Logical foundations for reasoning about transformations of knowledge bases

Mohamed Chaabani Rachid Echahed Martin Strecker

July 23, 2013

Contents

Abstract

This is the formalization of a method of graph transformations. Transformations are expressed by an imperative programming language which is non-standard in the sense that it features conditions (in loops and selection statements) that are description logic (DL) formulas, and a non-deterministic assignment statement (a choice operator given by a DL formula). We sketch an operational semantics of the proposed programming language and then develop a matching Hoare calculus whose pre- and post-conditions are again DL formulas. A major difficulty resides in showing that the formulas generated when calculating weakest preconditions remain within the chosen DL fragment. In particular, this concerns substitutions whose result is not directly representable. We therefore explicitly add substitution as a constructor of the logic and show how it can be eliminated by an interleaving with the rules of a traditional tableau calculus.

1 Introduction

This is the formal proof document accompanying a shorter informal description [1].

The formalization is still ongoing, the present document is only a dump of the Isabelle theories, no major effort has been made to edit them. The outline is as follows:

- Sections 2 and 3 contain basic material, but are without deeper interest and should only be consulted for reference.
- Section 4 is the definition of the abstract syntax of the description logic $\mathcal{ALCQ}$.
- Section 5 contains definitions pertaining to variables, in particular bound variables, which are represented with de Bruijn indices, and functions for converting formulas to prenex normal form. Section 9 contains correspondings proofs.
- Section 6 is the definition of the semantics of the description logic $\mathcal{ALCQ}$, Section 8 contains corresponding proofs.
- Section 11 introduces operations for handling explicit substitutions, in particular eliminating them by pushing them down into constructors of the logic. Section 12 contains corresponding proofs, in particular showing that this elimination procedure terminates and is semantics preserving.
- Section 7 introduces the programming language for transforming graph structures.

- Section 10 is the definition of a Hoare logic for reasoning about these programs. This section contains a weakest-precondition calculus and shows the soundness of this calculus *wrt.* the semantics.

This formalization does not yet include a formalization of the tableau procedure described in Section 5 of [1]. We plan to adapt the method of [2].

2 Auxiliary definitions and lemmas

lemma *mp-obj* [*simp*]: $(A \wedge (A \longrightarrow B)) = (A \wedge B)$
by *blast*
lemma *mp-obj2* [*simp*]: $((A \longrightarrow B) \wedge A) = (A \wedge B)$
by *blast*

— Absorption rules
lemmas *absorb* = *Un-Int-distrib Un-Int-distrib2 Int-absorb1 Int-absorb2 Un-upper1 Un-upper2*

lemma *dom-empty-map-empty* [*simp*]: $(dom\ m = \{\}) = (m = empty)$
by *simp*

lemma *map-le-implies-ran-le*: $m \subseteq_m m' \Longrightarrow ran\ m \subseteq ran\ m'$
by (*fastforce simp add*: *dom-def ran-def map-le-def*)

lemma *image-dom*: m ‘ $dom\ m = Some$ ‘ $(ran\ m)$
by (*auto simp add*: *image-def dom-def ran-def*)

lemma *image-dom-ran*: $(the \circ m)$ ‘ $dom\ m = ran\ m$
by (*auto simp add*: *dom-def ran-def image-def*)

lemma *finite-dom-ran*: $finite\ (dom\ m) \Longrightarrow finite\ (ran\ m)$
by (*simp add*: *image-dom-ran* [*THEN sym*])

lemma *Some-the-map* [*simp*]: $x \in dom\ m \Longrightarrow Some\ (the\ (m\ x)) = m\ x$
by (*clarsimp simp add*: *dom-def*)

lemma *ran-map-upd-notin-dom*: $a \notin dom\ m \Longrightarrow ran\ (m(a \mapsto b)) = ran\ m \cup \{b\}$
by *auto*

lemma *ran-map-upd-subset*: $ran\ (m(a \mapsto b)) \subseteq ran\ m \cup \{b\}$
by (*auto simp add*: *ran-def*)

lemma *image-fun-upd*:
$(m$ ‘ $((f(x{:=}y))\ e)) = (if\ e = x\ then\ m$ ‘ $y\ else\ m$ ‘ $(f\ e))$

by (*clarsimp simp add*: *fun-upd-def image-def*)

definition
— injective map
inj-map :: ($'a \Rightarrow 'b$ *option*) $\Rightarrow$ *bool* **where**
inj-map m == *inj-on m* (*dom m*)

lemma *inj-map-empty* [*simp*]: *inj-map empty*
by (*simp add*: *inj-map-def*)

lemma *subset-inj-map*: $[\![$ *inj-map m*; $m' \subseteq_m m$ $]\!] \Longrightarrow$ *inj-map* m'
apply (*frule map-le-implies-dom-le*)
apply (*simp add*: *inj-map-def map-le-def inj-on-def*)
apply *blast*
done

lemma *inj-on-the*: *inj-map gm* $\Longrightarrow$ *inj-on* (*the* $\circ$ *gm*) (*dom gm*)
apply (*clarsimp simp add*: *inj-map-def inj-on-def*)
apply (*rename-tac x y v*)
apply (*drule-tac x=x* **in** *bspec*) **apply** *fastforce*
apply (*drule-tac x=y* **in** *bspec*) **apply** *fastforce*
apply *simp*
done

lemma *inj-on-map-upd*:
$a \notin$ *dom m* $\Longrightarrow$ *inj-on* ($m(a \mapsto b)$) (*dom m*) = *inj-on m* (*dom m*)
apply (*rule iffI*)

apply (*clarsimp simp add*: *inj-on-def*)
apply (*drule-tac bspec*) **prefer** *2*
apply (*drule-tac bspec*) **prefer** *2*
apply (*drule mp*) **prefer** *2*
apply *assumption*
apply (*auto simp add*: *inj-on-def*)
done

lemma *inj-map-map-upd*: $a \notin$ *dom m* $\Longrightarrow$ *inj-map* ($m(a \mapsto b)$) = ($b \notin$ *ran m* $\wedge$ *inj-map m*)
by (*simp add*: *inj-map-def inj-on-map-upd image-dom*) *auto*

definition
inv-m :: ($'a \Rightarrow 'b$ *option*) $\Rightarrow$ ($'b \Rightarrow 'a$ *option*) **where**
inv-m m == λ *y*. *if* ($y \in$ *ran m*) *then Some* (*SOME x*. *m x* = *Some y*) *else None*

```
lemma inv-m-empty [simp]: inv-m empty = empty
by (simp add: inv-m-def)

lemma dom-inv-m [simp]: dom (inv-m m) = ran m
by (simp add: inv-m-def ran-def dom-def)

lemma ran-inv-m [simp]: ran (inv-m m) ⊆ dom m
by (simp add: inv-m-def ran-def dom-def) (fast intro: someI2)

lemma restrict-map-le [simp]: m |` A ⊆m m
by (simp add: map-le-def)

lemma restrict-map-le-in-dom: ⟦ m ⊆m m'; dom m ⊆ A ⟧ ⟹ m ⊆m m' |` A
by (fastforce simp add: map-le-def )

lemma restrict-map-Some: ((m |` A) x = Some y) = (m x = Some y ∧ x ∈ A)
by (simp add: restrict-map-def)


lemma inv-m-map-upd: ⟦ a ∉ dom m; inj-map (m(a ↦ b))⟧ ⟹
  inv-m (m(a ↦ b)) = (inv-m m)(b ↦ a)
apply (simp add: inv-m-def ran-map-upd-notin-dom)
apply (simp add: fun-eq-iff)
apply (intro allI impI conjI)

apply (simp add: inj-map-map-upd)
defer
apply (simp add: inj-map-map-upd)
apply (rule some-equality) apply simp apply (simp add: ran-def)

apply (rule-tac f=Eps in arg-cong)
apply (simp add: fun-eq-iff)
apply auto
done

lemma o-m-inv-m-reduce:
  ⟦ a ∉ dom m2; inj-map (m2(a ↦ b))⟧ ⟹
  m1 ∘m (inv-m (m2(a ↦ b))) = (m1 ∘m inv-m m2)(b:= m1 a)
apply (simp add: inv-m-map-upd)
apply (simp add: map-comp-def)
apply (simp add: fun-eq-iff)
done



lemma restrict-map-le-eq: ((m|`A) ⊆m (m|`B)) = (dom m ∩ A ⊆ B)
apply (rule iffI)
apply (frule map-le-implies-dom-le) apply (fastforce simp only: dom-restrict)
```

apply (*simp add*: *map-le-def*) **apply** (*fastforce simp add*: *restrict-map-def*)
done

lemma *restrict-map-dom* [*simp*]: $m \subseteq_m m' \Longrightarrow m' |` (dom\ m) = m$
apply (*rule map-le-antisym*)
apply (*simp add*: *restrict-map-def map-le-def dom-def*)
apply (*erule restrict-map-le-in-dom*) **apply** *simp*
done

lemma *restrict-map-dom-subset*: $dom\ m \subseteq A \Longrightarrow m |` A = m$
apply (*rule map-le-antisym*)
apply *simp*
apply (*simp add*: *restrict-map-le-in-dom*)
done

lemma *ran-restrict-iff*: $(y \in ran\ (m |` A)) = (\exists x \in A.\ m\ x = Some\ y)$
apply (*rule iffI*)
apply (*erule ran-restrictD*)
apply (*auto simp add*: *ran-def restrict-map-def*)
done

lemma *Field-insert* [*simp*]: $Field\ (insert\ (a,a')\ A) = \{a,a'\} \cup Field\ A$
by (*fastforce simp add*: *Field-def Domain-unfold Domain-converse* [*symmetric*])

lemma *finite-Image*: $finite\ (Field\ R) \Longrightarrow finite\ (R\ `` S)$
apply (*rule finite-subset* [**where** *B*=*Range R*])
apply (*auto simp add*: *Field-def*)
done

lemma *Image-Field-subset* [*simp*]: $(R\ `` S) \subseteq Field\ R$
by (*auto simp add*: *Field-def*)

lemma *Field-converse* [*simp*]: $Field\ (R\hat{}{-1}) = Field\ R$
by (*auto simp add*: *Field-def*)

lemma *Field-Un*: $Field\ (R \cup S) = (Field\ R \cup Field\ S)$
by (*auto simp add*: *Field-def*)

lemma *Field-prod* [*simp*]: $Field\ (A \times A) = A$
by (*fastforce simp add*: *Field-def*)

lemma *Field-product-subset*: $(A \subseteq B \times B) = (Field\ A \subseteq B)$
by (*fastforce simp add*: *Field-def*)

lemma *Field-Diff-subseteq*: $Field\ (R - S) \subseteq Field\ R$
by (*auto simp add*: *Field-def*)

lemma *in-Field*: $(a,\ b) \in R \Longrightarrow \{a,b\} \subseteq \mathit{Field}\ R$
by (*fastforce simp add*: *Field-def Domain-unfold Domain-converse* [*symmetric*])

lemma *converse-empty-set* [*simp*]: $\{\}^{-1} = \{\}$
by (*simp add*: *converse-unfold*)

lemma *converse-insert*: $(\mathit{insert}\ (x,\ y)\ R)^{-1} = \mathit{insert}\ (y,\ x)\ (R^{-1})$
by (*unfold insert-def*) (*auto simp add*: *converse-Un*)

lemma *converse-Diff*: $(R - S)^{-1} = R^{-1} - S^{-1}$
by *auto*

lemma *Diff-Image*: $(R - S)$ `` $\{x\} = R$ `` $\{x\} - S$ `` $\{x\}$
by *blast*

lemma *finite-rel-finite-Field*: $[\![$ *Field* $R \subseteq A$; *finite* A $]\!] \Longrightarrow$ *finite* R
apply (*subgoal-tac* $R \subseteq A \times A$)
apply (*rule finite-subset*) **apply** *assumption+*
apply (*fast intro*: *finite-cartesian-product*)
apply (*fastforce simp add*: *Field-def*)
done

lemma *insert-Image-split*:
 ((*insert* $(a,\ b)$ R) `` $\{c\}$) = ((*if* $a = c$ *then* $\{b\}$ *else* $\{\}$) $\cup$ ($R$ `` $\{c\}$))
by *auto*

lemma *Image-rel-empty* [*simp*]: $\{\}$ `` $A = \{\}$
by *auto*

lemma *converse-mono*: $A \subseteq B \Longrightarrow (A)^{-1} \subseteq (B)^{-1}$
by *auto*

definition *rel-restrict* :: $('a \times 'b)$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ $('a \times 'b)$ *set* **where**
 rel-restrict $R\ A\ B = (R \cap (A \times B))$

definition *dom-of-range-restrict* :: $('a \times 'b)$ *set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ $'a$ *set* **where**
 dom-of-range-restrict $R\ C$ = (*Domain* (*rel-restrict* R *UNIV* C))

lemma *dom-of-range-restrict-expanded*: *dom-of-range-restrict* $R\ C = \{\ x.\ \exists\, y.\ ((x,y) \in R \wedge y \in C)\}$
by (*fastforce simp add*: *dom-of-range-restrict-def rel-restrict-def*)

lemma *rel-restrict-diff*: *rel-restrict* $(R1 - R2)\ A\ B$ = (((*rel-restrict* $R1$) $A\ B$) $-$ ((*rel-restrict* $R2$) $A\ B$))

by (*simp add*: *rel-restrict-def*) *blast*

lemma *rel-restrict-insert*: *rel-restrict* (*insert* (*x*, *y*) *R*) *A B* =
(*if* (*x* ∈ *A* ∧ *y* ∈ *B*) *then* (*insert* (*x*, *y*) (*rel-restrict R A B*)) *else* (*rel-restrict R A B*))
by (*simp add*: *rel-restrict-def*)

lemma *rel-restrict-empty* [*simp*]: *rel-restrict* {} *A B* = {}
by (*simp add*: *rel-restrict-def*)

lemma *rel-restrict-remove*: *b* ∈ *B* ⟹ (*a*, *b*) ∈ *R* ⟹
Range ((*rel-restrict R* {*a*} *B*) − {(*a*, *b*)}) = ((*Range* (*rel-restrict R* {*a*} *B*)) − {*b*})
by (*simp add*: *rel-restrict-def*) *blast*

lemma {(*x*, *y*). *x* ∈ *A* ∧ ((*x*,*y*) ∈ *R* ∧ *y* ∈ *C*)} = (*R* ∩ (*A* × *C*))
by *blast*

lemma {*y*. ∃ *x* ∈ *A*. ((*x*,*y*) ∈ *R* ∧ *y* ∈ *C*)} = *Range* (*R* ∩ (*A* × *C*))
by *blast*

lemma {*y*. ((*x*,*y*) ∈ *R* ∧ *y* ∈ *C*)} = *Range* (*R* ∩ ({*x*} × *C*))
by *blast*

definition *restrict-rel* :: [′*a rel*, ′*a set*] ⇒ ′*a rel* **where**
restrict-rel R A = *R* ∩ *A* × *A*

lemma *Field-restrict-rel* [*simp*]: *Field* (*restrict-rel R A*) ⊆ *A*
by (*auto simp add*: *restrict-rel-def Field-def*)

lemma *restrict-rel-empty* [*simp*]: *restrict-rel* {} *A* = {}
by (*simp add*: *restrict-rel-def*)

lemma *restrict-rel-insert*:
(*restrict-rel* (*insert* (*x*,*y*) *R*) *A*) =
(*if* (*x* ∈ *A* ∧ *y* ∈ *A*) *then* {(*x*, *y*)} *else* {}) ∪ (*restrict-rel R A*)
by (*simp add*: *restrict-rel-def*)

lemma *restrict-rel-insert2*:
(*restrict-rel* (*insert* (*x*,*y*) *R*) *A*) = (({(*x*, *y*)} ∩ *A* × *A*) ∪ (*restrict-rel R A*))
by (*simp add*: *restrict-rel-def*) *blast*

lemma *restrict-rel-insert-dom*:

restrict-rel R A = R ⟹ restrict-rel R (insert a A) = R
by (*fastforce simp add*: *restrict-rel-def*)

lemma *restrict-relD* [*dest*]:
(a, a′) ∈ restrict-rel R A ⟹ (a,a′) ∈ R ∧ a ∈ A ∧ a′ ∈ A
by (*simp add*: *restrict-rel-def*)

lemma *restrict-rel-Diff*: *restrict-rel (r − s) A = (restrict-rel r A) − (restrict-rel s A)*
by (*fastforce simp add*: *restrict-rel-def*)

lemma *restrict-rel-Un* [*simp*]:
restrict-rel (R ∪ S) A = restrict-rel R A ∪ restrict-rel S A
apply (*simp add*: *restrict-rel-def*)
apply *blast*
done

lemma *restrict-rel-mono*: *R ⊆ R′ ⟹ A ⊆ A′ ⟹ restrict-rel R A ⊆ restrict-rel R′ A′*
by (*fastforce simp add*: *restrict-rel-def*)

lemma *restrict-rel-Field-subset* [*simp*]:
Field R ⊆ A ⟹ restrict-rel R A = R
by (*simp add*: *restrict-rel-def Field-def*) *fast*

definition
fun-map-upd :: (′*a* => ′*b*) => (′*a* ~=> ′*b*) => (′*a* => ′*b*) **where**
fun-map-upd f m = (λk. case m k of None ⇒ f k | Some v ⇒ v)

lemma *fun-map-upd-empty* [*simp*]: *fun-map-upd f empty = f*
by (*simp add*: *fun-map-upd-def*)

lemma *fun-map-upd-upd* [*simp*]: *fun-map-upd f (m(x↦y)) = (fun-map-upd f m)(x:=y)*
by (*simp add*: *fun-map-upd-def fun-eq-iff*)

lemma *map-add-dom-disj1*: ⟦*dom tt1 ∩ dom tt2 = {}; tt1 b = Some ntp′* ⟧
⟹ *(tt1 ++ tt2) b = Some ntp′*
by (*auto simp add*: *map-add-Some-iff*)

lemma *map-add-dom-disj2*: ⟦*dom tt1 ∩ dom tt2 = {}; tt2 b = Some ntp′* ⟧
⟹ *(tt1 ++ tt2) b = Some ntp′*
by (*auto simp add*: *map-add-Some-iff*)

lemma *map-add-disj*:
⟦ *(m1 ++ m2) x = Some x′; dom m1 ∩ dom m2 = {}* ⟧ ⟹ *(m1 x = Some x′) ∨ (m2 x = Some x′)*

by *auto*

lemma *dom-map-comp*: *ran g* $\subseteq$ *dom f* $\Longrightarrow$ *dom* (*f o-m g*) = *dom g*
apply (*simp add*: *dom-def ran-def map-comp-def*)
apply (*auto split add*: *option.split-asm*)
done

lemma *map-add-image-ran*: *dom m2* = *A* $\Longrightarrow$(*the* $\circ$ *m1* ++ *m2*) ' *A* = *ran m2*
by (*clarsimp simp add*: *map-add-def image-def ran-def dom-def*) *auto*

lemma *map-add-image*: *dom m2* $\cap$ *A* = {} $\Longrightarrow$(*m1* ++ *m2*) ' *A* = (*m1*) ' *A*
by (*fastforce simp add*: *map-add-def image-def split*: *option.splits*)+

lemma *the-map-add-image*: *dom m2* $\cap$ *A* = {} $\Longrightarrow$(*the* $\circ$ (*m1* ++ *m2*)) ' *A* = (*the* $\circ$ *m1*) ' *A*
by (*simp add*: *image-compose map-add-image*)

lemma *image-Int-subset*: *A* $\subseteq$ *B* $\Longrightarrow$ *f* ' (*A* $\cap$ *B*) = *f* ' *A* $\cap$ *f* ' *B*
by (*fastforce simp add*: *image-def*)

lemma *dom-reduce-insert*:
  (*dom gm* = *insert a A*) = ($\exists$ *b gm'*. *gm* = *gm'*(*a*$\mapsto$*b*) $\wedge$ *dom gm'* = *A*)
apply (*rule iffI*)
apply (*rule-tac x*=*the* (*gm a*) **in** *exI*)
apply (*rule-tac x*=*gm*|' *A* **in** *exI*)

apply (*simp add*: *restrict-map-insert* [*THEN sym*] *restrict-map-dom-subset*)
apply *fastforce*
apply *clarsimp*
done

lemma *inj-map-pair* [*simp*]: *inj f* $\Longrightarrow$ *inj g* $\Longrightarrow$ *inj* (*map-pair f g*)
by (*simp add*: *map-pair-def inj-on-def*)

lemma *rtrancl-map-pair*: (*b0*, *c0*) $\in$ B^* $\Longrightarrow$ (*f b0*, *f c0*) $\in$ (*map-pair f f* ' *B*)*
apply (*induct b0 c0 rule*: *rtrancl.induct*)
apply *fast*
apply (*subgoal-tac* (*f b*, *f c*) $\in$ (*map-pair f f* ' *B*))
apply (*rule rtrancl-into-rtrancl*) **apply** *assumption*+
apply (*fastforce simp add*: *map-pair-def image-def*)
done

lemma *rtrancl-inclusion-map-pair*:
$A\hat{}* \subseteq B\hat{}* \Longrightarrow ((\textit{map-pair}\ f\ f)\ `\ A)\hat{}* \subseteq ((\textit{map-pair}\ f\ f)\ `\ B)\hat{}*$
apply *clarify*
apply (*rename-tac x y*)
apply (*erule-tac rtrancl.induct* [**where** $P = \lambda\ x\ y.\ (x, y) \in ((\textit{map-pair}\ f\ f)\ `\ B)\hat{}*$])
apply *fast*
apply (*subgoal-tac* $\exists\ b0\ c0.\ (b0, c0) \in A \wedge b = (f\ b0) \wedge c = (f\ c0)$)
prefer *2* **apply** (*fastforce simp add*: *map-pair-def image-def*)
apply *clarsimp*
apply (*subgoal-tac* $(b0, c0) \in B\hat{}*$) **prefer** *2* **apply** *blast*
apply (*fast intro*: *rtrancl-trans rtrancl-map-pair*)
done

definition
emorph :: $('a \Rightarrow 'b\ \textit{option}) \Rightarrow 'a \times 'a \Rightarrow 'b \times 'b$ **where**
$\textit{emorph}\ m = \textit{map-pair}\ (\textit{the} \circ m)\ (\textit{the} \circ m)$

lemma *emorph-pair* [*simp*]: $\textit{emorph}\ m\ (a, b) = (\textit{the}\ (m\ a), \textit{the}\ (m\ b))$
by (*simp add*: *emorph-def*)

lemma *emorph-converse*:
$\textit{emorph}\ gm\ `\ (R)^{-1} = (\textit{emorph}\ gm\ `\ R)^{-1}$
apply (*simp add*: *emorph-def converse-unfold image-def split-beta*)
apply (*rule Collect-cong*)
apply (*rule iffI*)
apply (*clarsimp*) **apply** (*rule bexI*) **prefer** *2* **apply** *assumption* **apply** *simp*
apply (*clarsimp*) **apply** *fast*
done

lemma *Field-emorph*: $\textit{Field}\ (\textit{emorph}\ m\ `\ R) = (\textit{the} \circ m)\ `\ (\textit{Field}\ R)$
by (*fastforce simp add*: *emorph-def image-def Field-def Domain-unfold Domain-converse* [*symmetric*])

lemma *rtrancl-inclusion-emorph*:
$A\hat{}* \subseteq B\hat{}* \Longrightarrow ((\textit{emorph}\ gm)\ `\ A)\hat{}* \subseteq ((\textit{emorph}\ gm)\ `\ B)\hat{}*$
by (*simp add*: *emorph-def rtrancl-inclusion-map-pair*)

lemma *inj-on-emorph*:
$\textit{inj-map}\ gm \Longrightarrow \textit{inj-on}\ (\textit{emorph}\ gm)\ (\textit{dom}\ gm \times \textit{dom}\ gm)$
by (*simp add*: *emorph-def map-pair-inj-on inj-on-the*)

lemma *rtrancl-id-trancl*: $R^* = \textit{Id} \cup R^+$
apply (*simp add*: *set-eq-iff rtrancl-eq-or-trancl*)
apply *blast*

done

3 Cardinalities of sets; finite and infinite sets

definition *card-le* :: *'a set* ⇒ *nat* ⇒ *bool* **where**
card-le B n == ((*finite B*) ∧ (*card B* < *n*))

definition *card-geq* :: *'a set* ⇒ *nat* ⇒ *bool* **where**
card-geq B n == ((¬ (*finite B*)) ∨ (*card B* ≥ *n*))

lemma *card-le-0* [*simp*]: *card-le B 0* = *False*
by (*simp add*: *card-le-def*)

lemma *card-geq-0* [*simp*]: *card-geq B 0*
by (*simp add*: *card-geq-def*)

lemma *not-card-le-card-geq* [*simp*]: (¬ (*card-le B n*)) = *card-geq B n*
apply (*simp add*: *card-geq-def card-le-def*)
apply *arith*
done

lemma *not-card-geq-card-le* [*simp*]: (¬ (*card-geq B n*)) = *card-le B n*
apply (*simp add*: *card-geq-def card-le-def*)
apply *arith*
done

lemma *empty-finite*: ∀ *a*. *a* ∉ *A* ⟹ *finite A*
by *auto*

lemma *non-finite-non-empty*: ¬ *finite A* ⟹ ∃ *a*. *a* ∈ *A*
apply (*insert empty-finite* [*of A*])
apply *blast*
done

lemma *card-le-Suc-insert*: *a* ∉ *B* ⟹ *card-le* (*insert a B*) (*Suc n*) = *card-le B n*
by (*auto simp add*: *card-le-def*)

lemma *card-geq-Suc-insert*: *a* ∉ *B* ⟹ *card-geq* (*insert a B*) (*Suc n*) = *card-geq B n*
by (*auto simp add*: *card-geq-def*)

lemma *card-le-Suc-diff*: *a* ∈ *B* ⟹ *card-le* (*B* − {*a*}) *n* = *card-le B* (*Suc n*)
apply (*insert card-le-Suc-insert* [*of a B* − {*a*} *n*])
apply (*subgoal-tac insert a B* = *B*)
apply *auto*
done

lemma *card-geq-Suc-diff*: $a \in B \Longrightarrow$ *card-geq* $(B - \{a\})$ n = *card-geq* B (*Suc* n)
apply (*insert card-geq-Suc-insert* [*of a B* $- \{a\}$ *n*])
apply (*subgoal-tac insert a B* = *B*)
apply *auto*
done

4 Syntax of $\mathcal{ALCQ}$

We now give details of the formal definition of the logic $\mathcal{ALC}$. The type of roles, defined by:

datatype *role-op* = *RDiff* | *RAdd*

datatype (*'nr*, *'nc*, *'ni*) *subst* =
  RSubst 'nr role-op (*'ni* ∗ *'ni*)
| *ISubst 'ni 'ni*

datatype *numres-ord* = *Le* | *Geq*

fun *dual-numres-ord* :: *numres-ord* ⇒ *numres-ord* **where**
  dual-numres-ord Le = *Geq*
| *dual-numres-ord Geq* = *Le*

datatype (*'nr*, *'nc*, *'ni*) *concept* =
  AtomC 'nc
| *Top*
| *Bottom*
| *NotC* ((*'nr*, *'nc*, *'ni*) *concept*)
| *AndC* ((*'nr*, *'nc*, *'ni*) *concept*) ((*'nr*, *'nc*, *'ni*) *concept*)
| *OrC* ((*'nr*, *'nc*, *'ni*) *concept*) ((*'nr*, *'nc*, *'ni*) *concept*)
| *NumRestrC* (*numres-ord*) (*nat*) *'nr* ((*'nr*, *'nc*, *'ni*) *concept*)
| *SubstC* ((*'nr*, *'nc*, *'ni*) *concept*) (*'nr*, *'nc*, *'ni*) *subst*

definition *SomeC* :: *'nr* ⇒ ((*'nr*, *'nc*, *'ni*) *concept*) ⇒ ((*'nr*, *'nc*, *'ni*) *concept*)
where
  SomeC r c = (*NumRestrC Geq 1 r c*)

definition *AllC* :: *'nr* ⇒ ((*'nr*, *'nc*, *'ni*) *concept*) ⇒ ((*'nr*, *'nc*, *'ni*) *concept*)
where
  AllC r c = (*NumRestrC Le 1 r* (*NotC c*))

datatype ('*nr*, '*nc*, '*ni*) *fact* =
    Inst ('*ni*) (('*nr*, '*nc*, '*ni*) *concept*)
  | *AtomR bool* ('*nr*) ('*ni*) ('*ni*)
  | *Eq bool* '*ni* '*ni*

datatype *binop* = *Conj* | *Disj*
datatype *quantif* = *QAll* | *QEx*

fun *dual-binop* :: *binop* ⇒ *binop* **where**
    dual-binop Conj = *Disj*
  | *dual-binop Disj* = *Conj*

fun *dual-quantif* :: *quantif* ⇒ *quantif* **where**
    dual-quantif QAll = *QEx*
  | *dual-quantif QEx* = *QAll*

datatype ('*nr*, '*nc*, '*ni*) *form* =
    FalseFm
  | *FactFm* (('*nr*, '*nc*, '*ni*) *fact*)
  | *NegFm* (('*nr*, '*nc*, '*ni*) *form*)
  | *BinopFm binop* (('*nr*, '*nc*, '*ni*) *form*) (('*nr*, '*nc*, '*ni*) *form*)
  | *QuantifFm quantif* (('*nr*, '*nc*, '*ni*) *form*)
  | *SubstFm* (('*nr*, '*nc*, '*ni*) *form*) ('*nr*, '*nc*, '*ni*) *subst*

abbreviation *TrueFm* :: (('*nr*, '*nc*, '*ni*) *form*) **where**
  TrueFm == (*NegFm FalseFm*)

abbreviation *ConjFm* :: (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) **where**
  ConjFm a b == (*BinopFm Conj a b*)
abbreviation *DisjFm* :: (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) **where**
  DisjFm a b == (*BinopFm Disj a b*)

definition *ImplFm* :: (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) **where**
  ImplFm a b = (*DisjFm* (*NegFm a*) *b*)
definition *IfThenElseFm* ::
  (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) ⇒ (('*nr*, '*nc*, '*ni*) *form*) **where**
  IfThenElseFm c a b = *ConjFm* (*ImplFm c a*) (*ImplFm* (*NegFm c*) *b*)

abbreviation *AllFm* :: (('nr, 'nc, 'ni) form) ⇒ (('nr, 'nc, 'ni) form) **where**
AllFm f == (*QuantifFm QAll f*)
abbreviation *ExFm* :: (('nr, 'nc, 'ni) form) ⇒ (('nr, 'nc, 'ni) form) **where**
ExFm f == (*QuantifFm QEx f*)

fun *univ-quantif* :: *bool* ⇒ (('nr, 'nc, 'ni) *form*) ⇒ *bool* **where**
univ-quantif pos FalseFm = *True*
| *univ-quantif pos* (*FactFm f*) = *True*
| *univ-quantif pos* (*NegFm f*) = (*univ-quantif* (¬ *pos*) *f*)
| *univ-quantif pos* (*BinopFm bop f1 f2*) = ((*univ-quantif pos f1*) ∧ (*univ-quantif pos f2*))
| *univ-quantif pos* (*QuantifFm q f*) = (((*pos* ∧ *q* = *QAll*) ∨ ((¬ *pos*) ∧ *q* = *QEx*)) ∧ *univ-quantif pos f*)
| *univ-quantif pos* (*SubstFm f sb*) = (*univ-quantif pos f*)

fun *quantif-free* :: (('nr, 'nc, 'ni) *form*) ⇒ *bool* **where**
quantif-free FalseFm = *True*
| *quantif-free* (*FactFm f*) = *True*
| *quantif-free* (*NegFm f*) = (*quantif-free f*)
| *quantif-free* (*BinopFm bop f1 f2*) = ((*quantif-free f1*) ∧ (*quantif-free f2*))
| *quantif-free* (*QuantifFm q f*) = *False*
| *quantif-free* (*SubstFm f sb*) = (*quantif-free f*)

end

5 Treatment of variables, in particular bound variables

datatype *'v var* =
Free 'v
| *Bound nat*

fun *shift-var* :: *nat* ⇒ *'ni var* ⇒ *'ni var* **where**
shift-var n (*Free w*) = *Free w*
| *shift-var n* (*Bound k*) = (*if n* ≤ *k then Bound* (*k* − *1*) *else Bound k*)

fun *lift-var* :: *nat* ⇒ *'v var* ⇒ *'v var* **where**
lift-var n (*Free v*) = *Free v*
| *lift-var n* (*Bound k*) = (*if n* ≤ *k then Bound* (*k* + *1*) *else Bound k*)

fun *lift-subst* :: *nat* ⇒ (*'nr*, *'nc*, *'ni var*) *subst* ⇒ (*'nr*, *'nc*, *'ni var*) *subst* **where**
lift-subst n (*RSubst r rop* (*v1*, *v2*)) = *RSubst r rop* (*lift-var n v1*, *lift-var n v2*)
| *lift-subst n* (*ISubst v v'*) = *ISubst* (*lift-var n v*) (*lift-var n v'*)

fun *lift-concept* :: *nat* ⇒ (*'nr*, *'nc*, *'ni var*) *concept* ⇒ (*'nr*, *'nc*, *'ni var*) *concept*
where
lift-concept n Bottom = *Bottom*
| *lift-concept n Top* = *Top*
| *lift-concept n* (*AtomC a*) = (*AtomC a*)
| *lift-concept n* (*AndC c1 c2*) = *AndC* (*lift-concept n c1*) (*lift-concept n c2*)
| *lift-concept n* (*OrC c1 c2*) = *OrC* (*lift-concept n c1*) (*lift-concept n c2*)
| *lift-concept n* (*NotC c*) = *NotC* (*lift-concept n c*)
| *lift-concept n* (*NumRestrC nro nb r c*) = (*NumRestrC nro nb r* (*lift-concept n c*))
| *lift-concept n* (*SubstC c sb*) = *SubstC* (*lift-concept n c*) (*lift-subst n sb*)

fun *lift-fact* :: *nat* ⇒ (*'nr*, *'nc*, *'ni var*) *fact* ⇒ (*'nr*, *'nc*, *'ni var*) *fact* **where**
lift-fact n (*Inst x c*) = (*Inst* (*lift-var n x*) (*lift-concept n c*))
| *lift-fact n* (*AtomR sign r x y*) = (*AtomR sign r* (*lift-var n x*) (*lift-var n y*))
| *lift-fact n* (*Eq sign x y*) = *Eq sign* (*lift-var n x*) (*lift-var n y*)

fun *lift-form* :: *nat* ⇒ (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *form* **where**
lift-form n FalseFm = *FalseFm*
| *lift-form n* (*FactFm fct*) = *FactFm* (*lift-fact n fct*)
| *lift-form n* (*NegFm f*) = *NegFm* (*lift-form n f*)
| *lift-form n* (*BinopFm bop f1 f2*) = *BinopFm bop* (*lift-form n f1*) (*lift-form n f2*)
| *lift-form n* (*QuantifFm q f*) = *QuantifFm q* (*lift-form* (*n+1*) *f*)
| *lift-form n* (*SubstFm f sb*) = (*SubstFm* (*lift-form n f*) (*lift-subst n sb*))

fun *shuffle-right* :: *binop* ⇒ (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *form* **where**
shuffle-right bop f1 (*QuantifFm q f2*) = *QuantifFm q* (*shuffle-right bop* (*lift-form 0 f1*) *f2*)
| *shuffle-right bop f1 f2* = *BinopFm bop f1 f2*

fun *shuffle-left* :: *binop* ⇒ (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *form* **where**
shuffle-left bop (*QuantifFm q f1*) *f2* = *QuantifFm q* (*shuffle-left bop f1* (*lift-form 0 f2*))
| *shuffle-left bop f1 f2* = *shuffle-right bop f1 f2*

fun *lift-bound-above-negfm* :: (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *form*
where
lift-bound-above-negfm (*QuantifFm q f*) = *QuantifFm* (*dual-quantif q*) (*lift-bound-above-negfm f*)
| *lift-bound-above-negfm f* = *NegFm f*

fun *lift-bound-above-substfm* ::

('nr, 'nc, 'ni var) form ⇒ ('nr, 'nc, 'ni var) subst ⇒ ('nr, 'nc, 'ni var) form
where
lift-bound-above-substfm (QuantifFm q f) sb = QuantifFm q (lift-bound-above-substfm f (lift-subst 0 sb))
| *lift-bound-above-substfm f sb = SubstFm f sb*

fun *lift-bound* :: ('nr, 'nc, 'ni var) form ⇒ ('nr, 'nc, 'ni var) form **where**
lift-bound FalseFm = FalseFm
| *lift-bound (FactFm fct) = FactFm fct*
| *lift-bound (NegFm f) = lift-bound-above-negfm (lift-bound f)*
| *lift-bound (BinopFm bop f1 f2) = shuffle-left bop (lift-bound f1) (lift-bound f2)*
| *lift-bound (QuantifFm q f) = QuantifFm q (lift-bound f)*
| *lift-bound (SubstFm f sb) = (lift-bound-above-substfm (lift-bound f) sb)*

definition *bind* :: quantif ⇒ 'ni ⇒ ('nr, 'nc, 'ni var) form ⇒ ('nr, 'nc, 'ni var) form **where**
bind q v f = QuantifFm q (SubstFm (lift-form 0 f) (ISubst (Free v) (Bound 0)))

6 Semantics of $\mathcal{ALCQ}$

typedecl *domtype*

The type *domtype* is the type of elements of the interpretation domain. Then The interpretation is defined as follows:

record ('nr, 'nc, 'ni) Interp =
idomain :: domtype set
interp-c :: 'nc ⇒ domtype set
interp-r :: 'nr ⇒ (domtype * domtype) set
interp-i :: 'ni ⇒ domtype

fun *interpRO* :: role-op ⇒ 'nr ⇒ ('ni * 'ni) ⇒ ('nr, 'nc, 'ni) Interp ⇒ (domtype * domtype) set **where**
interpRO RDiff r (x, y) i = (interp-r i r) − {(interp-i i x, interp-i i y)}
| *interpRO RAdd r (x, y) i = insert (interp-i i x, interp-i i y) (interp-r i r)*

fun *interp-numres-ord* :: numres-ord ⇒ 'a set ⇒ nat ⇒ bool **where**
interp-numres-ord Le = card-le
| *interp-numres-ord Geq = card-geq*

definition *interp-i-modif* :: 'ni ⇒ domtype ⇒ ('nr, 'nc, 'ni) Interp ⇒ ('nr, 'nc, 'ni) Interp **where**
interp-i-modif x xi i = i(|interp-i := (interp-i i)(x := xi) |)

definition *interp-r-modif* :: *'nr* ⇒ (*domtype* ∗ *domtype*) *set* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* **where**
interp-r-modif r ri i = *i*⦇*interp-r* := (*interp-r i*)(*r* := *ri*) ⦈

fun *interp-subst* :: (*'nr*, *'nc*, *'ni*) *subst* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* **where**
interp-subst (*RSubst r rop p*) *i* = (*interp-r-modif r* (*interpRO rop r p i*) *i*)
| *interp-subst* (*ISubst v v'*) *i* = (*interp-i-modif v* (*interp-i i v'*) *i*)

fun *interp-subst-closure* :: (*'nr*, *'nc*, *'ni*) *subst list* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* **where**
interp-subst-closure [] *i* = *i*
| *interp-subst-closure* (*sb* # *sbsts*) *i* = *interp-subst sb* (*interp-subst-closure sbsts i*)

fun *interp-concept* :: (*'nr*, *'nc*, *'ni*) *concept* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* ⇒ *domtype set* **where**
interp-concept Bottom i = {}
| *interp-concept Top i* = *UNIV*
| *interp-concept* (*AtomC a*) *i* = *interp-c i a*
| *interp-concept* (*AndC c1 c2*) *i* = (*interp-concept c1 i*) ∩ (*interp-concept c2 i*)
| *interp-concept* (*OrC c1 c2*) *i* = (*interp-concept c1 i*) ∪ (*interp-concept c2 i*)
| *interp-concept* (*NotC c*) *i* = − (*interp-concept c i*)
| *interp-concept* (*NumRestrC nro n r c*) *i* =
{*x* . *interp-numres-ord nro* (*Range* (*rel-restrict* (*interp-r i r*) {*x*} (*interp-concept c i*))) *n*}
| *interp-concept* (*SubstC c sb*) *i* = (*interp-concept c*) (*interp-subst sb i*)

fun *interp-fact* :: (*'nr*, *'nc*, *'ni*) *fact* ⇒ (*'nr*, *'nc*, *'ni*) *Interp* ⇒ *bool* **where**
interp-fact (*Inst x c*) *icr* = ((*interp-i icr x*) ∈ (*interp-concept c icr*))
| *interp-fact* (*AtomR sign r x y*) *icr* =
(*if sign*
then ((*interp-i icr x*, *interp-i icr y*) ∈ (*interp-r icr r*))
else ((*interp-i icr x*, *interp-i icr y*) ∉ (*interp-r icr r*)))
| *interp-fact* (*Eq sign x y*) *icr* =
(*if sign*
then ((*interp-i icr x*) = (*interp-i icr y*))
else ((*interp-i icr x*) ≠ (*interp-i icr y*)))

definition *interp-bound* :: *domtype* ⇒ (*'nr*, *'nc*, *'ni var*) *Interp* ⇒ (*'nr*, *'nc*, *'ni var*) *Interp* **where**
interp-bound xi i = *i*⦇*interp-i* := ((*interp-i i*) ∘ (*shift-var 0*))(*Bound 0* := *xi*) ⦈

fun *interp-form* :: ('*nr*, '*nc*, '*ni var*) *form* ⇒ ('*nr*, '*nc*, '*ni var*) *Interp* ⇒ *bool*
where
interp-form FalseFm i = *False*
| *interp-form* (*FactFm f*) *i* = *interp-fact f i*
| *interp-form* (*NegFm f*) *i* = (¬ (*interp-form f i*))
| *interp-form* (*BinopFm Conj f1 f2*) *i* = ((*interp-form f1 i*) ∧ (*interp-form f2 i*))
| *interp-form* (*BinopFm Disj f1 f2*) *i* = ((*interp-form f1 i*) ∨ (*interp-form f2 i*))
| *interp-form* (*QuantifFm QAll f*) *i* = (∀ *xi*. *interp-form f* (*interp-bound xi i*))
| *interp-form* (*QuantifFm QEx f*) *i* = (∃ *xi*. *interp-form f* (*interp-bound xi i*))
| *interp-form* (*SubstFm f sb*) *i* = (*interp-form f*) (*interp-subst sb i*)

definition *lift-impl a b* = (λ *s*. *a s* ⟶ *b s*)
definition *lift-ite c a b* = (λ *s*. *if c s then a s else b s*)

definition *validFm* :: (('*nr*, '*nc*, '*ni var*) *form*) ⇒ *bool* **where**
validFm f ≡ (∀ *i*. (*interp-form f i*))

definition *delete-edge* :: '*ni* ⇒ '*nr* ⇒ '*ni* ⇒ ('*nr*, '*nc*, '*ni*) *Interp* ⇒ ('*nr*, '*nc*, '*ni*) *Interp*
where *delete-edge v1 r v2 s* =
s (| *interp-r* := (*interp-r s*)(*r*:= (*interp-r s r*) − { (*interp-i s v1*, *interp-i s v2*) }) |)

definition *generate-edge* :: '*ni* ⇒ '*nr* ⇒ '*ni* ⇒ ('*nr*, '*nc*, '*ni*) *Interp* ⇒ ('*nr*, '*nc*, '*ni*) *Interp*
where *generate-edge v1 r v2 s* =
s (| *interp-r* := (*interp-r s*)(*r*:= *insert* (*interp-i s v1*, *interp-i s v2*) (*interp-r s r*)) |)

7 Programming language

datatype ('*r*, '*c*, '*i*) *stmt*
= *SKIP*
| *EDel* '*i* '*r* '*i*
| *EGen* '*i* '*r* '*i*
| *SelAss* '*i* (('*r*, '*c*, '*i var*) *form*)
| *Seq* ('*r*, '*c*, '*i*) *stmt* ('*r*, '*c*, '*i*) *stmt* (-;/ - [*60*, *61*] *60*)
| *If* (('*r*, '*c*, '*i var*) *form*) (('*r*, '*c*, '*i*) *stmt*) (('*r*, '*c*, '*i*) *stmt*) ((*IF* -/ *THEN* -/ *ELSE* -) [*0*, *0*, *61*] *61*)
| *While* (('*r*, '*c*, '*i var*) *form*) (('*r*, '*c*, '*i var*) *form*) (('*r*, '*c*, '*i*) *stmt*) ((*WHILE* {-}/ -/ *DO* -) [*0*, *0*, *61*] *61*)

fun *form-prop-in-stmt* :: (('nr, 'nc, 'ni var) form ⇒ bool) ⇒ ('nr, 'nc, 'ni) stmt ⇒ bool **where**
form-prop-in-stmt fp SKIP = True
| *form-prop-in-stmt fp (EDel v1 r v2) = True*
| *form-prop-in-stmt fp (EGen v1 r v2) = True*
| *form-prop-in-stmt fp (SelAss v b) = fp b*
| *form-prop-in-stmt fp* (c_1 ; c_2) = (*form-prop-in-stmt fp* c_1 ∧ *form-prop-in-stmt fp* c_2)
| *form-prop-in-stmt fp (IF b THEN* c_1 *ELSE* c_2) =
(*fp b* ∧ *form-prop-in-stmt fp* c_1 ∧ *form-prop-in-stmt fp* c_2)
| *form-prop-in-stmt fp (WHILE {iv} b DO c)* = (*fp iv* ∧ *fp b* ∧ *form-prop-in-stmt fp c*)

inductive
big-step :: ('r, 'c, 'i) stmt × ('r, 'c, 'i var) Interp ⇒ ('r, 'c, 'i var) Interp ⇒ bool (-⇒- [61,61]60)
where
Skip: (*SKIP*, s) ⇒ s |
EDel: s' = *delete-edge* (*Free v1*) r (*Free v2*) s ⟹ (*EDel v1 r v2*, s) ⇒ s' |
EGen: s' = *generate-edge* (*Free v1*) r (*Free v2*) s ⟹ (*EGen v1 r v2*, s) ⇒ s' |

SelAssTrue: ∃ vi. (s' = *interp-i-modif* (*Free v*) vi s ∧ *interp-form* b s') ⟹
(*SelAss v b*, s) ⇒ s' |

Seq: (c_1,s_1) ⇒ s_2 ⟹
(c_2,s_2) ⇒ s_3 ⟹
(c_1;c_2, s_1) ⇒ s_3 |

IfTrue: *interp-form* b s ⟹
(c_1,s) ⇒ t ⟹
(*IF b THEN* c_1 *ELSE* c_2, s) ⇒ t |
IfFalse: ¬ *interp-form* b s ⟹
(c_2,s) ⇒ t ⟹
(*IF b THEN* c_1 *ELSE* c_2, s) ⇒ t |

WhileFalse: ¬*interp-form* b s ⟹ (*WHILE {iv} b DO c*,s) ⇒ s |
WhileTrue: *interp-form* b s_1 ⟹
(c,s_1) ⇒ s_2 ⟹
(*WHILE {iv} b DO c*, s_2) ⇒ s_3 ⟹
(*WHILE {iv} b DO c*, s_1) ⇒ s_3

declare *big-step.intros* [*intro*]

lemmas *big-step-induct* = *big-step.induct*[*split-format*(*complete*)]

inductive-cases *SkipE*[*elim*!]: (*SKIP*,*s*) ⇒ *t*
inductive-cases *EDelE*[*elim*!]: (*EDel v1 r v2*, *s*) ⇒ *s'*
inductive-cases *EGenE*[*elim*!]: (*EGen v1 r v2*, *s*) ⇒ *s'*
inductive-cases *ESelAss*[*elim*!]: (*SelAss v b*, *s*) ⇒ *s'*
inductive-cases *SeqE*[*elim*!]: (*c1*;*c2*,*s1*) ⇒ *s3*
inductive-cases *IfE*[*elim*!]: (*IF b THEN c1 ELSE c2*,*s*) ⇒ *t*
inductive-cases *WhileE*[*elim*]: (*WHILE* {*iv*} *b DO c*,*s*) ⇒ *t*

8 Semantics (Proofs)

lemma *interp-c-interp-i-modif* [*simp*]:
  interp-c (*interp-i-modif r ri i*) *c* = *interp-c i c*
by (*simp add*: *interp-i-modif-def*)

lemma *interp-c-interp-r-modif* [*simp*]:
  interp-c (*interp-r-modif r ri i*) *c* = *interp-c i c*
by (*simp add*: *interp-r-modif-def*)

lemma *interp-i-interp-r-modif* [*simp*]:
  interp-i (*interp-r-modif r ri i*) *x* = *interp-i i x*
by (*simp add*: *interp-r-modif-def*)

lemma *interp-r-interp-i-modif* [*simp*]:
  interp-r (*interp-i-modif v vi i*) *r* = *interp-r i r*
by (*simp add*: *interp-i-modif-def*)

lemma *interp-i-interp-i-modif-eq* [*simp*]:
  interp-i (*interp-i-modif v v' i*) *v* = *v'*
by (*simp add*: *interp-i-modif-def*)

lemma *interp-i-interp-i-modif-neq* [*simp*]:
  $v \neq v'' \Longrightarrow$ *interp-i* (*interp-i-modif v v' i*) *v''* = (*interp-i i v''*)
by (*simp add*: *interp-i-modif-def*)

lemma *interp-r-interp-r-modif-eq* [*simp*]:
  (*interp-r* (*interp-r-modif r* (*interpRO rop r p i*) *i*) *r*) = *interpRO rop r p i*
by (*simp add*: *interp-r-modif-def*)

lemma *interp-r-interp-r-modif-neq* [*simp*]:
$r' \neq r \Longrightarrow$ (*interp-r* (*interp-r-modif r* (*interpRO rop r p i*) *i*) r') = *interp-r i* r'
by (*simp add*: *interp-r-modif-def*)

lemma *interp-r-modif-interp-r* [*simp*]: (*interp-r-modif r* (*interp-r i r*) *i*) = *i*
by (*simp add*: *interp-r-modif-def*)

lemma *interp-form-ImplFm* [*simp*]:
interp-form (*ImplFm f1 f2*) = (*lift-impl* (*interp-form f1*) (*interp-form f2*))
by (*simp add*: *ImplFm-def lift-impl-def fun-eq-iff*)

lemma *interp-form-IfThenElseFm* [*simp*]: *interp-form* (*IfThenElseFm c a b*) =
(*lift-ite* (*interp-form c*) (*interp-form a*) (*interp-form b*))
by (*simp add*: *IfThenElseFm-def lift-ite-def lift-impl-def fun-eq-iff*)

lemma (*Range* (*rel-restrict* (*interpR r i*) {*x*} (*interp-concept c i*))) = {*y*. ((*x*,*y*) ∈ (*interpR r i*) ∧ *y* ∈ (*interp-concept c i*))}
by (*simp add*: *rel-restrict-def*) *blast*

lemma *Bottom-NumRestrC*: *interp-concept Bottom i* = *interp-concept* (*NumRestrC Le 0 r c*) *i*
by *simp*

lemma *Top-NumRestrC*: *interp-concept Top i* = *interp-concept* (*NumRestrC Geq 0 r c*) *i*
by *simp*

lemma *NotC-NumRestrC-Le*: *interp-concept* (*NotC* (*NumRestrC Le n r c*)) *i* = *interp-concept* (*NumRestrC Geq n r c*) *i*
by (*simp add*: *set-eq-iff*)

lemma *NotC-NumRestrC-Geq*: *interp-concept* (*NotC* (*NumRestrC Geq n r c*)) *i* = *interp-concept* (*NumRestrC Le n r c*) *i*
by (*simp add*: *set-eq-iff*)

lemma *NotC-SubstC*: *interp-concept* (*NotC* (*SubstC c sb*)) *i* = *interp-concept* (*SubstC* (*NotC c*) *sb*) *i*
by (*simp add*: *set-eq-iff*)

lemma *interp-concept-SubstC-AtomC*:
  interp-concept (*SubstC* (*AtomC a*) *sb*) *i* = *interp-concept* (*AtomC a*) *i*
by (*case-tac sb*) *simp-all*

lemma *interp-concept-SubstC-AndC*:
  interp-concept (*SubstC* (*AndC c1 c2*) *sb*) *i* =
  interp-concept (*AndC* (*SubstC c1 sb*) (*SubstC c2 sb*)) *i*
by *simp*

lemma *interp-concept-SubstC-OrC*:
  interp-concept (*SubstC* (*OrC c1 c2*) *sb*) *i* =
  interp-concept (*OrC* (*SubstC c1 sb*) (*SubstC c2 sb*)) *i*
by *simp*

lemma *interp-concept-SubstC-NotC*:
  interp-concept (*SubstC* (*NotC c*) *sb*) *i* =
  interp-concept (*NotC* (*SubstC c sb*)) *i*
by *simp*

lemma *interp-concept-SubstC-NumRestrC-other-role*:
  $r \neq r' \Longrightarrow$
  interp-concept (*SubstC* (*NumRestrC nro n r' c*) (*RSubst r rop p*)) *i* =
  interp-concept (*NumRestrC nro n r'* (*SubstC c* (*RSubst r rop p*))) *i*
by *simp*

lemma *interp-form-SubstFm-NegFm*:
  interp-form (*SubstFm* (*NegFm f*) *sb*) *i* =
  interp-form (*NegFm* (*SubstFm f sb*)) *i*
by *simp*

lemma *interp-form-SubstFm-ConjFm*:
  interp-form (*SubstFm* (*BinopFm bop f1 f2*) *sb*) *i* =
  interp-form (*BinopFm bop* (*SubstFm f1 sb*) (*SubstFm f2 sb*)) *i*
by (*case-tac sb*) ((*case-tac bop*), *simp-all*)+

lemma *interp-fact-NumRestrC-Neq-RAdd-expl-subst*:
interp-form (*NegFm* (*ConjFm* (*ConjFm*
(*FactFm* (*Eq True x v1*))
(*FactFm* (*Inst v2* (*SubstC c* (*RSubst r RAdd* (*v1*, *v2*))))))
(*FactFm* (*AtomR False r v1 v2*)))) *i*
⟹ *interp-fact* (*Inst x* (*SubstC* (*NumRestrC nro n r c*) (*RSubst r RAdd* (*v1*, *v2*)))) *i* =
interp-fact (*Inst x* (*NumRestrC nro n r* (*SubstC c* (*RSubst r RAdd* (*v1*, *v2*)))))
i
apply (*simp only*: *interp-form.simps de-Morgan-conj*)
apply (*elim disjE*)
apply (*clarsimp simp add*: *rel-restrict-diff rel-restrict-insert interp-r-modif-def insert-absorb*)+
done

lemma *interp-numres-ord-Eq-RAdd*:
⟦*interp-i i v2* ∈ *ci*; (*interp-i i v1*, *interp-i i v2*) ∉ *interp-r i r* ⟧
⟹ *interp-numres-ord nro*
(*Range* (*rel-restrict* (*interp-r* (*interp-r-modif r* (*insert* (*interp-i i v1*, *interp-i i v2*) (*interp-r i r*)) *i*) *r*) {*interp-i i v1*} *ci*))
(*Suc n*) =
interp-numres-ord nro (*Range* (*rel-restrict* (*interp-r i r*) {*interp-i i v1*} *ci*))
n
apply (*simp add*: *interp-r-modif-def*)
apply (*simp add*: *rel-restrict-diff rel-restrict-insert*)
apply (*subgoal-tac interp-i i v2* ∉ *Range* (*rel-restrict* (*interp-r i r*) {*interp-i i v1*} *ci*))
prefer *2* **apply** (*simp add*: *rel-restrict-def*) **apply** *fastforce*
apply (*case-tac nro*)
apply (*simp add*: *card-le-Suc-insert*)
apply (*simp add*: *card-geq-Suc-insert*)
done

lemma *interp-numres-ord-Eq-RDiff*:
⟦*interp-i i v2* ∈ *ci*; (*interp-i i v1*, *interp-i i v2*) ∈ *interp-r i r*⟧
⟹ *interp-numres-ord nro*
(*Range* (*rel-restrict* (*interp-r* (*interp-r-modif r* (*interp-r i r* − {(*interp-i i v1*, *interp-i i v2*)}) *i*) *r*) {*interp-i i v1*} *ci*))
n =
interp-numres-ord nro (*Range* (*rel-restrict* (*interp-r i r*) {*interp-i i v1*} *ci*))
(*Suc n*)
apply (*simp add*: *interp-r-modif-def*)
apply (*simp add*: *rel-restrict-diff rel-restrict-insert*)
apply (*clarsimp simp add*: *rel-restrict-remove*)
apply (*subgoal-tac interp-i i v2* ∈ *Range* (*rel-restrict* (*interp-r i r*) {*interp-i i v1*}

ci))
 prefer *2* **apply** (*fastforce simp add*: *rel-restrict-def*)
apply (*case-tac nro*)
apply (*simp add*: *card-le-Suc-diff*)
apply (*simp add*: *card-geq-Suc-diff*)
done

9 Treatment of variables (Proofs)

lemma *lift-var-not-bound* [*simp*]: ¬ (*lift-var 0 v* = *Bound 0*)
by (*case-tac v*) *simp-all*

lemma *shift-var-lift-var* [*simp*]:
 shift-var n (*lift-var n v*) = *v*
by (*case-tac v*) *simp+*

lemma *lift-var-shift-var*:
 x ≠ *Bound n* ⟹ *lift-var n* (*shift-var n x*) = *x*
apply (*case-tac x*)
apply *clarsimp+*
apply *arith*
done

lemma *interp-c-interp-bound* [*simp*]:
 interp-c (*interp-bound xi i*) *nc* = *interp-c i nc*
by (*simp add*: *interp-bound-def*)

lemma *interp-r-interp-bound* [*simp*]:
 (*interp-r* (*interp-bound xi i*) *r*) = *interp-r i r*
by (*simp add*: *interp-bound-def*)

lemma *interp-i-interp-bound-i-Bound0* [*simp*]:
 interp-i (*interp-bound xi i*) (*Bound 0*) = *xi*
by (*simp add*: *interp-bound-def*)

lemma *interp-i-interp-bound-i-BoundSuc* [*simp*]:
 interp-i (*interp-bound xi i*) (*Bound* (*Suc k*)) = *interp-i i* (*Bound k*)
by (*simp add*: *interp-bound-def*)

lemma *interp-i-interp-bound-i-Free* [*simp*]:
 (*interp-i* (*interp-bound xi i*)) (*Free v*) = *interp-i i* (*Free v*)
by (*simp add*: *interp-bound-def*)

lemma *interp-i-interp-bound-lift-var* [*simp*]:
 interp-i (*interp-bound xi i*) (*lift-var 0 v*) = *interp-i i v*
by (*case-tac v*) *simp+*

lemma *interpRO-interp-bound* [*simp*]:
(*interpRO rop nr* (*lift-var 0 v1*, *lift-var 0 v2*) (*interp-bound xi i*)) =
(*interpRO rop nr* (*v1*, *v2*) *i*)
by (*case-tac rop*) *simp-all*

definition *fun-replace-at n xi i* = (λ *v*.
(*case v of*
(*Free w*) ⇒ (*interp-i i*) (*Free w*)
| (*Bound k*) ⇒ (*if n* = *k then xi else* (*if n* < *k then* (*interp-i i*) (*Bound* (*k* − *1*))
else (*interp-i i*) (*Bound k*)))))

definition *interp-replace-at n xi i* = *i*(|*interp-i* := *fun-replace-at n xi i*|)

lemma *interp-replace-at-0-interp-bound*:
interp-bound xi i = *interp-replace-at 0 xi i*
apply (*simp add*: *interp-replace-at-def fun-replace-at-def interp-bound-def*)
apply (*cases i*)
apply (*clarsimp simp add*: *fun-eq-iff split add*: *var.split*)
done

lemma *fun-replace-at-lift-var* [*simp*]:
fun-replace-at n xi i (*lift-var n v*) = *interp-i i v*
by (*case-tac v*) (*simp add*: *fun-replace-at-def*)+

lemma *interp-replace-at-lift-var* [*simp*]:
(*interp-i* (*interp-replace-at n xi i*) (*lift-var n v*)) = *interp-i i v*
by (*simp add*: *interp-replace-at-def*)

lemma *interp-c-interp-replace-at* [*simp*]:
interp-c (*interp-replace-at n xi i*) *nc* = *interp-c i nc*
by (*simp add*: *interp-replace-at-def*)

lemma *interp-r-interp-replace-at* [*simp*]:
(*interp-r* (*interp-replace-at n xi i*) *r*) = *interp-r i r*
by (*simp add*: *interp-replace-at-def*)

lemma *interpRO-interp-replace-at* [*simp*]:
(*interpRO rop nr* (*lift-var n v1*, *lift-var n v2*) (*interp-replace-at n xi i*)) =
(*interpRO rop nr* (*v1*, *v2*) *i*)
by (*case-tac rop*) *simp-all*

lemma *interp-replace-at-interp-r-modif*:
interp-replace-at n xi (*interp-r-modif r ri i*) = *interp-r-modif r ri* (*interp-replace-at n xi i*)

apply (*simp add*: *interp-r-modif-def fun-replace-at-def interp-replace-at-def*)
apply (*cases i*)
apply (*clarsimp simp add*: *fun-eq-iff split add*: *var.split*)
done

lemma *fun-replace-at-interp-i*:
(*fun-replace-at n xi* (*i*(|*interp-i* := (*interp-i i*)(*y* := *yi*)|))) = ((*fun-replace-at n xi i*)(*lift-var n y* := *yi*))
apply (*simp add*: *fun-eq-iff*)
apply (*simp add*: *fun-replace-at-def split add*: *var.split*)
apply (*auto split add*: *split-if-asm*)
done

lemma *interp-replace-at-interp-i-modif*:
interp-replace-at n xi (*interp-i-modif y yi i*) =
interp-i-modif (*lift-var n y*) *yi* (*interp-replace-at n xi i*)
apply (*simp add*: *interp-i-modif-def interp-replace-at-def*)
apply (*simp add*: *fun-replace-at-interp-i*)
done

lemma *interp-subst-interp-replace-at* [*simp*]:
interp-subst (*lift-subst n sb*) (*interp-replace-at n xi i*) = *interp-replace-at n xi* (*interp-subst sb i*)
apply (*case-tac sb*)
apply (*case-tac prod*)
apply *simp*
apply (*simp add*: *interp-replace-at-interp-r-modif*)
apply (*simp add*: *interp-replace-at-interp-i-modif*)
done

lemma *interp-subst-lift-subst-interp-bound* [*simp*]:
(*interp-subst* (*lift-subst 0 sb*) (*interp-bound xi i*)) = (*interp-bound xi* (*interp-subst sb i*))
by (*simp add*: *interp-replace-at-0-interp-bound*)

lemma *interp-subst-closure-lift-subst* [*rule-format*]:
∀ *xi i*. (*interp-subst-closure* (*map* (*lift-subst 0*) *sbsts*) (*interp-bound xi i*)) = (*interp-bound xi* (*interp-subst-closure sbsts i*))
apply (*induct sbsts*)
apply *simp*
apply *clarsimp*
done

lemma *fun-replace-at-shift-var*:
  (*fun-replace-at n a i* ∘ *shift-var 0*)(*Bound 0* := *xi*) = *fun-replace-at* (*Suc n*) *a* (*i*(|*interp-i* := (*interp-i i* ∘ *shift-var 0*)(*Bound 0* := *xi*)|))
apply (*rule ext*)
apply (*case-tac x*)
apply (*simp add*: *fun-replace-at-def*)
apply (*simp add*: *fun-replace-at-def*)
apply *auto*
done

lemma *interp-bound-fun-replace-at*:
  (*interp-bound xi* (*i*(|*interp-i* := *fun-replace-at n a i*|))) = (*interp-replace-at* (*Suc n*) *a* (*interp-bound xi i*))
apply (*simp add*: *interp-bound-def*)
apply (*simp add*: *interp-replace-at-def interp-bound-def*)
apply (*simp add*: *fun-replace-at-shift-var*)
done

lemma *interp-concept-interp-replace-at* [*rule-format*, *simp*]:
  ∀ *i*. *interp-concept* (*lift-concept n c*) (*interp-replace-at n xi i*) = *interp-concept c i*
by (*induct c*) *simp-all*

lemma *interp-fact-interp-replace-at* [*simp*]:
  interp-fact (*lift-fact n fact*) (*interp-replace-at n xi i*) = *interp-fact fact i*
by (*induct fact*) *simp-all*

lemma *interp-form-interp-replace-at* [*rule-format*, *simp*]:
  ∀ *n i xi*. *interp-form* (*lift-form n frm*) (*interp-replace-at n xi i*) = *interp-form frm i*
apply (*induct frm*)
apply *simp-all*
apply *clarsimp* **apply** (*case-tac binop*) **apply** *simp* **apply** *simp*
apply (*case-tac quantif*)
apply (*clarsimp simp add*: *interp-replace-at-def interp-bound-fun-replace-at*)
apply (*clarsimp simp add*: *interp-replace-at-def interp-bound-fun-replace-at*)
done

lemma *interp-form-interp-bound* [*simp*]:
  interp-form (*lift-form 0 f*) (*interp-bound xi i*) = *interp-form f i*
by (*simp add*: *interp-replace-at-0-interp-bound*)

lemma *interp-form-shuffle-right* [*rule-format*]:
  ∀ *f1 i*. *interp-form* (*shuffle-right bop f1 f2*) *i* = *interp-form* (*BinopFm bop f1 f2*) *i*
apply (*induct f2*)

prefer *5*
apply *clarify*
apply (*case-tac bop*)
apply (*case-tac quantif*)
apply *simp-all*
apply (*case-tac quantif*)
apply *simp-all*
done

lemma *interp-form-shuffle-left* [*rule-format*]:
∀ *f2 i*. *interp-form* (*shuffle-left bop f1 f2*) *i* = *interp-form* (*BinopFm bop f1 f2*) *i*
apply (*induct f1*)
prefer *5*
apply *clarify*
apply (*case-tac bop*)
apply (*case-tac quantif*)
apply *simp-all*
apply (*case-tac quantif*)
apply *simp-all*
apply (*simp add*: *interp-form-shuffle-right*)+
done

lemma *interp-form-lift-bound-above-negfm* [*rule-format*, *simp*]:
∀ *i*. *interp-form* (*lift-bound-above-negfm f*) *i* = (¬ *interp-form f i*)
apply (*induct f*)
prefer *5*
apply (*case-tac quantif*)
apply *auto*
done

lemma *interp-form-lift-bound-above-substfm* [*rule-format*, *simp*]:
∀ *i sb*. *interp-form* (*lift-bound-above-substfm f sb*) *i* = (*interp-form* (*SubstFm f sb*) *i*)
apply (*induct f*)
apply *simp-all*
apply (*case-tac quantif*)
apply *simp-all*
done

lemma *interp-form-lift-bound* [*rule-format*, *simp*]:
∀ *i*. *interp-form* (*lift-bound f*) *i* = *interp-form f i*
apply (*induct f*)
apply *simp-all*
apply (*case-tac binop*)
apply (*simp add*: *interp-form-shuffle-left*)+
apply (*case-tac quantif*)

apply *simp-all*
done

lemma *interp-i-modif-interp-bound*:
interp-i-modif (*Free x*) *xi* (*interp-bound xi i*) = *interp-replace-at 0 xi* (*interp-i-modif* (*Free x*) *xi i*)
apply (*simp add*: *interp-replace-at-0-interp-bound*)
apply (*simp add*: *interp-replace-at-interp-i-modif*)
done

lemma *interp-form-bind*:
(*interp-form* (*bind QAll v frm*) *i*) = (∀ *vi*. *interp-form frm* (*interp-i-modif* (*Free v*) *vi i*))
apply (*simp add*: *bind-def*)
apply (*simp add*: *interp-i-modif-interp-bound*)
done

10 Hoare logic where the conditions of the Hoare triples are DL formulae

inductive
hoare :: ('*r*, '*c*, '*i var*) *form* ⇒ ('*r*, '*c*, '*i*) *stmt* ⇒ ('*r*, '*c*, '*i var*) *form* ⇒ *bool*
(⊢ ({(*1-*)}/ (-)/ {(*1-*)}) *50*)
where
Skip: ⊢ {*P*} *SKIP* {*P*} |
EDel: ⊢ { *SubstFm Q* (*RSubst r RDiff* (*Free v1*, *Free v2*)) } *EDel v1 r v2* {*Q*} |
EGen: ⊢ { *SubstFm Q* (*RSubst r RAdd* (*Free v1*, *Free v2*)) } *EGen v1 r v2* {*Q*} |
SelAss: ⊢ { *bind QAll v* (*ImplFm b Q*) } (*SelAss v b*) {*Q*} |

Seq: ⟦ ⊢ {*P*} c_1 {*Q*}; ⊢ {*Q*} c_2 {*R*} ⟧
⟹ ⊢ {*P*} (c_1;c_2) {*R*} |

If: ⟦ ⊢ {*ConjFm P b*} c_1 {*Q*}; ⊢ {*ConjFm P* (*NegFm b*) } c_2 {*Q*} ⟧
⟹ ⊢ {*P*} *IF b THEN* c_1 *ELSE* c_2 {*Q*} |
While: ⊢ {*ConjFm P b*} *c* {*P*} ⟹
⊢ {*P*} *WHILE* {*iv*} *b DO c* {*ConjFm P* (*NegFm b*)} |

conseq: *validFm* ((*ImplFm P' P*) ::('*r*, '*c*, '*i var*) *form*) ⟹
⊢ {*P*} *c* {*Q*} ⟹
validFm ((*ImplFm Q Q'*) ::('*r*, '*c*, '*i var*) *form*) ⟹
⊢ {*P'*} *c* {*Q'*}

lemmas [*simp*] = *hoare.Skip hoare.EDel hoare.EGen hoare.SelAss hoare.Seq hoare.If*

lemmas [*intro*!] = *hoare.Skip hoare.EDel hoare.EGen hoare.SelAss hoare.Seq hoare.If*

lemma *strengthen-pre*:
$[\![$ *validFm* (*ImplFm* P' *P*); $\vdash \{P\}$ *c* $\{Q\}$ $]\!] \Longrightarrow \vdash \{P'\}$ *c* $\{Q\}$
apply (*erule conseq*)
apply *assumption*
apply (*simp add*: *validFm-def lift-impl-def*)
done

lemma *weaken-post*:
$[\![\vdash \{P\}$ *c* $\{Q\}$; *validFm* (*ImplFm* *Q* Q') $]\!] \Longrightarrow \vdash \{P\}$ *c* $\{Q'\}$
apply (*rule conseq*)
prefer *2* **apply** *assumption*
prefer *2* **apply** *assumption*
apply (*simp add*: *validFm-def lift-impl-def*)
done

lemma *While'*:
assumes $\vdash$ {*ConjFm P b*} *c* {*P*} **and** *validFm* (*ImplFm* (*ConjFm P* (*NegFm b*)) *Q*)
shows $\vdash$ {*P*} *WHILE* {*iv*} *b DO c* {*Q*}
by(*rule weaken-post*[*OF While*[*OF assms*(*1*)] *assms*(*2*)])

definition
hoare-valid :: ('*r*, '*c*, '*i var*) *form* ⇒ ('*r*, '*c*, '*i*) *stmt* ⇒ ('*r*, '*c*, '*i var*) *form* ⇒ *bool*
(⊨ {(*1*-)}/ (-)/ {(*1*-)} *50*) **where**
⊨ {*P*}*c*{*Q*} = (∀ *s t*. (*c*,*s*) ⇒ *t* ⟶ *interp-form P s* ⟶ *interp-form Q t*)

lemma *interp-form-SubstFm-delete*:
interp-form (*SubstFm Q* (*RSubst r RDiff* (*v1*, *v2*)))
= (*interp-form Q*) ∘ (*delete-edge v1 r v2*)
by (*rule ext*) (*simp add*: *interp-r-modif-def delete-edge-def*)

lemma *interp-form-SubstFm-generate*:
interp-form (*SubstFm Q* (*RSubst r RAdd* (*v1*, *v2*)))
= (*interp-form Q*) ∘ (*generate-edge v1 r v2*)
by (*rule ext*) (*simp add*: *interp-r-modif-def generate-edge-def*)

lemma *hoare-sound-while* [*rule-format*]:
((*WHILE* {*iv*} *b DO c*, *s*) ⇒ *t*) ⟹
⊨ {*ConjFm P b*} *c* {*P*} ⟹
interp-form P s ⟶ *interp-form P t* ∧ ¬ *interp-form b t*
apply (*induction* (*WHILE* {*iv*} *b DO c*) *s t rule*: *big-step-induct*)

apply *simp*
apply (*simp add*: *hoare-valid-def*)
done

lemma *hoare-sound*: $\vdash \{P\}c\{Q\} \implies \models \{P\}c\{Q\}$
proof(*induction rule*: *hoare.induct*)
  case (*Skip P*) **thus** *?case*
    by (*auto simp*: *hoare-valid-def*)
next
  case (*EDel Q r v1 v2*) **thus** *?case*
    by (*clarsimp simp only*: *hoare-valid-def interp-form-SubstFm-delete*, *simp*)
next
  case (*EGen Q r v1 v2*) **thus** *?case*
    by (*clarsimp simp only*: *hoare-valid-def interp-form-SubstFm-generate*, *simp*)
next
  case (*SelAss v b Q*) **thus** *?case*
    by (*auto simp add*: *hoare-valid-def interp-form-bind ImplFm-def*)
next
  case (*If P b c1 Q c2*) **thus** *?case*
    by (*auto simp add*: *hoare-valid-def interp-form-IfThenElseFm*)
next
  case (*Seq P c1 Q c2 R*) **thus** *?case*
    by (*auto simp add*: *hoare-valid-def*)
next
  case (*While P b c iv*) **thus** *?case*
    by (*clarsimp simp add*: *hoare-valid-def hoare-sound-while*)
next
  case (*conseq P' P c Q Q'*) **thus** *?case*
    by (*auto simp add*: *hoare-valid-def validFm-def lift-impl-def*)
qed

fun *wp-dl* :: ('*r*, '*c*, '*i*) *stmt* ⇒ ('*r*, '*c*, '*i var*) *form* ⇒ ('*r*, '*c*, '*i var*) *form* **where**
  wp-dl SKIP Qd = *Qd*
| *wp-dl* (*EDel v1 r v2*) *Qd* = *SubstFm Qd* (*RSubst r RDiff* (*Free v1*, *Free v2*))
| *wp-dl* (*EGen v1 r v2*) *Qd* = *SubstFm Qd* (*RSubst r RAdd* (*Free v1*, *Free v2*))
| *wp-dl* (*SelAss v b*) *Qd* = *bind QAll v* (*ImplFm b Qd*)
| *wp-dl* (c_1 ; c_2) *Qd* = *wp-dl* c_1 (*wp-dl* c_2 *Qd*)
| *wp-dl* (*IF b THEN* c_1 *ELSE* c_2) *Qd* = *IfThenElseFm b* (*wp-dl* c_1 *Qd*) (*wp-dl* c_2 *Qd*)
| *wp-dl* (*WHILE* {*iv*} *b DO c*) *Qd* = *iv*

fun *vc* :: (*'r*, *'c*, *'i*) *stmt* ⇒ (*'r*, *'c*, *'i var*) *form* ⇒ (*'r*, *'c*, *'i var*) *form* **where**
  vc SKIP Qd = *TrueFm*
| *vc* (*EDel v1 r v2*) *Qd* = *TrueFm*
| *vc* (*EGen v1 r v2*) *Qd* = *TrueFm*
| *vc* (*SelAss v b*) *Qd* = *TrueFm*
| *vc* (c_1 ; c_2) *Qd* = *ConjFm* (*vc* c_1 (*wp-dl* c_2 *Qd*)) (*vc* c_2 *Qd*)
| *vc* (*IF b THEN* c_1 *ELSE* c_2) *Qd* = *ConjFm* (*vc* c_1 *Qd*) (*vc* c_2 *Qd*)
| *vc* (*WHILE* {*iv*} *b DO c*) *Qd* =
      ConjFm (*ConjFm*
          (*ImplFm* (*ConjFm iv* (*NegFm b*)) *Qd*)
          (*ImplFm* (*ConjFm iv b*) (*wp-dl c iv*)))
          (*vc c iv*)

lemma *quantif-free-univ-quantif* [*simp*]:
  quantif-free frm ⟶ *univ-quantif b frm*
by (*induct frm arbitrary*: *b*) *auto*

lemma *univ-quantif-lift-form* [*simp*]:
  univ-quantif b frm ⟶ *univ-quantif b* (*lift-form n frm*)
by (*induct frm arbitrary*: *b n*) *auto*

lemma *univ-quantif-wp-dl* [*rule-format*, *simp*]:
  univ-quantif True q ⟶ *form-prop-in-stmt quantif-free c* ⟶ *univ-quantif True* (*wp-dl c q*)
by (*induct c arbitrary*: *q*) (*auto simp add*: *bind-def ImplFm-def IfThenElseFm-def*)

lemma *univ-quantif-vc* [*rule-format*, *simp*]:
  univ-quantif True q ⟶ *form-prop-in-stmt quantif-free c* ⟶ *univ-quantif True* (*vc c q*)
by (*induct c arbitrary*: *q*) (*auto simp add*: *bind-def ImplFm-def IfThenElseFm-def*)

Soundness:

lemma *vc-sound*: *validFm* (*vc c Q*) ⟹ ⊢ {*wp-dl c Q*} *c* {*Q*}
proof(*induction c arbitrary*: *Q*)
  case (*While I b c*)
  show *?case*
  proof(*simp*, *rule While'*)
    from ‹*validFm* (*vc* (*While I b c*) *Q*)›
    have *vc*: *validFm* (*vc c I*) **and** *IQ*: *validFm* (*ImplFm* (*ConjFm I* (*NegFm b*)) *Q*) **and**
          pre: *validFm* (*ImplFm* (*ConjFm I b*) (*wp-dl c I*)) **by** (*simp-all add*: *validFm-def*)
    have ⊢ {*wp-dl c I*} *c* {*I*} **by** (*rule While.IH* [*OF vc*])
    with *pre* **show** ⊢ {*ConjFm I b*} *c* {*I*}
      by(*rule strengthen-pre*)
    show *validFm* (*ImplFm* (*ConjFm I* (*NegFm b*)) *Q*) **by**(*rule IQ*)

qed
next
case *SKIP* **show** *?case* **by** *simp*
next
case (*EDel i1 r i2*) **show** *?case* **by** *simp*
next
case (*EGen i1 r i2*) **show** *?case* **by** *simp*
next
case (*SelAss i frm*) **show** *?case* **by** *simp*
next
case (*Seq c1 c2*) **thus** *?case* **by** (*auto simp add*: *validFm-def*)
next
case (*If b c1 c2*) **thus** *?case* **apply** (*auto intro*: *hoare.conseq simp add*: *validFm-def lift-impl-def lift-ite-def*)
apply (*rule hoare.conseq*) **prefer** *2* **apply** *blast* **apply** (*simp add*: *validFm-def lift-impl-def lift-ite-def*)+
apply (*rule hoare.conseq*) **prefer** *2* **apply** *blast* **by** (*simp add*: *validFm-def lift-impl-def lift-ite-def*)+
qed

lemma *validFm-ConjFm*: *validFm* (*ConjFm a b*) = (*validFm a* ∧ *validFm b*)
by (*fastforce simp add*: *validFm-def*)

lemma *validFm-mp*: *validFm* (*ImplFm P Q*) ⟹ *validFm P* ⟹ *validFm Q*
by (*simp add*: *validFm-def lift-impl-def*)

lemma *wp-dl-mono*:
validFm (*ImplFm P P′*) ⟹ *validFm* (*ImplFm* (*wp-dl c P*) (*wp-dl c P′*))
apply (*induction c arbitrary*: *P P′ s*)
apply (*simp-all add*: *validFm-def lift-impl-def interp-form-bind lift-ite-def*)+
apply *blast*
done

lemma *vc-mono*:
validFm (*ImplFm P P′*) ⟹ *validFm* (*ImplFm* (*vc c P*) (*vc c P′*))
apply (*induction c arbitrary*: *P P′ s*)
apply (*simp-all add*: *validFm-def lift-impl-def interp-form-bind lift-ite-def*)+
apply (*clarsimp*)
apply (*insert wp-dl-mono*)
apply (*simp-all add*: *validFm-def lift-impl-def interp-form-bind lift-ite-def*)+
apply *blast*
done

corollary *vc-sound′*:
(*validFm* (*ConjFm* (*vc c Q*) (*ImplFm P* (*wp-dl c Q*)))) ⟹ ⊢ {*P*} *c* {*Q*}
by (*simp only*: *validFm-ConjFm*) (*metis strengthen-pre vc-sound*)

11 Explicit substitutions

11.1 Functions used in termination arguments

fun *subst-closure-concept* :: (*'nr*, *'nc*, *'ni*) *concept* ⇒ (*'nr*, *'nc*, *'ni*) *subst list* ⇒ (*'nr*, *'nc*, *'ni*) *concept* **where**
subst-closure-concept c [] = *c*
| *subst-closure-concept c* (*sb* # *sbsts*) = *subst-closure-concept* (*SubstC c sb*) *sbsts*

fun *subst-closure-form* :: (*'nr*, *'nc*, *'ni*) *form* ⇒ (*'nr*, *'nc*, *'ni*) *subst list* ⇒ (*'nr*, *'nc*, *'ni*) *form* **where**
subst-closure-form fm [] = *fm*
| *subst-closure-form fm* (*sb* # *sbsts*) = *subst-closure-form* (*SubstFm fm sb*) *sbsts*

fun *height-concept* :: (*'nr*, *'nc*, *'ni*) *concept* ⇒ *nat* **where**
height-concept Bottom = *1*
| *height-concept Top* = *1*
| *height-concept* (*AtomC a*) = *1*
| *height-concept* (*AndC c1 c2*) = *Suc* (*max* (*height-concept c1*) (*height-concept c2*))
| *height-concept* (*OrC c1 c2*) = *Suc* (*max* (*height-concept c1*) (*height-concept c2*))
| *height-concept* (*NotC c*) = *Suc* (*height-concept c*)
| *height-concept* (*NumRestrC nro n r c*) = *Suc* (*height-concept c*)
| *height-concept* (*SubstC c sb*) = *Suc* (*height-concept c*)

fun *height-fact* :: (*'nr*, *'nc*, *'ni*) *fact* ⇒ *nat* **where**
height-fact (*Inst x c*) = *height-concept c*
| *height-fact* - = *0*

fun *height-form* :: (*'nr*, *'nc*, *'ni*) *form* ⇒ *nat* **where**
height-form FalseFm = *0*
| *height-form* (*FactFm fct*) = *Suc* (*height-fact fct*)
| *height-form* (*NegFm f*) = *Suc* (*height-form f*)
| *height-form* (*BinopFm bop f1 f2*) = *Suc* (*max* (*height-form f1*) (*height-form f2*))
| *height-form* (*QuantifFm q f*) = *Suc* (*height-form f*)
| *height-form* (*SubstFm f sb*) = *Suc* (*height-form f*)

fun *subst-height-concept* :: ('*nr*, '*nc*, '*ni*) *concept* ⇒ *nat* **where**
subst-height-concept Bottom = *0*
| *subst-height-concept Top* = *0*
| *subst-height-concept* (*AtomC a*) = *0*
| *subst-height-concept* (*AndC c1 c2*) = *max* (*subst-height-concept c1*) (*subst-height-concept c2*)
| *subst-height-concept* (*OrC c1 c2*) = *max* (*subst-height-concept c1*) (*subst-height-concept c2*)
| *subst-height-concept* (*NotC c*) = *subst-height-concept c*
| *subst-height-concept* (*NumRestrC nro n r c*) = *subst-height-concept c*
| *subst-height-concept* (*SubstC c sb*) = *height-concept c* + *subst-height-concept c*

fun *subst-height-fact* :: ('*nr*, '*nc*, '*ni*) *fact* ⇒ ('*nr*, '*nc*, '*ni*) *subst list* ⇒ *nat* **where**
subst-height-fact (*Inst x c*) *sbsts* = *subst-height-concept* (*subst-closure-concept c sbsts*)
| *subst-height-fact* - *sbsts* = *length sbsts*

fun *subst-height-form* :: ('*nr*, '*nc*, '*ni*) *form* ⇒ ('*nr*, '*nc*, '*ni*) *subst list* ⇒ *nat* **where**
subst-height-form FalseFm sbsts = *0*
| *subst-height-form* (*FactFm fct*) *sbsts* = *subst-height-fact fct sbsts*
| *subst-height-form* (*NegFm f*) *sbsts* = *subst-height-form f sbsts*
| *subst-height-form* (*BinopFm bop f1 f2*) *sbsts* = *max* (*subst-height-form f1 sbsts*) (*subst-height-form f2 sbsts*)
| *subst-height-form* (*QuantifFm q f*) *sbsts* = *subst-height-form f sbsts*
| *subst-height-form* (*SubstFm f sb*) *sbsts* = *subst-height-form f* (*sb* # *sbsts*)

lemma *height-concept-positive* [*simp*]: *0* < *height-concept c*
by (*induct c*) *auto*

lemma *subst-height-concept-mono-closure* [*rule-format*]:
∀ *c1 c2*. *subst-height-concept c1* < *subst-height-concept c2* ⟶ *height-concept c1* ≤ *height-concept c2* ⟶
subst-height-concept (*subst-closure-concept c1 sbsts*) < *subst-height-concept* (*subst-closure-concept c2 sbsts*)
by (*induct sbsts*) *simp-all*

lemma *length-subst-height-concept*:
length sbsts < *subst-height-concept* (*subst-closure-concept* (*SubstC c sb*) *sbsts*)
apply (*induct sbsts*)
apply *clarsimp*
apply *clarsimp*
apply (*subgoal-tac subst-height-concept* (*subst-closure-concept* (*SubstC c sb*) *sbsts*)
< *subst-height-concept* (*subst-closure-concept* (*SubstC* (*SubstC c sb*) *a*)

sbsts))
apply *arith*
apply (*simp add*: *subst-height-concept-mono-closure*)
done

lemma *subst-height-concept-positive* [*simp*]:
0 < *subst-height-concept* (*subst-closure-concept* (*SubstC c sb*) *sbsts*)
by (*insert length-subst-height-concept* [*of sbsts c sb*]) *arith*

11.2 Moving substitutions further downwards

fun *push-rsubst-concept-numrestrc* ::
role-op ⇒ (*'ni* ∗ *'ni*) ⇒ *'ni* ⇒ *numres-ord* ⇒ *nat* ⇒ *'nr* ⇒ (*'nr*, *'nc*, *'ni*) *concept*

⇒ (*'nr*, *'nc*, *'ni*) *form* **where**
push-rsubst-concept-numrestrc RAdd (*v1*, *v2*) *x Le 0 r c* = *FalseFm*
| *push-rsubst-concept-numrestrc RAdd* (*v1*, *v2*) *x Geq 0 r c* = *TrueFm*
| *push-rsubst-concept-numrestrc RAdd* (*v1*, *v2*) *x nro* (*Suc n*) *r c* =
(*IfThenElseFm* (*ConjFm* (*ConjFm*
(*FactFm* (*Eq True x v1*))
(*FactFm* (*Inst v2* (*SubstC c* (*RSubst r RAdd* (*v1*, *v2*))))))
(*FactFm* (*AtomR False r v1 v2*)))
(*FactFm* (*Inst x* (*NumRestrC nro n r* (*SubstC c* (*RSubst r RAdd* (*v1*, *v2*))))))
(*FactFm* (*Inst x* (*NumRestrC nro* (*Suc n*) *r* (*SubstC c* (*RSubst r RAdd* (*v1*, *v2*)))))))
| *push-rsubst-concept-numrestrc RDiff* (*v1*, *v2*) *x nro n r c* =
(*IfThenElseFm* (*ConjFm* (*ConjFm*
(*FactFm* (*Eq True x v1*))
(*FactFm* (*Inst v2* (*SubstC c* (*RSubst r RDiff* (*v1*, *v2*))))))
(*FactFm* (*AtomR True r v1 v2*)))
(*FactFm* (*Inst x* (*NumRestrC nro* (*Suc n*) *r* (*SubstC c* (*RSubst r RDiff* (*v1*, *v2*))))))
(*FactFm* (*Inst x* (*NumRestrC nro n r* (*SubstC c* (*RSubst r RDiff* (*v1*, *v2*)))))))

fun *push-rsubst-concept* :: *'nr* ⇒ *role-op* ⇒ (*'ni* ∗ *'ni*) ⇒ *'ni* ⇒ (*'nr*, *'nc*, *'ni*) *concept* ⇒ (*'nr*, *'nc*, *'ni*) *form* **where**
push-rsubst-concept r rop v1v2 x (*AtomC a*) = (*FactFm* (*Inst x* (*AtomC a*)))
| *push-rsubst-concept r rop v1v2 x Top* = (*FactFm* (*Inst x Top*))
| *push-rsubst-concept r rop v1v2 x Bottom* = (*FactFm* (*Inst x Bottom*))
| *push-rsubst-concept r rop v1v2 x* (*NotC c*) = (*FactFm* (*Inst x* (*NotC* (*SubstC c* (*RSubst r rop v1v2*)))))
| *push-rsubst-concept r rop v1v2 x* (*AndC c1 c2*) =
(*FactFm* (*Inst x* (*AndC* (*SubstC c1* (*RSubst r rop v1v2*)) (*SubstC c2* (*RSubst r rop v1v2*)))))
| *push-rsubst-concept r rop v1v2 x* (*OrC c1 c2*) =
(*FactFm* (*Inst x* (*OrC* (*SubstC c1* (*RSubst r rop v1v2*)) (*SubstC c2* (*RSubst r rop v1v2*)))))
| *push-rsubst-concept r rop v1v2 x* (*NumRestrC nro n r' c*) =
(*if r* = *r'*

then push-rsubst-concept-numrestrc rop v1v2 x nro n r c
else (FactFm (Inst x (NumRestrC nro n r′ (SubstC c (RSubst r rop v1v2))))))
| *push-rsubst-concept r rop v1v2 x (SubstC c sb) = (SubstFm (FactFm (Inst x (SubstC c sb))) (RSubst r rop v1v2))*

definition *subst-AtomR-RDiff sign r x y v1 v2* ≡
(let fm = (ConjFm (DisjFm (FactFm (Eq False x v1)) (FactFm (Eq False y v2))) (FactFm (AtomR True r x y))) in
(if sign then fm else (NegFm fm)))

definition *subst-AtomR-RAdd sign r x y v1 v2* ≡
(let fm = (DisjFm (ConjFm (FactFm (Eq True x v1)) (FactFm (Eq True y v2))) (FactFm (AtomR True r x y))) in
(if sign then fm else (NegFm fm)))

fun *push-rsubst-fact* :: *′nr ⇒ role-op ⇒ (′ni ∗ ′ni) ⇒ (′nr, ′nc, ′ni) fact ⇒ (′nr, ′nc, ′ni) form* **where**
push-rsubst-fact r rop v1v2 (Inst x c) = push-rsubst-concept r rop v1v2 x c
| *push-rsubst-fact r rop v1v2 (AtomR sign r′ x y) =*
(if r = r′
then (let (v1, v2) = v1v2 in
(case rop of
RDiff ⇒ (subst-AtomR-RDiff sign r x y v1 v2)
| *RAdd ⇒ (subst-AtomR-RAdd sign r x y v1 v2)))*
else FactFm (AtomR sign r′ x y))
| *push-rsubst-fact r rop v1v2 (Eq sign x y) = FactFm (Eq sign x y)*

fun *replace-var* :: *′ni ⇒ ′ni ⇒ ′ni ⇒ ′ni* **where**
replace-var v1 v2 w = (if w = v1 then v2 else w)

fun *subst-vars* :: *(′ni ∗ ′ni) list ⇒ ′ni ⇒ ′ni* **where**
subst-vars [] x = x
| *subst-vars ((v1, v2) # sbsts) x = subst-vars sbsts (replace-var v1 v2 x)*

fun *push-isubst-concept* :: *(′nr, ′nc, ′ni) concept ⇒ (′ni ∗ ′ni) list ⇒ (′nr, ′nc, ′ni) concept* **where**
push-isubst-concept (AtomC a) sbsts = (AtomC a)
| *push-isubst-concept Top sbsts = Top*
| *push-isubst-concept Bottom sbsts = Bottom*
| *push-isubst-concept (NotC c) sbsts = (NotC (push-isubst-concept c sbsts))*
| *push-isubst-concept (AndC c1 c2) sbsts =*
(AndC (push-isubst-concept c1 sbsts) (push-isubst-concept c2 sbsts))
| *push-isubst-concept (OrC c1 c2) sbsts =*
(OrC (push-isubst-concept c1 sbsts) (push-isubst-concept c2 sbsts))
| *push-isubst-concept (NumRestrC nro n r′ c) sbsts =*
(NumRestrC nro n r′ (push-isubst-concept c sbsts))
| *push-isubst-concept (SubstC c (RSubst r rop (x1, x2))) sbsts =*

(*SubstC* (*push-isubst-concept c sbsts*) (*RSubst r rop* ((*subst-vars sbsts x1*), (*subst-vars sbsts x2*))))
| *push-isubst-concept* (*SubstC c* (*ISubst x1 x2*)) *sbsts* = *push-isubst-concept c* ((*x1*,*x2*) # *sbsts*)

fun *push-isubst-fact* :: *'ni* ⇒ *'ni* ⇒ (*'nr*, *'nc*, *'ni*) *fact* ⇒ (*'nr*, *'nc*, *'ni*) *form*
where
push-isubst-fact v1 v2 (*Inst x c*) = *FactFm* (*Inst* (*replace-var v1 v2 x*) (*push-isubst-concept c* [(*v1*,*v2*)]))
| *push-isubst-fact v1 v2* (*AtomR sign r' x y*) = *FactFm* (*AtomR sign r'* (*replace-var v1 v2 x*) (*replace-var v1 v2 y*))
| *push-isubst-fact v1 v2* (*Eq sign x y*) = *FactFm* (*Eq sign* (*replace-var v1 v2 x*) (*replace-var v1 v2 y*))

fun *push-subst-fact* :: (*'nr*, *'nc*, *'ni*) *fact* ⇒ (*'nr*, *'nc*, *'ni*) *subst* ⇒ (*'nr*, *'nc*, *'ni*) *form* **where**
push-subst-fact fct (*RSubst r rop p*) = *push-rsubst-fact r rop p fct*
| *push-subst-fact fct* (*ISubst v1 v2*) = *push-isubst-fact v1 v2 fct*

type-synonym (*'nr*, *'nc*, *'ni*) *extract-res* = ((*'ni*) ∗ ((*'nr*, *'nc*, *'ni*) *concept*) ∗ ((*'nr*, *'nc*, *'ni*) *subst*)) *option*

fun *extract-subst* :: (*'nr*, *'nc*, *'ni*) *fact* ⇒ ((*'nr*, *'nc*, *'ni*) *extract-res*) **where**
extract-subst (*Inst x* (*SubstC c sb*)) = *Some* (*x*, *c*, *sb*)
| *extract-subst fct* = *None*

function *push-subst-form* :: (*'nr*, *'nc*, *'ni var*) *form* ⇒ (*'nr*, *'nc*, *'ni var*) *subst list* ⇒ (*'nr*, *'nc*, *'ni var*) *form* **where**
push-subst-form FalseFm sbsts = *FalseFm*
| *push-subst-form* (*FactFm fct*) *sbsts* =
(*case extract-subst fct of*
None ⇒
(*case sbsts of*
[] ⇒ (*FactFm fct*)
| *sb* # *sbsts'* ⇒ *push-subst-form* (*push-subst-fact fct sb*) *sbsts'*)
| *Some*(*x*, *c*, *sb*) ⇒
(*case sb of*
(*RSubst r rop v1v2*) ⇒ *push-subst-form* (*FactFm* (*Inst x c*)) ((*RSubst r rop v1v2*)#*sbsts*)
| (*ISubst v1 v2*) ⇒ *push-subst-form* (*FactFm* (*Inst x* (*push-isubst-concept c* [(*v1*, *v2*)]))) *sbsts*))
| *push-subst-form* (*NegFm f*) *sbsts* = *NegFm* (*push-subst-form f sbsts*)
| *push-subst-form* (*BinopFm bop f1 f2*) *sbsts* = (*BinopFm bop* (*push-subst-form f1 sbsts*) (*push-subst-form f2 sbsts*))
| *push-subst-form* (*QuantifFm q f*) *sbsts* = (*QuantifFm q* (*push-subst-form f* (*map* (*lift-subst 0*) *sbsts*)))
| *push-subst-form* (*SubstFm f sb*) *sbsts* = *push-subst-form f* (*sb*#*sbsts*)

by *pat-completeness auto*

end

theory *SubstProofs* **imports** *Subst SemanticsProofs VariablesProofs*
begin

12 Explicit substitutions (Proofs)

12.1 Termination of pushing substititutions

lemma *push-subst-fact-decr-rsubst*: *extract-subst fct* = *None* ⟹ *sb* = (*RSubst r rop v1v2*) ⟹
(*subst-height-form* (*push-subst-fact fct sb*) *sbsts* < *subst-height-fact fct* (*sb* # *sbsts*))
apply (*case-tac fct*)
prefer *3*

apply *simp*
prefer *2*

apply (*clarsimp simp add*: *split-def subst-AtomR-RDiff-def subst-AtomR-RAdd-def Let-def*
split add: *split-if-asm role-op.split*)

apply (*rename-tac x c*)
apply (*case-tac c*)
apply (*fastforce intro*: *subst-height-concept-mono-closure*)+

apply (*case-tac v1v2*)
apply *simp*
apply (*intro conjI impI*)
apply (*case-tac rop*)
apply (*simp add*: *IfThenElseFm-def ImplFm-def length-subst-height-concept subst-height-concept-mono-closure*
apply *clarsimp*
apply (*rename-tac x nro n c v1 v2*)
apply (*case-tac n*)
apply (*case-tac nro*) **apply** *simp-all*
apply (*simp add*: *IfThenElseFm-def ImplFm-def length-subst-height-concept subst-height-concept-mono-closure*

done

lemma *height-concept-push-isubst-concept* [*rule-format*, *simp*]:
∀ *sbsts*. *height-concept* (*push-isubst-concept c sbsts*) ≤ *height-concept c*
apply (*induct c*)
apply *simp-all*
apply *clarsimp* **apply** (*drule-tac x*=*sbsts* **in** *spec*)+ **apply** *arith*
apply *clarsimp* **apply** (*drule-tac x*=*sbsts* **in** *spec*)+ **apply** *arith*
apply *clarsimp*
apply (*case-tac subst*)
apply *clarsimp*+
apply (*rename-tac c sbsts v1 v2*)
apply (*drule-tac x*=((*v1*, *v2*) # *sbsts*) **in** *spec*)
apply *simp*
done

lemma *subst-height-concept-push-isubst-concept* [*rule-format*, *simp*]:
∀ *sbsts*. *subst-height-concept* (*push-isubst-concept c sbsts*) ≤ *subst-height-concept c*
apply (*induct c*)
apply *simp-all*
apply *clarsimp* **apply** (*drule-tac x*=*sbsts* **in** *spec*)+ **apply** *arith*
apply *clarsimp* **apply** (*drule-tac x*=*sbsts* **in** *spec*)+ **apply** *arith*
apply *clarsimp*
apply (*case-tac subst*)
apply *clarsimp*+
apply (*drule-tac x*=*sbsts* **in** *spec*)
apply (*subgoal-tac height-concept* (*push-isubst-concept c sbsts*) ≤ *height-concept c*)
apply *arith* **apply** *simp*

apply *clarsimp*
apply (*rename-tac c sbsts v1 v2*)
apply (*drule-tac x*=((*v1*, *v2*) # *sbsts*) **in** *spec*)
apply *simp*
done

lemma *push-subst-fact-decr-isubst*: *extract-subst fct* = *None* ⟹ *sb* = (*ISubst v1 v2*) ⟹
(*subst-height-form* (*push-subst-fact fct sb*) *sbsts* < *subst-height-fact fct* (*sb* # *sbsts*))
apply *simp*
apply (*case-tac fct*)
prefer *3*

apply *simp*
prefer *2*

apply (*clarsimp*)

apply (*rename-tac x c*)

apply *simp*
apply (*rule subst-height-concept-mono-closure*)
apply *simp+*
apply (*subgoal-tac subst-height-concept* (*push-isubst-concept c* [(*v1*, *v2*)]) ≤ *subst-height-concept* *c*) **prefer** *2* **apply** *simp*
apply (*subgoal-tac 0* < *height-concept c*) **prefer** *2* **apply** *simp*
apply *arith*

apply *simp*
apply (*subgoal-tac height-concept* (*push-isubst-concept c* [(*v1*, *v2*)]) ≤ *height-concept* *c*) **prefer** *2* **apply** *simp*
apply *arith*
done

lemma *push-subst-fact-decr*: *extract-subst fct* = *None* ⟹
subst-height-form (*push-subst-fact fct sb*) *sbsts'* < *subst-height-fact fct* (*sb* # *sbsts'*)
apply (*case-tac sb*)
apply (*rule push-subst-fact-decr-rsubst*) **apply** *assumption+*
apply (*rule push-subst-fact-decr-isubst*) **apply** *assumption+*
done

lemma *extract-subst-Some*:
(*extract-subst fct* = *Some* (*x*, *c*, *sb*)) = (*fct* = (*Inst x* (*SubstC c sb*)))
apply (*case-tac fct*)
apply (*case-tac concept*)
apply *simp-all*
done

lemma *push-subst-extract-some-SubstC*: *extract-subst fct* = *Some*(*x*, *c*, *sb*) ⟹
subst-height-concept (*subst-closure-concept* (*SubstC c sb*) *sbsts*) = *subst-height-fact* *fct sbsts*
by (*clarsimp simp add*: *extract-subst-Some*)

lemma *height-fact-extract-some-decr*: *extract-subst fct* = *Some*(*x*, *c*, *sb*) ⟹
height-concept c < (*height-fact fct*)
by (*clarsimp simp add*: *extract-subst-Some*)

lemma *push-subst-extract-some-isubst*: *extract-subst fct* = *Some*(*x*, *c*, *sb*) ⟹
subst-height-concept (*subst-closure-concept* (*push-isubst-concept c vsbsts*) *sbsts*) < *subst-height-fact fct sbsts*
apply (*clarsimp simp add*: *extract-subst-Some*)
apply (*rule subst-height-concept-mono-closure*)
apply *simp-all*
apply (*insert subst-height-concept-push-isubst-concept* [*of c vsbsts*])
apply (*subgoal-tac 0* < *height-concept c*) **prefer** *2* **apply** *simp* **apply** *arith*

apply (*insert height-concept-push-isubst-concept* [*of c vsbsts*])

apply (*subgoal-tac 0 < height-concept c*) **prefer** *2* **apply** *simp* **apply** *arith*
done

lemma *subst-height-concept-under-closure* [*rule-format*]: ∀ *c1 c2*.
subst-height-concept c1 = *subst-height-concept c2* ⟶ *height-concept c1* = *height-concept c2* ⟶
subst-height-concept (*subst-closure-concept c1 sbsts*) = *subst-height-concept* (*subst-closure-concept c2 sbsts*)
by (*induct sbsts*) *simp-all*

lemma *subst-height-concept-same-length* [*rule-format*]:
length sbsts1 = *length sbsts2* ⟹
∀ *c*. (*subst-height-concept* (*subst-closure-concept c sbsts1*) = *subst-height-concept* (*subst-closure-concept c sbsts2*))
apply(*induct rule*:*list-induct2*)
apply *simp*
apply (*fastforce intro*: *subst-height-concept-under-closure*)
done

lemma *subst-height-fact-same-length*:
length sbsts1 = *length sbsts2* ⟹
subst-height-fact f sbsts1 = *subst-height-fact f sbsts2*
apply (*induct f*)
apply *simp-all*
apply (*erule subst-height-concept-same-length*)
done

lemma *subst-height-form-same-length* [*rule-format*]:
∀ *sbsts1 sbsts2*. *length sbsts1* = *length sbsts2* ⟶
subst-height-form f sbsts1 = *subst-height-form f sbsts2*
apply (*induct f*)
apply *clarsimp+*
apply (*erule subst-height-fact-same-length*)
apply *clarify* **apply** (*simp* (*no-asm*)) **apply** *fast*
apply *clarify* **apply** (*drule spec*)+ **apply** (*drule mp*, *assumption*)+ **apply** *simp*
apply *clarify* **apply** (*drule spec*)+ **apply** (*drule mp*, *assumption*)+ **apply** *simp*
apply *clarify*
apply (*simp* (*no-asm*))
apply (*drule spec*)+ **apply** (*drule mp*) **prefer** *2* **apply** *assumption* **apply** *simp*
done

termination *push-subst-form*
apply (*relation measures* [(λ*p*. (*subst-height-form* (*fst p*) (*snd p*))), (λ*p*.(*height-form* (*fst p*)))])
apply *simp-all*

apply (*simp add*: *push-subst-fact-decr*)

apply (*clarsimp simp add*: *push-subst-extract-some-SubstC*) **apply** (*clarsimp simp only*: *height-fact-extract-some-decr*)

apply (*clarsimp simp add*: *push-subst-extract-some-SubstC*) **apply** (*erule push-subst-extract-some-isubst*)

apply *arith*
apply *arith*
apply (*rule disjI2*) **apply** (*rule subst-height-form-same-length*) **apply** *simp*
done

12.2 Structural correctness of pushing substitutions

fun *subst-hidden-in-concept* :: (*'nr*, *'nc*, *'ni*) *concept* ⇒ *bool* **where**
  subst-hidden-in-concept (*SubstC c sb*) = *False*
| *subst-hidden-in-concept* - = *True*

fun *subst-hidden-in-fact* :: (*'nr*, *'nc*, *'ni*) *fact* ⇒ *bool* **where**
  subst-hidden-in-fact (*Inst x c*) = *subst-hidden-in-concept c*
| *subst-hidden-in-fact* (*AtomR sign r x y*) = *True*
| *subst-hidden-in-fact* (*Eq sign x y*) = *True*

fun *subst-hidden-in-form* :: (*'nr*, *'nc*, *'ni*) *form* ⇒ *bool* **where**
  subst-hidden-in-form FalseFm = *True*
| *subst-hidden-in-form* (*FactFm fct*) = *subst-hidden-in-fact fct*
| *subst-hidden-in-form* (*NegFm f*) = *subst-hidden-in-form f*
| *subst-hidden-in-form* (*BinopFm bop f1 f2*) = (*subst-hidden-in-form f1* ∧ *subst-hidden-in-form f2*)
| *subst-hidden-in-form* (*QuantifFm q f*) = *subst-hidden-in-form f*
| *subst-hidden-in-form* (*SubstFm f sb*) = *False*

lemma *extract-subst-subst-hidden-in-fact*:
  extract-subst fct = *None* ⟹ *subst-hidden-in-fact fct*
apply (*case-tac fct*)
apply (*case-tac concept*)
apply *clarsimp+*
done

lemma *subst-hidden-in-form-push-subst-form* [*simp*]:
  subst-hidden-in-form (*push-subst-form fm sbsts*)
apply (*induct fm sbsts rule*: *push-subst-form.induct*)
apply *simp-all*
apply (*simp split add*: *option.split list.split subst.split*)
apply (*intro conjI impI allI*)
apply (*clarsimp simp add*: *extract-subst-subst-hidden-in-fact*)+
done

12.3 Semantics-preservation of pushing substitutions

lemma *quantif-free-push-isubst-fact* [*simp*]:
quantif-free (*push-isubst-fact v1 v2 fct*)
by (*case-tac fct*) *auto*

lemma *quantif-free-push-rsubst-fact* [*simp*]:
quantif-free (*push-rsubst-fact r rop p fct*)
apply (*case-tac fct*)
apply *simp-all*
apply (*case-tac concept*)
apply *simp-all*
apply (*case-tac rop*)
apply *simp-all*
apply (*case-tac p*, *simp-all add*: *IfThenElseFm-def ImplFm-def*)
apply (*case-tac p*, *simp-all add*: *IfThenElseFm-def ImplFm-def*)
apply *clarsimp*
apply (*rename-tac x nro n c v1 v2*)
apply (*case-tac n*) **apply** *simp-all*
apply (*case-tac nro*) **apply** *simp-all*
apply (*simp-all add*: *IfThenElseFm-def ImplFm-def*)
apply (*clarsimp simp add*: *split-def subst-AtomR-RDiff-def Let-def subst-AtomR-RAdd-def split add*: *role-op.split*)+
done

lemma *quantif-free-push-subst-fact* [*simp*]: *quantif-free* (*push-subst-fact fct sb*)
by (*case-tac sb*) *simp-all*

definition *var-pair-to-substs vps* = (*map* (λ (*v1*, *v2*). *ISubst v1 v2*) *vps*)

lemma *var-pair-to-substs-nil* [*simp*]: *var-pair-to-substs* [] = []
by (*simp add*: *var-pair-to-substs-def*)

lemma *var-pair-to-substs-cons* [*simp*]:
var-pair-to-substs ((*v1*, *v2*)#*sbsts*) = (*ISubst v1 v2*) # (*var-pair-to-substs sbsts*)
by (*simp add*: *var-pair-to-substs-def*)

lemma *interp-form-SubstFm-FactFm-Rel-AtomR-RDiff*:
interp-fact (*AtomR sign r x y*) (*interp-subst* (*RSubst r RDiff* (*v1*, *v2*)) *i*) =
interp-form (*subst-AtomR-RDiff sign r x y v1 v2*) *i*
by (*simp add*: *subst-AtomR-RDiff-def interp-r-modif-def*) *fast*

lemma *interp-form-SubstFm-FactFm-Rel-AtomR-RAdd*:
interp-fact (*AtomR sign r x y*) (*interp-subst* (*RSubst r RAdd* (*v1*, *v2*)) *i*) =
interp-form (*subst-AtomR-RAdd sign r x y v1 v2*) *i*
by (*simp add*: *subst-AtomR-RAdd-def interp-r-modif-def*)

lemma *interp-form-push-rsubst-fact-AtomR*:
interp-form (*push-rsubst-fact r rop* (*v1*, *v2*) (*AtomR sign r′ x1 x2*)) *i* =

interp-fact (*AtomR sign r′ x1 x2*) (*interp-subst* (*RSubst r rop* (*v1*, *v2*)) *i*)
apply (*case-tac r* = *r′*)
apply (*case-tac rop*)
apply (*simp only*: *interp-form-SubstFm-FactFm-Rel-AtomR-RDiff*) **apply** *simp*
apply (*simp only*: *interp-form-SubstFm-FactFm-Rel-AtomR-RAdd*) **apply** *simp*
apply *simp*
done

lemma *interp-form-push-rsubst-concept-numrestrc-RAdd*:
interp-fact (*Inst x* (*NumRestrC nro* (*Suc n*) *r c*)) (*interp-subst* (*RSubst r RAdd* (*v1*, *v2*)) *i*) =
interp-form (*push-rsubst-concept-numrestrc RAdd* (*v1*, *v2*) *x nro* (*Suc n*) *r c*) *i*
apply (*simp only*: *push-rsubst-concept-numrestrc.simps*)

apply (*simp only*: *interp-form.simps interp-form-IfThenElseFm lift-ite-def*)
apply (*split split-if*)
apply (*intro impI conjI*)
apply (*clarsimp simp add*: *interp-numres-ord-Eq-RAdd*)
apply (*simp only*: *interp-form.simps de-Morgan-conj*)
apply (*elim disjE*)
apply (*clarsimp simp add*: *rel-restrict-diff rel-restrict-insert interp-r-modif-def*)

apply (*clarsimp simp add*: *rel-restrict-diff rel-restrict-insert interp-r-modif-def*)
apply (*simp add*: *insert-absorb*)
done

lemma *interp-form-push-rsubst-concept-numrestrc-RDiff*:
interp-fact (*Inst x* (*NumRestrC nro n r c*)) (*interp-subst* (*RSubst r RDiff* (*v1*, *v2*)) *i*) =
interp-form (*push-rsubst-concept-numrestrc RDiff* (*v1*, *v2*) *x nro n r c*) *i*
apply (*simp only*: *push-rsubst-concept-numrestrc.simps*)
apply (*simp only*: *interp-form.simps interp-form-IfThenElseFm lift-ite-def*)
apply (*split split-if*)
apply (*intro impI conjI*)
apply (*clarsimp simp add*: *interp-numres-ord-Eq-RDiff*)
apply (*simp only*: *interp-form.simps de-Morgan-conj*)
apply (*elim disjE*)
apply (*clarsimp simp add*: *rel-restrict-diff rel-restrict-insert interp-r-modif-def*)+
done

lemma *interp-form-push-rsubst-concept-numrestrc*:
interp-fact (*Inst x* (*NumRestrC nro n r c*)) (*interp-subst* (*RSubst r rop* (*v1*, *v2*)) *i*) =
interp-form (*push-rsubst-concept-numrestrc rop* (*v1*, *v2*) *x nro n r c*) *i*
apply (*case-tac rop*)
apply (*simp only*: *interp-form-push-rsubst-concept-numrestrc-RDiff*)
apply (*case-tac n*)
apply (*case-tac nro*)

apply *simp*
apply *simp*
apply (*simp only*: *interp-form-push-rsubst-concept-numrestrc-RAdd*)
done

lemma *interp-form-push-rsubst-fact-Inst*:
interp-fact (*Inst x c*) (*interp-subst* (*RSubst r rop* (*v1*, *v2*)) *i*) =
interp-form (*push-rsubst-concept r rop* (*v1*, *v2*) *x c*) *i*
proof (*cases c*)
case (*NumRestrC numres-ord nat nr concept*) **thus** *?thesis*
apply (*simp only*: *push-rsubst-concept.simps split add*: *split-if*)
apply (*intro conjI impI*)
apply (*simp only*: *interp-form-push-rsubst-concept-numrestrc*)
by *simp*
qed *simp-all*

lemma *interp-form-push-rsubst-fact*:
interp-form (*push-rsubst-fact r rop v1v2 fct*) *i* = *interp-fact fct* (*interp-subst* (*RSubst r rop v1v2*) *i*)
apply (*case-tac v1v2*)
apply (*case-tac fct*)
apply (*simp only*: *push-rsubst-fact.simps interp-form-push-rsubst-fact-Inst*)
apply (*simp only*: *interp-form-push-rsubst-fact-AtomR*)
apply *simp*
done

lemma *interp-subst-RSubst-ISubst*:
interp-subst (*RSubst nr role-op* (*x1*, *x2*)) (*interp-subst* (*ISubst v1 v2*) *i*) =
interp-subst (*ISubst v1 v2*) (*interp-subst* (*RSubst nr role-op* (*replace-var v1 v2 x1*, *replace-var v1 v2 x2*)) *i*)
apply (*simp add*: *interp-i-modif-def interp-r-modif-def*)
apply (*cases i*)
apply (*case-tac role-op*, *auto*)+
done

lemma *interp-c-interp-subst-closure-var-pair-to-substs* [*simp*]:
interp-c (*interp-subst-closure* (*var-pair-to-substs sbsts*) *i*) *a* = *interp-c i a*
by (*induct sbsts*) (*simp add*: *var-pair-to-substs-def split-def*)+

lemma *interp-r-interp-subst-closure-var-pair-to-substs* [*simp*]:
(*interp-r* (*interp-subst-closure* (*var-pair-to-substs sbsts*) *i*) *r*) = *interp-r i r*
by (*induct sbsts*) (*simp add*: *var-pair-to-substs-def split-def*)+

lemma *interp-subst-RSubst* [*rule-format*]: ∀ *v1 v2 i*.
(*interp-subst* (*RSubst nr role-op* (*v1*, *v2*)) (*interp-subst-closure* (*var-pair-to-substs sbsts*) *i*))
=

(*interp-subst-closure* (*var-pair-to-substs sbsts*)
(*interp-subst* (*RSubst nr role-op* (*subst-vars sbsts v1*, *subst-vars sbsts v2*)) *i*))
apply (*induct sbsts*)
apply *simp*

apply (*rename-tac a sbsts*)
apply (*rule allI*)+
apply (*case-tac a*)
apply (*simp del*: *interp-subst.simps add*: *interp-subst-RSubst-ISubst*)
done

lemma *interp-concept-push-isubst-concept*:
fixes *c*
shows (*interp-concept* (*push-isubst-concept c sbsts*) *i* =
interp-concept c (*interp-subst-closure* (*var-pair-to-substs sbsts*) *i*))
proof (*induct c arbitrary*: *sbsts i*)
case (*SubstC c subst*) **show** *?case*
proof (*simp* (*no-asm-use*), *induct subst*)
case (*ISubst ni1 ni2*)
have *interp-concept* (*push-isubst-concept* (*SubstC c* (*ISubst ni1 ni2*)) *sbsts*) *i* =
interp-concept (*push-isubst-concept c* ((*ni1*, *ni2*) # *sbsts*)) *i* **by** *simp*
also have ... = *interp-concept c* (*interp-subst-closure* (*var-pair-to-substs* ((*ni1*, *ni2*) # *sbsts*)) *i*)
by (*simp add*: *SubstC.hyps*)
finally show *?case* **by** (*simp add*: *var-pair-to-substs-def del*: *interp-subst.simps*)
next
case (*RSubst nr role-op prod*) **show** *?case*
apply (*case-tac prod*)
apply (*simp add*: *SubstC.hyps del*: *interp-subst.simps*)
by (*simp only*: *interp-subst-RSubst*)
qed
qed *simp-all*

lemma *interp-form-push-isubst-fact*:
interp-form (*push-isubst-fact v1 v2 fct*) *i* = *interp-fact fct* (*interp-subst* (*ISubst v1 v2*) *i*)
apply (*case-tac fct*)
apply (*simp add*: *interp-concept-push-isubst-concept*)
apply *simp*+
done

lemma *interp-form-push-subst-fact*:
interp-form (*push-subst-fact fct sb*) *i* = *interp-fact fct* (*interp-subst sb i*)
apply (*case-tac sb*)
apply (*simp del*: *interp-subst.simps add*: *interp-form-push-rsubst-fact*)

apply (*simp del*: *interp-subst.simps add*: *interp-form-push-isubst-fact*)
done

lemma *interp-form-push-subst-form-sbsts* [*rule-format*]:
shows *interp-form* (*push-subst-form fm sbsts*) *i* = *interp-form fm* (*interp-subst-closure sbsts i*)
proof (*induction fm sbsts arbitrary*: *i rule*: *push-subst-form.induct*)
 case *1* **show** *?case* **by** *simp*
next
 case *3* **thus** *?case* **by** *simp*
next
 case (*4 bop f1 f2 sbsts i*) **thus** *?case*
  by (*simp*, *case-tac bop*, *simp*+)
next
 case (*5 q f sbsts i*) **thus** *?case*
  by (*case-tac q*, (*simp add*: *interp-subst-closure-lift-subst*)+)
next
 case (*6 f sb sbsts i*) **thus** *?case*
  by *simp*
next
 case (*2 fct sbsts i*) **show** *?case*
 proof (*cases extract-subst fct*)
  case *None*
  hence *esn*: *extract-subst fct* = *None* **by** *simp*
  thus *?thesis*
  proof (*cases sbsts*)
   case *Nil* **thus** *?thesis* **by** (*simp add*: *esn*)
  next
   case (*Cons sb sbsts′*)
   hence *sbsts* = *sb* # *sbsts′* **by** *simp*
   moreover hence *interp-form* (*push-subst-form* (*push-subst-fact fct sb*) *sbsts′*) *i* =
    interp-form (*push-subst-fact fct sb*) (*interp-subst-closure sbsts′ i*)
    by (*simp add*: *2.IH esn*)
   ultimately show *?thesis* **by** (*simp add*: *esn interp-form-push-subst-fact*)
  qed
 next
  case (*Some a*)
  hence *esn0*: *extract-subst fct* = *Some a* **by** *simp*
  thus *?thesis*
  proof (*cases a*) **case** (*fields x c sb*)
   hence *esn*: *extract-subst fct* = *Some*(*x*, *c*, *sb*) **by** (*simp add*: *esn0*)
   moreover from *esn* **have** *fctInstSubstC*: (*fct* = (*Inst x* (*SubstC c sb*))) **by** (*simp add*: *extract-subst-Some*)
   thus *?thesis*
   proof (*cases sb*)
    case (*RSubst nr role-op prod*)
    hence *sbRSubst*: *sb* = *RSubst nr role-op prod* **by** *simp*

moreover have *IH-RSubst*:
interp-form (*push-subst-form* (*FactFm* (*Inst x c*)) (*RSubst nr role-op prod* # *sbsts*)) *i* =
interp-form (*FactFm* (*Inst x c*)) (*interp-subst-closure* (*RSubst nr role-op prod* # *sbsts*) *i*)
by (*simp add*: *2.IH sbRSubst esn del*: *interp-form.simps push-subst-form.simps*)
show *?thesis*
apply (*simp only*: *fctInstSubstC*)
apply (*subst push-subst-form.simps*)
apply (*simp add*: *esn sbRSubst IH-RSubst del*: *interp-subst-closure.simps push-subst-form.simps interp-fact.simps*)
by (*simp del*: *push-subst-form.simps*)
next
case (*ISubst ni1 ni2*)
hence *sbISubst*: *sb* = *ISubst ni1 ni2* **by** *simp*
moreover have *IH-ISubst*:
interp-form (*push-subst-form* (*FactFm* (*Inst x* (*push-isubst-concept c* [(*ni1*, *ni2*)]))) *sbsts*) *i* =
interp-form (*FactFm* (*Inst x* (*push-isubst-concept c* [(*ni1*, *ni2*)]))) (*interp-subst-closure sbsts i*)
by (*simp add*: *2.IH sbISubst esn del*: *interp-form.simps push-subst-form.simps*)
show *?thesis*
apply (*simp only*: *fctInstSubstC*)
apply (*subst push-subst-form.simps*)
apply (*simp add*: *esn sbISubst IH-ISubst del*: *interp-subst-closure.simps push-subst-form.simps*)
by (*simp add*: *interp-concept-push-isubst-concept*)
qed
qed
qed
qed

lemma *interp-form-push-subst-form*:
interp-form (*push-subst-form fm* []) *i* = *interp-form fm i*
by (*simp add*: *interp-form-push-subst-form-sbsts*)

end

References

[1] M. Chaabani, R. Echahed, and M. Strecker. Logical foundations for reasoning about transformations of knowledge bases. Motivation and informal outline of proof development, see http://www.irit.fr/~Martin.Strecker/Publications/dl_transfo2013.pdf, June 2013.

[2] M. Chaabani, M. Mezghiche, and M. Strecker. Vérification d'une

méthode de preuve pour la logique de description $\mathcal{ALC}$. In Y. Ait-Ameur, editor, *Proc. 10ème Journées Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, pages 149–163, June 2010.