

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	10
4	Graph rewriting and saturation	18
5	Semantics in labeled graphs	22
6	Standard Models	25
7	Translating terms into Graphs	26
8	Standard Rules	29
9	Combined correctness	34

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)" <proof>

lemma graph_single[intro]: "graph (LG {(a,b,c)} {b,c})" <proof>

abbreviation finite_graph where
  "finite_graph X ≡ graph X ∧ finite (vertices X) ∧ finite (edges X)"

lemma restrict_idemp[simp]:
  "restrict (restrict x) = restrict x"
  <proof>

lemma vertices_restrict[simp]:
  "vertices (restrict G) = vertices G"
  <proof>

lemma restrictI[intro]:
```

assumes "edges $G \subseteq \{(l,v1,v2). v1 \in \text{vertices } G \wedge v2 \in \text{vertices } G\}$ "

shows " $G = \text{restrict } G$ "
 <proof>

lemma restrict_subsd[dest]:

assumes "edges $G \subseteq \text{edges } (\text{restrict } G)$ "
 shows " $G = \text{restrict } G$ "
 <proof>

lemma restrictD[dest]:

assumes " $G = \text{restrict } G$ "
 shows "edges $G \subseteq \{(l,v1,v2). v1 \in \text{vertices } G \wedge v2 \in \text{vertices } G\}$ "
 <proof>

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]:

" $((l1,v1,v2), (l2,v3,v4)) \in \text{on_triple } R \longleftrightarrow (v1,v3) \in R \wedge (v2,v4) \in R \wedge l1 = l2$ "
 <proof>

lemma on_triple_univ[intro!]:

"univalent $f \implies \text{univalent } (\text{on_triple } f)$ "
 <proof>

lemma on_tripleD[dest]:

assumes " $((l1,v1,v2), (l2,v3,v4)) \in \text{on_triple } R$ "
 shows " $l2 = l1$ " " $(v1,v3) \in R$ " " $(v2,v4) \in R$ "
 <proof>

lemma on_triple_ID_restrict[simp]:

shows " $\text{on_triple } (\text{Id_on } (\text{vertices } G)) \text{ `` edges } G = \text{edges } (\text{restrict } G)$ "
 <proof>

lemma relcomp_on_triple[simp]:

shows " $\text{on_triple } (R \circ S) = \text{on_triple } R \circ \text{on_triple } S$ "
 <proof>

lemma on_triple_preserves_finite[intro]:

"finite $E \implies \text{finite } (\text{on_triple } (\text{BNF_Def.Gr } A f) \text{ `` } E)$ "
 <proof>

lemma on_triple_fst[simp]:

assumes "vertices $G = \text{Domain } g$ " "graph G "
 shows " $x \in \text{fst ' on_triple } g \text{ `` } (\text{edges } G) \longleftrightarrow x \in \text{fst ' edges } G$ "
 <proof>

```

definition edge_preserving where
  "edge_preserving h e1 e2  $\equiv$ 
    ( $\forall (k, v1, v2) \in e1. \forall v1' v2'. ((v1, v1') \in h \wedge (v2, v2') \in h) \rightarrow (k, v1', v2') \in e2$ )"

lemma edge_preserving_atomic:
  assumes "edge_preserving h1 e1 e2" "(v1, v1')  $\in$  h1" "(v2, v2')  $\in$  h1"
  "(k, v1, v2)  $\in$  e1"
  shows "(k, v1', v2')  $\in$  e2"
  <proof>

lemma edge_preservingI[intro]:
  assumes "on_triple R '' E  $\subseteq$  G"
  shows "edge_preserving R E G"
  <proof>

lemma on_triple_dest[dest]:
  assumes "on_triple R '' E  $\subseteq$  G"
  "(l, x, y)  $\in$  E" "(x, xx)  $\in$  R" "(y, yy)  $\in$  R"
  shows "(l, xx, yy)  $\in$  G"
  <proof>

lemma edge_preserving:
  shows "edge_preserving R E G  $\longleftrightarrow$  on_triple R '' E  $\subseteq$  G"
  <proof>

lemma edge_preserving_subset:
  assumes "R1  $\subseteq$  R2" "E1  $\subseteq$  E2" "edge_preserving R2 E2 G"
  shows "edge_preserving R1 E1 G"
  <proof>

lemma edge_preserving_unionI[intro]:
  assumes "edge_preserving f A G" "edge_preserving f B G"
  shows "edge_preserving f (A  $\cup$  B) G"
  <proof>

lemma compose_preserves_edge_preserving:
  assumes "edge_preserving h1 e1 e2" "edge_preserving h2 e2 e3"
  shows "edge_preserving (h1  $\circ$  h2) e1 e3" <proof>

lemma edge_preserving_Id[intro]: "edge_preserving (Id_on y) x x"
  <proof>

```

This is an alternate version of definition 10. We require `@termvertices s = 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 ⊆ vertices t"
        "univalent h"
        "edge_preserving h (edges s) (edges t)"
        "s = restrict s" "t = restrict t"
shows "graph_homomorphism s t h" <proof>

```

lemma *graph_homomorphism_composes*[intro]:

```
assumes "graph_homomorphism a b x"
        "graph_homomorphism b c y"
shows "graph_homomorphism a c (x ∘ y)" <proof>

```

lemma *graph_homomorphism_empty*[simp]:

```
"graph_homomorphism (LG {} {}) G f ↔ f = {} ∧ graph G"
<proof>

```

lemma *graph_homomorphism_Id*[intro]:

```
shows "graph_homomorphism (restrict a) (restrict a) (Id_on (vertices
a))"
<proof>

```

lemma *Id_on_vertices_identity*:

```
assumes "graph_homomorphism a b f"
        "(aa, ba) ∈ f"
shows "(aa, ba) ∈ Id_on (vertices a) ∘ f"
        "(aa, ba) ∈ f ∘ Id_on (vertices b)"
<proof>

```

Alternate version of definition 7.

abbreviation *subgraph*

```
where "subgraph G1 G2
  ≡ graph_homomorphism G1 G2 (Id_on (vertices G1))"

```

lemma *subgraph_trans*:

```
assumes "subgraph G1 G2" "subgraph G2 G3"
shows "subgraph G1 G3"
<proof>

```

Just before Definition 7 in the paper, a notation is introduced for ap-

plying 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)"`

`<proof>`

lemma `map_graph_comp[simp]`:

`assumes "Range g ⊆ Domain f"`

`shows "map_graph (g ∘ f) = map_graph f ∘ map_graph g"`

`<proof>`

lemma `map_graph_returns_restricted`:

`assumes "vertices G = Domain f"`

`shows "map_graph f G = restrict (map_graph f G)"`

`<proof>`

lemma `map_graph_preserves_restricted[intro]`:

`assumes "graph G"`

`shows "graph (map_graph f G)"`

`<proof>`

lemma `map_graph_edge_preserving[intro]`:

`shows "edge_preserving f (edges G) (edges (map_graph f G))"`

`<proof>`

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>`

lemma `map_graph_homo_simp`:

`"graph_homomorphism G (map_graph f G) f"`

`↔ univalent f ∧ vertices G = Domain f ∧ graph G"`

`<proof>`

abbreviation `on_graph` where

`"on_graph G f ≡ BNF_Def.Gr (vertices G) f"`

abbreviation `map_graph_fn` where

`"map_graph_fn G f ≡ map_graph (on_graph G f) G"`

lemma `map_graph_fn_graphI[intro]`:

`"graph (map_graph_fn G f)" <proof>`

```

lemma on_graph_id[simp]:
  shows "on_graph B id = Id_on (vertices B)"
  ⟨proof⟩

lemma in_on_graph[intro]:
  assumes "x ∈ vertices G" "(a x,y) ∈ b"
  shows "(x, y) ∈ on_graph G a 0 b"
  ⟨proof⟩

lemma on_graph_comp:
  "on_graph G (f o g) = on_graph G g 0 on_graph (map_graph_fn G g) f"
  ⟨proof⟩

lemma map_graph_fn_eqI:
  assumes "∧ x. x ∈ vertices G ⇒ f x = g x"
  shows "map_graph_fn G f = map_graph_fn G g" (is "?l = ?r")
  ⟨proof⟩

lemma map_graph_fn_comp[simp]:
  "map_graph_fn G (f o g) = map_graph_fn (map_graph_fn G g) f"
  ⟨proof⟩

lemma map_graph_fn_id[simp]:
  "map_graph_fn X id = restrict X"
  "map_graph (Id_on (vertices X)) X = restrict X"
  ⟨proof⟩

lemma graph_homo[intro!]:
  assumes "graph G"
  shows "graph_homomorphism G (map_graph_fn G f) (on_graph G f)"
  ⟨proof⟩

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⟩

lemma edge_preserving_on_graphI[intro]:
  assumes "∧ l x y. (l,x,y)∈edges X ⇒ x∈vertices X ⇒ y ∈ vertices X ⇒ (l,f x,f y) ∈ Y"
  shows "edge_preserving (on_graph X f) (edges X) Y"
  ⟨proof⟩

lemma subgraph_subset:
  assumes "subgraph G1 G2"
  shows "vertices G1 ⊆ vertices G2" "edges (restrict G1) ⊆ edges G2"
  ⟨proof⟩

```

Our definition of subgraph is equivalent to definition 7.

```
lemma subgraph_def2:
  assumes "graph G1" "graph G2"
  shows "subgraph G1 G2  $\longleftrightarrow$  vertices G1  $\subseteq$  vertices G2  $\wedge$  edges G1  $\subseteq$ 
edges G2"
<proof>
```

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 G1 G2 = LG (edges G1  $\cup$  edges G2) (vertices G1  $\cup$  vertices
G2)"
```

```
lemma graph_unionI[intro]:
  assumes "edges G1  $\subseteq$  edges G2"
          "vertices G1  $\subseteq$  vertices G2"
  shows "graph_union G1 G2 = G2"
<proof>
```

```
lemma graph_union_iff:
  shows "graph_union G1 G2 = G2  $\longleftrightarrow$  (edges G1  $\subseteq$  edges G2  $\wedge$  vertices
G1  $\subseteq$  vertices G2)"
<proof>
```

```
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)"
<proof>
```

```
lemma graph_union_vertices[simp]:
"vertices (graph_union G1 G2) = vertices G1  $\cup$  vertices G2"
<proof>
```

```
lemma graph_union_edges[simp]:
"edges (graph_union G1 G2) = edges G1  $\cup$  edges G2"
<proof>
```

```
lemma graph_union_preserves_restrict[intro]:
  assumes "G1 = restrict G1" "G2 = restrict G2"
  shows "graph_union G1 G2 = restrict (graph_union G1 G2)"
<proof>
```

```
lemma graph_map_union[intro]:
  assumes " $\bigwedge$  i::nat. graph_union (map_graph (g i) X) Y = Y" " $\bigwedge$  i j.
i  $\leq$  j  $\implies$  g i  $\subseteq$  g j"
  shows "graph_union (map_graph ( $\bigcup$  i. g i) X) Y = Y"
```

<proof>

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

lemma *subgraph_def*:

"*subgraph* $G_1 G_2 = (G_1 = \text{restrict } G_1 \wedge G_2 = \text{restrict } G_2 \wedge \text{graph_union } G_1 G_2 = G_2)$ "

<proof>

lemma *subgraph_refl[simp]*:

"*subgraph* $G G = (G = \text{restrict } G)$ "

<proof>

lemma *subgraph_restrict[simp]*:

"*subgraph* $G (\text{restrict } G) = \text{graph } G$ "

<proof>

Definition 10. We write *graph_homomorphism* instead of embedding.

lemma *graph_homomorphism_def2*:

shows "*graph_homomorphism* $G_1 G_2 f =$
 $(\text{vertices } G_1 = \text{Domain } f \wedge \text{univalent } f \wedge G_1 = \text{restrict } G_1 \wedge G_2 = \text{restrict } G_2 \wedge \text{graph_union } (\text{map_graph } f G_1) G_2 = G_2)$ "
(is "*?lhs = ?rhs*")

<proof>

lemma *map_graph_preserves_subgraph[intro]*:

assumes "*subgraph* $A B$ "

shows "*subgraph* $(\text{map_graph } f A) (\text{map_graph } f B)$ "

<proof>

lemma *graph_homomorphism_concr_graph*:

assumes "*graph* G " "*graph* $(LG e v)$ "

shows "*graph_homomorphism* $(LG e v) G x \longleftrightarrow$
 $x \text{ `` } v \subseteq \text{vertices } G \wedge \text{on_triple } x \text{ `` } e \subseteq \text{edges } G \wedge \text{univalent } x \wedge \text{Domain } x = v$ "

<proof>

lemma *subgraph_preserves_hom*:

assumes "*subgraph* $A B$ "

"*graph_homomorphism* $X A h$ "

shows "*graph_homomorphism* $X B h$ "

<proof>

lemma *graph_homo_union_id*:

assumes "*graph_homomorphism* $(\text{graph_union } A B) G f$ "

shows "*graph* $A \implies \text{graph_homomorphism } A G (\text{Id_on } (\text{vertices } A) 0 f)$ "

"*graph* $B \implies \text{graph_homomorphism } B G (\text{Id_on } (\text{vertices } B) 0 f)$ "

<proof>

```

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>

lemma graph_homomorphism_on_graph:
  assumes "graph_homomorphism A B R"
  shows "graph_homomorphism A (map_graph_fn B f) (R  $\circ$  on_graph B f)"
  <proof>

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  $\Rightarrow$  ('l, 'v) labeled_graph)"

```

Definition 8.

```

definition chain :: "('l, 'v) graph_seq  $\Rightarrow$  bool" where
  "chain S  $\equiv$   $\forall$  i. subgraph (S i) (S (i + 1))"

```

```

lemma chain_then_restrict:
  assumes "chain S" shows "S i = restrict (S i)"
  <proof>

```

```

lemma chain:
  assumes "chain S"
  shows "j  $\geq$  i  $\implies$  subgraph (S i) (S j)"
  <proof>

```

```

lemma chain_def2:
  "chain S = ( $\forall$  i j. j  $\geq$  i  $\longrightarrow$  subgraph (S i) (S j))"
  <proof>

```

Second part of definition 8.

```

definition chain_sup :: "('l, 'v) graph_seq  $\Rightarrow$  ('l, 'v) labeled_graph"
where
  "chain_sup S  $\equiv$  LG ( $\bigcup$  i. edges (S i)) ( $\bigcup$  i. vertices (S i))"

```

```

lemma chain_sup_const[simp]:
  "chain_sup ( $\lambda x. S$ ) = S"
  <proof>

lemma chain_sup_subgraph[intro]:
  assumes "chain S"
  shows "subgraph (S j) (chain_sup S)"
  <proof>

lemma chain_sup_graph[intro]:
  assumes "chain S"
  shows "graph (chain_sup S)"
  <proof>

lemma map_graph_chain_sup:
  "map_graph g (chain_sup S) = chain_sup (map_graph g o S)"
  <proof>

lemma graph_union_chain_sup[intro]:
  assumes " $\bigwedge i. \text{graph\_union } (S i) C = C$ "
  shows "graph_union (chain_sup S) C = C"
  <proof>

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)"
  <proof>

  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" <proof>

lemma agree_on_comm[intro]: "agree_on X f g = agree_on X g f" <proof>
lemma agree_on_refl[intro]:
  "agree_on R f f" <proof>

```

```

lemma agree_on_trans:
  assumes "agree_on X f g" "agree_on X g h"
  shows "agree_on X f h" <proof>

lemma agree_on_equivp:
  shows "equivp (agree_on G)"
  <proof>

lemma agree_on_subset:
  assumes "f  $\subseteq$  g" "vertices G  $\subseteq$  Domain f" "univalent g"
  shows "agree_on G f g"
  <proof>

lemma agree_iff_subset[simp]:
  assumes "graph_homomorphism G X f" "univalent g"
  shows "agree_on G f g  $\longleftrightarrow$  f  $\subseteq$  g"
  <proof>

lemma agree_on_ext:
  assumes "agree_on G f1 f2"
  shows "agree_on G (f1  $\cup$  g) (f2  $\cup$  g)"
  <proof>

lemma agree_on_then_eq:
  assumes "agree_on G f1 f2" "Domain f1 = vertices G" "Domain f2 = vertices G"
  shows "f1 = f2"
  <proof>

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"
  <proof>

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"
  <proof>

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 $\langle proof \rangle$

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"
 $\langle proof \rangle$

lemma extensible_chain_sup[intro]:
 assumes "chain S" "extensible R (S j) f"
 shows "extensible R (chain_sup S) f"
 $\langle proof \rangle$

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"
 $\langle proof \rangle$

lemma maintainedD[dest]:
 assumes "maintained (A,B) G" "graph_homomorphism A G f"
 shows "extensible (A,B) G f"
 $\langle proof \rangle$

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
 $\langle proof \rangle$

lemma extensible_refl[intro]:
 "graph_homomorphism R G f \Longrightarrow extensible (R,R) G f"
 $\langle proof \rangle$

lemma maintained_refl[intro]:
 "maintained (R,R) G" $\langle proof \rangle$

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. finite_graph F
   $\longrightarrow$  subgraph F (fst R)
   $\longrightarrow$  extensible (F,fst R) G f
   $\longrightarrow$  graph_homomorphism F G f
   $\longrightarrow$  extensible (F,snd R) G f"

```

```

lemma fin_maintainedI [intro]:
  assumes " $\wedge$  F f. finite_graph F
     $\implies$  subgraph F (fst R)
     $\implies$  extensible (F,fst R) G f
     $\implies$  graph_homomorphism F G f
     $\implies$  extensible (F,snd R) G f"
  shows "fin_maintained R G" <proof>

```

```

lemma maintained_then_fin_maintained[simp]:
  assumes maintained:"maintained R G"
  shows "fin_maintained R G"
<proof>

```

```

lemma fin_maintained_maintained:
  assumes "finite_graph (fst R)"
  shows "fin_maintained R G  $\longleftrightarrow$  maintained R G" (is "?lhs = ?rhs")
<proof>

```

```

lemma extend_for_chain:
  assumes "g 0 = f"
    and " $\wedge$  i. graph_homomorphism (S i) C (g i)"
    and " $\wedge$  i. agree_on (S i) (g i) (g (i + 1))"
    and "chain S"
  shows "extensible (S 0, chain_sup S) C f"
<proof>

```

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 " $\wedge$  R. R  $\in$  Rs  $\implies$  maintained R G"
    " $\wedge$  R. R  $\in$  Rs  $\implies$  subgraph (fst R) (snd R)"
    "graph G"
  shows "consequence_graph Rs G"
<proof>

```

```

lemma consequence_graphD[dest]:
  assumes "consequence_graph Rs G"
  shows " $\wedge$  R. R  $\in$  Rs  $\implies$  maintained R G"
    " $\wedge$  R. R  $\in$  Rs  $\implies$  subgraph (fst R) (snd R)"
    "graph G"

```

<proof>

Definition 8 states: If furthermore S is a subgraph of G , and (S, G) is maintained in each consequence graph maintaining R_s , then G is a least consequence graph of S maintaining R_s . 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  $\Rightarrow$  (('l, 'v) Graph_PreRule) set  $\Rightarrow$  ('l, 'c) labeled_graph
 $\Rightarrow$  ('l, 'c) labeled_graph  $\Rightarrow$  bool"
  where "least _ Rs S G  $\equiv$  subgraph S G  $\wedge$ 
        ( $\forall$  C :: ('l, 'x) 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) labeled_graph. consequence_graph Rs C  $\implies$  maintained
(S,G) C"
  shows "least (t:: 'x itself) Rs S G"
<proof>
```

definition *least_consequence_graph*

```
:: "'x itself  $\Rightarrow$  (('l, 'v) Graph_PreRule) set
 $\Rightarrow$  ('l, 'c) labeled_graph  $\Rightarrow$  ('l, 'c) labeled_graph  $\Rightarrow$  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) labeled_graph. consequence_graph Rs C  $\implies$  maintained
(S,G) C"
  shows "least_consequence_graph (t:: 'x itself) Rs S G"
<proof>
```

Definition 12.

definition *fair_chain* where

```
"fair_chain Rs S  $\equiv$  chain S  $\wedge$ 
( $\forall$  R f i. (R  $\in$  Rs  $\wedge$  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  $\implies$  graph_homomorphism (fst R) (S i) f  $\implies$   $\exists$  j.
extensible R (S j) f"
  shows "fair_chain Rs S"
<proof>
```

```

lemma fair_chainD:
  assumes "fair_chain Rs S"
  shows "chain S"
        "R ∈ Rs ⇒ graph_homomorphism (fst R) (S i) f ⇒ ∃ j. extensible
R (S j) f"
  ⟨proof⟩

```

```

lemma find_graph_occurence_vertices:
  assumes "chain S" "finite V" "univalent f" "f `` V ⊆ vertices (chain_sup
S)"
  shows "∃ i. f `` V ⊆ vertices (S i)"
  ⟨proof⟩

```

```

lemma find_graph_occurence_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)"
  ⟨proof⟩

```

```

lemma find_graph_occurence:
  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⟩

```

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 ∈ Rs ⇒ subgraph (fst R) (snd R)
∧ finite_graph (fst R)"
  shows "consequence_graph Rs (chain_sup S)"
  ⟨proof⟩

```

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 ⇒ ('a, 'c) Graph_PreRule ⇒ ('a, 'b) labeled_graph ⇒
('a, 'b) labeled_graph ⇒
  ('c × 'b) set ⇒ ('c × 'b) set ⇒ bool" where
"weak_universal _ R G1 G2 f1 f2 ≡ (∀ h1 h2 G::('a, 'x) labeled_graph.
  (graph_homomorphism (snd R) G h1 ∧ graph_homomorphism G1
G h2 ∧ f1 ∘ h2 ⊆ h1)
  → (∃ h. graph_homomorphism G2 G h ∧ h2 ⊆ h))"

```

```

lemma weak_universalD[dest]:
  assumes "weak_universal (t:: 'x itself) R (G1::('a, 'b) labeled_graph)
  G2 f1 f2"
  shows " $\bigwedge h_1 h_2 G::('a, 'x)$  labeled_graph.
         graph_homomorphism (snd R) G h1  $\implies$  graph_homomorphism G1 G h2
 $\implies f_1 \circ h_2 \subseteq h_1$ 
          $\implies (\exists h. \text{graph\_homomorphism } G_2 \ G \ h \ \wedge \ h_2 \subseteq h)$ "
  <proof>

```

```

lemma weak_universalI[intro]:
  assumes " $\bigwedge h_1 h_2 G::('a, 'x)$  labeled_graph.
         graph_homomorphism (snd R) G h1  $\implies$  graph_homomorphism G1 G h2
 $\implies f_1 \circ h_2 \subseteq h_1$ 
          $\implies (\exists h. \text{graph\_homomorphism } G_2 \ G \ h \ \wedge \ h_2 \subseteq h)$ "
  shows "weak_universal (t:: 'x itself) R (G1::('a, 'b) labeled_graph)
  G2 f1 f2"
  <proof>

```

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  $\wedge$ 
( $\exists f_1 f_2. \text{graph\_homomorphism (fst R) } G_1 \ f_1 \ \wedge$ 
  graph_homomorphism (snd R) G2 f2  $\wedge$ 
  f1  $\subseteq$  f2  $\wedge$ 
  weak_universal t R G1 G2 f1 f2)"
)

```

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
 $\wedge (\forall i. (\text{graph (S i)} \ \wedge \ S \ i = S \ (\text{Suc } i)) \vee (\exists R \in Rs. \text{pushout\_step}$ 
  t R (S i) (S (Suc i))))"

```

```

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

```

```

lemma Simple_WPC_Chain[simp]:
  assumes "Simple_WPC t Rs S"
  shows "chain S"
  <proof>

```

Definition 14, second part.

```

inductive WPC ::
  "'x itself  $\Rightarrow$  (('a, 'b) Graph_PreRule) set  $\Rightarrow$  (('a, 'd) graph_seq)
 $\Rightarrow$  bool"
  where
    wpc_simpl [simp, intro]: "Simple_WPC t Rs S  $\Longrightarrow$  WPC t Rs S"
    | wpc_combo [simp, intro]: "chain S  $\Longrightarrow$  ( $\bigwedge$  i.  $\exists$  S'. S' 0 = S i  $\wedge$  chain_sup
S' = S (Suc i)  $\wedge$  WPC t Rs S')  $\Longrightarrow$  WPC t Rs S"

```

```

lemma extensible_from_chainI:
  assumes ch:"chain S"
  and igh:"graph_homomorphism (S 0) C f"
  and ind:" $\bigwedge$  f i. graph_homomorphism (S i) C f  $\Longrightarrow$ 
 $\exists$  h. (graph_homomorphism (S (Suc i)) C h)  $\wedge$  agree_on (S
i) f h"
  shows "extensible (S 0, chain_sup S) C f"
  <proof>

```

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)"
  <proof>

```

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)"
  <proof>

```

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.. $\text{fst X}$ }"

```

```

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"
  <proof>

definition worklist :: "nat  $\times$  ('a  $\times$  nat  $\times$  nat) set
   $\Rightarrow$  (('a, 'b) labeled_graph  $\times$  ('a, 'b) labeled_graph) set
   $\Rightarrow$  (nat  $\times$  ('a, 'b) Graph_PreRule  $\times$  ('b  $\times$  nat) set) set"

where
  "worklist G Rs  $\equiv$  let G = graph_of G
  in {(N,R,f). R  $\in$  Rs  $\wedge$  graph_homomorphism (fst R) G f  $\wedge$  N = nextMax (Range
  f)
   $\wedge$   $\neg$  extensible R G f }"

definition valid_selection where
  "valid_selection Rs G R f  $\equiv$ 
  let wl = worklist G Rs in
  (nextMax (Range f), R,f)  $\in$  wl  $\wedge$ 
  ( $\forall$  (N,_)  $\in$  wl. N  $\geq$  nextMax (Range f))  $\wedge$ 
  graph_rule R"

lemma valid_selection_exists:
  assumes "worklist G Rs  $\neq$  {}"
  "set_of_graph_rules Rs"
  shows " $\exists$  L R f. valid_selection Rs G R f"
  <proof>

definition valid_selector where
  "valid_selector Rs selector  $\equiv$   $\forall$  G.
  (worklist G Rs  $\neq$  {}  $\longrightarrow$  ( $\exists$  (R,f)  $\in$  UNIV. selector G = Some (R,f)
   $\wedge$  valid_selection Rs G R f))  $\wedge$ 
  (worklist G Rs = {}  $\longrightarrow$  selector G = None)"

lemma valid_selectorD[dest]:
  assumes "valid_selector Rs selector"
  shows "worklist G Rs = {}  $\longleftrightarrow$  selector G = None"
  "selector G = Some (R,f)  $\implies$  valid_selection Rs G R f"
  <proof>

```

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)"
  <proof>
```

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))"
  <proof>
```

```
lemma nextMax_Un_eq[simp]:
"finite x  $\implies$  finite y  $\implies$  nextMax (x  $\cup$  y) = max (nextMax x) (nextMax
y)"
  <proof>
```

```
lemma extend:
  assumes "graph_homomorphism (fst R) (LG E {0.. $n$ } f) "graph_rule R"
  defines "g  $\equiv$  extend n R f"
  defines "G'  $\equiv$  LG ((on_triple g `` (edges (snd R)))  $\cup$  E) {0.. $\max n$  (nextMax
(Range g))}"
  shows "graph_homomorphism (snd R) G' g" "agree_on (fst R) f g" "f  $\subseteq$ 
g"
    "subgraph (LG E {0.. $n$ } G'"
    "weak_universal (t:: 'x itself) R (LG E {0.. $n$ } G' f g)"
  <proof>
```

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  $\equiv$  graph_of G''"
  assumes "graph G"
  defines "g  $\equiv$  extend (fst G'') R f"
  defines "G'  $\equiv$  LG (on_triple g `` edges (snd R)  $\cup$  (snd G'')) {0.. $\max$ 
(fst G'') (nextMax (Range g))}"
  shows "pushout_step (t:: 'x itself) R G G'"
  <proof>
```

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⟩
```

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⟩
```

lemma `inj_on_infinite`:

```
  assumes "infinite A" "inj_on f A" "range f ⊆ B"
  shows "infinite B"
⟨proof⟩
```

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⟩
```

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⟩
```

```

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⟩

```

Algorithm 1, abstractly.

```

abbreviation the_lcg where
"the_lcg sel Rs init ≡ chain_sup (λi. graph_of (mk_chain sel Rs init
i))"

```

```

lemma mk_chain_edges:
  assumes "valid_selector Rules sel"
    "⋃ ((edges o snd) ‘ Rules) ⊆ L × UNIV"
    "edges (graph_of G) ⊆ L × UNIV"
  shows "edges (graph_of (mk_chain sel Rules G i)) ⊆ L × UNIV"
⟨proof⟩

```

```

lemma the_lcg_edges:
  assumes "valid_selector Rules sel"
    "fst ‘ (⋃ ((edges o snd) ‘ Rules)) ⊆ L" (is "fst ‘?fR ⊆ _")
    "fst ‘ snd G ⊆ L"
  shows "fst ‘ edges (the_lcg sel Rules G) ⊆ L"
⟨proof⟩

```

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⟩

```

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) ∈ edges G}"

```

```

lemma getRel_dom:
  assumes "graph G"
  shows "(a,b) ∈ getRel 1 G ⇒ a ∈ vertices G"
        "(a,b) ∈ getRel 1 G ⇒ b ∈ vertices G"
  ⟨proof⟩

lemma getRel_subgraph[simp]:
  assumes "(y, z) ∈ getRel 1 G" "subgraph G G'"
  shows "(y,z) ∈ getRel 1 G'" ⟨proof⟩

lemma getRel_homR:
  assumes "(y, z) ∈ getRel 1 G" "(y,u) ∈ f" "(z,v) ∈ f"
  shows "(u, v) ∈ getRel 1 (map_graph f G)"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma getRel_hom_map[simp]:
  assumes "graph G"
  shows "getRel 1 (map_graph_fn G f) = map_prod f f ‘ (getRel 1 G)"
  ⟨proof⟩

  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 $G S \equiv :G:[fst S] = :G:[snd S]$ "

notation holds (infix $\langle \models \rangle$ 55)

abbreviation subset_sentence where

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

notation subset_sentence (infix $\langle \sqsubseteq \rangle$ 60)

Lemma 1.

lemma sentence_iff[simp]:

" $G \models e_1 \sqsubseteq e_2 = (:G:[e_1] \subseteq :G:[e_2])$ " and

eq_as_subsets:

" $G \models (e_1, e_2) = (G \models e_1 \sqsubseteq e_2 \wedge G \models e_2 \sqsubseteq e_1)$ "

$\langle proof \rangle$

lemma map_graph_in[intro]:

assumes "graph G " "(a, b) $\in :G:[e]$ "

shows "($f a, f b$) $\in :map_graph_fn G f:[e]$ "

$\langle proof \rangle$

lemma semantics_subset_vertices:

assumes "graph A " **shows** " $:A:[e] \subseteq vertices A \times vertices A$ "

$\langle proof \rangle$

lemma semantics_in_vertices:

assumes "graph A " "(a, b) $\in :A:[e]$ "

shows " $a \in vertices A$ " " $b \in vertices A$ "

$\langle proof \rangle$

lemma map_graph_semantics[simp]:

assumes "graph A " and $i:inj_on f (vertices A)$ "

shows " $:map_graph_fn A f:[e] = map_prod f f ' (:A:[e])$ "

$\langle proof \rangle$

lemma graph_union_semantics:

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

$\langle proof \rangle$

lemma subgraph_semantics:

assumes "subgraph $A B$ " "(a, b) $\in :A:[e]$ "

shows "(a, b) $\in :B:[e]$ "

$\langle proof \rangle$

lemma graph_homomorphism_semantics:

assumes "graph_homomorphism $A B f$ " "(a, b) $\in :A:[e]$ " "(a, a') $\in f$ " "(b, b') $\in f$ "

shows "(a', b') $\in :B:[e]$ "

$\langle proof \rangle$

```

lemma graph_homomorphism_nonempty:
  assumes "graph_homomorphism A B f" ":A:[e] ≠ {}"
  shows ":B:[e] ≠ {}"
⟨proof⟩

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⟩

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 ≡ A_Lbl S_Bot"
abbreviation "a_top ≡ A_Lbl S_Top"
abbreviation "a_idt ≡ A_Lbl S_Idt"
notation a_bot (<⊥>)
notation a_top (<⊤>)
notation a_idt (<1>)

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

abbreviation ident_rel where
"ident_rel idt G ≡ getRel idt G = (λ x.(x,x)) ' vertices G"

lemma ident_relI[intro]:
  assumes min:"∧ x. x ∈ vertices G ⇒ (x,x) ∈ getRel idt G"
  and max1:"∧ x y. (x,y) ∈ getRel idt G ⇒ x = y"
  and max2:"∧ x y. (x,y) ∈ getRel idt G ⇒ x ∈ vertices G"
  shows "ident_rel idt G"
⟨proof⟩

  Definition 4, generically.

definition standard :: "('l × 'v) set ⇒ 'l ⇒ 'l ⇒ 'l ⇒ ('l, 'v) labeled_graph
⇒ bool" where

```

```

"standard C b t idt G
  ≡ G = restrict G
  ∧ vertices G ≠ {}
  ∧ ident_rel idt G
  ∧ getRel b G = {}
  ∧ getRel t G = {(x,y). x∈vertices G ∧ y∈vertices G}
  ∧ (∀ (l,v) ∈ C. getRel l G = {(v,v)})"

```

Definition 4.

```

abbreviation standard' :: "'v set ⇒ ('v,'a) std_graph ⇒ bool" where
"standard' C ≡ standard ((λ c. (S_Const c,Inl c)) ' C) S_Bot S_Top S_Idt"

```

Definition 5.

```

definition model :: "'v set ⇒ ('v,'a) std_graph ⇒ ('v std_sentence) set
⇒ bool" where
"model C G T ≡ standard' C G ∧ (∀ S ∈ T. G ⊨ S)"

```

Definition 5.

```

abbreviation consistent :: "'b itself ⇒ 'v set ⇒ ('v std_sentence) set
⇒ bool" where
"consistent _ C T ≡ ∃ (G::('v,'b) std_graph). model C G T"

```

Definition 6.

```

definition entails :: "'b itself ⇒ 'v set ⇒ ('v std_sentence) set ⇒
'v std_sentence ⇒ bool" where
"entails _ C T S ≡ ∀ (G::('v,'b) std_graph). model C G T → G ⊨ S"

```

```

lemma standard_top_not_bot[intro]:
"standard' C G ⇒ :G:[⊥] ≠ :G:[⊤]"
  <proof>

```

Lemma 2.

```

lemma consistent_iff_entails_nonsense:
"consistent t C T = (¬ entails t C T (⊥,⊤))"
  <proof>

```

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 ⇒ ('c, nat) labeled_graph" where
"translation (A_Lbl l) = LG {(1,0,1)} {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..<card r} = r ∧ {0,1}⊆r"
```

lemma *inv_translationI4*[intro]:

```

assumes "finite r" "∧ x. x < card r ⇒ x ∈ r"
shows "r={0..<card r}"
⟨proof⟩

```

lemma *inv_translationI*[intro!]:

```

assumes "finite r" "∧ x. x < card r ⇒ x ∈ r" "0 ∈ r" "Suc 0 ∈ r"
shows "inv_translation r"
⟨proof⟩

```

lemma *verts_in_translation_finite*[intro]:

```

"finite (vertices (translation X))"
"finite (edges (translation X))"
"0 ∈ vertices (translation X)"
"Suc 0 ∈ vertices (translation X)"
⟨proof⟩

```

lemma *inv_tr_card_min*:

```

assumes "inv_translation r"
shows "card r ≥ 2"
⟨proof⟩

```

lemma *verts_in_translation*[intro]:

```

"inv_translation (vertices (translation X))"
⟨proof⟩

```

lemma *translation_graph*[intro]:

```

"graph (translation X)"
⟨proof⟩

```

lemma *graph_rule_translation*[intro]:

```

"graph_rule (translation X, translation (A_Int X Y))"
⟨proof⟩

```

```

lemma graph_hom_translation[intro]:
  "graph_homomorphism (LG {} {0,1}) (translation X) (Id_on {0,1})"
  ⟨proof⟩

lemma translation_right_to_left:
  assumes f:"graph_homomorphism (translation e) G f" "(0, x) ∈ f" "(1,
y) ∈ f"
  shows "(x, y) ∈ :G:[e]"
  ⟨proof⟩

lemma translation_homomorphism:
  assumes "graph_homomorphism (translation e) G f"
  shows "f `` {0} × f `` {1} ⊆ :G:[e]" ":G:[e] ≠ {}"
  ⟨proof⟩

  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⟩

abbreviation transl_rule ::
  "'a sentence ⇒ ('a, nat) labeled_graph × ('a, nat) labeled_graph"
where
  "transl_rule R ≡ (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 ⟷
G ⊨ eL ⊆ eR" (is "?rhs = ?lhs")
  ⟨proof⟩

lemma translation_self[intro]:
  "(0, 1) ∈ :translation e:[e]"
  ⟨proof⟩

  Lemma 6 is only used on rules of the form eL ⊆ eR. The requirement
of G being a graph can be dropped for one direction.

lemma maintained_holds[intro]:
  assumes ":G:[eL] ⊆ :G:[eR]"
  shows "maintained (transl_rule (eL ⊆ eR)) G"
  ⟨proof⟩

lemma maintained_holds_subset_iff[simp]:
  assumes "graph G"

```

```

shows "maintained (transl_rule (e_L  $\sqsubseteq$  e_R)) G  $\longleftrightarrow$  (:G:[e_L]  $\subseteq$  :G:[e_R])"
<proof>

```

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] = {}  $\longleftrightarrow$  ( $\forall$  (l',x,y) $\in$ edges G. l'  $\neq$  1)"
<proof>

```

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 :: "'l  $\Rightarrow$  ('l,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>

```

Definition 18.

```

definition nonempty_rule :: "('l,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>

```

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)"
<proof>

```

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"
<proof>

```

```

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"
<proof>

```

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>

```

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>

lemma finite_identity_rules[intro]:
  assumes "finite L"
  shows "finite (identity_rules L)"
  <proof>

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>

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 "( $\lambda$  v. getRel l G `` {v}) respects (getRel I G)" (is "?g1")
    and "( $\lambda$  v. (getRel l G)-1 `` {v}) respects (getRel I G)" (is "?g2")
  <proof>

```

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 o 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)  $\in$  edges G  $\iff$  (l,f x,f y)  $\in$  edges G)"
  <proof>

```

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:" $\bigwedge$  x. x $\in$ vertices
G  $\implies$  (f o f) x = f x"
    and maint:"maintained r G"
  shows "maintained r (map_graph_fn G f)"
  <proof>

```

Definition 20.

```

definition const_exists where
  "const_exists c  $\equiv$  transl_rule ( $\top \sqsubseteq A\_Cmp (A\_Cmp \top (A\_Lbl (S\_Const c)))$ )
   $\top$ )"
definition const_exists_rev where
  "const_exists_rev c  $\equiv$  transl_rule ( $A\_Cmp (A\_Cmp (A\_Lbl (S\_Const c)) \top)$ 
  ( $A\_Lbl (S\_Const c) \sqsubseteq A\_Lbl (S\_Const c)$ )")
definition const_prop where
  "const_prop c  $\equiv$  transl_rule ( $A\_Lbl (S\_Const c) \sqsubseteq 1$ )"
definition const_disj where
  "const_disj c1 c2  $\equiv$  transl_rule ( $A\_Cmp (A\_Lbl (S\_Const c_1)) (A\_Lbl (S\_Const$ 
   $c_2)) \sqsubseteq \perp$ )"

```

```

lemma constant_rules:
  assumes "standard' C G" "c  $\in$  C"
  shows "maintained (const_exists c) G"
        "maintained (const_exists_rev c) G"
        "maintained (const_prop c) G"
        "c'  $\in$  C  $\implies$  c  $\neq$  c'  $\implies$  maintained (const_disj c c') G"
  <proof>

```

```

definition constant_rules where
  "constant_rules C  $\equiv$  const_exists ' C  $\cup$  const_exists_rev ' C  $\cup$  const_prop
  ' C
   $\cup$  {const_disj c1 c2 | c1 c2. c1  $\in$  C  $\wedge$  c2  $\in$  C  $\wedge$  c1
   $\neq$  c2}"

```

```

lemma constant_rules_graph_rule:
  assumes "x  $\in$  constant_rules C"
  shows "graph_rule x"
  <proof>

```

```

lemma finite_constant[intro]:
  assumes "finite C"
  shows "finite (constant_rules C)"
  <proof>

```

```

lemma standard_maintains_constant_rules:
  assumes "standard' C G" "R  $\in$  constant_rules C"
  shows "maintained R G"
  <proof>

```

```

lemma constant_rules_empty[simp]:
  "constant_rules {} = {}"
  <proof>

```

Definition 20, continued.

```

definition standard_rules :: "'a set  $\Rightarrow$  'a Standard_Constant set  $\Rightarrow$  (('a
  Standard_Constant, nat) labeled_graph  $\times$  ('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"
  ⟨proof⟩
```

lemma identity_rules_mono:

```
  assumes "C1 ⊆ C2"
  shows "identity_rules C1 ⊆ identity_rules C2"
  ⟨proof⟩
```

lemma standard_rules_mono:

```
  assumes "C1 ⊆ C2" "L1 ⊆ L2"
  shows "standard_rules C1 L1 ⊆ standard_rules C2 L2"
  ⟨proof⟩
```

lemma maintainedA_invmono:

```
  assumes "C1 ⊆ C2" "L1 ⊆ L2"
  shows "maintainedA (standard_rules C2 L2) G ⇒ maintainedA (standard_rules
C1 L1) G"
  ⟨proof⟩
```

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⟩
```

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⟩
```

lemma standard_maintains_identity_rules:

```
  assumes "standard' C G" "x ∈ identity_rules L"
  shows "maintained x G"
  ⟨proof⟩
```

lemma standard_maintains_rules:

```
  assumes "standard' C G"
  shows "maintainedA (standard_rules C L) G"
  ⟨proof⟩
```

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"
  ⟨proof⟩

  Lemma 8.
  This is a slightly stronger version of Lemma 8: we reason about main-
  tained 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⟩

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 ' ⋃ ((edges o snd) ' Rs) ∪ {S_Bot,S_Top,S_Idt} ∪ S_Const
' C;
  Rules = Rs ∪ (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 ' ⋃ ((edges o snd) ' Rs) ∪ {S_Bot,S_Top,S_Idt} ∪ S_Const
' C ∪ fst ' edges init;
  Rules = Rs ∪ (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) ∈ :mdl:⌈snd R⌋ ∨ getRel S_Bot mdl ≠ {}"

```

```

definition transl_rules where
"transl_rules T = (⋃ (x, y)∈T. {(translation x, translation (A_Int x
y)), (translation y, translation (A_Int y x))})"

```

```

lemma gr_transl_rules:
"x ∈ transl_rules T ⇒ graph_rule x"
⟨proof⟩
term entails

```

```

lemma check_consistency:
  assumes "finite T" "finite C"
  shows "check_consistency C (transl_rules T) ⇔ consistent (t::nat
itself) C T"
  (is "?lhs = ?rhs")
⟨proof⟩

```

```

lemma check_entailment:
  assumes "finite T" "finite C"
  shows "check_entailment C (transl_rules T) S ⇔ entails (t::nat itself)
C T (fst S, (A_Int (fst S) (snd S)))"
  (is "?lhs = ?rhs")
⟨proof⟩

```

```

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. Amersand: Applying relation algebra in practice. In H. Swart, de, editor, *International Conference on Relational and Algebraic Methods in Computer Science*, pages 280–293. Springer, Berlin, Heidelberg, May 2011.