α q2 where t_split: "t = (q1, α, q2)"
      using prod_cases3 by blast
    moreover
    have "def_var [t] x start ∈ fw_may.S_hat t (fw_may.S_hat_edge_list π d_init_RD)"
      unfolding fw_may.S_hat_def def_var_def def_of_def using True t_split by (cases α) auto
    ultimately
    show ?thesis by auto
  next
    case False
    obtain q1 α q2 where t_split: "t = (q1, α, q2)"
      using prod_cases3 by blast
    from False have "def_var (π @ [t]) x start = def_var π x start"
      by (simp add: def_var_def)
    moreover
    from snoc.IH have "def_var π x start ∈ fw_may.S_hat t (fw_may.S_hat_edge_list π d_init_RD)"
      by (simp add: False def_var_def_edge_S_hat)
    then have "def_var π x start ∈ fw_may.S_hat_edge_list (π @ [t]) d_init_RD"
      unfolding fw_may.S_hat_edge_list_def2 by simp
    ultimately
    show ?thesis
      using snoc by auto
  qed
qed

lemma last_overwrites:
  "def_var (π @ [(q1, x ::= exp, q2)]) x start = (x, Some q1, q2)"
proof -
  have "x ∈ def_edge (q1, x ::= exp, q2)"
    by auto
  then have "∃e∈set (π @ [(q1, x ::= exp, q2)]). x ∈ def_edge e"
    by auto
  have "def_var (π @ [(q1, x ::= exp, q2)]) x start = def_of x (last (filter (λe. x ∈ def_edge e) (π
@ [(q1, x ::= exp, q2)])))"
    unfolding def_var_def by auto
  also
  have "... = def_of x (q1, x ::= exp, q2)"
    by auto
  also
  have "... = (x, Some q1, q2)"
```

```
    unfolding def_of_def by auto
  finally
  show ?thesis
    .
qed

lemma S_hat_edge_list_last: "fw_may.S_hat_edge_list (π @ [e]) d_init_RD = fw_may.S_hat e (fw_may.S_hat_edge_lis
π d_init_RD)"
  using fw_may.S_hat_edge_list_def2 foldl_conv_foldr by simp

lemma def_var_if_S_hat:
  assumes "(x,q1,q2) ∈ fw_may.S_hat_edge_list π d_init_RD"
  shows "(x,q1,q2) = (def_var π) x start"
  using assms
proof (induction π rule: rev_induct)
  case Nil
  then show ?case
    by (metis append_is_Nil_conv d_init_RD_def def_var_def in_set_conv_decomp fw_may.S_hat_edge_list.simps(1)
list.distinct(1) mem_Sigma_iff singletonD)
next
  case (snoc e π)

  from snoc(2) have "(x, q1, q2) ∈ fw_may.S_hat e (fw_may.S_hat_edge_list π d_init_RD)"
    using S_hat_edge_list_last by blast

  then have "(x, q1, q2) ∈ fw_may.S_hat_edge_list π d_init_RD - kill_set_RD e ∨ (x, q1, q2) ∈ gen_set_RD
e"
    unfolding fw_may.S_hat_def by auto
  then show ?case
  proof
    assume a: "(x, q1, q2) ∈ fw_may.S_hat_edge_list π d_init_RD - kill_set_RD e"
    then have "(x, q1, q2) = def_var π x start"
      using snoc by auto
    moreover
    from a have "(x, q1, q2) ∉ kill_set_RD e"
      by auto
    then have "def_var (π @ [e]) x start = def_var π x start"
    proof -
      assume def_not_kill: "(x, q1, q2) ∉ kill_set_RD e"
      obtain q q' α where "e = (q, α, q')"
        by (cases e) auto
      then have "x ∉ def_edge e"
        using def_not_kill by (cases α) auto
      then show ?thesis
        by (simp add: def_var_def)
    qed
    ultimately
    show ?case
      by auto
  next
    assume a: "(x, q1, q2) ∈ gen_set_RD e"
    obtain q q' α where "e = (q, α, q')"
      by (cases e) auto
    then have "∃exp theq1. e = (theq1, x ::= exp, q2) ∧ q1 = Some theq1"
      using a by (cases α) auto
    then obtain exp theq1 where exp_theq1_p: "e = (theq1, x ::= exp, q2) ∧ q1 = Some theq1"
      by auto
    then have "(x, q1, q2) = def_var (π @ [(theq1, x ::= exp, q2)]) x start"
      using last_overwrites[of π theq1 x exp q2] by auto
    then show ?case
      using exp_theq1_p by auto
  qed
qed
```

```
lemma def_var_UNIV_S_hat_edge_list: "(λx. def_var π x start) ‘ UNIV = fw_may.S_hat_edge_list π d_init_RD"
proof (rule; rule)
  fix x
  assume "x ∈ range (λx. def_var π x start)"
  then show "x ∈ fw_may.S_hat_edge_list π d_init_RD"
    using def_var_S_hat_edge_list by blast
next
  fix x
  assume "x ∈ fw_may.S_hat_edge_list π d_init_RD"
  then show "x ∈ range (λx. def_var π x start)"
    by (metis def_var_if_S_hat prod.collapse range_eqI)
qed

lemma def_path_S_hat_path: "def_path π start = fw_may.S_hat_path π d_init_RD"
  using fw_may.S_hat_path_def def_path_def def_var_UNIV_S_hat_edge_list by metis

definition summarizes_RD :: "(pred, ('n,'v action,('n,'v) def) cst) pred_val ⇒ bool" where
  "summarizes_RD ϱ ⟷ (∀π d. π ∈ path_with_word_from start ⟶ d ∈ def_path π start ⟶
                       ϱ ⊨_lh may⟨[Cst_N (end_of π), Cst_E d]⟩.)"

theorem RD_sound:
  assumes "ϱ ⊨_lst fw_may.ana_pg_fw_may s_BV"
  shows "summarizes_RD ϱ"
  using assms def_path_S_hat_path fw_may.sound_ana_pg_fw_may unfolding fw_may.summarizes_fw_may_def
summarizes_RD.simps
  using edges_def in_mono edges_def start_def start_def summarizes_RD_def by fastforce

end

end
theory Live_Variables imports Reaching_Definitions begin
```

— We encode the Live Variables analysis into Datalog. First we define the analysis, then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

13 Live Variables Analysis

```
fun use_action :: "'v action ⇒ 'v set" where
  "use_action (x ::= a) = fv_arith a"
| "use_action (Bool b) = fv_boolean b"
| "use_action Skip = {}"

fun use_edge :: "('n,'v action) edge ⇒ 'v set" where
  "use_edge (q1, α, q2) = use_action α"

definition use_edge_list :: "('n,'v action) edge list ⇒ 'v ⇒ bool" where
  "use_edge_list π x = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧
                                  x ∈ use_edge e ∧
                                  (¬(∃e' ∈ set π1. x ∈ def_edge e')))"

definition use_path :: "'n list × 'v action list ⇒ 'v set" where
  "use_path π = {x. use_edge_list (LTS.transition_list π) x}"

locale analysis_LV = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges .

definition analysis_dom_LV :: "'v set" where
  "analysis_dom_LV = UNIV"
```

```
fun kill_set_LV :: "('n,'v action) edge ⇒ 'v set" where
  "kill_set_LV (q_o, x ::= a, q_s) = {x}"
| "kill_set_LV (q_o, Bool b, q_s) = {}"
| "kill_set_LV (v, Skip, vc) = {}"

fun gen_set_LV :: "('n,'v action) edge ⇒ 'v set" where
  "gen_set_LV (q_o, x ::= a, q_s) = fv_arith a"
| "gen_set_LV (q_o, Bool b, q_s) = fv_boolean b"
| "gen_set_LV (v, Skip, vc) = {}"

definition d_init_LV :: "'v set" where
  "d_init_LV = {}"

interpretation bw_may: analysis_BV_backward_may pg analysis_dom_LV kill_set_LV gen_set_LV d_init_LV
  using analysis_BV_backward_may.intro analysis_LV_axioms analysis_LV_def analysis_dom_LV_def
    finite_UNIV subset_UNIV analysis_BV_backward_may_axioms_def finite_program_graph_def by metis

lemma use_edge_list_S_hat_edge_list:
  assumes "use_edge_list π x"
  shows "x ∈ bw_may.S_hat_edge_list π d_init_LV"
  using assms
proof (induction π)
  case Nil
  then have False
    unfolding use_edge_list_def by auto
  then show ?case
    by metis
next
  case (Cons e π)
  note Cons_inner = Cons
  from Cons(2) have "∃π1 π2 e'. e # π = π1 @ [e'] @ π2 ∧
                                x ∈ use_edge e' ∧
                                ¬ (∃e''∈set π1. x ∈ def_edge e'')"
    unfolding use_edge_list_def by auto
  then obtain π1 π2 e' where π1_π2_e'_p:
    "e # π = π1 @ [e'] @ π2"
    "x ∈ use_edge e'"
    "¬(∃e''∈set π1. x ∈ def_edge e'')"
    by auto
  then show ?case
  proof (cases π1)
    case Nil
    have "e = e'"
      using π1_π2_e'_p(1) Nil by auto
    then have x_used_a: "x ∈ use_edge e"
      using π1_π2_e'_p(2) by auto
    obtain p α q where a_split: "e = (p, α, q)"
      by (cases e)
    show ?thesis
      using x_used_a bw_may.S_hat_def a_split by (cases α) auto
  next
    case (Cons hd_π1 tl_π1)
    obtain p α q where e_split: "e' = (p, α, q)"
      by (cases e')
    have "(π = tl_π1 @ (p, α, q) # π2) ∧ x ∈ use_action α ∧ (∀e'∈set tl_π1. x ∉ def_edge e')"
      using Cons π1_π2_e'_p e_split by auto
    then have "use_edge_list π x"
      unfolding use_edge_list_def by force
    then have x_in_S_hat_π: "x ∈ bw_may.S_hat_edge_list π d_init_LV"
      using Cons_inner by auto
    have "e ∈ set π1"
      using π1_π2_e'_p(1) Cons(1) by auto
    then have x_not_def_a: "¬x ∈ def_edge e"
```

```
      using π1_π2_e'_p(3) by auto

    obtain p' α' q' where a_split: "e = (p', α', q')"
      by (cases e)

    show ?thesis
    proof (cases "x ∈ kill_set_LV e")
      case True
      show ?thesis
        using True a_split x_not_def_a by (cases α'; force)
    next
      case False
      then show ?thesis
        by (simp add: bw_may.S_hat_def x_in_S_hat_π)
    qed
  qed
qed

lemma S_hat_edge_list_use_edge_list:
  assumes "x ∈ bw_may.S_hat_edge_list π d_init_LV"
  shows "use_edge_list π x"
  using assms
proof (induction π)
  case Nil
  then have False
    using d_init_LV_def by auto
  then show ?case
    by metis
next
  case (Cons e π)
  from Cons(2) have "x ∈ bw_may.S_hat_edge_list π d_init_LV - kill_set_LV e ∪ gen_set_LV e"
    unfolding bw_may.S_hat_edge_list.simps unfolding bw_may.S_hat_def by auto
  then show ?case
  proof
    assume a: "x ∈ bw_may.S_hat_edge_list π d_init_LV - kill_set_LV e"
    then have "x ∈ bw_may.S_hat_edge_list π d_init_LV"
      by auto
    then have "use_edge_list π x"
      using Cons by auto
    then have "∃π1 π2 e'. π = π1 @ [e'] @ π2 ∧ x ∈ use_edge e' ∧ ¬(∃e''∈set π1. x ∈ def_edge e'')"
      unfolding use_edge_list_def by auto
    then obtain π1 π2 e' where π1_π2_e'_p:
      "π = π1 @ [e'] @ π2"
      "x ∈ use_edge e'"
      "¬(∃e''∈set π1. x ∈ def_edge e'')"
      by auto
    obtain q1 α q2 where e_split: "e = (q1, α, q2)"
      by (cases e) auto
    from a have "x ∉ kill_set_LV e"
      by auto
    then have x_not_killed: "x ∉ kill_set_LV (q1, α, q2)"
      using e_split by auto
    have "use_edge_list ((q1, α, q2) # π) x"
    proof (cases α)
      case (Asg y exp)
      then have "x ∉ kill_set_LV (q1, y ::= exp, q2)"
        using x_not_killed by auto
      then have x_not_y: "x ≠ y"
        by auto
      have "(q1, y ::= exp, q2) # π = ((q1, y ::= exp, q2) # π1) @ [e'] @ π2"
        using π1_π2_e'_p by force
      moreover
      have "¬ (∃e'∈set ((q1, y ::= exp, q2) # π1). x ∈ def_edge e')"
```

```
      using π1_π2_e'_p x_not_y by force
    ultimately
    have "use_edge_list ((q1, y ::= exp, q2) # π) x"
      unfolding use_edge_list_def using π1_π2_e'_p x_not_y by metis
    then show ?thesis
      by (simp add: Asg)
  next
    case (Bool b)
    have "(q1, Bool b, q2) # π = ((q1, Bool b, q2) # π1) @ [e'] @ π2"
      using π1_π2_e'_p unfolding use_edge_list_def by auto
    moreover
    have "¬ (∃e'∈set ((q1, Bool b, q2) # π1). x ∈ def_edge e')"
      using π1_π2_e'_p unfolding use_edge_list_def by auto
    ultimately
    have "use_edge_list ((q1, Bool b, q2) # π) x"
      unfolding use_edge_list_def using π1_π2_e'_p by metis
    then show ?thesis
      using Bool by auto
  next
    case Skip
    have "(q1, Skip, q2) # π = ((q1, Skip, q2) # π1) @ [e'] @ π2"
      using π1_π2_e'_p unfolding use_edge_list_def by auto
    moreover
    have "¬ (∃e'∈set ((q1, Skip, q2) # π1). x ∈ def_edge e')"
      using π1_π2_e'_p unfolding use_edge_list_def by auto
    ultimately
    have "use_edge_list ((q1, Skip, q2) # π) x"
      unfolding use_edge_list_def using π1_π2_e'_p by metis
    then show ?thesis
      using Skip by auto
  qed
  then show "use_edge_list (e # π) x"
    using e_split by auto
next
  assume a: "x ∈ gen_set_LV e"
  obtain p α q where a_split: "e = (p, α, q)"
    by (cases e)
  have "use_edge_list ((p, α, q) # π) x"
    using a a_split unfolding use_edge_list_def by (cases α; force)
  then show "use_edge_list (e # π) x"
    using a_split by auto
qed
qed

lemma use_edge_list_set_S_hat_edge_list:
  "{x. use_edge_list π x} = bw_may.S_hat_edge_list π d_init_LV"
  using use_edge_list_S_hat_edge_list S_hat_edge_list_use_edge_list by auto

lemma use_path_S_hat_path: "use_path π = bw_may.S_hat_path π d_init_LV"
  by (simp add: use_edge_list_set_S_hat_edge_list bw_may.S_hat_path_def use_path_def)

definition summarizes_LV :: "(pred, ('n,'v action,'v) cst) pred_val ⇒ bool" where
  "summarizes_LV ϱ ⟷ (∀π d. π ∈ path_with_word_to end ⟶ d ∈ use_path π ⟶
                        ϱ ⊨_lh may⟨[Cst_N (start_of π), Cst_E d]⟩.)"

theorem LV_sound:
  assumes "ϱ ⊨_lst bw_may.ana_pg_bw_may s_BV"
  shows "summarizes_LV ϱ"
proof -
  from assms have "bw_may.summarizes_bw_may ϱ"
    using bw_may.sound_ana_pg_bw_may[of ϱ] by auto
  then show ?thesis
    unfolding summarizes_LV_def bw_may.summarizes_bw_may_def edges_def edges_def
```

```
      end_def end_def use_path_S_hat_path by blast
qed

end

end
theory Available_Expressions imports Reaching_Definitions begin
```

— We encode the Available Expressions analysis into Datalog. First we define the analysis, then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

14 Available Expressions

```
fun ae_arith :: "'v arith ⇒ 'v arith set" where
  "ae_arith (Integer i) = {}"
| "ae_arith (Arith_Var v) = {}"
| "ae_arith (Arith_Op a1 opr a2) = ae_arith a1 ∪ ae_arith a1 ∪ {Arith_Op a1 opr a2}"
| "ae_arith (Minus a) = ae_arith a"

lemma finite_ae_arith: "finite (ae_arith a)"
  by (induction a) auto

fun ae_boolean :: "'v boolean ⇒ 'v arith set" where
  "ae_boolean true = {}"
| "ae_boolean false = {}"
| "ae_boolean (Rel_Op a1 opr a2) = ae_arith a1 ∪ ae_arith a2"
| "ae_boolean (Bool_Op b1 opr b2) = ae_boolean b1 ∪ ae_boolean b2"
| "ae_boolean (Neg b) = ae_boolean b"

lemma finite_ae_boolean: "finite (ae_boolean b)"
  using finite_ae_arith by (induction b) auto

fun aexp_action :: "'v action ⇒ 'v arith set" where
  "aexp_action (x ::= a) = ae_arith a"
| "aexp_action (Bool b) = ae_boolean b"
| "aexp_action Skip = {}"

lemma finite_aexp_action: "finite (aexp_action α)"
  using finite_ae_arith finite_ae_boolean by (cases α) auto

fun aexp_edge :: "('n,'v action) edge ⇒ 'v arith set" where
  "aexp_edge (q1, α, q2) = aexp_action α"

lemma finite_aexp_edge: "finite (aexp_edge (q1, α, q2))"
  using finite_aexp_action by auto

fun aexp_pg :: "('n,'v action) program_graph ⇒ 'v arith set" where
  "aexp_pg pg = ⋃(aexp_edge ‘ (fst pg))"

definition aexp_edge_list :: "('n,'v action) edge list ⇒ 'v arith ⇒ bool" where
  "aexp_edge_list π a = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧ a ∈ aexp_edge e ∧ (∀e' ∈ set ([e] @ π2). fv_arith
a ∩ def_edge e' = {}))"

definition aexp_path :: "'n list × 'v action list ⇒ 'v arith set" where
  "aexp_path π = {a. aexp_edge_list (transition_list π) a}"


locale analysis_AE = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges .
```

```
definition analysis_dom_AE :: "'v arith set" where
  "analysis_dom_AE = aexp_pg pg"

lemma finite_analysis_dom_AE: "finite analysis_dom_AE"
proof -
  have "finite (⋃ (aexp_edge ` edges))"
    by (metis aexp_edge.elims finite_UN finite_aexp_edge finite_program_graph_axioms
        finite_program_graph_def)
  then show ?thesis
    unfolding analysis_dom_AE_def using edges_def by force
qed

fun kill_set_AE :: "('n,'v action) edge ⇒ 'v arith set" where
  "kill_set_AE (q_o, x ::= a, q_s) = {a'. x ∈ fv_arith a'}"
| "kill_set_AE (q_o, Bool b, q_s) = {}"
| "kill_set_AE (v, Skip, vc) = {}"

fun gen_set_AE :: "('n,'v action) edge ⇒ 'v arith set" where
  "gen_set_AE (q_o, x ::= a, q_s) = {a'. a' ∈ ae_arith a ∧ x ∉ fv_arith a'}"
| "gen_set_AE (q_o, Bool b, q_s) = ae_boolean b"
| "gen_set_AE (v, Skip, vc) = {}"

definition d_init_AE :: "'v arith set" where
  "d_init_AE = {}"

interpretation fw_must: analysis_BV_forward_must pg analysis_dom_AE kill_set_AE gen_set_AE d_init_AE
  using analysis_BV_forward_must.intro analysis_AE_axioms analysis_AE_def
    d_init_AE_def empty_subsetI finite_analysis_dom_AE analysis_BV_forward_must_axioms.intro
  by metis

lemma aexp_edge_list_S_hat_edge_list:
  assumes "a ∈ aexp_edge (q, α, q')"
  assumes "fv_arith a ∩ def_edge (q, α, q') = {}"
  shows "a ∈ fw_must.S_hat (q, α, q') R"
  using assms unfolding fw_must.S_hat_def by (cases α) auto

lemma empty_inter_fv_arith_def_edge:
  assumes "aexp_edge_list (π @ [e]) a"
  shows "fv_arith a ∩ def_edge e = {}"
proof -
  from assms(1) obtain π1 π2 e' where π1_π2_e'_p:
    "π @ [e] = π1 @ [e'] @ π2"
    "a ∈ aexp_edge e'"
    "(∀e''∈set ([e'] @ π2). fv_arith a ∩ def_edge e'' = {})"
    unfolding aexp_edge_list_def by auto
  from this(1) have "e ∈ set ([e'] @ π2)"
    by (metis append_is_Nil_conv last_appendR last_in_set snoc_eq_iff_butlast)
  then show "fv_arith a ∩ def_edge e = {}"
    using π1_π2_e'_p(3) by auto
qed

lemma aexp_edge_list_append_singleton:
  assumes "aexp_edge_list (π @ [e]) a"
  shows "aexp_edge_list π a ∨ a ∈ aexp_edge e"
proof -
  from assms(1) obtain π1 π2 e' where π1_π2_e'_p:
    "π @ [e] = π1 @ [e'] @ π2"
    "a ∈ aexp_edge e'"
    "(∀e''∈set ([e'] @ π2). fv_arith a ∩ def_edge e'' = {})"
    unfolding aexp_edge_list_def by auto
  from this(1) have "e ∈ set ([e'] @ π2)"
    by (metis append_is_Nil_conv last_appendR last_in_set snoc_eq_iff_butlast)
  then have "e = e' ∨ e ∈ set π2"
```

```
    by auto
  then show ?thesis
  proof
    assume "e = e'"
    then have "a ∈ aexp_edge e"
      using π1_π2_e'_p by auto
    then show ?thesis
      by auto
  next
    assume "e ∈ set π2"
    then have "π = π1 @ [e'] @ (butlast π2)"
      by (metis π1_π2_e'_p(1) append_is_Nil_conv butlast_append butlast_snoc
          in_set_conv_decomp_first)
    moreover
    have "a ∈ aexp_edge e'"
      by (simp add: π1_π2_e'_p(2))
    moreover
    have "(∀e''∈set ([e'] @ butlast π2). fv_arith a ∩ def_edge e'' = {})"
      by (metis π1_π2_e'_p(3) butlast.simps(1) butlast_append in_set_butlastD)
    ultimately
    have "aexp_edge_list π a"
      unfolding aexp_edge_list_def by blast
    then show ?thesis
      by auto
  qed
qed

lemma gen_set_AE_subset_aexp_edge:
  assumes "a ∈ gen_set_AE e"
  shows "a ∈ aexp_edge e"
proof -
  obtain q α q' where "e = (q, α, q')"
    by (cases e)
  then show ?thesis
    using assms by (cases α) auto
qed

lemma empty_inter_fv_arith_def_edge':
  assumes "a ∈ gen_set_AE e"
  shows "fv_arith a ∩ def_edge e = {}"
proof -
  obtain q α q' where "e = (q, α, q')"
    by (cases e)
  then show ?thesis
    using assms by (cases α) auto
qed

lemma empty_inter_fv_arith_def_edge'':
  assumes "a ∉ kill_set_AE e"
  shows "fv_arith a ∩ def_edge e = {}"
proof -
  obtain q α q' where "e = (q, α, q')"
    by (cases e)
  then show ?thesis
    using assms by (cases α) auto
qed

lemma S_hat_edge_list_aexp_edge_list:
  assumes "a ∈ fw_must.S_hat_edge_list π d_init_AE"
  shows "aexp_edge_list π a"
  using assms
proof (induction π rule: rev_induct)
  case Nil
```

```
  then show ?case
    unfolding d_init_AE_def by auto
next
  case (snoc e π)
  from snoc(2) have "a ∈ (fw_must.S_hat_edge_list π d_init_AE - kill_set_AE e) ∨ a ∈ gen_set_AE e"
    using fw_must.S_hat_def by auto
  then show ?case
  proof
    assume a_S_hat: "a ∈ fw_must.S_hat_edge_list π d_init_AE - kill_set_AE e"
    then have "aexp_edge_list π a"
      using snoc by auto
    moreover
    from a_S_hat have "a ∉ kill_set_AE e"
      by auto
    then have "fv_arith a ∩ def_edge e = {}"
      using empty_inter_fv_arith_def_edge'' by auto
    ultimately show "aexp_edge_list (π @ [e]) a"
      unfolding aexp_edge_list_def by force
  next
    assume a_gen: "a ∈ gen_set_AE e"
    then have "a ∈ aexp_edge e"
      using gen_set_AE_subset_aexp_edge by auto
    moreover
    from a_gen have "(fv_arith a ∩ def_edge e = {})"
      using empty_inter_fv_arith_def_edge' by auto
    ultimately
    show "aexp_edge_list (π @ [e]) a"
      unfolding aexp_edge_list_def
      by (metis append_Nil2 empty_iff empty_set insert_iff list.set(2))
  qed
qed

lemma not_kill_set_AE_iff_fv_arith_def_edge_disjoint:
  "fv_arith a ∩ def_edge e = {} ⟷ a ∉ kill_set_AE e"
proof -
  obtain q α q' where e_split: "e = (q, α, q')"
    by (cases e)
  then show ?thesis
     by (cases α) auto
qed

lemma gen_set_AE_AE_iff_fv_arith_def_edge_disjoint_and_aexp_edge:
  "a ∈ aexp_edge e ∧ fv_arith a ∩ def_edge e = {} ⟷ a ∈ gen_set_AE e"
proof -
  obtain q α q' where e_split: "e = (q, α, q')"
    by (cases e)
  then show ?thesis
    by (cases α) auto
qed

lemma aexp_edge_list_S_hat_edge_list':
  assumes "aexp_edge_list π a"
  shows "a ∈ fw_must.S_hat_edge_list π d_init_AE"
  using assms
proof (induction π rule: rev_induct)
  case Nil
  then have False
    unfolding aexp_edge_list_def by auto
  then show ?case
    by metis
next
  case (snoc e π)
  have fvae: "fv_arith a ∩ def_edge e = {}"
```

```
    using snoc(2) empty_inter_fv_arith_def_edge by metis

  have "aexp_edge_list π a ∨ a ∈ aexp_edge e"
    using snoc(2)
    by (simp add: aexp_edge_list_append_singleton)
  then show ?case
  proof
    assume "aexp_edge_list π a"
    then have "a ∈ fw_must.S_hat_edge_list π d_init_AE"
      using snoc by auto
    moreover
    have "a ∉ kill_set_AE e"
      using fvae not_kill_set_AE_iff_fv_arith_def_edge_disjoint by auto
    ultimately
    show ?case
      using fw_must.S_hat_def by auto
  next
    assume "a ∈ aexp_edge e"
    then have "a ∈ gen_set_AE e"
      using fvae gen_set_AE_AE_iff_fv_arith_def_edge_disjoint_and_aexp_edge by metis
    then show ?case
      using fw_must.S_hat_def by auto
  qed
qed

lemma aexp_edge_list_S_hat_edge_list_iff:
  "aexp_edge_list π a ⟷ a ∈ fw_must.S_hat_edge_list π d_init_AE"
  using S_hat_edge_list_aexp_edge_list aexp_edge_list_S_hat_edge_list' by blast

lemma aexp_path_S_hat_path_iff:
  "a ∈ aexp_path π ⟷ a ∈ fw_must.S_hat_path π d_init_AE"
  using S_hat_edge_list_aexp_edge_list aexp_edge_list_S_hat_edge_list' aexp_path_def fw_must.S_hat_path_def
by blast

definition summarizes_AE :: "(pred, ('n, 'a, 'v arith) cst) pred_val ⇒ bool" where
   "summarizes_AE ϱ ⟷
     (∀q d.
         ϱ ⊨_lh must⟨[q, d]⟩. ⟶
          (∀π. π ∈ path_with_word_from_to start (the_Node_id q) ⟶ (the_Elem_id d) ∈ aexp_path π))"

theorem AE_sound:
  assumes "ϱ ⊨_lst (fw_must.ana_pg_fw_must) s_BV"
  shows "summarizes_AE ϱ"
proof -
  from assms have "fw_must.summarizes_fw_must ϱ"
    using fw_must.sound_ana_pg_fw_must by auto
  then show ?thesis
    unfolding summarizes_AE_def fw_must.summarizes_fw_must_def edges_def edges_def
      end_def end_def aexp_path_S_hat_path_iff start_def start_def by force
qed

end

end
theory Very_Busy_Expressions imports Available_Expressions begin
```

— We encode the Very Busy Expressions analysis into Datalog. First we define the analysis, then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

15 Very Busy Expressions

```
definition vbexp_edge_list :: "('n,'v action) edge list ⇒ 'v arith ⇒ bool" where
  "vbexp_edge_list π a = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧ a ∈ aexp_edge e ∧ (∀e' ∈ set π1. fv_arith
```

```
a ∩ def_edge e' = {}))"

definition vbexp_path :: "'n list × 'v action list ⇒ 'v arith set" where
  "vbexp_path π = {a. vbexp_edge_list (LTS.transition_list π) a}"

locale analysis_VB = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges .

definition analysis_dom_VB :: "'v arith set" where
  "analysis_dom_VB = aexp_pg pg"

lemma finite_analysis_dom_VB: "finite analysis_dom_VB"
proof -
  have "finite (⋃ (aexp_edge ` edges))"
    by (metis aexp_edge.elims finite_UN finite_aexp_edge finite_program_graph_axioms
        finite_program_graph_def)
  then show ?thesis
    unfolding analysis_dom_VB_def edges_def by auto
qed

fun kill_set_VB :: "('n,'v action) edge ⇒ 'v arith set" where
  "kill_set_VB (q_o, x ::= a, q_s) = {a'. x ∈ fv_arith a'}"
| "kill_set_VB (q_o, Bool b, q_s) = {}"
| "kill_set_VB (v, Skip, vc) = {}"

fun gen_set_VB :: "('n,'v action) edge ⇒ 'v arith set" where
  "gen_set_VB (q_o, x ::= a, q_s) = ae_arith a"
| "gen_set_VB (q_o, Bool b, q_s) = ae_boolean b"
| "gen_set_VB (v, Skip, vc) = {}"

definition d_init_VB :: "'v arith set" where
  "d_init_VB = {}"

interpretation bw_must: analysis_BV_backward_must pg analysis_dom_VB kill_set_VB gen_set_VB d_init_VB
  using analysis_VB_axioms analysis_VB_def
  by (metis analysis_BV_backward_must_axioms_def analysis_BV_backward_must_def bot.extremum
      d_init_VB_def finite_analysis_dom_VB)

lemma aexp_edge_list_S_hat_edge_list:
  assumes "a ∈ aexp_edge (q, α, q')"
  assumes "fv_arith a ∩ def_edge (q, α, q') = {}"
  shows "a ∈ bw_must.S_hat (q, α, q') R"
  using assms unfolding bw_must.S_hat_def by (cases α) auto

lemma empty_inter_fv_arith_def_edge:
  assumes "vbexp_edge_list (e # π) a"
  shows "fv_arith a ∩ def_edge e = {} ∨ a ∈ aexp_edge e"
proof -
  from assms(1) obtain π1 π2 e' where π1_π2_e'_p:
    "e # π = π1 @ [e'] @ π2"
    "a ∈ aexp_edge e'"
    "(∀e''∈set π1. fv_arith a ∩ def_edge e'' = {})"
    unfolding vbexp_edge_list_def by auto
  from this(1) have "e ∈ set (π1 @ [e'])"
    by (metis append_self_conv2 hd_append2 list.sel(1) list.set_sel(1) snoc_eq_iff_butlast)
  then have "e ∈ set π1 ∨ e = e'"
    by auto
  then show "fv_arith a ∩ def_edge e = {} ∨ a ∈ aexp_edge e"
  proof
    assume "e ∈ set π1"
```

```
    then have "fv_arith a ∩ def_edge e = {}"
      using π1_π2_e'_p(3) by auto
    then show "fv_arith a ∩ def_edge e = {} ∨ a ∈ aexp_edge e"
       by auto
  next
    assume "e = e''"
    then have "a ∈ aexp_edge e"
      using π1_π2_e'_p(2) by auto
    then show "fv_arith a ∩ def_edge e = {}  ∨ a ∈ aexp_edge e"
      by auto
  qed
qed

lemma vbexp_edge_list_Cons:
  assumes "vbexp_edge_list (e # π) a"
  shows "vbexp_edge_list π a ∨ a ∈ aexp_edge e"
proof -
  from assms(1) obtain π1 π2 e' where π1_π2_e'_p:
    "e # π = π1 @ [e'] @ π2"
    "a ∈ aexp_edge e'"
    "(∀e''∈set π1. fv_arith a ∩ def_edge e'' = {})"
    unfolding vbexp_edge_list_def by auto
  from this(1) have "e ∈ set (π1 @ [e'])"
    by (metis append_assoc hd_append2 list.sel(1) list.set_sel(1) snoc_eq_iff_butlast)
  then have "e = e' ∨ e ∈ set π1"
    by auto
  then show ?thesis
  proof
    assume "e = e''"
    then have "a ∈ aexp_edge e"
      using π1_π2_e'_p by auto
    then show ?thesis
      by auto
  next
    assume "e ∈ set π1"
    then have "π = tl π1 @ [e'] @ π2"
      using π1_π2_e'_p by (metis equals0D list.sel(3) set_empty tl_append2)
    moreover
    have "a ∈ aexp_edge e'"
      by (simp add: π1_π2_e'_p(2))
    moreover
    have "(∀e''∈ set (tl π1). fv_arith a ∩ def_edge e'' = {})"
      by (metis π1_π2_e'_p(3) list.set_sel(2) tl_Nil)
    ultimately
    have "vbexp_edge_list π a"
      unfolding vbexp_edge_list_def by metis
    then show ?thesis
      by auto
  qed
qed

lemma gen_set_VB_is_aexp_edge:
  "a ∈ gen_set_VB e ⟷ a ∈ aexp_edge e"
proof -
  obtain q α q' where e_split: "e = (q, α, q')"
    by (cases e)
  then show ?thesis
    by (cases α) auto
qed

lemma empty_inter_fv_arith_def_edge'':
  "a ∉ kill_set_VB e ⟷ fv_arith a ∩ def_edge e = {}"
proof -
```

```
  obtain q α q' where e_split: "e = (q, α, q')"
    by (cases e)
  then show ?thesis
    by (cases α) auto
qed

lemma S_hat_edge_list_aexp_edge_list:
  assumes "a ∈ bw_must.S_hat_edge_list π d_init_VB"
  shows "vbexp_edge_list π a"
  using assms
proof (induction π)
  case Nil
  then show ?case
    unfolding d_init_VB_def by auto
next
  case (Cons e π)
  from Cons(2) have "a ∈ (bw_must.S_hat_edge_list π d_init_VB - kill_set_VB e) ∨ a ∈ gen_set_VB e"
    using bw_must.S_hat_def by auto
  then show ?case
  proof
    assume a_S_hat: "a ∈ bw_must.S_hat_edge_list π d_init_VB - kill_set_VB e"
    then have "vbexp_edge_list π a"
      using Cons by auto
    moreover
    from a_S_hat have "a ∉ kill_set_VB e"
      by auto
    then have "fv_arith a ∩ def_edge e = {}"
      using empty_inter_fv_arith_def_edge'' by auto
    ultimately show "vbexp_edge_list (e # π) a"
      unfolding vbexp_edge_list_def by (metis Cons_eq_appendI set_ConsD)
  next
    assume a_gen: "a ∈ gen_set_VB e"
    then have "a ∈ aexp_edge e"
      using gen_set_VB_is_aexp_edge by auto
    then show "vbexp_edge_list (e # π) a"
      unfolding vbexp_edge_list_def by (metis append_Cons append_Nil empty_iff empty_set)
  qed
qed

lemma aexp_edge_list_S_hat_edge_list':
  assumes "vbexp_edge_list π a"
  shows "a ∈ bw_must.S_hat_edge_list π d_init_VB"
  using assms
proof (induction π)
  case Nil
  then have False
    unfolding vbexp_edge_list_def by auto
  then show ?case
    by metis
next
  case (Cons e π)
  have fvae: "fv_arith a ∩ def_edge e = {} ∨ a ∈ aexp_edge e"
    using Cons(2) empty_inter_fv_arith_def_edge by force

  have "vbexp_edge_list π a ∨ a ∈ aexp_edge e"
    using Cons(2)
    by (simp add: vbexp_edge_list_Cons)
  then show ?case
  proof
    assume "vbexp_edge_list π a"
    then have "a ∈ bw_must.S_hat_edge_list π d_init_VB"
      using Cons by auto
    moreover
```

```
    have "a ∉ kill_set_VB e ∨ a ∈ gen_set_VB e"
      using fvae empty_inter_fv_arith_def_edge'' gen_set_VB_is_aexp_edge by auto
    ultimately
    show ?case
      using bw_must.S_hat_def by auto
  next
    assume "a ∈ aexp_edge e"
    then have "a ∈ gen_set_VB e"
      using fvae empty_inter_fv_arith_def_edge'' gen_set_VB_is_aexp_edge by auto
    then show ?case
      using bw_must.S_hat_def by auto
  qed
qed

lemma vbexp_edge_list_S_hat_edge_list_iff:
  "vbexp_edge_list π a ⟷ a ∈ bw_must.S_hat_edge_list π d_init_VB"
  using S_hat_edge_list_aexp_edge_list aexp_edge_list_S_hat_edge_list' by blast

lemma vbexp_path_S_hat_path_iff:
  "a ∈ vbexp_path π ⟷ a ∈ bw_must.S_hat_path π d_init_VB"
  by (simp add: bw_must.S_hat_path_def vbexp_edge_list_S_hat_edge_list_iff vbexp_path_def)

definition summarizes_VB where
  "summarizes_VB ϱ ⟷
     (∀q d.
         ϱ ⊨_lh must⟨[q, d]⟩. ⟶
          (∀π. π ∈ path_with_word_from_to (the_Node_id q) end ⟶ the_Elem_id d ∈ vbexp_path π))"

theorem VB_sound:
  assumes "ϱ ⊨_lst (bw_must.ana_pg_bw_must) s_BV"
  shows "summarizes_VB ϱ"
proof -
  from assms have "bw_must.summarizes_bw_must ϱ"
    using bw_must.sound_ana_pg_bw_must by auto
  then show ?thesis
    unfolding summarizes_VB_def bw_must.summarizes_bw_must_def edges_def edges_def
      end_def end_def vbexp_path_S_hat_path_iff start_def start_def by force
qed

end

end
theory Reachable_Nodes imports Bit_Vector_Framework begin
```

— We define an analysis for collecting the reachable nodes in a program graph. First we define it analysis, and then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

16 Reachable Nodes

```
fun nodes_on_edge :: "('n,'v) edge ⇒ 'n set" where
  "nodes_on_edge (q1, α, q2) = {q1, q2}"

definition node_on_edge_list :: "('n,'v) edge list ⇒ 'n ⇒ bool" where
  "node_on_edge_list π q = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧ q ∈ nodes_on_edge e)"

definition nodes_on_path :: "'n list × 'v action list ⇒ 'n set" where
  "nodes_on_path π = {q. node_on_edge_list (LTS.transition_list π) q}"

locale analysis_RN = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges .
```

```
definition analysis_dom_RN :: "'n set" where
  "analysis_dom_RN = UNIV"

fun kill_set_RN :: "('n,'v action) edge ⇒ 'n set" where
  "kill_set_RN (q_o, α, q_s) = {}"

fun gen_set_RN :: "('n,'v action) edge ⇒ 'n set" where
  "gen_set_RN (q_o, α, q_s) = {q_o}"

definition d_init_RN :: "'n set" where
  "d_init_RN = {end}"

interpretation bw_may: analysis_BV_backward_may pg analysis_dom_RN kill_set_RN gen_set_RN d_init_RN
  using analysis_BV_backward_may.intro[of pg analysis_dom_RN kill_set_RN gen_set_RN d_init_RN]
    analysis_BV_backward_may_axioms_def[of pg analysis_dom_RN] finite_program_graph_axioms
    finite_program_graph_axioms finite_program_graph_def[of pg] analysis_dom_RN_def by auto

lemma node_on_edge_list_S_hat_edge_list:
  assumes "ts ∈ transition_list_path"
  assumes "trans_tl (last ts) = end"
  assumes "node_on_edge_list ts q"
  shows "q ∈ bw_may.S_hat_edge_list ts d_init_RN"
  using assms
proof (induction rule: LTS.transition_list_path.induct[OF assms(1)])
  case (1 q' l q'')
  then show ?case
    by (simp add: d_init_RN_def Cons_eq_append_conv bw_may.S_hat_def node_on_edge_list_def)
next
  case (2 q' l q'' l' q''' ts)
  from 2(6) obtain π1 π2 e where
    "(q', l, q'') # (q'', l', q''') # ts = π1 @ [e] @ π2"
    "q ∈ nodes_on_edge e"
    unfolding node_on_edge_list_def by auto
  then have "e = (q', l, q'') ∨ e ∈ set ((q'', l', q''') # ts)"
    by (metis (no_types, lifting) append_Cons hd_append in_set_conv_decomp list.sel(1) list.sel(3) tl_append2)
  then show ?case
  proof
    assume "e = (q', l, q'')"
    then have "q = q' ∨ q = q''"
      using ‹q ∈ nodes_on_edge e› by auto
    then show ?case
    proof
      assume "q = q'"
      then show ?case
        unfolding bw_may.S_hat_edge_list.simps bw_may.S_hat_def by auto
    next
      assume "q = q''"
      then have "q ∈ bw_may.S_hat_edge_list ((q'', l', q''') # ts) d_init_RN"
        using "2.IH" "2.hyps"(2) "2.prems"(2) node_on_edge_list_def by fastforce
      then show ?case
        unfolding bw_may.S_hat_edge_list.simps bw_may.S_hat_def by auto
    qed
  next
    assume "e ∈ set ((q'', l', q''') # ts)"
    then have "q ∈ bw_may.S_hat_edge_list ((q'', l', q''') # ts) d_init_RN"
      by (metis "2.IH" "2.hyps"(2) "2.prems"(2) ‹q ∈ nodes_on_edge e› append_Cons append_Nil in_set_conv_decomp
last.simps list.distinct(1) node_on_edge_list_def)
    then show ?case
      unfolding bw_may.S_hat_edge_list.simps bw_may.S_hat_def by auto
  qed
qed
```

```
lemma S_hat_edge_list_node_on_edge_list:
  assumes "π ≠ []"
  assumes "trans_tl (last π) = end"
  assumes "q ∈ bw_may.S_hat_edge_list π d_init_RN"
  shows "node_on_edge_list π q"
  using assms
proof (induction π)
  case Nil
  then show ?case
    by auto
next
  case (Cons e π)
  from Cons(4) have
    "q ∈ bw_may.S_hat_edge_list π d_init_RN - kill_set_RN e ∨
     q ∈ gen_set_RN e"
    using bw_may.S_hat_def by auto
  then show ?case
  proof
    assume q_Shat: "q ∈ bw_may.S_hat_edge_list π d_init_RN - kill_set_RN e"
    have "π ≠ [] ∨ π = []"
      by auto
    then show ?thesis
    proof
      assume "π ≠ []"
      then show "node_on_edge_list (e # π) q"
        using Cons(1,3)
        by (metis Diff_iff q_Shat append_Cons last.simps node_on_edge_list_def)
    next
      assume "π = []"
      then show "node_on_edge_list (e # π) q"
        using d_init_RN_def q_Shat
        by (metis Cons.prems(2) Diff_empty append.left_neutral append_Cons bw_may.S_hat_edge_list.simps(1)
insertI1 insert_commute kill_set_RN.elims last_ConsL nodes_on_edge.elims node_on_edge_list_def singleton_iff
trans_tl.simps)
 qed
  next
    assume "q ∈ gen_set_RN e"
    then show ?thesis
      unfolding node_on_edge_list_def
      by (metis append.left_neutral append_Cons empty_iff gen_set_RN.elims insert_iff nodes_on_edge.simps)
  qed
qed

lemma node_on_edge_list_UNIV_S_hat_edge_list:
  assumes "π ∈ transition_list_path"
  assumes "π ≠ []"
  assumes "trans_tl (last π) = end"
  shows "{q. node_on_edge_list π q} = bw_may.S_hat_edge_list π d_init_RN"
  using assms node_on_edge_list_S_hat_edge_list S_hat_edge_list_node_on_edge_list by auto

lemma nodes_singleton_if_path_with_word_empty':
  assumes "(ss, w) ∈ path_with_word"
  assumes "w = []"
  shows "∃l. ss = [l]"
using assms proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case
    by auto
next
  case (2 s' ss w s l)
  then show ?case
    by auto
```

```
qed

lemma nodes_singleton_if_path_with_word_empty:
  assumes "(ss, []) ∈ path_with_word"
  shows "∃l. ss = [l]"
  using nodes_singleton_if_path_with_word_empty' assms by auto

lemma path_with_word_append_last:
  assumes "(ss,w) ∈ path_with_word"
  assumes "w ≠ []"
  shows "last (transition_list (s # ss, l # w)) = last (transition_list (ss, w))"
  using assms proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s')
  then show ?case
    by auto
next
  case (2 s'' ss w s' l)
  then show ?case
    by auto
qed

lemma last_trans_tl:
  assumes "(ss,w) ∈ path_with_word"
  assumes "w ≠ []"
  shows "last ss = trans_tl (last (LTS.transition_list (ss,w)))"
using assms proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case
    by auto
next
  case (2 s' ss w s l)
  show ?case
  proof (cases "w = []")
    case True
    then have "ss = []"
      using 2 nodes_singleton_if_path_with_word_empty by auto
    then show ?thesis
      using True 2
      by auto
  next
    case False
    from 2 have "(s' # ss, w) ∈ path_with_word"
      by auto
    moreover
    have "w ≠ []"
      using False by auto
    ultimately
    have "last (s' # ss) = trans_tl (last (transition_list (s' # ss, w)))"
      using 2 by auto
    moreover
    have "last (transition_list (s' # ss, w)) =
        last (transition_list (s # s' # ss, l # w))"
      using path_with_word_append_last[of "s' # ss" w s l] ‹w ≠ []›
      using ‹(s' # ss, w) ∈ LTS.path_with_word edges› by auto
    ultimately
    show ?thesis
      by auto
  qed
qed

lemma nodes_tail_empty_if_path_with_word_empty:
  assumes "(ss,w) ∈ path_with_word"
  assumes "w ≠ []"
```

```
  shows "transition_list (ss,w) ≠ []"
using assms proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case by auto
next
  case (2 s' ss w s l)
  then show ?case
    by auto
qed

lemma empty_transition_list_if_empty_word:
  assumes "π ∈ path_with_word"
  assumes "snd π ≠ []"
  shows "transition_list π ≠ []"
  using assms nodes_tail_empty_if_path_with_word_empty by (cases π) auto

lemma two_nodes_if_nonempty_word:
  assumes "(ss, w) ∈ path_with_word"
  assumes "w ≠ []"
  shows "∃s' s'' ss'. ss = s' # s'' # ss'"
using assms
proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case
    by auto
next
  case (2 s' ss w s l)
  then show ?case
    by auto
qed

lemma path_with_word_empty_word:
  assumes "(ss', w) ∈ path_with_word"
  assumes "ss' = s' # ss"
  assumes "w = []"
  shows "ss = []"
using assms
proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case
    by auto
next
  case (2 s' ss w s l)
  then show ?case
    by auto
qed

lemma transition_list_path_if_path_with_word:
  assumes "(ss,w) ∈ path_with_word"
  assumes "w ≠ []"
  shows "transition_list (ss,w) ∈ transition_list_path"
  using assms
proof (induction rule: LTS.path_with_word.induct[OF assms(1)])
  case (1 s)
  then show ?case
    by auto
next
  case (2 s' ss w s l)
  then show ?case
  proof (cases "w = []")
    case True
    then have s_empt: "ss = []"
      using 2(1) path_with_word_empty_word by blast
```

```
    from 2 have "[(s, l, s')] ∈ transition_list_path"
      using LTS.transition_list_path.intros(1)[of s l s' edges] by auto
    then show ?thesis
      using True s_empt by auto
  next
    case False
    then obtain l' w' where w_eql: "w = l' # w'"
      by (cases w) auto
    obtain s'' ss' where ss_eql: "ss = s'' # ss'"
      using 2(1) False two_nodes_if_nonempty_word[of "s' #ss" w] by auto
    have "(s, l, s') ∈ edges"
      using 2 by auto
    moreover
    have "(s', l', s'') # transition_list (s'' # ss', w') ∈ transition_list_path"
      using 2 w_eql ss_eql by auto
    ultimately
    have "(s, l, s') # (s', l', s'') # transition_list (s'' # ss', w')
          ∈ transition_list_path"
      by (rule LTS.transition_list_path.intros(2)[of s l s'])
    then show ?thesis
      unfolding ss_eql w_eql by simp
  qed
qed

lemma nodes_on_path_S_hat_path:
  assumes "π ∈ path_with_word"
  assumes "snd π ≠ []"
  assumes "last (fst π) = end"
  shows "nodes_on_path π = bw_may.S_hat_path π d_init_RN"
proof -
  have "trans_tl (last (LTS.transition_list π)) = end"
    using assms(1,2,3) last_trans_tl[of "fst π" "snd π"] by auto
  moreover
  have "LTS.transition_list π ≠ []"
    using assms(1,2) empty_transition_list_if_empty_word using assms by auto
  moreover
  have "(LTS.transition_list π) ∈ transition_list_path"
    using assms(1) assms(2) transition_list_path_if_path_with_word[of "fst π" "snd π"]
    by auto
  ultimately
  show ?thesis
    by (simp add: bw_may.S_hat_path_def node_on_edge_list_UNIV_S_hat_edge_list nodes_on_path_def)
qed

definition summarizes_RN where
  "summarizes_RN ϱ ⟷ (∀π d. π ∈ path_with_word_to end ⟶ d ∈ nodes_on_path π ⟶
                        ϱ ⊨_lh may⟨[Cst_N (start_of π), Cst_E d]⟩.)"

theorem RN_sound:
  assumes "ϱ ⊨_lst bw_may.ana_pg_bw_may s_BV"
  shows "summarizes_RN ϱ"
proof -
  from assms have summary: "bw_may.summarizes_bw_may ϱ"
    using bw_may.sound_ana_pg_bw_may[of ϱ] by auto
  {
    fix π d
    assume π_path_to_end: "π ∈ path_with_word_to end"
    assume d_on_path: "d ∈ nodes_on_path π"
    have π_path: "π ∈ LTS.path_with_word edges"
      using π_path_to_end edges_def edges_def by auto
    have π_end: "end_of π = end"
      using π_path_to_end end_def end_def by fastforce
```

```
    then have "last (fst π) = end"
      using end_def end_def end_of_def by auto
    then have "d ∈ bw_may.S_hat_path π d_init_RN"
      using π_path_to_end d_on_path nodes_on_path_S_hat_path[of π] Nil_is_append_conv list.discI
        mem_Collect_eq node_on_edge_list_def nodes_on_path_def prod.exhaust_sel
        transition_list.simps(2) nodes_singleton_if_path_with_word_empty
      by (metis (mono_tags, lifting))
    then have "ϱ ⊨lh may⟨[Cst_N (start_of π), Cst_E d]⟩."
      using π_path π_end summary unfolding bw_may.summarizes_bw_may_def mem_Collect_eq by metis
  }

  then show ?thesis
    unfolding summarizes_RN_def by auto
qed

end

end
```

References

[BCD17] Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. Certifying standard and stratified Datalog inference engines in SSReflect. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2017.

[BDA18] Angela Bonifati, Stefania Dumbrava, and Emilio Jesús Gallego Arias. Certified graph view maintenance with Regular Datalog. *Theory Pract. Log. Program.*, 18(3-4):372–389, 2018.

[Bre10] Joachim Breitner. Shivers' control flow analysis. *Archive of Formal Proofs*, November 2010. https://isa-afp.org/entries/Shivers-CFA.html, Formal proof development.

[Bre15] Joachim Breitner. The safety of Call Arity. *Archive of Formal Proofs*, February 2015. https://isa-afp.org/entries/Call_Arity.html, Formal proof development.

[Bre16] Joachim Breitner. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation*. PhD thesis, Karlsruhe Institute of Technology, 2016.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.

[CP10] David Cachera and David Pichardie. A certified denotational abstract interpreter. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2010.

[Dum16] Stefania-Gabriela Dumbrava. *A Coq Formalization of Relational and Deductive Databases -and Mechanizations of Datalog*. PhD thesis, University of Paris-Saclay, France, 2016.

[Imm11] Fabian Immler. RIPEMD-160. *Archive of Formal Proofs*, January 2011. https://isa-afp.org/entries/RIPEMD-160-SPARK.html, Formal proof development.

[Jia21] Nan Jiang. A data flow analysis algorithm for computing dominators. *Archive of Formal Proofs*, September 2021. https://isa-afp.org/entries/Dominance_CHK.html, Formal proof development.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL 2015)*, pages 247–259, 2015.

[KKB13] Jael Kriener, Andy King, and Sandrine Blazy. Proofs you can believe in: proving equivalences between Prolog semantics in Coq. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 37–48. ACM, 2013.

[LM08] Peter Lammich and Markus Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2008.

[LMO07] Peter Lammich and Markus Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. *Archive of Formal Proofs*, December 2007. https://isa-afp.org/entries/Program-Conflict-Analysis.html, Formal proof development.

[Nip12] Tobias Nipkow. Abstract interpretation of annotated commands. In *Proceedings of the Third International Conference on Interactive Theorem Proving (ITP 2012)*, pages 116–132, 2012.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[NN20] Flemming Nielson and Hanne Riis Nielson. Program analysis (an appetizer). *CoRR*, abs/2012.10086, 2020.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[Prz88] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[SHN24] Anders Schlichtkrull, René Rydhof Hansen, and Flemming Nielson. Isabelle-verified correctness of datalog programs for program analysis. In Jiman Hong and Juw Won Park, editors, *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024*, pages 1731–1734. ACM, 2024.

[SSST22] Anders Schlichtkrull, Morten Konggaard Schou, Jirí Srba, and Dmitriy Traytel. Differential testing of pushdown reachability with a formally verified oracle. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 369–379. IEEE, 2022.

[SSST23] Anders Schlichtkrull, Morten Konggaard Schou, Jií Srba, and Dmitriy Traytel. Labeled transition systems. *Archive of Formal Proofs*, October 2023. https://isa-afp.org/entries/Labeled_Transition_Systems.html, Formal proof development.

[TGMK25] Johannes Tantow, Lukas Gerlach, Stephan Mennicke, and Markus Krötzsch. Verifying Datalog reasoning with Lean. In *Proceedings of the 16th International Conference on Interactive Theorem Proving (ITP 2025)*, 2025.

[Was08] Daniel Wasserrab. Towards certified slicing. *Archive of Formal Proofs*, September 2008. https://isa-afp.org/entries/Slicing.html, Formal proof development.

[Was09] Daniel Wasserrab. Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. *Archive of Formal Proofs*, November 2009. https://isa-afp.org/entries/HRB-Slicing.html, Formal proof development.

[Whi06] Nathan Whitehead. A certified distributed security logic for authorizing code. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2006.

[WL08] Daniel Wasserrab and Andreas Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in isabelle/hol. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2008.

[WLS09] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-based noninterference and its modular proof. In Stephen Chong and David A. Naumann, editors, *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 31–44. ACM, 2009.