Sorted Rewriting, Conditional Rewriting, and Logically Constrained Rewriting

Akihisa Yamada

National Institute of Advanced Industrial Science and Technology, Japan

May 8, 2026

Abstract

This entry provides various materials for sorted term rewrite systems (sorted TRSs), (sorted) conditional TRSs (CTRSs), and logically constrained TRSs (LCTRSs). For (C)TRSs we formalize the fundamental result that the rewrite steps induced by a (C)TRS is the least rewrite relation that models the (C)TRS. For LCTRSs we simply formulate logics as sorted algebras with the bool sort and logical symbols which are interpreted as expected. This allows us to define rewrite steps of LCTRSs as rewrite steps of an (infinite) TRS.

Contents

1 Introduction

This entry provides various materials needed for *logically constrained term rewrite systems* (LCTRS) [2], a term rewrite system (TRS) where rules come with constraints which are interpreted in a background logic.

Building upon the IsaFoR (Isabelle Formalization of Rewriting [4]) formalizations of unsorted terms [3] and sorted terms [6], we start with formalizing sorted TRSs. Here we formalize fundamental results such as that the rewrite relation induced by a sorted TRS is the least rewrite relation which models the TRS. We also formalize (sorted) conditional TRSs (CTRSs) and the fundamental result that the many-step reduction is the least rewrite preorder which models the CTRS.

As LCTRSs require *logics*, we provide a light-weight and rewriting-oriented formalization of logics. While it is standard to treat logical symbols and function symbols differently—cf. the IsaFoL (Isabelle Formalization of Logics) entry [1]—in this development we consider a logic just as a sorted algebra where the signature contains logic symbols and their interpretations are as expected. This became possible thanks to the formalization of sorted terms, and is crucial for LCTRSs since the formalism allows logical symbols to appear inside terms.

We also provide means to extend an algebra with another algebra, and to extend an algebra with logic. This allows us to build concrete logics modularly: we define algebras for natural numbers and integers, and obtain corresponding logics by extending these algebras with a pure Bool logic.

2 More About Binary Relations

theory *Binary-Relations-More*
imports *Complete-Non-Orders.Well-Relations Main HOL−Library.FuncSet Abstract−Rewriting.Abstract-Rewriting*
begin

2.1 Missing Lemmas

lemma *relpowp-Suc'*: $r \hat{\ } (Suc\ n) = r\ OO\ r \hat{\ } n$
⟨*proof*⟩

lemma *tranclp-greater*: $r \leq r^{++}$ ⟨*proof*⟩

lemma *relation-ofI*:
assumes $r\ x\ y$ **and** $x \in X$ **and** $y \in X$
shows $(x,y) \in$ *relation-of* $r\ X$
⟨*proof*⟩

lemma *relation-ofE*:
assumes $(x,y) \in$ *relation-of* $r\ X \Longrightarrow r\ x\ y \Longrightarrow x \in X \Longrightarrow y \in X \Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *in-relation-of-UNIV*[*simp*]: $(x,y) \in$ *relation-of* r *UNIV* $\longleftrightarrow r\ x\ y$
⟨*proof*⟩

lemma *relation-of-mono*:
assumes $r \leq s\ X \subseteq Y$ **shows** *relation-of* $r\ X \subseteq$ *relation-of* $s\ Y$
⟨*proof*⟩

lemmas *relation-of-subrel* = *relation-of-mono*[*OF - subset-refl*]
lemmas *relation-of-subset* = *relation-of-mono*[*OF order.refl*]

lemma *relation-of-sup*: *relation-of* $(r \sqcup s)\ X =$ *relation-of* $r\ X \cup$ *relation-of* $s\ X$
⟨*proof*⟩

lemma *relation-of-UNIV-OO*:
relation-of $(r\ OO\ s)$ *UNIV* = *relation-of* r *UNIV* O *relation-of* s *UNIV*
⟨*proof*⟩

lemma *relation-of-UNIV-trancl*:
relation-of $(tranclp\ r)$ *UNIV* = *trancl* (*relation-of* r *UNIV*)
⟨*proof*⟩

lemma *relation-of-UNIV-rtrancl*:
relation-of $(rtranclp\ r)$ *UNIV* = *rtrancl* (*relation-of* r *UNIV*)
⟨*proof*⟩

lemma *rtrancl-relation-of*: $(relation\text{-}of\ r\ UNIV)^* = relation\text{-}of\ (rtranclp\ r)\ UNIV$
⟨*proof*⟩

lemma *in-rel-un*: *in-rel* $(r \cup s) =$ *in-rel* $r \sqcup$ *in-rel* s
⟨*proof*⟩

lemma *in-rel-relcomp*: *in-rel* $(r\ O\ s) =$ *in-rel* $r\ OO$ *in-rel* s
⟨*proof*⟩

lemma *in-rel-trancl*: *in-rel* $(S^+) = (\textit{in-rel}\ S)^{++}$
⟨*proof*⟩

lemma *in-rel-rtrancl*: *in-rel* $(S^*) = (\textit{in-rel}\ S)^{**}$
⟨*proof*⟩

lemma *tranclpD2*: *tranclp* $r\ x\ y \Longrightarrow \exists z.\ \textit{rtranclp}\ r\ x\ z \wedge r\ z\ y$
⟨*proof*⟩

lemma *tranclp-mono*:
assumes *rs*: $r \leq s$ **shows** *tranclp* $r \leq \textit{tranclp}\ s$
⟨*proof*⟩

lemma *mono-tranclp*[*mono*]:
$(\bigwedge x\ y.\ r\ x\ y \longrightarrow s\ x\ y) \Longrightarrow \textit{tranclp}\ r\ x\ y \longrightarrow \textit{tranclp}\ s\ x\ y$
⟨*proof*⟩

lemmas *tranclp-subrel* = *tranclp-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *rule-format*]

lemmas *rtranclp-subrel* = *rtranclp-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *rule-format*]

lemma [*simp*]:
shows *tranclp-tranclp*: *tranclp* $(\textit{tranclp}\ r) = \textit{tranclp}\ r$
⟨*proof*⟩

lemma *rtranclp-iff-tranclp*: *rtranclp* $r\ s\ t \longleftrightarrow s = t \vee \textit{tranclp}\ r\ s\ t$
⟨*proof*⟩

lemma *tranclp-imp-relpow*: $r^{++}\ x\ y \Longrightarrow \exists n{>}0.\ (r \widehat{\ } n)\ x\ y$ **for** $x\ y$
⟨*proof*⟩

lemma *relpow-imp-tranclp*:
assumes *xy*: $(r \widehat{\ } n)\ x\ y$ **shows** $n = 0 \wedge x = y \vee r^{++}\ x\ y$
⟨*proof*⟩

lemma *tranclp-is-Sup-relpowp*: $r^{++} = (\bigsqcup n{\in}\{0{<}..\}.\ r \widehat{\ } n)$
⟨*proof*⟩

lemma *relpowp-dual*: **fixes** $r :: 'a \Rightarrow 'a \Rightarrow \textit{bool}$ **shows** $r^-\ \widehat{\ }\ n = (r\ \widehat{\ }\ n)^-$
⟨*proof*⟩

lemma *symclp-mono*: $r \leq s \Longrightarrow \textit{symclp}\ r \leq \textit{symclp}\ s$
⟨*proof*⟩

lemma *wf-iff-wfP*: *wf* $R \longleftrightarrow \textit{wfP}\ (\textit{in-rel}\ R)$
⟨*proof*⟩

lemma *wfP-trancl*: *wfP* $r^{++} \longleftrightarrow$ *wfP r*
⟨*proof*⟩

lemma *rtranclp-relcomp-rtranclp-le*:
(*rtranclp r OO rtranclp r*) $\leq$ *rtranclp r*
⟨*proof*⟩

lemma *wfP-iff-nonempty-minimal*: *wfP r* $\longleftrightarrow$ ($\forall X.\ X \neq \{\} \longrightarrow (\exists x \in X.\ \forall y \in X.\ \neg\ r\ y\ x)$)
⟨*proof*⟩

lemmas *wfP-imp-nonempty-minimal* = *wfP-iff-nonempty-minimal*[*THEN iffD1*, *rule-format*]

lemmas *nonempty-minimal-imp-wfP* = *wfP-iff-nonempty-minimal*[*THEN iffD2*, *rule-format*]

lemma *quasi-commute-imp-SN-Un-iff*: *quasi-commute r s* $\Longrightarrow$ *SN* $(r \cup s) \longleftrightarrow$ *SN r* $\wedge$ *SN s*
⟨*proof*⟩

lemma *symclp-symmetric*: *symmetric A* (*symclp r*) ⟨*proof*⟩

context *reflexive* **begin**

interpretation *tranclp*: *quasi-ordered-set A tranclp* ($\sqsubseteq$)
⟨*proof*⟩

lemmas *tranclp-quasi-order* = *tranclp.quasi-ordered-set-axioms*
and *tranclp-reflexive* = *tranclp.reflexive-axioms*

interpretation *symclp*: *tolerance A symclp* ($\sqsubseteq$)
⟨*proof*⟩

lemmas *symclp-tolerance* = *symclp.tolerance-axioms*
and *symclp-reflexive* = *symclp.reflexive-axioms*

end

2.1.1 Extending Binary Relations AFP Entry

lemma *sup-Restrp*: $r \sqcup s \upharpoonright A = (r \upharpoonright A) \sqcup (s \upharpoonright A)$
⟨*proof*⟩

lemma *tranclp-Restrp-mem*[*simp*]:
assumes *tranclp* $(s \upharpoonright A)$ *x y*

shows $x \in A$ $y \in A$
⟨*proof*⟩

lemma *Restrp-dual*: $(r \restriction A)^- = r^- \restriction A$
⟨*proof*⟩

lemma *Restrp-mono*: **assumes** *rs*: $r \leq s$ **and** *AB*: $A \subseteq B$ **shows** $r \restriction A \leq s \restriction B$
⟨*proof*⟩

lemmas *Restrp-subrel* = *Restrp-mono*[*OF - subset-refl*]
lemmas *Restrp-subset* = *Restrp-mono*[*OF order.refl*]

interpretation *reflclp*: *reflexive UNIV reflclp r*
rewrites $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$
and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

lemmas *symclp-cases*[*consumes 1*, *case-names fw bw*, *elim*!] = *symclp-def*[*unfolded atomize-eq*, *THEN iffD1*, *THEN disjE*]

interpretation *symclp*: *symmetric UNIV symclp r*
rewrites $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$
and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

interpretation *tranclp*: *transitive UNIV tranclp r*
rewrites $(sympartp\ (tranclp\ r))^- \equiv sympartp\ (tranclp\ r)$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$
and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

interpretation *rtranclp*: *quasi-ordered-set UNIV rtranclp r*
rewrites $(sympartp\ (rtranclp\ r))^- \equiv sympartp\ (rtranclp\ r)$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$
and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

interpretation *transymclp*: *partial-equivalence UNIV tranclp* (*symclp r*)
rewrites $(sympartp\ (rtranclp\ r))^- \equiv sympartp\ (rtranclp\ r)$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$

and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

interpretation *rtransymclp*: *equivalence UNIV rtranclp* (*symclp r*)
rewrites $(sympartp\ (rtranclp\ (symclp\ r)))^{-} \equiv sympartp\ (rtranclp\ (symclp\ r))$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$
and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

lemma (**in** *symmetric*) *reflclp-symmetric*: *symmetric A* (*reflclp* (∼))
⟨*proof*⟩

locale *compatible* =
related-set + *less-syntax* +
assumes *strict-implies-weak*: $x \sqsubset y \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow x \sqsubseteq y$
assumes *weak-strict-trans*[*trans*]: $x \sqsubseteq y \Longrightarrow y \sqsubset z \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow z \in A \Longrightarrow x \sqsubset z$
assumes *strict-weak-trans*[*trans*]: $x \sqsubset y \Longrightarrow y \sqsubseteq z \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow z \in A \Longrightarrow x \sqsubset z$
begin

sublocale *strict*: *transitive A* (⊏)
⟨*proof*⟩

end

interpretation *rtranclp*: *compatible UNIV rtranclp r tranclp r*
rewrites $(sympartp\ (tranclp\ r))^{-} \equiv sympartp\ (tranclp\ r)$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge X.\ X \subseteq UNIV \equiv True$
and $\bigwedge P1.\ (True \Longrightarrow PROP\ P1) \equiv PROP\ P1$
and $\bigwedge P1.\ (True \Longrightarrow P1) \equiv Trueprop\ P1$
⟨*proof*⟩

context *compatible-ordering* **begin**

sublocale *compatible*
⟨*proof*⟩

end

lemma *trans-relation-of*: *trans* (*relation-of r A*) ⟷ *transitive A r*
⟨*proof*⟩

2.1.2 Relation defined precisely on a set

locale *relation-on* = *related-set* +
  assumes *mem-left*: $a \sqsubseteq b \Longrightarrow a \in A$
    and *mem-right*: $a \sqsubseteq b \Longrightarrow b \in A$
begin

lemma *Restrp-eq*[*simp*]: $(\sqsubseteq)\restriction A = (\sqsubseteq)$
  ⟨*proof*⟩

lemma *dual-relation-on*: *relation-on* A $(\sqsubseteq)^-$
  ⟨*proof*⟩

lemma *symmetric-imp-symclp-eq*:
  assumes *symmetric* A $(\sqsubseteq)$
  shows *symclp* $(\sqsubseteq) = (\sqsubseteq)$
⟨*proof*⟩

lemma *tranclp-relation-on*: *relation-on* A $(\sqsubseteq)^{++}$
⟨*proof*⟩

lemma *symclp-relation-on*: *relation-on* A (*symclp* $(\sqsubseteq)$)
  ⟨*proof*⟩

lemma *symmetric-UNIV*: *symmetric* A $(\sqsubseteq) \longleftrightarrow$ *symmetric UNIV* $(\sqsubseteq)$
⟨*proof*⟩

end

interpretation *Restrp*: *relation-on* A $r\restriction A$
  ⟨*proof*⟩

interpretation *Restrp.tranclp*: *relation-on* A *tranclp* $(r\restriction A)$
  ⟨*proof*⟩

lemma *relation-on-sup*:
  relation-on A $(r \sqcup s) \longleftrightarrow$ *relation-on* A $r \wedge$ *relation-on* A s
  ⟨*proof*⟩

2.1.3 Closed Relations

locale *relation-closed* = *related-set* +
  assumes *closed*: $a \sqsubseteq b \Longrightarrow a \in A \Longrightarrow b \in A$
begin

interpretation *tranclp*: *relation-closed* A $(\sqsubseteq)^{++}$
⟨*proof*⟩

lemmas *tranclp-closed* = *tranclp.relation-closed-axioms*

lemma *reflclp-closed*: *relation-closed A* $(\sqsubseteq)^{==}$
⟨*proof*⟩

lemma *rtranclp-closed*: *relation-closed A* $(\sqsubseteq)^{**}$
⟨*proof*⟩

lemma *tranclp-partial-equivalence*:
assumes *symmetric A* $(\sqsubseteq)$ **shows** *partial-equivalence A* $(\sqsubseteq)^{++}$
⟨*proof*⟩

lemma *tranclp-equivalence*:
assumes *tolerance A* $(\sqsubseteq)$
shows *equivalence A* (*tranclp* $(\sqsubseteq)$)
⟨*proof*⟩

end

2.1.4 Function Space on Specific Input

definition *fun-ord-on* **where** *fun-ord-on I* $(\sqsubseteq)$ *f g* $\equiv \forall\, i \in I.\ f\ i \sqsubseteq g\ i$
for *le* (**infix** $\sqsubseteq$ *50*)

lemmas *fun-ord-onI* = *fun-ord-on-def*[*unfolded atomize-eq*, *THEN iffD2*, *rule-format*]
lemmas *fun-ord-onD* = *fun-ord-on-def*[*unfolded atomize-eq*, *THEN iffD1*, *rule-format*]

lemma *fun-ord-on-empty*[*simp*]: *fun-ord-on* $\{\}$ = $\top$
⟨*proof*⟩

lemma *fun-ord-on-UNIV*[*simp*]: *fun-ord-on UNIV* = *fun-ord*
⟨*proof*⟩

lemma *dual-fun-ord-on*: (*fun-ord-on I r*)$^{-}$ = *fun-ord-on I* r^{-}
⟨*proof*⟩

lemma (**in** *reflexive*) *fun-reflexive*:
reflexive $(I \to A)$ (*fun-ord-on I* $(\sqsubseteq)$)
⟨*proof*⟩

lemma (**in** *irreflexive*) *fun-irreflexive*:
assumes *I0*: $I \neq \{\}$
shows *irreflexive* $(I \to A)$ (*fun-ord-on I* $(\sqsubset)$)
⟨*proof*⟩

lemma (**in** *semiattractive*) *fun-semiattractive*:
semiattractive $(I \to A)$ (*fun-ord-on I* $(\sqsubseteq)$)
⟨*proof*⟩

lemma (**in** *attractive*) *fun-attractive*:
attractive $(I \to A)$ (*fun-ord-on I* $(\sqsubseteq)$)

⟨*proof*⟩

lemma (**in** *transitive*) *fun-transitive*:
transitive (*I* → *A*) (*fun-ord-on I* (⊑))
⟨*proof*⟩

lemma (**in** *symmetric*) *fun-symmetric*:
symmetric (*I* → *A*) (*fun-ord-on I* (∼))
⟨*proof*⟩

lemma (**in** *quasi-ordered-set*) *fun-quasi-order*:
quasi-ordered-set (*I* → *A*) (*fun-ord-on I* (⊑))
⟨*proof*⟩

lemma (**in** *tolerance*) *fun-tolerance*:
tolerance (*I* → *A*) (*fun-ord-on I* (∼))
⟨*proof*⟩

lemma (**in** *equivalence*) *fun-equivalence*:
equivalence (*I* → *A*) (*fun-ord-on I* (∼))
⟨*proof*⟩

2.2 Lists

lemma *dual-list-all2*: (*list-all2 r*)⁻ = *list-all2 r*⁻ ⟨*proof*⟩

lemma (**in** *reflexive*) *lists-reflexive*: *reflexive* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *transitive*) *lists-transitive*: *transitive* (*lists A*) (*list-all2* (⊑)) (**is** *transitive ?A ?r*)
⟨*proof*⟩

lemma (**in** *symmetric*) *lists-symmetric*: *symmetric* (*lists A*) (*list-all2* (∼))
⟨*proof*⟩

lemma (**in** *antisymmetric*) *lists-antisymmetric*: *antisymmetric* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *semiattractive*) *lists-semiattractive*: *semiattractive* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *attractive*) *lists-attractive*: *attractive* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *quasi-ordered-set*) *lists-quasi-ordered-set*: *quasi-ordered-set* (*lists A*) (*list-all2* (⊑))

⟨*proof*⟩

lemma (**in** *near-ordered-set*) *lists-near-ordered-set*: *near-ordered-set* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *pseudo-ordered-set*) *lists-pseudo-ordered-set*: *pseudo-ordered-set* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *partially-ordered-set*) *lists-partially-ordered-set*: *partially-ordered-set* (*lists A*) (*list-all2* (⊑))
⟨*proof*⟩

lemma (**in** *tolerance*) *lists-tolerance*: *tolerance* (*lists A*) (*list-all2* (∼))
⟨*proof*⟩

lemma (**in** *partial-equivalence*) *lists-partial-equivalence*: *partial-equivalence* (*lists A*) (*list-all2* (∼))
⟨*proof*⟩

lemma (**in** *equivalence*) *lists-equivalence*: *equivalence* (*lists A*) (*list-all2* (∼))
⟨*proof*⟩

2.2.1 Reflexive Closure Restricted on Set

definition *id-on* **where** *id-on A x y* ≡ *x* = *y* ∧ *y* ∈ *A*

lemmas *id-onI*[*intro*!] = *id-on-def*[*unfolded atomize-eq*, *THEN iffD2*, *rule-format*]
lemmas *id-onE*[*elim*!] = *id-on-def*[*unfolded atomize-eq*, *THEN iffD1*, *elim-format*, *unfolded conj-imp-eq-imp-imp*]

lemma *id-on-relcompp*[*simp*]: (*id-on A OO r*) *x y* ⟷ *x* ∈ *A* ∧ *r x y*
and *relcompp-id-on*[*simp*]: (*r OO id-on A*) *x y* ⟷ *r x y* ∧ *y* ∈ *A*
⟨*proof*⟩

lemma *id-on-relcompp-id-on-simps*[*simp*]:
id-on A OO id-on B = *id-on* (*A* ∩ *B*)
r OO id-on A OO id-on B = *r OO id-on* (*A* ∩ *B*)
id-on A OO id-on B OO s = *id-on* (*A* ∩ *B*) *OO s*
r OO id-on A OO id-on B OO s = *r OO id-on* (*A* ∩ *B*) *OO s*
⟨*proof*⟩

lemma *Restrp-eq-relcomp*: *r*↾*A* = *id-on A OO r OO id-on A* ⟨*proof*⟩

definition *reflclp-on A r* ≡ *r* ⊔ *id-on A*

lemma *reflclp-onI*: *r x y* ∨ *x* = *y* ∧ *x* ∈ *A* ⟹ *reflclp-on A r x y*
⟨*proof*⟩

lemma *reflclp-onE*[*consumes 1*, *case-names refl base*]:
reflclp-on A r x y $\Longrightarrow$ ($x = y \Longrightarrow x \in A \Longrightarrow$ *thesis*) $\Longrightarrow$ ($r\ x\ y \Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*
⟨*proof*⟩

lemma *reflclp-on-refl*[*simp*]: $x \in A \Longrightarrow$ *reflclp-on A r x x*
and *reflclp-on-base*[*simp*]: $r\ x\ y \Longrightarrow$ *reflclp-on A r x y*
⟨*proof*⟩

lemma *reflclp-on-mono*:
$r \leq s \Longrightarrow$ *reflclp-on A r* $\leq$ *reflclp-on A s*
⟨*proof*⟩

lemmas *reflclp-on-subrel* = *reflclp-on-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *rule-format*]

lemma *mono-reflclp-on*[*mono*]:
($\bigwedge x\ y.\ r\ x\ y \longrightarrow s\ x\ y$) $\Longrightarrow$ *reflclp-on A r x y* $\longrightarrow$ *reflclp-on A s x y*
⟨*proof*⟩

interpretation *reflclp-on*: *reflexive A* ‹*reflclp-on A r*›
⟨*proof*⟩

lemma (**in** *reflexive*) *reflclp-on-id*[*simp*]: *reflclp-on A* ($\sqsubseteq$) = ($\sqsubseteq$)
⟨*proof*⟩

lemma *reflclp-on-UNIV*: *reflclp-on UNIV* = *reflclp*
⟨*proof*⟩

lemma *reflclp-on-Restr*: *reflclp-on A r* $\restriction$ *B* = *reflclp-on* (*A* $\cap$ *B*) (*r* $\restriction$ *B*)
⟨*proof*⟩

lemma (**in** *transitive*) *reflclp-on-transitive*: *transitive A* (*reflclp-on B* ($\sqsubseteq$))
⟨*proof*⟩

lemma (**in** *transitive*) *reflclp-on-quasi-order*: *quasi-ordered-set A* (*reflclp-on A* ($\sqsubseteq$))
⟨*proof*⟩

lemma (**in** *symmetric*) *reflclp-on-symmetric*: *symmetric A* (*reflclp-on B* ($\sim$))
⟨*proof*⟩

lemma (**in** *symmetric*) *reflclp-on-tolerance*: *tolerance A* (*reflclp-on A* ($\sim$))
⟨*proof*⟩

lemma (**in** *partial-equivalence*) *reflclp-equivalence*: *equivalence A* (*reflclp* ($\sim$))
⟨*proof*⟩

lemma (**in** *partial-equivalence*) *reflclp-on-equivalence*: *equivalence A* (*reflclp-on A*

(∼))
⟨*proof*⟩

lemma *sympartp-dual*: (*sympartp r*)$^{-}$ = *sympartp r*
⟨*proof*⟩

Restricted reflexive transitive closures should better be defined as the (unrestricted) transitive closure of restricted reflexive closure, since then facts of transitive closure can be reused.

abbreviation *rtranclp-on A r* ≡ *tranclp* (*reflclp-on A r*)

thm *reflclp-on.tranclp-quasi-order*

interpretation *rtranclp-on*: *reflexive A rtranclp-on A r*
⟨*proof*⟩

lemma *rtranclp-on-eq-reflclp-on*: *rtranclp-on A r* = *reflclp-on A* (*tranclp r*)
⟨*proof*⟩

lemma *rtranclp-on-induct*[*consumes 1*, *case-names refl base step*]:
assumes ∗: *rtranclp-on A r x y*
and *0*: *x* ∈ *A* ⟹ *P x*
and *1*: ⋀*y*. *r x y* ⟹ *P y*
and *step*: ⋀*y z*. *tranclp r x y* ⟹ *r y z* ⟹ *P y* ⟹ *P z*
shows *P y*
⟨*proof*⟩

abbreviation *rsymclp-on A r* ≡ *symclp* (*reflclp-on A r*)

interpretation *rsymclp-on*: *reflexive A rsymclp-on A r*
⟨*proof*⟩

lemma *rsymclp-on-eq-reflclp-on*: *rsymclp-on A r* = *reflclp-on A* (*symclp r*)
⟨*proof*⟩

abbreviation *rtransymclp-on A r* ≡ *tranclp* (*rsymclp-on A r*)

interpretation *rtransymclp-on*: *reflexive A rtransymclp-on A r*
⟨*proof*⟩

lemma *rtransymclp-eq-reflclp*: *rtransymclp-on A r* = *reflclp-on A* (*tranclp* (*symclp r*))
⟨*proof*⟩

lemma *rtransymclp-on-equivalence*: *equivalence A* (*rtransymclp-on A r*)
⟨*proof*⟩

context *relation-on* **begin**

interpretation *reflclp-on*: *relation-on A reflclp-on A* ($\sqsubseteq$)
⟨*proof*⟩

lemmas *reflclp-relation-on* = *reflclp-on.relation-on-axioms*

lemma *relcompp-id-on*[*simp*]: ($\sqsubseteq$) *OO id-on A* = ($\sqsubseteq$)
⟨*proof*⟩

lemma *relcompp-id-on-ext*[*simp*]: ($\sqsubseteq$) *OO id-on A OO r* = ($\sqsubseteq$) *OO r*
⟨*proof*⟩

lemma *id-on-relcompp*[*simp*]: *id-on A OO* ($\sqsubseteq$) = ($\sqsubseteq$)
⟨*proof*⟩

lemma *id-on-relcompp-ext*[*simp*]: *id-on A OO* ($\sqsubseteq$) *OO r* = ($\sqsubseteq$) *OO r*
⟨*proof*⟩

lemma *reflclp-on-eq-reflclp-Restrp*:
reflclp-on A ($\sqsubseteq$) = *reflclp* ($\sqsubseteq$) ↾ *A*
⟨*proof*⟩

end

2.2.2 Dual

lemma *dual-le-symclp*: $r^- \leq$ *symclp r*
⟨*proof*⟩

lemma *list-all2-dual*: *list-all2* r^- *xs ys* $\Longrightarrow$ *list-all2 r ys xs*
⟨*proof*⟩

lemma *reflclp-dual*: (*reflclp r*)$^-$ = *reflclp* r^- ⟨*proof*⟩

lemma *rtranclp-dual*: (*rtranclp r*)$^-$ = *rtranclp* r^-
⟨*proof*⟩

lemma *tranclp-dual*: (*tranclp r*)$^-$ = *tranclp* r^-
⟨*proof*⟩

lemma *reflclp-on-dual*: (*reflclp-on A r*)$^-$ = *reflclp-on A* r^-
⟨*proof*⟩

lemma *symclp-dual*: (*symclp r*)$^-$ = *symclp r*
⟨*proof*⟩

lemma *tranclp-le-transymclp*: *tranclp r* $\leq$ *tranclp* (*symclp r*)
⟨*proof*⟩

lemmas *tranclp-imp-transymclp* =

tranclp-le-transymclp[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *rule-format*]

lemma *dual-tranclp-le-transymclp*: $(tranclp\ r)^- \leq tranclp\ (symclp\ r)$
⟨*proof*⟩

lemmas *dual-tranclp-imp-transymclp* =
dual-tranclp-le-transymclp[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *rule-format*]

end

3 More About Sorted Terms

theory *Sorted-Terms-More*
imports *Sorted-Terms.Sorted-Contexts*
begin

declare *domIff*[*iff del*]

lemma *grounding-subst*:
assumes ϑ: $\vartheta :_s X\ |`\ XG \to \mathcal{T}(F,\emptyset)$ **and** *var*: $\forall x \in dom\ X - XG.\ \vartheta\ x = Var\ x$
shows $\vartheta :_s X \to \mathcal{T}(F,\ X\ |`(-XG))$
⟨*proof*⟩

Below defines an algebra where every element has a representation as a ground term. To avoid complication in the type system, the type of variables in the ground representations must be *unit*.

locale *sorted-algebra-represented* = *sorted-algebra* +
term-of-unit: *sorted-map* η A $\mathcal{T}(F)$ **for** η +
assumes *eval-term-of-unit*: $\bigwedge a\ \sigma\ \alpha.\ a : \sigma\ in\ A \Longrightarrow I[\![\eta\ a]\!]\alpha = a$
begin

Because locales do not support polymorphic parameters, we need to extend η in polymorphic to the type of variables.

abbreviation *term-of* **where** *term-of a* ≡ η *a* · *undefined*

sublocale *term-of*: *sorted-map term-of A* $\mathcal{T}(F,V)$ **for** V
⟨*proof*⟩

lemma *eval-term-of*: **assumes** $a : \sigma\ in\ A$ **shows** $I[\![term\text{-}of\ a]\!]\alpha = a$
⟨*proof*⟩

lemma *map-eval-term-of*: $ds :_l \sigma s\ in\ A \Longrightarrow [I[\![term\text{-}of\ d]\!]\alpha.\ d \leftarrow ds] = ds$
⟨*proof*⟩

lemma *eval-subst-term-of*:
assumes *s*: $s : \sigma\ in\ \mathcal{T}(F,V)$ **and** α: $\alpha :_s V \to A$
shows $I[\![s{\cdot}(term\text{-}of{\circ}\alpha)]\!]\beta = I[\![s]\!]\alpha$
⟨*proof*⟩

lemma *eval-upd-as-subst*:
assumes *s*: $s : \sigma$ *in* $\mathcal{T}(F, V)$
and α: $\alpha :_s V \to A$ **and** *a*: $a : \tau$ *in* A
shows $I[\![s]\!]\alpha(v{:=}a) = I[\![s \cdot Var(v{:=}\textit{term-of } a)]\!]\alpha$
⟨*proof*⟩

lemma *term-of-actxt*:
assumes $C : \sigma \to \tau$ *in aContext* F A
shows $I\langle I[\![\textit{map-args-actxt term-of } C]\!]_c\ \alpha; a\rangle = I\langle C; a\rangle$
⟨*proof*⟩

lemma *term-of-subst*: $a : \sigma$ *in* $A \Longrightarrow \textit{term-of } a \cdot \vartheta = \textit{term-of } a$
⟨*proof*⟩

lemma *term-of-unit-subst*: $a : \sigma$ *in* $A \Longrightarrow \eta\ a \cdot \vartheta = \textit{term-of } a$
⟨*proof*⟩

lemma *term-of-unit*: $a : \sigma$ *in* $A \Longrightarrow \textit{term-of } a = \eta\ a$
⟨*proof*⟩

lemma *map-term-of-subst*: $ds :_l \sigma s$ *in* $A \Longrightarrow [\textit{term-of } d \cdot \vartheta.\ d \leftarrow ds] = \textit{map term-of } ds$
⟨*proof*⟩

end

locale *sorted-algebra-const* = *sorted-algebra* +
const: *subsignature-algebra* C F +
const: *sorted-algebra-represented* C **for** C
begin

thm *const.Term-subsset*

end

Restricting range of a partial map.

definition *restrict-ran* (**infix** $\restriction^r$ *100*) **where**
$(A \restriction^r S)\ a \equiv \textit{if } A\ a \in \textit{Some } `\ S \textit{ then } A\ a \textit{ else None}$

lemma *ran-restrict-ran*: $\textit{ran}\ (A \restriction^r S) = \textit{ran}\ A \cap S$
⟨*proof*⟩

lemma *restrict-ran-le*: $A \restriction^r S \subseteq_m A$ ⟨*proof*⟩

lemma *hastype-in-restrict-ran*: $a : \sigma$ *in* $A \restriction^r S \longleftrightarrow a : \sigma$ *in* $A \wedge \sigma \in S$
⟨*proof*⟩

lemma *hastype-in-restrict-ranI*[*intro*!]: $a : \sigma$ *in* $A \Longrightarrow \sigma \in S \Longrightarrow a : \sigma$ *in* $A \restriction^r S$

⟨*proof*⟩

Sorts of Signature

definition *sorts-ssig F* ≡ ⋃{*set σs* ∪ {*τ*} |*σs τ f*. *f* : *σs* → *τ in F*}

lemma assumes *f* : *σs* → *τ in F*
shows *sorts-ssig-arg*: *σ* ∈ *set σs* ⟹ *σ* ∈ *sorts-ssig F*
and *sorts-ssig-ret*: *τ* ∈ *sorts-ssig F*
⟨*proof*⟩

lemma *nth-arg-in-sorts-ssig*:
assumes *f* : *σs* → *τ in F*
shows *n* < *length σs* ⟹ *σs*!*n* ∈ *sorts-ssig F*
⟨*proof*⟩

lemma *hastype-in-sorts-ssig-ret*:
assumes *f*: *f* : *σs* → *τ in F*
shows *a* : *τ in A* ↾*r sorts-ssig F* ⟷ *a* : *τ in A*
⟨*proof*⟩

lemma *hastype-in-sorts-ssig-args*:
assumes *f*: *f* : *σs* → *τ in F*
shows *as* :*l σs in A* ↾*r sorts-ssig F* ⟷ *as* :*l σs in A*
⟨*proof*⟩

lemma *sorted-algebra-sorts-ssig*:
sorted-algebra F (*A* ↾*r sorts-ssig F*) *I* ⟷ *sorted-algebra F A I*
⟨*proof*⟩

lemma *sorted-algebra-carrier-cong*:
assumes *A* ↾*r sorts-ssig F* = *B* ↾*r sorts-ssig F*
shows *sorted-algebra F A I* ⟷ *sorted-algebra F B I*
⟨*proof*⟩

lemma (**in** *sorted-algebra*) *sorted-algebra-carrier*:
assumes *A* ↾*r sorts-ssig F* = *B* ↾*r sorts-ssig F*
shows *sorted-algebra F B I*
⟨*proof*⟩

lemma (**in** *sort-preserving*) *sorted-image-restrict-ran*: (*f* ‘*s A*) ↾*r S* = *f* ‘*s A* ↾*r S*
⟨*proof*⟩

end
theory *Sorted-Relations*
imports *Sorted-Terms.Sorted-Sets Binary-Relations-More*
begin

declare *domIff*[*iff del*]

4 Sorted Relations

This part contains notions about binary relations over sorted sets.

4.1 Subject Reduction Property

locale *subject-reduction* =
  fixes A **and** *less-eq* (**infix** $\sqsubseteq$ *50*)
  assumes *subject-reduction*: $a \sqsubseteq b \Longrightarrow a : \sigma \ in \ A \Longrightarrow b : \sigma \ in \ A$

lemma *eq-subject-reduction*: *subject-reduction* A $(=)$
  ⟨*proof*⟩

lemma *Sup-subject-reduction*:
  assumes $\forall\, r \in R.$ *subject-reduction* A r
  shows *subject-reduction* A $(\bigsqcup R)$
  ⟨*proof*⟩

context *subject-reduction* **begin**

sublocale *relation-closed dom A*
  ⟨*proof*⟩

lemma *closed-in-dom*: $a \sqsubseteq b \Longrightarrow a \in dom\ A \Longrightarrow b \in dom\ A$
  ⟨*proof*⟩

lemma *list-all2-subject-reduction*: *list-all2* $(\sqsubseteq)$ as $bs \Longrightarrow as :_l \sigma s \ in \ A \Longrightarrow bs :_l \sigma s \ in \ A$
  ⟨*proof*⟩

lemma *relcompp-subject-reduction*:
  assumes *subject-reduction* A r
  shows *subject-reduction* A $((\sqsubseteq)\ OO\ r)$
  ⟨*proof*⟩

lemma *relpowp-subject-reduction*: *subject-reduction* A $((\sqsubseteq)^{\frown\frown}n)$
  ⟨*proof*⟩

interpretation *less-eq-symmetrize*⟨*proof*⟩

lemma *reflclp-subject-reduction*: *subject-reduction* A (*reflclp* $(\sqsubseteq)$)
  ⟨*proof*⟩

lemma *sympartp-subject-reduction*: *subject-reduction* A $(\sim)$
  ⟨*proof*⟩

lemma *equiv-subject-reduction*: *subject-reduction* A $(\simeq)$
  ⟨*proof*⟩

lemma *tranclp-subject-reduction*: *subject-reduction A* (*tranclp* (⊑))
⟨*proof*⟩

lemma *rtranclp-subject-reduction*: *subject-reduction A* (*rtranclp* (⊑))
⟨*proof*⟩

lemma *reflclp-on-subject-reduction*: *subject-reduction A* (*reflclp-on* (*dom A*) (⊑))
⟨*proof*⟩

lemma *tranclp-reflclp-on-subject-reduction*: *subject-reduction A* (*reflclp-on* (*dom A*) (*tranclp* (⊑)))
⟨*proof*⟩

end

lemma *subject-reduction-cong*:
assumes *r*: ⋀*a b*. *a* ∈ *dom A* ⟹ *r a b* ⟷ *r*′ *a b*
shows *subject-reduction A r* ⟷ *subject-reduction A r*′
⟨*proof*⟩

locale *sorted-relation* = *subject-reduction* + *dual*: *subject-reduction A* (⊑)⁻

lemma *Sup-sorted-relation*:
assumes ∀ *r* ∈ *R*. *sorted-relation A r*
shows *sorted-relation A* (⨆ *R*)
⟨*proof*⟩

lemma *sup-sorted-relation*:
assumes *sorted-relation A r sorted-relation A s*
shows *sorted-relation A* (*r* ⊔ *s*)
⟨*proof*⟩

context *sorted-relation* **begin**

lemma *relpowp-sorted-relation*: *sorted-relation A* ((⊑)^^*n*)
⟨*proof*⟩

lemma *related-imp-hastype-iff*: *a* ⊑ *b* ⟹ *a* : *σ in A* ⟷ *b* : *σ in A*
⟨*proof*⟩

lemma *related-imp-in-dom-iff*: **assumes** *ab*: *a* ⊑ *b* **shows** *a* ∈ *dom A* ⟷ *b* ∈ *dom A*
⟨*proof*⟩

lemma *list-all2-sorted-relation*: *list-all2* (⊑) *as bs* ⟹ *as* :$_l$ *σs in A* ⟷ *bs* :$_l$ *σs in A*
⟨*proof*⟩

lemma *dual-sorted-relation*: *sorted-relation A* (⊑)⁻ ⟨*proof*⟩

lemma *reflclp-sorted-relation*: *sorted-relation A* (*reflclp* (⊑))
⟨*proof*⟩

lemma *tranclp-sorted-relation*: *sorted-relation A* (*tranclp* (⊑))
⟨*proof*⟩

lemma *rtranclp-sorted-relation*: *sorted-relation A* (*rtranclp* (⊑))
⟨*proof*⟩

lemma *reflclp-on-sorted-relation*: *sorted-relation A* (*reflclp-on* (*dom A*) (⊑))
⟨*proof*⟩

lemma *tranclp-reflclp-on-sorted-relation*: *sorted-relation A* (*reflclp-on* (*dom A*) (*tranclp* (⊑)))
⟨*proof*⟩

lemma *symclp-sorted-relation*: *sorted-relation A* (*symclp* (⊑))
⟨*proof*⟩

end

lemma *sorted-relation-cong*:
assumes *r1*: $\bigwedge a\ b.\ a \in dom\ A \Longrightarrow r\ a\ b \longleftrightarrow r'\ a\ b$
assumes *r2*: $\bigwedge a\ b.\ b \in dom\ A \Longrightarrow r\ a\ b \longleftrightarrow r'\ a\ b$
shows *sorted-relation A r* $\longleftrightarrow$ *sorted-relation A r'*
⟨*proof*⟩

lemma *sorted-relation-iff*: *sorted-relation A r* $\longleftrightarrow$ $(\forall\ a\ b\ \sigma.\ r\ a\ b \longrightarrow a : \sigma\ in\ A \longleftrightarrow b : \sigma\ in\ A)$
⟨*proof*⟩

interpretation *eq*: *sorted-relation A* (=) **for** *A*
⟨*proof*⟩

context *subject-reduction* **begin**

interpretation *less-eq-symmetrize*⟨*proof*⟩

lemma *equiv-sorted-relation*: *sorted-relation A* (≃)
⟨*proof*⟩

end

end
theory *Sorted-Rules*
imports *Sorted-Terms.Sorted-Terms*
begin

5 Sorted Rules

This part defines datatypes for sorted rewrite rules and inference rules.

declare *Ball-Pair-conv*[*simp*]

5.1 Equation

An *equation* is a pair of terms.

type-synonym $('f,'v)$ *term-pair* $= ('f,'v)$ *term* $\times$ $('f,'v)$ *term*

abbreviation *Equation* :: $('f,'v)$ *term* $\Rightarrow$ $('f,'v)$ *term* $\Rightarrow$ - (- $\rightsquigarrow$ - [*51*,*51*]*24*)
  where $s \rightsquigarrow t \equiv (s,t)$

Equation $s \rightsquigarrow t$ represents that the two terms are in relation, where the relation is specified later.

syntax
  -ball-term-pair :: *pttrn* $\Rightarrow$ *pttrn* $\Rightarrow$ $('f,'v)$ *term-pair set* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
  ((*3*$\forall$ ('(- $\rightsquigarrow$ -')/ $\in$ -)./ -) [*0*, *0*, *0*, *10*] *10*)
  -bex-term-pair :: *pttrn* $\Rightarrow$ *pttrn* $\Rightarrow$ $('f,'v)$ *term-pair set* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
  ((*3*$\exists$ ('(- $\rightsquigarrow$ -')/ $\in$ -)./ -) [*0*, *0*, *0*, *10*] *10*)

translations
  $\forall (x \rightsquigarrow y) \in R.\ e \rightharpoonup \forall (x,y) \in R.\ e$
  $\exists (x \rightsquigarrow y) \in R.\ e \rightharpoonup \exists (x,y) \in R.\ e$

primrec *vars-term-pair* **where** *vars-term-pair* $(l \rightsquigarrow r) =$ *vars* $l \cup$ *vars* r

adhoc-overloading *vars* $\rightleftharpoons$ *vars-term-pair*

abbreviation *vars-term-pair-set* :: $('f,'v)$ *term-pair set* $\Rightarrow$ $'v$ *set* **where**
  vars-term-pair-set $C \equiv \bigcup$(*vars* ' C)

adhoc-overloading *vars* $\rightleftharpoons$ *vars-term-pair-set*

abbreviation *vars-term-pair-list* :: $('f,'v)$ *term-pair list* $\Rightarrow$ $'v$ *set* **where**
  vars-term-pair-list cs $\equiv$ *vars* (*set cs*)

adhoc-overloading *vars* $\rightleftharpoons$ *vars-term-pair-list*

primrec *map-vars-term-pair* **where**
  map-vars-term-pair f $(l \rightsquigarrow r) =$ (*map-vars f l* $\rightsquigarrow$ *map-vars f r*)

adhoc-overloading *map-vars* $\rightleftharpoons$ *map-vars-term-pair*

adhoc-overloading *map-vars* $\rightleftharpoons$ *map* $\circ$ *map-vars-term-pair*

5.2 Axioms – Rewrite Rules

An axiom is an equation whose variables are considered universally quantified. So the following datatype additionally specifies the sorted set of variables which it can contain. An axiom is also seen as a rewrite rule.

datatype (*dead 'f*, *dead 'v*, *dead 's*) *axiom* =
Axiom (*vars*: *'v* ⇀ *'s*) (*lhs*: (*'f*,*'v*) *term*) (*rhs*: (*'f*,*'v*) *term*)
(-. - ⇝ - [*100*,*51*,*51*]*21*)

hide-const (**open**) *axiom.vars axiom.lhs axiom.rhs*

declare *axiom.split*[*split*] *axiom.split-asm*[*split*]

abbreviation *ball-axiom* **where** *ball-axiom R P* ≡ ∀ *X l r*. (*X*. *l* ⇝ *r*) ∈ *R* ⟶ *P X l r*
abbreviation *bex-axiom* **where** *bex-axiom R P* ≡ ∃ *X l r*. (*X*. *l* ⇝ *r*) ∈ *R* ∧ *P X l r*

syntax *-ball-axiom* :: *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ (*'f*,*'v*,*'s*) *axiom set* ⇒ *bool* ⇒ *bool*
((*3*∀ (*'*(-. - ⇝ -*'*)/ ∈ -)./ -) [*0*, *0*, *0*, *0*, *10*] *10*)
-bex-axiom :: *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ (*'f*,*'v*,*'s*) *axiom set* ⇒ *bool* ⇒ *bool*
((*3*∃ (*'*(-. - ⇝ -*'*)/ ∈ -)./ -) [*0*, *0*, *0*, *0*, *10*] *10*)

translations
∀ (*X*. *l* ⇝ *r*) ∈ *R*. *e* ⇌ *CONST ball-axiom R* (λ*X l r*. *e*)
∃ (*X*. *l* ⇝ *r*) ∈ *R*. *e* ⇌ *CONST bex-axiom R* (λ*X l r*. *e*)

5.3 Inference Rules – Conditional Rewrite Rules

An inference rule is an axiom extended with a list of premises. An inference rule is also seen as a conditional rewrite rule.

datatype (*dead 'f*, *dead 'v*, *dead 's*) *rule* =
Rule (*vars*: *'v* ⇀ *'s*) (*lhs*: (*'f*,*'v*) *term*) (*rhs*: (*'f*,*'v*) *term*) (*prems*: ((*'f*,*'v*) *term-pair*) *list*)
(-. - ⇝ - ⇐ - [*100*,*51*,*51*,*51*]*21*)

hide-const (**open**) *rule.vars rule.lhs rule.rhs rule.prems*

declare *rule.split*[*split*] *rule.split-asm*[*split*]

lemma *rule-split-all*: (∀ *ϱ*. *P ϱ*) ⟷ (∀ *V l r cs V*. *P* (*V*. *l* ⇝ *r* ⇐ *cs*))
⟨*proof*⟩

abbreviation *ball-rule* **where** *ball-rule R P* ≡ ∀ *X l r cs*. (*X*. *l* ⇝ *r* ⇐ *cs*) ∈ *R* ⟶ *P X l r cs*
abbreviation *bex-rule* **where** *bex-rule R P* ≡ ∃ *X l r cs*. (*X*. *l* ⇝ *r* ⇐ *cs*) ∈ *R* ∧ *P X l r cs*

syntax
-ball-rule :: *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ ('*f*,'*v*,'*s*) *rule set* ⇒ *bool* ⇒ *bool*
((*3*∀('(-. - ⇝ - ⇐ -')/ ∈ -)./ -) [*0*, *0*, *0*, *0*, *0*, *10*] *10*)
-bex-rule :: *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ ('*f*,'*v*,'*s*) *rule set* ⇒ *bool* ⇒ *bool*
((*3*∃('(-. - ⇝ - ⇐ -')/ ∈ -)./ -) [*0*, *0*, *0*, *0*, *0*, *10*] *10*)

translations
∀(*X*. *l* ⇝ *r* ⇐ *cs*) ∈ *R*. *e* ⇌ *CONST ball-rule R* (λ*X l r cs*. *e*)
∃(*X*. *l* ⇝ *r* ⇐ *cs*) ∈ *R*. *e* ⇌ *CONST bex-rule R* (λ*X l r cs*. *e*)

primrec *vars-rule* **where**
vars-rule (*V*. *l* ⇝ *r* ⇐ *cs*) = *vars l* ∪ *vars r* ∪ *vars cs*

adhoc-overloading *vars* ⇌ *vars-rule*

primrec *map-vars-rule* **where**
map-vars-rule f (*V*. *l* ⇝ *r* ⇐ *cs*) = ((*f* ‘*s* *V*). *map-vars f l* ⇝ *map-vars f r* ⇐ *map-vars f cs*)

adhoc-overloading *map-vars* ⇌ *map-vars-rule*

lemma *vars-rule-lhs*: *vars* (*rule.lhs* ϱ) ⊆ *vars* ϱ
and *vars-rule-rhs*: *vars* (*rule.rhs* ϱ) ⊆ *vars* ϱ
⟨*proof*⟩

lemma
assumes (*s* ⇝ *t*) ∈ *set* (*rule.prems* ϱ)
shows *vars-rule-prems-left*: *vars s* ⊆ *vars* ϱ
and *vars-rule-prems-right*: *vars t* ⊆ *vars* ϱ
⟨*proof*⟩

One can see axioms as unconditional inference rules.

definition *unconditional* **where** *unconditional* ≡ λ(*V*. *l* ⇝ *r*) ⇒ *V*. *l* ⇝ *r* ⇐ []

lemma *unconditional*[*simp*]: *unconditional* (*V*. *l* ⇝ *r*) = (*V*. *l* ⇝ *r* ⇐ []) ⟨*proof*⟩

lemma *in-unconditional*[*simp*]: (*V*. *l* ⇝ *r* ⇐ *cs*) ∈ *unconditional* ‘ *R* ⟷ (*V*. *l* ⇝ *r*) ∈ *R* ∧ *cs* = []
⟨*proof*⟩

5.4 Sortedness of Rules

When rules are well-typed, then the derivation is a sorted relation. Generally, we allow relating terms of different sorts.

definition *sorted-rule F* ≡ λ(*V*. *l* ⇝ *r* ⇐ *cs*) ⇒
l ∈ *dom* $\mathcal{T}$(*F*,*V*) ∧ *r* ∈ *dom* $\mathcal{T}$(*F*,*V*) ∧
(∀(*s* ⇝ *t*) ∈ *set cs*. *s* ∈ *dom* $\mathcal{T}$(*F*,*V*) ∧ *t* ∈ *dom* $\mathcal{T}$(*F*,*V*))

lemma *sorted-rule*:
sorted-rule F (*V*. *l* ⇝ *r* ⇐ *cs*) ⟷ *l* ∈ *dom* $\mathcal{T}(F,V)$ ∧ *r* ∈ *dom* $\mathcal{T}(F,V)$ ∧
(∀ (*s* ⇝ *t*) ∈ *set cs*. *s* ∈ *dom* $\mathcal{T}(F,V)$ ∧ *t* ∈ *dom* $\mathcal{T}(F,V)$)
⟨*proof*⟩

lemma *sorted-ruleI*:
assumes ϱ = (*V*. *l* ⇝ *r* ⇐ *cs*)
and *l* : σ *in* $\mathcal{T}(F,V)$ **and** *r* : σ′ *in* $\mathcal{T}(F,V)$
and ⋀*s t*. (*s* ⇝ *t*) ∈ *set cs* ⟹ ∃ τ τ′. *s* : τ *in* $\mathcal{T}(F,V)$ ∧ *t* : τ′ *in* $\mathcal{T}(F,V)$
shows *sorted-rule F* ϱ
⟨*proof*⟩

lemma assumes *sorted-rule F* (*V*. *l* ⇝ *r* ⇐ *cs*)
shows *sorted-rule-cond-domD*:
⋀*s t*. (*s* ⇝ *t*) ∈ *set cs* ⟹ *s* ∈ *dom* $\mathcal{T}(F,V)$ ∧ *t* ∈ *dom* $\mathcal{T}(F,V)$
and *sorted-rule-domD*: *l* ∈ *dom* $\mathcal{T}(F,V)$ *r* ∈ *dom* $\mathcal{T}(F,V)$
⟨*proof*⟩

Often we consider relating a term to a term of the same sort. We say a rule is *sort safe* if the conclusion relates terms of the same sort, when all conditions do so.

definition *sort-safe* **where** *sort-safe F* ϱ ≡
sorted-rule F ϱ ∧ (
case ϱ *of* (*V*. *l* ⇝ *r* ⇐ *cs*) ⇒
((∀ (*s* ⇝ *t*) ∈ *set cs*. ∃ τ. *s* : τ *in* $\mathcal{T}(F,V)$ ∧ *t* : τ *in* $\mathcal{T}(F,V)$) ⟶
(∃ σ. *l* : σ *in* $\mathcal{T}(F,V)$ ∧ *r* : σ *in* $\mathcal{T}(F,V)$))))

lemma *sort-safeI*:
assumes ϱ = (*V*. *l* ⇝ *r* ⇐ *cs*)
and *l* : σ *in* $\mathcal{T}(F,V)$ **and** *r* : σ′ *in* $\mathcal{T}(F,V)$
and ⋀*s t*. (*s* ⇝ *t*) ∈ *set cs* ⟹ ∃ τ τ′. *s* : τ *in* $\mathcal{T}(F,V)$ ∧ *t* : τ′ *in* $\mathcal{T}(F,V)$
and ∀ (*s* ⇝ *t*) ∈ *set cs*. ∃ τ. *s* : τ *in* $\mathcal{T}(F,V)$ ∧ *t* : τ *in* $\mathcal{T}(F,V)$ ⟹ ∃ σ. *l* : σ *in* $\mathcal{T}(F,V)$ ∧ *r* : σ *in* $\mathcal{T}(F,V)$
shows *sort-safe F* ϱ
⟨*proof*⟩

lemma *sort-safe-imp-sorted*: *sort-safe F* ϱ ⟹ *sorted-rule F* ϱ
⟨*proof*⟩

lemma
assumes *sort-safe F* (*V*. *l* ⇝ *r* ⇐ *cs*)
shows *sort-safeD*:
∀ (*s* ⇝ *t*) ∈ *set cs*. ∃ τ. *s* : τ *in* $\mathcal{T}(F,V)$ ∧ *t* : τ *in* $\mathcal{T}(F,V)$ ⟹ ∃ σ. *l* : σ *in* $\mathcal{T}(F,V)$ ∧ *r* : σ *in* $\mathcal{T}(F,V)$
⟨*proof*⟩

definition *sort-safe-rules F R* ≡ ∀ ϱ ∈ *R*. *sort-safe F* ϱ

lemma *sort-safe-rules-Un*: *sort-safe-rules F* (*R* ∪ *S*) ⟷ *sort-safe-rules F R* ∧

sort-safe-rules F S
⟨*proof*⟩

lemmas *sort-safe-rules-UnI*[*intro*!] = *sort-safe-rules-Un*[*THEN iffD2*, *unfolded conj-imp-eq-imp-imp*]

abbreviation *sorted-rules F R* ≡ ∀ ϱ ∈ *R*. *sorted-rule F* ϱ

lemma *sort-safe-rules-imp-sorted*: *sort-safe-rules F R* ⟹ *sorted-rules F R*
⟨*proof*⟩

end

6 Models

theory *Models*
imports *Sorted-Rules*
begin

Here we formalize *relational models* [5] of sets of rules. Relational models assert two objects are in specified relation where standard models assert equality.

6.1 Satisfaction of Equations

Given an interpretation of function symbols and a relation, a variable assignment *satisfies* an equation if the evaluation of the terms are in relation:

definition *satisfies* ((*2*(-:/-;/- ⊨)/ (- ⇝/ -)) [*51*,*51*,*3*,*51*,*51*]*4*) **where**
I:(⊑);α ⊨ *s* ⇝ *t* ≡ *I*⟦*s*⟧α ⊑ *I*⟦*t*⟧α **for** *le* (**infix** ⊑ *50*)

Particularly important case is where the relation is the equality.

abbreviation *eq-satisfies* ((*2*(-;/- ⊨)/ (- =/ -)) [*51*,*3*,*51*,*51*]*4*) **where**
I;α ⊨ *s* = *t* ≡ *I*:(=);α ⊨ *s* ⇝ *t*

lemmas *satisfiesI* = *satisfies-def*[*unfolded atomize-eq*, *THEN iffD2*]
lemmas *satisfiesD* = *satisfies-def*[*unfolded atomize-eq*, *THEN iffD1*]
lemmas *satisfiesE* = *satisfiesD*[*elim-format*]

lemma *satisfies-mono*:
assumes *r* ≤ *r*′ **shows** *satisfies I r* ≤ *satisfies I r*′
⟨*proof*⟩

lemma *dual-satisfies*[*simp*]: (*satisfies I r* α)⁻ = *satisfies I r*⁻ α
⟨*proof*⟩

lemma *satisfies-subst*: (*I*:*r*;α ⊨ *s*·ϑ ⇝ *t*·ϑ) ⟷ (*I*:*r*;*I*⟦ϑ⟧$_s$ α ⊨ *s* ⇝ *t*)
⟨*proof*⟩

lemma *Term-Var-satisfies*[*simp*]:
(*Fun*:(⊑);*Var* ⊨ $s \rightsquigarrow t$) ⟷ $s \sqsubseteq t$ **for** *less-eq* (**infix** ⊑ *50*)
⟨*proof*⟩

lemma *satisfies-same-vars*:
assumes ∀ $x \in$ *vars* $s \cup$ *vars* t. $\alpha\ x = \beta\ x$
shows (*I*:*r*;α ⊨ $s \rightsquigarrow t$) ⟷ (*I*:*r*;β ⊨ $s \rightsquigarrow t$)
⟨*proof*⟩

interpretation *eq-satisfies*: *equivalence UNIV eq-satisfies I* α
⟨*proof*⟩

lemma (**in** *sorted-algebra*) *eq-satisfies-has-same-type*:
assumes α: $\alpha :_s V \to A$ **and** *st*: *I*;α ⊨ $s = t$
and *s*: $s : \sigma$ *in* $\mathcal{T}(F,V)$ **and** *t*: $t : \tau$ *in* $\mathcal{T}(F,V)$
shows $\sigma = \tau$
⟨*proof*⟩

6.2 Validity of Axioms

An axiom is *valid* if any assignment satisfies it.

definition *valid* ((*2*(-:/-:/- ⊨)/ -./ (- ⇝/ -)) [*51*,*51*,*51*,*100*,*51*,*51*]*4*) **where**
A:*I*:(⊑) ⊨ *V*. $s \rightsquigarrow t \equiv \forall \alpha :_s V \to A$. (*I*:(⊑);$\alpha$ ⊨ $s \rightsquigarrow t$) **for** *le* (**infix** ⊑ *50*)

abbreviation *valid-eq* ((*2*(-:/- ⊨)/ -./ (- =/ -)) [*51*,*51*,*100*,*51*,*51*]*4*) **where**
A:*I* ⊨ *V*. $s = t \equiv$ *A*:*I*:(=) ⊨ *V*. $s \rightsquigarrow t$

lemmas *validI*[*intro?*] = *valid-def*[*unfolded atomize-eq*, *THEN iffD2*, *rule-format*]
lemmas *validD* = *valid-def*[*unfolded atomize-eq*, *THEN iffD1*, *rule-format*, *simp*]
lemmas *validE* = *valid-def*[*unfolded atomize-eq*, *THEN iffD1*, *elim-format*, *rule-format*]

interpretation *valid-eq*: *equivalence UNIV valid-eq A I V*
⟨*proof*⟩

lemma *valid-mono*:
assumes $r \leq r'$ **shows** *valid A I r* ≤ *valid A I r'*
⟨*proof*⟩

lemma *dual-valid*[*simp*]: (*valid A I r V*)⁻ = *valid A I r*⁻ *V*
⟨*proof*⟩

lemma (**in** *sorted-algebra*) *valid-subst-closed*:
assumes *st*: *A*:*I*:*r* ⊨ *V*. $s \rightsquigarrow t$ **and** ϑ: $\vartheta :_s V \to \mathcal{T}(F,W)$
shows *A*:*I*:*r* ⊨ *W*. $s{\cdot}\vartheta \rightsquigarrow t{\cdot}\vartheta$
⟨*proof*⟩

Validity in the term algebra with respect to a relation implies the relation (and vice versa if the relation is closed under substitution).

lemma *Term-valid-relates*: $\mathcal{T}(F,V)$:*Fun*:*r* ⊨ *V*. $s \rightsquigarrow t \Longrightarrow r\ s\ t$

⟨*proof*⟩

6.2.1 Models

An algebra coupled with a binary relation *models* an inference rule if all assignments that satisfy the premise satisfy the conclusion.

definition *models-rule* ((*2*-:/-:/- ⊨/ -./ (*2*- ⇝/ - ⇐/ -)) [*51*,*51*,*51*,*100*,*51*,*51*,*51*]*4*)
where
  $A{:}I{:}e \models V.\ l \rightsquigarrow r \Leftarrow cs \equiv$
  $\forall \alpha :_s V \to A.\ (\forall (s \rightsquigarrow t) \in set\ cs.\ (I{:}e{;}\alpha \models s \rightsquigarrow t)) \longrightarrow (I{:}e{;}\alpha \models l \rightsquigarrow r)$

abbreviation *models-rule-eq* ((*2*-:/- ⊨/ -./ (*2*- =/ - ⇐/ -)) [*51*,*51*,*100*,*51*,*51*,*51*]*4*)
where
  $A{:}I \models V.\ l = r \Leftarrow cs \equiv A{:}I{:}(=) \models V.\ l \rightsquigarrow r \Leftarrow cs$

lemma *models-ruleI*:
  assumes $\bigwedge \alpha.\ \alpha :_s V \to A \Longrightarrow \forall (s \rightsquigarrow t) \in set\ cs.\ (I{:}e{;}\alpha \models s \rightsquigarrow t) \Longrightarrow I{:}e{;}\alpha \models l \rightsquigarrow r$
  shows $A{:}I{:}e \models V.\ l \rightsquigarrow r \Leftarrow cs$
  ⟨*proof*⟩

lemma *models-ruleD*:
  $A{:}I{:}e \models V.\ l \rightsquigarrow r \Leftarrow cs \Longrightarrow$
   $\alpha :_s V \to A \Longrightarrow (\bigwedge s\ t.\ (s \rightsquigarrow t) \in set\ cs \Longrightarrow (I{:}e{;}\alpha \models s \rightsquigarrow t)) \Longrightarrow I{:}e{;}\alpha \models l \rightsquigarrow r$
  ⟨*proof*⟩

An algebra coupled with a binary relation *models* a set of rules, if it models all the rules.

definition *models* ((*2*-:/-:/- ⊨/ -) [*51*,*51*,*51*,*51*]*4*) **where**
  $A{:}I{:}e \models R \equiv \forall (V.\ l \rightsquigarrow r \Leftarrow cs) \in R.\ (A{:}I{:}e \models V.\ l \rightsquigarrow r \Leftarrow cs)$

abbreviation *models-eq* ((*2*-:/- ⊨/ -) [*51*,*51*,*51*]*4*) **where**
  $A{:}I \models R \equiv A{:}I{:}(=) \models R$

lemmas *modelsI* = *models-def*[*unfolded atomize-eq*, *THEN iffD2*, *rule-format*]
lemmas *modelsD* = *models-def*[*unfolded atomize-eq*, *THEN iffD1*, *rule-format*]
lemmas *modelsE* = *models-def*[*unfolded atomize-eq*, *THEN iffD1*, *elim-format*, *rule-format*]

lemma *models-empty*[*simp*]: $A{:}I{:}e \models \{\}$ ⟨*proof*⟩

lemma *models-Un*[*simp*]: $(A{:}I{:}e \models R \cup R') \longleftrightarrow (A{:}I{:}e \models R) \wedge (A{:}I{:}e \models R')$
  ⟨*proof*⟩

lemma *models-cmono*: $R \subseteq R' \Longrightarrow A{:}I{:}e \models R' \Longrightarrow A{:}I{:}e \models R$ ⟨*proof*⟩

lemma *models-axiom*[*simp*]: $(A{:}I{:}e \models V.\ l \rightsquigarrow r \Leftarrow []) \longleftrightarrow (A{:}I{:}e \models V.\ l \rightsquigarrow r)$

⟨*proof*⟩

lemma *models-unconditional-iff*:
(*A*:*I*:(⊑) ⊨ *unconditional* ‘ *R*) ⟷ (∀(*X*. *l* ⇝ *r*) ∈ *R*. ∀α :$_s$ *X* → *A*. *I*⟦*l*⟧α ⊑ *I*⟦*r*⟧α)
for *rel* (**infix** ⊑ *50*)
⟨*proof*⟩

lemmas *models-unconditionalI* =
models-unconditional-iff[*THEN iffD2*, *rule-format*]

lemmas *models-unconditionalD* =
models-unconditional-iff[*THEN iffD1*, *rule-format*]

lemma *models-unconditional-mono*:
assumes *rel* ≤ *rel*′
shows (*A*:*I*:*rel* ⊨ *unconditional* ‘ *R*) ⟹ (*A*:*I*:*rel*′ ⊨ *unconditional* ‘ *R*)
⟨*proof*⟩

end

7 Ordered Algebras

We formalize algebras associated with a binary relation, especially those in which properties of the binary relation is preserved in terms. Order properties are assumed only over well-typed elements (*dom A*), as we do not want to specify how ill-typed elements are related.

locale *reflexive-algebra* = *sorted-algebra* + *reflexive dom A*
begin

lemma *satisfies-reflexive*:
assumes α: α :$_s$ *V* → *A*
shows *reflexive* (*dom* $\mathcal{T}(F,V)$) (*satisfies I* (⊑) α)
⟨*proof*⟩

sublocale *valid*: *reflexive dom* $\mathcal{T}(F,V)$ *valid A I* (⊑) *V*
⟨*proof*⟩

sublocale *lists*: *reflexive* ‹*lists* (*dom A*)› ‹*list-all2* (⊑)› ⟨*proof*⟩

end

locale *attractive-algebra* = *sorted-algebra* + *attractive dom A*
begin

lemma *satisfies-attractive*:
assumes α: α :$_s$ *V* → *A*
shows *attractive* (*dom* $\mathcal{T}(F,V)$) (*satisfies I* (⊑) α)

⟨*proof*⟩

sublocale *valid*: *attractive dom* $\mathcal{T}(F,V)$ *valid A I* $(\sqsubseteq)$ *V*
⟨*proof*⟩

sublocale *lists*: *attractive* ‹*lists* (*dom A*)› ‹*list-all2* $(\sqsubseteq)$› ⟨*proof*⟩

end

locale *transitive-algebra* = *sorted-algebra* + *transitive dom A*
begin

sublocale *attractive-algebra*⟨*proof*⟩

lemma *satisfies-transitive*:
assumes α: $\alpha :_s V \to A$
shows *transitive* (*dom* $\mathcal{T}(F,V)$) (*satisfies I* $(\sqsubseteq)$ α)
⟨*proof*⟩

sublocale *valid*: *transitive* ‹*dom* $\mathcal{T}(F,V)$› ‹*valid A I* $(\sqsubseteq)$ *V*› **for** *V*
⟨*proof*⟩

sublocale *lists*: *transitive* ‹*lists* (*dom A*)› ‹*list-all2* $(\sqsubseteq)$› ⟨*proof*⟩

end

locale *quasi-ordered-algebra* = *sorted-algebra* + *quasi-ordered-set dom A*
begin

sublocale *transitive-algebra* + *reflexive-algebra*⟨*proof*⟩

lemma *satisfies-quasi-order*: $\alpha :_s V \to A \Longrightarrow$ *quasi-ordered-set* (*dom* $\mathcal{T}(F,V)$)
(*satisfies I* $(\sqsubseteq)$ α)
⟨*proof*⟩

sublocale *valid*: *quasi-ordered-set dom* $\mathcal{T}(F,V)$ *valid A I* $(\sqsubseteq)$ *V*⟨*proof*⟩

sublocale *lists*: *quasi-ordered-set* ‹*lists* (*dom A*)› ‹*list-all2* $(\sqsubseteq)$› ⟨*proof*⟩

end

We do not consider antisymmetry alone here: it is not preserved in the term algebra, because different terms may have the same evaluation. So we just define ordered algebra and derive that terms are quasi-ordered.

locale *ordered-algebra* = *sorted-algebra* + *partially-ordered-set dom A*
begin

sublocale *quasi-ordered-algebra*⟨*proof*⟩

end

Irreflexivity is carried over to terms, if the algebra is inhabited.

context *inhabited* **begin**

lemma *Term-inhabited*: *inhabited* $\mathcal{T}(F,A)$
⟨*proof*⟩

end

locale *irreflexive-algebra* = *sorted-algebra* + *irreflexive dom A* + *inhabited*
begin

lemma *satisfies-irreflexive*:
assumes α: $\alpha :_s V \to A$
shows *irreflexive* (*dom* $\mathcal{T}(F,V)$) (*satisfies I* ($\sqsubset$) α)
⟨*proof*⟩

sublocale *valid*: *irreflexive dom* $\mathcal{T}(F,V)$ *valid A I* ($\sqsubset$) *V*
⟨*proof*⟩

end

Hence, strict order is carried over to terms.

locale *strict-ordered-algebra* = *sorted-algebra* + *strict-ordered-set dom A* + *inhabited*
begin

sublocale *transitive-algebra F A I* ($\sqsubset$) + *irreflexive-algebra*⟨*proof*⟩

sublocale *valid*: *strict-ordered-set dom* $\mathcal{T}(F,V)$ *valid A I* ($\sqsubset$) *V*⟨*proof*⟩

end

end

8 Monotone Algebras

For monotonicity, we only consider elements of the same type. It is possible to consider elements of different types and overloaded function symbols, but then one cannot have the nice correspondence of monotonicity and congruence under quasi-order.

locale *monotone* = **fixes** *F A I* **and** *less-eq* (**infix** $\sqsubseteq$ *50*)
assumes *comp-arg*:
$a \sqsubseteq b \Longrightarrow$
$f : \pi s \ @ \ \sigma \ \# \ \varrho s \to \tau$ *in* $F \Longrightarrow$
$ls :_l \pi s$ *in* $A \Longrightarrow a : \sigma$ *in* $A \Longrightarrow b : \sigma$ *in* $A \Longrightarrow rs :_l \varrho s$ *in* $A \Longrightarrow$

$I\ f\ (ls\ @\ a\ \#\ rs) \sqsubseteq I\ f\ (ls\ @\ b\ \#\ rs)$

lemma *eq-monotone*: *monotone F A I* (=)
⟨*proof*⟩

locale *monotone-algebra* = *sorted-algebra* + *monotone*
begin

The compatibility of the interpretation of function symbols carries over to contexts.

lemma *ctxt-closed*:
assumes *ab*: $a \sqsubseteq b$ **and** *C*: $C : \sigma \to \tau$ *in aContext F A*
and *a*: $a : \sigma$ *in A* **and** *b*: $b : \sigma$ *in A*
shows $I\langle C;a\rangle \sqsubseteq I\langle C;b\rangle$
⟨*proof*⟩

lemma *ctxtI*:
assumes *ab*: $a \sqsubseteq b$ **and** *C*: $C : \sigma \to \tau$ *in aContext F A*
and *a*: $a : \sigma$ *in A* **and** *b*: $b : \sigma$ *in A*
and *s*: $s = I\langle C;a\rangle$ **and** *t*: $t = I\langle C;b\rangle$
shows $s \sqsubseteq t$
⟨*proof*⟩

interpretation *less-eq-dualize*⟨*proof*⟩

interpretation *dual*: *monotone-algebra F A I* (⊒)
⟨*proof*⟩

lemmas *dual-monotone* = *dual.monotone-axioms*

sublocale *symclp*: *monotone F A I symclp* (⊑)
⟨*proof*⟩

sublocale *reflclp*: *monotone F A I reflclp* (⊑)
⟨*proof*⟩

sublocale *reflclp-on*: *monotone F A I reflclp-on* (*dom A*) (⊑)
⟨*proof*⟩

lemma *monotone-subalgebra*:
assumes *sub*: $A' \subseteq_m A$ **and** *alg*: *sorted-algebra F A' I*
shows *monotone-algebra F A' I* (⊑)
⟨*proof*⟩

context fixes α *V* **assumes** α: $\alpha :_s V \to A$
begin

interpretation *satisfies*: *monotone-algebra F* $\mathcal{T}(F,V)$ *Fun satisfies I* (⊑) α
⟨*proof*⟩

lemmas *satisfies-ctxt-closed* = *satisfies.ctxt-closed*

lemmas *satisfies-monotone* = *satisfies.monotone-axioms*

end

interpretation *valid*: *monotone-algebra F* $\mathcal{T}(F,V)$ *Fun valid A I* $(\sqsubseteq)$ *V*
⟨*proof*⟩

sublocale *valid*: *monotone F* $\mathcal{T}(F,V)$ *Fun valid A I* $(\sqsubseteq)$ *V*⟨*proof*⟩

lemmas *valid-ctxt-closed* = *valid.ctxt-closed*

lemma *dual-monotone-algebra*: *monotone-algebra F A I* $(\sqsubseteq)^-$
⟨*proof*⟩

end

lemma *monotone-algebra-cong*:
assumes *r*: $\bigwedge a\ b.\ a \in dom\ A \Longrightarrow b \in dom\ A \Longrightarrow r\ a\ b \longleftrightarrow r'\ a\ b$
shows *monotone-algebra F A I r* $\longleftrightarrow$ *monotone-algebra F A I r′*
⟨*proof*⟩

Monotonicity can be derived from closure under contexts.

lemma (**in** *sorted-algebra*) *monotone-iff-ctxt-closed*:
fixes *less-eq* (**infix** $\sqsubseteq$ *50*)
shows *monotone F A I* $(\sqsubseteq) \longleftrightarrow (\forall\ C\ \sigma\ \tau\ a\ b.$
$C : \sigma \to \tau$ *in aContext F A* $\longrightarrow a : \sigma$ *in A* $\longrightarrow b : \sigma$ *in A* $\longrightarrow a \sqsubseteq b \longrightarrow I\langle C;a\rangle \sqsubseteq I\langle C;b\rangle)$
(**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

lemma (**in** *sorted-algebra*) *Sup-monotone*:
assumes $\forall\ r \in R.$ *monotone F A I r*
shows *monotone F A I* $(\bigsqcup R)$
⟨*proof*⟩

lemma (**in** *sorted-algebra*) *sup-monotone*:
assumes *monotone F A I r monotone F A I s*
shows *monotone F A I* $(r \sqcup s)$
⟨*proof*⟩

Monotonicity is preserved by the transitive closure, if the relation satisfies the subject reduction property.

locale *monotone-algebra-subject-reduction* = *sorted-algebra* + *monotone* + *subject-reduction*
begin

sublocale *monotone-algebra*⟨*proof*⟩

lemma *relcompp-monotone-algebra*:
  assumes *monotone-algebra F A I r*
  shows *monotone F A I* ((⊑) *OO r*)
⟨*proof*⟩

interpretation *relpowp*: *monotone-algebra-subject-reduction F A I* (⊑)^^*n*
  ⟨*proof*⟩

sublocale *relpowp*: *monotone-algebra F A I* (⊑)^^*n*
  + *relpowp*: *subject-reduction A* (⊑)^^*n*⟨*proof*⟩

interpretation *reflclp*: *monotone-algebra-subject-reduction F A I reflclp* (⊑)
  ⟨*proof*⟩

sublocale *reflclp*: *monotone-algebra F A I reflclp* (⊑)
  + *reflclp*: *subject-reduction A reflclp* (⊑)⟨*proof*⟩

interpretation *tranclp*: *monotone-algebra-subject-reduction F A I tranclp* (⊑)
  ⟨*proof*⟩

sublocale *tranclp*: *monotone-algebra F A I tranclp* (⊑)
  + *tranclp*: *subject-reduction A tranclp* (⊑) ⟨*proof*⟩

interpretation *rtranclp*: *monotone-algebra-subject-reduction F A I* (*rtranclp* (⊑))
  ⟨*proof*⟩

sublocale *rtranclp*: *monotone-algebra F A I* (*rtranclp* (⊑))
  + *rtranclp*: *subject-reduction A rtranclp* (⊑)⟨*proof*⟩

interpretation *reflclp-on*: *monotone-algebra-subject-reduction F A I reflclp-on* (*dom A*) (⊑)
  ⟨*proof*⟩

sublocale *reflclp-on*: *monotone-algebra F A I reflclp-on* (*dom A*) (⊑)
  + *reflclp-on*: *subject-reduction A reflclp-on* (*dom A*) (⊑)⟨*proof*⟩

interpretation *rtranclp-on*: *monotone-algebra-subject-reduction F A I* (*rtranclp-on* (*dom A*) (⊑))⟨*proof*⟩

sublocale *rtranclp-on*: *monotone-algebra F A I* (*rtranclp-on* (*dom A*) (⊑))
  + *rtranclp-on*: *subject-reduction A rtranclp-on* (*dom A*) (⊑)⟨*proof*⟩

end

lemma (**in** *sorted-algebra-represented*) *monotone-iff-eval-ctxt*:
  fixes *less-eq* (**infix** ⊑ *50*)
  shows *monotone-algebra F A I* (⊑) ⟷ (∀ *C σ τ a b α*.

$C : \sigma \to \tau$ *in* $\mathcal{C}(F,\emptyset::unit \rightharpoonup \text{-}) \longrightarrow a : \sigma$ *in* $A \longrightarrow b : \sigma$ *in* $A \longrightarrow a \sqsubseteq b \longrightarrow$
$I\langle I[\![C]\!]_c\ \alpha;\ a\rangle \sqsubseteq I\langle I[\![C]\!]_c\ \alpha;\ b\rangle)$
(**is** $?l \longleftrightarrow ?r$)
$\langle proof \rangle$

8.1 Ordered Monotone Algebras

locale *reflexive-monotone-algebra* = *reflexive-algebra* + *monotone*
begin

sublocale *monotone-algebra*$\langle proof \rangle$

interpretation *valid*: *reflexive-monotone-algebra* $F\ \mathcal{T}(F,V)$ *Fun valid* $A\ I\ (\sqsubseteq)$
$V\langle proof \rangle$

lemmas *valid-reflexive-monotone-algebra* = *valid.reflexive-monotone-algebra-axioms*

end

locale *transitive-monotone-algebra* = *transitive-algebra* + *monotone*
begin

sublocale *monotone-algebra*$\langle proof \rangle$

interpretation *valid*: *transitive-monotone-algebra* $F\ \mathcal{T}(F,V)$ *Fun valid* $A\ I\ (\sqsubseteq)$
$V\langle proof \rangle$

lemmas *valid-transitive-monotone-algebra* = *valid.transitive-monotone-algebra-axioms*

end

locale *quasi-ordered-monotone-algebra* = *quasi-ordered-algebra* + *monotone*
begin

sublocale *monotone-algebra* + *reflexive-monotone-algebra* + *transitive-monotone-algebra*$\langle proof \rangle$

interpretation *valid*: *quasi-ordered-monotone-algebra* $F\ \mathcal{T}(F,V)$ *Fun valid* $A\ I$
$(\sqsubseteq)\ V\langle proof \rangle$

lemmas *valid-quasi-ordered-monotone-algebra* = *valid.quasi-ordered-monotone-algebra-axioms*

interpretation *less-eq-dualize*$\langle proof \rangle$

lemma *dual-quasi-ordered-monotone-algebra*:
quasi-ordered-monotone-algebra $F\ A\ I\ (\sqsupseteq)$
$\langle proof \rangle$

end

locale *strict-ordered-monotone-algebra* = *strict-ordered-algebra* + *monotone F A I* $(\sqsubset)$
begin

sublocale *transitive-monotone-algebra F A I* $(\sqsubset)\langle proof\rangle$

end

end

9 Sorted Rewrite Relations

theory *Sorted-Rewrite-Relations*
imports *Monotone-Algebras*
begin

named-theorems *rewriting-simps*

declare *relpowp-Suc-I2*[*trans*]

An ARS is just a set equipped with a binary relation. We introduce a locale to allow for convenient reasoning on many-step rewriting.

locale *ars* = *less-eq-syntax*
begin

Declaring some trans rules for relpowp:

lemma *relpowp2*[*trans*]: $a \sqsubseteq b \Longrightarrow b \sqsubseteq c \Longrightarrow ((\sqsubseteq) \hat{\ }\hat{\ } Suc\ (Suc\ 0))\ a\ c$
$\langle proof\rangle$

lemmas *relpowp-Suc-I*[*trans*] = *relpowp-Suc-I*[*of* - $(\sqsubseteq)$]
lemmas *relpowp-Suc-I2*[*trans*] = *relpowp-Suc-I2*[*of* - $(\sqsubseteq)$]

lemmas *relpowp-imp-rtranclp*[*simp*] = *relpowp-imp-rtranclp*[*of* - $(\sqsubseteq)$]

lemma *relpowp-imp-trancl*[*simp*]: $((\sqsubseteq)\hat{\ }\hat{\ }Suc\ n)\ s\ t \Longrightarrow (\sqsubseteq)^{++}\ s\ t$
$\langle proof\rangle$

declare *relpowp.simps*(*2*)[*simp del*]

lemma *relpowp-tranclp-trans*[*trans*]: $((\sqsubseteq)\hat{\ }\hat{\ }n)\ s\ t \Longrightarrow (\sqsubseteq)^{++}\ t\ u \Longrightarrow (\sqsubseteq)^{++}\ s\ u$
$\langle proof\rangle$

lemma *tranclp-relpowp-trans*[*trans*]: $(\sqsubseteq)^{++}\ s\ t \Longrightarrow ((\sqsubseteq)\hat{\ }\hat{\ }n)\ t\ u \Longrightarrow (\sqsubseteq)^{++}\ s\ u$
$\langle proof\rangle$

end

9.1 Closure Under Substitutions

We would like to define closure under substitution so that the source and target variables can have different types, but we cannot make polymorphic assumptions in a locale. Hence, we fix the two types of variables and specify two relations. This leads to a more general notion for a term relation that implies validity in an algebra. When the algebra is the term algebra, this property is closure under substitutions.

locale *stable* =
  fixes V **and** A **and** I **and** *less-eqT* (**infix** $\preceq$ *50*) **and** *less-eq* (**infix** $\sqsubseteq$ *50*)
  assumes *stable*: $s \preceq t \Longrightarrow \alpha :_s V \to A \Longrightarrow I[\![s]\!]\alpha \sqsubseteq I[\![t]\!]\alpha$

lemma *eq-stable*: *stable* V A I $(=)$ $(=)$
  ⟨*proof*⟩

lemma *Sup-stable*:
  assumes $\forall\, r \in R.$ *stable* V A I r *le*
  shows *stable* V A I $(\bigsqcup R)$ *le*
  ⟨*proof*⟩

context *stable* **begin**

lemma *super-stable*:
  assumes $(\sqsubseteq) \leq$ *le* **shows** *stable* V A I $(\preceq)$ *le*
  ⟨*proof*⟩

lemma *sub-stable*:
  assumes *le* $\leq (\preceq)$ **shows** *stable* V A I *le* $(\sqsubseteq)$
  ⟨*proof*⟩

lemma *dual-stable*: *stable* V A I $(\preceq)^-$ $(\sqsubseteq)^-$
  and *symclp-stable*: *stable* V A I (*symclp* $(\preceq)$) (*symclp* $(\sqsubseteq)$)
  ⟨*proof*⟩

lemma *relcompp-stable*:
  fixes *leA'* (**infix** $\sqsubseteq''$ *50*) **and** *leT'* (**infix** $\preceq''$ *50*)
  assumes *stable* V A I $(\preceq')$ $(\sqsubseteq')$
  shows *stable* V A I $((\preceq)$ *OO* $(\preceq'))$ $((\sqsubseteq)$ *OO* $(\sqsubseteq'))$
  ⟨*proof*⟩

lemma *relpowp-stable*: *stable* V A I $((\preceq)^{\frown\frown}n)$ $((\sqsubseteq)^{\frown\frown}n)$
  ⟨*proof*⟩

lemma *rtranclp-stable*: *stable* V A I (*rtranclp* $(\preceq)$) (*rtranclp* $(\sqsubseteq)$)
  ⟨*proof*⟩

lemma *tranclp-stable*: *stable* V A I (*tranclp* $(\preceq)$) (*tranclp* $(\sqsubseteq)$)
  ⟨*proof*⟩

lemma *related-imp-valid*:
assumes $s \preceq t$ **shows** $A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

end

locale *stable-algebra* = *sorted-algebra* + *stable*
begin

lemma *reflclp-on-stable*:
stable V A I (*reflclp-on* (*dom* $\mathcal{T}(F,V)$) ($\preceq$)) (*reflclp-on* (*dom* A) ($\sqsubseteq$))
⟨*proof*⟩

end

lemma *stable-mono*:
assumes $le \leq le'$ **shows** *stable* V A I leT $le \leq$ *stable* V A I leT le'
⟨*proof*⟩

lemma *stable-cmono*:
assumes $leT \leq leT'$ **shows** *stable* V A I $leT' \leq$ *stable* V A I leT
⟨*proof*⟩

Validity is stable under evaluation.

interpretation *valid*: *stable* V A I *valid* A I r V r
⟨*proof*⟩

Closure under substitution is a special instance, where the target is a term algebra.

locale *subst-closed-general* = *stable* V $\mathcal{T}(F,W)$ *Fun* ($\preceq$) ($\sqsubseteq$)
for F **and** V **and** W **and** *less-eqT* (**infix** $\preceq$ *50*) **and** *less-eq* (**infix** $\sqsubseteq$ *50*)
begin

sublocale *term-target*: *sorted-algebra* F $\mathcal{T}(F,W)$ *Fun*⟨*proof*⟩

sublocale *stable-algebra* F $\mathcal{T}(F,W)$ *Fun* ⟨*proof*⟩

sublocale *reflclp-on*: *stable* V $\mathcal{T}(F,W)$ *Fun reflclp-on* (*dom* $\mathcal{T}(F,V)$) ($\preceq$) *reflclp-on* (*dom* $\mathcal{T}(F,W)$) ($\sqsubseteq$)
⟨*proof*⟩

sublocale *symclp*: *stable* V $\mathcal{T}(F,W)$ *Fun symclp* ($\preceq$) *symclp* ($\sqsubseteq$)
⟨*proof*⟩

sublocale *rsymclp-on*: *stable* V $\mathcal{T}(F,W)$ *Fun rsymclp-on* (*dom* $\mathcal{T}(F,V)$) ($\preceq$) *rsymclp-on* (*dom* $\mathcal{T}(F,W)$) ($\sqsubseteq$)
⟨*proof*⟩

sublocale *relpowp*: *stable* V $\mathcal{T}(F,W)$ *Fun* $(\preceq)^{\frown\frown}n$ $(\sqsubseteq)^{\frown\frown}n$
⟨*proof*⟩

sublocale *rtranclp*: *stable* V $\mathcal{T}(F,W)$ *Fun rtranclp* $(\preceq)$ *rtranclp* $(\sqsubseteq)$
⟨*proof*⟩

sublocale *tranclp*: *stable* V $\mathcal{T}(F,W)$ *Fun tranclp* $(\preceq)$ *tranclp* $(\sqsubseteq)$
⟨*proof*⟩

sublocale *rtranclp-on*: *stable* V $\mathcal{T}(F,W)$ *Fun rtranclp-on* $(dom\ \mathcal{T}(F,V))$ $(\preceq)$ *rtranclp-on* $(dom\ \mathcal{T}(F,W))$ $(\sqsubseteq)$
⟨*proof*⟩

sublocale *rtransymclp-on*: *stable* V $\mathcal{T}(F,W)$ *Fun rtransymclp-on* $(dom\ \mathcal{T}(F,V))$ $(\preceq)$ *rtransymclp-on* $(dom\ \mathcal{T}(F,W))$ $(\sqsubseteq)$
⟨*proof*⟩

end

lemma *subst-closed-general-iff*:
subst-closed-general F V W $(\preceq)$ $(\sqsubseteq)$ $\longleftrightarrow$
$(\forall\, s\ t\ \vartheta.\ \ s \preceq t \longrightarrow \vartheta :_s V \to \mathcal{T}(F,W) \longrightarrow s{\cdot}\vartheta \sqsubseteq t{\cdot}\vartheta)$
for *le1* (**infix** $\preceq$ *50*) **and** *le2* (**infix** $\sqsubseteq$ *50*)
⟨*proof*⟩

locale *subst-closed* = *subst-closed-general* **where** *less-eq* = $(\preceq)$ **and** $W{=}V$
begin

lemma *valid-eq-relate*: *valid* $\mathcal{T}(F,V)$ *Fun* $(\preceq)$ V = $(\preceq)$
⟨*proof*⟩

lemma *reflclp-on-subst-closed*:
subst-closed F V (*reflclp-on* $(dom\ \mathcal{T}(F,V))$ $(\preceq)$)
⟨*proof*⟩

end

9.2 Rewrite Relations

A relation closed under context is where the term algebra is monotone.

locale *ctxt-closed* = *monotone* F $\mathcal{T}(F,V)$ *Fun* $(\preceq)$ **for** F V **and** *less-eqT* (**infix** $\preceq$ *50*)
begin

sublocale *term'*: *sorted-algebra* F $\mathcal{T}(F,V)$ *Fun*⟨*proof*⟩

sublocale *monotone-algebra* F $\mathcal{T}(F,V)$ *Fun* $(\preceq)$⟨*proof*⟩

sublocale *reflclp-on*: *monotone-algebra F* $\mathcal{T}(F,V)$ *Fun reflclp-on* (*dom* $\mathcal{T}(F,V)$) ($\preceq$)⟨*proof*⟩

sublocale *symclp*: *monotone-algebra F* $\mathcal{T}(F,V)$ *Fun symclp* ($\preceq$)⟨*proof*⟩

thm *ctxt-closed*

end

lemma *ctxt-closed F V* ($\preceq$) $\longleftrightarrow$ ($\forall f\ \pi s\ \sigma\ \varrho s\ \tau\ ls\ s\ t\ rs$.
f : πs @ σ # $\varrho s \to \tau$ *in F* $\longrightarrow$
ls :$_l$ πs *in* $\mathcal{T}(F,V)$ $\longrightarrow$ *s* : σ *in* $\mathcal{T}(F,V)$ $\longrightarrow$ *t* : σ *in* $\mathcal{T}(F,V)$ $\longrightarrow$ *rs* :$_l$ ϱs *in* $\mathcal{T}(F,V)$ $\longrightarrow$
s $\preceq$ *t* $\longrightarrow$ *Fun f* (*ls* @ *s* # *rs*) $\preceq$ *Fun f* (*ls* @ *t* # *rs*))
for *le* (**infix** $\preceq$ *50*)
⟨*proof*⟩

Especially, context-closed relation with subject reduction property maintains monotonicity over transitive closures.

locale *ctxt-closed-subject-reduction* =
ctxt-closed + *subject-reduction* $\mathcal{T}(F,V)$ ($\preceq$)
begin

sublocale *monotone-algebra-subject-reduction F* $\mathcal{T}(F,V)$ *Fun* ($\preceq$)⟨*proof*⟩

thm *tranclp.ctxt-closed*

end

locale *rewrite-relation* = *ctxt-closed* + *subst-closed*
begin

lemma *reflclp-on-rewrite-relation*:
rewrite-relation F V (*reflclp-on* (*dom* $\mathcal{T}(F,V)$) ($\preceq$))
⟨*proof*⟩

end

locale *rewrite-preorder* = *rewrite-relation* + *quasi-ordered-set dom* $\mathcal{T}(F,V)$ ($\preceq$)
begin

sublocale *quasi-ordered-monotone-algebra F* $\mathcal{T}(F,V)$ *Fun* ($\preceq$)⟨*proof*⟩

end

locale *sorted-rewrite-relation* = *ctxt-closed* + *subst-closed* + *sorted-relation* $\mathcal{T}(F,V)$ ($\preceq$)
begin

sublocale *ctxt-closed-subject-reduction*⟨*proof*⟩

interpretation *symclp*: *ctxt-closed-subject-reduction F V symclp* (⪯)
⟨*proof*⟩

sublocale *symclp*: *monotone F* $\mathcal{T}(F,V)$ *Fun symclp* (⪯)⟨*proof*⟩

sublocale *transymclp*: *monotone F* $\mathcal{T}(F,V)$ *Fun tranclp* (*symclp* (⪯))⟨*proof*⟩

sublocale *reflclp-on*: *monotone F* $\mathcal{T}(F,V)$ *Fun reflclp-on* (*dom* $\mathcal{T}(F,V)$) (⪯)
+ *rsymclp-on*: *monotone F* $\mathcal{T}(F,V)$ *Fun rsymclp-on* (*dom* $\mathcal{T}(F,V)$) (⪯)
+ *rtransymclp-on*: *monotone F* $\mathcal{T}(F,V)$ *Fun rtransymclp-on* (*dom* $\mathcal{T}(F,V)$) (⪯)
⟨*proof*⟩

sublocale *rtranclp-on*: *rewrite-preorder F V rtranclp-on* (*dom* $\mathcal{T}(F,V)$) (⪯)
⟨*proof*⟩

end

end

10 Sorted Rewriting

theory *Sorted-Rewriting*
imports *Sorted-Rewrite-Relations*
begin

We define a *sorted term rewrite system* as a set of sorted axioms over a signature. To define the rewrite relation over terms, we also specify the sorted set of variables which the rewritten terms may contain.

10.1 Root Rewrite Steps

The *root rewrite steps* are the instances of rewrite rules.

definition *rootstepp* (′(−-:-:-→$^{\varepsilon}$′) [*51*,*51*,*51*]*1000*) **where**
(−*F*:*V*:*R*→$^{\varepsilon}$) *s t* ≡ ∃(*X*. *l* ⇝ *r*) ∈ *R*. ∃ *ϑ* :$_s$ *X* → $\mathcal{T}(F,V)$. *s* = *l*·*ϑ* ∧ *t* = *r*·*ϑ*

interpretation *rootstepp*: *ars* (−*F*:*V*:*R*→$^{\varepsilon}$) **for** *F V R*⟨*proof*⟩

abbreviation *rootstep-op* (((*2*-) /−-:-:-→$^{\varepsilon}$/ (*2*-)) [*51*,*51*,*51*,*51*,*51*]*50*)
where *s* −*F*:*V*:*R*→$^{\varepsilon}$ *t* ≡ (−*F*:*V*:*R*→$^{\varepsilon}$) *s t*

lemma
s −*F*:*V*:*R*→$^{\varepsilon}$ *t* ≡ ∃(*X*. *l* ⇝ *r*) ∈ *R*. ∃ *ϑ* :$_s$ *X* → $\mathcal{T}(F,V)$. *s* = *l*·*ϑ* ∧ *t* = *r*·*ϑ*
⟨*proof*⟩

abbreviation *rootstep* ({−-:-:-→$^{\varepsilon}$} [*51*,*51*,*51*]*1000*)

where $\{-F{:}V{:}R{\to}^{\varepsilon}\} \equiv \{(s,t).\ s\ -F{:}V{:}R{\to}^{\varepsilon}\ t\}$

lemma *rootsteppI*: $(X.\ l \rightsquigarrow r) \in R \Longrightarrow \vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow s = l{\cdot}\vartheta \Longrightarrow t = r{\cdot}\vartheta \Longrightarrow s\ -F{:}V{:}R{\to}^{\varepsilon}\ t$
⟨*proof*⟩

lemma *rootstepp-root*: $(X.\ l \rightsquigarrow r) \in R \Longrightarrow \vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow l{\cdot}\vartheta\ -F{:}V{:}R{\to}^{\varepsilon}\ r{\cdot}\vartheta$
⟨*proof*⟩

lemma *rootsteppE*[*consumes 1*, *case-names root*]:
assumes $s\ -F{:}V{:}R{\to}^{\varepsilon}\ t$
and $\bigwedge X\ l\ r\ \vartheta.\ (X.\ l \rightsquigarrow r) \in R \Longrightarrow \vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow s = l{\cdot}\vartheta \Longrightarrow t = r{\cdot}\vartheta \Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *rootstepp-un*: $(-F{:}V{:}R{\cup}S{\to}^{\varepsilon}) = (-F{:}V{:}R{\to}^{\varepsilon}) \sqcup (-F{:}V{:}S{\to}^{\varepsilon})$
⟨*proof*⟩

lemma *rootstep-un*: $\{-F{:}V{:}R{\cup}S{\to}^{\varepsilon}\} = \{-F{:}V{:}R{\to}^{\varepsilon}\} \cup \{-F{:}V{:}S{\to}^{\varepsilon}\}$
⟨*proof*⟩

lemma *rootstepp-le-fun*:
assumes *FG*: $F \subseteq_m G$ **and** *st*: $s\ -F{:}V{:}R{\to}^{\varepsilon}\ t$ **shows** $s\ -G{:}V{:}R{\to}^{\varepsilon}\ t$
⟨*proof*⟩

lemma *rootstepp-mono-fun*:
assumes *FG*: $F \subseteq_m G$ **shows** $(-F{:}V{:}R{\to}^{\varepsilon}) \leq (-G{:}V{:}R{\to}^{\varepsilon})$
⟨*proof*⟩

lemma *rootstepp-le-var*:
assumes *VW*: $V \subseteq_m W$ **and** *st*: $s\ -F{:}V{:}R{\to}^{\varepsilon}\ t$ **shows** $s\ -F{:}W{:}R{\to}^{\varepsilon}\ t$
⟨*proof*⟩

lemma *rootstepp-mono-var*:
assumes *VW*: $V \subseteq_m W$ **shows** $(-F{:}V{:}R{\to}^{\varepsilon}) \leq (-F{:}W{:}R{\to}^{\varepsilon})$
⟨*proof*⟩

Rewrite steps are closed under substitution. Note that here the types of variables are polymorphic.

interpretation *rootstepp*: *subst-closed-general F V W* $(-F{:}V{:}R{\to}^{\varepsilon})$ $(-F{:}W{:}R{\to}^{\varepsilon})$
⟨*proof*⟩

thm *rootstepp.stable*
thm *rootstepp.tranclp.stable*

interpretation *rootstepp*: *subst-closed F V* $(-F{:}V{:}R{\to}^{\varepsilon})$⟨*proof*⟩

10.2 Rewrite Steps

The *rewrite steps* are their closure under sorted contexts.

inductive *stepp* ($'(-\text{-:-:-}\to')$ [*51*,*51*,*51*]*1000*) **for** F V R **where**
  root: $(-F\text{:}V\text{:}R\to)$ s t **if** s $-F\text{:}V\text{:}R\to^{\varepsilon}$ t
| *comp*: $(-F\text{:}V\text{:}R\to)$ (*Fun* f (ls @ s # rs)) (*Fun* f (ls @ t # rs))
  if $f : \pi s @ \sigma \# \varrho s \to \tau$ *in* F
  and $ls :_l \pi s$ *in* $\mathcal{T}(F,V)$ **and** $s : \sigma$ *in* $\mathcal{T}(F,V)$ **and** $t : \sigma$ *in* $\mathcal{T}(F,V)$ **and** $rs :_l \varrho s$ *in* $\mathcal{T}(F,V)$
  and $(-F\text{:}V\text{:}R\to)$ s t
  for f πs σ ϱs τ ls rs s t

hide-fact(**open**) *stepp.root stepp.comp*

declare *stepp.root*[*simp*]

abbreviation *step-op* ((*2-*) $/-\text{-:-:-}\to$ /(*2-*)) [*51*,*51*,*51*,*51*,*51*]*50*)
  where s $-F\text{:}V\text{:}R\to$ $t \equiv (-F\text{:}V\text{:}R\to)$ s t

abbreviation *step* ({$-\text{-:-:-}\to$}) [*51*,*51*,*51*]*1000*)
  where $\{-F\text{:}V\text{:}R\to\} \equiv \{(s,t).\ s\ -F\text{:}V\text{:}R\to\ t\}$

abbreviation(*input*) *dual-stepp* ($'(\leftarrow\text{-:-:-}-')$ [*51*,*51*,*51*]*1000*)
  where $(\leftarrow F\text{:}V\text{:}R-) \equiv (-F\text{:}V\text{:}R\to)^-$

abbreviation *dual-step* ({$\leftarrow\text{-:-:-}-$} [*51*,*51*,*51*]*1000*)
  where $\{\leftarrow F\text{:}V\text{:}R-\} \equiv \{-F\text{:}V\text{:}R\to\}^{-1}$

For reflexive closure, it is convenient to stay within well-sorted terms.

definition *step-reflclp* ($'(-\text{-:-:-}\to^=')$ [*51*,*51*,*51*]*1000*)
  where $(-F\text{:}V\text{:}R\to^=) \equiv$ *reflclp-on* (*dom* $\mathcal{T}(F,V)$) $(-F\text{:}V\text{:}R\to)$

abbreviation *step-reflcl-op* ((*2-*)/ $-\text{-:-:-}\to^=$ /(*2-*)) [*51*,*51*,*51*,*51*,*51*]*50*)
  where s $-F\text{:}V\text{:}R\to^=$ $t \equiv (-F\text{:}V\text{:}R\to^=)$ s t

abbreviation *step-reflcl* ({$-\text{-:-:-}\to^=$} [*51*,*51*,*51*]*1000*)
  where $\{-F\text{:}V\text{:}R\to^=\} \equiv \{(s,t).\ s\ -F\text{:}V\text{:}R\to^=\ t\}$

abbreviation(*input*) *dual-step-reflclp* ($'(\leftarrow\text{-:-:-}-^=')$ [*51*,*51*,*51*]*1000*)
  where $(\leftarrow F\text{:}V\text{:}R-^=) \equiv (-F\text{:}V\text{:}R\to^=)^-$

abbreviation(*input*) *dual-step-reflcl* ({$\leftarrow\text{-:-:-}-^=$} [*51*,*51*,*51*]*1000*)
  where $\{\leftarrow F\text{:}V\text{:}R-^=\} \equiv \{-F\text{:}V\text{:}R\to^=\}^{-1}$

abbreviation *stepp-fold* ((*2-*)/ $-\text{-:-:-}\to\hat{}$- /(*2-*)) [*51*,*51*,*51*,*51*,*1000*,*51*]*50*)
  where s $-F\text{:}V\text{:}R\to\hat{}n$ $t \equiv ((-F\text{:}V\text{:}R\to)\hat{}\hat{}n)$ s t

abbreviation *step-trancl-op* ((*2-*)/ $-\text{-:-:-}\to^+$ /(*2-*)) [*51*,*51*,*51*,*51*,*51*]*50*)
  where s $-F\text{:}V\text{:}R\to^+$ $t \equiv (-F\text{:}V\text{:}R\to)^{++}$ s t

abbreviation *step-rtranclp* (*'*(*−-:-:-→*'*) [*51*,*51*,*51*]*1000*)
  where (*−F*:*V*:*R*→*) ≡ (*−F*:*V*:*R*→$^{=}$)$^{++}$

abbreviation *step-rtrancl-op* ((*(2-)*/ *−-:-:-→* /(2-)*) [*51*,*51*,*51*,*51*,*51*]*50*)
  where *s −F*:*V*:*R*→* *t* ≡ (*−F*:*V*:*R*→*) *s t*

abbreviation *step-rtrancl* ({*−-:-:-→**} [*51*,*51*,*51*]*1000*)
  where {*−F*:*V*:*R*→*} ≡ {(*s*,*t*). *s −F*:*V*:*R*→* *t*}

abbreviation *dual-step-rtrancl* ({←-:-:-−*} [*51*,*51*,*51*]*1000*)
  where {←*F*:*V*:*R*−*} ≡ {*−F*:*V*:*R*→*}$^{-1}$

abbreviation *step-symclp* (*'*(←-:-:-→*'*) [*51*,*51*,*51*]*1000*)
  where (←*F*:*V*:*R*→) ≡ *symclp* (*−F*:*V*:*R*→)

abbreviation *step-symcl-op* ((*(2-)*/ ←-:-:-→ */(2-)*) [*51*,*51*,*51*,*51*,*51*]*50*)
  where *s* ←*F*:*V*:*R*→ *t* ≡ (←*F*:*V*:*R*→) *s t*

abbreviation *step-symcl* ({←-:-:-→} [*51*,*51*,*51*]*1000*)
  where {←*F*:*V*:*R*→} ≡ {(*s*,*t*). *s* ←*F*:*V*:*R*→ *t*}

abbreviation *step-rsymclp* (*'*(←-:-:-→$^{=}$*'*) [*51*,*51*,*51*]*1000*)
  where (←*F*:*V*:*R*→$^{=}$) ≡ *symclp* (*−F*:*V*:*R*→$^{=}$)

abbreviation *step-rsymcl-op* ((*(2-)*/ ←-:-:-→$^{=}$ */(2-)*) [*51*,*51*,*51*,*51*,*51*]*50*)
  where *s* ←*F*:*V*:*R*→$^{=}$ *t* ≡ (←*F*:*V*:*R*→$^{=}$) *s t*

abbreviation *step-rtransymclp* (*'*(←-:-:-→*'*) [*51*,*51*,*51*]*1000*)
  where (←*F*:*V*:*R*→*) ≡ (←*F*:*V*:*R*→$^{=}$)$^{++}$

abbreviation *step-rtransymcl-op* ((*(2-)*/ ←-:-:-→* */(2-)*) [*51*,*51*,*51*,*51*,*51*]*50*)
  where *s* ←*F*:*V*:*R*→* *t* ≡ (←*F*:*V*:*R*→*) *s t*

abbreviation *step-rtransymcl* ({←-:-:-→*} [*51*,*51*,*51*]*1000*)
  where {←*F*:*V*:*R*→*} ≡ {(*s*,*t*). *s* ←*F*:*V*:*R*→* *t*}

abbreviation *step-rsymcl* ({←-:-:-→$^{=}$} [*51*,*51*,*51*]*1000*)
  where {←*F*:*V*:*R*→$^{=}$} ≡ {(*s*,*t*). *s* ←*F*:*V*:*R*→$^{=}$ *t*}

interpretation *stepp*: *ars* (*−F*:*V*:*R*→) **for** *F V R*⟨*proof*⟩

lemma *rootstepp-le-step*[*rewriting-simps*]: (*−F*:*V*:*R*→$^{\varepsilon}$) ≤ (*−F*:*V*:*R*→) ⟨*proof*⟩

We first state that rewrite step is closed under substitution with heterogeneous variables.

interpretation *stepp*: *subst-closed-general F V W* (*−F*:*V*:*R*→) (*−F*:*W*:*R*→)
  rewrites ⋀*X*. *reflclp-on* (*dom* $\mathcal{T}$(*F*,*X*)) (*−F*:*X*:*R*→) ≡ (*−F*:*X*:*R*→$^{=}$)
⟨*proof*⟩

This gives, e.g., that transitive closures are closed under substitutions.

thm *stepp.stable*[*of F V R s t ϑ W*]
thm *stepp.rtranclp-on.stable*
thm *stepp.rtransymclp-on.stable*

interpretation *stepp*: *subst-closed F V* $(-F{:}V{:}R{\to})$
rewrites $\bigwedge X.$ *reflclp-on* (*dom* $\mathcal{T}(F,X)$) $(-F{:}X{:}R{\to}) \equiv (-F{:}X{:}R{\to}^{=})$
⟨*proof*⟩

Then we state that the rewrite step is a rewrite relation.

interpretation *stepp*: *rewrite-relation F V* $(-F{:}V{:}R{\to})$
rewrites $\bigwedge X.$ *reflclp-on* (*dom* $\mathcal{T}(F,X)$) $(-F{:}X{:}R{\to}) \equiv (-F{:}X{:}R{\to}^{=})$
⟨*proof*⟩

thm *stepp.ctxt-closed*
thm *stepp.reflclp-on.ctxt-closed*
thm *stepp.symclp.ctxt-closed*

lemma *steppI-ctxt-rootstepp*:
assumes *st*: $s\ {-F{:}V{:}R{\to}^{\varepsilon}}\ t$ **and** *C*: $C : \sigma \to \tau$ *in* $\mathcal{C}(F,V)$
and *s*: $s : \sigma$ *in* $\mathcal{T}(F,V)$ **and** *t*: $t : \sigma$ *in* $\mathcal{T}(F,V)$
and *u*: $u = C\langle s\rangle$ **and** *v*: $v = C\langle t\rangle$
shows $u\ {-F{:}V{:}R{\to}}\ v$
⟨*proof*⟩

lemmas *stepp-ctxt-rootstepp* = *steppI-ctxt-rootstepp*[*OF - - - - refl refl*]

The rewrite step is either the root step or a typed root step under nonempty typed context. Distinguishing the first case is necessary only when the rewrite rules are not well-typed.

lemma *stepp-iff-rootstep-or*:
$s\ {-F{:}V{:}R{\to}}\ t \longleftrightarrow$
$s\ {-F{:}V{:}R{\to}^{\varepsilon}}\ t \vee$
$(\exists\, C\ \sigma\ \tau\ u\ v.\ C : \sigma \to \tau$ *in* $\mathcal{C}(F,V) \wedge u : \sigma$ *in* $\mathcal{T}(F,V) \wedge v : \sigma$ *in* $\mathcal{T}(F,V) \wedge C \neq \Box \wedge$
$u\ {-F{:}V{:}R{\to}^{\varepsilon}}\ v \wedge s = C\langle u\rangle \wedge t = C\langle v\rangle)$
(**is** *?l* $\longleftrightarrow$ *?m* $\vee$ *?r s t*)
⟨*proof*⟩

lemma *steppE-rootstep-or*[*consumes 1*, *case-names root ctxt*]:
assumes $s\ {-F{:}V{:}R{\to}}\ t$
and $s\ {-F{:}V{:}R{\to}^{\varepsilon}}\ t \Longrightarrow$ *thesis*
and $\bigwedge C\ \sigma\ \tau\ u\ v.\ C : \sigma \to \tau$ *in* $\mathcal{C}(F,V) \Longrightarrow$
$u : \sigma$ *in* $\mathcal{T}(F,V) \Longrightarrow v : \sigma$ *in* $\mathcal{T}(F,V) \Longrightarrow C \neq \Box \Longrightarrow$
$u\ {-F{:}V{:}R{\to}^{\varepsilon}}\ v \Longrightarrow s = C\langle u\rangle \Longrightarrow t = C\langle v\rangle \Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *stepp-un*: $(-F{:}V{:}R{\cup}S{\rightarrow}) = (-F{:}V{:}R{\rightarrow}) \sqcup (-F{:}V{:}S{\rightarrow})$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *step-un*: $\{-F{:}V{:}R{\cup}S{\rightarrow}\} = \{-F{:}V{:}R{\rightarrow}\} \cup \{-F{:}V{:}S{\rightarrow}\}$ ⟨*proof*⟩

lemma *Context-le-fun*:
assumes *FG*: $F \subseteq_m G$ **and** *C*: $C : \sigma \rightarrow \tau$ *in* $\mathcal{C}(F,V)$ **shows** $C : \sigma \rightarrow \tau$ *in* $\mathcal{C}(G,V)$
⟨*proof*⟩

lemma *Context-mono-fun*: **assumes** *FG*: $F \subseteq_m G$ **shows** $\mathcal{C}(F,V) \subseteq_m \mathcal{C}(G,V)$
⟨*proof*⟩

lemma *Context-le-var*:
assumes *VW*: $V \subseteq_m W$ **and** *C*: $C : \sigma \rightarrow \tau$ *in* $\mathcal{C}(F,V)$ **shows** $C : \sigma \rightarrow \tau$ *in* $\mathcal{C}(F,W)$
⟨*proof*⟩

lemma *Context-mono-var*: **assumes** *VW*: $V \subseteq_m W$ **shows** $\mathcal{C}(F,V) \subseteq_m \mathcal{C}(F,W)$
⟨*proof*⟩

lemma *step-le-fun*: **assumes** *FG*: $F \subseteq_m G$ **and** *st*: $s\ {-}F{:}V{:}R{\rightarrow}\ t$ **shows** $s\ {-}G{:}V{:}R{\rightarrow}\ t$
⟨*proof*⟩

lemma *step-mono-fun*: **assumes** *FG*: $F \subseteq_m G$ **shows** $(-F{:}V{:}R{\rightarrow}) \leq (-G{:}V{:}R{\rightarrow})$
⟨*proof*⟩

lemma *step-le-var*: **assumes** *VW*: $V \subseteq_m W$ **and** *st*: $s\ {-}F{:}V{:}R{\rightarrow}\ t$ **shows** $s\ {-}F{:}W{:}R{\rightarrow}\ t$
⟨*proof*⟩

lemma *step-mono-var*: **assumes** *VW*: $V \subseteq_m W$ **shows** $(-F{:}V{:}R{\rightarrow}) \leq (-F{:}W{:}R{\rightarrow})$
⟨*proof*⟩

10.3 Closures

interpretation *step-reflclp*: *reflexive dom* $\mathcal{T}(F,V)$ $(-F{:}V{:}R{\rightarrow}^{=})$
⟨*proof*⟩

lemma *step-reflclp-iff*: $s\ {-}F{:}V{:}R{\rightarrow}^{=}\ t \longleftrightarrow s = t \wedge s \in \mathit{dom}\ \mathcal{T}(F,V) \vee s\ {-}F{:}V{:}R{\rightarrow}\ t$
⟨*proof*⟩

lemma *step-reflclpE*[*consumes 1*, *case-names refl step*]:
assumes $s\ {-}F{:}V{:}R{\rightarrow}^{=}\ t$
and $\bigwedge\sigma.\ s : \sigma$ *in* $\mathcal{T}(F,V) \Longrightarrow s = t \Longrightarrow$ *thesis*
and $s\ {-}F{:}V{:}R{\rightarrow}\ t \Longrightarrow$ *thesis*
shows *thesis*

⟨*proof*⟩

lemma *step-reflclp-refl*[*simp*]: $s : \sigma$ *in* $\mathcal{T}(F,V) \Longrightarrow s\ -F{:}V{:}R{\to}^{=}\ s$
and *step-reflclp-step*[*simp*]: $s\ -F{:}V{:}R{\to}\ t \Longrightarrow s\ -F{:}V{:}R{\to}^{=}\ t$
⟨*proof*⟩

lemma *stepp-le-reflcl*[*rewriting-simps*]: $(-F{:}V{:}R{\to}) \leq (-F{:}V{:}R{\to}^{=})$ ⟨*proof*⟩

interpretation *step-reflclp*: *reflexive-monotone-algebra* $F\ \mathcal{T}(F,V)$ *Fun* $(-F{:}V{:}R{\to}^{=})$
⟨*proof*⟩

interpretation *step-tranclp*: *transitive-algebra* $F\ \mathcal{T}(F,V)$ *Fun* $(-F{:}V{:}R{\to})^{++}$
⟨*proof*⟩

lemma *step-trancl*[*simp*]: $(s,t) \in \{-F{:}V{:}R{\to}\}^{+} \longleftrightarrow s\ -F{:}V{:}R{\to}^{+}\ t$
⟨*proof*⟩

lemma *step-rtranclp-eq*: $(-F{:}V{:}R{\to}^{*}) = rtranclp\text{-}on\ (dom\ \mathcal{T}(F,V))\ (-F{:}V{:}R{\to})$
⟨*proof*⟩

interpretation *step-rtranclp*: *quasi-ordered-algebra* $F\ \mathcal{T}(F,V)$ *Fun* $(-F{:}V{:}R{\to}^{*})$
⟨*proof*⟩

lemma *step-rtranclp-trancl*[*rewriting-simps*]: $s\ -F{:}V{:}R{\to}^{+}\ t \Longrightarrow s\ -F{:}V{:}R{\to}^{*}\ t$
⟨*proof*⟩

lemma *step-tranclp-step*[*rewriting-simps*]: $s\ -F{:}V{:}R{\to}\ t \Longrightarrow s\ -F{:}V{:}R{\to}^{+}\ t$ ⟨*proof*⟩

lemma [*rewriting-simps*]:
shows *step-tranclp-le-rtrancl*: $(-F{:}V{:}R{\to})^{++} \leq (-F{:}V{:}R{\to}^{*})$
and *stepp-le-rtrancl*: $(-F{:}V{:}R{\to}) \leq (-F{:}V{:}R{\to}^{*})$
⟨*proof*⟩

lemma *step-rtranclp-eq-reflclp*:
$(-F{:}V{:}R{\to}^{*}) = reflclp\text{-}on\ (dom\ \mathcal{T}(F,V))\ (-F{:}V{:}R{\to})^{++}$
⟨*proof*⟩

lemma *step-rtranclpE*[*consumes 1*, *case-names refl trancl*]:
assumes $s\ -F{:}V{:}R{\to}^{*}\ t$
and $\bigwedge\sigma.\ s : \sigma$ *in* $\mathcal{T}(F,V) \Longrightarrow s = t \Longrightarrow thesis$
and $s\ -F{:}V{:}R{\to}^{+}\ t \Longrightarrow thesis$
shows *thesis* ⟨*proof*⟩

lemma *stepp-transs*[*trans*]:
$s\ -F{:}V{:}R{\to}^{=}\ t \Longrightarrow t\ -F{:}V{:}R{\to}\ u \Longrightarrow s\ -F{:}V{:}R{\to}^{+}\ u$
$s\ -F{:}V{:}R{\to}\ t \Longrightarrow t\ -F{:}V{:}R{\to}^{=}\ u \Longrightarrow s\ -F{:}V{:}R{\to}^{+}\ u$
$s\ -F{:}V{:}R{\to}^{=}\ t \Longrightarrow t\ -F{:}V{:}R{\to}^{=}\ u \Longrightarrow s\ -F{:}V{:}R{\to}^{*}\ u$
$s\ -F{:}V{:}R{\to}^{*}\ t \Longrightarrow t\ -F{:}V{:}R{\to}\ u \Longrightarrow s\ -F{:}V{:}R{\to}^{+}\ u$

$s -F:V:R\to t \Longrightarrow t -F:V:R\to^* u \Longrightarrow s -F:V:R\to^+ u$
$s -F:V:R\to^+ t \Longrightarrow t -F:V:R\to^= u \Longrightarrow s -F:V:R\to^+ u$
$s -F:V:R\to^= t \Longrightarrow t -F:V:R\to^+ u \Longrightarrow s -F:V:R\to^+ u$
$s -F:V:R\to^+ t \Longrightarrow t -F:V:R\to^* u \Longrightarrow s -F:V:R\to^+ u$
$s -F:V:R\to^* t \Longrightarrow t -F:V:R\to^+ u \Longrightarrow s -F:V:R\to^+ u$
⟨*proof*⟩

lemma *step-rsymclp-iff*: $s \leftarrow F:V:R\to^= t \longleftrightarrow (s = t \wedge s \in dom\ \mathcal{T}(F,V)) \vee s \leftarrow F:V:R\to t$
⟨*proof*⟩

lemma *step-rsymclpE*[*consumes 1*, *case-names refl sym*]:
assumes $s \leftarrow F:V:R\to^= t$
and $\bigwedge\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow s = t \Longrightarrow thesis$
and $s \leftarrow F:V:R\to t \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma [*rewriting-simps*]:
shows *step-rsymclp-refl*: $s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow s \leftarrow F:V:R\to^= s$
and *step-rsymclp-symcl*: $s \leftarrow F:V:R\to t \Longrightarrow s \leftarrow F:V:R\to^= t$
⟨*proof*⟩

lemma [*rewriting-simps*]:
shows *step-symclp-le-rsymcl*: $(\leftarrow F:V:R\to) \leq (\leftarrow F:V:R\to^=)$
and *stepp-le-rsymcl*: $(-F:V:R\to) \leq (\leftarrow F:V:R\to^=)$
and *stepp-dual-le-rsymcl*: $(\leftarrow F:V:R-) \leq (\leftarrow F:V:R\to^=)$
⟨*proof*⟩

interpretation *step-rsymclp-on*: *reflexive dom* $\mathcal{T}(F,V)$ $(\leftarrow F:V:R\to^=)$
⟨*proof*⟩

lemma *step-tranclp-le-rtransymcl*[*rewriting-simps*]: $(-F:V:R\to)^{++} \leq (\leftarrow F:V:R\to^*)$
⟨*proof*⟩

lemma *step-rtransymclp-trancl*[*rewriting-simps*]: $s -F:V:R\to^+ t \Longrightarrow s \leftarrow F:V:R\to^* t$
⟨*proof*⟩

lemma *step-dual-rtransymclp-trancl*[*rewriting-simps*]: $t -F:V:R\to^+ s \Longrightarrow s \leftarrow F:V:R\to^* t$
⟨*proof*⟩

lemma *step-rtransymclp-rtrancl*[*rewriting-simps*]: $s -F:V:R\to^* t \Longrightarrow s \leftarrow F:V:R\to^* t$
⟨*proof*⟩

lemma *step-dual-rtransymclp-rtrancl*[*rewriting-simps*]: $t -F:V:R\to^* s \Longrightarrow s \leftarrow F:V:R\to^* t$

⟨*proof*⟩

lemma *step-rtransymclp-transs*[*trans*]:
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow t -_{F:V:R}\rightarrow\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow u -_{F:V:R}\rightarrow\ t \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s -_{F:V:R}\rightarrow\ t \Longrightarrow t \leftarrow_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$t -_{F:V:R}\rightarrow\ s \Longrightarrow t \leftarrow_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow t -_{F:V:R}{}^{=}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow u -_{F:V:R}{}^{=}\ t \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s -_{F:V:R}{}^{=}\ t \Longrightarrow t \leftarrow_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$t -_{F:V:R}{}^{=}\ s \Longrightarrow t \leftarrow_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow t -_{F:V:R}{}^{+}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow u -_{F:V:R}{}^{+}\ t \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s -_{F:V:R}{}^{+}\ t \Longrightarrow t \leftarrow_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$t -_{F:V:R}{}^{+}\ s \Longrightarrow t \leftarrow_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
$s \leftarrow_{F:V:R}{}^{*}\ t \Longrightarrow t -_{F:V:R}{}^{*}\ u \Longrightarrow s \leftarrow_{F:V:R}{}^{*}\ u$
⟨*proof*⟩

10.4 Sorted Rewrite Systems

Now we demand that all rewrite rules respect sorts.

locale *sorted-trs* =
fixes $F :: ('f,'s)$ *ssig* **and** $R :: ('f,'x,'s)$ *axiom set*
assumes *axiom-typed*:
$(X.\ l \rightsquigarrow r) \in R \Longrightarrow \exists \sigma.\ l : \sigma\ in\ \mathcal{T}(F,X) \wedge r : \sigma\ in\ \mathcal{T}(F,X)$
begin

lemmas *axiom-hastypeE* = *axiom-typed*[*THEN exE*, *unfolded conj-imp-eq-imp-imp*]

lemma *axiom-hastype-iff*: $(X.\ l \rightsquigarrow r) \in R \Longrightarrow l : \sigma\ in\ \mathcal{T}(F,X) \longleftrightarrow r : \sigma\ in\ \mathcal{T}(F,X)$
⟨*proof*⟩

lemma
assumes $(X.\ l \rightsquigarrow r) \in R$
shows *lhs-in-dom*: $l \in dom\ \mathcal{T}(F,X)$ **and** *rhs-in-dom*: $r \in dom\ \mathcal{T}(F,X)$
⟨*proof*⟩

lemma *rootstepp-typed*: $s -_{F:V:R}\rightarrow^{\varepsilon} t \Longrightarrow \exists \sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

sublocale *rootstepp*: *relation-on dom* $\mathcal{T}(F,V)$ $(-_{F:V:R}\rightarrow^{\varepsilon})$
⟨*proof*⟩

lemma *stepp-typed*: $s -_{F:V:R}\rightarrow t \Longrightarrow \exists \sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

sublocale *stepp*: *relation-on dom* $\mathcal{T}(F,V)$ $(-_{F:V:R}\rightarrow)$
⟨*proof*⟩

sublocale *step-reflclp*: *relation-on dom* $\mathcal{T}(F,V)$ $(-F{:}V{:}R{\to}^{=})$
⟨*proof*⟩

sublocale *step-tranclp*: *relation-on dom* $\mathcal{T}(F,V)$ *tranclp* $(-F{:}V{:}R{\to})$
⟨*proof*⟩

sublocale *step-rtranclp*: *relation-on dom* $\mathcal{T}(F,V)$ $(-F{:}V{:}R{\to}^{*})$
⟨*proof*⟩

sublocale *step-symclp*: *relation-on dom* $\mathcal{T}(F,V)$ $(\leftarrow F{:}V{:}R{\to})$
⟨*proof*⟩

sublocale *step-rsymclp*: *relation-on dom* $\mathcal{T}(F,V)$ $(\leftarrow F{:}V{:}R{\to}^{=})$
⟨*proof*⟩

sublocale *step-rtransymclp*: *relation-on dom* $\mathcal{T}(F,V)$ $(\leftarrow F{:}V{:}R{\to}^{*})$
⟨*proof*⟩

lemma *step-reflclp-typed*: $s\ {-}F{:}V{:}R{\to}^{=}\ t \Longrightarrow \exists\,\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *step-tranclp-typed*: $s\ {-}F{:}V{:}R{\to}^{+}\ t \Longrightarrow \exists\,\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *step-rtranclp-typed*: $s\ {-}F{:}V{:}R{\to}^{*}\ t \Longrightarrow \exists\,\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *step-symclp-typed*: $s \leftarrow F{:}V{:}R{\to}\ t \Longrightarrow \exists\,\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *step-rsymclp-typed*: $s \leftarrow F{:}V{:}R{\to}^{=}\ t \Longrightarrow \exists\,\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *step-rtransymclp-typed*: $s \leftarrow F{:}V{:}R{\to}^{*}\ t \Longrightarrow \exists\,\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *stepp-iff-ctxt-rule*:
$s\ {-}F{:}V{:}R{\to}\ t \longleftrightarrow$
$(\exists\, C\ \sigma\ \tau\ X\ l\ r\ \vartheta.$
$C : \sigma \to \tau\ in\ \mathcal{C}(F,V) \wedge (X.\ l \rightsquigarrow r) \in R \wedge \vartheta :_s X \to \mathcal{T}(F,V) \wedge l : \sigma\ in\ \mathcal{T}(F,X)$
$\wedge$
$s = C\langle l{\cdot}\vartheta\rangle \wedge t = C\langle r{\cdot}\vartheta\rangle)$ (**is** $?l \longleftrightarrow (\exists\, C\ \sigma.\ ?r\ C\ \sigma)$)

⟨*proof*⟩

lemma *stepp-iff-ctxt-rootstepp*:
$s\ -F{:}V{:}R{\to}\ t \longleftrightarrow$
$(\exists\ C\ \sigma\ \tau\ u\ v.$
$C : \sigma \to \tau\ in\ \mathcal{C}(F,V) \wedge u : \sigma\ in\ \mathcal{T}(F,V) \wedge s = C\langle u\rangle \wedge t = C\langle v\rangle \wedge u\ -F{:}V{:}R{\to}^{\varepsilon}$
$v)$
⟨*proof*⟩

lemma *steppE-ctxt-rule*:
assumes $s\ -F{:}V{:}R{\to}\ t$
and $\bigwedge C\ \sigma\ \tau\ X\ l\ r\ \vartheta.$
$C : \sigma \to \tau\ in\ \mathcal{C}(F,V) \Longrightarrow (X.\ l \rightsquigarrow r) \in R \wedge \vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow l : \sigma\ in$
$\mathcal{T}(F,X) \Longrightarrow$
$s = C\langle l{\cdot}\vartheta\rangle \Longrightarrow t = C\langle r{\cdot}\vartheta\rangle \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

sublocale *rootstepp*: *sorted-relation* $\mathcal{T}(F,V)\ (-F{:}V{:}R{\to}^{\varepsilon})$
⟨*proof*⟩

sublocale *stepp*: *sorted-rewrite-relation* $F\ V\ (-F{:}V{:}R{\to})$
rewrites $\bigwedge X.$ *reflclp-on* $(dom\ \mathcal{T}(F,X))\ (-F{:}X{:}R{\to}) \equiv (-F{:}X{:}R{\to}^{=})$
⟨*proof*⟩

thm *stepp.rtranclp-on.ctxt-closed*

sublocale *step-symclp*: *sorted-relation* $\mathcal{T}(F,V)\ (\leftarrow F{:}V{:}R{\to})$
⟨*proof*⟩

sublocale *step-reflclp*: *sorted-relation* $\mathcal{T}(F,V)\ (-F{:}V{:}R{\to}^{=})$
⟨*proof*⟩

sublocale *step-rsymclp*: *sorted-relation* $\mathcal{T}(F,V)\ (\leftarrow F{:}V{:}R{\to}^{=})$
⟨*proof*⟩

lemma *arg-stepp-imp-ex-stepp*:
assumes *fss*: $Fun\ f\ ss \in dom\ \mathcal{T}(F,V)$ **and** *s*: $s \in set\ ss$
and *st*: $s\ -F{:}V{:}R{\to}\ t$ **shows** $\exists\, t'.\ Fun\ f\ ss\ -F{:}V{:}R{\to}\ t'$
⟨*proof*⟩

lemma *step-rtranclp-induct*[*consumes 1*, *case-names refl step*]:
assumes *st*: $s\ -F{:}V{:}R{\to}^{*}\ t$
and *Refl*: $\bigwedge\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow P\ s$
and *Step*: $\bigwedge t\ u.\ s\ -F{:}V{:}R{\to}^{*}\ t \Longrightarrow t\ -F{:}V{:}R{\to}\ u \Longrightarrow P\ t \Longrightarrow P\ u$
shows $P\ t$
⟨*proof*⟩

lemma *step-rtranclp-induct-dual*[*consumes 1*, *case-names refl step*]:

assumes *st*: $s\ {-}F{:}V{:}R{\to}^*\ t$
and *Refl*: $\bigwedge\sigma.\ t : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow P\ t$
and *Step*: $\bigwedge s\ u.\ s\ {-}F{:}V{:}R{\to}\ u \Longrightarrow u\ {-}F{:}V{:}R{\to}^*\ t \Longrightarrow P\ u \Longrightarrow P\ s$
shows $P\ s$
⟨*proof*⟩

lemma *step-rtransymclp-induct*[*consumes 1*, *case-names refl step*]:
assumes *st*: $s \leftarrow F{:}V{:}R{\to}^*\ t$
and *Refl*: $\bigwedge\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow P\ s$
and *Step*: $\bigwedge t\ u.\ s \leftarrow F{:}V{:}R{\to}^*\ t \Longrightarrow t \leftarrow F{:}V{:}R{\to}\ u \Longrightarrow P\ t \Longrightarrow P\ u$
shows $P\ t$
⟨*proof*⟩

sublocale *step-tranclp*: *transitive-monotone-algebra* $F\ \mathcal{T}(F,V)\ Fun\ ({-}F{:}V{:}R{\to})^{++}$⟨*proof*⟩

end

10.5 Models of Rewrite Systems

The term algebra coupled with rewrite steps is a model of the TRS.

lemma *rootstepp-models*: $\mathcal{T}(F,V){:}Fun{:}({-}F{:}V{:}R{\to}^{\varepsilon}) \models unconditional\ `\ R$
⟨*proof*⟩

lemma *stepp-models*: $\mathcal{T}(F,V){:}Fun{:}({-}F{:}V{:}R{\to}) \models unconditional\ `\ R$
⟨*proof*⟩

lemma *step-tranclp-models*: $\mathcal{T}(F,V){:}Fun{:}({-}F{:}V{:}R{\to})^{++} \models unconditional\ `\ R$
and *step-reflclp-models*: $\mathcal{T}(F,V){:}Fun{:}({-}F{:}V{:}R{\to}^{=}) \models unconditional\ `\ R$
and *step-rtranclp-models*: $\mathcal{T}(F,V){:}Fun{:}({-}F{:}V{:}R{\to}^*) \models unconditional\ `\ R$
and *step-symclp-models*: $\mathcal{T}(F,V){:}Fun{:}(\leftarrow F{:}V{:}R{\to}) \models unconditional\ `\ R$
and *step-rsymclp-models*: $\mathcal{T}(F,V){:}Fun{:}(\leftarrow F{:}V{:}R{\to}^{=}) \models unconditional\ `\ R$
and *step-rtransymclp-models*: $\mathcal{T}(F,V){:}Fun{:}(\leftarrow F{:}V{:}R{\to}^*) \models unconditional\ `\ R$
⟨*proof*⟩

Moreover, root steps are valid in any models.

context *sorted-algebra* **begin**

context
fixes *less-eq* (**infix** $\sqsubseteq$ *50*) **and** R
assumes *models*: $A{:}I{:}(\sqsubseteq) \models unconditional\ `\ R$
begin

lemma *rootstepp-imp-valid*:
assumes *st*: $s\ {-}F{:}V{:}R{\to}^{\varepsilon}\ t$
shows $A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *rootstepp-le-valid*: $({-}F{:}V{:}R{\to}^{\varepsilon}) \leq valid\ A\ I\ (\sqsubseteq)\ V$
⟨*proof*⟩

end

end

Therefore, a root step implies validity in the term algebra.

lemmas *rootstepp-imp-valid-term* = *term.rootstepp-imp-valid*[*OF rootstepp-models*]

Any relation closed under substitutions contains the root steps if it models the TRS.

lemma (**in** *subst-closed*) *rootstepp-le*:
$\mathcal{T}(F,V)$:*Fun*:$(\preceq) \models$ *unconditional* ‘ $R \Longrightarrow (-F{:}V{:}R{\to}^{\varepsilon}) \leq (\preceq)$
⟨*proof*⟩

Therefore, the root steps is the least term model which is closed under substitution.

theorem *rootstepp-eq-least*:
$(-F{:}V{:}R{\to}^{\varepsilon}) = (LEAST\ r.$ *subst-closed* $F\ V\ r \wedge (\mathcal{T}(F,V)$:*Fun*:$r \models$ *unconditional* ‘ $R))$
⟨*proof*⟩

The rewrite steps are valid in all monotone models.

context *monotone-algebra* **begin**

context fixes R
assumes *models*: $A{:}I{:}(\sqsubseteq) \models$ *unconditional* ‘ R
begin

interpretation *valid*: *monotone-algebra* $F\ \mathcal{T}(F,V)$ *Fun valid* $A\ I\ (\sqsubseteq)\ V$⟨*proof*⟩

lemma *stepp-imp-valid*: $s\ -F{:}V{:}R{\to}\ t \Longrightarrow A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *stepp-le-valid*: $(-F{:}V{:}R{\to}) \leq$ *valid* $A\ I\ (\sqsubseteq)\ V$
⟨*proof*⟩

end

end

In particular, any rewrite relation that models a TRS contains the rewrite step.

lemma (**in** *rewrite-relation*) *stepp-le*:
$\mathcal{T}(F,V)$:*Fun*:$(\preceq) \models$ *unconditional* ‘ $R \Longrightarrow (-F{:}V{:}R{\to}) \leq (\preceq)$
⟨*proof*⟩

Therefore, the rewrite step is the least rewrite relation that models the TRS.

theorem *stepp-eq-least*:

$(-F{:}V{:}R{\to}) = (LEAST\ r.\ rewrite\text{-}relation\ F\ V\ r \wedge (\mathcal{T}(F,V){:}Fun{:}r \models unconditional\ `\ R))$
⟨*proof*⟩

Similar results hold for rewrite preorders.

context *reflexive-monotone-algebra* **begin**

context fixes R
assumes *models*: $A{:}I{:}(\sqsubseteq) \models unconditional\ `\ R$
begin

lemma *step-reflclp-imp-valid*:
$s\ -F{:}V{:}R{\to}^{=}\ t \Longrightarrow A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *step-reflclp-le-valid*:
$(-F{:}V{:}R{\to}^{=}) \leq valid\ A\ I\ (\sqsubseteq)\ V$
⟨*proof*⟩

end

end

For transitivity, rewrite rule must relate a sorted term to a sorted term. Sorts need not be the same but for simplicity we assume *sorted-trs* assumption.

context *transitive-monotone-algebra* **begin**

context fixes R
assumes R: *sorted-trs* $F\ R$ **and** *models*: $A{:}I{:}(\sqsubseteq) \models unconditional\ `\ R$
begin

interpretation R: *sorted-trs* ⟨*proof*⟩

lemma *step-tranclp-imp-valid*:
assumes *st*: $s\ -F{:}V{:}R{\to}^{+}\ t$
shows $A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *step-tranclp-le-valid*:
$(-F{:}V{:}R{\to})^{++} \leq valid\ A\ I\ (\sqsubseteq)\ V$
⟨*proof*⟩

end

end

context *quasi-ordered-monotone-algebra* **begin**

context fixes R

assumes *R*: *sorted-trs F R* **and** *models*: *A*:*I*:(⊑) ⊨ *unconditional* ‘ *R*
begin

lemma *step-rtranclp-imp-valid*:
s −*F*:*V*:*R*→* *t* ⟹ *A*:*I*:(⊑) ⊨ *V*. *s* ⇝ *t*
⟨*proof*⟩

lemma *step-rtranclp-le-valid*: (−*F*:*V*:*R*→*) ≤ *valid A I* (⊑) *V*
⟨*proof*⟩

end

end

lemma (**in** *rewrite-preorder*) *step-rtranclp-le*:
sorted-trs F R ⟹ $\mathcal{T}$(*F*,*V*):*Fun*:(≼) ⊨ *unconditional* ‘ *R* ⟹ (−*F*:*V*:*R*→*) ≤ (≼)
⟨*proof*⟩

Finally, the rewrite relation is the least rewrite preorder.

theorem (**in** *sorted-trs*) *step-rtranclp-eq-least*:
(−*F*:*V*:*R*→*) = (*LEAST r*. *rewrite-preorder F V r* ∧ ($\mathcal{T}$(*F*,*V*):*Fun*:*r* ⊨ *unconditional* ‘ *R*))
⟨*proof*⟩

end

11 Conditional Rewriting

theory *Conditional-Rewriting*
imports *Sorted-Rewrite-Relations*
begin

Here we define sorted conditional rewriting. As conditional rewrite rules we just use inference rules.

It is more convenient to define the full rewrite step inductively, as satisfaction of conditions involves full rewrite steps. Afterwards we define the root rewrite step as a subset of the full rewrite step.

inductive *cstepp* (′(=-:-:-⇒′) [*51*,*51*,*51*]*1000*) **for** *F V R* **where**
root: (=*F*:*V*:*R*⇒) (*l*·ϑ) (*r*·ϑ)
if (*X*. *l* ⇝ *r* ⇐ *cs*) ∈ *R* **and** ϑ :$_s$ *X* → $\mathcal{T}$(*F*,*V*)
and ∀(*s* ⇝ *t*) ∈ *set cs*. (=*F*:*V*:*R*⇒)** (*s*·ϑ) (*t*·ϑ)
for *X l r cs* ϑ
| *comp*: (=*F*:*V*:*R*⇒) (*Fun f* (*ls* @ *s* # *rs*)) (*Fun f* (*ls* @ *t* # *rs*))
if *f* : πs@σ#ϱs → τ *in F*
and *ls* :$_l$ πs *in* $\mathcal{T}$(*F*,*V*)
and *s* : σ *in* $\mathcal{T}$(*F*,*V*)
and *t* : σ *in* $\mathcal{T}$(*F*,*V*)

and $rs :_l \varrho s$ *in* $\mathcal{T}(F,V)$
and $(=F{:}V{:}R\Rightarrow)\ s\ t$
for $f\ \pi s\ \sigma\ \varrho s\ \tau\ ls\ rs\ s\ t$

hide-fact(**open**) *cstepp.root cstepp.comp*

abbreviation *cstepp-op* $(((2\text{-})/\ =\text{-}:\text{-}:\text{-}\Rightarrow\ /(2\text{-}))\ [51,51,51,51,51]50)$ **where**
$s\ =F{:}V{:}R\Rightarrow\ t \equiv (=F{:}V{:}R\Rightarrow)\ s\ t$

abbreviation *cstep* $(\{=\text{-}:\text{-}:\text{-}\Rightarrow\}[51,51,51]1000)$ **where**
$\{=F{:}V{:}R\Rightarrow\} \equiv \{(s,t).\ s\ =F{:}V{:}R\Rightarrow\ t\}$

definition *cstep-reflclp* $('(=\text{-}:\text{-}:\text{-}\Rightarrow^{=}{}')\ [51,51,51]1000)$ **where**
$(=F{:}V{:}R\Rightarrow^{=}) \equiv$ *reflclp-on* $(dom\ \mathcal{T}(F,V))\ (=F{:}V{:}R\Rightarrow)$

abbreviation *cstepp-reflcl-op* $(((2\text{-})/\ =\text{-}:\text{-}:\text{-}\Rightarrow^{=}\ /(2\text{-}))\ [51,51,51,51,51]50)$ **where**
$s\ =F{:}V{:}R\Rightarrow^{=}\ t \equiv (=F{:}V{:}R\Rightarrow^{=})\ s\ t$

abbreviation *cstep-reflcl* $(\{=\text{-}:\text{-}:\text{-}\Rightarrow^{=}\}\ [51,51,51]1000)$
where $\{=F{:}V{:}R\Rightarrow^{=}\} \equiv \{(s,t).\ s\ =F{:}V{:}R\Rightarrow^{=}\ t\}$

abbreviation (*input*) *dual-cstepp* $('(\Leftarrow\text{-}:\text{-}:\text{-}=')\ [51,51,51]1000)$
where $(\Leftarrow F{:}V{:}R=) \equiv (=F{:}V{:}R\Rightarrow)^{-}$

abbreviation *cstep-tranclp* $('(=\text{-}:\text{-}:\text{-}\Rightarrow^{+}{}')\ [51,51,51]1000)$ **where**
$(=F{:}V{:}R\Rightarrow^{+}) \equiv$ *tranclp* $(=F{:}V{:}R\Rightarrow)$

abbreviation *cstep-trancl-op* $(((2\text{-})/\ =\text{-}:\text{-}:\text{-}\Rightarrow^{+}\ /(2\text{-}))\ [51,51,51,51,51]50)$
where $s\ =F{:}V{:}R\Rightarrow^{+}\ t \equiv (=F{:}V{:}R\Rightarrow^{+})\ s\ t$

abbreviation *cstep-rtranclp* $('(=\text{-}:\text{-}:\text{-}\Rightarrow^{*}{}')\ [51,51,51]1000)$ **where**
$(=F{:}V{:}R\Rightarrow^{*}) \equiv$ *tranclp* $(=F{:}V{:}R\Rightarrow^{=})$

abbreviation *cstep-rtrancl-op* $(((2\text{-})/\ =\text{-}:\text{-}:\text{-}\Rightarrow^{*}\ /(2\text{-}))\ [51,51,51,51,51]50)$
where $s\ =F{:}V{:}R\Rightarrow^{*}\ t \equiv (=F{:}V{:}R\Rightarrow^{*})\ s\ t$

abbreviation *cstep-rtrancl* $(\{=\text{-}:\text{-}:\text{-}\Rightarrow^{*}\}\ [51,51,51]1000)$
where $\{=F{:}V{:}R\Rightarrow^{*}\} \equiv \{(s,t).\ s\ =F{:}V{:}R\Rightarrow^{*}\ t\}$

abbreviation *cstep-symclp* $('(\Leftarrow\text{-}:\text{-}:\text{-}\Rightarrow')\ [51,51,51]1000)$
where $(\Leftarrow F{:}V{:}R\Rightarrow) \equiv$ *symclp* $(=F{:}V{:}R\Rightarrow)$

abbreviation *cstep-symcl-op* $(((2\text{-})/\ \Leftarrow\text{-}:\text{-}:\text{-}\Rightarrow\ /(2\text{-}))\ [51,51,51,51,51]50)$
where $s \Leftarrow F{:}V{:}R\Rightarrow\ t \equiv (\Leftarrow F{:}V{:}R\Rightarrow)\ s\ t$

abbreviation *cstep-symcl* $(\{\Leftarrow\text{-}:\text{-}:\text{-}\Rightarrow\}\ [51,51,51]1000)$
where $\{\Leftarrow F{:}V{:}R\Rightarrow\} \equiv \{(s,t).\ s \Leftarrow F{:}V{:}R\Rightarrow\ t\}$

abbreviation *cstep-rsymclp* $('(\Leftarrow\text{-}:\text{-}:\text{-}\Rightarrow^{=}{}')\ [51,51,51]1000)$ **where**

$(\Leftarrow F{:}\,V{:}R\Rightarrow^{=}) \equiv \mathit{symclp}\ (=F{:}\,V{:}R\Rightarrow^{=})$

abbreviation *cstep-rsymcl-op* $(((2\text{-})/\ \Leftarrow\text{-}{:}\text{-}{:}\text{-}\Rightarrow^{=}\ /(2\text{-}))\ [51,51,51,51,51]50)$ **where**
$s \Leftarrow F{:}\,V{:}R\Rightarrow^{=} t \equiv (\Leftarrow F{:}\,V{:}R\Rightarrow^{=})\ s\ t$

abbreviation *cstep-rsymcl* $(\{\Leftarrow\text{-}{:}\text{-}{:}\text{-}\Rightarrow^{=}\}\ [51,51,51]1000)$
where $\{\Leftarrow F{:}\,V{:}R\Rightarrow^{=}\} \equiv \{(s,t).\ s \Leftarrow F{:}\,V{:}R\Rightarrow^{=} t\}$

abbreviation *cstep-rtransymclp* $('(\Leftarrow\text{-}{:}\text{-}{:}\text{-}\Rightarrow^{*}')\ [51,51,51]1000)$
where $(\Leftarrow F{:}\,V{:}R\Rightarrow^{*}) \equiv (\Leftarrow F{:}\,V{:}R\Rightarrow^{=})^{++}$

abbreviation *cstep-rtransymcl-op* $(((2\text{-})/\ \Leftarrow\text{-}{:}\text{-}{:}\text{-}\Rightarrow^{*}\ /(2\text{-}))\ [51,51,51,51,51]50)$
where $s \Leftarrow F{:}\,V{:}R\Rightarrow^{*} t \equiv (\Leftarrow F{:}\,V{:}R\Rightarrow^{*})\ s\ t$

interpretation *ars* $(=F{:}\,V{:}R\Rightarrow)$ **for** $F\ V\ R\langle \mathit{proof}\rangle$

lemma *cstepp-induct*[*consumes 1*, *case-names root comp*]:
fixes P (**infix** $\sqsubseteq$ *50*)
assumes *st*: $s =F{:}\,V{:}R\Rightarrow t$
and *root*: $\bigwedge X\ l\ r\ cs\ \vartheta.$
$(X.\ l \rightsquigarrow r \Leftarrow cs) \in R \Longrightarrow$
$\vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow$
$\forall s\ t.\ (s \rightsquigarrow t) \in \mathit{set}\ cs \longrightarrow ((=F{:}\,V{:}R\Rightarrow) \sqcap (\sqsubseteq))^{**}\ (s \cdot \vartheta)\ (t \cdot \vartheta) \Longrightarrow$
$l \cdot \vartheta \sqsubseteq r \cdot \vartheta$
and *comp*: $\bigwedge f\ \pi s\ \sigma\ \varrho s\ \tau\ ls\ rs\ s\ t.$
$f : \pi s\ @\ \sigma\ \#\ \varrho s \to \tau\ \mathit{in}\ F \Longrightarrow$
$ls :_l \pi s\ \mathit{in}\ \mathcal{T}(F,V) \Longrightarrow$
$s : \sigma\ \mathit{in}\ \mathcal{T}(F,V) \Longrightarrow$
$t : \sigma\ \mathit{in}\ \mathcal{T}(F,V) \Longrightarrow$
$rs :_l \varrho s\ \mathit{in}\ \mathcal{T}(F,V) \Longrightarrow$
$s =F{:}\,V{:}R\Rightarrow t \Longrightarrow$
$s \sqsubseteq t \Longrightarrow$
$\mathit{Fun}\ f\ (ls\ @\ s\ \#\ rs) \sqsubseteq \mathit{Fun}\ f\ (ls\ @\ t\ \#\ rs)$
shows $s \sqsubseteq t$
$\langle \mathit{proof}\rangle$

Conditional rewrite step is a rewrite relation.

interpretation *cstepp*: *subst-closed-general* $F\ V\ W\ (=F{:}\,V{:}R\Rightarrow)\ (=F{:}\,W{:}R\Rightarrow)$
rewrites $\bigwedge X.$ *reflclp-on* $(\mathit{dom}\ \mathcal{T}(F,X))\ (=F{:}X{:}R\Rightarrow) \equiv (=F{:}X{:}R\Rightarrow^{=})$
$\langle \mathit{proof}\rangle$
thm *cstepp.rtranclp-on.stable*

interpretation *cstepp*: *rewrite-relation* $F\ V\ (=F{:}\,V{:}R\Rightarrow)$
rewrites $\bigwedge X.$ *reflclp-on* $(\mathit{dom}\ \mathcal{T}(F,X))\ (=F{:}X{:}R\Rightarrow) \equiv (=F{:}X{:}R\Rightarrow^{=})$
$\langle \mathit{proof}\rangle$

thm *cstepp.reflclp-on.ctxt-closed*

lemma *cstep-reflclpE*:

assumes $s =F: V: R\Rightarrow^{=} t$ **and** $\bigwedge\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow s = t \Longrightarrow thesis$
and $s =F: V: R\Rightarrow t \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

interpretation *cstep-reflclp*: *reflexive dom* $\mathcal{T}(F,V)$ $(=F: V: R\Rightarrow^{=})$
⟨*proof*⟩

interpretation *cstep-rsymclp*: *reflexive dom* $\mathcal{T}(F,V)$ $(\Leftarrow F: V: R\Rightarrow^{=})$
⟨*proof*⟩

lemma *cstep-rtranclp-iff*: $s =F: V: R\Rightarrow^{*} t \longleftrightarrow (s = t \wedge s \in dom\ \mathcal{T}(F,V)) \vee s =F: V: R\Rightarrow^{+} t$
⟨*proof*⟩

lemma [*rewriting-simps*]:
shows *cstep-rtranclp-refl*: $s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow s =F: V: R\Rightarrow^{*} s$
and *cstep-rtranclp-trancl*: $s =F: V: R\Rightarrow^{+} t \Longrightarrow s =F: V: R\Rightarrow^{*} t$
and *cstep-rtranclp-step*: $s =F: V: R\Rightarrow t \Longrightarrow s =F: V: R\Rightarrow^{*} t$
⟨*proof*⟩

lemma *cstep-rtranclp-eq-reflclp*:
$(=F: V: R\Rightarrow^{*}) = reflclp\text{-}on\ (dom\ \mathcal{T}(F,V))\ (=F: V: R\Rightarrow)^{++}$
⟨*proof*⟩

lemma *cstep-rtranclpE*[*consumes 1*, *case-names refl trancl*]:
assumes $s =F: V: R\Rightarrow^{*} t$
and $\bigwedge\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow s = t \Longrightarrow thesis$
and $s =F: V: R\Rightarrow^{+} t \Longrightarrow thesis$
shows *thesis* ⟨*proof*⟩

Now we define the conditional root rewrite step.

definition *crootstepp* (′(=-:-:-⇒$^{\varepsilon}$′) [*51*,*51*,*51*]*1000*)
where $(=F: V: R\Rightarrow^{\varepsilon})\ s\ t \equiv$
$\exists(X.\ l \rightsquigarrow r \Leftarrow cs) \in R.\ \exists\vartheta :_s X \to \mathcal{T}(F,V).\ (\forall(s \rightsquigarrow t) \in set\ cs.\ s{\cdot}\vartheta =F: V: R\Rightarrow^{*} t{\cdot}\vartheta) \wedge$
$s = l{\cdot}\vartheta \wedge t = r{\cdot}\vartheta$

abbreviation *crootstep-op* (((*2*-)/ =-:-:-⇒$^{\varepsilon}$ /(*2*-)) [*51*,*51*,*51*,*51*,*51*]*50*)
where $s =F: V: R\Rightarrow^{\varepsilon} t \equiv (=F: V: R\Rightarrow^{\varepsilon})\ s\ t$

lemma *crootsteppI*:
assumes $(X.\ l \rightsquigarrow r \Leftarrow cs) \in R$ **and** $\vartheta :_s X \to \mathcal{T}(F,V)$
and $\bigwedge u\ v.\ (u \rightsquigarrow v) \in set\ cs \Longrightarrow u{\cdot}\vartheta =F: V: R\Rightarrow^{*} v{\cdot}\vartheta$
and $s = l{\cdot}\vartheta$ **and** $t = r{\cdot}\vartheta$
shows $s =F: V: R\Rightarrow^{\varepsilon} t$
⟨*proof*⟩

lemma *crootstepp-rule*:

assumes $(X.\ l \rightsquigarrow r \Leftarrow cs) \in R$ **and** $\vartheta :_s X \to \mathcal{T}(F,V)$
and $\bigwedge u\ v.\ (u \rightsquigarrow v) \in set\ cs \Longrightarrow u{\cdot}\vartheta =F{:}V{:}R{\Rightarrow}^*\ v{\cdot}\vartheta$
shows $l{\cdot}\vartheta =F{:}V{:}R{\Rightarrow}^{\varepsilon}\ r{\cdot}\vartheta$
⟨*proof*⟩

lemma *crootsteppE*:
assumes $s =F{:}V{:}R{\Rightarrow}^{\varepsilon}\ t$
and $\bigwedge X\ l\ r\ cs\ \vartheta.\ (X.\ l \rightsquigarrow r \Leftarrow cs) \in R \Longrightarrow \vartheta :_s X \to \mathcal{T}(F,V) \Longrightarrow$
$\forall\, u\ v.\ (u \rightsquigarrow v) \in set\ cs \longrightarrow u{\cdot}\vartheta =F{:}V{:}R{\Rightarrow}^*\ v{\cdot}\vartheta \Longrightarrow s = l{\cdot}\vartheta \Longrightarrow t = r{\cdot}\vartheta \Longrightarrow$
thesis
shows *thesis*
⟨*proof*⟩

lemma *cstepp-root*: **assumes** $s =F{:}V{:}R{\Rightarrow}^{\varepsilon}\ t$ **shows** $s =F{:}V{:}R{\Rightarrow}\ t$
⟨*proof*⟩

interpretation *crootstepp*: *subst-closed-general F V W* $(=F{:}V{:}R{\Rightarrow}^{\varepsilon})$ $(=F{:}W{:}R{\Rightarrow}^{\varepsilon})$
⟨*proof*⟩

thm *crootstepp.symclp.stable*

The conditional reduction is a model of the CTRS.

lemma *cstep-rtranclp-models*: $\mathcal{T}(F,V)$:*Fun*:$(=F{:}V{:}R{\Rightarrow}^*) \models R$
⟨*proof*⟩

locale *sorted-ctrs* =
fixes $F :: ('f,'s)$ *ssig* **and** $R :: ('f,'x,'s)$ *rule set*
assumes *sort-safe*: *sort-safe-rules F R*
begin

lemma *cstepp-typed*: $s =F{:}V{:}R{\Rightarrow}\ t \Longrightarrow \exists\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

sublocale *cstepp*: *sorted-rewrite-relation F V* $(=F{:}V{:}R{\Rightarrow})$
rewrites $\bigwedge X.$ *reflclp-on* (*dom* $\mathcal{T}(F,X)$) $(=F{:}X{:}R{\Rightarrow}) \equiv (=F{:}X{:}R{\Rightarrow}^{=})$
⟨*proof*⟩

sublocale *cstep-reflclp*: *sorted-relation* $\mathcal{T}(F,V)$ $(=F{:}V{:}R{\Rightarrow}^{=})$
⟨*proof*⟩

sublocale *cstep-tranclp*: *sorted-relation* $\mathcal{T}(F,V)$ $(=F{:}V{:}R{\Rightarrow})^{++}$
⟨*proof*⟩

sublocale *cstep-rtranclp*: *sorted-relation* $\mathcal{T}(F,V)$ $(=F{:}V{:}R{\Rightarrow}^*)$
⟨*proof*⟩

sublocale *cstep-symclp*: *sorted-relation* $\mathcal{T}(F,V)$ $(\Leftarrow F{:}V{:}R{\Rightarrow})$
⟨*proof*⟩

sublocale *cstep-rsymclp*: *sorted-relation* $\mathcal{T}(F,V)$ $(\Leftarrow F{:}V{:}R{\Rightarrow}^{=})$
⟨*proof*⟩

sublocale *cstep-rtransymclp*: *sorted-relation* $\mathcal{T}(F,V)$ $(\Leftarrow F{:}V{:}R{\Rightarrow}^{*})$
⟨*proof*⟩

thm *cstepp.rtranclp-on.ctxt-closed*

lemma *cstep-symclp-typed*: $s \Leftarrow F{:}V{:}R{\Rightarrow}\ t \Longrightarrow \exists\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *cstep-tranclp-typed*: $s = F{:}V{:}R{\Rightarrow}^{+}\ t \Longrightarrow \exists\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *cstep-rtransymclp-typed*: $s \Leftarrow F{:}V{:}R{\Rightarrow}^{*}\ t \Longrightarrow \exists\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \wedge t : \sigma\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma *cstep-rtranclp-induct*[*consumes 1*, *case-names refl step*]:
assumes *st*: $s = F{:}V{:}R{\Rightarrow}^{*}\ t$
and *Refl*: $\bigwedge\sigma.\ s : \sigma\ in\ \mathcal{T}(F,V) \Longrightarrow P\ s$
and *Step*: $\bigwedge t\ u.\ s = F{:}V{:}R{\Rightarrow}^{*}\ t \Longrightarrow t = F{:}V{:}R{\Rightarrow}\ u \Longrightarrow P\ t \Longrightarrow P\ u$
shows $P\ t$
⟨*proof*⟩

end

interpretation *cstep-reflclp*: *reflexive-algebra* $F\ \mathcal{T}(F,V)\ Fun\ (=F{:}V{:}R{\Rightarrow}^{=})$⟨*proof*⟩

interpretation *cstep-tranclp*: *transitive-algebra* $F\ \mathcal{T}(F,V)\ Fun\ (=F{:}V{:}R{\Rightarrow})^{++}$
⟨*proof*⟩

interpretation *cstep-rtranclp*: *quasi-ordered-algebra* $F\ \mathcal{T}(F,V)\ Fun\ (=F{:}V{:}R{\Rightarrow}^{*})$
⟨*proof*⟩

interpretation *cstep-reflclp*: *reflexive-monotone-algebra* $F\ \mathcal{T}(F,V)\ Fun\ (=F{:}V{:}R{\Rightarrow}^{=})$
⟨*proof*⟩

sublocale *sorted-ctrs* $\subseteq$ *cstep-tranclp*: *transitive-monotone-algebra* $F\ \mathcal{T}(F,V)\ Fun\ (=F{:}V{:}R{\Rightarrow})^{++}$⟨*proof*⟩

Because conditions are evaluated by full many-step reductions, models of CTRS make sense only if they are quasi-ordered monotone.

context *quasi-ordered-monotone-algebra* **begin**

context fixes R

assumes *ctrs*: *sorted-ctrs F R* **and** *models*: $A{:}I{:}(\sqsubseteq) \models R$
begin

interpretation *sorted-ctrs F R* ⟨*proof*⟩

lemma *cstepp-imp-valid*: **assumes** *st*: $s =F{:}V{:}R\Rightarrow t$ **shows** $A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *cstepp-le-valid*: $(=F{:}V{:}R\Rightarrow) \leq$ *valid A I* $(\sqsubseteq)$ *V* ⟨*proof*⟩

lemma *cstep-tranclp-imp-valid*: $s =F{:}V{:}R\Rightarrow^+ t \Longrightarrow A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *cstep-tranclp-le-valid*: $(=F{:}V{:}R\Rightarrow^+) \leq$ *valid A I* $(\sqsubseteq)$ *V*
⟨*proof*⟩

lemma *cstep-rtranclp-imp-valid*: $s =F{:}V{:}R\Rightarrow^* t \Longrightarrow A{:}I{:}(\sqsubseteq) \models V.\ s \rightsquigarrow t$
⟨*proof*⟩

lemma *cstep-rtranclp-le-valid*: $(=F{:}V{:}R\Rightarrow^*) \leq$ *valid A I* $(\sqsubseteq)$ *V*
⟨*proof*⟩

end

end

lemma (**in** *rewrite-preorder*) *cstep-rtranclp-le*:
$\mathit{sorted\text{-}ctrs}\ F\ R \Longrightarrow \mathcal{T}(F,V){:}\mathit{Fun}{:}(\preceq) \models R \Longrightarrow (=F{:}V{:}R\Rightarrow^*) \leq (\preceq)$
⟨*proof*⟩

The many-step conditional rewriting is the least rewrite preorder.

lemma (**in** *sorted-ctrs*) *cstep-rtraclp-eq-least*:
$(=F{:}V{:}R\Rightarrow^*) = (\mathit{LEAST}\ r.\ \mathit{rewrite\text{-}preorder}\ F\ V\ r \wedge (\mathcal{T}(F,V){:}\mathit{Fun}{:}r \models R))$
⟨*proof*⟩

end
theory *Variadic-Signature*
imports *Sorted-Terms.Sorted-Terms*
begin

This theory introduces a convenient notation for variadic signatures.

definition *variadic-upd F f* σ τ *r* ≡
if fst r $= f \wedge (\forall \sigma' \in \mathit{set}\ (\mathit{snd}\ r).\ \sigma' = \sigma)$ *then Some* τ *else F r*

syntax
-variadic-upd :: *'a* ⇒ *'a* ⇒ *'a* ⇒ *maplet* (*'*(-,/ -..*'*) /↦/ -)

translations
-Update F (*-variadic-upd f* σ τ) ⇌ *CONST variadic-upd F f* σ τ

lemma *hastype-in-variadic*:
$f : \sigma s \to \tau$ *in* $F((f',\sigma..) \mapsto \tau') \longleftrightarrow$
(*if* $f = f' \wedge (\forall \sigma' \in$ *set* $\sigma s.\ \sigma' = \sigma)$ *then* $\tau = \tau'$ *else* $f : \sigma s \to \tau$ *in* F)
⟨*proof*⟩

lemma *dom-variadic-upd*[*simp*]: *dom* $(F((f,\sigma..)\mapsto\tau)) =$ *dom* $F \cup \{f\} \times$ *lists* $\{\sigma\}$
⟨*proof*⟩

end

12 Logic

We define propositional logic as a special case of equational logic, where special sort "bool" and logical operators exist.

theory *Logic*
imports *Models Variadic-Signature Sorted-Terms-More*
begin

lemma *all-set-iff-replicate*: $(\forall x \in$ *set* $xs.\ x = y) \longleftrightarrow (\exists n.\ xs =$ *replicate* $n\ y)$
⟨*proof*⟩

lemma *all-set-image-conv-ex-map*:
$(\forall fa \in$ *set* $fas.\ fa \in f$ ' $A) \longleftrightarrow (\exists as \in$ *lists* $A.\ fas =$ *map* $f\ as)$
⟨*proof*⟩

lemma (**in** *subsignature*) *has-same-type-subsig*: $f : \sigma s \to \tau$ *in* $F \Longrightarrow f : \sigma s \to \tau'$ *in* $G \longleftrightarrow \tau = \tau'$
⟨*proof*⟩

lemma *all-set-hastype*:
assumes $\forall a \in$ *set as.* $a : \sigma$ *in* A
shows *as* $:_l \sigma s$ *in* $A \longleftrightarrow \sigma s =$ *replicate* (*length as*) σ
⟨*proof*⟩

12.1 Syntax

The following locales introduce notations for propositional logic.

locale *truth-syntax* = **fixes** *conjF* :: *'f*
begin

abbreviation *trueT* ≡ *Fun conjF* []

abbreviation *satisfies-formula* ((-:-;- ⊨ -) [*51*,*51*,*51*,*25*]*4*) **where**
$I{:}r;\alpha \models \varphi \equiv I{:}r;\alpha \models \varphi \rightsquigarrow$ *trueT*

abbreviation *satisfies-formula-eq* ((-;- ⊨ -) [*51*,*51*,*25*]*4*) **where**

$I;\alpha \models \varphi \equiv I{:}(=);\alpha \models \varphi$

lemma *satisfies-formula-subst*: $(I{:}r;\alpha \models \varphi{\cdot}\vartheta) \longleftrightarrow (I{:}r;I[\![\vartheta]\!]_s\ \alpha \models \varphi)$
⟨*proof*⟩

lemma *satisfies-formula-same-vars*:
assumes $\forall\, x \in$ *vars-term* $\varphi.\ \alpha\ x = \beta\ x$
shows $(I{:}r;\alpha \models \varphi) \longleftrightarrow (I{:}r;\beta \models \varphi)$
⟨*proof*⟩

abbreviation *valid-formula* ((-:-:- ⊨ -. -) [*51*,*51*,*51*,*100*,*25*]*4*) **where**
$A{:}I{:}r \models X.\ \varphi \equiv A{:}I{:}r \models X.\ \varphi \rightsquigarrow \mathit{trueT}$

abbreviation *valid-formula-eq* ((-:- ⊨ -. -) [*51*,*51*,*100*,*25*]*4*) **where**
$A{:}I \models X.\ \varphi \equiv A{:}I{:}(=) \models X.\ \varphi$

end

locale *logic-symbols-syntax* = **fixes** *conjF disjF negF* :: *'f*
begin

sublocale *truth-syntax*⟨*proof*⟩

abbreviation $\mathit{falseT} \equiv \mathit{Fun}\ \mathit{disjF}\ []$

abbreviation *andT* (**infixl** $\wedge_t$ *35*) **where** $s \wedge_t t \equiv \mathit{Fun}\ \mathit{conjF}\ [s,t]$

abbreviation *orT* (**infixr** $\vee_t$ *30*) **where** $s \vee_t t \equiv \mathit{Fun}\ \mathit{disjF}\ [s,t]$

abbreviation *notT* ($\neg_t$ - [*40*]*40*) **where** $\neg_t\ s \equiv \mathit{Fun}\ \mathit{negF}\ [s]$

definition *impT* (**infixr** $\longrightarrow_t$ *25*) **where** $s \longrightarrow_t t \equiv \neg_t\ s \vee_t t$

abbreviation $\mathit{list\text{-}exT}\ f\ ss \equiv \mathit{Fun}\ \mathit{disjF}\ (\mathit{map}\ f\ ss)$

abbreviation $\mathit{list\text{-}allT}\ f\ ss \equiv \mathit{Fun}\ \mathit{conjF}\ (\mathit{map}\ f\ ss)$

lemma *logic-cases*[*case-names Not Conj Disj Other*]:
assumes $f = \mathit{negF} \Longrightarrow \mathit{thesis}$
and $f \neq \mathit{negF} \Longrightarrow f = \mathit{conjF} \Longrightarrow \mathit{thesis}$
and $f \neq \mathit{negF} \Longrightarrow f \neq \mathit{conjF} \Longrightarrow f = \mathit{disjF} \Longrightarrow \mathit{thesis}$
and $f \neq \mathit{negF} \Longrightarrow f \neq \mathit{conjF} \Longrightarrow f \neq \mathit{disjF} \Longrightarrow \mathit{thesis}$
shows *thesis*
⟨*proof*⟩

abbreviation $\mathit{const\text{-}of\text{-}bool}\ b \equiv \mathit{if}\ b\ \mathit{then}\ \mathit{trueT}\ \mathit{else}\ \mathit{falseT}$

end

The following locale defines the signature of logic symbols.

locale *logic-signature-syntax* =
  fixes *boolS* :: *'s* **and** *conjF disjF negF* :: *'f*
begin

interpretation *logic-symbols-syntax*⟨*proof*⟩

We define the (constructor) signature of logic symbols.

definition *C* **where**
  $C \equiv [(conjF,[]) \mapsto boolS,\ (disjF,[]) \mapsto boolS]$

lemma *hastype-in-C*: $f : \sigma s \to \tau\ in\ C \longleftrightarrow$
  $(f = conjF \vee f = disjF) \wedge \sigma s = [] \wedge \tau = boolS$
  ⟨*proof*⟩

lemma *hastype-in-C-E*[*elim*!]:
  assumes $f : \sigma s \to \tau\ in\ C$
    and $f = conjF \Longrightarrow \sigma s = [] \Longrightarrow \tau = boolS \Longrightarrow thesis$
    and $f = disjF \Longrightarrow \sigma s = [] \Longrightarrow \tau = boolS \Longrightarrow thesis$
  shows *thesis*
  ⟨*proof*⟩

lemma *conj-hastype-in-C-iff*[*simp*]: $conjF : \sigma s \to \tau\ in\ C \longleftrightarrow \sigma s = [] \wedge \tau = boolS$
  and *disj-hastype-in-C-iff*[*simp*]: $disjF : \sigma s \to \tau\ in\ C \longleftrightarrow \sigma s = [] \wedge \tau = boolS$
  ⟨*proof*⟩

lemma *dom-C*[*simp*]: $dom\ C = \{(conjF,[]),(disjF,[])\}$
  ⟨*proof*⟩

definition *F* **where**
  $F \equiv [(negF,[boolS]) \mapsto boolS,\ (conjF,boolS..) \mapsto boolS,\ (disjF,boolS..) \mapsto boolS\ ]$

lemma *dom-F*: $dom\ F = \{(negF,[boolS])\} \cup \{conjF,disjF\} \times lists\ \{boolS\}$
  ⟨*proof*⟩

lemma *hastype-in-F*: $f : \sigma s \to \tau\ in\ F \longleftrightarrow$
  $\tau = boolS \wedge ($
    $(f = conjF \vee f = disjF) \wedge (\forall \sigma \in set\ \sigma s.\ \sigma = boolS) \vee$
    $f = negF \wedge \sigma s = [boolS])$
  ⟨*proof*⟩

lemma *hastype-in-F-E*:
  assumes $f : \sigma s \to \tau\ in\ F$
    and $\bigwedge n.\ \tau = boolS \Longrightarrow f = conjF \Longrightarrow \sigma s = replicate\ n\ boolS \Longrightarrow thesis$
    and $\bigwedge n.\ \tau = boolS \Longrightarrow f = disjF \Longrightarrow \sigma s = replicate\ n\ boolS \Longrightarrow thesis$
    and $f = negF \Longrightarrow \tau = boolS \Longrightarrow \sigma s = [boolS] \Longrightarrow thesis$
  shows *thesis*
  ⟨*proof*⟩

lemma *conj-hastype-iff*[*simp*]: $conjF : \sigma s \to \tau\ in\ F \longleftrightarrow (\forall \sigma \in set\ \sigma s.\ \sigma = boolS)$

$\wedge\ \tau = boolS$
and *disj-hastype-iff*[*simp*]: $disjF : \sigma s \to \tau\ in\ F \longleftrightarrow (\forall \sigma \in set\ \sigma s.\ \sigma = boolS) \wedge \tau = boolS$
⟨*proof*⟩

lemma *Fun-conj-hastype-iff*: $Fun\ conjF\ \varphi s : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow (\forall \varphi \in set\ \varphi s.\ \varphi : boolS\ in\ \mathcal{T}(F,V)) \wedge \tau = boolS$
and *Fun-disj-hastype-iff*: $Fun\ disjF\ \varphi s : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow (\forall \varphi \in set\ \varphi s.\ \varphi : boolS\ in\ \mathcal{T}(F,V)) \wedge \tau = boolS$
⟨*proof*⟩

lemma *and-hastype-iff*[*simp*]: $(\varphi \wedge_t \psi) : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow \varphi : boolS\ in\ \mathcal{T}(F,V) \wedge \psi : boolS\ in\ \mathcal{T}(F,V) \wedge \tau = boolS$
and *or-hastype-iff*[*simp*]: $(\varphi \vee_t \psi) : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow \varphi : boolS\ in\ \mathcal{T}(F,V) \wedge \psi : boolS\ in\ \mathcal{T}(F,V) \wedge \tau = boolS$
⟨*proof*⟩

lemma *const-of-bool-hastype-in-C*[*simp*]:
$const\text{-}of\text{-}bool\ b : \sigma\ in\ \mathcal{T}(C,V) \longleftrightarrow \sigma = boolS$
⟨*proof*⟩

lemma *const-of-bool-hastype-in-F*[*simp*]:
$const\text{-}of\text{-}bool\ b : \sigma\ in\ \mathcal{T}(F,V) \longleftrightarrow \sigma = boolS$
⟨*proof*⟩

lemma *sorts-ssig-F*[*simp*]: $sorts\text{-}ssig\ F = \{boolS\}$
⟨*proof*⟩

lemma *sorts-ssig-C*[*simp*]: $sorts\text{-}ssig\ C = \{boolS\}$
⟨*proof*⟩

end

locale *logic-signature* = *logic-signature-syntax* +
assumes *neg-neq-conj*[*simp*]: $negF \neq conjF$
and *neg-neq-disj*[*simp*]: $negF \neq disjF$
and *conj-neq-disj*[*simp*]: $conjF \neq disjF$
begin

lemmas *conj-neq-neg*[*simp*] = *neg-neq-conj*[*symmetric*]
lemmas *disj-neq-neg*[*simp*] = *neg-neq-disj*[*symmetric*]
lemmas *disj-neq-conj*[*simp*] = *conj-neq-disj*[*symmetric*]

interpretation *logic-symbols-syntax*⟨*proof*⟩

lemma *neg-hastype-iff*[*simp*]: $negF : \sigma s \to \tau\ in\ F \longleftrightarrow \sigma s = [boolS] \wedge \tau = boolS$
⟨*proof*⟩

lemma *Fun-neg-hastype-iff*: $Fun\ negF\ \varphi s : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow (\exists \varphi.\ \varphi s = [\varphi] \wedge \varphi$

: *boolS in* $\mathcal{T}(F,V) \wedge \tau = boolS$)
⟨*proof*⟩

lemma *not-hastype-iff*[*simp*]: $(\neg_t\ \varphi) : \tau$ *in* $\mathcal{T}(F,V) \longleftrightarrow \varphi$: *boolS in* $\mathcal{T}(F,V) \wedge \tau = boolS$
⟨*proof*⟩

lemma *imp-hastype-iff*[*simp*]: $(\varphi \longrightarrow_t \psi)$: *boolS in* $\mathcal{T}(F,V) \longleftrightarrow \varphi$: *boolS in* $\mathcal{T}(F,V) \wedge \psi$: *boolS in* $\mathcal{T}(F,V)$
⟨*proof*⟩

end

12.2 Semantics

A logic is an algebra over the logic signature, which interprets logical operators in the expected manner.

locale *truth-interpretation-syntax* =
fixes $I :: 'f \Rightarrow 'a\ list \Rightarrow 'a$ **and** *conjF* :: $'f$
begin

sublocale *truth-syntax*⟨*proof*⟩

abbreviation *true* **where** *true* ≡ *I conjF* []

lemma *satisfies-formula-iff*: $(I{:}(\sim);\alpha \models \varphi) \longleftrightarrow I[\![\varphi]\!]\alpha \sim true$
for *equiv* (**infix** $\sim$ *50*) ⟨*proof*⟩

lemmas *satisfies-formulaI* = *satisfies-formula-iff*[*THEN iffD2*]
lemmas *satisfies-formulaD* = *satisfies-formula-iff*[*THEN iffD1*]

end

locale *logic-syntax* =
fixes $F :: ('f,'s)$ *ssig* **and** $A :: 'a \rightharpoonup 's$ **and** $I :: 'f \Rightarrow 'a\ list \Rightarrow 'a$
and *boolS* :: $'s$ **and** *conjF disjF negF* :: $'f$
begin

sublocale *logic-symbols-syntax*⟨*proof*⟩
sublocale *truth-interpretation-syntax*⟨*proof*⟩

abbreviation *false* **where** *false* ≡ *I disjF* []

sublocale *logic*: *logic-signature-syntax*⟨*proof*⟩

end

We introduce *quasi-logic*, where the interpretations of formulas can be multi-valued, but relates to either true or false.

locale *quasi-truth-algebra* = *sorted-algebra* + *equivalence-syntax* + *truth-interpretation-syntax* +
  constrains *equiv* :: -
  assumes *true-is-true*: $true \sim true$
begin

sublocale *truth-interpretation-syntax*⟨*proof*⟩

lemma *valid-formula-subst*:
  assumes *val*: $A{:}I{:}(\sim) \models X.\ \varphi$ **and** ϑ: $\vartheta :_s X \to \mathcal{T}(F,V)$
  shows $A{:}I{:}(\sim) \models V.\ \varphi{\cdot}\vartheta$
  ⟨*proof*⟩

lemma *satisfies-true*[*simp*]: $I{:}(\sim);\alpha \models trueT$ ⟨*proof*⟩

lemma *valid-true*[*simp*]: $A{:}I{:}(\sim) \models X.\ trueT$ ⟨*proof*⟩

end

locale *quasi-logic* = *sorted-algebra* +
  equivalence-syntax + *logic-syntax* +
  logic: *subsignature logic.F F* +
  constrains *equiv* :: -
  assumes *neg-is-true*: $a : boolS\ in\ A \Longrightarrow$
    $I\ negF\ [a] \sim true \longleftrightarrow \neg\ (a \sim true)$
  assumes *conj-is-true*: $\forall\, a \in set\ as.\ a : boolS\ in\ A \Longrightarrow$
    $I\ conjF\ as \sim true \longleftrightarrow (\forall\, a \in set\ as.\ a \sim true)$
  assumes *disj-is-true*: $\forall\, a \in set\ as.\ a : boolS\ in\ A \Longrightarrow$
    $I\ disjF\ as \sim true \longleftrightarrow (\exists\, a \in set\ as.\ a \sim true)$
begin

sublocale *quasi-truth-algebra*
  ⟨*proof*⟩

sublocale *logic-part*: *subsignature-algebra logic.F F*⟨*proof*⟩

lemma *false-is-not-true*: $\neg false \sim true$
  ⟨*proof*⟩

lemma assumes $\forall\, \sigma \in set\ \sigma s.\ \sigma = boolS$
  shows *conj-hastype-iff-bool*[*simp*]: $conjF : \sigma s \to \tau\ in\ F \longleftrightarrow \tau = boolS$
    and *disj-hastype-iff-bool*[*simp*]: $disjF : \sigma s \to \tau\ in\ F \longleftrightarrow \tau = boolS$
  ⟨*proof*⟩

lemma *conj-hastype-replicate*: $conjF : replicate\ n\ boolS \to boolS\ in\ F$ ⟨*proof*⟩

lemma *disj-hastype-replicate*: $disjF : replicate\ n\ boolS \to boolS\ in\ F$ ⟨*proof*⟩

lemma assumes $\forall\, \varphi \in set\ \varphi s.\ \varphi : boolS\ in\ \mathcal{T}(F,V)$

shows *Fun-conj-hastype*: $Fun\ conjF\ \varphi s : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow \tau = boolS$
and *Fun-disj-hastype*: $Fun\ disjF\ \varphi s : \tau\ in\ \mathcal{T}(F,V) \longleftrightarrow \tau = boolS$
⟨*proof*⟩

lemma *true-hastype-in-Term-iff*[*simp*]: $trueT : \sigma\ in\ \mathcal{T}(F,V) \longleftrightarrow \sigma = boolS$
and *false-hastype-in-Term-iff*[*simp*]: $falseT : \sigma\ in\ \mathcal{T}(F,V) \longleftrightarrow \sigma = boolS$
⟨*proof*⟩

lemma *true-hastype-iff*[*simp*]: $true : \sigma\ in\ A \longleftrightarrow \sigma = boolS$
and *false-hastype-iff*[*simp*]: $false : \sigma\ in\ A \longleftrightarrow \sigma = boolS$
⟨*proof*⟩

sublocale *logic*: *logic-signature*
⟨*proof*⟩

lemma *neg-hastype*: $negF : [boolS] \to boolS\ in\ F$
⟨*proof*⟩

lemma *not-hastype-in-ssig*[*simp*]: $negF : [boolS] \to \tau\ in\ F \longleftrightarrow \tau = boolS$
⟨*proof*⟩

lemma *not-hastype-in-Term*[*simp*]: $\varphi : boolS\ in\ \mathcal{T}(F,V) \Longrightarrow (\neg_t\ \varphi) : boolS\ in\ \mathcal{T}(F,V)$
⟨*proof*⟩

lemma
assumes *as*: $\forall\, a \in set\ as.\ a : boolS\ in\ A$
shows *intp-conj-hastype*[*simp*]: $I\ conjF\ as : \tau\ in\ A \longleftrightarrow \tau = boolS$
and *intp-disj-hastype*[*simp*]: $I\ disjF\ as : \tau\ in\ A \longleftrightarrow \tau = boolS$
⟨*proof*⟩

lemma *intp-neg-hastype*[*simp*]: $a : boolS\ in\ A \Longrightarrow I\ negF\ [a] : \tau\ in\ A \longleftrightarrow \tau = boolS$
⟨*proof*⟩

lemma
assumes *a*: $a : boolS\ in\ A$ **and** *bs*: $\forall\, b \in set\ bs.\ b : boolS\ in\ A$
shows *conj-Cons*: $I\ conjF\ (a\#bs) \sim true \longleftrightarrow a \sim true \wedge I\ conjF\ bs \sim true$
and *disj-Cons*: $I\ disjF\ (a\#bs) \sim true \longleftrightarrow a \sim true \vee I\ disjF\ bs \sim true$
⟨*proof*⟩

lemma
assumes $I[\![\varphi]\!]\alpha : boolS\ in\ A$ **and** $\forall\, \psi \in set\ \varphi s.\ I[\![\psi]\!]\alpha : boolS\ in\ A$
shows *satisfies-conj-Cons'*: $(I{:}(\sim);\alpha \models Fun\ conjF\ (\varphi\ \#\ \varphi s)) \longleftrightarrow (I{:}(\sim);\alpha \models \varphi) \wedge (I{:}(\sim);\alpha \models Fun\ conjF\ \varphi s)$
and *satisfies-disj-Cons'*: $(I{:}(\sim);\alpha \models Fun\ disjF\ (\varphi\ \#\ \varphi s)) \longleftrightarrow (I{:}(\sim);\alpha \models \varphi) \vee (I{:}(\sim);\alpha \models Fun\ disjF\ \varphi s)$
⟨*proof*⟩

lemma
assumes $\forall \varphi \in$ *set* φs. $I[\![\varphi]\!]\alpha$: *boolS in A*
shows *satisfies-conj*: $(I{:}(\sim);\alpha \models$ *Fun conjF* $\varphi s) \longleftrightarrow (\forall \varphi \in$ *set* φs. $(I{:}(\sim);\alpha \models \varphi))$
and *satisfies-disj*: $(I{:}(\sim);\alpha \models$ *Fun disjF* $\varphi s) \longleftrightarrow (\exists \varphi \in$ *set* φs. $(I{:}(\sim);\alpha \models \varphi))$
⟨*proof*⟩

lemmas *satisfies-conjD* = *satisfies-conj*[*THEN iffD1*, *rotated* −*1*]
lemmas *satisfies-disjD* = *satisfies-disj*[*THEN iffD1*, *rotated* −*1*]
lemmas *satisfies-conjI* = *satisfies-conj*[*THEN iffD2*, *unfolded conj-imp-eq-imp-imp*]
lemmas *satisfies-disjI* = *satisfies-disj*[*THEN iffD2*]

lemma *satisfies-and*:
assumes $I[\![\varphi]\!]\alpha$: *boolS in A* **and** ψ: $I[\![\psi]\!]\alpha$: *boolS in A*
shows $(I{:}(\sim);\alpha \models \varphi \wedge_t \psi) \longleftrightarrow (I{:}(\sim);\alpha \models \varphi) \wedge (I{:}(\sim);\alpha \models \psi)$
⟨*proof*⟩

lemma *satisfies-or*:
assumes $I[\![\varphi]\!]\alpha$: *boolS in A* **and** ψ: $I[\![\psi]\!]\alpha$: *boolS in A*
shows $(I{:}(\sim);\alpha \models \varphi \vee_t \psi) \longleftrightarrow (I{:}(\sim);\alpha \models \varphi) \vee (I{:}(\sim);\alpha \models \psi)$
⟨*proof*⟩

lemmas *satisfies-andD* = *satisfies-and*[*THEN iffD1*, *rotated* −*1*]
lemmas *satisfies-orD* = *satisfies-or*[*THEN iffD1*, *rotated* −*1*]
lemmas *satisfies-andI* = *satisfies-and*[*THEN iffD2*, *unfolded conj-imp-eq-imp-imp*]
lemmas *satisfies-orI* = *satisfies-or*[*THEN iffD2*]

lemma *satisfies-not*:
assumes $I[\![\varphi]\!]\alpha$: *boolS in A*
shows $(I{:}(\sim);\alpha \models \neg_t \varphi) \longleftrightarrow \neg(I{:}(\sim);\alpha \models \varphi)$
⟨*proof*⟩

lemmas *satisfies-notD* = *satisfies-not*[*THEN iffD1*, *rotated* −*1*]
lemmas *satisfies-notI* = *satisfies-not*[*THEN iffD2*, *unfolded not-def*, *rule-format*]

lemma *satisfies-imp*:
assumes $I[\![\varphi]\!]\alpha$: *boolS in A* **and** $I[\![\psi]\!]\alpha$: *boolS in A*
shows $(I{:}(\sim);\alpha \models \varphi \longrightarrow_t \psi) \longleftrightarrow ((I{:}(\sim);\alpha \models \varphi) \longrightarrow (I{:}(\sim);\alpha \models \psi))$
⟨*proof*⟩

lemmas *satisfies-impD* = *satisfies-imp*[*THEN iffD1*, *rule-format*, *rotated* −*2*]
lemmas *satisfies-impI* = *satisfies-imp*[*THEN iffD2*, *rule-format*]

lemma
assumes φ : *boolS in* $\mathcal{T}(F,V)$ **and** $\forall \psi \in$ *set* φs. ψ : *boolS in* $\mathcal{T}(F,V)$ **and** $\alpha :_s V \to A$
shows *satisfies-conj-Cons*: $(I{:}(\sim);\alpha \models$ *Fun conjF* $(\varphi \# \varphi s)) \longleftrightarrow (I{:}(\sim);\alpha \models \varphi) \wedge (I{:}(\sim);\alpha \models$ *Fun conjF* $\varphi s)$

and *satisfies-disj-Cons*: $(I{:}(\sim);\alpha \models \mathit{Fun}\ \mathit{disjF}\ (\varphi\ \#\ \varphi s)) \longleftrightarrow (I{:}(\sim);\alpha \models \varphi) \vee (I{:}(\sim);\alpha \models \mathit{Fun}\ \mathit{disjF}\ \varphi s)$
⟨*proof*⟩

lemma *valid-and*:
assumes φ : *boolS in* $\mathcal{T}(F,V)$ **and** ψ : *boolS in* $\mathcal{T}(F,V)$
shows $(A{:}I{:}(\sim) \models V.\ \varphi \wedge_t \psi) \longleftrightarrow (A{:}I{:}(\sim) \models V.\ \varphi) \wedge (A{:}I{:}(\sim) \models V.\ \psi)$
⟨*proof*⟩

lemma *valid-conj*:
assumes $\forall \varphi \in \mathit{set}\ \varphi s.\ \varphi$: *boolS in* $\mathcal{T}(F,V)$
shows $(A{:}I{:}(\sim) \models V.\ \mathit{Fun}\ \mathit{conjF}\ \varphi s) \longleftrightarrow (\forall \varphi \in \mathit{set}\ \varphi s.\ (A{:}I{:}(\sim) \models V.\ \varphi))$
⟨*proof*⟩

lemma *valid-impD*:
assumes *imp*: $A{:}I{:}(\sim) \models V.\ \varphi \longrightarrow_t \psi$ **and** *prem*: $I{:}(\sim);\alpha \models \varphi$
and α: $\alpha :_s V \to A$ **and** φ: φ : *boolS in* $\mathcal{T}(F,V)$ **and** ψ: ψ : *boolS in* $\mathcal{T}(F,V)$
shows $I{:}(\sim);\alpha \models \psi$
⟨*proof*⟩

lemma *valid-impI*:
assumes $*$: $\bigwedge \alpha.\ \alpha :_s V \to A \Longrightarrow I{:}(\sim);\alpha \models \varphi \Longrightarrow I{:}(\sim);\alpha \models \psi$
and φ: φ : *boolS in* $\mathcal{T}(F,V)$ **and** ψ: ψ : *boolS in* $\mathcal{T}(F,V)$
shows $A{:}I{:}(\sim) \models V.\ \varphi \longrightarrow_t \psi$
⟨*proof*⟩

lemma *bool-in-sorts-ssig*: *boolS* $\in$ *sorts-ssig F*
⟨*proof*⟩

end

lemma *quasi-logic-cong*:
assumes F: $F = F'$
and A: $A = A'$
and I: $\bigwedge f\ \sigma s\ \tau\ as.\ f : \sigma s \to \tau\ \mathit{in}\ F' \Longrightarrow as :_l \sigma s\ \mathit{in}\ A' \Longrightarrow I\ f\ as = I'\ f\ as$
and *le*: $\bigwedge a.\ a$: *boolS in* $A' \Longrightarrow le\ a\ (I'\ \mathit{conjF}\ []) \longleftrightarrow le'\ a\ (I'\ \mathit{conjF}\ [])$
and *boolS*: *boolS* $=$ *boolS'*
and *conjF*: *conjF* $=$ *conjF'*
shows *quasi-logic F A I le boolS conjF disjF negF* $\longleftrightarrow$
quasi-logic F' A' I' le' boolS' conjF' disjF negF
⟨*proof*⟩

A logic is a quasi-logic with equality.

locale *logic* = *quasi-logic* **where** *equiv* = (=)
begin

lemma $A{:}I \models X.\ \varphi \equiv \forall \alpha :_s X \to A.\ I[\![\varphi]\!]\alpha = \mathit{true}$
⟨*proof*⟩

end

locale *logic-const* = *sorted-algebra-const* + *logic*

12.3 Propositional Logic

Here we define the propositional logic.

context *logic-signature-syntax* **begin**

definition $A :: bool \rightharpoonup 's$ **where**
$A\ b \equiv Some\ boolS$

lemma *hastype-in-A*[*simp*]: $b : \sigma\ in\ A \longleftrightarrow \sigma = boolS$ ⟨*proof*⟩

lemma *dom-A*[*simp*]: $dom\ A = UNIV$ ⟨*proof*⟩

lemma *A-restrict-ran*: $A \restriction^r S = (if\ boolS \in S\ then\ A\ else\ \emptyset)$
⟨*proof*⟩

definition I **where** $I\ f\ bs \equiv$
$if\ f = conjF\ then\ \forall\, b \in set\ bs.\ b$
$else\ if\ f = disjF\ then\ \exists\, b \in set\ bs.\ b$
$else\ \neg\ bs!0$

sublocale *truth-interpretation-syntax* I⟨*proof*⟩

sublocale *sorted-algebra* $F\ A\ I$
⟨*proof*⟩

sublocale *subsignature-algebra* $F\ F\ A\ I$
⟨*proof*⟩

end

context *logic-signature* **begin**

lemma *I-simps*[*simp*]:
$I\ conjF\ bs \longleftrightarrow (\forall\, b \in set\ bs.\ b)$
$I\ disjF\ bs \longleftrightarrow (\exists\, b \in set\ bs.\ b)$
$I\ negF\ bs \longleftrightarrow \neg\ bs!0$
⟨*proof*⟩

sublocale *logic* $F\ A\ I$
⟨*proof*⟩

sublocale *logic-const* $F\ A\ I$ *const-of-bool* C
⟨*proof*⟩

lemma *true-hastype-in-C*: $trueT : boolS\ in\ \mathcal{T}(C, V)$

and *false-hastype-in-C*: $falseT : boolS$ *in* $\mathcal{T}(C,V)$
⟨*proof*⟩

end

end

13 Logically Constrained Rewriting

theory *Constrained-Rewriting*
imports *Logic Sorted-Rewriting*
begin

Constrained rules extend rewrite rules with an extra term which represents the constraint.

datatype (*dead* $'f$, *dead* $'v$, *dead* $'s$) *lcrule* =
LCRule (*vars*: $'v \rightharpoonup 's$) (*lhs*: $('f,'v)$ *term*) (*rhs*: $('f,'v)$ *term*) (*constraint*: $('f,'v)$ *term*)
(-. - ⇝ - | - [100,51,51,30]21)

hide-const(**open**) *lcrule.vars lcrule.lhs lcrule.rhs lcrule.constraint*

abbreviation *ball-lcrule* **where** *ball-lcrule R P* ≡ ∀ *X l r c*. (*X*. *l* ⇝ *r* | *c*) ∈ *R* ⟶ *P X l r c*
abbreviation *bex-lcrule* **where** *bex-lcrule R P* ≡ ∃ *X l r c*. (*X*. *l* ⇝ *r* | *c*) ∈ *R* ∧ *P X l r c*

syntax
-ball-lcrule :: *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ $('f,'v,'s)$ *lcrule set* ⇒ *bool* ⇒ *bool*
((3∀('(-. - ⇝ - | -')/ ∈ -)./ -) [0, 0, 0, 0, 0, 10] 10)
-bex-lcrule :: *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ *pttrn* ⇒ $('f,'v,'s)$ *lcrule set* ⇒ *bool* ⇒ *bool*
((3∃('(-. - ⇝ - | -')/ ∈ -)./ -) [0, 0, 0, 0, 0, 10] 10)

translations
∀(*X*. *l* ⇝ *r* | *c*) ∈ *R*. *e* ⇌ *CONST ball-lcrule R* (λ*X l r c*. *e*)
∃(*X*. *l* ⇝ *r* | *c*) ∈ *R*. *e* ⇌ *CONST bex-lcrule R* (λ*X l r c*. *e*)

To be a well-typed constrained rewrite rule, the left- and right-hand sides must have the same type and the constraint must be of bool type.

locale *lcrule-syntax* =
fixes F' F :: $('f,'s)$ *ssig* **and** *boolS* :: $'s$
begin

definition *lcrule* ≡ λ(*X*. *l* ⇝ *r* | φ) ⇒
(∃ σ. *l* : σ *in* $\mathcal{T}(F',X)$ ∧ *r* : σ *in* $\mathcal{T}(F',X)$ ∧ φ : *boolS in* $\mathcal{T}(F,X)$)

lemma *lcrule-simp*:
lcrule $(X.\ l \rightsquigarrow r \mid \varphi) \longleftrightarrow (\exists \sigma.\ l : \sigma\ in\ \mathcal{T}(F',X) \wedge r : \sigma\ in\ \mathcal{T}(F',X) \wedge \varphi : boolS\ in\ \mathcal{T}(F,X))$
⟨*proof*⟩

lemma *lcruleI*:
$\varrho = (X.\ l \rightsquigarrow r \mid \varphi) \Longrightarrow l : \sigma\ in\ \mathcal{T}(F',X) \Longrightarrow r : \sigma\ in\ \mathcal{T}(F',X) \Longrightarrow \varphi : boolS\ in\ \mathcal{T}(F,X) \Longrightarrow lcrule\ \varrho$
⟨*proof*⟩

lemma *lcrule-has-same-type*: $lcrule\ (X.\ l \rightsquigarrow r \mid \varphi) \Longrightarrow l : \sigma\ in\ \mathcal{T}(F',X) \longleftrightarrow r : \sigma\ in\ \mathcal{T}(F',X)$
⟨*proof*⟩

end

A well-typed LCTRS requires that all rules are well-typed and signatures satisfy expected inclusion.

locale *lctrs* = *lcrule-syntax F' F boolS* **for** *F' R F boolS* +
assumes *tsig-le-sig*: $F \subseteq_m F'$
and *typed*: $\varrho \in R \Longrightarrow lcrule\ \varrho$
begin

lemmas *tTerm-le* = *Term-mono-left*[*OF tsig-le-sig*]

end

The LCTRS semantics demands that instantiations of rules are restricted to those which map variables in constraints to *values*. Therefore we need to specify what are values: here, rather than assuming a *constant symbol* for each semantic values [2], we only require every value $v : \sigma\ in\ A$ is represented by a ground constructor term $\eta\ v : \sigma\ in\ \mathcal{T}(C)$.

locale *lctrs-semantics-syntax* =
fixes F' :: $('f,'s)$ *ssig*
and R :: $('f,'v,'s)$ *lcrule set*
and F :: $('f,'s)$ *ssig*
and A :: $'a \rightharpoonup 's$
and I :: $'f \Rightarrow 'a\ list \Rightarrow 'a$
and η :: $'a \Rightarrow ('f,unit)\ term$
and C :: $('f,'s)$ *ssig*
and *boolS* :: $'s$
and *conjF* :: $'f$
begin

sublocale *truth-syntax*⟨*proof*⟩

We do not define constrained rewrite steps from scratch. We just define the (sorted) TRS which the LCTRS represents. That is, the instances of

constrained rules with respect to ground constructor substitutions over the variables in the constraints that satisfy the constraint.

definition *as-trs* ≡
  {(*X*. *l*·ϑ ⇝ *r*·ϑ) | *X l r* φ ϑ.
  (*X*. *l* ⇝ *r* | φ) ∈ *R* ∧
  ϑ :$_s$ *X* |‘ *vars* φ → 𝒯(*C*,∅) ∧ (∀ *x* ∈ *dom X* − *vars* φ. ϑ *x* = *Var x*) ∧
  (*A*:*I* ⊨ ∅. φ·ϑ)}

lemma *as-trs-iff*: (*X*. *l*′ ⇝ *r*′) ∈ *as-trs* ⟷ (∃ *l r* φ ϑ.
  (*X*. *l* ⇝ *r* | φ) ∈ *R* ∧
  ϑ :$_s$ *X* |‘ *vars* φ → 𝒯(*C*,∅) ∧ (∀ *x* ∈ *dom X* − *vars* φ. ϑ *x* = *Var x*) ∧ (*A*:*I* ⊨∅.
φ·ϑ) ∧
  l′ = *l*·ϑ ∧ *r*′ = *r*·ϑ)
  ⟨*proof*⟩

lemma *as-trsI*: (*X*. *l* ⇝ *r* | φ) ∈ *R* ⟹ *A*:*I* ⊨ ∅. φ·ϑ ⟹
  ϑ :$_s$ *X* |‘ *vars* φ → 𝒯(*C*,∅) ⟹ ∀ *x* ∈ *dom X* − *vars* φ. ϑ *x* = *Var x* ⟹ (*X*. *l*·ϑ
⇝ *r*·ϑ) ∈ *as-trs*
  ⟨*proof*⟩

end

Rewriting in LCTRS requires extra rewrite steps of theory terms called “calculation step”. Below we define a one that reduces a ground basic theory term in one step to the representation of its value. We cannot just add definitions in existing locales, because the new definitions will not be imported to existing interpretations (in Isabelle 2025).

locale *sorted-algebra-calculation* = *sorted-algebra-const* +
  constrains *A* :: ′*a* ⇀ ′*s* **and** *I* :: ′*f* ⇒ - **and** η :: -
begin

To avoid complication in types, this TRS contains ground rules whose type of variables is *unit*.

definition *calculation-trs* :: (′*f*,*unit*,′*s*) *axiom set* **where**
  calculation-trs ≡
  {(∅. *Fun f ls* ⇝ η (*I f* [*I*⟦*l*⟧*undefined*. *l* ← *ls*])) |*f ls* σ*s* τ.
  f : σ*s* → τ *in F* ∧ ¬*f* : σ*s* → τ *in C* ∧ *ls* :$_l$ σ*s in* 𝒯(*C*)}

A root step of the calculation TRS is polymorphic, as expected:

lemma *calculation-rootstepp-iff*:
  s −*F*′:*V*:*calculation-trs*→$^{\varepsilon}$ *t* ⟷ (∃ *f ls* σ*s* τ.
   f : σ*s* → τ *in F* ∧ ¬*f* : σ*s* → τ *in C* ∧ *ls* :$_l$ σ*s in* 𝒯(*C*,∅) ∧
   s = *Fun f ls* ∧ *t* = *const.term-of* (*I*⟦*s*⟧α)) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

sublocale *calculation-trs*: *sorted-trs F calculation-trs*
  ⟨*proof*⟩

end

locale *lctrs-semantics* = *lctrs-semantics-syntax* +
  lctrs + *sorted-algebra-calculation*
begin

sublocale *quasi-truth-algebra F A I* (=) ⟨*proof*⟩

sublocale *as-trs*: *sorted-trs F′ as-trs*
  ⟨*proof*⟩

lemma *lctrs-rootstepp-iff*: $s\ -F'{:}V{:}as\text{-}trs{\to}^{\varepsilon}\ t \longleftrightarrow$ (
  $\exists (X.\ l \rightsquigarrow r \mid \varphi) \in R.\ \exists \vartheta :_s X \to \mathcal{T}(F', V).$
  $\vartheta :_s X \mid `vars\ \varphi \to \mathcal{T}(C,\emptyset) \wedge (A{:}I \models \emptyset.\ \varphi{\cdot}\vartheta) \wedge s = l{\cdot}\vartheta \wedge t = r{\cdot}\vartheta)$
  (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *lctrs-rootsteppI*: $(X.\ l \rightsquigarrow r \mid \varphi) \in R \Longrightarrow \vartheta :_s X \to \mathcal{T}(F', V) \Longrightarrow$
  $\vartheta :_s X \mid `vars\ \varphi \to \mathcal{T}(C,\emptyset) \Longrightarrow (A{:}I \models \emptyset.\ \varphi{\cdot}\vartheta) \Longrightarrow s = l{\cdot}\vartheta \Longrightarrow t = r{\cdot}\vartheta \Longrightarrow$
  $s\ -F'{:}V{:}as\text{-}trs{\to}^{\varepsilon}\ t$
  ⟨*proof*⟩

end

end

13.1 Sorted Injections

theory *Sorted-Injections*
  imports *Sorted-Terms-More*
begin

One can define the image of an algebra with respect to an injection.

definition *image-intp η A I f as* ≡ *η* (*I f* (*map* (*inv-into* (*dom A*) *η*) *as*))

lemma *image-intp-Nil*[*simp*]:
  image-intp η A I g [] = *η* (*I g* [])
  ⟨*proof*⟩

locale *sorted-injection* =
  fixes *η A*
  assumes *inj-on*: *inj-on η* (*dom A*)
begin

lemma *inv-app*[*simp*]:
  shows *a* : *σ in A* ⟹ *inv-into* (*dom A*) *η* (*η a*) = *a*
  ⟨*proof*⟩

lemma *map-inv-app*: $as :_l \sigma s\ in\ A \Longrightarrow map\ (inv\text{-}into\ (dom\ A)\ \eta \circ \eta)\ as = as$
⟨*proof*⟩

lemma *in-dom-inj*: $a \in dom\ A \Longrightarrow a' \in dom\ A \Longrightarrow \eta\ a = \eta\ a' \longleftrightarrow a = a'$
⟨*proof*⟩

lemma *hastype-inj*: $a : \sigma\ in\ A \Longrightarrow a' : \sigma'\ in\ A \Longrightarrow \eta\ a = \eta\ a' \longleftrightarrow a = a'$
⟨*proof*⟩

sublocale *sort-preserving* η
⟨*proof*⟩

lemma *image-intp-map*:
assumes $as :_l \sigma s\ in\ A$ **shows** $image\text{-}intp\ \eta\ A\ I\ g\ (map\ \eta\ as) = \eta\ (I\ g\ as)$
⟨*proof*⟩

lemma *sorted-injection-subsset*:
assumes $A' \subseteq_m A$
shows *sorted-injection* $\eta\ A'$
⟨*proof*⟩

end

lemma *sorted-injection-cong*:
$(\bigwedge a.\ a \in dom\ A \Longrightarrow f\ a = f'\ a) \Longrightarrow sorted\text{-}injection\ f\ A = sorted\text{-}injection\ f'\ A$
⟨*proof*⟩

lemma *sorted-injection-o*:
assumes *sorted-injection* $f\ A$ **and** *sorted-injection* $g\ (f\ ``^s\ A)$
shows *sorted-injection* $(g \circ f)\ A$
⟨*proof*⟩

lemma *unisorted-injection*[*simp*]: *sorted-injection* $f\ (unisorted\ A) \longleftrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

interpretation *Inl*: *sorted-injection Inl*
⟨*proof*⟩

interpretation *Inr*: *sorted-injection Inr*
⟨*proof*⟩

One can define the image of an algebra with respect to an injection.

locale *sorted-algebra-injection* = *source*: *sorted-algebra* + *sorted-injection*
begin

sublocale *sorted-distributive* $F\ \eta\ A\ I$ ‹*image-intp* $\eta\ A\ I$›
⟨*proof*⟩

lemmas *image-sorted-algebra* = *image.sorted-algebra-axioms*

Therefore, one can embed an algebra in a larger carrier if the relevant sorts are preserved.

lemma *embed-sorted-algebra*:
assumes *eq*: $B \restriction^r \mathit{sorts\text{-}ssig}\ F = \eta\ {}^{`s}\ A \restriction^r \mathit{sorts\text{-}ssig}\ F$
shows *sorted-algebra F B* (*image-intp* η *A I*)
⟨*proof*⟩

end

end

14 Extension of Algebras

theory *Algebra-Extensions*
imports *Sorted-Injections*
begin

lemma *all-set-isl*: $(\forall\, a{\in}\mathit{set}\ \mathit{as}.\ \mathit{isl}\ a) \longleftrightarrow (\exists\, \mathit{ls}.\ \mathit{as} = \mathit{map}\ \mathit{Inl}\ \mathit{ls})$
⟨*proof*⟩

lemma *not-isl*: $\neg \mathit{isl}\ x \longleftrightarrow (\exists\, a.\ x = \mathit{Inr}\ a)$
⟨*proof*⟩

lemma *all-set-not-isl*: $(\forall\, a{\in}\mathit{set}\ \mathit{as}.\ \neg\ \mathit{isl}\ a) \longleftrightarrow (\exists\, \mathit{ls}.\ \mathit{as} = \mathit{map}\ \mathit{Inr}\ \mathit{ls})$
⟨*proof*⟩

lemma *map-Inl-eq-map-Inr*: $\mathit{map}\ \mathit{Inl}\ \mathit{xs} = \mathit{map}\ \mathit{Inr}\ \mathit{ys} \longleftrightarrow \mathit{xs} = [] \wedge \mathit{ys} = []$
⟨*proof*⟩

lemma *map-Inr-eq-map-Inl*: $\mathit{map}\ \mathit{Inr}\ \mathit{ys} = \mathit{map}\ \mathit{Inl}\ \mathit{xs} \longleftrightarrow \mathit{xs} = [] \wedge \mathit{ys} = []$
⟨*proof*⟩

lemmas $\mathit{islE} = \mathit{isl\text{-}def}[\mathit{THEN}\ \mathit{iffD1}, \mathit{THEN}\ \mathit{exE}]$

lemma *map-le-map-add2*:
assumes $\mathit{dom}\ F \cap \mathit{dom}\ G = \{\}$ **shows** $F \subseteq_m F +\!\!+ G$
⟨*proof*⟩

lemma *hastype-in-add*:
$a : \sigma\ \mathit{in}\ A +\!\!+ B \longleftrightarrow a : \sigma\ \mathit{in}\ B \vee a \notin \mathit{dom}\ B \wedge a : \sigma\ \mathit{in}\ A$
⟨*proof*⟩

lemma *fun-hastype-in-add*:
$f : \sigma s \to \tau\ \mathit{in}\ F +\!\!+ G \longleftrightarrow f : \sigma s \to \tau\ \mathit{in}\ G \vee (f,\sigma s) \notin \mathit{dom}\ G \wedge f : \sigma s \to \tau\ \mathit{in}\ F$
⟨*proof*⟩

lemma *fun-hastype-in-add-disj*:

assumes *dom F* ∩ *dom G* = {}
shows *f* : σ*s* → τ *in F* ++ *G* ⟷ *f* : σ*s* → τ *in G* ∨ *f* : σ*s* → τ *in F*
⟨*proof*⟩

lemma *hastype-in-imageD*:
assumes *fa* : σ *in f* ‘*s A*
shows ∃ *a* : σ *in A*. *fa* = *f a*
⟨*proof*⟩

lemma *hastype-list-in-imageD*:
assumes *fas* :*l* σ*s in f* ‘*s A*
shows ∃ *as*. *as* :*l* σ*s in A* ∧ *fas* = *map f as*
⟨*proof*⟩

lemmas *hastype-list-in-imageE* =
hastype-list-in-imageD[*THEN exE*,*unfolded conj-imp-eq-imp-imp*]

lemma *all-set-hastype-in-imageD*:
assumes ∀ *fa* ∈ *set fas*. *fa* : σ *in f* ‘*s A*
shows ∃ *as*. (∀ *a* ∈ *set as*. *a* : σ *in A*) ∧ *fas* = *map f as*
⟨*proof*⟩

lemmas *all-set-hastype-in-imageE* =
all-set-hastype-in-imageD[*THEN exE*, *unfolded conj-imp-eq-imp-imp*]

14.1 Disjoint Sum of Sorted Sets

The disjoint sum of two sorted sets A and B are just *case-sum A B*.

lemma *Inl-hastype*[*simp*]: *Inl a* : σ *in case-sum A B* ⟷ *a* : σ *in A*
and *Inr-hastype*[*simp*]: *Inr b* : σ *in case-sum A B* ⟷ *b* : σ *in B*
⟨*proof*⟩

lemma *hastype-in-case-sum*: *x* : σ *in case-sum A B* ⟷
(∃ *a*. *x* = *Inl a* ∧ *a* : σ *in A*) ∨ (∃ *b*. *x* = *Inr b* ∧ *b* : σ *in B*)
⟨*proof*⟩

14.2 Extending Signature and Interpretations

We merge an F-interpretation I with a G-interpretation J into an $(F+G)$-interpretation. To this end, we test if I is applicable to given function symbol and arguments, in which case apply I and otherwise apply J.

definition *interpretable F A f as* ≡
case those (*map A as*) *of Some* σ*s* ⇒ (*f*,σ*s*) ∈ *dom F* | *None* ⇒ *False*

lemma *interpretableI*:
assumes *f* : σ*s* → τ *in F* **and** *as* :*l* σ*s in A* **shows** *interpretable F A f as*
⟨*proof*⟩

lemma *interpretableE*:
assumes *interpretable F A f as*
and $\bigwedge \sigma s\ \tau.\ f : \sigma s \to \tau$ *in* $F \Longrightarrow$ *as* $:_l \sigma s$ *in* $A \Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *interpretable-iff*:
assumes *as* $:_l \sigma s$ *in* A
shows *interpretable F A f as* $\longleftrightarrow (f,\sigma s) \in$ *dom F*
⟨*proof*⟩

definition *extend-intp F A I J f as* $\equiv$
if interpretable F A f as then I f as else J f as

lemma *extend-intp1*:
assumes $\neg$ *interpretable F A f as*
shows *extend-intp F A I J f as* $=$ *J f as*
⟨*proof*⟩

lemma *extend-intp2*:
assumes $f : \sigma s \to \tau$ *in* F **and** *as* $:_l \sigma s$ *in* A
shows *extend-intp F A I J f as* $=$ *I f as*
⟨*proof*⟩

lemma *interpretable-sorts-ssig*: *interpretable F* $(A \restriction^r$ *sorts-ssig F*$)$ $=$ *interpretable F A*
⟨*proof*⟩

lemma *extend-intp-sorts-ssig*: *extend-intp F* $(A \restriction^r$ *sorts-ssig F*$)$ $=$ *extend-intp F A*
⟨*proof*⟩

lemma *extend-intp-cong*: $F = F' \Longrightarrow A \restriction^r$ *sorts-ssig* $F' = A' \restriction^r$ *sorts-ssig* $F' \Longrightarrow$
extend-intp F A $=$ *extend-intp F' A'*
⟨*proof*⟩

definition *extend-image-intp* η *F A I* $\equiv$ *extend-intp F* $(\eta\ ``^s\ A)$ (*image-intp* η *A I*)

lemma *extend-image-intp1*:
assumes $\neg$ *interpretable F* $(\eta\ ``^s\ A)$ *f as*
shows *extend-image-intp* η *F A I J f as* $=$ *J f as*
⟨*proof*⟩

lemma(**in** *sorted-injection*) *extend-image-intp2*:
assumes *f*: $f : \sigma s \to \tau$ *in* F **and** *as*: *as* $:_l \sigma s$ *in* A
shows *extend-image-intp* η *F A I J f* $(map\ \eta$ *as*$) = \eta$ $(I\ f$ *as*$)$
⟨*proof*⟩

context *sorted-algebra* **begin**

lemma *extend-intp-eval1*:
  assumes *disj*: $dom\ F \cap dom\ G = \{\}$
    and *s*: $s : \sigma\ in\ \mathcal{T}(F,V)$
    and α: $\alpha :_s V \to A$
  shows $extend\text{-}intp\ G\ A\ J\ I[\![s]\!]\alpha = I[\![s]\!]\alpha$
  ⟨*proof*⟩

lemma *extend-intp-eval2*:
  assumes *s*: $s : \sigma\ in\ \mathcal{T}(F,V)$
    and α: $\alpha :_s V \to A$
  shows $extend\text{-}intp\ F\ A\ I\ J[\![s]\!]\alpha = I[\![s]\!]\alpha$
⟨*proof*⟩

One can extend the signature by merging interpretations.

lemma *extend-sorted-algebra*:
  assumes *GAJ*: *sorted-algebra G A J*
  shows *sorted-algebra* $(G{+}{+}F)\ A$ (*extend-intp F A I J*)
⟨*proof*⟩

end

context *sorted-algebra-injection* **begin**

lemma *extend-image-intp-eval2*:
  assumes *s*: $s : \sigma\ in\ \mathcal{T}(F,V)$ **and** α: $\alpha :_s V \to A$
  shows $extend\text{-}image\text{-}intp\ \eta\ F\ A\ I\ J[\![s]\!](\eta \circ \alpha) = \eta\ (I[\![s]\!]\alpha)$
  ⟨*proof*⟩

In combination, one can extend algebra after injecting over a larger carrier.

lemma *extend-image-sorted-algebra*:
  assumes *GBJ*: *sorted-algebra G B J*
    and *BA*: $B \restriction^r sorts\text{-}ssig\ F = \eta\ ``^s\ A \restriction^r sorts\text{-}ssig\ F$
  shows *sorted-algebra* $(G{+}{+}F)\ B$ (*extend-image-intp* $\eta\ F\ A\ I\ J$)
⟨*proof*⟩

end

end

15 Extending Algebra into Logic

theory *Logic-Extensions*
  imports *Logic Algebra-Extensions*
begin

Let us define the image under f of a relation R by $\{(f\ x,\ f\ y)\ |\ (x,y) \in R\}$.

definition *image-rel f R* ≡ *map-prod f f ‘ R*
definition *image-relp f r a b* ≡ ∃ *x y. r x y* ∧ *a* = *f x* ∧ *b* = *f y*

lemma *image-rel*: (*a*,*b*) ∈ *image-rel f R* ⟷ (∃ (*x*,*y*) ∈ *R*. *a* = *f x* ∧ *b* = *f y*)
⟨*proof*⟩

lemma *image-relp*: *image-relp f r a b* ⟷ (∃ *x y. r x y* ∧ *a* = *f x* ∧ *b* = *f y*)
⟨*proof*⟩

lemma *image-relp-eq-range*: *image-relp f* (=) = (=) ↾ *range f*
⟨*proof*⟩

lemma *in-rel-image*: *in-rel* (*image-rel f R*) = *image-relp f* (*in-rel R*)
⟨*proof*⟩

context *quasi-logic* **begin**

The injective image of a quasi logic is a quasi logic with respect to the image relation.

lemma *image-quasi-logic*:
assumes *inj-on η* (*dom A*)
shows *quasi-logic F* (*η* ‘*s A*) (*image-intp η A I*) (*image-relp η* ((∼) ↾ *dom A*)) *boolS conjF disjF negF*
⟨*proof*⟩

One can extend the carrier of logic if relevant sorts are preserved.

lemma *quasi-logic-carrier*:
assumes *B* ↾*r sorts-ssig F* = *A* ↾*r sorts-ssig F*
shows *quasi-logic F B I* (∼) *boolS conjF disjF negF*
⟨*proof*⟩

Therefore, one can embed logic by injection.

lemma *embed-quasi-logic*:
assumes *η*: *inj-on η* (*dom A*)
and *BA*: *B* ↾*r sorts-ssig F* = *η* ‘*s A* ↾*r sorts-ssig F*
shows *quasi-logic F B* (*image-intp η A I*) (*image-relp η* ((∼) ↾ *dom A*)) *boolS conjF disjF negF*
⟨*proof*⟩

One can extend the signature and interpretation just as algebras.

lemma *extend-true*[*simp*]: *extend-intp F A I J conjF* [] = *true*
⟨*proof*⟩

lemma *extend-false*[*simp*]: *extend-intp F A I J disjF* [] = *false*
⟨*proof*⟩

lemma *extend-image-intp-true*[*simp*]: *extend-image-intp f F A I J conjF* [] = *f true*

⟨*proof*⟩

lemma *extend-image-intp-false*[*simp*]: *extend-image-intp f F A I J disjF* [] = *f false*
⟨*proof*⟩

lemma *extend-quasi-logic*:
assumes *J*: *sorted-algebra G A J*
shows *quasi-logic* (*G*++*F*) *A* (*extend-intp F A I J*) (∼) *boolS conjF disjF negF*
⟨*proof*⟩

Finally, one can extend the signature, interpretation and carrier after embedding.

lemma *extend-image-quasi-logic*:
assumes *J*: *sorted-algebra G B J*
and *η*: *inj-on η* (*dom A*)
and *BA*: *B* ↾^{r} *sorts-ssig F* = *η* ‘^{s} *A* ↾^{r} *sorts-ssig F*
shows *quasi-logic* (*G*++*F*) *B* (*extend-image-intp η F A I J*) (*image-relp η* ((∼) ↾ *dom A*)) *boolS conjF disjF negF*
⟨*proof*⟩

end

context *logic* **begin**

lemmas *logic-carrier* = *quasi-logic-carrier*[*folded logic-def*]

lemmas *extend-logic* = *extend-quasi-logic*[*folded logic-def*]

lemma *image-logic*:
assumes *inj-on η* (*dom A*)
shows *logic F* (*η* ‘^{s} *A*) (*image-intp η A I*) *boolS conjF disjF negF*
⟨*proof*⟩

lemma *extend-image-logic*:
assumes *J*: *sorted-algebra G B J*
and *η*: *inj-on η* (*dom A*)
and *BA*: *B* ↾^{r} *sorts-ssig F* = *η* ‘^{s} *A* ↾^{r} *sorts-ssig F*
shows *logic* (*G*++*F*) *B* (*extend-image-intp η F A I J*) *boolS conjF disjF negF*
⟨*proof*⟩

end

end

16 Concrete Logics

16.1 Bool Logic

Here we define the core logic, that can be used to "logicalize" algebras. To be able to extend easily, we fix types of symbols to strings.

theory *Core-Logic*
  imports *Logic-Extensions*
begin

definition *bool-s* ($bool_s$) **where** $bool_s \equiv$ *''Bool''*
definition *conj-f* ($conj_f$) **where** $conj_f \equiv$ *''and''*
definition *disj-f* ($disj_f$) **where** $disj_f \equiv$ *''or''*
definition *neg-f* (neg_f) **where** $neg_f \equiv$ *''not''*

interpretation *Bool*: *logic-signature-syntax* $bool_s$ $conj_f$ $disj_f$ $neg_f \langle proof \rangle$

interpretation *Bool*: *truth-syntax* $conj_f \langle proof \rangle$
notation *Bool.trueT* ($True_t$)

interpretation *Bool*: *logic-symbols-syntax* $conj_f$ $disj_f$ $neg_f \langle proof \rangle$

notation *Bool.falseT* ($False_t$)

interpretation *Bool*: *logic-signature* $bool_s$ $conj_f$ $disj_f$ neg_f
  $\langle proof \rangle$

thm *Bool.sorts-ssig-F*

Any algebra over strings can be "logicalized", if precisely truth values have bool sort.

abbreviation *logicalize-intp* $\equiv$ *extend-image-intp Inl Bool.F Bool.A Bool.I*

thm *Bool.extend-image-logic*[*OF - inj-Inl*]

lemma *logicalize*:
  assumes *FAI*: *sorted-algebra F A I* **and** *b*: $\{a.\ a : bool_s\ in\ A\} = range\ Inl$
  shows *logic* $(F{+}{+}Bool.F)$ *A* (*logicalize-intp I*) $bool_s$ $conj_f$ $disj_f$ neg_f
  $\langle proof \rangle$

end

16.2 Natural Arithmetic

theory *Nats-Logic*
  imports *Core-Logic*
begin

The natural number algebra consists of zero and the successor operator as

constructors, and some defined operators and relations.

definition *nat-s* (nat_s) **where** $nat_s \equiv ''nat''$
definition *zero-f* (0_f) **where** $0_f \equiv ''0''$
definition *Suc-f* (Suc_f) **where** $Suc_f \equiv ''Suc''$
definition *add-f* ($+_f$) **where** $+_f \equiv ''+''$
definition *mult-f* ($*_f$) **where** $*_f \equiv ''*''$
definition *less-f* ($<_f$) **where** $<_f \equiv ''<''$

lemma *bool-neq-nat*[*simp*]: $bool_s \neq nat_s$
⟨*proof*⟩

lemma *Nat-syms-neq*[*simp*]:
$0_f \neq Suc_f\ 0_f \neq +_f\ 0_f \neq *_f\ 0_f \neq <_f\ 0_f \neq conj_f\ 0_f \neq disj_f\ 0_f \neq neg_f$
$Suc_f \neq +_f\ Suc_f \neq *_f\ Suc_f \neq <_f\ Suc_f \neq conj_f\ Suc_f \neq disj_f\ Suc_f \neq neg_f$
$+_f \neq <_f\ +_f \neq *_f\ +_f \neq conj_f\ +_f \neq disj_f\ +_f \neq neg_f$
$*_f \neq <_f\ *_f \neq conj_f\ *_f \neq disj_f\ *_f \neq neg_f$
$<_f \neq conj_f\ <_f \neq disj_f\ <_f \neq neg_f$
⟨*proof*⟩

lemmas [*simp*] = *Nat-syms-neq*[*symmetric*]

abbreviation *zero-t* (0_t) **where** $0_t \equiv Fun\ 0_f\ []$
abbreviation *suc-t* (Suc_t) **where** $Suc_t\ s \equiv Fun\ Suc_f\ [s]$
abbreviation *add-t* (**infix** $+_t$ *70*) **where** $s +_t t \equiv Fun\ +_f\ [s,t]$
abbreviation *mult-t* (**infix** $*_t$ *70*) **where** $s *_t t \equiv Fun\ *_f\ [s,t]$
abbreviation *less-t* (**infix** $<_t$ *50*) **where** $s <_t t \equiv Fun\ <_f\ [s,t]$

definition $NatC \equiv [(0_f,[]) \mapsto nat_s,\ (Suc_f,[nat_s]) \mapsto nat_s]$

abbreviation $NatBoolC \equiv NatC +\!\!+ Bool.C$

definition $NatD \equiv [(+_f,nat_s..) \mapsto nat_s,\ (*_f,nat_s..) \mapsto nat_s,\ (<_f,[nat_s,nat_s]) \mapsto bool_s]$

abbreviation $NatF \equiv NatC +\!\!+ NatD$

abbreviation $NatBoolF \equiv NatF +\!\!+ Bool.F$

lemma *hastype-in-NatC*: $f : \sigma s \to \tau\ in\ NatC \longleftrightarrow$
$f = 0_f \wedge \sigma s = [] \wedge \tau = nat_s \vee$
$f = Suc_f \wedge \sigma s = [nat_s] \wedge \tau = nat_s$ ⟨*proof*⟩

lemma *hastype-in-NatC-E*:
assumes $f : \sigma s \to \tau\ in\ NatC$
and $f = 0_f \Longrightarrow \sigma s = [] \Longrightarrow \tau = nat_s \Longrightarrow thesis$
and $f = Suc_f \Longrightarrow \sigma s = [nat_s] \Longrightarrow \tau = nat_s \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *hastype-in-NatC-simps*[*simp*]:
$0_f : \sigma s \to \tau$ *in* $NatC \longleftrightarrow \sigma s = [] \wedge \tau = nat_s$
$Suc_f : \sigma s \to \tau$ *in* $NatC \longleftrightarrow \sigma s = [nat_s] \wedge \tau = nat_s$
⟨*proof*⟩

lemma *Fun-hastype-in-NatC*:
shows [*simp*]: *Fun* 0_f $ss : \tau$ *in* $\mathcal{T}(NatC, V) \longleftrightarrow ss = [] \wedge \tau = nat_s$
and *Fun* Suc_f $ss : \tau$ *in* $\mathcal{T}(NatC, V) \longleftrightarrow$
$\tau = nat_s \wedge (\exists s.\ ss = [s] \wedge s : nat_s$ *in* $\mathcal{T}(NatC, V))$
⟨*proof*⟩

lemma *Suc-t-hastype-in-NatC*[*simp*]:
$Suc_t\ s : \sigma$ *in* $\mathcal{T}(NatC, V) \longleftrightarrow \sigma = nat_s \wedge s : nat_s$ *in* $\mathcal{T}(NatC, V)$
⟨*proof*⟩

lemma *dom-NatC*[*simp*]: *dom* $NatC = \{(0_f,[]),(Suc_f,[nat_s])\}$
⟨*proof*⟩

lemma *NatC-le-extended*: $NatC \subseteq_m NatC$ ++ *Bool.C*
⟨*proof*⟩

lemma *hastype-in-NatD*: $f : \sigma s \to \tau$ *in* $NatD \longleftrightarrow$
$(f = +_f \vee f = *_f) \wedge (\forall \sigma \in set\ \sigma s.\ \sigma = nat_s) \wedge \tau = nat_s \vee$
$f = <_f \wedge \sigma s = [nat_s,nat_s] \wedge \tau = bool_s$
⟨*proof*⟩

lemma *hastype-in-NatD-E*:
assumes $f : \sigma s \to \tau$ *in* $NatD$
and $\bigwedge n.\ f = +_f \Longrightarrow \sigma s = replicate\ n\ nat_s \Longrightarrow \tau = nat_s \Longrightarrow thesis$
and $\bigwedge n.\ f = *_f \Longrightarrow \sigma s = replicate\ n\ nat_s \Longrightarrow \tau = nat_s \Longrightarrow thesis$
and $\bigwedge n.\ f = <_f \Longrightarrow \sigma s = [nat_s,nat_s] \Longrightarrow \tau = bool_s \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *hastype-in-NatD-simps*[*simp*]:
$+_f : \sigma s \to \tau$ *in* $NatD \longleftrightarrow (\forall \sigma \in set\ \sigma s.\ \sigma = nat_s) \wedge \tau = nat_s$
$*_f : \sigma s \to \tau$ *in* $NatD \longleftrightarrow (\forall \sigma \in set\ \sigma s.\ \sigma = nat_s) \wedge \tau = nat_s$
$<_f : \sigma s \to \tau$ *in* $NatD \longleftrightarrow \sigma s = [nat_s,nat_s] \wedge \tau = bool_s$
⟨*proof*⟩

lemma *NatC-NatD*: *dom* $NatC \cap$ *dom* $NatD = \{\}$
⟨*proof*⟩

lemmas *hastype-in-NatF* = *fun-hastype-in-add-disj*[*OF NatC-NatD*]

lemma *hastype-in-NatF-simps*[*simp*]:
$0_f : \sigma s \to \tau$ *in* $NatF \longleftrightarrow \sigma s = [] \wedge \tau = nat_s$
$Suc_f : \sigma s \to \tau$ *in* $NatF \longleftrightarrow \sigma s = [nat_s] \wedge \tau = nat_s$
$+_f : \sigma s \to \tau$ *in* $NatF \longleftrightarrow (\forall \sigma \in set\ \sigma s.\ \sigma = nat_s) \wedge \tau = nat_s$

$*_f : \sigma s \to \tau$ *in NatF* $\longleftrightarrow (\forall \sigma \in$ *set* $\sigma s.\ \sigma = nat_s) \wedge \tau = nat_s$
$<_f : \sigma s \to \tau$ *in NatF* $\longleftrightarrow \sigma s = [nat_s, nat_s] \wedge \tau = bool_s$
⟨*proof*⟩

lemma *zero-t-hastype-in-NatF*[*simp*]: $0_t : \sigma$ *in* $\mathcal{T}(NatF, V) \longleftrightarrow \sigma = nat_s$
⟨*proof*⟩

lemma *Suc-t-hastype-in-NatF*[*simp*]:
$Suc_t\ s : \sigma$ *in* $\mathcal{T}(NatF, V) \longleftrightarrow \sigma = nat_s \wedge s : nat_s$ *in* $\mathcal{T}(NatF, V)$
⟨*proof*⟩

lemma *less-t-hastype-in-NatF*[*simp*]:
$(s <_t t) : \sigma$ *in* $\mathcal{T}(NatF, V) \longleftrightarrow$
$\sigma = bool_s \wedge s : nat_s$ *in* $\mathcal{T}(NatF, V) \wedge t : nat_s$ *in* $\mathcal{T}(NatF, V)$
⟨*proof*⟩

lemma *Fun-add-hastype-in-NatF*[*simp*]:
Fun $+_f$ *ss* $: \sigma$ *in* $\mathcal{T}(NatF, V) \longleftrightarrow$
$\sigma = nat_s \wedge (\forall s \in$ *set ss.* $s : nat_s$ *in* $\mathcal{T}(NatF, V))$
⟨*proof*⟩

lemma *Fun-mult-hastype-in-NatF*[*simp*]:
Fun $*_f$ *ss* $: \sigma$ *in* $\mathcal{T}(NatF, V) \longleftrightarrow$
$\sigma = nat_s \wedge (\forall s \in$ *set ss.* $s : nat_s$ *in* $\mathcal{T}(NatF, V))$
⟨*proof*⟩

lemma *NatC-BoolC*: *dom NatC* $\cap$ *dom Bool.C* $= \{\}$
⟨*proof*⟩

lemmas *hastype-in-NatBoolC* $=$ *fun-hastype-in-add-disj*[*OF NatC-BoolC*]

lemma *NatF-BoolF*: *dom NatF* $\cap$ *dom Bool.F* $= \{\}$
⟨*proof*⟩

lemmas *hastype-in-NatBoolF* $=$ *fun-hastype-in-add-disj*[*OF NatF-BoolF*]

fun *const-of-nat* **where**
const-of-nat 0 $= 0_t$
| *const-of-nat* (*Suc n*) $=$ Suc_t (*const-of-nat n*)

lemma *const-of-nat-hastype*: *const-of-nat n* $: nat_s$ *in* $\mathcal{T}(NatC, V)$
⟨*proof*⟩

abbreviation *NatBool-const* $\equiv$ *case-sum Bool.const-of-bool const-of-nat*

definition *NatA* (*a*::*nat*) $\equiv$ *Some* nat_s

abbreviation *NatBoolA* $\equiv$ *case-sum Bool.A NatA*

lemma *hastype-in-NatA*[*simp*]: $a : s \ in \ NatA \longleftrightarrow s = nat_s$
⟨*proof*⟩

lemma *hastype-in-NatBoolA*:
$a : \sigma \ in \ NatBoolA \longleftrightarrow (\sigma = bool_s \wedge a \in range \ Inl \vee \sigma = nat_s \wedge a \in range \ Inr)$
⟨*proof*⟩

definition *NatI f as* ≡
if $f = Suc_f$ *then Inr* (*Suc* (*projr* (*as*!*0*)))
else if $f = +_f$ *then Inr* ($\sum a \leftarrow as.\ projr\ a$)
else if $f = *_f$ *then Inr* ($\prod a \leftarrow as.\ projr\ a$)
else if $f = <_f$ *then Inl* (*projr* (*as*!*0*) < *projr* (*as*!*1*))
else Inr 0

abbreviation *NatBoolI* ≡ *logicalize-intp NatI*

lemma *NatI-simps*:
$NatI\ 0_f\ as = Inr\ 0$
$NatI\ Suc_f\ as = Inr$ (*Suc* (*projr* (*as*!*0*)))
$NatI\ +_f\ as = Inr$ ($\sum a \leftarrow as.\ projr\ a$)
$NatI\ *_f\ as = Inr$ ($\prod a \leftarrow as.\ projr\ a$)
$NatI\ <_f\ as = Inl$ (*projr* (*as*!*0*) < *projr* (*as*!*1*))
⟨*proof*⟩

interpretation *Nat*: *sorted-algebra NatF NatBoolA NatI*
⟨*proof*⟩

interpretation *Nat-const*: *sorted-algebra NatC NatBoolA NatI*
⟨*proof*⟩

lemma *Nat-eval-const-of-nat*[*simp*]: $NatI[\![const\text{-}of\text{-}nat\ n]\!]\alpha = Inr\ n$
⟨*proof*⟩

lemma *logicalize-intp-Nat*[*simp*]:
logicalize-intp I 0_f *as* = *I* 0_f *as*
logicalize-intp I Suc_f *as* = *I* Suc_f *as*
logicalize-intp I $+_f$ *as* = *I* $+_f$ *as*
logicalize-intp I $*_f$ *as* = *I* $*_f$ *as*
logicalize-intp I $<_f$ *as* = *I* $<_f$ *as*
⟨*proof*⟩

lemma *NatBool-eval-const-of-nat*: $NatBoolI[\![const\text{-}of\text{-}nat\ b]\!]\alpha = Inr\ b$ ⟨*proof*⟩

interpretation *NatBool*: *logic NatBoolF NatBoolA NatBoolI* $bool_s$ $conj_f$ $disj_f$ neg_f
⟨*proof*⟩

interpretation *NatBool*: *logic-const NatBoolF NatBoolA NatBoolI NatBool-const*

NatBoolC
bool$_s$ *conj*$_f$ *disj*$_f$ *neg*$_f$
⟨*proof*⟩

end

16.3 Integer Arithmetic

theory *Ints-Logic*
imports *Nats-Logic Show.Number-Parser*
begin

For integer arithmetic, we allow string representations of integers to be constants, i.e., those strings that can be parsed as integers.

lemma *isl-int-of-string-imp-show-neq*:
assumes *isl* (*int-of-string s*) **shows** *show* (*i*::*int*) ≠ *s*
⟨*proof*⟩

definition *int-s* (*int*$_s$) **where** *int*$_s$ ≡ ''*Int*''

lemma *bool-neq-int*[*simp*]: *bool*$_s$ ≠ *int*$_s$
⟨*proof*⟩
declare *bool-neq-int*[*symmetric*,*simp*]

lemma *isl-int-of-string-less*: *isl* (*int-of-string* <$_f$)
and *isl-int-of-string-add*: *isl* (*int-of-string* +$_f$)
and *isl-int-of-string-mult*: *isl* (*int-of-string* *$_f$)
and *isl-int-of-string-conj*: *isl* (*int-of-string conj*$_f$)
and *isl-int-of-string-disj*: *isl* (*int-of-string disj*$_f$)
⟨*proof*⟩

lemma *show-int-neq-less*[*simp*]: *show* (*i*::*int*) ≠ <$_f$
and *show-int-neq-add*[*simp*]: *show* (*i*::*int*) ≠ +$_f$
and *show-int-neq-mult*[*simp*]: *show* (*i*::*int*) ≠ *$_f$
and *show-int-neq-conj*[*simp*]: *show* (*i*::*int*) ≠ *conj*$_f$
and *show-int-neq-disj*[*simp*]: *show* (*i*::*int*) ≠ *disj*$_f$
⟨*proof*⟩

definition *IntC* ≡ λ(*s*,σ*s*).
if σ*s* = [] ∧ *int-of-string s* ∈ *range Inr then Some int*$_s$ *else None*

abbreviation *IntBoolC* ≡ *IntC* ++ *Bool.C*

definition *IntD* ≡ [(+$_f$,*int*$_s$..) ↦ *int*$_s$, (*$_f$,*int*$_s$..) ↦ *int*$_s$, (<$_f$,[*int*$_s$,*int*$_s$]) ↦ *bool*$_s$]

abbreviation *IntF* ≡ *IntC* ++ *IntD*

abbreviation *IntBoolF* ≡ *IntF* ++ *Bool.F*

lemma *hastype-in-IntC*:
$s : \sigma s \to \tau$ *in* *IntC* $\longleftrightarrow$ $\sigma s = [] \wedge \tau = int_s \wedge$ *int-of-string* $s \in$ *range Inr*
⟨*proof*⟩

lemma *IntC-BoolC*: *dom IntC* $\cap$ *dom Bool.C* $= \{\}$
⟨*proof*⟩

lemmas *hastype-in-IntBoolC* = *fun-hastype-in-add-disj*[*OF IntC-BoolC*]

lemma *hastype-in-IntD*: $f : \sigma s \to \tau$ *in* *IntD* $\longleftrightarrow$
$(f = +_f \vee f = *_f) \wedge (\forall \sigma \in set\ \sigma s.\ \sigma = int_s) \wedge \tau = int_s \vee$
$f = <_f \wedge \sigma s = [int_s, int_s] \wedge \tau = bool_s$
⟨*proof*⟩

lemma *IntC-IntD*: *dom IntC* $\cap$ *dom IntD* $= \{\}$
⟨*proof*⟩

lemmas *hastype-in-IntF* = *fun-hastype-in-add-disj*[*OF IntC-IntD*]

lemma *IntF-BoolF*: *dom IntF* $\cap$ *dom Bool.F* $= \{\}$
⟨*proof*⟩

lemmas *hastype-in-IntBoolF* = *fun-hastype-in-add-disj*[*OF IntF-BoolF*]

definition *IntA* (*i*::*int*) $\equiv$ *Some* int_s

abbreviation *IntBoolA* $\equiv$ *case-sum Bool.A IntA*

lemma *hastype-in-IntA*[*simp*]: $a : \sigma$ *in* *IntA* $\longleftrightarrow$ $\sigma = int_s$
⟨*proof*⟩

definition *IntI* :: *string* $\Rightarrow$ (*bool*+*int*) *list* $\Rightarrow$ *bool*+*int* **where** *IntI f as* $\equiv$
if $f = +_f$ *then Inr* $(\sum a \leftarrow as.\ projr\ a)$
else if $f = *_f$ *then Inr* $(\prod a \leftarrow as.\ projr\ a)$
else if $f = <_f$ *then Inl* (*projr* (*as*!*0*) < *projr* (*as*!*1*))
else case int-of-string f of Inr i $\Rightarrow$ *Inr i*

abbreviation *IntBoolI* $\equiv$ *logicalize-intp IntI*

lemma *IntI-simps*:
int-of-string s = *Inr i* $\Longrightarrow$ *IntI s as* = *Inr i*
IntI $+_f$ *as* = *Inr* $(\sum a \leftarrow as.\ projr\ a)$
IntI $*_f$ *as* = *Inr* $(\prod a \leftarrow as.\ projr\ a)$
IntI $<_f$ *as* = *Inl* (*projr* (*as*!*0*) < *projr* (*as*!*1*))
⟨*proof*⟩

interpretation *Int*: *sorted-algebra IntF IntBoolA IntI*
⟨*proof*⟩

interpretation *Int.const*: *sorted-algebra IntC IntBoolA IntI*
⟨*proof*⟩

abbreviation *const-of-int i* ≡ *Fun* (*show i*) []

abbreviation *IntBool-const* ≡ *case-sum Bool.const-of-bool const-of-int*

lemma *IntBoolI-int-of-string*:
assumes *int-of-string f* = *Inr i*
shows *IntBoolI f* [] = *Inr i*
⟨*proof*⟩

lemma *IntBoolI-show*[*simp*]:
IntBoolI (*show i*) [] = *Inr i*
⟨*proof*⟩

interpretation *IntBool*: *logic IntBoolF IntBoolA IntBoolI bool*$_s$ *conj*$_f$ *disj*$_f$ *neg*$_f$
⟨*proof*⟩

interpretation *IntBool*:
logic-const IntBoolF IntBoolA IntBoolI IntBool-const IntBoolC bool$_s$ *conj*$_f$ *disj*$_f$ *neg*$_f$
⟨*proof*⟩

end

17 Examples

theory *Test-Sorted-Rewriting*
imports *Conditional-Rewriting Constrained-Rewriting Ints-Logic*
begin

definition *min-f* (*min*$_f$) **where** *min*$_f$ ≡ ″*min*″
definition *even-f* (*even*$_f$) **where** *even*$_f$ ≡ ″*even*″

lemma *syms-neq*[*simp*]: *0*$_f$ ≠ *min*$_f$ *0*$_f$ ≠ *even*$_f$
Suc$_f$ ≠ *min*$_f$ *Suc*$_f$ ≠ *even*$_f$
+$_f$ ≠ *min*$_f$ +$_f$ ≠ *even*$_f$
*$_f$ ≠ *min*$_f$ *$_f$ ≠ *even*$_f$
<$_f$ ≠ *min*$_f$ <$_f$ ≠ *even*$_f$
min$_f$ ≠ *even*$_f$ *min*$_f$ ≠ *conj*$_f$ *min*$_f$ ≠ *disj*$_f$ *min*$_f$ ≠ *neg*$_f$
even$_f$ ≠ *conj*$_f$ *even*$_f$ ≠ *disj*$_f$ *even*$_f$ ≠ *neg*$_f$
⟨*proof*⟩

lemmas [*simp*] = *syms-neq*[*symmetric*]

17.1 Less-Than TRS

lemma
$(s +_t t) : nat_s \ in \ \mathcal{T}(NatF, V) \longleftrightarrow$
$s : nat_s \ in \ \mathcal{T}(NatF, V) \wedge t : nat_s \ in \ \mathcal{T}(NatF, V)$
$(s <_t t) : bool_s \ in \ \mathcal{T}(NatF, V) \longleftrightarrow$
$s : nat_s \ in \ \mathcal{T}(NatF, V) \wedge t : nat_s \ in \ \mathcal{T}(NatF, V)$
⟨*proof*⟩

abbreviation $lt0 \equiv [''x'' \mapsto nat_s].\ (Var\ ''x'' <_t 0_t) \rightsquigarrow False_t$

abbreviation $ltSuc \equiv [''y'' \mapsto nat_s].\ (0_t <_t Suc_t\ (Var\ ''y'')) \rightsquigarrow True_t$

abbreviation $ltSucSuc \equiv [''x'' \mapsto nat_s, ''y'' \mapsto nat_s].$
$(Suc_t\ (Var\ ''x'') <_t Suc_t\ (Var\ ''y'')) \rightsquigarrow (Var\ ''x'' <_t Var\ ''y'')$

definition $Rlt\ (\mathcal{R}_<)$ **where** $\mathcal{R}_< \equiv \{lt0, ltSuc, ltSucSuc\}$

lemma *Rlt*: $lt0 \in \mathcal{R}_<\ ltSuc \in \mathcal{R}_<\ ltSucSuc \in \mathcal{R}_<$
⟨*proof*⟩

interpretation *Rlt*: *sorted-trs NatBoolF* $\mathcal{R}_<$
⟨*proof*⟩

lemma $(0_t <_t 0_t) -NatBoolF: V: \mathcal{R}_< \rightarrow^{\varepsilon} False_t$
⟨*proof*⟩

lemma $(0_t <_t Suc_t\ 0_t) -NatBoolF: V: \mathcal{R}_< \rightarrow^{\varepsilon} True_t$
⟨*proof*⟩

17.2 Addition TRS

abbreviation $add0 \equiv [''x'' \mapsto nat_s].\ 0_t +_t Var\ ''x'' \rightsquigarrow Var\ ''x''$

abbreviation $addSuc \equiv$
$[''x'' \mapsto nat_s,\ ''y'' \mapsto nat_s].$
$Suc_t\ (Var\ ''x'') +_t Var\ ''y'' \rightsquigarrow Suc_t\ (Var\ ''x'' +_t Var\ ''y'')$

definition $Radd\ (\mathcal{R}_+)$ **where** $\mathcal{R}_+ \equiv \{add0, addSuc\}$

lemma *Radd*: $add0 \in \mathcal{R}_+\ addSuc \in \mathcal{R}_+$
⟨*proof*⟩

interpretation *Radd*: *sorted-trs NatF* $\mathcal{R}_+$
⟨*proof*⟩

lemma *Suc0-add*:
assumes *s*: $s : nat_s \ in \ \mathcal{T}(NatF, V)$
shows $Suc_t\ 0_t +_t s -NatF: V: \mathcal{R}_+ \rightarrow\ ^\frown 2\ Suc_t\ s$
⟨*proof*⟩

lemma *SucSuc0-add*:
assumes *s*: $s : nat_s$ *in* $\mathcal{T}(NatF, V)$
shows $Suc_t\ (Suc_t\ 0_t) +_t s -NatF{:}V{:}\mathcal{R}_+{\rightarrow}\hat{}3\ Suc_t\ (Suc_t\ s)$
⟨*proof*⟩

17.3 Even TRS

definition *Feven* ≡ [
$(conj_f,[]) \mapsto bool_s$,
$(disj_f,[]) \mapsto bool_s$,
$(0_f,[]) \mapsto nat_s$,
$(Suc_f,[nat_s]) \mapsto nat_s$,
$(even_f,[nat_s]) \mapsto bool_s$]

lemma *Feven*[*simp*]:
$conj_f : \sigma s \to \tau$ *in Feven* $\longleftrightarrow \sigma s = [] \wedge \tau = bool_s$
$disj_f : \sigma s \to \tau$ *in Feven* $\longleftrightarrow \sigma s = [] \wedge \tau = bool_s$
$0_f : \sigma s \to \tau$ *in Feven* $\longleftrightarrow \sigma s = [] \wedge \tau = nat_s$
$Suc_f : \sigma s \to \tau$ *in Feven* $\longleftrightarrow \sigma s = [nat_s] \wedge \tau = nat_s$
$even_f : \sigma s \to \tau$ *in Feven* $\longleftrightarrow \sigma s = [nat_s] \wedge \tau = bool_s$
⟨*proof*⟩

abbreviation *even-t* ($even_t$) **where** $even_t\ s \equiv Fun\ even_f\ [s]$

abbreviation *axiom-even0* ≡ ∅. $even_t\ 0_t \rightsquigarrow True_t$

abbreviation *axiom-even1* ≡ ∅. $even_t\ (Suc_t\ 0_t) \rightsquigarrow False_t$

abbreviation *axiom-evenSucSuc* ≡ [$''x'' \mapsto nat_s$].
$even_t\ (Suc_t\ (Suc_t\ (Var\ ''x'')))\ \rightsquigarrow\ even_t\ (Var\ ''x'')$

definition *Reven* ≡ {*axiom-even0*, *axiom-even1*, *axiom-evenSucSuc*}

lemma *Reven*: *axiom-even0* ∈ *Reven* *axiom-even1* ∈ *Reven* *axiom-evenSucSuc* ∈ *Reven*
⟨*proof*⟩

lemma *evenSucSucSuc*: $even_t\ (Suc_t\ (Suc_t\ (Suc_t\ 0_t))) -Feven{:}V{:}Reven{\rightarrow}\hat{}2\ False_t$
⟨*proof*⟩

17.4 Even CTRS

abbreviation *rule-evenSucFalse* ≡ [$''x'' \mapsto nat_s$].
$even_t\ (Suc_t\ (Var\ ''x''))\ \rightsquigarrow\ False_t \Leftarrow [even_t\ (Var\ ''x'')\ \rightsquigarrow\ True_t]$

abbreviation *rule-evenSucTrue* ≡ [$''x'' \mapsto nat_s$].
$even_t\ (Suc_t\ (Var\ ''x''))\ \rightsquigarrow\ True_t \Leftarrow [even_t\ (Var\ ''x'')\ \rightsquigarrow\ False_t]$

definition $RevenC \equiv \{unconditional\ axiom\text{-}even0,\ rule\text{-}evenSucFalse,\ rule\text{-}evenSucTrue\}$

lemma *RevenC*:
$unconditional\ axiom\text{-}even0 \in RevenC$
$rule\text{-}evenSucFalse \in RevenC$
$rule\text{-}evenSucTrue \in RevenC$
⟨*proof*⟩

interpretation *RevenC*: *sorted-ctrs Feven RevenC*
⟨*proof*⟩

lemma $even_t\ (Suc_t\ (Suc_t\ (Suc_t\ 0_t))) =_{Feven:V:RevenC} \Rightarrow^{\varepsilon}\ False_t$
⟨*proof*⟩

17.5 LCTRS

definition $Fmin \equiv NatBoolF\ ++\ [(min_f,[nat_s,nat_s]) \mapsto nat_s]$

lemma *Fmin-disj*: $dom\ (NatF\ ++\ Bool.F) \cap dom\ [(min_f,[nat_s,nat_s]) \mapsto nat_s] = \{\}$
⟨*proof*⟩

lemmas *hastype-in-Fmin* = *fun-hastype-in-add-disj*[*OF Fmin-disj*, *folded Fmin-def*]

lemma *min-hastype-in-Fmin*[*simp*]: $min_f : \sigma s \to \tau\ in\ Fmin \longleftrightarrow \sigma s = [nat_s,nat_s] \wedge \tau = nat_s$
⟨*proof*⟩

abbreviation *lcrule-min1* $\equiv [''x'' \mapsto nat_s,\ ''y'' \mapsto nat_s].$
$Fun\ min_f\ [Var\ ''x'',\ Var\ ''y''] \rightsquigarrow Var\ ''x'' \mid Var\ ''x'' <_t Var\ ''y''$

abbreviation *lcrule-min2* $\equiv [''x'' \mapsto nat_s,\ ''y'' \mapsto nat_s].$
$Fun\ min_f\ [Var\ ''x'',\ Var\ ''y''] \rightsquigarrow Var\ ''y'' \mid Var\ ''y'' <_t Var\ ''x''$

definition *lctrs-min* $\equiv \{lcrule\text{-}min1,\ lcrule\text{-}min2\}$

lemma *lctrs-min*: *lcrule-min1* $\in$ *lctrs-min lcrule-min2* $\in$ *lctrs-min*
⟨*proof*⟩

Deriving calculation TRS.

interpretation *NatBool*: *sorted-algebra-calculation NatBoolF NatBoolA NatBoolI NatBool-const NatBoolC*⟨*proof*⟩

interpretation *Fmin*: *lcrule-syntax Fmin NatBoolF* $bool_s$⟨*proof*⟩

interpretation *lctrs-min*: *lctrs-semantics Fmin lctrs-min NatBoolF NatBoolA NatBoolI NatBool-const NatBoolC* $bool_s$ $conj_f$
⟨*proof*⟩

lemma $Fun\ min_f\ [Suc_t\ 0_t,\ 0_t]\ -Fmin{:}V{:}lctrs\text{-}min.as\text{-}trs\rightarrow\ 0_t$
⟨*proof*⟩

Demonstrating a calculation step:

lemma $Suc_t\ 0_t\ +_t\ 0_t\ -Fmin{:}V{:}NatBool.calculation\text{-}trs\rightarrow\ Suc_t\ 0_t$
⟨*proof*⟩

end

References

[1] J. C. Blanchette, A. Popescu, and D. Traytel. Abstract completeness. *Archive of Formal Proofs*, April 2014. https://isa-afp.org/entries/Abstract_Completeness.html, Formal proof development.

[2] C. Kop and N. Nishida. Term rewriting with logical constraints. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2013.

[3] C. Sternagel and R. Thiemann. First-order terms. *Archive of Formal Proofs*, February 2018. https://isa-afp.org/entries/First_Order_Terms.html, Formal proof development.

[4] R. Thiemann and C. Sternagel. Certification of Termination Proofs Using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.

[5] V. van Oostrom. Sub-birkhoff. In Y. Kameyama and P. J. Stuckey, editors, *7th International Symposium on Functional and Logic Programming, FLOPS 2004*, volume 2998 of *LNCS*, pages 180–195. Springer, 2004.

[6] A. Yamada and R. Thiemann. Sorted terms. *Archive of Formal Proofs*, May 2024. https://isa-afp.org/entries/Sorted_Terms.html, Formal proof development.