

Graph Saturation

Sebastiaan J. C. Joosten

April 14, 2026

Abstract

This is an Isabelle/HOL formalisation of graph saturation, closely following a paper by the author on graph saturation [2]. Nine out of ten lemmas of the original paper are proven in this formalisation. The formalisation additionally includes two theorems that show the main premise of the paper: that consistency and entailment are decided through graph saturation. This formalisation does not give executable code, and it did not implement any of the optimisations suggested in the paper.

Contents

1	Introduction	1
2	Labeled Graphs	2
3	Rules, and the chains we can make with them	15
4	Graph rewriting and saturation	31
5	Semantics in labeled graphs	44
6	Standard Models	48
7	Translating terms into Graphs	49
8	Standard Rules	63
9	Combined correctness	82

1 Introduction

Although the formalisation follows a paper by the author on graph saturation [2], it is foremost a formalisation. This document highlights the differences, where applicable. Nevertheless, the reader is advised to start by read-

ing [2]. A copy might be available on <http://sjcjoosten.nl/4-publications/joosten18/>.

The first publication of this graph saturation algorithm is in [1]. While that paper contains a somewhat more category-theoretical view, it also has fewer proofs and less rigor. Graph Saturation was originally developed to potentially benefit the Ampersand compiler [4].

2 Labeled Graphs

We define graphs as in the paper. Graph homomorphisms and subgraphs are defined slightly differently. Their correspondence to the definitions in the paper is given by separate lemmas. After defining graphs, we only talk about the semantics until after defining homomorphisms. The reason is that graph rewriting can be done without knowing about semantics.

```
theory LabeledGraphs
imports MissingRelation
begin
```

```
datatype ('l,'v) labeled_graph
  = LG (edges:"('l × 'v × 'v) set") (vertices:"'v set")
```

```
definition restrict where
```

```
"restrict G = LG {(l,v1,v2) ∈ edges G. v1 ∈ vertices G ∧ v2 ∈ vertices G } (vertices G)"
```

Definition 1. We define graphs and show that any graph with no edges (in particular the empty graph) is indeed a graph.

```
abbreviation graph where
  "graph X ≡ X = restrict X"
```

```
lemma graph_empty_e[intro]: "graph (LG {} v)" unfolding restrict_def
by auto
```

```
lemma graph_single[intro]: "graph (LG {(a,b,c)} {b,c})" unfolding restrict_def
by auto
```

```
abbreviation finite_graph where
```

```
"finite_graph X ≡ graph X ∧ finite (vertices X) ∧ finite (edges X)"
```

```
lemma restrict_idemp[simp]:
  "restrict (restrict x) = restrict x"
by(cases x,auto simp:restrict_def)
```

```
lemma vertices_restrict[simp]:
  "vertices (restrict G) = vertices G"
by(cases G,auto simp:restrict_def)
```

```

lemma restrictI[intro]:
  assumes "edges G  $\subseteq$  {(l,v1,v2). v1  $\in$  vertices G  $\wedge$  v2  $\in$  vertices G
  }"
  shows "G = restrict G"
  using assms by(cases G,auto simp:restrict_def)

lemma restrict_subsd[dest]:
  assumes "edges G  $\subseteq$  edges (restrict G)"
  shows "G = restrict G"
  using assms by(cases G,auto simp:restrict_def)

lemma restrictD[dest]:
  assumes "G = restrict G"
  shows "edges G  $\subseteq$  {(l,v1,v2). v1  $\in$  vertices G  $\wedge$  v2  $\in$  vertices G }"
proof -
  have "edges (restrict G)  $\subseteq$  {(l,v1,v2). v1  $\in$  vertices G  $\wedge$  v2  $\in$  vertices
  G }"
  by (cases G,auto simp:restrict_def)
  thus ?thesis using assms by auto
qed

definition on_triple where "on_triple R  $\equiv$  {(l,s,t),(l',s',t')} . l=l'
 $\wedge$  (s,s')  $\in$  R  $\wedge$  (t,t')  $\in$  R}"

lemma on_triple[simp]:
  " $((l1,v1,v2),(l2,v3,v4)) \in$  on_triple R  $\longleftrightarrow$  (v1,v3) $\in$  R  $\wedge$  (v2,v4)  $\in$ 
  R  $\wedge$  l1 = l2"
unfolding on_triple_def by auto

lemma on_triple_univ[intro!]:
  "univalent f  $\implies$  univalent (on_triple f)"
unfolding on_triple_def univalent_def by auto

lemma on_tripleD[dest]:
  assumes " $((l1,v1,v2),(l2,v3,v4)) \in$  on_triple R"
  shows "l2 = l1" "(v1,v3) $\in$  R" "(v2,v4)  $\in$  R"
using assms unfolding on_triple_def by auto

lemma on_triple_ID_restrict[simp]:
  shows "on_triple (Id_on (vertices G)) ‘‘ edges G = edges (restrict G)’’"
unfolding on_triple_def by(cases G,auto simp:restrict_def)

lemma relcomp_on_triple[simp]:
  shows "on_triple (R  $\circ$  S) = on_triple R  $\circ$  on_triple S"
unfolding on_triple_def by fast

lemma on_triple_preserves_finite[intro]:
  "finite E  $\implies$  finite (on_triple (BNF_Def.Gr A f) ‘‘ E)'"

```

```

by (auto simp:on_triple_def BNF_Def.Gr_def)

lemma on_triple_fst[simp]:
  assumes "vertices G = Domain g" "graph G"
  shows "x ∈ fst ` on_triple g `` (edges G) ⟷ x ∈ fst ` edges G"
proof
  assume "x ∈ fst ` on_triple g `` edges G"
  then obtain a b where "(x,a,b) ∈ on_triple g `` edges G" by force
  then obtain c d where "(x,c,d) ∈ edges G" unfolding on_triple_def
  by auto
  thus "x ∈ fst ` edges G" by force next
  assume "x ∈ fst ` edges G"
  then obtain a b where ab:"(x,a,b) ∈ edges G" by force
  then obtain c d where "(a,c) ∈ g" "(b,d) ∈ g" using assms by force
  hence "(x,c,d) ∈ on_triple g `` edges G" using ab unfolding on_triple_def
  by auto
  thus "x ∈ fst ` on_triple g `` edges G" by (metis fst_conv image_iff)
qed

definition edge_preserving where
  "edge_preserving h e1 e2 ≡
  (∀ (k,v1,v2) ∈ e1. ∀ v1' v2'. ((v1, v1') ∈ h ∧ (v2,v2') ∈ h)
  → (k,v1',v2') ∈ e2)"

lemma edge_preserving_atomic:
  assumes "edge_preserving h1 e1 e2" "(v1, v1') ∈ h1" "(v2, v2') ∈ h1"
  "(k, v1, v2) ∈ e1"
  shows "(k, v1', v2') ∈ e2"
using assms unfolding edge_preserving_def by auto

lemma edge_preservingI[intro]:
  assumes "on_triple R `` E ⊆ G"
  shows "edge_preserving R E G"
  unfolding edge_preserving_def proof(clarify,goal_cases)
  case (1 a s t v1' v2')
  thus ?case by (intro assms[THEN subsetD]) (auto simp:on_triple_def)
  qed

lemma on_triple_dest[dest]:
  assumes "on_triple R `` E ⊆ G"
  "(l,x,y) ∈ E" "(x,xx) ∈ R" "(y,yy) ∈ R"
  shows "(l,xx,yy) ∈ G"
  using assms unfolding Image_def on_triple_def by blast

lemma edge_preserving:
  shows "edge_preserving R E G ⟷ on_triple R `` E ⊆ G"
proof
  assume "edge_preserving R E G"
  hence "∧ k v1 v2 v1' v2'. (k, v1, v2) ∈ E ⟹

```

```

      (v1, v1') ∈ R ⇒ (v2, v2') ∈ R ⇒ (k, v1', v2') ∈ G"
    unfolding edge_preserving_def by auto
  then show "on_triple R '' E ⊆ G" unfolding Image_def by auto
qed auto

```

```

lemma edge_preserving_subset:
  assumes "R1 ⊆ R2" "E1 ⊆ E2" "edge_preserving R2 E2 G"
  shows "edge_preserving R1 E1 G"
  using assms unfolding edge_preserving_def by blast

```

```

lemma edge_preserving_unionI[intro]:
  assumes "edge_preserving f A G" "edge_preserving f B G"
  shows "edge_preserving f (A ∪ B) G"
  using assms unfolding edge_preserving_def by blast

```

```

lemma compose_preserves_edge_preserving:
  assumes "edge_preserving h1 e1 e2" "edge_preserving h2 e2 e3"
  shows "edge_preserving (h1 ∘ h2) e1 e3" unfolding edge_preserving_def
proof(standard,standard,standard,standard,standard,standard,goal_cases)
  case (1 _ k _ v1 v2 v1'' v2'')
  hence 1:"(k, v1, v2) ∈ e1" "(v1, v1'') ∈ h1 ∘ h2" "(v2, v2'') ∈ h1
  ∘ h2" by auto
  then obtain v1' v2' where
    v:"(v1,v1') ∈ h1" "(v1',v1'') ∈ h2" "(v2,v2') ∈ h1" "(v2',v2'') ∈
  h2" by auto
  from edge_preserving_atomic[OF assms(1) v(1,3) 1(1)]
    edge_preserving_atomic[OF assms(2) v(2,4)]
  show ?case by metis
qed

```

```

lemma edge_preserving_Id[intro]: "edge_preserving (Id_on y) x x"
unfolding edge_preserving_def by auto

```

This is an alternate version of definition 10. We require $\text{@termvertices } s = \text{Domain } h$ to ensure that graph homomorphisms are sufficiently unique: The partiality follows the definition in the paper, per the remark before Def. 7. but it means that we cannot use Isabelle's total functions for the homomorphisms. We show that graph homomorphisms and embeddings coincide in a separate lemma.

```

definition graph_homomorphism where
  "graph_homomorphism G1 G2 f
  = ( vertices G1 = Domain f
    ∧ graph G1 ∧ graph G2
    ∧ f '' vertices G1 ⊆ vertices G2
    ∧ univalent f
    ∧ edge_preserving f (edges G1) (edges G2)
  )"

```

```

lemma graph_homomorphismI:

```

```

assumes "vertices s = Domain h"
        "h ‘‘ vertices s  $\subseteq$  vertices t"
        "univalent h"
        "edge_preserving h (edges s) (edges t)"
        "s = restrict s" "t = restrict t"
shows "graph_homomorphism s t h" using assms unfolding graph_homomorphism_def
by auto

```

```

lemma graph_homomorphism_composes[intro]:
  assumes "graph_homomorphism a b x"
          "graph_homomorphism b c y"
  shows "graph_homomorphism a c (x  $\circ$  y)" proof(rule graph_homomorphismI,goal_cases)
  case 1
    have "vertices a  $\subseteq$  Domain x" "x ‘‘ vertices a  $\subseteq$  Domain y"
      using assms(1,2)[unfolded graph_homomorphism_def] by blast+
    from this Domain_0[OF this]
    show ?case using assms[unfolded graph_homomorphism_def] by auto
  next
  case 2 from assms show ?case unfolding graph_homomorphism_def by blast

```

```

qed (insert assms,auto simp:graph_homomorphism_def intro:compose_preserves_edge_preservin

```

```

lemma graph_homomorphism_empty[simp]:
  "graph_homomorphism (LG {} {}) G f  $\longleftrightarrow$  f = {}  $\wedge$  graph G"
unfolding graph_homomorphism_def by auto

```

```

lemma graph_homomorphism_Id[intro]:
  shows "graph_homomorphism (restrict a) (restrict a) (Id_on (vertices
a))"
  by (rule graph_homomorphismI;auto simp:edge_preserving_def)

```

```

lemma Id_on_vertices_identity:
  assumes "graph_homomorphism a b f"
          "(aa, ba)  $\in$  f"
  shows "(aa, ba)  $\in$  Id_on (vertices a)  $\circ$  f"
          "(aa, ba)  $\in$  f  $\circ$  Id_on (vertices b)"
  using assms unfolding graph_homomorphism_def by auto

```

Alternate version of definition 7.

```

abbreviation subgraph
  where "subgraph G1 G2
 $\equiv$  graph_homomorphism G1 G2 (Id_on (vertices G1))"

```

```

lemma subgraph_trans:
  assumes "subgraph G1 G2" "subgraph G2 G3"
  shows "subgraph G1 G3"
proof-
  from assms[unfolded graph_homomorphism_def]
  have "Id_on (vertices G1)  $\circ$  Id_on (vertices G2) = Id_on (vertices G1)"

```

```

    by auto
  with graph_homomorphism_composes[OF assms] show ?thesis by auto
qed

```

Just before Definition 7 in the paper, a notation is introduced for applying a function to a graph. We use `map_graph` for this, and the version `map_graph_fn` in case that its first argument is a total function rather than a partial one.

```

definition map_graph :: "('c × 'b) set ⇒ ('a, 'c) labeled_graph ⇒ ('a,
'b) labeled_graph" where
  "map_graph f G = LG (on_triple f `` (edges G)) (f `` (vertices G))"

```

```

lemma map_graph_selectors[simp]:
  "vertices (map_graph f G) = f `` (vertices G)"
  "edges (map_graph f G) = on_triple f `` (edges G)"
  unfolding map_graph_def by auto

```

```

lemma map_graph_comp[simp]:
  assumes "Range g ⊆ Domain f"
  shows "map_graph (g ∘ f) = map_graph f ∘ map_graph g"
proof(standard,goal_cases)
  case (1 x)
  from assms have "map_graph (g ∘ f) x = (map_graph f ∘ map_graph g)
x"
  unfolding map_graph_def by auto
  thus ?case by auto
qed

```

```

lemma map_graph_returns_restricted:
  assumes "vertices G = Domain f"
  shows "map_graph f G = restrict (map_graph f G)"
  using assms by(cases G,auto simp:map_graph_def restrict_def)

```

```

lemma map_graph_preserves_restricted[intro]:
  assumes "graph G"
  shows "graph (map_graph f G)"
proof(rule restrictI,standard) fix x
  assume "x ∈ edges (map_graph f G)"
  with assms show "x ∈ {(1, v1, v2). v1∈vertices (map_graph f G) ∧ v2∈vertices
(map_graph f G)}"
  by(cases x,auto simp:map_graph_def)
qed

```

```

lemma map_graph_edge_preserving[intro]:
  shows "edge_preserving f (edges G) (edges (map_graph f G))"
  unfolding map_graph_def by auto

```

```

lemma map_graph_homo[intro]:
  assumes "univalent f" "vertices G = Domain f" "G = restrict G"

```

```

  shows "graph_homomorphism G (map_graph f G) f"
proof(rule graph_homomorphismI)
  show "f `` vertices G  $\subseteq$  vertices (map_graph f G)"
    unfolding map_graph_def by auto
  show "edge_preserving f (edges G) (edges (map_graph f G))" by auto
  show "map_graph f G = restrict (map_graph f G)" using assms by auto
qed fact+

```

```

lemma map_graph_homo_simp:
  "graph_homomorphism G (map_graph f G) f
 $\longleftrightarrow$  univalent f  $\wedge$  vertices G = Domain f  $\wedge$  graph G"
proof
  show "graph_homomorphism G (map_graph f G) f  $\implies$ 
    univalent f  $\wedge$  vertices G = Domain f  $\wedge$  G = restrict G"
    unfolding graph_homomorphism_def by blast
qed auto

```

```

abbreviation on_graph where
  "on_graph G f  $\equiv$  BNF_Def.Gr (vertices G) f"

```

```

abbreviation map_graph_fn where
  "map_graph_fn G f  $\equiv$  map_graph (on_graph G f) G"

```

```

lemma map_graph_fn_graphI[intro]:
  "graph (map_graph_fn G f)" unfolding map_graph_def restrict_def by auto

```

```

lemma on_graph_id[simp]:
  shows "on_graph B id = Id_on (vertices B)"
  unfolding BNF_Def.Gr_def by auto

```

```

lemma in_on_graph[intro]:
  assumes "x  $\in$  vertices G" "(a x,y)  $\in$  b"
  shows "(x, y)  $\in$  on_graph G a 0 b"
  using assms unfolding BNF_Def.Gr_def by auto

```

```

lemma on_graph_comp:
  "on_graph G (f o g) = on_graph G g 0 on_graph (map_graph_fn G g) f"
  unfolding BNF_Def.Gr_def by auto

```

```

lemma map_graph_fn_eqI:
  assumes " $\bigwedge$  x. x  $\in$  vertices G  $\implies$  f x = g x"
  shows "map_graph_fn G f = map_graph_fn G g" (is "?l = ?r")
proof -
  { fix a ac ba
    assume "(a, ac, ba)  $\in$  edges G" "ac  $\in$  vertices G" "ba  $\in$  vertices
G"
  }
  hence " $\exists$ x $\in$ edges G. (x, a, g ac, g ba)  $\in$  on_triple (on_graph G f)"
    " $\exists$ x $\in$ edges G. (x, a, g ac, g ba)  $\in$  on_triple (on_graph G g)"
    using assms by (metis in_Gr on_triple)+

```

```

}
hence e:"edges ?l = edges ?r" using assms by (auto simp:Image_def)

have v:"vertices ?l = vertices ?r" using assms by (auto simp:image_def)
from e v show ?thesis by(cases ?l,cases ?r,auto)
qed

lemma map_graph_fn_comp[simp]:
"map_graph_fn G (f o g) = map_graph_fn (map_graph_fn G g) f"
  unfolding on_graph_comp by auto

lemma map_graph_fn_id[simp]:
"map_graph_fn X id = restrict X"
"map_graph (Id_on (vertices X)) X = restrict X"
  unfolding BNF_Def.Gr_def map_graph_def on_triple_def restrict_def by
(cases X,force)+

lemma graph_homo[intro!]:
  assumes "graph G"
  shows "graph_homomorphism G (map_graph_fn G f) (on_graph G f)"
  using assms unfolding map_graph_homo_simp BNF_Def.Gr_def univalent_def
by auto

lemma graph_homo_inv[intro!]:
  assumes "graph G" "inj_on f (vertices G)"
  shows "graph_homomorphism (map_graph_fn G f) G (converse (on_graph G
f))"
proof(rule graph_homomorphismI)
  show "univalent ((on_graph G f)-1)" using assms(2)
  unfolding univalent_def BNF_Def.Gr_def inj_on_def by auto
  show "edge_preserving ((on_graph G f)-1) (edges (map_graph_fn G f))
(edges G)"
  using assms unfolding edge_preserving inj_on_def by auto auto
qed (insert assms(1),auto)

lemma edge_preserving_on_graphI[intro]:
  assumes " $\bigwedge l x y. (l,x,y) \in \text{edges } X \implies x \in \text{vertices } X \implies y \in \text{vertices } X \implies (l, f x, f y) \in Y$ "
  shows "edge_preserving (on_graph X f) (edges X) Y"
  using assms unfolding edge_preserving_def BNF_Def.Gr_def by auto

lemma subgraph_subset:
  assumes "subgraph G1 G2"
  shows "vertices G1  $\subseteq$  vertices G2" "edges (restrict G1)  $\subseteq$  edges G2"
proof -
  have vrt:"Id_on (vertices G1) " " vertices G1  $\subseteq$  vertices G2"
  and ep:"edge_preserving (Id_on (vertices G1)) (edges G1) (edges G2)"
  using assms unfolding graph_homomorphism_def by auto
  hence "edges (restrict G1)  $\subseteq$  edges G2"

```

```

    using assms unfolding edge_preserving by auto
  thus "vertices  $G_1 \subseteq$  vertices  $G_2$ " "edges (restrict  $G_1$ )  $\subseteq$  edges  $G_2$ "
    using vrt by auto
qed

```

Our definition of subgraph is equivalent to definition 7.

```

lemma subgraph_def2:
  assumes "graph  $G_1$ " "graph  $G_2$ "
  shows "subgraph  $G_1 G_2 \iff$  vertices  $G_1 \subseteq$  vertices  $G_2 \wedge$  edges  $G_1 \subseteq$ 
edges  $G_2$ "
proof
  assume "vertices  $G_1 \subseteq$  vertices  $G_2 \wedge$  edges  $G_1 \subseteq$  edges  $G_2$ "
  hence v:"vertices  $G_1 \subseteq$  vertices  $G_2$ " and "edges  $G_1 \subseteq$  edges  $G_2$ " by
auto
  hence ep:"edge_preserving (Id_on (vertices  $G_1$ )) (edges  $G_1$ ) (edges  $G_2$ )"
    unfolding edge_preserving_def by auto
  show "subgraph  $G_1 G_2$ "
    using assms(2) v ep graph_homomorphism_Id[of " $G_1$ ",folded assms]
    unfolding graph_homomorphism_def by auto
next
  assume sg:"subgraph  $G_1 G_2$ "
  hence vrt:"Id_on (vertices  $G_1$ ) ‘‘ vertices  $G_1 \subseteq$  vertices  $G_2$ ”
    and ep:"edge_preserving (Id_on (vertices  $G_1$ )) (edges  $G_1$ ) (edges  $G_2$ )"
    unfolding graph_homomorphism_def by auto
  hence "edges  $G_1 \subseteq$  edges  $G_2$ "
    using assms unfolding edge_preserving by auto
  thus "vertices  $G_1 \subseteq$  vertices  $G_2 \wedge$  edges  $G_1 \subseteq$  edges  $G_2$ "
    using vrt by auto
qed

```

We also define *graph_union*. In contrast to the paper, our definition ignores the labels. The corresponding definition in the paper is written just above Definition 7. Adding labels to graphs would require a lot of unnecessary additional bookkeeping. Nowhere in the paper is the union actually used on different sets of labels, in which case these definitions coincide.

definition *graph_union* where
"*graph_union* $G_1 G_2 = LG$ (edges $G_1 \cup$ edges G_2) (vertices $G_1 \cup$ vertices G_2)"

```

lemma graph_unionI[intro]:
  assumes "edges  $G_1 \subseteq$  edges  $G_2$ "
    "vertices  $G_1 \subseteq$  vertices  $G_2$ "
  shows "graph_union  $G_1 G_2 = G_2$ "
  using assms unfolding graph_union_def by (cases " $G_2$ ",auto)

```

```

lemma graph_union_iff:
  shows "graph_union  $G_1 G_2 = G_2 \iff$  (edges  $G_1 \subseteq$  edges  $G_2 \wedge$  vertices
 $G_1 \subseteq$  vertices  $G_2$ )"
  unfolding graph_union_def by (cases " $G_2$ ",auto)

```

```

lemma graph_union_idemp[simp]:
  "graph_union A A = A"
  "graph_union A (graph_union A B) = (graph_union A B)"
  "graph_union A (graph_union B A) = (graph_union B A)"
  unfolding graph_union_def by auto

lemma graph_union_vertices[simp]:
  "vertices (graph_union G1 G2) = vertices G1 ∪ vertices G2"
  unfolding graph_union_def by auto
lemma graph_union_edges[simp]:
  "edges (graph_union G1 G2) = edges G1 ∪ edges G2"
  unfolding graph_union_def by auto

lemma graph_union_preserves_restrict[intro]:
  assumes "G1 = restrict G1" "G2 = restrict G2"
  shows "graph_union G1 G2 = restrict (graph_union G1 G2)"
proof -
  let ?e = "edges G1 ∪ edges G2"
  let ?v = "vertices G1 ∪ vertices G2"
  let ?r = "{(l, v1, v2). (l, v1, v2) ∈ ?e ∧ v1 ∈ ?v ∧ v2 ∈ ?v}"
  { fix l v1 v2
    assume a: "(l, v1, v2) ∈ ?e"
    have "(l, v1, v2) ∈ ?r" proof(cases "(l, v1, v2) ∈ edges (restrict G1)")
      case True
        hence "(l, v1, v2) ∈ edges G1" "v1 ∈ vertices G1" "v2 ∈ vertices
G1"
        by (auto simp: restrict_def)+
      thus ?thesis by auto
    next
      case False hence "(l, v1, v2) ∈ edges (restrict G2)" using a assms
    by auto
    hence "(l, v1, v2) ∈ edges G2" "v1 ∈ vertices G2" "v2 ∈ vertices
G2"
    by (auto simp: restrict_def)+
    then show ?thesis by auto
  }
  qed
  }
  hence "?e = ?r" by auto
  thus ?thesis unfolding graph_union_def by auto
qed

lemma graph_map_union[intro]:
  assumes "∧ i::nat. graph_union (map_graph (g i) X) Y = Y" "∧ i j.
i ≤ j ⇒ g i ⊆ g j"
  shows "graph_union (map_graph (∪ i. g i) X) Y = Y"
proof
  from assms have e: "edges (map_graph (g i) X) ⊆ edges Y"
    and v: "vertices (map_graph (g i) X) ⊆ vertices Y" for i

```

```

by (auto simp:graph_union_iff)
{ fix a ac ba aa b x xa
  assume a:"(a, ac, ba) ∈ edges X" "(ac, aa) ∈ g x" "(ba, b) ∈ g xa"
  have "(a, aa, b) ∈ edges Y"
  proof(cases "x < xa")
    case True
    hence "(a, ac, ba) ∈ edges X" "(ac, aa) ∈ g xa" "(ba, b) ∈ g xa"
      using a assms(2)[of x xa] by auto
    then show ?thesis using e[of xa] by auto
  next
    case False
    hence "(a, ac, ba) ∈ edges X" "(ac, aa) ∈ g x" "(ba, b) ∈ g x"
      using a assms(2)[of xa x] by auto
    then show ?thesis using e[of x] by auto
  qed
}
thus "edges (map_graph (⋃ i. g i) X) ⊆ edges Y" by auto
show "vertices (map_graph (⋃ i. g i) X) ⊆ vertices Y" using v by auto
qed

```

We show that *subgraph* indeed matches the definition in the paper (Definition 7).

```

lemma subgraph_def:
"subgraph G1 G2 = (G1 = restrict G1 ∧ G2 = restrict G2 ∧ graph_union G1 G2 = G2)"
proof
  assume assms:"subgraph G1 G2"
  hence r:"G2 = restrict G2" "G1 = restrict G1"
    unfolding graph_homomorphism_def by auto
  from subgraph_subset[OF assms]
  have ss:"vertices (restrict G1) ⊆ vertices G2" "edges (restrict G1)
⊆ edges G2" by auto
  show "G1 = restrict G1 ∧ G2 = restrict G2 ∧ graph_union G1 G2 = G2"
  proof(cases G2)
    case (LG x1 x2) show ?thesis using ss r
      unfolding graph_union_def LG by auto
    qed next
  assume gu: "G1 = restrict G1 ∧ G2 = restrict G2 ∧ graph_union G1 G2
= G2"
  hence sub:"(edges G1 ∪ edges G2) ⊆ edges G2"
    "vertices G1 ⊆ vertices G2"
    unfolding graph_union_def by (cases G2;auto)+
  have r:"G1 = restrict G1" "G2 = restrict G2" using gu by auto
  show "subgraph G1 G2" unfolding subgraph_def2[OF r] using sub by auto
qed

```

```

lemma subgraph_refl[simp]:
"subgraph G G = (G = restrict G)"
  unfolding subgraph_def graph_union_def by(cases G,auto)

```

```

lemma subgraph_restrict[simp]:
  "subgraph G (restrict G) = graph G"
  using subgraph_refl subgraph_def by auto

  Definition 10. We write graph_homomorphism instead of embedding.

lemma graph_homomorphism_def2:
  shows "graph_homomorphism G1 G2 f =
    (vertices G1 = Domain f ∧ univalent f ∧ G1 = restrict G1 ∧ G2 = restrict
    G2 ∧ graph_union (map_graph f G1) G2 = G2)"
    (is "?lhs = ?rhs")
proof
  let ?m = "map_graph f G1"
  assume ?rhs
  hence assms : "vertices G1 = Domain f" "univalent f" "G1 = restrict
  G1"
    and sg: "subgraph ?m G2"
    and f_id: "f 0 Id_on (f `` vertices G1) = f" unfolding subgraph_def
  by auto
  hence "edge_preserving (Id_on (vertices ?m)) (edges ?m) (edges G2)"
    unfolding graph_homomorphism_def by auto
  hence "on_triple (f 0 Id_on (f `` vertices G1)) `` edges G1 ⊆ edges
  G2"
    unfolding relcomp_Image edge_preserving map_graph_selectors relcomp_on_triple.
  hence "edge_preserving f (edges G1) (edges G2)"
    unfolding edge_preserving f_id.
  thus ?lhs
    using sg assms unfolding graph_homomorphism_def
    by auto next
  assume ih: ?lhs
  hence "vertices G1 = Domain f ∧ univalent f ∧ G1 = restrict G1 ∧ subgraph
  (map_graph f G1) G2"
    unfolding graph_homomorphism_def edge_preserving
    by auto
  thus ?rhs unfolding subgraph_def by auto
qed

```

```

lemma map_graph_preserves_subgraph[intro]:
  assumes "subgraph A B"
  shows "subgraph (map_graph f A) (map_graph f B)"
  using assms unfolding subgraph_def by (auto simp: graph_union_iff)

```

```

lemma graph_homomorphism_concr_graph:
  assumes "graph G" "graph (LG e v)"
  shows "graph_homomorphism (LG e v) G x ↔
    x `` v ⊆ vertices G ∧ on_triple x `` e ⊆ edges G ∧ univalent
  x ∧ Domain x = v"
  using assms unfolding graph_homomorphism_def2 graph_union_iff by auto

```

```

lemma subgraph_preserves_hom:
  assumes "subgraph A B"
    "graph_homomorphism X A h"
  shows "graph_homomorphism X B h"
  using assms by (meson graph_homomorphism_def2 map_graph_preserves_restricted
subgraph_def subgraph_trans)

lemma graph_homo_union_id:
  assumes "graph_homomorphism (graph_union A B) G f"
  shows "graph A  $\implies$  graph_homomorphism A G (Id_on (vertices A) 0 f)"
    "graph B  $\implies$  graph_homomorphism B G (Id_on (vertices B) 0 f)"
  using assms unfolding graph_homomorphism_def edge_preserving
  by (auto dest:edge_preserving_atomic)

lemma graph_homo_union[intro]:
  assumes
    "graph_homomorphism A G f_a"
    "graph_homomorphism B G f_b"
    "Domain f_a  $\cap$  Domain f_b = Domain (f_a  $\cap$  f_b)"
  shows "graph_homomorphism (graph_union A B) G (f_a  $\cup$  f_b)"
  proof(rule graph_homomorphismI)
    have v0:"f_a '' vertices A  $\subseteq$  vertices G" "f_b '' vertices B  $\subseteq$  vertices
G"
      "vertices A = Domain f_a" "vertices B = Domain f_b"
      "graph A" "graph B"
      "univalent f_a" "univalent f_b"
      "edge_preserving f_a (edges A) (edges G)"
      "edge_preserving f_b (edges B) (edges G)"
    using assms(1,2) unfolding graph_homomorphism_def by blast+
    hence v: "f_a '' vertices (graph_union A B)  $\subseteq$  vertices G"
      "f_b '' vertices (graph_union A B)  $\subseteq$  vertices G" by auto
    show uni:"univalent (f_a  $\cup$  f_b)" using assms(3) v0 by auto
    show "(f_a  $\cup$  f_b) '' vertices (graph_union A B)  $\subseteq$  vertices G" us-
ing v by auto
    have f_a:"Id_on (vertices A) 0 (f_a  $\cup$  f_b) = f_a"
      using uni v0(3)
      by (cases A,auto simp:univalent_def on_triple_def Image_def)
    have onA:"on_triple (f_a  $\cup$  f_b) '' edges A = on_triple (Id_on (vertices
A) 0 (f_a  $\cup$  f_b)) '' edges A"
      unfolding relcomp_on_triple relcomp_Image on_triple_ID_restrict v0(5)[symmetric]
    ..
    have f_b:"Id_on (vertices B) 0 (f_a  $\cup$  f_b) = f_b"
      using uni v0(4) unfolding Un_commute[of f_a _]
      by (cases B,auto simp:univalent_def on_triple_def Image_def)
    have onB:"on_triple (f_a  $\cup$  f_b) '' edges B = on_triple (Id_on (vertices
B) 0 (f_a  $\cup$  f_b)) '' edges B"
      unfolding relcomp_on_triple relcomp_Image on_triple_ID_restrict v0(6)[symmetric]
    ..

```

```

    have "edge_preserving (f_a ∪ f_b) (edges A) (edges G)"
      "edge_preserving (f_a ∪ f_b) (edges B) (edges G)"
      using v0(9,10) unfolding edge_preserving onA[unfolded f_a] onB[unfolded
f_b] by auto
    thus "edge_preserving (f_a ∪ f_b) (edges (graph_union A B)) (edges
G)"
      by auto
qed (insert assms[unfolded graph_homomorphism_def],auto)

lemma graph_homomorphism_on_graph:
  assumes "graph_homomorphism A B R"
  shows "graph_homomorphism A (map_graph_fn B f) (R O on_graph B f)"
proof -
  from assms have "Range R ⊆ vertices B"
    and ep: "edge_preserving R (edges A) (edges B)" unfolding graph_homomorphism_def
  by auto
  hence d:"Domain R ⊆ Domain (R O on_graph B f)" unfolding Domain_id_on
  by auto
  have v:"vertices (map_graph (R O on_graph B f) A) ⊆ vertices (map_graph_fn
B f)"
    unfolding BNF_Def.Gr_def map_graph_selectors by auto
  have e:"edges (map_graph (R O on_graph B f) A) ⊆ edges (map_graph_fn
B f)" using ep
    unfolding BNF_Def.Gr_def map_graph_selectors edge_preserving by auto
  have u:"graph_union (map_graph (R O on_graph B f) A) (map_graph_fn B
f) = map_graph_fn B f"
    using e v graph_unionI by metis
  from d assms u show "graph_homomorphism A (map_graph_fn B f) (R O on_graph
B f)"
    unfolding graph_homomorphism_def2 by auto
qed

end

```

3 Rules, and the chains we can make with them

This describes graph rules, and the reasoning is fully on graphs here (no semantics). The formalisation builds up to Lemma 4 in the paper.

```

theory RulesAndChains
imports LabeledGraphs
begin

```

```

type_synonym ('l, 'v) graph_seq = "(nat ⇒ ('l, 'v) labeled_graph)"

```

Definition 8.

```

definition chain :: "('l, 'v) graph_seq ⇒ bool" where
  "chain S ≡ ∀ i. subgraph (S i) (S (i + 1))"

```

```

lemma chain_then_restrict:
  assumes "chain S" shows "S i = restrict (S i)"
  using assms[unfolded chain_def graph_homomorphism_def] by auto

lemma chain:
  assumes "chain S"
  shows "j ≥ i ⇒ subgraph (S i) (S j)"
proof(induct "j-i" arbitrary:i j)
  case 0
  then show ?case using chain_then_restrict[OF assms] assms[unfolded
chain_def] by auto
next
  case (Suc x)
  hence j:"i + x + 1 = j" by auto
  thus ?case
    using subgraph_trans[OF Suc(1) assms[unfolded chain_def,rule_format,of
"i+x"],of i,unfolded j]
    using Suc by auto
qed

```

```

lemma chain_def2:
  "chain S = (∀ i j. j ≥ i → subgraph (S i) (S j))"
proof
  show "chain S ⇒ ∀ i j. i ≤ j → subgraph (S i) (S j)" using chain
  by auto
  show "∀ i j. i ≤ j → subgraph (S i) (S j) ⇒ chain S" unfolding
chain_def by simp
qed

```

Second part of definition 8.

```

definition chain_sup :: "('l, 'v) graph_seq ⇒ ('l, 'v) labeled_graph"
where
  "chain_sup S ≡ LG (⋃ i. edges (S i)) (⋃ i. vertices (S i))"

```

```

lemma chain_sup_const[simp]:
  "chain_sup (λ x. S) = S"
  unfolding chain_sup_def by auto

```

```

lemma chain_sup_subgraph[intro]:
  assumes "chain S"
  shows "subgraph (S j) (chain_sup S)"
proof -
  have c1: "S j = restrict (S j)" for j
    using assms[unfolded chain_def,rule_format,of j] graph_homomorphism_def
  by auto
  hence c2: "chain_sup S = restrict (chain_sup S)"
    unfolding chain_sup_def by fastforce
  have c3: "graph_union (S j) (chain_sup S) = chain_sup S"
    unfolding chain_sup_def graph_union_def by auto

```

show ?thesis unfolding subgraph_def using c1 c2 c3 by auto
qed

lemma chain_sup_graph[intro]:
 assumes "chain S"
 shows "graph (chain_sup S)"
 using chain_sup_subgraph[OF assms]
 unfolding subgraph_def by auto

lemma map_graph_chain_sup:
 "map_graph g (chain_sup S) = chain_sup (map_graph g o S)"
 unfolding map_graph_def chain_sup_def by auto

lemma graph_union_chain_sup[intro]:
 assumes " $\bigwedge i. \text{graph_union } (S\ i) C = C$ "
 shows " $\text{graph_union } (\text{chain_sup } S) C = C$ "
proof
 from assms have e:"edges (S i) \subseteq edges C" and v:"vertices (S i) \subseteq
 vertices C" for i
 by (auto simp:graph_union_iff)
 show "edges (chain_sup S) \subseteq edges C" using e unfolding chain_sup_def
by auto
 show "vertices (chain_sup S) \subseteq vertices C" using v unfolding chain_sup_def
by auto
qed

type_synonym ('l,'v) Graph_PreRule = "('l, 'v) labeled_graph \times ('l,
'v) labeled_graph"

Definition 9.

abbreviation graph_rule :: "('l,'v) Graph_PreRule \Rightarrow bool" where
 "graph_rule R \equiv subgraph (fst R) (snd R) \wedge finite_graph (snd R)"

definition set_of_graph_rules :: "('l,'v) Graph_PreRule set \Rightarrow bool" where
 "set_of_graph_rules Rs \equiv $\forall R \in Rs. \text{graph_rule } R$ "

lemma set_of_graph_rulesD[dest]:
 assumes "set_of_graph_rules Rs" "R \in Rs"
 shows "finite_graph (fst R)" "finite_graph (snd R)" "subgraph (fst R)
(snd R)"
 using assms(1)[unfolded set_of_graph_rules_def] assms(2)
 rev_finite_subset[of "vertices (snd R)"]
 rev_finite_subset[of "edges (snd R)"]
 unfolding subgraph_def graph_union_iff by auto

We define agree_on as an equivalence.

definition agree_on where
 "agree_on G f₁ f₂ \equiv ($\forall v \in \text{vertices } G. f_1 \text{ `` } \{v\} = f_2 \text{ `` } \{v\}$)"

```

lemma agree_on_empty[intro,simp]: "agree_on (LG {} {}) f g" unfolding
agree_on_def by auto

lemma agree_on_comm[intro]: "agree_on X f g = agree_on X g f" unfold-
ing agree_on_def by auto
lemma agree_on_refl[intro]:
  "agree_on R f f" unfolding agree_on_def by auto
lemma agree_on_trans:
  assumes "agree_on X f g" "agree_on X g h"
  shows "agree_on X f h" using assms unfolding agree_on_def by auto

lemma agree_on_equivp:
  shows "equivp (agree_on G)"
  by (auto intro:agree_on_trans intro!:equivpI simp:reflp_def symp_def
transp_def agree_on_comm)

lemma agree_on_subset:
  assumes "f  $\subseteq$  g" "vertices G  $\subseteq$  Domain f" "univalent g"
  shows "agree_on G f g"
  using assms unfolding agree_on_def by auto

lemma agree_iff_subset[simp]:
  assumes "graph_homomorphism G X f" "univalent g"
  shows "agree_on G f g  $\longleftrightarrow$  f  $\subseteq$  g"
  using assms unfolding agree_on_def graph_homomorphism_def by auto

lemma agree_on_ext:
  assumes "agree_on G f1 f2"
  shows "agree_on G (f1  $\cup$  g) (f2  $\cup$  g)"
  using assms unfolding agree_on_def by auto

lemma agree_on_then_eq:
  assumes "agree_on G f1 f2" "Domain f1 = vertices G" "Domain f2 = vertices
G"
  shows "f1 = f2"
proof -
  from assms have agr:" $\bigwedge v. v \in \text{Domain } f_1 \implies f_1 \text{ `` } \{v\} = f_2 \text{ `` } \{v\}$ "
unfolding agree_on_def by auto
  have agr2:" $\bigwedge v. v \notin \text{Domain } f_1 \implies f_1 \text{ `` } \{v\} = \{\}$ "
    " $\bigwedge v. v \notin \text{Domain } f_2 \implies f_2 \text{ `` } \{v\} = \{\}$ " by auto
  with agr agr2 assms have " $\bigwedge v. f_1 \text{ `` } \{v\} = f_2 \text{ `` } \{v\}$ " by blast
  thus ?thesis by auto
qed

lemma agree_on_subg_compose:
  assumes "agree_on R g h" "agree_on F f g" "subgraph F R"
  shows "agree_on F f h"
  using assms unfolding agree_on_def subgraph_def graph_union_iff by

```

auto

definition extensible :: "('l,'x) Graph_PreRule \Rightarrow ('l,'v) labeled_graph
 \Rightarrow ('x \times 'v) set \Rightarrow bool"
 where
 "extensible R G f \equiv (\exists g. graph_homomorphism (snd R) G g \wedge agree_on (fst R) f g)"

lemma extensibleI[*intro*]:
 assumes "graph_homomorphism R2 G g" "agree_on R1 f g"
 shows "extensible (R1,R2) G f"
 using *assms* *unfolding* *extensible_def* *by auto*

lemma extensibleD[*elim*]:
 assumes "extensible R G f"
 " \bigwedge g. graph_homomorphism (snd R) G g \implies agree_on (fst R) f g \implies *thesis*"
 shows *thesis* **using** *assms* *extensible_def* *by blast*

lemma extensible_refl_concr[*simp*]:
 assumes "graph_homomorphism (LG e₁ v) G f"
 shows "extensible (LG e₁ v, LG e₂ v) G f \longleftrightarrow graph_homomorphism (LG e₂ v) G f"
proof
 assume "extensible (LG e₁ v, LG e₂ v) G f"
 then obtain g **where** g: "graph_homomorphism (LG e₂ v) G g" "agree_on (LG e₁ v) f g"
 unfolding *extensible_def* *by auto*
 hence d: "Domain f = Domain g" "univalent f" "univalent g" **using** *assms*
 unfolding *graph_homomorphism_def* *by auto*
 from g **have** *subs*: "f \subseteq g"
 by (*subst* *agree_iff_subset* [*symmetric*, *OF assms*], *auto simp: graph_homomorphism_def*)
 with d **have** "f = g" *by auto*
 thus "graph_homomorphism (LG e₂ v) G f" **using** g *by auto*
qed (*auto simp: assms extensible_def*)

lemma extensible_chain_sup[*intro*]:
 assumes "chain S" "extensible R (S j) f"
 shows "extensible R (chain_sup S) f"
proof -
 from *assms* **obtain** g **where** g: "graph_homomorphism (snd R) (S j) g \wedge agree_on (fst R) f g"
 unfolding *extensible_def* *by auto*
 have [*simp*]: "g \circ Id_on (vertices (S j)) = g" **using** g [*unfolded graph_homomorphism_def*]
by auto
 from g *assms*(1)
 have "graph_homomorphism (snd R) (S j) g" "subgraph (S j) (chain_sup S)" **by auto**
 from *graph_homomorphism_composes* [*OF this*]

have "graph_homomorphism (snd R) (chain_sup S) g" by auto
 thus ?thesis using g unfolding extensible_def by blast
 qed

Definition 11.

definition maintained :: "('l,'x) Graph_PreRule \Rightarrow ('l,'v) labeled_graph
 \Rightarrow bool"
 where "maintained R G $\equiv \forall f. \text{graph_homomorphism } (fst R) G f \longrightarrow \text{extensible } R G f"$

abbreviation maintainedA
 :: "('l,'x) Graph_PreRule set \Rightarrow ('l, 'v) labeled_graph \Rightarrow bool"
 where "maintainedA Rs G $\equiv \forall R \in Rs. \text{maintained } R G"$

lemma maintainedI[*intro*]:
 assumes " $\bigwedge f. \text{graph_homomorphism } A G f \Longrightarrow \text{extensible } (A,B) G f"$
 shows "maintained (A,B) G"
 using assms unfolding maintained_def by auto

lemma maintainedD[*dest*]:
 assumes "maintained (A,B) G" "graph_homomorphism A G f"
 shows "extensible (A,B) G f"
 using assms unfolding maintained_def by auto

lemma maintainedD2[*dest*]:
 assumes "maintained (A,B) G" "graph_homomorphism A G f"
 " $\bigwedge g. \text{graph_homomorphism } B G g \Longrightarrow f \subseteq g \Longrightarrow \text{thesis}"$
 shows thesis
 using maintainedD[OF assms(1,2),unfolded extensible_def]

proof
 fix g
 assume "graph_homomorphism (snd (A, B)) G g \wedge agree_on (fst (A, B))
 f g"
 hence "graph_homomorphism B G g" "f \subseteq g"
 using assms(2) unfolding graph_homomorphism_def2 agree_on_def by auto
 from assms(3)[OF this] show thesis.
 qed

lemma extensible_refl[*intro*]:
 "graph_homomorphism R G f \Longrightarrow extensible (R,R) G f"
 unfolding extensible_def by auto

lemma maintained_refl[*intro*]:
 "maintained (R,R) G" by auto

Alternate version of definition 8.

definition fin_maintained :: "('l,'x) Graph_PreRule \Rightarrow ('l,'v) labeled_graph
 \Rightarrow bool"
 where
 "fin_maintained R G $\equiv \forall F f. \text{finite_graph } F$

```

    → subgraph F (fst R)
    → extensible (F,fst R) G f
    → graph_homomorphism F G f
    → extensible (F,snd R) G f"

```

```

lemma fin_maintainedI [intro]:
  assumes "∧ F f. finite_graph F
    ⇒ subgraph F (fst R)
    ⇒ extensible (F,fst R) G f
    ⇒ graph_homomorphism F G f
    ⇒ extensible (F,snd R) G f"
  shows "fin_maintained R G" using assms unfolding fin_maintained_def
  by auto

```

```

lemma maintained_then_fin_maintained[simp]:
  assumes maintained:"maintained R G"
  shows "fin_maintained R G"
proof
  fix F f
  assume subg:"subgraph F (fst R)"
    and ext:"extensible (F, fst R) G f" and igh:"graph_homomorphism
F G f"
  from ext[unfolded extensible_def prod.sel] obtain g where
    g:"graph_homomorphism (fst R) G g" "agree_on F f g" by blast
  from maintained[unfolded maintained_def,rule_format,OF g(1)] g(2) subg
    agree_on_subg_compose
  show "extensible (F, snd R) G f" unfolding extensible_def prod.sel
  by blast
qed

```

```

lemma fin_maintained_maintained:
  assumes "finite_graph (fst R)"
  shows "fin_maintained R G ↔ maintained R G" (is "?lhs = ?rhs")
proof
  from assms rev_finite_subset
  have fin:"finite (vertices (fst R))"
    "finite (edges (fst R))"
    "subgraph (fst R) (fst R)"
  unfolding subgraph_def graph_union_iff by auto
  assume ?lhs
  with fin have "extensible (fst R, fst R) G f ⇒ graph_homomorphism
(fst R) G f
    ⇒ extensible R G f" for f unfolding fin_maintained_def by
  auto
  thus ?rhs by (simp add: extensible_refl maintained_def)
qed simp

```

```

lemma extend_for_chain:
  assumes "g 0 = f"

```

```

    and " $\bigwedge i. \text{graph\_homomorphism } (S\ i)\ C\ (g\ i)$ "
    and " $\bigwedge i. \text{agree\_on } (S\ i)\ (g\ i)\ (g\ (i + 1))$ "
    and "chain S"
  shows "extensible (S 0, chain_sup S) C f"
proof
  let ?g = " $\bigcup i. g\ i$ "
  from assms(4)[unfolded chain_def subgraph_def graph_union_iff]
  have v:"vertices (S i)  $\subseteq$  vertices (S (i + 1))"
    and e:"edges (S i)  $\subseteq$  edges (S (i + 1))" for i by auto
  { fix a b i
    assume a:"(a, b)  $\in$  g i"
    hence "a  $\in$  vertices (S i)" using assms(2)[of i]
      unfolding graph_homomorphism_def2 by auto
    from assms(3)[unfolded agree_on_def,rule_format,OF this] a
    have "(a, b)  $\in$  g (Suc i)" by auto
  }
  hence gi:"g i  $\subseteq$  g (Suc i)" for i by auto
  have gij:"i  $\leq$  j  $\implies$  g i  $\subseteq$  g j" for i j proof(induct j)
    case (Suc j) with gi[of j] show ?case by (cases "i = Suc j",auto)
  qed auto
  from assms(1) have f_subset:"f  $\subseteq$  ?g" by auto
  from assms(2)[of 0,unfolded assms(1)] have domf:"Domain f = vertices (S 0)"
    and grC:"graph C" and v_dom:"vertices (S i) = Domain (g i)" for i
  using assms(2)
    unfolding graph_homomorphism_def by auto
  { fix x y z i j assume "(x, y)  $\in$  g i" "(x, z)  $\in$  g j"
    with gij[of i "max i j"] gij[of j "max i j"]
    have "(x,y)  $\in$  g (max i j)" "(x,z)  $\in$  g (max i j)" by auto
    with assms(2)[unfolded graph_homomorphism_def]
    have "y = z" by auto
  } note univ_strong = this
  hence univ:"univalent ?g" unfolding univalent_def by auto
  { fix xa x i
    assume "(xa, x)  $\in$  g i"
    hence "x  $\in$  vertices (map_graph (g i) (S i))"
      using assms(2) unfolding graph_homomorphism_def by auto
    hence "x  $\in$  vertices C"
      using assms(2) unfolding graph_homomorphism_def2 graph_union_iff
  }
  by blast
  } note eq_v = this
  { fix l x y x' y' j i
    assume "(l,x,y)  $\in$  edges (S j)" "(x, x')  $\in$  g i" "(y, y')  $\in$  g i"
    with gij[of i "max i j"] gij[of j "max i j"]
      chain[OF assms(4),unfolded subgraph_def graph_union_iff, of i "max i j"]
      chain[OF assms(4),unfolded subgraph_def graph_union_iff, of j "max i j"]
    have "(x,x')  $\in$  g (max i j)" "(y,y')  $\in$  g (max i j)"
  }

```

```

      "(l,x,y) ∈ edges (S (max i j))" by auto
    hence "(l, x', y') ∈ edges C"
      using assms(2)[unfolded graph_homomorphism_def2 graph_union_iff]
  by auto
} note eq_e = this
have "graph_union (map_graph (g i) (chain_sup S)) C = C" for i
  unfolding graph_union_iff using eq_e eq_v
  unfolding graph_homomorphism_def2 chain_sup_def by auto
hence subg:"graph_union (map_graph ?g (chain_sup S)) C = C"
  apply (rule graph_map_union) using gij by auto
have "(⋃ i. vertices (S i)) = (⋃ i. Domain (g i))" using v_dom by auto
hence vd:"vertices (chain_sup S) = Domain ?g"
  unfolding chain_sup_def by auto
show "graph_homomorphism (chain_sup S) C ?g"
  unfolding graph_homomorphism_def2
  using univ chain_sup_graph[OF assms(4)] grC vd subg by auto
show "agree_on (S 0) f ?g" using agree_on_subset[OF f_subset _ univ]
  domf by auto
qed

```

Definition 8, second part.

definition *consequence_graph*

where "*consequence_graph* Rs G \equiv graph G \wedge (\forall R \in Rs. subgraph (fst R) (snd R) \wedge maintained R G)"

lemma *consequence_graphI*[intro]:

```

assumes " $\bigwedge$  R. R  $\in$  Rs  $\implies$  maintained R G"
        " $\bigwedge$  R. R  $\in$  Rs  $\implies$  subgraph (fst R) (snd R)"
        "graph G"
shows "consequence_graph Rs G"
  unfolding consequence_graph_def fin_maintained_def using assms by auto

```

lemma *consequence_graphD*[dest]:

```

assumes "consequence_graph Rs G"
shows " $\bigwedge$  R. R  $\in$  Rs  $\implies$  maintained R G"
      " $\bigwedge$  R. R  $\in$  Rs  $\implies$  subgraph (fst R) (snd R)"
      "graph G"
  using assms unfolding consequence_graph_def fin_maintained_def by auto

```

Definition 8 states: If furthermore S is a subgraph of G, and (S, G) is maintained in each consequence graph maintaining Rs, then G is a least consequence graph of S maintaining Rs. Note that the type of 'each consequence graph' isn't given here. Taken literally, this should mean 'for every possible type'. We avoid quantifying on types by making the type an argument. Consequently, when proving 'least', the first argument should be free.

definition *least*

```

:: "'x itself  $\implies$  (('l, 'v) Graph_PreRule) set  $\implies$  ('l, 'c) labeled_graph
 $\implies$  ('l, 'c) labeled_graph  $\implies$  bool"

```

where "least _ Rs S G \equiv subgraph S G \wedge
 $(\forall C :: ('l, 'x) \text{ labeled_graph. consequence_graph Rs C} \longrightarrow$
 maintained (S,G) C)"

lemma leastI[intro]:
 assumes "subgraph S (G:: ('l, 'c) labeled_graph)"
 " $\wedge C :: ('l, 'x) \text{ labeled_graph. consequence_graph Rs C} \implies$ maintained
 (S,G) C"
 shows "least (t:: 'x itself) Rs S G"
 using assms unfolding least_def by auto

definition least_consequence_graph
 :: "'x itself \Rightarrow (('l, 'v) Graph_PreRule) set
 $\Rightarrow ('l, 'c) \text{ labeled_graph} \Rightarrow ('l, 'c) \text{ labeled_graph} \Rightarrow \text{bool}"$
 where "least_consequence_graph t Rs S G \equiv consequence_graph Rs G \wedge
 least t Rs S G"

lemma least_consequence_graphI[intro]:
 assumes "consequence_graph Rs (G:: ('l, 'c) labeled_graph)"
 "subgraph S G"
 " $\wedge C :: ('l, 'x) \text{ labeled_graph. consequence_graph Rs C} \implies$ maintained
 (S,G) C"
 shows "least_consequence_graph (t:: 'x itself) Rs S G"
 using assms unfolding least_consequence_graph_def least_def by auto

Definition 12.

definition fair_chain where
 "fair_chain Rs S \equiv chain S \wedge
 $(\forall R f i. (R \in Rs \wedge \text{graph_homomorphism (fst R) (S i) f}) \longrightarrow (\exists j.$
 extensible R (S j) f))"

lemma fair_chainI[intro]:
 assumes "chain S"
 " $\wedge R f i. R \in Rs \implies \text{graph_homomorphism (fst R) (S i) f} \implies \exists j.$
 extensible R (S j) f"
 shows "fair_chain Rs S"
 using assms unfolding fair_chain_def by blast

lemma fair_chainD:
 assumes "fair_chain Rs S"
 shows "chain S"
 " $R \in Rs \implies \text{graph_homomorphism (fst R) (S i) f} \implies \exists j. \text{extensible}$
 R (S j) f"
 using assms unfolding fair_chain_def by blast+

lemma find_graph_occurence_vertices:
 assumes "chain S" "finite V" "univalent f" "f '' V \subseteq vertices (chain_sup
 S)"
 shows " $\exists i. f '' V \subseteq \text{vertices (S i)}$ "

```

    using assms(2,4)
  proof(induct V)
    case empty thus ?case by auto
  next
    case (insert v V)
    from insert.prem1 have V:"f '' V ⊆ vertices (chain_sup S)"
      and v:"f '' {v} ⊆ vertices (chain_sup S)" by auto
    from insert.hyps(3)[OF V] obtain i where i:"f '' V ⊆ vertices (S i)"
  by auto
  have "∃ j. f '' {v} ⊆ vertices (S j)"
  proof(cases "(f '' {v}) = {}")
    case False
    then obtain v' where f:"(v,v') ∈ f" by auto
    hence "v' ∈ vertices (chain_sup S)" using v by auto
    then show ?thesis using assms(3) f unfolding chain_sup_def by auto
  qed auto
  then obtain j where j:"f '' {v} ⊆ vertices (S j)" by blast
  have sg:"subgraph (S i) (S (max i j))" "subgraph (S j) (S (max i j))"
    by(rule chain[OF assms(1)],force)+
  have V:"(f ∩ V × UNIV) '' V ⊆ vertices (S (max i j))"
    using i subgraph_subset[OF sg(1)] by auto
  have v:"f '' {v} ⊆ vertices (S (max i j))" using j subgraph_subset[OF
sg(2)] by auto
  have "f '' insert v V ⊆ vertices (S (max i j))" using v V by auto
  thus ?case by blast
qed

lemma find_graph_occurrence_edges:
  assumes "chain S" "finite E" "univalent f"
    "on_triple f '' E ⊆ edges (chain_sup S)"
  shows "∃ i. on_triple f '' E ⊆ edges (S i)"
  using assms(2,4)
  proof(induct E)
    case empty thus ?case unfolding graph_homomorphism_def by auto
  next
    case (insert e E)
    have univ:"univalent (on_triple f)" using assms(3) by auto
    have [simp]:"restrict (S i) = S i" for i
      using chain[OF assms(1),unfolded subgraph_def,of i i] by auto
    from insert.prem1 have E:"on_triple f '' E ⊆ edges (chain_sup S)"
      and e:"on_triple f '' {e} ⊆ edges (chain_sup S)" by auto
    with insert.hyps obtain i where i:"on_triple f '' E ⊆ edges (S i)"
  by auto
  have "∃ j. on_triple f '' {e} ⊆ edges (S j)"
  proof(cases "on_triple f '' {e} = {}")
    case False
    then obtain e' where f:"(e,e') ∈ on_triple f" by auto
    hence "e' ∈ edges (chain_sup S)" using e by auto
    then show ?thesis using univ f unfolding chain_sup_def by auto
  qed

```

```

qed auto
then obtain j where j:"on_triple f ‘‘ {e} ⊆ edges (S j)" by blast
have sg:"subgraph (S i) (S (max i j))" "subgraph (S j) (S (max i j))"
  by(rule chain[OF assms(1)],force)+
have E:"on_triple f ‘‘ E ⊆ edges (S (max i j))"
  using i subgraph_subset[OF sg(1)] by auto
have e:"on_triple f ‘‘ {e} ⊆ edges (S (max i j))" using j subgraph_subset[OF
sg(2)] by auto
have "on_triple f ‘‘ insert e E ⊆ edges (S (max i j))" using e E by
auto
thus ?case by blast
qed

lemma find_graph_occurrence:
  assumes "chain S" "finite E" "finite V" "graph_homomorphism (LG E V)
(chain_sup S) f"
  shows "∃ i. graph_homomorphism (LG E V) (S i) f"
proof -
  have [simp]:"restrict (S i) = S i" for i
    using chain[OF assms(1),unfolded subgraph_def,of i i] by auto
  from assms[unfolded graph_homomorphism_def edge_preserving labeled_graph.sel]
  have u:"univalent f"
    and e:"on_triple f ‘‘ E ⊆ edges (chain_sup S)"
    and v:"f ‘‘ V ⊆ vertices (chain_sup S)"
    by blast+
  from find_graph_occurrence_edges[OF assms(1,2) u e]
  obtain i where i:"on_triple f ‘‘ E ⊆ edges (S i)" by blast
  from find_graph_occurrence_vertices[OF assms(1,3) u v]
  obtain j where j:"f ‘‘ V ⊆ vertices (S j)" by blast
  have sg:"subgraph (S i) (S (max i j))" "subgraph (S j) (S (max i j))"
    by(rule chain[OF assms(1)],force)+
  have e:"on_triple f ‘‘ E ⊆ edges (S (max i j))"
    and v:"f ‘‘ V ⊆ vertices (S (max i j))"
    using i j subgraph_subset(2)[OF sg(1)] subgraph_subset(1)[OF sg(2)]
  by auto
  have "graph_homomorphism (LG E V) (S (max i j)) f"
  proof(rule graph_homomorphismI)
    from assms[unfolded graph_homomorphism_def edge_preserving labeled_graph.sel]
  e v
  show "vertices (LG E V) = Domain f"
    and "univalent f"
    and "LG E V = restrict (LG E V)"
    and "f ‘‘ vertices (LG E V) ⊆ vertices (S (max i j))"
    and "edge_preserving f (edges (LG E V)) (edges (S (max i j)))"
    and "S (max i j) = restrict (S (max i j))" by auto
  qed
  thus ?thesis by auto
qed

```

Lemma 3. Recall that in the paper, graph rules use finite graphs, i.e.

both sides should be finite. We strengthen lemma 3 by requiring only the left hand side to be a finite graph.

```

lemma fair_chain_impl_consequence_graph:
  assumes "fair_chain Rs S" "\ R. R \in Rs \implies subgraph (fst R) (snd R)
  \ finite_graph (fst R)"
  shows "consequence_graph Rs (chain_sup S)"
proof -
  { fix R assume a:"R \in Rs"
    have fin_v:"finite (vertices (fst R))" and fin_e: "finite (edges
  (fst R))"
      using assms(2)[OF a] by auto
    { fix f assume "graph_homomorphism (LG (edges (fst R)) (vertices (fst
  R))) (chain_sup S) f"
      with find_graph_occurence[OF fair_chainD(1)[OF assms(1)] fin_e fin_v]

      obtain i where "graph_homomorphism (fst R) (S i) f" by auto
      from fair_chainD(2)[OF assms(1) a this] obtain j
        where "extensible R (S j) f" by blast
      hence "extensible R (chain_sup S) f" using fair_chainD(1)[OF assms(1)]
  by auto
    }
    hence "maintained R (chain_sup S)" unfolding maintained_def by auto
  } note mnt = this
  from assms have "chain S" unfolding fair_chain_def by auto
  thus ?thesis unfolding consequence_graph_def using mnt assms(2) by
  blast
qed

```

We extract the weak universal property from the definition of weak pushout step. Again, the paper allows for arbitrary types in the quantifier, but we fix the type here in the definition that will be used in *pushout_step*. The type used here should suffice (and we cannot quantify over types anyways)

```

definition weak_universal ::
  "'x itself \Rightarrow ('a, 'c) Graph_PreRule \Rightarrow ('a, 'b) labeled_graph \Rightarrow
  ('a, 'b) labeled_graph \Rightarrow
  ('c \times 'b) set \Rightarrow ('c \times 'b) set \Rightarrow bool" where
  "weak_universal _ R G1 G2 f1 f2 \equiv (\forall h1 h2 G::('a, 'x) labeled_graph.
  (graph_homomorphism (snd R) G h1 \wedge graph_homomorphism G1
  G h2 \wedge f1 O h2 \subseteq h1)
  \longrightarrow (\exists h. graph_homomorphism G2 G h \wedge h2 \subseteq h))"

```

```

lemma weak_universalD[dest]:
  assumes "weak_universal (t:: 'x itself) R (G1::('a, 'b) labeled_graph)
  G2 f1 f2"
  shows "\ h1 h2 G::('a, 'x) labeled_graph.
  graph_homomorphism (snd R) G h1 \implies graph_homomorphism G1 G h2
  \implies f1 O h2 \subseteq h1

```

$\implies (\exists h. \text{graph_homomorphism } G_2 \ G \ h \ \wedge \ h_2 \subseteq h)$ "
using *assms unfolding weak_universal_def by metis*

lemma *weak_universalI*[intro]:
assumes " $\bigwedge h_1 \ h_2 \ G :: ('a, 'x) \text{ labeled_graph.}$
 $\text{graph_homomorphism (snd } R) \ G \ h_1 \implies \text{graph_homomorphism } G_1 \ G \ h_2$
 $\implies f_1 \ 0 \ h_2 \subseteq h_1$
 $\implies (\exists h. \text{graph_homomorphism } G_2 \ G \ h \ \wedge \ h_2 \subseteq h)$ "
shows "*weak_universal* (t :: 'x itself) R (G₁ :: ('a, 'b) labeled_graph)
G₂ f₁ f₂"
using *assms unfolding weak_universal_def by force*

Definition 13

definition *pushout_step* ::
"'x itself \Rightarrow ('a, 'c) Graph_PreRule \Rightarrow ('a, 'b) labeled_graph \Rightarrow
('a, 'b) labeled_graph \Rightarrow bool" **where**
"*pushout_step* t R G₁ G₂ \equiv subgraph G₁ G₂ \wedge
 $(\exists f_1 \ f_2. \text{graph_homomorphism (fst } R) \ G_1 \ f_1 \ \wedge$
 $\text{graph_homomorphism (snd } R) \ G_2 \ f_2 \ \wedge$
 $f_1 \subseteq f_2 \ \wedge$
 $\text{weak_universal } t \ R \ G_1 \ G_2 \ f_1 \ f_2$
)"

Definition 14

definition *Simple_WPC* ::
"'x itself \Rightarrow (('a, 'b) Graph_PreRule) set \Rightarrow (('a, 'd) graph_seq)
 \Rightarrow bool" **where**
"*Simple_WPC* t Rs S \equiv set_of_graph_rules Rs
 $\wedge (\forall i. (\text{graph } (S \ i) \ \wedge \ S \ i = S \ (\text{Suc } i)) \vee (\exists R \in \text{Rs. } \text{pushout_step}$
t R (S i) (S (Suc i))))"

lemma *Simple_WPCI* [intro]:
assumes "*set_of_graph_rules* Rs" "*graph* (S 0)"
" $\bigwedge i. (S \ i = S \ (\text{Suc } i)) \vee (\exists R \in \text{Rs. } \text{pushout_step } t \ R \ (S \ i)$
(S (Suc i))"
shows "*Simple_WPC* t Rs S"

proof -
have "*graph* (S i)" **for** i **proof**(*induct* i)
case (Suc i)
then show ?case **using** *assms(3) unfolding pushout_step_def subgraph_def*
by *metis*
qed (*fact assms*)
thus ?thesis **using** *assms unfolding Simple_WPC_def by auto*
qed

lemma *Simple_WPC_Chain*[simp]:
assumes "*Simple_WPC* t Rs S"
shows "*chain* S"
proof -

```

have "subgraph (S i) (S (Suc i))" for i using assms
  unfolding Simple_WPC_def pushout_step_def by (cases "graph (S i) ∧
S i = S (Suc i)", auto)
  thus ?thesis unfolding chain_def by auto
qed

```

Definition 14, second part.

```

inductive WPC ::
  "'x itself ⇒ (('a, 'b) Graph_PreRule) set ⇒ (('a, 'd) graph_seq)
⇒ bool"
where
  wpc_simpl [simp, intro]: "Simple_WPC t Rs S ⇒ WPC t Rs S"
  | wpc_combo [simp, intro]: "chain S ⇒ (∧ i. ∃ S'. S' 0 = S i ∧ chain_sup
S' = S (Suc i) ∧ WPC t Rs S') ⇒ WPC t Rs S"

```

```

lemma extensible_from_chainI:
  assumes ch:"chain S"
  and igh:"graph_homomorphism (S 0) C f"
  and ind:"∧ f i. graph_homomorphism (S i) C f ⇒
    ∃h. (graph_homomorphism (S (Suc i)) C h) ∧ agree_on (S
i) f h"
  shows "extensible (S 0, chain_sup S) C f"
proof -
  have ch:"chain S" using assms by auto
  hence r0:"∃x. graph_homomorphism (S 0) C x ∧ (0 = 0 → x = f)"
    using igh by auto
  { fix i x
    assume "graph_homomorphism (S i) C x ∧ (i = 0 → x = f)"
    hence "graph_homomorphism (S i) C x" by auto
    from ind[OF this]
    have "∃y. (graph_homomorphism (S (Suc i)) C y ∧ (Suc i = 0 → y
= f)) ∧ agree_on (S i) x y"
      by auto
    }
  with r0
  have "∃ g. (∀ i. (graph_homomorphism (S i) C (g i) ∧ (i = 0 → g
i = f))
    ∧ agree_on (S i) (g i) (g (Suc i)))" by (rule dependent_nat_choice)
  then obtain g where
    mtn:"g 0 = f"
    "graph_homomorphism (S i) C (g i)"
    "agree_on (S i) (g i) (g (i + 1))" for i by auto
  from extend_for_chain[OF mtn ch] show ?thesis.
qed

```

Towards Lemma 4, this is the key inductive property.

```

lemma wpc_least:
  assumes "WPC (t:: 'x itself) Rs S"
  shows "least t Rs (S 0) (chain_sup S)"

```

```

using assms
proof(induction S)
  case (wpc_simpl t Rs S)
  hence gr:"set_of_graph_rules Rs"
  and ps:" $\bigwedge i. S\ i = S\ (Suc\ i) \vee (\exists R \in Rs. \text{pushout\_step}\ t\ R\ (S\ i)\ (S\ (i + 1)))$ "
  unfolding Simple_WPC_def by auto
  have ch[intro]:"chain S" using wpc_simpl by auto
  show ?case
  proof fix C:: "('a, 'x) labeled_graph"
    assume cgC:"consequence_graph Rs C"
    show "maintained (S 0, chain_sup S) C"
    proof(standard,rule extensible_from_chainI,goal_cases)
      case (3 f x i)
      show ?case proof(cases "S i = S (Suc i)")
        case True
        with 3 show ?thesis by auto
      next
        case False
        with ps[of i,unfolded pushout_step_def] obtain R f1 f2 where
          R:"(fst R, snd R)  $\in$  Rs" and f1:"graph_homomorphism (fst R) (S
i) f1"
          and wu:"weak_universal t R (S i) (S (i + 1)) f1 f2" by auto
          from graph_homomorphism_composes[OF f1 3(2)]
          have ih_comp:"graph_homomorphism (fst R) C (f1 0 x)".
          with maintainedD[OF consequence_graphD(1)[OF cgC R]]
          have "extensible (fst R, snd R) C (f1 0 x)" by auto
          from this[unfolded extensible_def prod.sel]
          obtain g where g:"graph_homomorphism (snd R) C g" "f1 0 x  $\subseteq$ 
g"
          using agree_iff_subset[OF ih_comp] unfolding graph_homomorphism_def
          by auto
          from weak_universalD[OF wu g(1) 3(2) g(2)] obtain h where
            h:"graph_homomorphism (S (i + 1)) C h" "x  $\subseteq$  h" by auto
          hence "agree_on (S i) x h"
            by(subst agree_iff_subset[OF 3(2)], auto simp:graph_homomorphism_def)
          then show ?thesis using h(1) by auto
        qed
      qed auto
    qed auto
  next
  case (wpc_combo S t Rs)
  hence ps:" $\bigwedge i. \exists S'. S'\ 0 = S\ i \wedge$ 
    chain_sup S' = S (Suc i)  $\wedge$ 
    WPC t Rs S'  $\wedge$ 
    least t Rs (S' 0) (chain_sup S')"
  and ch[intro]:"chain S" unfolding Simple_WPC_def by auto
  show ?case proof fix C :: "('a, 'x) labeled_graph"
    assume cgC:"consequence_graph Rs C"

```

```

show "maintained (S 0, chain_sup S) C"
proof(standard,rule extensible_from_chainI,goal_cases)
  case (3 f g i)
  from ps[of i] have "least t Rs (S i) (S (Suc i))" by auto
  with cgC have ss:"subgraph (S i) (S (Suc i))" "maintained (S i,
S (Suc i)) C"
    unfolding least_def by auto
  from ss(2) 3(2) have "extensible (S i, S (Suc i)) C g" by auto
  thus ?case unfolding extensible_def prod.sel.
qed auto
qed auto
qed

```

Lemma 4.

```

lemma wpc_least_consequence_graph:
  assumes "WPC t Rs S" "consequence_graph Rs (chain_sup S)"
  shows "least_consequence_graph t Rs (S 0) (chain_sup S)"
  using wpc_least assms unfolding least_consequence_graph_def by auto
end

```

4 Graph rewriting and saturation

Here we describe graph rewriting, again without connecting it to semantics.

```

theory GraphRewriting
  imports RulesAndChains
  "HOL-Library.Infinite_Set"
begin

```

To describe Algorithm 1, we give a single step instead of the recursive call. This allows us to reason about its effect without dealing with non-termination. We define a worklist, saying what work can be done. A valid selection needs to be made in order to ensure fairness. To do a step, we define the function `extend`, and use it in `make_step`. A function that always makes a valid selection is used in this step.

```

abbreviation graph_of where
  "graph_of  $\equiv \lambda X. LG (snd X) \{0..<fst X\}$ "

```

```

definition nextMax :: "nat set  $\Rightarrow$  nat"
where
  "nextMax x  $\equiv$  if x = {} then 0 else Suc (Max x)"

```

```

lemma nextMax_max[intro]:
  assumes "finite x" "v  $\in$  x"
  shows "v < nextMax x" "v  $\leq$  nextMax x"
  using Max.coboundedI[OF assms] assms(2) unfolding nextMax_def by auto

```

```

definition worklist :: "nat  $\times$  ('a  $\times$  nat  $\times$  nat) set

```

$$\Rightarrow ((\text{'a}, \text{'b}) \text{ labeled_graph} \times (\text{'a}, \text{'b}) \text{ labeled_graph}) \text{ set}$$

$$\Rightarrow (\text{nat} \times (\text{'a}, \text{'b}) \text{ Graph_PreRule} \times (\text{'b} \times \text{nat}) \text{ set}) \text{ set}"$$

where

```
"worklist G Rs ≡ let G = graph_of G
  in {(N,R,f). R ∈ Rs ∧ graph_homomorphism (fst R) G f ∧ N = nextMax (Range
f)
      ∧ ¬ extensible R G f }"
```

definition valid_selection where

```
"valid_selection Rs G R f ≡
  let wl = worklist G Rs in
    (nextMax (Range f), R,f) ∈ wl ∧
    (∀ (N,_) ∈ wl. N ≥ nextMax (Range f)) ∧
    graph_rule R"
```

lemma valid_selection_exists:

```
assumes "worklist G Rs ≠ {}"
        "set_of_graph_rules Rs"
shows "∃ L R f. valid_selection Rs G R f"
```

proof -

```
  define wl where "wl = worklist G Rs" hence wl_ne:"wl ≠ {}" using assms(1)
  by auto
  let ?N = "LEAST N. N ∈ Domain wl"
  from wl_ne have "∃ N. N ∈ Domain wl" by auto
  with LeastI2 have "?N ∈ Domain wl" by metis
  then obtain L R f where NLRf:"(?N, (L,R),f) ∈ wl" by auto
  hence N_def:"?N = nextMax (Range f)"
    and in_Rs: "(L,R) ∈ Rs" unfolding wl_def worklist_def Let_def by
  auto
  from Least_le wl_ne Domain.intros case_prodI2
  have min:"(∀ (N',_) ∈ wl. N' ≥ ?N)" by (metis (no_types, lifting))
  from in_Rs have "finite_graph R" "subgraph L R"
    using assms(2)[unfolded set_of_graph_rules_def] by auto
  with min NLRf N_def show ?thesis unfolding wl_def[symmetric] valid_selection_def
  by auto
qed
```

definition valid_selector where

```
"valid_selector Rs selector ≡ ∀ G.
  (worklist G Rs ≠ {} → (∃ (R,f) ∈ UNIV. selector G = Some (R,f)
    ∧ valid_selection Rs G R f)) ∧
  (worklist G Rs = {} → selector G = None)"
```

lemma valid_selectorD[dest]:

```
assumes "valid_selector Rs selector"
shows "worklist G Rs = {} ↔ selector G = None"
      "selector G = Some (R,f) ⇒ valid_selection Rs G R f"
using assms[unfolded valid_selector_def, rule_format, of G]
by (cases "worklist G Rs = {}", auto)
```

The following gives a valid selector. This selector is not useful as concrete implementation, because it used the choice operation.

```

definition non_constructive_selector where
  "non_constructive_selector Rs G  $\equiv$  let wl = worklist G Rs in
    if wl = {} then None else Some (SOME (R,f). valid_selection Rs G R
  f) "

lemma non_constructive_selector:
  assumes "set_of_graph_rules Rs"
  shows "valid_selector Rs (non_constructive_selector Rs)"
  unfolding valid_selector_def proof((clarify,standard;clarify),goal_cases)
  case (1 n E)
  let ?x = "(SOME (R, f). valid_selection Rs (n, E) R f)"
  from valid_selection_exists[OF 1 assms]
  have " $\exists$  R f. valid_selection Rs (n, E) R f" by auto
  hence " $\exists$  x. valid_selection Rs (n, E) (fst x) (snd x)"
    by auto
  from this prod.case_eq_if tfl_some
  have " $\neg$  valid_selection Rs (n, E) (fst ?x) (snd ?x)  $\implies$  False"
    by (metis (mono_tags, lifting))
  thus ?case unfolding non_constructive_selector_def Let_def using 1 by
    (auto simp:prod_eq_iff)
qed (auto simp:non_constructive_selector_def)

```

The following is used to make a weak pushout step. In the paper, we aren't too specific on how this should be done. Here we are. We work on natural numbers in order to be able to pick fresh elements easily.

```

definition extend ::
  "nat  $\Rightarrow$  ('b, 'a::linorder) Graph_PreRule  $\Rightarrow$  ('a  $\times$  nat) set  $\Rightarrow$  ('a
 $\times$  nat) set" where
  "extend n R f  $\equiv$  f  $\cup$ 
    (let V_new = sorted_list_of_set (vertices (snd R) - vertices (fst R))
      in set (zip V_new [n.. $(n+\text{length } V\_new)$ ]))"

```

```

lemma nextMax_set[simp]:
  assumes "sorted xs"
  shows "nextMax (set xs) = (if xs = Nil then 0 else Suc (last xs))"
  using assms
proof(induct xs)
  case Nil show ?case unfolding nextMax_def by auto
next
  case (Cons a list)
  hence "list  $\neq$  []  $\implies$  fold max list a = last list"
    using list_sorted_max by (metis last.simps)
  thus ?case unfolding nextMax_def Max.set_eq_fold by auto
qed

```

```

lemma nextMax_Un_eq[simp]:

```

```

"finite x  $\implies$  finite y  $\implies$  nextMax (x  $\cup$  y) = max (nextMax x) (nextMax
y)"
  unfolding nextMax_def using Max_Un by auto

lemma extend:
  assumes "graph_homomorphism (fst R) (LG E {0..\equiv extend n R f"
  defines "G'  $\equiv$  LG ((on_triple g `` (edges (snd R)))  $\cup$  E) {0..\subseteq
g"
    "subgraph (LG E {0..\subseteq vertices (snd R)" and gr_R:"graph
(snd R)"
    unfolding subgraph_def graph_union_iff
    by auto
  hence fin_R_L[simp]:"finite ?R_L"
    and fin_L:"finite (vertices (fst R))"
    using finite_subset by auto
  from assms have f_dom:"Domain f = vertices (fst R)"
    and f_uni:"univalent f" unfolding graph_homomorphism_def by auto
  from assms[unfolded graph_homomorphism_def]
  have "f `` vertices (fst R)  $\subseteq$  vertices (LG E {0..\subseteq {0..\leq n"
    using f_ran Max_in[OF fin_f] by (simp add:nextMax_def Suc_leI subset_eq)

  have "x  $\in$  Domain ?g  $\implies$  x  $\notin$  Domain f" for x unfolding f_dom Let_def
  by auto
  hence g_not_f:"(x,y)  $\in$  ?g  $\implies$  (x,z)  $\notin$  f" for x y z by blast
  have uni_g':"univalent ?g" "univalent (converse ?g)" unfolding Let_def
  by auto
  with f_uni have uni_g:"univalent g" by (auto simp:g_def extend_def
g_not_f)

```

```

from fin_g have "(a,b) ∈ g ⇒ b < Suc (Max (Range g))" for a b
  unfolding less_Suc_eq_le by (rule Max.coboundedI) force
hence "(a,b) ∈ g ⇒ b < nextMax (Range g)" for a b
  unfolding nextMax_def by (cases "Range g = {}",auto)
hence in_g: "(a,b) ∈ g ⇒ b < max n (nextMax (Range g))" for a b by
fastforce
let ?G = "LG E {0..<n}"
have gr_G: "graph ?G" using assms(1) unfolding graph_homomorphism_def
by blast
hence "(a, aa, b) ∈ E ⇒ b < max n c" "(a, aa, b) ∈ E ⇒ aa < max
n c"
  for a aa b c by fastforce+
hence gr_G': "graph G'" unfolding G'_def restrict_def using in_g by
auto
show "subgraph (LG E {0..<n}) G'"
  unfolding subgraph_def2[OF gr_G gr_G'] unfolding G'_def by auto
have g_dom: "vertices (snd R) = Domain g" using subsLR
  unfolding g_def extend_def Domain_Un_eq f_dom by (auto simp: Let_def)
show "graph_homomorphism (snd R) G' g"
  by (intro graph_homomorphismI[OF g_dom _ uni_g _ gr_R gr_G']
      (auto simp: G'_def intro: in_g))
show "f ⊆ g" by (auto simp: g_def extend_def)
thus "agree_on (fst R) f g" using f_dom uni_g agree_on_subset equalityE
by metis
show "weak_universal t R ?G G' f g" proof fix a: "('b × 'x) set" fix
b G
  assume a: "graph_homomorphism (snd R) G a"
    "graph_homomorphism ?G G b" "f 0 b ⊆ a"
  hence univ_b: "univalent b" and univ_a: "univalent a"
    and rng_b: "Range b ⊆ vertices G" and rng_a: "Range a ⊆ vertices
G"
    and ep_b: "edge_preserving b (edges (LG E {0..<n})) (edges G)"
    and ep_a: "edge_preserving a (edges (snd R)) (edges G)"
  unfolding graph_homomorphism_def prod.sel labeled_graph.sel by blast+
from a have dom_b: "Domain b = {0..<n}"
  and dom_a: "Domain a = vertices (snd R)" and v6: "graph G"
  unfolding graph_homomorphism_def prod.sel labeled_graph.sel by auto

  have help_dom_b: "(y, z) ∈ b ⇒ n ≤ y ⇒ False" for y z using dom_b
  by (metis Domain.DomainI atLeastLessThan_iff not_less)
  have disj_doms: "Domain b ∩ Domain (?g-1 0 a) = {}" using help_dom_b
  unfolding Let_def by (auto dest!: set_zip_leftD)

  have "max n (nextMax (Range ?g)) = n + length (sorted_list_of_set
?R_L)" (is "_ = ?len")
  unfolding Let_def Range_snd set_map[symmetric] map_snd_zip[OF ln]
nextMax_set[OF sorted_upt]
  by (fastforce simp del: length_sorted_list_of_set)
  hence n_eq: "?len = max n (nextMax (Range g))"

```

```

    unfolding Range_snd[symmetric] g_def extend_def Range_Un_eq
      nextMax_Un_eq[OF fin_f fin_g'(2)] max.assoc[symmetric]
max_absorb1[OF nextMax_f]
  by auto

let ?h = "b  $\cup$  ?g-1 0 a"

have dg:"Domain (?g-1) = {n.. $\max$  n (nextMax (Range g))}"
  unfolding Let_def Domain_converse Range_set_zip[OF ln] atLeastLessThan_upt
  unfolding Range_snd n_eq ..
have "?g '' Domain a = ?g '' (?R_L  $\cup$  vertices (fst R))"
  using dom_a subsLR by auto
also have "... = ?g '' ?R_L  $\cup$  ?g '' vertices (fst R)" by blast
also have "?g '' vertices (fst R) = {}" apply(rule Image_outside_Domain)
  unfolding Let_def Domain_set_zip[OF ln] by auto
also have "?g '' ?R_L = Range ?g" apply(rule Image_Domain)
  unfolding Let_def Domain_set_zip[OF ln] by auto
finally have dg2:"?g '' Domain a = {n.. $\max$  n (nextMax (Range g))}"
  unfolding Let_def Range_set_zip[OF ln] set_sorted_list_of_set[OF
fin_R_L]
  unfolding n_eq set_upt by auto
have "Domain (?g-1 0 a) = {n.. $\max$  n (nextMax (Range g))}"
  unfolding Domain_id_on converse_converse dg dg2 by auto
hence v1: "vertices G' = Domain ?h" unfolding G'_def Domain_Un_eq
dom_b by auto
have "b '' vertices G'  $\subseteq$  vertices G" "(?g-1 0 a) '' vertices G'  $\subseteq$ 
vertices G"
  using rng_a rng_b by auto
hence v2: "?h '' vertices G'  $\subseteq$  vertices G" by blast
have v3: "univalent ?h"
  using disj_doms univalent_union[OF univ_b univalent_composes[OF
uni_g'(2) univ_a]] by blast

{ fix l x y x' y' assume a2:"(l,x,y)  $\in$  edges G'" "(x,x')  $\in$  ?h" "(y,y')
 $\in$  ?h"
  have "(l,x',y')  $\in$  edges G" proof(cases "(l,x,y)  $\in$  edges ?G")
  case True
  with gr_G[THEN restrictD]
  have "x  $\in$  Domain b" "y  $\in$  Domain b" unfolding dom_b by auto
  hence "x  $\notin$  Domain (converse ?g 0 a)" "y  $\notin$  Domain (converse ?g
0 a)"
    using disj_doms by blast+
  hence "(x,x')  $\in$  b" "(y,y')  $\in$  b" using a2 by auto
  with ep_b True show ?thesis unfolding edge_preserving by auto
next
  have "g 0 ?h = f 0 b  $\cup$  ?g 0 b  $\cup$  ((f 0 ?g-1) 0 a  $\cup$  (?g 0 ?g-1)
0 a)"
    unfolding g_def extend_def by blast
  also have "(?g 0 ?g-1) = Id_on ?R_L"

```

```

      unfolding univalent_0_converse[OF uni_g'(2)] unfolding Let_def
by auto
      also have "(f 0 ?g-1) = {}" using f_ran unfolding Let_def by
(auto dest!:set_zip_leftD)
      also have "?g 0 b = {}" using help_dom_b unfolding Let_def by
(auto dest!:set_zip_rightD)
      finally have g0h:"g 0 ?h ⊆ a" using a(3) by blast
      case False
      hence "(l,x,y) ∈ on_triple g ‘‘ edges (snd R)'' using a2(1) un-
folding G'_def by auto
      then obtain r_x r_y
      where r:"(l,r_x,r_y) ∈ edges (snd R)" "(r_x,x) ∈ g" "(r_y,y)
∈ g" by auto
      hence "(r_x,x') ∈ a" "(r_y,y') ∈ a" using g0h a2(2,3) by auto
      hence "(l,x',y') ∈ on_triple a ‘‘ edges (snd R)'' using r(1) un-
folding on_triple_def by auto
      thus ?thesis using ep_a unfolding edge_preserving by auto
      qed
    }
    hence v4: "edge_preserving ?h (edges G') (edges G)" by auto
    have "graph_homomorphism G' G ?h" by (fact graph_homomorphismI[OF
v1 v2 v3 v4 gr_G' v6])
    thus "∃ h. graph_homomorphism G' G h ∧ b ⊆ h" by auto
  qed
qed

```

Showing that the extend function indeed creates a valid pushout.

```

lemma selector_pushout:
  assumes "valid_selector Rs selector" "selector G'' = Some (R,f)"
  defines "G ≡ graph_of G''"
  assumes "graph G"
  defines "g ≡ extend (fst G'') R f"
  defines "G' ≡ LG (on_triple g ‘‘ edges (snd R) ∪ (snd G'')'' {0..

```

Making a single step in Algorithm 1. A prerequisite is that its first argument is a *valid_selector*.

definition *make_step* where

```
"make_step selector S ≡
  case selector S of
    None ⇒ S |
    Some (R,f) ⇒ (let g = extend (fst S) R f in
      (max (fst S) (nextMax (Range g)), (on_triple g ‘‘ (edges (snd
R))) ∪ (snd S)))"
```

lemma *WPC_through_make_step*:

```
assumes "set_of_graph_rules Rs" "graph (graph_of (X 0))"
  and makestep: "∀ i. X (Suc i) = make_step selector (X i)"
  and selector: "valid_selector Rs selector"
shows "Simple_WPC t Rs (λ i. graph_of (X i))" "chain (λ i. graph_of
(X i))"
```

proof

```
note ms = makestep[unfolded make_step_def,rule_format]
have gr:"graph (graph_of (X i))" for i proof(induct i)
  case (Suc i)
  then show ?case proof(cases "selector (X i)")
    case None
    then show ?thesis using ms Suc by auto
  next
  case (Some a)
  then obtain R f where Some:"selector (X i) = Some (R,f)" by fastforce
  then show ?thesis using ms[of i,unfolded Some Let_def]
    selector_pushout[OF selector Some Suc,of t
      ,unfolded pushout_step_def subgraph_def]
    by auto
  qed
qed (fact assms)
show "chain (λ i. graph_of (X i))" unfolding chain_def
proof(clarify) fix i
  show "subgraph (graph_of (X i)) (graph_of (X (i + 1)))"
  proof(cases "selector (X i)")
    case None
    then show ?thesis using ms gr by (auto intro!:graph_homomorphismI)
  next
  case Some
  then obtain R f where Some:"selector (X i) = Some (R,f)" by fastforce
  then show ?thesis using ms selector_pushout[OF selector Some gr,of
t]
    unfolding pushout_step_def Let_def by simp
  qed
qed
show "graph_of (X i) = graph_of (X (Suc i)) ∨
  (∃ R∈Rs. pushout_step t R (graph_of (X i)) (graph_of (X (Suc
i))))" for i
```

```

proof(cases "selector (X i)")
  case None thus ?thesis using ms by auto
next
  case Some
  then obtain R f where Some:"selector (X i) = Some (R,f)" by fastforce
  hence "R ∈ Rs"
  using valid_selectorD(2)[OF selector,unfolded valid_selection_def
worklist_def Let_def]
  by(cases R,blast)
  then show ?thesis using ms[of i,unfolded Some Let_def] selector_pushout[OF
selector Some gr]
  unfolding make_step_def by auto
qed
qed (fact assms)+

lemma N_occurs_finitely_often:
  assumes "finite Rs" "set_of_graph_rules Rs" "graph (graph_of (X 0))"
  and makestep: "∧ i. X (Suc i) = make_step selector (X i)"
  and selector: "valid_selector Rs selector"
  shows "finite {(R,f). ∃ i. R ∈ Rs ∧ graph_homomorphism (fst R) (graph_of
(X i)) f
  ∧ nextMax (Range f) ≤ N}" (is "finite {(R,f).?P
R f}")
proof -
  have prod_eq : "(∀ x ∈ {(x, y). A x y}. B x) ↔ (∀ x. A (fst x) (snd
x) → B x)"
  "(x ∈ {(x, y). A x y} ↔ (A (fst x) (snd x)))"
  for A B x by auto
  let ?S = "{(R,f).?P R f}"
  let "?Q R f" = "Domain f = vertices (fst R::('a, 'b) Graph_PreRule)"
  ∧ univalent f ∧ nextMax (Range f) ≤ N"
  have seteq:"(∪ R∈Rs. {(R', f). R' = R ∧ ?Q R f}) = {(R,f). R ∈ Rs
  ∧ ?Q R f}" by auto
  have "∀ R ∈ Rs. finite {(R',f). R' = R ∧ ?Q R f}"
  proof fix R assume "R ∈ Rs"
    hence fin:"finite (vertices (fst R))" using assms by auto
    hence fin2:"finite (Pow (vertices (fst R) × {0..N}))" by auto
    have fin:"Domain x = vertices (fst R) ⇒ univalent x ⇒ finite
(snd ' x)"
    for x:: "('b × nat) set" using fin univalent_finite[of x] by simp
    hence "Domain f = vertices (fst R) ⇒
    univalent f ⇒ (a,b) ∈ f ⇒ nextMax (Range f) ≤ N ⇒ b ≤ N"
  for f a b
  unfolding Range_snd using image_eqI nextMax_max(2) snd_conv order.trans
  by metis
  hence sub:"{f. ?Q R f} ⊆ Pow (vertices (fst R) × {0..N})"
  using nextMax_max[OF fin] by (auto simp:Range_snd image_def)
  from finite_subset[OF sub fin2] show "finite {(R',f). R' = R ∧ ?Q
R f}" by auto

```

```

qed
from this[folded finite_UN[OF assms(1)],unfolding seteq]
have fin:"finite {(R,f). R ∈ Rs ∧ ?Q R f}".
have "?P R f ⇒ R ∈ Rs ∧ ?Q R f" for R f
  unfolding graph_homomorphism_def by auto
hence "?S ⊆ {(R,f). R ∈ Rs ∧ ?Q R f}" unfolding subset_eq prod_eq
by blast
from finite_subset[OF this fin] show ?thesis by auto
qed

lemma inj_on_infinite:
  assumes "infinite A" "inj_on f A" "range f ⊆ B"
  shows "infinite B"
proof -
  from assms[unfolding infinite_iff_countable_subset] obtain g::"nat ⇒
'a" where
  g:"inj g ∧ range g ⊆ A" by blast
  hence i:"inj (f o g)" using assms(2) using comp_inj_on inj_on_subset
  by blast
  have "range (f o g) ⊆ B" using assms(3) by auto
  with i show ?thesis
  unfolding infinite_iff_countable_subset by blast
qed

lemma makestep_makes_selector_inj:
  assumes "selector (X y) = Some (R,f)"
  "selector (X x) = Some (R,f)"
  "valid_selector Rs selector"
  and step: "∀ i. X (Suc i) = make_step selector (X i)"
  and chain:"chain (λ i. graph_of (X i))"
  shows "x = y"
proof(rule ccontr)
  assume a:"x ≠ y"
  define x' y' where "x' ≡ min x y" "y' ≡ max x y"
  hence xy:"selector (X x') = Some (R, f)" "selector (X y') = Some (R,
f)" "x' < y'"
  using assms(1,2) a unfolding min_def max_def by auto
  with valid_selectorD assms
  have "valid_selection Rs (X x') R f" "valid_selection Rs (X y') R f"
  by auto
  hence not_ex:"¬ extensible R (graph_of (X y')) f"
  and hom:"graph_homomorphism (fst R) (graph_of (X x')) f" "graph_rule
R"
  unfolding valid_selection_def Let_def worklist_def by auto
  have X:"X (Suc x') = (max (fst (X x')) (nextMax (Range (extend (fst
(X x')) R f))),
on_triple (extend (fst (X x')) R f) "edges (snd R) ∪ snd (X
x'))"
  unfolding step[unfolding make_step_def Let_def,rule_format] xy by auto

```

```

let ?ex = "extend (fst (X x')) R f"
have hom:"graph_homomorphism (snd R) (graph_of (X (Suc x'))) ?ex"
  and agr:"agree_on (fst R) f ?ex" using extend(1,2)[OF hom] un-
folding X by auto
from xy have "Suc x' ≤ y'" by auto
with chain[unfolded chain_def2] have "subgraph (graph_of (X (Suc x')))
(graph_of (X y'))" by auto
from subgraph_preserves_hom[OF this hom]
have hom:"graph_homomorphism (snd R) (graph_of (X y')) ?ex".
with agr have "extensible R (graph_of (X y')) f" unfolding extensible_def
by auto
thus False using not_ex by auto
qed

```

lemma fair_through_make_step:

```

assumes "finite Rs" "set_of_graph_rules Rs" "graph (graph_of (X 0))"

  and makestep: "∀ i. X (Suc i) = make_step selector (X i)"
  and selector: "valid_selector Rs selector"
shows "fair_chain Rs (λ i. graph_of (X i))"
proof
show chn:"chain (λ i. graph_of (X i))" using WPC_through_make_step assms
by blast
fix R f i
assume Rs:"R ∈ Rs" and h:"graph_homomorphism (fst R) (graph_of (X
i)) f"
hence R:"finite (vertices (snd R))" "subgraph (fst R) (snd R)" "finite
(vertices (fst R))"
using assms by auto
hence f:"finite f" "finite (Range f)" "finite (Domain f)" "univalent
f"
using h unfolding graph_homomorphism_def Range_snd by auto
define N where "N ≡ nextMax (Range f)"
fix S
let "?Q X' j" = "fst X' ∈ Rs
  ∧ graph_homomorphism (fst (fst X')) (graph_of (X (j+i)))
(snd X')
  ∧ nextMax (Range (snd X')) ≤ N"
let ?S = "{(R,f). ∃ j. ?Q (R,f) j}"
from assms(4) have "∧ ia. X (Suc ia + i) = make_step selector (X (ia
+ i))" by auto
note r = assms(1,2) chain_then_restrict[OF chn] this assms(5)
from N_occurs_finitely_often[of Rs "λ j. X (j + i)",OF r]
have fin_S:"finite ?S" by auto
{ assume a:"∀ j. ¬ extensible R (graph_of (X j)) f"
let "?P X' j" = "?Q X' j ∧ Some X' = selector (X (j+i))"
{ fix j let ?j = "j+i" have "?j ≥ i" by auto
from subgraph_preserves_hom[OF chain[OF chn this] h]
have h:"graph_homomorphism (fst R) (graph_of (X ?j)) f".

```

```

have "¬ extensible R (graph_of (X ?j)) f" using a by blast
with h Rs have wl:"(nextMax (Range f),R,f) ∈ worklist (X ?j) Rs"

  unfolding worklist_def Let_def set_eq_iff by auto
  hence "worklist (X ?j) Rs ≠ {}" by auto
  with valid_selectorD[OF selector]
  obtain R' f'
    where sel:"Some (R',f') = selector (X ?j)"
          "valid_selection Rs (X ?j) R' f'" by auto
  hence max:"(nextMax (Range f'), R', f') ∈ worklist (X ?j) Rs"
        "(∀ (N, _) ∈ worklist (X ?j) Rs. nextMax (Range f') ≤ N)"
    unfolding valid_selection_def Let_def by auto
  with wl have "nextMax (Range f') ≤ N" unfolding N_def by auto
  with max(1)[unfolded worklist_def Let_def mem_Collect_eq prod.case]
sel(1)
  have "∃ X'. ?P X' j" by (metis fst_conv snd_conv)
}
then obtain ch where ch:"∧ j. ?P (ch j) j" by metis
have inj:"inj ch" proof fix x y assume "ch x = ch y"
  with ch[of x] ch[of y]
  have "selector (X (x + i)) = Some (ch x)" "selector (X (y + i))
= Some (ch x)" by auto
  with makestep_makes_selector_inj[OF _ _ selector makestep chn] have
"x + i = y + i"
  by (cases "ch x",metis (full_types))
  thus "x = y" by auto
qed
  have "ch x ∈ ?S" for x using ch[of x] unfolding mem_Collect_eq by(intro
case_prodI2) metis
  hence "range ch ⊆ ?S" unfolding UNIV_def by(rule image_Collect_subsetI)
  with infinite_iff_countable_subset inj have "infinite ?S" by blast
  with fin_S have "False" by auto
}
thus "∃ j. extensible R (graph_of (X j)) f" by auto
qed

fun mk_chain where
  "mk_chain sel Rs init 0 = init" |
  "mk_chain sel Rs init (Suc n) = mk_chain sel Rs (make_step sel init)
n"

lemma mk_chain:
  "∀ i. mk_chain sel Rs init (Suc i) = make_step sel (mk_chain sel Rs
init i)"
proof
  fix i
  show "mk_chain sel Rs init (Suc i) = make_step sel (mk_chain sel Rs
init i)"
  by (induct i arbitrary:init,auto)

```

qed

Algorithm 1, abstractly.

abbreviation *the_lcg* where

```
"the_lcg sel Rs init  $\equiv$  chain_sup ( $\lambda i$ . graph_of (mk_chain sel Rs init i))"
```

lemma *mk_chain_edges*:

```
  assumes "valid_selector Rules sel"
    "  $\bigcup ((edges \ o \ snd) \ ' \ Rules) \subseteq L \times UNIV$ "
    "edges (graph_of G)  $\subseteq L \times UNIV$ "
  shows "edges (graph_of (mk_chain sel Rules G i))  $\subseteq L \times UNIV$ "
using assms(3) proof(induct i arbitrary:G)
  case 0
  then show ?case using assms(2) by auto
next
  case (Suc i G)
  hence "edges (graph_of (make_step sel G))  $\subseteq L \times UNIV$ "
  proof(cases "sel G")
    case None show ?thesis unfolding None make_step_def using Suc by
  auto
  next
    case (Some a)
    then obtain R f where Some:"sel G = Some (R, f)" by fastforce
    hence "(a, x, y)  $\in$  edges (snd R)  $\implies a \in L$ " for a x y
      using assms(2) valid_selectorD(2)[OF assms(1) Some]
      unfolding valid_selection_def Let_def worklist_def by auto
    then show ?thesis unfolding Some make_step_def Let_def using Suc
  by auto
  qed
  thus ?case unfolding mk_chain.simps by(rule Suc)
qed
```

lemma *the_lcg_edges*:

```
  assumes "valid_selector Rules sel"
    "fst ' ( $\bigcup ((edges \ o \ snd) \ ' \ Rules)) \subseteq L$ " (is "fst '?fR  $\subseteq$  _")
    "fst ' snd G  $\subseteq L$ "
  shows "fst ' edges (the_lcg sel Rules G)  $\subseteq L$ "
proof -
  from assms have "fst '?fR  $\times UNIV \subseteq L \times UNIV$ " "fst '(edges (graph_of
G))  $\times UNIV \subseteq L \times UNIV$ "
  by auto
  hence "( $\bigcup ((edges \ o \ snd) \ ' \ Rules)) \subseteq L \times UNIV$ " "edges (graph_of G)
 $\subseteq L \times UNIV$ "
  using fst_UNIV[of ?fR] fst_UNIV[of "(edges (graph_of G))"] by blast+
  note assms = assms(1) this
  have "edges (graph_of (mk_chain sel Rules G i))  $\subseteq L \times UNIV$ " for i
    using mk_chain_edges[OF assms,unfolded Times_subset_cancel2[OF UNIV_I]].
  hence "edges (the_lcg sel Rules G)  $\subseteq L \times UNIV$ " unfolding chain_sup_def
```

```

by auto
  thus ?thesis by auto
qed

  Lemma 9.

lemma lcg_through_make_step:
assumes "finite Rs" "set_of_graph_rules Rs" "graph (graph_of init)"
        "valid_selector Rs sel"
  shows "least_consequence_graph t Rs (graph_of init) (the_lcg sel Rs
init)"
proof -
  from assms have gr:"graph (graph_of (mk_chain sel Rs init 0))" by auto
  note assms = assms(1,2) this mk_chain assms(4)
  from set_of_graph_rulesD[OF assms(2)]
  have "( $\bigwedge R. R \in Rs \implies \text{subgraph (fst R) (snd R) } \wedge \text{finite\_graph (fst R)}$ )" by auto
  from fair_chain_impl_consequence_graph[OF fair_through_make_step[OF
assms] this]
    wpc_simpl[OF WPC_through_make_step(1)[OF assms(2-)], THEN wpc_least]

  show ?thesis unfolding least_consequence_graph_def by auto
qed

end

```

5 Semantics in labeled graphs

```

theory LabeledGraphSemantics
imports LabeledGraphs
begin

```

GetRel describes the main way we interpret graphs: as describing a set of binary relations.

```

definition getRel where
"getRel l G = {(x,y). (l,x,y)  $\in$  edges G}"

```

```

lemma getRel_dom:
  assumes "graph G"
  shows "(a,b)  $\in$  getRel l G  $\implies$  a  $\in$  vertices G"
        "(a,b)  $\in$  getRel l G  $\implies$  b  $\in$  vertices G"
  using assms unfolding getRel_def by auto

```

```

lemma getRel_subgraph[simp]:
  assumes "(y, z)  $\in$  getRel l G" "subgraph G G'"
  shows "(y,z)  $\in$  getRel l G'" using assms by (auto simp:getRel_def subgraph_def
graph_union_iff)

```

```

lemma getRel_homR:
  assumes "(y, z)  $\in$  getRel l G" "(y,u)  $\in$  f" "(z,v)  $\in$  f"

```

```

shows "(u, v) ∈ getRel 1 (map_graph f G)"
using assms by (auto simp:getRel_def on_triple_def)

lemma getRel_hom[intro]:
  assumes "(y, z) ∈ getRel 1 G" "y ∈ vertices G" "z ∈ vertices G"
  shows "(f y, f z) ∈ getRel 1 (map_graph_fn G f)"
  using assms by (auto intro!:getRel_homR)

lemma getRel_hom_map[simp]:
  assumes "graph G"
  shows "getRel 1 (map_graph_fn G f) = map_prod f f ‘ (getRel 1 G)"
proof
  { fix x y
    assume a:"(x, y) ∈ getRel 1 G"
    hence "x ∈ vertices G" "y ∈ vertices G" using assms unfolding getRel_def
  by auto
    hence "(f x, f y) ∈ getRel 1 (map_graph_fn G f)" using a by auto
  }
  then show "map_prod f f ‘ getRel 1 G ⊆ getRel 1 (map_graph_fn G f)"
  by auto
qed (cases G, auto simp:getRel_def)

```

The thing called term in the paper is called *allegorical_term* here. This naming is chosen because an allegory has precisely these operations, plus identity.

```

datatype 'v allegorical_term
= A_Int "'v allegorical_term" "'v allegorical_term"
| A_Cmp "'v allegorical_term" "'v allegorical_term"
| A_Cnv "'v allegorical_term"
| A_Lbl (a_lbl : 'v)

```

The interpretation of terms, Definition 2.

```

fun semantics where
"semantics G (A_Int a b) = semantics G a ∩ semantics G b" |
"semantics G (A_Cmp a b) = semantics G a ∩ semantics G b" |
"semantics G (A_Cnv a) = converse (semantics G a)" |
"semantics G (A_Lbl l) = getRel 1 G"

```

```

notation semantics (<:_:[_]> 55)

```

```

type_synonym 'v sentence = "'v allegorical_term × 'v allegorical_term"

```

```

datatype 'v Standard_Constant = S_Top | S_Bot | S_Idt | S_Const 'v

```

Definition 3. We don't define sentences but instead simply work with pairs of terms.

```

abbreviation holds where
"holds G S ≡ :G:[fst S] = :G:[snd S]"
notation holds (infix <|=> 55)

```

```

abbreviation subset_sentence where
  "subset_sentence A B  $\equiv$  (A,A_Int A B)"

notation subset_sentence (infix <math>\sqsubseteq</math> 60)

  Lemma 1.

lemma sentence_iff[simp]:
  "G  $\models$  e1  $\sqsubseteq$  e2 = (:G:[e1]  $\subseteq$  :G:[e2])" and
  eq_as_subsets:
  "G  $\models$  (e1, e2) = (G  $\models$  e1  $\sqsubseteq$  e2  $\wedge$  G  $\models$  e2  $\sqsubseteq$  e1)"
  by auto

lemma map_graph_in[intro]:
  assumes "graph G" "(a,b)  $\in$  :G:[e]"
  shows "(f a, f b)  $\in$  :map_graph_fn G f:[e]"
  using assms by(induct e arbitrary: a b,auto intro!:relcompI)

lemma semantics_subset_vertices:
  assumes "graph A" shows ":A:[e]  $\subseteq$  vertices A  $\times$  vertices A"
  using assms by(induct e,auto simp:getRel_def)

lemma semantics_in_vertices:
  assumes "graph A" "(a,b)  $\in$  :A:[e]"
  shows "a  $\in$  vertices A" "b  $\in$  vertices A"
  using assms by(induct e arbitrary:a b,auto simp:getRel_def)

lemma map_graph_semantics[simp]:
  assumes "graph A" and i:"inj_on f (vertices A)"
  shows ":map_graph_fn A f:[e] = map_prod f f ' (:A:[e])"
  proof(induct e)
    have io:"inj_on (map_prod f f) (vertices A  $\times$  vertices A)"
      using i unfolding inj_on_def by simp
    note s = semantics_subset_vertices[OF assms(1)]
    case (A_Int e1 e2) thus ?case by (auto simp:inj_on_image_Int[OF io s])
  next
    case (A_Cmp e1 e2)
    { fix xa ya xb yb assume "(xa, ya)  $\in$  :A:[e1]" "(xb, yb)  $\in$  :A:[e2]"
      "f ya = f xb"
      moreover hence "ya = xb"
        using i[unfolded inj_on_def] semantics_in_vertices[OF assms(1)]
      by auto
      ultimately have "(f xa, f yb)  $\in$  map_prod f f ' ((:A:[e1])  $\cup$  (:A:[e2]))"
      by auto
    }
    with A_Cmp show ?case by auto
  qed (insert assms,auto)

lemma graph_union_semantics:
  shows "(:A:[e])  $\cup$  (:B:[e])  $\subseteq$  :graph_union A B:[e]"

```

```

by(induct e,auto simp:getRel_def)

lemma subgraph_semantics:
  assumes "subgraph A B" "(a,b) ∈ :A:[e]"
  shows "(a,b) ∈ :B:[e]"
  using assms by(induct e arbitrary: a b,auto intro!:relcompI)

lemma graph_homomorphism_semantics:
  assumes "graph_homomorphism A B f" "(a,b) ∈ :A:[e]" "(a,a') ∈ f" "(b,b')
  ∈ f"
  shows "(a',b') ∈ :B:[e]"
  using assms proof(induct e arbitrary: a b a' b')
  have g:"graph A" using assms unfolding graph_homomorphism_def2 by auto
  case (A_Cmp e1 e2)
  then obtain y where y:"(a, y) ∈ :A:[e1]" "(y, b) ∈ :A:[e2]" by auto
  hence "y∈vertices A" using semantics_in_vertices[OF g] by auto
  with A_Cmp obtain y' where "(y,y') ∈ f" unfolding graph_homomorphism_def
  by auto
  from A_Cmp(1)[OF assms(1) y(1) A_Cmp(5) this] A_Cmp(2)[OF assms(1) y(2)
  this A_Cmp(6)]
  show ?case by auto
next
  case (A_Lbl x) thus ?case by (auto simp:getRel_def graph_homomorphism_def2
  graph_union_iff)
qed auto

lemma graph_homomorphism_nonempty:
  assumes "graph_homomorphism A B f" ":A:[e] ≠ {}"
  shows ":B:[e] ≠ {}"
proof-
  from assms have g:"graph A" unfolding graph_homomorphism_def by auto
  from assms obtain a b where ab:"(a,b) ∈ :A:[e]" by auto
  from semantics_in_vertices[OF g ab] obtain a' b' where
    "(a,a') ∈ f" "(b,b') ∈ f" using assms(1) unfolding graph_homomorphism_def
  by auto
  from graph_homomorphism_semantics[OF assms(1) ab this] show ?thesis
  by auto
qed

lemma getRel_map_fn[intro]:
  assumes "a2 ∈ vertices G" "b2 ∈ vertices G" "(a2,b2) ∈ getRel l G"
    "f a2 = a" "f b2 = b"
  shows "(a,b) ∈ getRel l (map_graph_fn G f)"
proof -
  from assms(1,2)
  have "((l, a2, b2), (l, f a2, f b2)) ∈ on_triple {(a, f a) |a. a ∈
  vertices G}" by auto
  thus ?thesis using assms(3-) by (simp add:getRel_def BNF_Def.Gr_def
  Image_def,blast)

```

qed

end

6 Standard Models

We define the kind of models we are interested in here. In particular, we care about standard graphs. To allow some reuse, we distinguish a generic version called *standard*, from an instantiated abbreviation *standard'*. There is little we can prove about these definition here, except for Lemma 2.

```
theory StandardModels
imports LabeledGraphSemantics Main
begin

abbreviation "a_bot  $\equiv$  A_Lbl S_Bot"
abbreviation "a_top  $\equiv$  A_Lbl S_Top"
abbreviation "a_idt  $\equiv$  A_Lbl S_Idt"
notation a_bot (< $\perp$ >)
notation a_top (< $\top$ >)
notation a_idt (<1>)

type_synonym 'v std_term = "'v Standard_Constant allegorical_term"
type_synonym 'v std_sentence = "'v std_term  $\times$  'v std_term"
type_synonym ('v,'a) std_graph = "('v Standard_Constant, ('v+'a)) labeled_graph"

abbreviation ident_rel where
"ident_rel idt G  $\equiv$  getRel idt G = ( $\lambda$  x.(x,x)) ' vertices G"

lemma ident_relI[intro]:
  assumes min:" $\bigwedge$  x. x  $\in$  vertices G  $\implies$  (x,x)  $\in$  getRel idt G"
  and max1:" $\bigwedge$  x y. (x,y)  $\in$  getRel idt G  $\implies$  x = y"
  and max2:" $\bigwedge$  x y. (x,y)  $\in$  getRel idt G  $\implies$  x  $\in$  vertices G"
shows "ident_rel idt G"
proof
  from max1 max2 have " $\bigwedge$  x y. (x,y)  $\in$  getRel idt G  $\implies$  (x = y  $\wedge$  x  $\in$ 
vertices G)" by simp
  thus "getRel idt G  $\subseteq$  ( $\lambda$ x. (x, x)) ' vertices G" by auto
  show " $(\lambda$ x. (x, x)) ' vertices G  $\subseteq$  getRel idt G" using min by auto
qed
```

Definition 4, generically.

```
definition standard :: "('l  $\times$  'v) set  $\implies$  'l  $\implies$  'l  $\implies$  'l  $\implies$  ('l, 'v) labeled_graph
 $\implies$  bool" where
"standard C b t idt G
 $\equiv$  G = restrict G
 $\wedge$  vertices G  $\neq$  {}
 $\wedge$  ident_rel idt G
```

```

 $\wedge$  getRel b G = {}
 $\wedge$  getRel t G = {(x,y). x $\in$ vertices G  $\wedge$  y $\in$ vertices G}
 $\wedge$  ( $\forall$  (l,v)  $\in$  C. getRel l G = {(v,v)})"

```

Definition 4.

```

abbreviation standard' :: "'v set  $\Rightarrow$  ('v,'a) std_graph  $\Rightarrow$  bool" where
"standard' C  $\equiv$  standard (( $\lambda$  c. (S_Const c,Inl c)) ' C) S_Bot S_Top S_Idt"

```

Definition 5.

```

definition model :: "'v set  $\Rightarrow$  ('v,'a) std_graph  $\Rightarrow$  ('v std_sentence) set
 $\Rightarrow$  bool" where
"model C G T  $\equiv$  standard' C G  $\wedge$  ( $\forall$  S  $\in$  T. G  $\models$  S)"

```

Definition 5.

```

abbreviation consistent :: "'b itself  $\Rightarrow$  'v set  $\Rightarrow$  ('v std_sentence) set
 $\Rightarrow$  bool" where
"consistent _ C T  $\equiv$   $\exists$  (G::('v,'b) std_graph). model C G T"

```

Definition 6.

```

definition entails :: "'b itself  $\Rightarrow$  'v set  $\Rightarrow$  ('v std_sentence) set  $\Rightarrow$ 
'v std_sentence  $\Rightarrow$  bool" where
"entails _ C T S  $\equiv$   $\forall$  (G::('v,'b) std_graph). model C G T  $\longrightarrow$  G  $\models$  S"

```

```

lemma standard_top_not_bot[intro]:
"standard' C G  $\Longrightarrow$  :G:[ $\perp$ ]  $\neq$  :G:[ $\top$ ]"
  unfolding standard_def by auto

```

Lemma 2.

```

lemma consistent_iff_entails_nonsense:
"consistent t C T = ( $\neg$  entails t C T ( $\perp$ , $\top$ ))"
proof
  show "consistent t C T  $\Longrightarrow$   $\neg$  entails t C T ( $\perp$ ,  $\top$ )"
    using standard_top_not_bot unfolding entails_def model_def
    by fastforce
qed (auto simp:entails_def model_def)

```

end

7 Translating terms into Graphs

We define the translation function and its properties.

```

theory RuleSemanticsConnection
imports LabeledGraphSemantics RulesAndChains
begin

```

Definition 15.

```

fun translation :: "'c allegorical_term  $\Rightarrow$  ('c, nat) labeled_graph" where
"translation (A_Lbl l) = LG {(l,0,1)} {0,1}" |

```

```

"translation (A_Cnv e) = map_graph_fn (translation e) (λ x. if x<2 then
(1-x) else x)" |
"translation (A_Cmp e1 e2)
= (let G1 = translation e1 ; G2 = translation e2
in graph_union (map_graph_fn G1 (λ x. if x=0 then 0 else x+card(vertices
G2)-1))
(map_graph_fn G2 (λ x. if x=0 then card (vertices
G2) else x)))" |
"translation (A_Int e1 e2)
= (let G1 = translation e1 ; G2 = translation e2
in graph_union G1 (map_graph_fn G2 (λ x. if x<2 then x else x+card(vertices
G1)-2)))"

```

definition *inv_translation* where

```
"inv_translation r ≡ {0.. $\text{card } r$ } = r ∧ {0,1} ⊆ r"
```

lemma *inv_translationI4*[intro]:

```
assumes "finite r" "∧ x. x < card r ⇒ x ∈ r"
shows "r={0.. $\text{card } r$ }"
```

proof(insert *assms*,induct "card r" arbitrary:r)

```
case (Suc x r)
```

```
let ?r = "r - {x}"
```

```
from Suc have p:"x = card ?r" "finite ?r" by auto
```

```
have p2:"xa < card ?r ⇒ xa ∈ ?r" for xa
```

```
using Suc.prem(2)[of xa] Suc.hyps(2) unfolding p(1)[symmetric] by
```

```
auto
```

```
from Suc.hyps(1)[OF p p2] have "?r={0.. $\text{card } ?r$ "}.
```

```
with Suc.hyps(2) Suc.prem(1) show ?case
```

```
by (metis atLeast0_lessThan_Suc card_Diff_singleton_if insert_Diff
n_not_Suc_n p(1))
```

```
qed auto
```

lemma *inv_translationI*[intro!]:

```
assumes "finite r" "∧ x. x < card r ⇒ x ∈ r" "0 ∈ r" "Suc 0 ∈ r"
shows "inv_translation r"
```

proof -

```
from inv_translationI4[OF assms(1,2),symmetric]
```

```
have c:" {0.. $\text{card } r$ } = r " by auto
```

```
from assms(3,4) have "{0,1} ⊆ r" by auto
```

```
with c inv_translation_def show ?thesis by auto
```

```
qed
```

lemma *verts_in_translation_finite*[intro]:

```
"finite (vertices (translation X))"
```

```
"finite (edges (translation X))"
```

```
"0 ∈ vertices (translation X)"
```

```
"Suc 0 ∈ vertices (translation X)"
```

proof(atomize(full),induction X)

```
case (A_Int X1 X2)
```

```

    then show ?case by (auto simp:Let_def)
  next
    case (A_Cmp X1 X2)
    then show ?case by (auto simp:Let_def)
  next
    have [simp]: "{x::nat. x < 2} = {0,1}" by auto
    case (A_Cnv X)
    then show ?case by auto
qed auto

lemma inv_tr_card_min:
  assumes "inv_translation r"
  shows "card r ≥ 2"
proof -
  note [simp] = inv_translation_def
  have "{0..<x} = r ⇒ 2 ≤ x ↔ 0 ∈ r ∧ 1 ∈ r" for x by auto
  thus ge2:"card r ≥ 2" using assms by auto
qed

lemma verts_in_translation[intro]:
  "inv_translation (vertices (translation X))"
proof(induct X)
  { fix r
    assume assms:"inv_translation r"
    note [simp] = inv_translation_def
    from assms have a1:"finite r"
      by (intro card_ge_0_finite) auto
    have [simp]: "{0..<Suc x} = {0..<x} ∪ {x}" for x by auto
    note ge2 = inv_tr_card_min[OF assms]
    from ge2 assms have r0:"r ∩ {0} = {0}" "r ∩ {x. x < 2} = {0,1}"
  by auto
  have [intro!]: "∧x. x ∈ r ⇒ x < card r"
  and g6:"∧x. x < card r ↔ x ∈ r"
  using assms[unfolded inv_translation_def] atLeastLessThan_iff by
blast+
  have g4:"r ∩ {x. ¬ x < 2} = {2..<card r}"
  "r ∩ (Collect ((<) 0)) = {1..<card r}" using assms by fastforce+
  have ins:"1 ∈ r" "0 ∈ r" using assms by auto
  have d:"Suc (Suc (card r - 2)) = card r"
  using ge2 One_nat_def Suc_diff_Suc Suc_pred
  numeral_2_eq_2 by presburger
  note ge2 ins g4 g6 r0 d
} note inv_translationD[simp] = this
{
  fix a b c
  assume assm:"b ≤ (a::nat)"
  have "(λx. x + a - b) ‘ {b..<c} = {a..<c+a-b}" (is "?lhs = ?rhs")
proof -
  from assm have "?lhs = (λx. x + (a - b)) ‘ {b..<c}" by auto

```

```

    also have "... = ?rhs"
      unfolding linordered_semidom_class.image_add_atLeastLessThan'
using assm by auto
  finally show ?thesis by auto
qed
} note e[simp] = this
{ fix r z
  assume a1: "inv_translation z" and a2: "inv_translation r"
  let ?z2 = "card z + card r - 2"
  let ?z1 = "card z + card r - Suc 0"
  from a1 a2
  have le1: "Suc 0 ≤ card r"
    by (metis Suc_leD inv_translationD(1) numerals(2))
  hence le2: "card r ≤ ?z1"
    by (metis Suc_leD a1 inv_translationD(1) numerals(2) ordered_cancel_comm_monoid_diff_
with le1 have b: "{card r ..< ?z1} ∪ {Suc 0 ..< card r} = {Suc 0 ..<
?z1}"
  by auto
  have a: "(insert (card r) {0..<card z + card r - Suc 0}) = {0..<card
z + card r - Suc 0}"
  using le1 le2 a1 a2
  by (metis Suc_leD add_Suc_right atLeastLessThan_iff diff_Suc_Suc
insert_absorb inv_translationD(1) linorder_not_less not_less_eq_eq numerals(2)
ordered_cancel_comm_monoid_diff_class.le_add_diff)
  from a1 a2
  have "card z + card r - 2 ≥ card (r::nat set)"
    by (simp add: ordered_cancel_comm_monoid_diff_class.le_add_diff)
  with a2
  have c: "card (r ∪ {card r..<?z2}) = ?z2"
    by (metis atLeast0LessThan card_atLeastLessThan diff_zero inv_translation_def
ivl_disj_un_one(2))+
  note a b c
} note [simp] = this
have [simp]: "a < x ⇒ insert a {Suc a..<x} = {a..<x}" for a x by auto
{ case (A_Int X1 X2)
  let ?v1 = "vertices (translation X1)"
  from A_Int have [simp]: "(insert 0 (insert (Suc 0) (?v1 ∪ x))) = ?v1
∪ x"
  for x unfolding inv_translation_def by auto
  from A_Int show ?case by (auto simp: Let_def linorder_not_le)
next
case (A_Cmp X1 X2)
  hence "2 ≤ card (vertices (translation X1))" "2 ≤ card (vertices (translation
X2))" by auto
  hence "1 ≤ card (vertices (translation X1))" "1 ≤ card (vertices (translation
X2))"
  "1 < card (vertices (translation X1)) + card (vertices (translation
X2)) - 1"
  by auto

```

```

    from this A_Cmp
    show ?case by (auto simp:Let_def)
next
  case (A_Cnv X)
  thus ?case by (auto simp:Let_def)
}
qed auto

lemma translation_graph[intro]:
  "graph (translation X)"
  by (induct X, auto simp:Let_def)

lemma graph_rule_translation[intro]:
  "graph_rule (translation X, translation (A_Int X Y))"
  using verts_in_translation_finite[of X] verts_in_translation_finite[of
  "A_Int X Y"]
  translation_graph[of X] translation_graph[of "A_Int X Y"]
  by (auto simp:Let_def subgraph_def2)

lemma graph_hom_translation[intro]:
  "graph_homomorphism (LG {} {0,1}) (translation X) (Id_on {0,1})"
  using verts_in_translation[of X]
  unfolding inv_translation_def graph_homomorphism_def2 by auto

lemma translation_right_to_left:
  assumes f:"graph_homomorphism (translation e) G f" "(0, x) ∈ f" "(1,
  y) ∈ f"
  shows "(x, y) ∈ :G:[e]"
  using f
proof(induct e arbitrary:f x y)
case (A_Int e1 e2 f x y)
  let ?f1 = "id"
  let ?f2 = "(λ x. if x < 2 then x else x + card (vertices (translation
  e1)) - 2)"
  let ?G1 = "translation e1"
  let ?G2 = "translation e2"
  have f1:"(0, x) ∈ on_graph ?G1 ?f1 0 f" "(1, y) ∈ on_graph ?G1 ?f1
  0 f"
  and f2:"(0, x) ∈ on_graph ?G2 ?f2 0 f" "(1, y) ∈ on_graph ?G2 ?f2
  0 f"
  using A_Int.prem(2,3) by (auto simp:BNF_Def.Gr_def relcomp_def)
  from A_Int.prem(1)
  have uni:"graph_homomorphism (graph_union ?G1 (map_graph_fn ?G2 ?f2))
  G f"
  by (auto simp:Let_def)
  from graph_homo_union_id(1)[OF uni translation_graph]
  have h1:"graph_homomorphism ?G1 (translation (A_Int e1 e2)) (on_graph
  ?G1 id)"
  by (auto simp:Let_def graph_homomorphism_def)

```

```

have "graph (map_graph_fn ?G2 ?f2)" by auto
from graph_homo_union_id(2)[OF uni this]
have h2:"graph_homomorphism ?G2 (translation (A_Int e1 e2)) (on_graph
?G2 ?f2)"
  by (auto simp:Let_def graph_homomorphism_def)
from A_Int.hyps(1)[OF graph_homomorphism_composes[OF h1 A_Int.prem(1)]]
f1]
  A_Int.hyps(2)[OF graph_homomorphism_composes[OF h2 A_Int.prem(1)]]
f2]
show ?case by auto
next
case (A_Cmp e1 e2 f x y)
let ?f1 = "(λ x. if x=0 then 0 else x+card(vertices (translation e2))-1)"
let ?f2 = "(λ x. if x=0 then card (vertices (translation e2)) else
x)"
let ?G1 = "translation e1"
let ?G2 = "translation e2"
let ?v = "card (vertices (translation e2))"
from A_Cmp.prem(1) have "?v ∈ Domain f" by (auto simp:Let_def graph_homomorphism_def)
then obtain v where v:"(?v,v) ∈ f" by auto
have f1:"(0, x) ∈ on_graph ?G1 ?f1 0 f" "(1, v) ∈ on_graph ?G1 ?f1
0 f"
  and f2:"(0, v) ∈ on_graph ?G2 ?f2 0 f" "(1, y) ∈ on_graph ?G2 ?f2
0 f"
  using A_Cmp.prem(2,3) v by auto
from A_Cmp.prem(1)
have uni:"graph_homomorphism (graph_union (map_graph_fn ?G1 ?f1) (map_graph_fn
?G2 ?f2)) G f"
  by (auto simp:Let_def)
have "graph (map_graph_fn ?G1 ?f1)" by auto
from graph_homo_union_id(1)[OF uni this]
have h1:"graph_homomorphism ?G1 (translation (A_Cmp e1 e2)) (on_graph
?G1 ?f1)"
  by (auto simp:Let_def graph_homomorphism_def2)
have "graph (map_graph_fn ?G2 ?f2)" by auto
from graph_homo_union_id(2)[OF uni this]
have h2:"graph_homomorphism ?G2 (translation (A_Cmp e1 e2)) (on_graph
?G2 ?f2)"
  by (auto simp:Let_def graph_homomorphism_def2)
from A_Cmp.hyps(1)[OF graph_homomorphism_composes[OF h1 A_Cmp.prem(1)]]
f1]
  A_Cmp.hyps(2)[OF graph_homomorphism_composes[OF h2 A_Cmp.prem(1)]]
f2]
show ?case by auto
next
case (A_Cnv e f x y)
let ?f = "(λ x. if x < 2 then 1 - x else x)"
let ?G = "translation e"
have i:"graph_homomorphism ?G (map_graph_fn ?G ?f) (on_graph ?G ?f)"

```

```

using A_Cnv by auto
  have "(0, y) ∈ on_graph ?G ?f 0 f" "(1, x) ∈ on_graph ?G ?f 0 f"
    using A_Cnv.prem3(3,2) by (auto simp:BNF_Def.Gr_def relcomp_def)
  from A_Cnv.hyps(1)[OF graph_homomorphism_composes[OF i] this] A_Cnv.prem3(1)
  show ?case by auto
next
case (A_Lbl 1 f x y)
  hence "edge_preserving f {(1,0,1)} (edges G)" unfolding graph_homomorphism_def
  by auto
  with A_Lbl(2,3) show ?case by (auto simp:getRel_def edge_preserving_def)
qed

lemma translation_homomorphism:
  assumes "graph_homomorphism (translation e) G f"
  shows "f `` {0} × f `` {1} ⊆ :G:[e]" ":G:[e] ≠ {}"
  using translation_right_to_left[OF assms] assms[unfolded graph_homomorphism_def2]
  verts_in_translation_finite[of e] by auto

  Lemma 5.

lemma translation:
  assumes "graph G"
  shows "(x, y) ∈ :G:[e] ↔ (∃ f. graph_homomorphism (translation e)
  G f ∧ (0,x) ∈ f ∧ (1,y) ∈ f)"
  (is "?lhs = ?rhs")
proof
  have [dest!]: "y + card (vertices (translation (e::'a allegorical_term)))
  - 2 < 2 ⇒ (y::nat) < 2"
  for y e using inv_tr_card_min[OF verts_in_translation,of e] by linarith
  {
    fix y fix e::"'a allegorical_term"
    assume "y + card (vertices (translation e)) - 2 ∈ vertices (translation
  e)"
    hence "y + card (vertices (translation e)) - 2 < card (vertices (translation
  e))"
    using verts_in_translation[of e,unfolded inv_translation_def] by
  auto
    hence "y < 2" using inv_tr_card_min[OF verts_in_translation,of e]
  by auto
  }
  note [dest!] = this
  {
    fix y fix e::"'a allegorical_term"
    assume "y + card (vertices (translation e)) - Suc 0 ∈ vertices (translation
  e)"
    hence "y + card (vertices (translation e)) - Suc 0 ∈ {0..<card (vertices
  (translation e))}"
    using verts_in_translation[of e,unfolded inv_translation_def] by
  simp
    hence "y = 0" using inv_tr_card_min[OF verts_in_translation,of e]
  by auto
  }
  note [dest!] = this
  {
    fix y fix e::"'a allegorical_term"

```

```

    assume "card (vertices (translation e)) ∈ vertices (translation
e)"
    hence "card (vertices (translation e)) ∈ {0..<card (vertices (translation
e))}"
    using verts_in_translation[of e,unfolded inv_translation_def] by
auto
    hence "False" by auto
  } note [dest!] = this
{
  fix y fix e::"'a allegorical_term"
  assume "y + card (vertices (translation e)) ≤ Suc 0"
  hence " card (vertices (translation e)) ≤ Suc 0" by auto
  hence "False" using inv_tr_card_min[OF verts_in_translation[of e]]
by auto
  } note [dest!] = this
assume ?lhs
then show ?rhs
proof(induct e arbitrary:x y)
  case (A_Int e1 e2)
  from A_Int have assm:"(x, y) ∈ :G:[e1]" "(x, y) ∈ :G:[e2]" by auto
  from A_Int(1)[OF assm(1)] obtain f1 where
    f1:"graph_homomorphism (translation e1) G f1" "(0, x) ∈ f1" "(1,
y) ∈ f1" by auto
  from A_Int(2)[OF assm(2)] obtain f2 where
    f2:"graph_homomorphism (translation e2) G f2" "(0, x) ∈ f2" "(1,
y) ∈ f2" by auto
  from f1 f2 have v:"Domain f1 = vertices (translation e1)" "Domain
f2 = vertices (translation e2)"
  unfolding graph_homomorphism_def by auto
  let ?f2 = "(λ x. if x < 2 then x else x + card (vertices (translation
e1)) - 2)"
  let ?tr2 = "on_graph (translation e2) ?f2"
  have inj2:"inj_on ?f2 (vertices (translation e2))" unfolding inj_on_def
by auto
  have "(0,0) ∈ ?tr2-1" "(1,1) ∈ ?tr2-1" by auto
  from this[THEN relcompI] f2(2,3)
  have zero_one:"(0,x) ∈ ?tr2-1 0 f2"
    "(1,y) ∈ ?tr2-1 0 f2" by auto
  {
    fix yb zb
    assume "(yb + card (vertices (translation e1)) - 2, zb) ∈ f1"
    hence "yb + card (vertices (translation e1)) - 2 ∈ vertices (translation
e1)" using v by auto
  }
  note in_f[dest!] = this
  have d_a:"Domain f1 ∩ Domain (?tr2-1 0 f2) = {0,1}"
    using zero_one by (auto simp:v)
  have d_b:"Domain (f1 ∩ ?tr2-1 0 f2) = {0,1}"
    using zero_one f1(2,3) by auto
  note cmp2 = graph_homomorphism_composes[OF graph_homo_inv[OF translation_graph
inj2] f2(1)]
  have "graph_homomorphism (translation (A_Int e1 e2)) G (f1 ∪ ?tr2-1

```

```

0 f₂)"
  using graph_homo_union[OF f₁(1) cmp2 d_a[folded d_b]]
  by (auto simp:Let_def)
  thus ?case using zero_one[THEN UnI2[of _ _ "f₁"]] by blast
next
  case (A_Cmp e₁ e₂)
  from A_Cmp obtain z where assm:"(x, z) ∈ :G:[e₁]" "(z, y) ∈ :G:[e₂]"
by auto
  from A_Cmp(1)[OF assm(1)] obtain f₁ where
    f₁:"graph_homomorphism (translation e₁) G f₁" "(0, x) ∈ f₁" "(1,
z) ∈ f₁" by auto
  from A_Cmp(2)[OF assm(2)] obtain f₂ where
    f₂:"graph_homomorphism (translation e₂) G f₂" "(0, z) ∈ f₂" "(1,
y) ∈ f₂" by auto
  from f₁ f₂ have v:"Domain f₁ = vertices (translation e₁)" "Domain
f₂ = vertices (translation e₂)"
  unfolding graph_homomorphism_def by auto
  let ?f₁ = "(λ x. if x=0 then 0 else x+card(vertices (translation e₂))-1)"
  let ?f₂ = "(λ x. if x=0 then card (vertices (translation e₂)) else
x)"
  let ?tr₁ = "on_graph (translation e₁) ?f₁"
  let ?tr₂ = "on_graph (translation e₂) ?f₂"
  have inj1:"inj_on ?f₁ (vertices (translation e₁))" unfolding inj_on_def
by auto
  have inj2:"inj_on ?f₂ (vertices (translation e₂))" unfolding inj_on_def
by auto
  have "(card (vertices (translation e₂)),0) ∈ ?tr₂⁻¹" "(1,1) ∈ ?tr₂⁻¹"
"(0,0) ∈ ?tr₁⁻¹" "(card (vertices (translation e₂)),1) ∈ ?tr₁⁻¹"
by auto
  from this[THEN relcompI] f₂(2,3) f₁(2,3)
  have zero_one:"(card (vertices (translation e₂)),z) ∈ ?tr₁⁻¹ 0 f₁"
"(0,x) ∈ ?tr₁⁻¹ 0 f₁"
"(card (vertices (translation e₂)),z) ∈ ?tr₂⁻¹ 0 f₂"
"(1,y) ∈ ?tr₂⁻¹ 0 f₂" by auto
  have [simp]:
    "ye ∈ vertices (translation e₂) ⇒
(if ye = 0 then card (vertices (translation e₂)) else ye) =
(if yd = 0 then 0 else yd + card (vertices (translation e₂)) -
1) ↔ ye = 0 ∧ yd = 1"
  for ye yd using v inv_tr_card_min[OF verts_in_translation,of "e₂"]
  by(cases "ye=0";cases "yd=0";auto)
  have d_a:"Domain (?tr₁⁻¹ 0 f₁) ∩ Domain (?tr₂⁻¹ 0 f₂) = {card (vertices
(translation e₂))}"
  using zero_one using [[simproc del: defined_all]] by (auto simp:
v)
  have d_b:"Domain (?tr₁⁻¹ 0 f₁ ∩ ?tr₂⁻¹ 0 f₂) = {card (vertices (translation
e₂))}"
  using zero_one f₁(2,3) using [[simproc del: defined_all]] by auto
  note cmp1 = graph_homomorphism_composes[OF graph_homo_inv[OF translation_graph

```

```

inj1] f1(1)]
  note cmp2 = graph_homomorphism_composes[OF graph_homo_inv[OF translation_graph
inj2] f2(1)]
  have "graph_homomorphism (translation (A_Cmp e1 e2)) G (?tr1-1 0
f1 ∪ ?tr2-1 0 f2)"
    unfolding Let_def translation.simps
    by (rule graph_homo_union[OF cmp1 cmp2 d_a[folded d_b]])
  thus ?case using zero_one by blast
next
  case (A_Cnv e)
  let ?G = "translation (A_Cnv e)"
  from A_Cnv obtain f where
    f:"graph_homomorphism (translation e) G f" "(0, y) ∈ f" "(1, x)
∈ f" by auto
  hence v:"Domain f = vertices (translation e)"
    unfolding graph_homomorphism_def by auto
  define n where "n ≡ card (vertices (translation e))"
  from verts_in_translation f inv_tr_card_min[OF verts_in_translation]
v(1)
  have n:"vertices (translation e) = {0..

```

```

    using assms A_Lbl xy unfolding graph_homomorphism_def2
    by (auto simp:univalent_def getRel_def on_triple_def Image_def graph_union_def
insert_absorb)
    then show ?case by auto
qed
qed (insert translation_right_to_left,auto)

abbreviation transl_rule ::
  "'a sentence  $\Rightarrow$  ('a, nat) labeled_graph  $\times$  ('a, nat) labeled_graph"
where
  "transl_rule R  $\equiv$  (translation (fst R),translation (snd R))"

  Lemma 6.

lemma maintained_holds_iff:
  assumes "graph G"
  shows "maintained (translation eL,translation (A_Int eL eR)) G  $\longleftrightarrow$ 
G  $\models$  eL  $\sqsubseteq$  eR" (is "?rhs = ?lhs")
proof
  assume lhs: ?lhs
  show ?rhs unfolding maintained_def proof(clarify) fix f
    assume f:"graph_homomorphism (fst (translation eL, translation (A_Int
eL eR))) G f"
    then obtain x y where f2:"(0,x)  $\in$  f" "(1,y)  $\in$  f" unfolding graph_homomorphism_def
      by (metis DomainE One_nat_def prod.sel(1) verts_in_translation_finite(3,4))
    with f have "(x,y)  $\in$  :G:[fst (eL  $\sqsubseteq$  eR)]" unfolding translation[OF
assms] by auto
    with lhs have "(x,y)  $\in$  :G:[snd (eL  $\sqsubseteq$  eR)]" by auto
    then obtain g where g: "graph_homomorphism (translation (A_Int eL
eR)) G g"
      and g2: "(0, x)  $\in$  g" "(1, y)  $\in$  g" unfolding translation[OF
assms] by auto
    have v:"vertices (translation (A_Int eL eR)) = Domain g"
      "vertices (translation eL) = Domain f" using f g
      unfolding graph_homomorphism_def by auto
    from subgraph_subset[of "translation eL" "translation (A_Int eL eR)"]
      graph_rule_translation[of eL eR]
    have dom_sub: "Domain f  $\subseteq$  Domain g"
      using v unfolding prod.sel by argo
    hence dom_le:"card (Domain f)  $\leq$  card (Domain g)"
      by (metis card.infinite card_mono inv_tr_card_min not_less rel_simps(51)
v(1) verts_in_translation)
    have c_f:"card (Domain f)  $\geq$  2" using inv_tr_card_min[OF verts_in_translation]
v by metis
    from f[unfolded graph_homomorphism_def]
    have ep_f:"edge_preserving f (edges (translation eL)) (edges G)"
      and uni_f:"univalent f" by auto
    let ?f = " $(\lambda x. \text{if } x < 2 \text{ then } x \text{ else } x + \text{card (vertices (translation
eL))} - 2)$ "
    define GR where "GR = map_graph_fn (translation eR) ?f"

```

```

from g[unfolded graph_homomorphism_def]
have "edge_preserving g (edges (translation (A_Int e_L e_R))) (edges
G)"
  and uni_g:"univalent g" by auto
from edge_preserving_subset[OF subset_refl _ this(1)]
have ep_g:"edge_preserving g (edges GR) (edges G)" by (auto simp:Let_def
GR_def)
  { fix a assume a:"a ∈ vertices (translation e_R)"
    hence "?f a ∈ vertices (translation (A_Int e_L e_R))" by (auto simp:Let_def)
    from this[unfolded v] verts_in_translation[of "A_Int e_L e_R",unfolded
inv_translation_def v]
    have "¬ a < 2 ⇒ a + card (Domain f) - 2 < card (Domain g)" by
auto
    } note[intro!] = this
  have [intro!]: "¬ aa < 2 ⇒ card (Domain f) ≤ aa + card (Domain
f) - 2" for aa by simp
  from v(2) restrictD[OF translation_graph[of e_L]]
  have df[dest]:"xa ∉ Domain f ⇒ (1,xa,xb) ∈ edges (translation e_L)
⇒ False"
    "xa ∉ Domain f ⇒ (1,xb,xa) ∈ edges (translation e_L)
⇒ False"
    for xa 1 xb unfolding edge_preserving by auto
  { fix 1 xa xb ya
    assume assm: "(1,xa,xb) ∈ edges GR"
    with c_f dom_le
    have "xa ∈ {0,1} ∪ {card (Domain f)..<card (Domain g)}"
      "xb ∈ {0,1} ∪ {card (Domain f)..<card (Domain g)}"
      unfolding GR_def v by auto
    hence minb:"xa ∈ {0,1} ∨ xa ≥ card (Domain f)" "xb ∈ {0,1} ∨
xb ≥ card (Domain f)"
      by auto
    { fix z xa assume minb:"xa ∈ {0,1} ∨ xa ≥ card (Domain f)" and
z:"(xa,z) ∈ f"
      from z verts_in_translation[of e_L,unfolded inv_translation_def
v]
      have "xa < card(Domain f)" by auto
      with minb verts_in_translation[of "A_Int e_L e_R",unfolded inv_translation_def
v]
      have x:"xa ∈ {0,1} ∧ xa ∈ Domain g" by auto
      then obtain v where g:"(xa,v) ∈ g" by auto
      consider "xa = 0 ∧ z = x" | "xa = 1 ∧ z = y"
        using x f2[THEN univalentD[OF uni_f]] z by auto
      hence "v = z" using g g2[THEN univalentD[OF uni_g]] by metis
      hence "(xa,z) ∈ g" using g by auto
    }
  }
  note minb[THEN this]
}
with f2 g2[THEN univalentD[OF uni_g]]
have dg:"(1,xa,xb) ∈ edges GR ⇒ (xa,ya) ∈ f ⇒ (xa,ya) ∈ g"

```

```

      "(1,xb,xa) ∈ edges GR ⇒ (xa,ya) ∈ f ⇒ (xa,ya) ∈ g"
      for xa l xb ya
      unfolding edge_preserving by (auto)
      have "vertices (translation eL) ⊆ vertices (translation (A_Int eL
eR))"
      by(rule subgraph_subset,insert graph_rule_translation,auto)
      hence subdom:"Domain f ⊆ Domain g" unfolding v.
      let ?g = "f ∪ (Id_on (UNIV - Domain f) 0 g)"
      have [simp]:"Domain ?g = Domain g" using subdom unfolding Domain_Un_eq
by auto
      have ih:"graph_homomorphism (translation (A_Int eL eR)) G ?g"
      proof(rule graph_homomorphismI)
        show "?g '' vertices (translation (A_Int eL eR)) ⊆ vertices G"
          using g[unfolded graph_homomorphism_def] f[unfolded graph_homomorphism_def]
          by (auto simp: v simp del:translation.simps)
        show "edge_preserving ?g (edges (translation (A_Int eL eR))) (edges
G)"
          unfolding Let_def translation.simps graph_union_edges proof
          show "edge_preserving ?g (edges (translation eL)) (edges G)"
            using edge_preserving_atomic[OF ep_f] unfolding edge_preserving
by auto
          have "edge_preserving ?g (edges GR) (edges G)"
            using edge_preserving_atomic[OF ep_g] dg unfolding edge_preserving
by (auto;blast)
          thus "edge_preserving ?g (edges (map_graph_fn (translation eR)
?f)) (edges G)"
            by (auto simp:GR_def)
          qed
        qed (insert f[unfolded graph_homomorphism_def] g[unfolded graph_homomorphism_def],auto
simp:Let_def)
        have ie:"agree_on (translation eL) f ?g" unfolding agree_on_def by
(auto simp:v)
        from ie ih show "extensible (translation eL, translation (A_Int
eL eR)) G f"
          unfolding extensible_def prod.sel by auto
        qed next
        assume rhs:?rhs
        { fix x y assume "(x,y) ∈ :G:[eL]"
          with translation[OF assms] obtain f
            where f:"graph_homomorphism (fst (translation eL, translation (A_Int
eL eR))) G f"
            "(0, x) ∈ f" "(1, y) ∈ f" by auto
          with rhs[unfolded maintained_def,rule_format,OF f(1),unfolded extensible_def]
          obtain g where g:"graph_homomorphism (translation (A_Int eL eR))
G g"
            "agree_on (translation eL) f g" by auto
          hence "(x,y) ∈ :G:[A_Int eL eR]" using f unfolding agree_on_def translation[OF
assms] by auto
        }

```

thus ?lhs by auto
qed

```

lemma translation_self[intro]:
  "(0, 1) ∈ :translation e:[e]"
proof(induct e)
  case (A_Int e1 e2)
  let ?f = "(λx. if x < 2 then x else x + card (vertices (translation
e1)) - 2)"
  have f: "(?f 0,?f 1) ∈:map_graph_fn (translation e2) ?f:[e2]"
    using map_graph_in[OF translation_graph A_Int(2),of ?f] by auto
  let ?G = "graph_union (translation e1) (map_graph_fn (translation e2)
?f)"
  have "{(0,1)} ⊆ :(translation e1):[e1]" using A_Int by auto
  moreover have "{(0,1)} ⊆ :map_graph_fn (translation e2) ?f:[e2]" us-
ing f by auto
  moreover have ":map_graph_fn (translation e2) ?f:[e2] ⊆ :?G:[e2]" ":translation
e1:[e1] ⊆ :?G:[e1]"
    using graph_union_semantics by blast+
  ultimately show ?case by (auto simp:Let_def)
next
  case (A_Cmp e1 e2)
  let ?f1 = "λx. if x = 0 then 0 else x + card (vertices (translation
e2)) - 1"
  have f1: "(?f1 0,?f1 1) ∈:map_graph_fn (translation e1) ?f1:[e1]"
    using map_graph_in[OF translation_graph A_Cmp(1),of ?f1] by auto
  let ?f2 = "λx. if x = 0 then card (vertices (translation e2)) else x"
  have f2: "(?f2 0,?f2 1) ∈:map_graph_fn (translation e2) ?f2:[e2]"
    using map_graph_in[OF translation_graph A_Cmp(2),of ?f2] by auto
  let ?G = "graph_union (map_graph_fn (translation e1) ?f1) (map_graph_fn
(translation e2) ?f2)"
  have "{(0,1)} = {(0,card (vertices (translation e2)))} 0 {(card (vertices
(translation e2)),1)}"
    by auto
  also have "{(0,card (vertices (translation e2)))} ⊆ :map_graph_fn (translation
e1) ?f1:[e1]"
    using f1 by auto
  also have ":map_graph_fn (translation e1) ?f1:[e1] ⊆ :?G:[e1]"
    using graph_union_semantics by auto
  also have "{(card (vertices (translation e2)),1)} ⊆ :map_graph_fn (translation
e2) ?f2:[e2]"
    using f2 by auto
  also have ":map_graph_fn (translation e2) ?f2:[e2] ⊆ :?G:[e2]"
    using graph_union_semantics by blast
  also have "(:?G:[e1]) 0 (:?G:[e2]) = :translation (A_Cmp e1 e2):[A_Cmp
e1 e2]"
    by (auto simp:Let_def)
  finally show ?case by auto
next

```

```

    case (A_Cnv e)
    from map_graph_in[OF translation_graph this, of "(λx. if x < (2::nat)
then 1 - x else x)"]
    show ?case using map_graph_in[OF translation_graph] by auto
qed (simp add:getRel_def)

```

Lemma 6 is only used on rules of the form $e_L \sqsubseteq e_R$. The requirement of G being a graph can be dropped for one direction.

```

lemma maintained_holds[intro]:
  assumes ":G:[e_L] ⊆ :G:[e_R]"
  shows "maintained (transl_rule (e_L ⊆ e_R)) G"
proof (cases "graph G")
  case True
  thus ?thesis using assms sentence_iff maintained_holds_iff prod.sel
  by metis
next
  case False
  thus ?thesis by (auto simp:maintained_def graph_homomorphism_def)
qed

```

```

lemma maintained_holds_subset_iff[simp]:
  assumes "graph G"
  shows "maintained (transl_rule (e_L ⊆ e_R)) G ⟷ (:G:[e_L] ⊆ :G:[e_R])"
  using assms maintained_holds_iff sentence_iff prod.sel by metis

```

end

8 Standard Rules

We define the standard rules here, and prove the relation to standard rules. This means proving that the graph rules do what they say they do.

```

theory StandardRules
imports StandardModels RuleSemanticsConnection
begin

```

Definition 16 makes this remark. We don't have a specific version of Definition 16.

```

lemma conflict_free:
":G:[A_Lbl 1] = {} ⟷ (∀ (l',x,y)∈edges G. l' ≠ 1)"
  by (auto simp:getRel_def)

```

Definition 17, abstractly. It's unlikely that we wish to use the top rule for any symbol except top, but stating it abstractly makes it consistent with the other rules.

```

definition top_rule :: "'1 ⇒ ('1,nat) Graph_PreRule" where
"top_rule t = (LG {} {0,1}, LG {(t,0,1)} {0,1})"

```

Proof that definition 17 does what it says it does.

```

lemma top_rule[simp]:
  assumes "graph G"
  shows "maintained (top_rule r) G  $\longleftrightarrow$  vertices G  $\times$  vertices G = getRel
r G"
proof
  assume a:"maintained (top_rule r) G"
  { fix a b assume "a  $\in$  vertices G" "b  $\in$  vertices G"
    hence "graph_homomorphism (LG {} {0, 1}) G {(0::nat,a),(1,b)}"
      using assms unfolding graph_homomorphism_def univalent_def by auto
    with a[unfolded maintained_def top_rule_def] extensible_refl_concr
    have "graph_homomorphism (LG {(r, 0, 1)} {0::nat, 1}) G {(0::nat,
a), (1, b)}" by simp
    hence "(a, b)  $\in$  getRel r G"
      unfolding graph_homomorphism_def2 graph_union_iff getRel_def by
auto
  }
  thus "vertices G  $\times$  vertices G = getRel r G" using getRel_dom[OF assms]
by auto next
  assume a:"vertices G  $\times$  vertices G = getRel r G"
  { fix f assume a2:"graph_homomorphism (fst (top_rule r)) G f"
    hence f:"f `` {0, 1}  $\subseteq$  vertices G" "on_triple f `` {}  $\subseteq$  edges G"
      "univalent f" "Domain f = {0, 1}"
      unfolding top_rule_def prod.sel graph_homomorphism_concr_graph[OF
assms graph_empty_e]
      by argo+
    from a2 have ih:"graph_homomorphism (LG {} {0, 1}) G f" unfolding
top_rule_def by auto
    have "extensible (top_rule r) G f" unfolding top_rule_def extensible_refl_concr[OF
ih]
      graph_homomorphism_concr_graph[OF assms graph_single]
      using f a[unfolded getRel_def] by fastforce
  }
  thus "maintained (top_rule r) G" unfolding maintained_def by auto
qed

```

Definition 18.

```

definition nonempty_rule :: "('1,nat) Graph_PreRule" where
"nonempty_rule = (LG {} {},LG {} {0})"

```

Proof that definition 18 does what it says it does.

```

lemma nonempty_rule[simp]:
  assumes "graph G"
  shows "maintained nonempty_rule G  $\longleftrightarrow$  vertices G  $\neq$  {}"
proof -
  have "vertices G = {}  $\implies$  graph_homomorphism (LG {} {0}) G x  $\implies$  False"
    "v  $\in$  vertices G  $\implies$  graph_homomorphism (LG {} {0}) G {(0,v)}"
    for v::"'b" and x::"(nat  $\times$  'b) set"
    unfolding graph_homomorphism_concr_graph[OF assms graph_empty_e] univalent_def
by blast+

```

thus ?thesis unfolding nonempty_rule_def maintained_def extensible_def
 by (auto intro:assms)
 qed

Definition 19.

```

definition reflexivity_rule :: "'l  $\Rightarrow$  ('l,nat) Graph_PreRule" where
  "reflexivity_rule t = (LG {} {0},LG {(t,0,0)} {0})"
definition symmetry_rule :: "'l  $\Rightarrow$  ('l,nat) Graph_PreRule" where
  "symmetry_rule t = (transl_rule (A_Cnv (A_Lbl t)  $\sqsubseteq$  A_Lbl t))"
definition transitive_rule :: "'l  $\Rightarrow$  ('l,nat) Graph_PreRule" where
  "transitive_rule t = (transl_rule (A_Cmp (A_Lbl t) (A_Lbl t)  $\sqsubseteq$  A_Lbl t))"
definition congruence_rule :: "'l  $\Rightarrow$  'l  $\Rightarrow$  ('l,nat) Graph_PreRule" where
  "congruence_rule t l = (transl_rule (A_Cmp (A_Cmp (A_Lbl t) (A_Lbl l))
  (A_Lbl t)  $\sqsubseteq$  A_Lbl l))"
abbreviation congruence_rules :: "'l  $\Rightarrow$  'l set  $\Rightarrow$  ('l,nat) Graph_PreRule
  set"
  where
  "congruence_rules t L  $\equiv$  congruence_rule t ' L"

lemma are_rules[intro]:
  "graph_rule nonempty_rule"
  "graph_rule (top_rule t)"
  "graph_rule (reflexivity_rule i)"
  unfolding reflexivity_rule_def top_rule_def nonempty_rule_def graph_homomorphism_def

```

by auto

Just before Lemma 7, we remark that if I is an identity, it maintains the identity rules.

```

lemma ident_rel_refl:
  assumes "graph G" "ident_rel idt G"
  shows "maintained (reflexivity_rule idt) G"
  unfolding reflexivity_rule_def
proof(rule maintainedI) fix f
  assume "graph_homomorphism (LG {} {0::nat}) G f"
  hence f:"Domain f = {0}" "graph G" "f ' ' {0}  $\subseteq$  vertices G" "univalent
  f"
  unfolding graph_homomorphism_def by force+
  from assms(2) univalentD[OF f(4)] f(3)
  have "edge_preserving f {(idt, 0, 0)} (edges G)" unfolding edge_preserving
  by (auto simp:getRel_def set_eq_iff image_def)
  with f have "graph_homomorphism (LG {(idt, 0, 0)} {0}) G f"
  "agree_on (LG {} {0}) f f" using assms
  unfolding graph_homomorphism_def labeled_graph.sel agree_on_def univalent_def
  by auto
  then show "extensible (LG {} {0}, LG {(idt, 0, 0)} {0}) G f"
  unfolding extensible_def prod.sel by auto
qed

```

lemma

```
assumes "ident_rel idt G"
shows ident_rel_trans:"maintained (transitive_rule idt) G"
      and ident_rel_symm :"maintained (symmetry_rule idt) G"
      and ident_rel_cong :"maintained (congruence_rule idt l) G"
unfolding transitive_rule_def symmetry_rule_def congruence_rule_def
by(intro maintained_holds,insert assms,force)+
```

Definition 19.

definition identity_rules ::

```
"'a Standard_Constant set  $\Rightarrow$  (('a Standard_Constant, nat) Graph_PreRule)
set" where
"identity_rules L  $\equiv$  {reflexivity_rule S_Idt,transitive_rule S_Idt,symmetry_rule
S_Idt}
       $\cup$  congruence_rules S_Idt L"
```

lemma identity_rules_graph_rule:

```
assumes "x  $\in$  identity_rules L"
shows "graph_rule x"
proof -
  from graph_rule_translation
  have gr:" $\wedge$  u v . graph_rule (transl_rule (u  $\sqsubseteq$  v))" by auto
  consider "x = reflexivity_rule S_Idt" | "x = transitive_rule S_Idt"
| "x = symmetry_rule S_Idt"
  | " $\exists$  v w. x = congruence_rule v w" using assms unfolding identity_rules_def
Un_iff by blast
  thus ?thesis using gr are_rules(3)
  unfolding congruence_rule_def transitive_rule_def symmetry_rule_def
  by cases fast+
qed
```

Definition 19, showing that the properties indeed do what they claim to do.

lemma

```
assumes g[intro]:"graph (G :: ('a, 'b) labeled_graph)"
shows reflexivity_rule: "maintained (reflexivity_rule l) G  $\implies$  refl_on
(vertices G) (getRel l G)"
      and transitive_rule: "maintained (transitive_rule l) G  $\implies$  trans
(getRel l G)"
      and symmetry_rule: "maintained (symmetry_rule l) G  $\implies$  sym (getRel
l G)"
proof -
  { from assms have gr:"getRel l G  $\subseteq$  vertices G  $\times$  vertices G" by (auto
simp:getRel_def)
  assume m:"maintained (reflexivity_rule l) G" (is "maintained ?r G")
  note [simp] = reflexivity_rule_def
  show r:"refl_on (vertices G) (getRel l G)"
  proof(rule refl_onI) fix x
    assume assm:"x  $\in$  vertices G" define f where "f = {(0::nat,x)}"
```

```

    have "graph_homomorphism (fst ?r) G f" using assm
      by (auto simp:graph_homomorphism_def univalent_def f_def)
    from m[unfolded maintained_def] this
    obtain g: "(nat × 'b) set"
      where g: "graph_homomorphism (snd ?r) G g"
            "agree_on (fst ?r) f g"
      unfolding extensible_def by blast
    have "∧ n v. (n,v) ∈ g ⇒ (n = 0) ∧ (v = x)" using g unfolding
ing
      agree_on_def graph_homomorphism_def f_def by auto
    with g(2) have "g = {(0,x)}" unfolding agree_on_def f_def by auto
    with g(1) show "(x,x) ∈ getRel 1 G"
      unfolding graph_homomorphism_def edge_preserving getRel_def by
auto
    qed
  }
  { assume m: "maintained (transitive_rule 1) G"
    from m[unfolded maintained_holds_subset_iff[OF g] transitive_rule_def]
    show "trans (getRel 1 G)" unfolding trans_def by auto
  }
  { assume m: "maintained (symmetry_rule 1) G"
    from m[unfolded maintained_holds_subset_iff[OF g] symmetry_rule_def]
    show "sym (getRel 1 G)" unfolding sym_def by auto
  }
}
qed

lemma finite_identity_rules[intro]:
  assumes "finite L"
  shows "finite (identity_rules L)"
  using assms unfolding identity_rules_def by auto

lemma equivalence:
  assumes gr: "graph G" and m: "maintainedA {reflexivity_rule I, transitive_rule
I, symmetry_rule I} G"
  shows "equiv (vertices G) (getRel I G)"
proof(rule equivI)
  show "getRel I G ⊆ vertices G × vertices G"
    by (metis gr semantics.simps(4) semantics_subset_vertices)
  show "refl_on (vertices G) (getRel I G)" using m by(intro reflexivity_rule[OF
gr], auto)
  show "sym (getRel I G)" using m by(intro symmetry_rule[OF gr], auto)
  show "trans (getRel I G)" using m by(intro transitive_rule[OF gr], auto)
qed

lemma congruence_rule:

  assumes g: "graph G"
    and mA: "maintainedA {reflexivity_rule I, transitive_rule I, symmetry_rule
I} G"

```

```

    and m:"maintained (congruence_rule I l) G"
    shows "(λ v. getRel l G `` {v}) respects (getRel I G)" (is "?g1")
    and "(λ v. (getRel l G)-1 `` {v}) respects (getRel I G)" (is "?g2")
  proof -

    note eq = equivalence[OF g mA]
    { fix y z
      assume aI:"(y, z) ∈ getRel I G"
      hence a2:"(z, y) ∈ getRel I G" using eq[unfolded equiv_def sym_def]
    }
  by auto
    hence a3:"(z, z) ∈ getRel I G" "(y, y) ∈ getRel I G"
      using eq[unfolded equiv_def refl_on_def] by auto
    { fix x
      { assume a1:"(y, x) ∈ getRel l G"
        hence "x ∈ vertices G" using g unfolding getRel_def by auto
        hence r:"(x, x) ∈ getRel I G" using eq[unfolded equiv_def refl_on_def]
      }
    }
  by auto
    note relcompI[OF relcompI[OF a2 a1] r]
    } note yx = this
    { assume a1:"(z, x) ∈ getRel l G"
      hence "x ∈ vertices G" using g unfolding getRel_def by auto
      hence r:"(x, x) ∈ getRel I G" using eq[unfolded equiv_def refl_on_def]
    }
  by auto
    note relcompI[OF relcompI[OF aI a1] r]
    } note zx = this
    from zx yx m[unfolded maintained_holds_subset_iff[OF g] congruence_rule_def]
    have "(y, x) ∈ getRel l G ↔ (z, x) ∈ getRel l G" by auto
    } note v1 = this
    { fix x
      { assume a1:"(x, y) ∈ getRel l G"
        hence "x ∈ vertices G" using g unfolding getRel_def by auto
        hence r:"(x, x) ∈ getRel I G" using eq[unfolded equiv_def refl_on_def]
      }
    }
  by auto
    note relcompI[OF relcompI[OF r a1] aI]
    } note yx = this
    { assume a1:"(x, z) ∈ getRel l G"
      hence "x ∈ vertices G" using g unfolding getRel_def by auto
      hence r:"(x, x) ∈ getRel I G" using eq[unfolded equiv_def refl_on_def]
    }
  by auto
    note relcompI[OF relcompI[OF r a1] a2]
    } note zx = this
    from zx yx m[unfolded maintained_holds_subset_iff[OF g] congruence_rule_def]
    have "(x, y) ∈ getRel l G ↔ (x, z) ∈ getRel l G" by auto
    } note v2 = this
  from v1 v2
  have "getRel l G `` {y} = getRel l G `` {z}"
    "(getRel l G)-1 `` {y} = (getRel l G)-1 `` {z}" by auto
}
thus ?g1 ?g2 unfolding congruent_def by force+

```

qed

Lemma 7, strengthened with an extra property to make subsequent proofs easier to carry out.

lemma identity_rules:

```

assumes "graph G"
        "maintainedA (identity_rules L) G"
        "fst ` edges G  $\subseteq$  L"
shows " $\exists f. f \circ f = f$ "
       $\wedge$  ident_rel S_Idt (map_graph_fn G f)
       $\wedge$  subgraph (map_graph_fn G f) G
       $\wedge$  ( $\forall l\ x\ y. (l,x,y) \in \text{edges } G \iff (l,f\ x,f\ y) \in \text{edges } G$ )"

```

proof -

```

have ma:"maintainedA {reflexivity_rule S_Idt, transitive_rule S_Idt,
symmetry_rule S_Idt} G"
  using assms(2) by (auto simp:identity_rules_def)
note equiv = equivalence[OF assms(1) this]
{ fix l x y
  assume "(x, y)  $\in$  getRel l G" hence l:"l  $\in$  L" using assms(3) un-
folding getRel_def by auto
  have r1:"( $\lambda v. \text{getRel } l\ G\ \{\{v\}\}$ ) respects getRel S_Idt G"
    apply(intro congruence_rule[OF assms(1) ma])
    using assms(2) l unfolding identity_rules_def by auto
  have r2:"( $\lambda v. (\text{getRel } l\ G)^{-1}\ \{\{v\}\}$ ) respects getRel S_Idt G"
    apply(intro congruence_rule[OF assms(1) ma])
    using assms(2) l unfolding identity_rules_def by auto
  note congr = r1 r2
} note congr = this
define P where P:"P = ( $\lambda x\ y. y \in \text{getRel } S\_Idt\ G\ \{\{x\}\}$ )"
{ fix x
  assume a:"getRel S_Idt G  $\{\{x\}\} \neq \{\}$ "
  hence " $\exists y. P\ x\ y$ " unfolding P by auto
  hence p:"P x (Eps (P x))" unfolding some_eq_ex by auto
  { fix y
    assume b:"P x y"
    hence "(x,y)  $\in$  getRel S_Idt G" unfolding P by auto
    from equiv_class_eq[OF equiv this]
    have "getRel S_Idt G  $\{\{x\}\} = \text{getRel } S\_Idt\ G\ \{\{y\}\}$ ".
  } note u = this[OF p]
  have "getRel S_Idt G  $\{\{Eps (P x)\}\} = \text{getRel } S\_Idt\ G\ \{\{x\}\}$ "
    unfolding u by (fact refl)
  hence "Eps (P (Eps (P x))) = Eps (P x)" unfolding P by auto
} note P_eq = this
define f where f:"f = ( $\lambda x. (\text{if } \text{getRel } S\_Idt\ G\ \{\{x\}\} = \{\} \text{ then } x \text{ else } (SOME\ y. P\ x\ y))$ )"
have "(f  $\circ$  f) x = f x" for x proof(cases "getRel S_Idt G  $\{\{x\}\} = \{\}$ ")
  case False
  then show ?thesis using P_eq by (simp add:o_def f)

```

```

qed (auto simp:o_def f)
hence idemp: "f o f = f" by auto

from equivE equiv have refl:"refl_on (vertices G) (getRel S_Idt G)"
by auto
hence [intro]:"x ∈ vertices G ⇒ (x, x) ∈ getRel S_Idt G" for x un-
folding refl_on_def by auto
hence vert_P:"x ∈ vertices G ⇒ (x, Eps (P x)) ∈ getRel S_Idt G" for
x
  unfolding P getRel_def by (metis tfl_some Image_singleton_iff getRel_def)
have r1:"x ∈ vertices G ⇔ P x x" for x using refl unfolding refl_on_def
P
  by (metis Image_singleton_iff assms(1) getRel_dom(2))
have r2[simp]:"getRel S_Idt G ‘‘ {x} = {} ⇔ x ∉ vertices G" for x
  using refl assms(1) unfolding refl_on_def
  by (metis Image_singleton_iff empty_iff equalsOI getRel_dom(1))
{ fix x y assume "(S_Idt,x,y) ∈ edges G"
  hence "(x,y) ∈ getRel S_Idt G" unfolding getRel_def by auto
  hence "getRel S_Idt G ‘‘ {x} = getRel S_Idt G ‘‘ {y}"
    using equiv_class_eq[OF equiv] by metis
  hence "Eps (P x) = Eps (P y)" unfolding P by auto
} note idt_eq = this
have ident:"ident_rel S_Idt (map_graph_fn G f)"
proof(rule ident_rell,goal_cases)
  case (1 x) thus ?case unfolding f by auto
next case (2 x y) thus ?case unfolding getRel_def by (auto simp:f intro!:idt_eq)
next case (3 x y) thus ?case unfolding getRel_def by auto
qed

{ fix l x y
  assume a:"(l,x,y) ∈ edges G" "x ∈ vertices G" "y ∈ vertices G"
  hence f:"(f x, x) ∈ getRel S_Idt G" "(f y, y) ∈ getRel S_Idt G"
    using vert_P equivE[OF equiv] sym_def unfolding f by auto
  from a have gr:"(x, y) ∈ getRel l G" unfolding getRel_def by auto
  from congruentD[OF congr(1)[OF gr] f(1)] congruentD[OF congr(2)[OF
gr] f(2)] a(1)
  have "(l,f x, f y) ∈ edges G" unfolding set_eq_iff getRel_def by
auto
} note gu1 = this
{ fix x assume a: "x ∈ vertices G"
  with vert_P have "(x,Eps (P x)) ∈ getRel S_Idt G" by auto
  hence "Eps (P x) ∈ vertices G" using assms(1) unfolding getRel_def
by auto
  hence "f x ∈ vertices G" using a unfolding f by auto
} note gu2 = this
have "graph_union (map_graph_fn G f) G = G"
  using gu1 gu2 assms(1) unfolding graph_union_def by(cases G,auto)
hence subg: "subgraph (map_graph_fn G f) G"
  unfolding subgraph_def using assms(1) by auto

```

```

have congr:"((l, x, y) ∈ edges G) = ((l, f x, f y) ∈ edges G)" for
l x y proof
  assume a:"((l, f x, f y) ∈ edges G)"
  hence gr:"(f x, f y) ∈ getRel l G" unfolding getRel_def by auto
  from a have fv:"f x ∈ vertices G" "f y ∈ vertices G" using assms(1)
by auto
  { fix x assume a:"f x ∈ vertices G" "x ∉ vertices G"
    with assms(1) have "getRel S_Idt G ‘ {x} = {}" by auto
    with a f have False by auto
  }
  with fv have v:"x ∈ vertices G" "y ∈ vertices G" by auto
  have gx:"(x, f x) ∈ getRel S_Idt G" and gy:"(y, f y) ∈ getRel S_Idt
G" by (auto simp: f v vert_P)
  from congruentD[OF congr(1)[OF gr] gx] gr
  have "(x, f y) ∈ getRel l G" by auto
  with congruentD[OF congr(2)[OF gr] gy]
  have "(x, y) ∈ getRel l G" by auto
  thus "((l, x, y) ∈ edges G)" unfolding getRel_def by auto next
  assume e:"((l, x, y) ∈ edges G)"
  hence "x ∈ vertices G" "y ∈ vertices G" using assms(1) by auto
  from gu1[OF e this]
  show "((l, f x, f y) ∈ edges G)".
qed

```

```

from idemp ident subg congr show ?thesis by auto
qed

```

The idempotency property of Lemma 7 suffices to show that 'maintained' is preserved.

```

lemma idemp_embedding_maintained_preserved:
  assumes subg:"subgraph (map_graph_fn G f) G" and f:"∧ x. x∈vertices
G ⇒ (f o f) x = f x"
  and maint:"maintained r G"
  shows "maintained r (map_graph_fn G f)"
proof -
  { fix h assume hom_h:"graph_homomorphism (fst r) (map_graph_fn G f)
h"
  from subgraph_preserves_hom[OF subg this] maint[unfolded maintained_def
extensible_def]
  obtain g where g:"graph_homomorphism (snd r) G g"
    "agree_on (fst r) h g" by blast
  { fix v x
  have subs:"h ‘ {v} ⊆ vertices (map_graph_fn G f)"
    using hom_h[unfolded graph_homomorphism_def] by auto
  assume "v∈vertices (fst r)" and x:"(v, x) ∈ g"
  hence "g ‘ {v} = h ‘ {v}" using g(2)[unfolded agree_on_def,rule_format,of
v] by auto
  hence "g ‘ {v} ⊆ vertices (map_graph_fn G f)" using subs by auto

```

```

    hence x2:"x ∈ vertices (map_graph_fn G f)" using x by auto
    then obtain y where "x = f y" "y ∈ vertices G" by auto
    hence f:"f x = x" using f x2 unfolding o_def by metis
    from x2 subgraph_subset[OF subg] have "(x, f x) ∈ on_graph G f"
  by auto
    with x have "(v, x) ∈ g 0 on_graph G f" "f x = x" unfolding f by
  auto
  }
  hence agr:"agree_on (fst r) h (g 0 on_graph G f)"
    using g(2) unfolding agree_on_def by auto
  have "extensible r (map_graph_fn G f) h"
    unfolding extensible_def using graph_homomorphism_on_graph[OF g(1)]
  agr by blast
}
thus ?thesis unfolding maintained_def by blast
qed

```

Definition 20.

```

definition const_exists where
"const_exists c ≡ transl_rule (⊤ ⊆ A_Cmp (A_Cmp ⊤ (A_Lbl (S_Const c)))
⊤)"
definition const_exists_rev where
"const_exists_rev c ≡ transl_rule (A_Cmp (A_Cmp (A_Lbl (S_Const c)) ⊤)
(A_Lbl (S_Const c)) ⊆ A_Lbl (S_Const c))"
definition const_prop where
"const_prop c ≡ transl_rule (A_Lbl (S_Const c) ⊆ 1)"
definition const_disj where
"const_disj c1 c2 ≡ transl_rule (A_Cmp (A_Lbl (S_Const c1)) (A_Lbl (S_Const
c2)) ⊆ ⊥)"

```

lemma constant_rules:

```

  assumes "standard' C G" "c ∈ C"
  shows "maintained (const_exists c) G"
    "maintained (const_exists_rev c) G"
    "maintained (const_prop c) G"
    "c' ∈ C ⇒ c ≠ c' ⇒ maintained (const_disj c c') G"

```

proof -

```

  note a = assms[unfolded standard_def]
  from a have g:"graph G" by auto
  from a
  have gr_c:"getRel (S_Const c) G = {(Inl c, Inl c)}"
    "getRel S_Idt G = Id_on (vertices G)" "getRel S_Bot G = {}"
    "getRel S_Top G = vertices G × vertices G" by auto
  with g have inlc:"Inl c ∈ vertices G" by (metis getRel_dom(1) singletonI)
  thus "maintained (const_exists c) G" "maintained (const_exists_rev c)
  G"
    "maintained (const_prop c) G"
  unfolding const_prop_def const_exists_rev_def const_exists_def maintained_holds_subset_
  g]

```

```

    by (auto simp:gr_c relcomp_unfold)
  assume "c' ∈ C"
  with a have gr_c':"getRel (S_Const c') G = {(Inl c', Inl c')} by auto
  thus "c ≠ c' ⇒ maintained (const_disj c c') G"
    unfolding const_disj_def maintained_holds_subset_iff[OF g] using gr_c
  by auto
qed

definition constant_rules where
  "constant_rules C ≡ const_exists ' C ∪ const_exists_rev ' C ∪ const_prop
  ' C
    ∪ {const_disj c1 c2 | c1 c2. c1 ∈ C ∧ c2 ∈ C ∧ c1
  ≠ c2}"

lemma constant_rules_graph_rule:
  assumes "x ∈ constant_rules C"
  shows "graph_rule x"
proof -
  from graph_rule_translation
  have gr:"∧ u v . graph_rule (transl_rule (u ⊆ v))" by auto
  consider "∃ v. x = const_exists v" | "∃ v. x = const_exists_rev v"
| "∃ v. x = const_prop v"
| "∃ v w. x = const_disj v w" using assms unfolding constant_rules_def
Un_iff by blast
  thus ?thesis using gr
    unfolding const_exists_def const_exists_rev_def const_prop_def const_disj_def
  by cases fast+
qed

lemma finite_constant[intro]:
  assumes "finite C"
  shows "finite (constant_rules C)"
proof -
  have "{const_disj c1 c2 | c1 c2. c1 ∈ C ∧ c2 ∈ C ∧ c1 ≠ c2} ⊆ case_prod
  const_disj ' (C × C)"
  by auto
  moreover have "finite ..." using assms by auto
  ultimately have "finite {const_disj c1 c2 | c1 c2. c1 ∈ C ∧ c2 ∈ C
  ∧ c1 ≠ c2}"
  by(rule finite_subset)
  thus ?thesis unfolding constant_rules_def using assms by blast
qed

lemma standard_maintains_constant_rules:
  assumes "standard' C G" "R ∈ constant_rules C"
  shows "maintained R G"
proof -
  from assms(2)[unfolded constant_rules_def]
  consider "∃ c ∈ C. R = const_exists c"

```

```

      | "∃ c ∈ C. R = const_exists_rev c"
      | "∃ c ∈ C. R = const_prop c"
      | "∃ c1 c2. c1 ∈ C ∧ c2 ∈ C ∧ c1 ≠ c2 ∧ R = const_disj c1
c2" by blast
  from this assms(1) show ?thesis by(cases,auto simp:constant_rules)
qed

lemma constant_rules_empty[simp]:
  "constant_rules {} = {}"
  by (auto simp:constant_rules_def)

  Definition 20, continued.

definition standard_rules :: "'a set ⇒ 'a Standard_Constant set ⇒ (('a
Standard_Constant, nat) labeled_graph × ('a Standard_Constant, nat) labeled_graph)
set"
  where
    "standard_rules C L ≡ constant_rules C ∪ identity_rules L ∪ {top_rule
S_Top,nonempty_rule}"

lemma constant_rules_mono:
  assumes "C1 ⊆ C2"
  shows "constant_rules C1 ⊆ constant_rules C2"
  using assms unfolding constant_rules_def
  by(intro Un_mono,auto)

lemma identity_rules_mono:
  assumes "C1 ⊆ C2"
  shows "identity_rules C1 ⊆ identity_rules C2"
  using assms unfolding identity_rules_def by auto

lemma standard_rules_mono:
  assumes "C1 ⊆ C2" "L1 ⊆ L2"
  shows "standard_rules C1 L1 ⊆ standard_rules C2 L2"
  using constant_rules_mono[OF assms(1)] identity_rules_mono[OF assms(2)]
  unfolding standard_rules_def by auto

lemma maintainedA_invmono:
  assumes "C1 ⊆ C2" "L1 ⊆ L2"
  shows "maintainedA (standard_rules C2 L2) G ⇒ maintainedA (standard_rules
C1 L1) G"
  using standard_rules_mono[OF assms] by auto

lemma maintained_preserved_by_isomorphism:
  assumes "∧ x. x ∈ vertices G ⇒ (f ∘ g) x = x" "graph G"
  and "maintained r (map_graph_fn G g)"
  shows "maintained r G"
proof(cases r)
  case (Pair L R)
  show ?thesis unfolding Pair proof(standard,goal_cases)

```

```

    case (1 h)
    from assms(3)[unfolded maintained_def Pair] graph_homomorphism_on_graph[OF
this, of g]
    have "extensible (L, R) (map_graph_fn G g) (h 0 on_graph G g)" by
auto
    then obtain h2
    where h2:"graph_homomorphism R (map_graph_fn G g) h2" "agree_on
L (h 0 on_graph G g) h2"
    unfolding extensible_def by auto
    from 1 have h_id:"h 0 Id_on (vertices G) = h" unfolding graph_homomorphism_def
by auto
    let ?h = "h2 0 on_graph (map_graph_fn G g) f"
    from assms(1) have "on_graph G (f ∘ g) = Id_on (vertices G)" by auto
    hence "map_graph_fn G (f ∘ g) = G" using assms(2) map_graph_fn_id
by auto
    with graph_homomorphism_on_graph[OF h2(1),of f]
    have igh:"graph_homomorphism R G ?h" by auto
    have "g x = g xa  $\implies$  x  $\in$  (vertices G)  $\implies$  xa  $\in$  (vertices G)  $\implies$ 
x = xa"
    for x xa using assms(1) o_def by metis
    hence "g x = g xa  $\implies$  x  $\in$  (vertices G)  $\implies$  xa  $\in$  (vertices G)  $\implies$ 
(x, xa)  $\in$  Id_on (vertices G)"
    for x xa by auto
    hence id:"(on_graph G g) 0 on_graph (map_graph_fn G g) f = Id_on (vertices
G)"
    using assms(1) by auto
    from agree_on_ext[OF h2(2),of "on_graph (map_graph_fn G g) f",unfolded
0_assoc]
    have agh:"agree_on L h ?h" unfolding agree_on_def id h_id.
    from igh agh show ?case unfolding extensible_def by auto
qed
qed

lemma standard_identity_rules:
  assumes "standard' C G"
  shows "maintained (reflexivity_rule S_Idt) G"
        "maintained (transitive_rule S_Idt) G"
        "maintained (symmetry_rule S_Idt) G"
        "maintained (congruence_rule S_Idt 1) G"
proof -
  note a = assms[unfolded standard_def]
  from a have g:"graph G" by auto
  from a
  have gr:"getRel S_Idt G = Id_on (vertices G)" "getRel S_Bot G = {}"
        "getRel S_Top G = vertices G  $\times$  vertices G"
    and v_gr:" $\forall$  a b. ((S_Idt, a, b)  $\in$  edges G) = (a  $\in$  vertices G  $\wedge$  b
= a)"
    unfolding getRel_def by auto
  thus "maintained (transitive_rule S_Idt) G" "maintained (symmetry_rule

```

```

S_Idt) G"
  "maintained (congruence_rule S_Idt 1) G"
  unfolding transitive_rule_def symmetry_rule_def congruence_rule_def
    maintained_holds_subset_iff[OF g]
  by (auto simp:gr relcomp_unfold)
{ fix f :: "(nat × ('a + 'b)) set"
  assume "graph_homomorphism (LG {} {0}) G f"
  hence u:"univalent f" and d:"Domain f = {0}"
    and r:"f `` {0} ⊆ vertices G" unfolding graph_homomorphism_def
by simp+
  from d obtain v where v:"(0,v) ∈ f" by auto
  hence f:"f = {(0,v)}"
    using d insert_iff mk_disjoint_insert all_not_in_conv old.prod.exhaust
      u[unfolded univalent_def] Domain.intros[of _ _ f,unfolded
d,THEN singletonD]
    by (metis (no_types))
  from v r have v:"v ∈ vertices G" by auto
  with v_gr have "(S_Idt, v, v) ∈ edges G" by auto
  hence "edge_preserving {(0, v)} {(S_Idt, 0, 0)} (edges G)" unfolding
ing edge_preserving by auto
  hence "graph_homomorphism (LG {(S_Idt, 0, 0)} {0}) G f" unfolding
f
    graph_homomorphism_def using g v by (auto simp:univalent_def)
}
}
thus "maintained (reflexivity_rule S_Idt) G"
  unfolding reflexivity_rule_def maintained_def by auto
qed

lemma standard_maintains_identity_rules:
  assumes "standard' C G" "x∈identity_rules L"
  shows "maintained x G"
proof -
  consider "x = reflexivity_rule S_Idt" | "x = transitive_rule S_Idt"
| "x = symmetry_rule S_Idt"
  | "∃ l. x = congruence_rule S_Idt l" using assms unfolding identity_rules_def
Un_iff by blast
  thus ?thesis using standard_identity_rules[OF assms(1)] by (cases,auto)
qed

lemma standard_maintains_rules:
  assumes "standard' C G"
  shows "maintainedA (standard_rules C L) G"
proof fix R
  assume "R ∈ standard_rules C L"
  then consider "R ∈ constant_rules C" | "R ∈ identity_rules L"
  | "R = top_rule S_Top" | "R = nonempty_rule" by (auto simp:standard_rules_def)
  thus "maintained R G"
    using assms standard_maintains_constant_rules[OF assms]
      standard_maintains_identity_rules[OF assms] by (cases,auto simp:standard_def)

```

qed

A case-split rule.

```

lemma standard_rules_edges:
  assumes "(lhs, rhs) ∈ standard_rules C L" "(l, x, y) ∈ edges rhs"
  shows "(l = S_Bot ⇒ thesis) ⇒
        (l = S_Top ⇒ thesis) ⇒
        (l = S_Idt ⇒ thesis) ⇒
        (l ∈ S_Const ' C ⇒ thesis) ⇒
        (l ∈ L ⇒ thesis) ⇒ thesis"
  using assms [[simproc del: defined_all]]
  by (auto simp: Let_def standard_rules_def constant_rules_def identity_rules_def
      const_exists_def const_exists_rev_def const_prop_def const_disj_def
      reflexivity_rule_def transitive_rule_def symmetry_rule_def congruence_rule_def
      top_rule_def nonempty_rule_def)

```

Lemma 8.

This is a slightly stronger version of Lemma 8: we reason about maintained rather than holds, and the quantification for maintained happens within the existential quantifier, rather than outside.

Due to the type system of Isabelle, we construct the concrete type *std_graph* for *G*. This in contrast to arguing that 'there exists a type large enough', as in the paper.

```

lemma maintained_standard_noconstants:
  assumes mnt:"maintainedA (standard_rules C L) G'"
  and gr:"graph (G'::( 'V Standard_Constant, 'V') labeled_graph)"
        "fst ' edges G' ⊆ L"
  and cf:"getRel S_Bot G' = {}"
  shows "∃ f g (G'::( 'V, 'V') std_graph). G = map_graph_fn G (f o g) ∧ G
        = map_graph_fn G' f
          ∧ subgraph (map_graph_fn G g) G'
          ∧ standard' C G
          ∧ (∀ r. maintained r G' → maintained r G)
          ∧ (∀ x y e. x ∈ vertices G' → y ∈ vertices G' →
              (g (f x), g (f y)) ∈ :map_graph_fn G g:[e] →
                (x,y) ∈ :G':[e])"
  proof -
    note mnt = mnt[unfolded standard_rules_def]
    from mnt have "maintainedA (identity_rules L) G'" by auto
    from identity_rules[OF gr(1) this gr(2)] obtain h where
      h:"h o h = h" "ident_rel S_Idt (map_graph_fn G' h)" "subgraph (map_graph_fn
        G' h) G'"
    "((l, x, y) ∈ edges G') = ((l, h x, h y) ∈ edges G'" for l x y
  by blast
  have mg:"∧ r. maintained r G' ⇒ maintained r (map_graph_fn G' h)"
    using idemp_embedding_maintained_preserved[OF h(3)] h(1) by auto
  from mnt have tr:"maintained (top_rule S_Top) G'" and ne:"maintained
    nonempty_rule G'" by auto

```

```

    from nonempty_rule[OF gr(1)] ne obtain x where x:"x ∈ vertices G'"
  by blast
  from tr[unfolded top_rule[OF gr(1)]] x have top_nonempty:"(x, x) ∈
  getRel S_Top G'" by auto
  have "∧ c. c ∈ C ⇒ ∃ v. (v, v) ∈ getRel (S_Const c) (map_graph_fn
  G' h)" proof(goal_cases)
    case (1 c)
    with mnt have cr5: "maintained (const_exists c) G'"
      and cr7: "maintained (const_prop c) G'" unfolding constant_rules_def
  by blast+
  from top_nonempty cr5[unfolded maintained_holds_subset_iff[OF gr(1)]]
  const_exists_def
  obtain y z where yz:"(y,z) ∈ getRel (S_Const c) G'" by auto
  from this gr(1) have yzv:"y ∈ vertices G'" "z ∈ vertices G'" by
  (auto simp:getRel_def)
  from getRel_hom[OF yz yzv]
  have hi:"(h y,h z) ∈ getRel (S_Const c) (map_graph_fn G' h)".
  with h(2) cr7[THEN mg,unfolded maintained_holds_subset_iff[OF map_graph_fn_graphI]]
  const_prop_def]
  have "h y = h z" by force
  thus "∃ v. (v,v) ∈ getRel (S_Const c) (map_graph_fn G' h)" using
  hi by auto
  qed
  hence "∀ c. ∃ v. c ∈ C → (v, v) ∈ getRel (S_Const c) (map_graph_fn
  G' h)" by blast
  from choice[OF this] obtain m
  where m:"∧ x. x ∈ C ⇒ (m x, m x) ∈ getRel (S_Const x) (map_graph_fn
  G' h)" by blast
  let ?m' = "λ x. if x ∈ m ' C then Inl (the_inv_into C m x) else Inr
  x"
  define f where "f ≡ ?m' o h"
  have "∧ x y. x ∈ C ⇒ y ∈ C ⇒ m x = m y ⇒ x = y" proof(goal_cases)
    case (1 x y)
    with m have "(m x,m x) ∈ getRel (S_Const y) (map_graph_fn G' h)"
      "(m x,m x) ∈ getRel (S_Const x) (map_graph_fn G' h)" by
  metis+
  hence mx: "(m x,m x) ∈ getRel (S_Const y) G'"
    "(m x,m x) ∈ getRel (S_Const x) G'" using h(3) by force+
  from 1(1,2) mnt have cr8:"x ≠ y ⇒ maintained (const_disj x y)
  G'"
    unfolding constant_rules_def by blast
  from cr8[unfolded maintained_holds_subset_iff[OF gr(1)]] const_disj_def]
  mx
  have "x≠y⇒(m x,m x) ∈ :G':[⊥]" by auto
  thus "x = y" using cf by auto
  qed
  hence "univalent (converse (BNF_Def.Gr C m))" unfolding univalent_def
  by auto
  hence inj_m:"inj_on m C" unfolding inj_on_def by auto

```

```

from inj_on_the_inv_into[OF inj_m] have inj_m':"inj ?m'" unfolding
inj_on_def by auto
define G where "G = map_graph_fn G' f"
hence G:"graph G" "f x ∈ vertices G" "getRel S_Bot G = {}" using x
cf unfolding getRel_def
by force+
from comp_inj_on[OF inj_on_the_inv_into[OF inj_m] inj_Inl, unfolded
o_def] inj_Inr
have inj_m':"inj_on ?m' (vertices G')" unfolding inj_on_def by auto
define g where "g = the_inv_into (vertices G') ?m'"

have gf_h:" $\bigwedge x. x \in \text{vertices } G' \implies (g \circ f) x = h x$ " unfolding g_def
f_def o_def
apply(rule the_inv_into_f_f[OF inj_m']) using h unfolding subgraph_def
graph_union_iff by auto

have mg_eq:"map_graph_fn G' (g ∘ f) = map_graph_fn G' h"
by (rule map_graph_fn_eqI[OF gf_h])

have " $\bigwedge x. x \in \text{vertices } G' \implies h x \in \text{vertices } G''$ " using h(3)
unfolding subgraph_def graph_union_iff by(cases G',auto)
hence gf_id:" $\bigwedge x. x \in \text{vertices } G' \implies (g \circ f) (h x) = (h x)$ "
using h(1) gf_h unfolding o_def by metis
{ fix x assume "x ∈ vertices G"
then obtain y where y:"f y = x" "y ∈ vertices G'" unfolding G_def
by auto
from gf_h[OF y(2)] have "(f ∘ g) (f y) = f (h y)" unfolding o_def
by auto
also have "... = f y" using h(1) unfolding f_def o_def by metis
finally have "(f ∘ g) x = x" unfolding y.
} note fg_id = this

have fg_inv:"map_graph_fn G (f ∘ g) = G"
using h(1) G_def f_def mg_eq map_graph_fn_comp by (metis (no_types,
lifting))

have ir:"ident_rel S_Idt G" unfolding set_eq_iff proof(standard,standard,goal_cases)
case (1 x)
from this[unfolded G_def]
obtain v1 v2 where v:"(v1,v2) ∈ getRel S_Idt G'" "x = (f v1,f v2)"
unfolding getRel_def map_graph_def on_triple_def by auto
hence vv:"v1 ∈ vertices G'" "v2 ∈ vertices G'" using gr unfolding
getRel_def by auto
with h(2) v(1) have "h v1 = h v2" unfolding image_def by blast
hence x:"x = (f v1,f v1)" unfolding f_def v by auto
from vv(1) show ?case unfolding x G_def by auto
next
case (2 x)

```

```

    hence x:"fst x = snd x" "fst x ∈ vertices G" by auto
    hence "(fst x) ∈ f ` vertices G'" unfolding G_def o_def by auto
    then obtain v where v:"v ∈ vertices G'" "f v = fst x" by auto
    hence hv:"h v ∈ vertices (map_graph_fn G' h)" by simp
    hence "(h v, h v) ∈ getRel S_Idt (map_graph_fn G' h)" unfolding h(2)
  by auto
    from getRel_hom[OF this hv hv]
    have "(?m' (h v), ?m' (h v)) ∈ getRel S_Idt (map_graph_fn G' (?m'
o h))"
      unfolding map_graph_fn_comp by fast
    hence "(f v, f v) ∈ getRel S_Idt (map_graph_fn G' f)" unfolding f_def
  by auto
    hence "(fst x, snd x) ∈ getRel S_Idt G" unfolding x v G_def by auto
    thus ?case unfolding G_def by auto
  qed

  from tr[unfolded top_rule[OF gr(1)]]
  have tr0:"getRel S_Top (map_graph_fn G' h)
    = {(x,y). x ∈ vertices (map_graph_fn G' h) ∧ y ∈ vertices (map_graph_fn
G' h)}"
    and tr:"getRel S_Top G = {(x, y). x ∈ vertices G ∧ y ∈ vertices
G}"
    unfolding G_def getRel_def on_triple_def map_graph_def by auto

  have m:"∧ x. x ∈ C ⇒ {(m x, m x)} = getRel (S_Const x) (map_graph_fn
G' h)" proof fix x
    assume x:"x ∈ C"
    { fix y z assume a:"(y,z) ∈ getRel (S_Const x) (map_graph_fn G' h)"
      let ?t = "getRel S_Top (map_graph_fn G' h)"
      let ?r = "getRel (S_Const x) (map_graph_fn G' h)"
      have mx:"(m x, m x) ∈ getRel (S_Const x) (map_graph_fn G' h)" us-
ing m x by auto
        with a have v:"y ∈ vertices (map_graph_fn G' h)"
          "z ∈ vertices (map_graph_fn G' h)"
          "m x ∈ vertices (map_graph_fn G' h)" unfolding getRel_def
    by force+
      with tr0 have "(m x, y) ∈ ?t" "(z, m x) ∈ ?t" by auto
      with a mx have lhs:"(m x, z) ∈ ?r 0 ?t 0 ?r" "(y, m x) ∈ ?r 0 ?t
0 ?r" by auto
      from x mnt have "maintained (const_exists_rev x) G'"
        and "maintained (const_prop x) G'" unfolding constant_rules_def
    by blast+
      hence cr6:"maintained (const_exists_rev x) (map_graph_fn G' h)"
        and cr7:"maintained (const_prop x) (map_graph_fn G' h)"
        by (intro mg, force)+
      hence "(m x, z) ∈ getRel S_Idt (map_graph_fn G' h)"
        "(y, m x) ∈ getRel S_Idt (map_graph_fn G' h)" using lhs
      unfolding maintained_holds_subset_iff[OF map_graph_fn_graphI]
        const_exists_rev_def const_prop_def by auto

```

```

    hence "y = m x" "z = m x" using h(2) by auto
  }
  thus "getRel (S_Const x) (map_graph_fn G' h)  $\subseteq$  {(m x, m x)}" by auto
qed (insert m,auto)

from mg_eq have mg_eq:"map_graph_fn G g = map_graph_fn G' h" unfolding
G_def map_graph_fn_comp.

{ fix l fix v: "'V + 'V'"
  assume a: "(l, v)  $\in$  ( $\lambda$ c. (S_Const c, Inl c)) ' C"
  hence "getRel l G = {(v, v)}" using m proof(cases l)
    case (S_Const x) hence x: "l = S_Const x" "v = Inl x" "x  $\in$  C" using
a by auto
    hence mx: "m x  $\in$  m ' C" by auto
    from m[OF x(3)] have "(m x, m x)  $\in$  getRel (S_Const x) (map_graph_fn
G' h)" by auto
    hence "(S_Const x, m x, m x)  $\in$  edges (map_graph_fn G' h)" unfolding
ing getRel_def by auto
    hence "m x  $\in$  vertices (map_graph_fn G' h)" unfolding map_graph_def
Image_def by auto
    then obtain x' where x': "m x = h x'" "x'  $\in$  vertices G'" by auto
    from h(1) have hmx[simp]: "h (m x) = m x" unfolding x' o_def by
metis
    hence fmx: "f (m x) = v" unfolding x f_def
      using the_inv_into_f_f[OF inj_m] inj_m[unfolded inj_on_def, rule_format, OF
x(3)] mx by auto
    have "{(f (m x), f (m x))} = getRel l (map_graph_fn G (f  $\circ$  g))"

      unfolding map_graph_fn_comp getRel_hom_map[OF map_graph_fn_graphI]
      m[OF x(3), folded mg_eq x(1), symmetric] by auto
    hence gr: "getRel l G = {(f (m x), f (m x))}" unfolding fg_inv by
blast
    show ?thesis unfolding gr fmx by (fact refl)
  qed auto
} note cr = this

have sg: "subgraph (map_graph_fn G g) G'" unfolding mg_eq using h(3).
have std: "standard' C G" unfolding standard_def using G ir tr cr by
blast
have mtd: " $\bigwedge$ r. maintained r G'  $\implies$  maintained r G" proof(goal_cases)
  case (1 r) from mg[OF 1, folded mg_eq] maintained_preserved_by_isomorphism[OF
fg_id G(1)]
  show ?case by metis
qed

{ fix x y e
  assume "x  $\in$  vertices G'" "y  $\in$  vertices G'"
    "(g (f x), g (f y))  $\in$  :map_graph_fn (map_graph_fn G' f) g: [e]"
  hence "(x, y)  $\in$  :G': [e]"

```

```

    proof(induct e arbitrary: x y)
      case (A_Cmp e1 e2)
        then obtain z where z:"(g (f x), z) ∈ :map_graph_fn (map_graph_fn
G' f) g:[e1]"
          "(z, g (f y)) ∈ :map_graph_fn (map_graph_fn
G' f) g:[e2]" by auto
          hence "z ∈ vertices (map_graph_fn (map_graph_fn G' f) g)"
            using semantics_in_vertices(1)[OF map_graph_fn_graphI] by metis
          then obtain z' where z':"z = g (f z')" "z' ∈ vertices G'" by
auto
          with A_Cmp(1)[OF A_Cmp(3) z'(2) z(1)[unfolded z']]
            A_Cmp(2)[OF z'(2) A_Cmp(4) z(2)[unfolded z']]
          have "(x, y) ∈ (:G':[e1]) 0 (:G':[e2])" by auto
          then show ?case by auto
        next
          case (A_Lbl l)
            hence "(l, g (f x), g (f y)) ∈ edges (map_graph_fn G g)"
              by (auto simp:getRel_def G_def)
            then obtain x' y'
              where "(l, x', y') ∈ edges G" "g (f x) = g x'" "g (f y) = g
y'" by auto
            then obtain x' y'
              where xy:"(l, x', y') ∈ edges G'" "g (f x) = g (f x')" "g (f
y) = g (f y')"
              unfolding G_def by auto
            hence "x' ∈ vertices G'" "y' ∈ vertices G'" using gr(1) by auto
            from this[THEN gf_h,unfolded o_def] A_Lbl(1,2)[THEN gf_h,unfolded
o_def]
            have "h x = h x'" "h y = h y'" using xy(2,3) by auto
            hence "(l, x, y) ∈ edges G'" using h(4)[of l x y] h(4)[of l x'
y'] xy(1) by auto
            then show ?case by (simp add:getRel_def)
          qed auto
        }
      hence cons:"(∀ x y e. x ∈ vertices G' → y ∈ vertices G' → (g (f
x), g (f y)) ∈ :map_graph_fn G g:[e] → (x,y) ∈ :G':[e])"
        unfolding G_def by auto

    show ?thesis using cons G_def fg_inv[symmetric] sg std mtd by blast
  qed
end

```

9 Combined correctness

This section does not correspond to any theorems in the paper. However, the main correctness proof is not a theorem in the paper either. As the paper sets out to prove that we can decide entailment and consistency, this

file shows how to combine the results so far and indeed establish those properties.

```

theory CombinedCorrectness
  imports GraphRewriting StandardRules
begin

definition the_model where
  "the_model C Rs
  ≡ let L = fst '  $\bigcup$  ((edges o snd) ' Rs)  $\cup$  {S_Bot,S_Top,S_Idt}  $\cup$  S_Const
  ' C;
      Rules = Rs  $\cup$  (standard_rules C L);
      sel = non_constructive_selector Rules
  in the_lcg sel Rules (0,{})"

definition entailment_model where
  "entailment_model C Rs init
  ≡ let L = fst '  $\bigcup$  ((edges o snd) ' Rs)  $\cup$  {S_Bot,S_Top,S_Idt}  $\cup$  S_Const
  ' C  $\cup$  fst ' edges init;
      Rules = Rs  $\cup$  (standard_rules C L);
      sel = non_constructive_selector Rules
  in the_lcg sel Rules (card (vertices init),edges init)"

abbreviation check_consistency where
  "check_consistency C Rs ≡ getRel S_Bot (the_model C Rs) = {}"

abbreviation check_entailment where
  "check_entailment C Rs R ≡
  let mdl = entailment_model C Rs (translation (fst R))
  in (0,1)  $\in$  :mdl:[snd R]  $\vee$  getRel S_Bot mdl  $\neq$  {}"

definition transl_rules where
  "transl_rules T = ( $\bigcup$  (x, y)  $\in$  T. {(translation x, translation (A_Int x
  y)), (translation y, translation (A_Int y x))})"

lemma gr_transl_rules:
  "x  $\in$  transl_rules T  $\implies$  graph_rule x"
  using graph_rule_translation unfolding transl_rules_def by blast
term entails

lemma check_consistency:
  assumes "finite T" "finite C"
  shows "check_consistency C (transl_rules T)  $\longleftrightarrow$  consistent (t::nat
  itself) C T"
  (is "?lhs = ?rhs")
proof -
  from assms(1) have fin_t:"finite (transl_rules T)" unfolding transl_rules_def
by fast
  define L where

```

```

    "L = fst '  $\bigcup$  ((edges o snd) ' transl_rules T)  $\cup$  {S_Bot,S_Top,S_Idt}
 $\cup$  S_Const ' C"
  have "finite ( $\bigcup$  ((edges o snd) ' transl_rules T))" using fin_t gr_transl_rules
by auto
  hence fin_l:"finite L" unfolding L_def using assms(2) by auto
  define Rules where "Rules = transl_rules T  $\cup$  standard_rules C L"
  hence fin_r:"finite Rules" using assms(2) fin_t fin_l unfolding standard_rules_def
by auto
  have incl_L:"fst ' ( $\bigcup$  ((edges o snd) ' Rules))  $\subseteq$  L"
    unfolding L_def Rules_def by (auto elim:standard_rules_edges)
  have " $\forall R \in$  transl_rules T. graph_rule R" using gr_transl_rules by blast
  moreover have " $\forall R \in$  constant_rules C. graph_rule R" using constant_rules_graph_rule
by auto
  moreover have " $\forall R \in$  identity_rules L. graph_rule R" using identity_rules_graph_rule
by auto
  moreover have " $\forall R \in$  {top_rule S_Top,nonempty_rule}. graph_rule R" us-
ing are_rules(1,2) by fastforce
  ultimately have gr:"set_of_graph_rules Rules"
    unfolding set_of_graph_rules_def Rules_def ball_Un standard_rules_def
    by blast
  define sel where "sel = non_constructive_selector Rules"
  hence sel:"valid_selector Rules sel" using gr non_constructive_selector
by auto
  define cfg where "cfg = the_lcg sel Rules (0, {})"
  have cfg:"cfg = the_model C (transl_rules T)"
    unfolding cfg_def sel_def Rules_def L_def the_model_def Let_def..
  have cfg_c:"least_consequence_graph TYPE('a + nat) Rules (graph_of (0,{}))
cfg"
    unfolding cfg_def
    by (rule lcg_through_make_step[OF fin_r gr _ sel],auto)
  hence cfg_sdt:"maintainedA (standard_rules C L) cfg"
    and cfg_g: "graph cfg"
    and cfg_l:"least TYPE('a + nat) Rules (graph_of (0, {})) cfg"
    and cfg_m:"r  $\in$  transl_rules T  $\implies$  maintained r cfg" for r
    unfolding Rules_def least_consequence_graph_def by auto
  have cfg_lbl:"fst ' edges cfg  $\subseteq$  L"
    unfolding cfg_def by (auto intro!: the_lcg_edges[OF sel incl_L])
  have d1: "?lhs  $\implies$  ?rhs" proof -
    assume ?lhs
    from maintained_standard_noconstants[OF cfg_sdt cfg_g cfg_lbl this[folded
cfg]]
  obtain G :: "('a Standard_Constant, 'a + nat) labeled_graph"
    where G_std:"standard' C G"
    and m:"maintained r cfg  $\implies$  maintained r G"
    for r :: "('a Standard_Constant, nat) Graph_PreRule"
    by blast
  hence g:"graph G" unfolding standard_def by auto
  have "(a,b) $\in$ T  $\implies$  G  $\models$  (a,b)" for a b proof(subst eq_as_subsets,standard)
    assume a:"(a,b) $\in$ T"

```

```

      from a cfg_m[unfolded transl_rules_def, THEN m]
      show "G  $\models$  a  $\sqsubseteq$  b" by (subst maintained_holds_iff[OF g, symmetric])
blast
      from a cfg_m[unfolded transl_rules_def, THEN m]
      show "G  $\models$  b  $\sqsubseteq$  a" by (subst maintained_holds_iff[OF g, symmetric])
blast
  qed
  hence h: "( $\forall S \in T. G \models S$ )" by auto
  with G_std show ?rhs unfolding model_def by blast
  qed
  have d2: " $\neg ?lhs \implies ?rhs \implies \text{False}$ " proof -
    assume " $\neg ?lhs$ "
    then obtain a b where ab: "(S_Bot, a, b)  $\in$  edges cfg"
      "a  $\in$  vertices cfg" "b  $\in$  vertices cfg"
      using cfg_g unfolding cfg getRel_def by auto
    assume ?rhs then obtain G :: "('a Standard_Constant, 'a + nat) labeled_graph"
      where G: "model C G T" by auto
    with model_def have std: "standard' C G" and holds: " $\forall S \in T. G \models S$ "
  by fast+
    hence g: "graph G" unfolding standard_def by auto
    from maintained_holds_iff[OF g] holds
    have "maintainedA (transl_rules T) G" unfolding transl_rules_def by
  auto
    hence mnt: "maintainedA Rules G" unfolding Rules_def
      using standard_maintains_rules[OF std] by auto
    from consequence_graphI[OF _ _ g] gr[unfolded set_of_graph_rules_def]
  mnt
    have cg: "consequence_graph Rules G" by fast
    with cfg_l[unfolded least_def]
    have mtd: "maintained (graph_of (0, {}), cfg) G" by blast
    have "graph_homomorphism (fst (graph_of (0::nat, {}), cfg)) G {}"
      unfolding graph_homomorphism_def using g by auto
    with mtd maintained_def have "extensible (graph_of (0, {}), cfg)
  G {}" by auto
    then obtain g where "edges (map_graph g cfg)  $\subseteq$  edges G" "vertices
  cfg = Domain g"
      unfolding extensible_def graph_homomorphism_def2 graph_union_iff
  by auto
    hence " $\exists$  a b. (S_Bot, a, b)  $\in$  edges G" using ab unfolding edge_preserving
  by auto
    thus False using std unfolding standard_def getRel_def by auto
  qed
  from d1 d2 show ?thesis by metis
  qed

lemma check_entailment:
  assumes "finite T" "finite C"
  shows "check_entailment C (transl_rules T) S  $\longleftrightarrow$  entails (t::nat itself)"

```

```

C T (fst S, (A_Int (fst S) (snd S)))"
  (is "?lhs = ?rhs")
proof -
  from assms(1) have fin_t:"finite (transl_rules T)" unfolding transl_rules_def
by fast
  define R where "R = transl_rule S"
  define init where "init = (card (vertices (fst R)), edges (fst R))"
  have gi[intro]:"graph (graph_of init)" and init:"graph_of init = translation
(fst S)"
  using verts_in_translation[of "fst S"] unfolding inv_translation_def
init_def R_def by auto
  define Rs where "Rs = transl_rules T"
  define L where
    "L = fst ' (∪ ((edges o snd) ' Rs)) ∪ {S_Bot,S_Top,S_Idt} ∪ S_Const
' C ∪ fst ' edges (fst R)"
  have "finite (∪ ((edges o snd) ' transl_rules T))" using fin_t gr_transl_rules
by auto
  hence fin_l:"finite L" unfolding L_def Rs_def R_def using assms(2)
by auto
  have fin_t:"finite Rs" using fin_t Rs_def by auto
  define Rules where "Rules = Rs ∪ standard_rules C L"
  hence fin_r:"finite Rules" using assms(2) fin_t fin_l unfolding standard_rules_def
by auto
  have incl_L:"fst ' (∪ ((edges o snd) ' Rules)) ⊆ L" "fst ' snd init
⊆ L"
  unfolding L_def Rules_def init_def by (auto elim:standard_rules_edges)
  have "∀R∈transl_rules T. graph_rule R" using gr_transl_rules by blast
  moreover have "∀R∈ constant_rules C. graph_rule R" using constant_rules_graph_rule
by auto
  moreover have "∀R∈ identity_rules L. graph_rule R" using identity_rules_graph_rule
by auto
  moreover have "∀R∈ {top_rule S_Top,nonempty_rule}. graph_rule R" us-
ing are_rules(1,2) by fastforce
  ultimately have gr:"set_of_graph_rules Rules"
  unfolding set_of_graph_rules_def Rules_def ball_Un standard_rules_def
Rs_def
  by blast
  define sel where "sel = non_constructive_selector Rules"
  hence sel:"valid_selector Rules sel" using gr non_constructive_selector
by auto
  define cfg where "cfg = the_lcg sel Rules init"
  have cfg:"cfg = entailment_model C Rs (fst R)"
  unfolding cfg_def sel_def Rules_def L_def entailment_model_def Let_def
init_def..
  have cfg_c:"least_consequence_graph TYPE('a + nat) Rules (graph_of init)
cfg"
  unfolding cfg_def by (rule lcg_through_make_step[OF fin_r gr gi sel])
  hence cfg_sdt:"maintainedA (standard_rules C L) cfg"
  and cfg_g: "graph cfg"

```

```

and cfg_l:"least TYPE('a + nat) Rules (graph_of init) cfg"
and cfg_m:"r ∈ Rs ⇒ maintained r cfg" for r
unfolding Rules_def least_consequence_graph_def by auto
have cfg_lbl:"fst ' edges cfg ⊆ L" unfolding cfg_def
  by (auto intro!: the_lcg_edges[OF sel incl_L])
have "(0,1) ∈ :translation (fst S):[fst S]" by (fact translation_self)
hence "(0,1) ∈ :graph_of init:[fst S]" unfolding init by auto
from subgraph_semantics[OF _ this] cfg_l[unfolded least_def]
have cfg_fst:"(0,1) ∈ :cfg:[fst S]" unfolding cfg_def by auto
from semantics_in_vertices[OF cfg_g this]
have cfg_01:"0 ∈ vertices cfg" "1 ∈ vertices cfg" "(0,1)∈vertices cfg×vertices
cfg" by auto
have d1: "¬ ?lhs ⇒ ?rhs ⇒ False" proof -
  assume "¬ ?lhs"
  hence gr:"(0,1) ∉ :cfg:[snd S]" "getRel S_Bot cfg = {}"
    unfolding entailment_model_def cfg_R_def Rs_def Let_def by auto
  from maintained_standard_noconstants[OF cfg_sdt cfg_g cfg_lbl gr(2)]
  obtain G :: "('a Standard_Constant, 'a + nat) labeled_graph" and
f g
  where fg:"G = map_graph_fn G (f ∘ g)"
    and f:"G = map_graph_fn cfg f" "subgraph (map_graph_fn G g) cfg"
    and G_std:"standard' C G"
    and m:"∧ r:: ('a Standard_Constant, nat) Graph_PreRule. maintained
r cfg ⇒ maintained r G"
    and e:"∧ x y e. x ∈ vertices cfg ⇒ y ∈ vertices cfg ⇒
(g (f x), g (f y)) ∈ :map_graph_fn G g:[e] ⇒
(x,y) ∈ :cfg:[e]"
  by clarify blast
  hence g:"graph G" unfolding standard_def by auto
  have "(a,b)∈T ⇒ G ⊨ (a,b)" for a b apply(subst eq_as_subsets)
    using cfg_m[unfolded transl_rules_def Rs_def,THEN m]
    unfolding maintained_holds_iff[OF g,symmetric] by blast
  hence h:"(∀S∈T. G ⊨ S)" by auto
  assume "?rhs"
  from this[unfolded entails_def model_def] G_std h have "G ⊨ fst
S ⊆ snd S" by blast
  with cfg_fst cfg_g f(1) have "(f 0, f 1) ∈ :G:[snd S]" by auto
  then have "(g (f 0), g (f 1)) ∈ :map_graph_fn G g:[snd S]" using
map_graph_in[OF g] by auto
  with e cfg_01(1,2) gr(1) show "False" by auto
qed
have "?lhs ⇒ model C G T ⇒ (a,b) ∈ :G:[fst S] ⇒ (a,b) ∈ :G:[snd
S]"
  for G :: "('a Standard_Constant, 'a + nat) labeled_graph" and a b
proof -
  assume mod:"model C G T"
  from mod model_def have std:"standard' C G" and holds:"∀S∈T. G
⊨ S" by fast+
  hence g:"graph G" unfolding standard_def by auto

```

```

with maintained_holds_iff[OF g] holds
have "maintainedA Rs G" unfolding transl_rules_def Rs_def by auto
hence mnt:"maintainedA Rules G" unfolding Rules_def
  using standard_maintains_rules[OF std] by auto
from consequence_graphI[OF _ _ g] gr[unfolded set_of_graph_rules_def]
mnt
  have cg:"consequence_graph Rules G" by fast
  with cfg_l[unfolded least_def] have mtd:"maintained (graph_of init,
cfg) G" by auto
  assume ab:"(a,b) ∈ :G:[fst S]"
  hence av:"a ∈ vertices G" bv:"b ∈ vertices G" using semantics_in_vertices[OF
g] by auto
  from ab translation[OF g] init
  obtain f where f:"graph_homomorphism (graph_of init) G f" "(0, a)
∈ f ∧ (1, b) ∈ f"
  by auto
  from maintainedD2[OF mtd f(1)] obtain g
  where g:"graph_homomorphism cfg G g" and "f ⊆ g" by blast
  with f have g01:"(0, a) ∈ g" "(1, b) ∈ g" by auto
  assume ?lhs
  then consider (maintained) "(0,1) ∈ :cfg:[snd S]" | (no_models) ":cfg:[⊥]
≠ {}"
  using cfg_g unfolding cfg entailment_model_def Let_def Rs_def R_def
by auto
  thus "(a,b) ∈ :G:[snd S]" proof(cases)
  case maintained
  from graph_homomorphism_semantics[OF g maintained g01] show ?thesis.
  next
  case no_models
  from graph_homomorphism_nonempty[OF g no_models]
  have "getRel S_Bot G ≠ {}" by auto
  hence False using std unfolding standard_def by auto
  thus ?thesis by auto
  qed
qed
hence d2:"?lhs ⇒ ?rhs" unfolding entails_def by auto
from d1 d2 show ?thesis by metis
qed
end

```

acknowledgements We thank Gerwin Klein for making an example submission in the AFP [3], which was of great help in making this submission.

References

- [1] S. J. C. Joosten. Parsing and printing of and with triples. In P. Höfner, D. Pous, and G. Struth, editors, *Relational and algebraic methods in*

computer science, Lecture Notes in Computer Science, pages 159–176. Springer, May 2017.

- [2] S. J. C. Joosten. Finding models through graph saturation. *Journal of Logical and Algebraic Methods in Programming*, 100:98–112, 11 2018.
- [3] G. Klein. Example submission. *Archive of Formal Proofs*, 2004.
- [4] G. Michels, S. J. C. Joosten, J. van der Woude, and S. Joosten. Am-
persand: Applying relation algebra in practice. In H. Swart, de, editor,
*International Conference on Relational and Algebraic Methods in Com-
puter Science*, pages 280–293. Springer, Berlin, Heidelberg, May 2011.