Type Annotations with Roundtrip Property

Kevin Kappelmann Maximilian Schäffeler Lukas Stevens
Mohammad Abdulaziz Andrei Popescu Dmitriy Traytel

June 4, 2026

Abstract

Type annotations are essential when printing terms in a way that preserves their meaning under reparsing and type inference. We study the problem of complete and minimal type annotations for rank-one polymorphic λ-calculus terms, as used in Isabelle. Building on prior work by Smolka, Blanchette et al., we show that a reverse-greedy approach leads to a locally minimal and complete set of annotations. Our development is a series of experiments featuring human-driven and AI-driven formalization workflows: a human and an AI agent independently produce pen-and-paper proofs, and the agent autoformalizes them in Isabelle. We refined the resulting AI proof developments for this AFP submission, the original version and our experimental setup can be found in the related Zenodo entry. We give a metatheoretical account of the problem, with a full formal specification and proofs in our paper "Just Type It in Isabelle! AI Agents Drafting, Mechanizing, and Generalizing from Human Hints".

Contents

theory *Smolka*
  imports *Main*
begin

1 Human-authored proof formalization

Formalization of the correctness proof of the Smolka-Blanchette algorithm for minimal type annotation of terms.

The algorithm takes a Church-typed term and removes type annotations while preserving unique type-reconstruction, achieving both completeness (unique well-typed completion) and local minimality (removing any further annotation breaks uniqueness).

1.1 Types

Corresponds to Subsection 1.1 of the paper. We fix an infinite set of type variables and define types as: $\sigma ::= \alpha | \sigma \Rightarrow \tau | (\sigma_1, ..., \sigma_n)\kappa$ We represent this as a datatype with *Arrow* as a distinguished binary constructor and *TyCon* for n-ary type constructors.

datatype ('*tv*, '*k*) *ty* =
  TyVar '*tv*
| *Arrow* ('*tv*, '*k*) *ty* ('*tv*, '*k*) *ty*
| *TyCon* '*k* ('*tv*, '*k*) *ty list*

Maybe-types: ('*tv*, '*k*) *ty option* where *None* represents $\bot$ (no type annotation) and *Some* σ represents an actual type.

type-synonym ('*tv*, '*k*) *mty* = ('*tv*, '*k*) *ty option*

1.1.1 Type variables of a type

tvars-ty: the set of type variables occurring in a type. Extended to maybe-types with *tvars-mty None* = {}.

fun *tvars-ty* :: ('*tv*, '*k*) *ty* ⇒ '*tv set* **where**
  tvars-ty (*TyVar a*) = {*a*}
| *tvars-ty* (*Arrow* σ τ) = *tvars-ty* σ ∪ *tvars-ty* τ
| *tvars-ty* (*TyCon* κ σs) = ⋃(*tvars-ty* ' *set* σs)

definition *tvars-mty* :: ('*tv*, '*k*) *mty* ⇒ '*tv set* **where**
  tvars-mty ξ = (*case* ξ *of None* ⇒ {} | *Some* σ ⇒ *tvars-ty* σ)

lemma *tvars-mty-None*[*simp*]: *tvars-mty None* = {}
  by (*simp add*: *tvars-mty-def*)

lemma *tvars-mty-Some*[*simp*]: *tvars-mty* (*Some* σ) = *tvars-ty* σ
  by (*simp add*: *tvars-mty-def*)

lemma *tvars-ty-finite*[*simp*, *intro*]: *finite* (*tvars-ty* τ)
by (*induction* τ) *auto*

lemma *tvars-mty-finite*[*simp*, *intro*]: *finite* (*tvars-mty* ξ)
by (*cases* ξ) *auto*

1.1.2 Type substitution

A type substitution is a function from type variables to types. *subst-ty* ϱ σ applies ϱ to σ. Extended to maybe-types with *subst-mty* ϱ *None* = *None*.

fun *subst-ty* :: ('*tv* ⇒ ('*tv*, '*k*) *ty*) ⇒ ('*tv*, '*k*) *ty* ⇒ ('*tv*, '*k*) *ty* **where**
subst-ty ϱ (*TyVar a*) = ϱ *a*
| *subst-ty* ϱ (*Arrow* σ τ) = *Arrow* (*subst-ty* ϱ σ) (*subst-ty* ϱ τ)
| *subst-ty* ϱ (*TyCon* κ σs) = *TyCon* κ (*map* (*subst-ty* ϱ) σs)

definition *subst-mty* :: ('*tv* ⇒ ('*tv*, '*k*) *ty*) ⇒ ('*tv*, '*k*) *mty* ⇒ ('*tv*, '*k*) *mty* **where**
subst-mty ϱ ξ = *map-option* (*subst-ty* ϱ) ξ

lemma *subst-mty-None*[*simp*]: *subst-mty* ϱ *None* = *None*
by (*simp add*: *subst-mty-def*)

lemma *subst-mty-Some*[*simp*]: *subst-mty* ϱ (*Some* σ) = *Some* (*subst-ty* ϱ σ)
by (*simp add*: *subst-mty-def*)

1.1.3 Substitution composition and single substitution

definition *comp-subst* ::
('*tv* ⇒ ('*tv*,'*k*) *ty*) ⇒ ('*tv* ⇒ ('*tv*,'*k*) *ty*) ⇒ ('*tv* ⇒ ('*tv*,'*k*) *ty*) (**infixl** ∘∘ *55*)
where
(ϱ ∘∘ ϱ') *a* = *subst-ty* ϱ (ϱ' *a*)

Composition ϱ ∘∘ ϱ' is defined by (ϱ ∘∘ ϱ') *a* = *subst-ty* ϱ (ϱ' *a*). Single substitution *single-subst* τ α maps α to τ and everything else to itself.

definition *single-subst* :: ('*tv*,'*k*) *ty* ⇒ '*tv* ⇒ ('*tv* ⇒ ('*tv*,'*k*) *ty*) **where**
single-subst τ α = (λ*a*. *if a* = α *then* τ *else TyVar a*)

lemma *single-subst-same*[*simp*]: *single-subst* τ α α = τ
by (*simp add*: *single-subst-def*)

lemma *single-subst-other*[*simp*]: *a* ≠ α ⟹ *single-subst* τ α *a* = *TyVar a*
by (*simp add*: *single-subst-def*)

1.1.4 Basic properties of substitution

lemma *subst-ty-comp*:
subst-ty ϱ (*subst-ty* ϱ' τ) = *subst-ty* (ϱ ∘∘ ϱ') τ
by (*induction* τ) (*auto simp*: *comp-subst-def*)

lemma *subst-ty-id*[*simp*]: *subst-ty TyVar* $\tau = \tau$
by (*induction* τ) (*auto simp*: *map-idI*)

lemma *subst-mty-id*[*simp*]: *subst-mty TyVar* $\xi = \xi$
by (*cases* ξ) (*auto simp*: *subst-mty-def*)

lemma *subst-mty-comp*: *subst-mty* ϱ (*subst-mty* ϱ' ξ) = *subst-mty* ($\varrho \circ\circ \varrho'$) ξ
by (*cases* ξ) (*auto simp*: *subst-mty-def subst-ty-comp comp-subst-def*)

lemma *subst-ty-agree*:
subst-ty ϱ τ = *subst-ty* ϱ' τ $\longleftrightarrow$ ($\forall \alpha \in$ *tvars-ty* τ. ϱ α = ϱ' α)
by (*induction* τ) *auto*

lemma *subst-ty-cong*:
($\bigwedge\alpha$. $\alpha \in$ *tvars-ty* $\tau \Longrightarrow \varrho$ α = ϱ' α) $\Longrightarrow$ *subst-ty* ϱ τ = *subst-ty* ϱ' τ
using *subst-ty-agree* **by** *blast*

lemma *subst-ty-swap*:
assumes $\beta \notin$ *tvars-ty* τ
shows *subst-ty* (*single-subst* (*TyVar* α) β) (*subst-ty* (*single-subst* (*TyVar* β) α) τ) = τ
using *assms*
by (*induction* τ) (*auto simp*: *map-idI single-subst-def*)

Substitution distributes over *tvars-ty*.

lemma *tvars-ty-subst*: *tvars-ty* (*subst-ty* ϱ τ) = $\bigcup$(*tvars-ty* ‘ ϱ ‘ *tvars-ty* τ)
by (*induction* τ) *auto*

lemma *tvars-mty-subst*:
tvars-mty (*subst-mty* ϱ ξ) = $\bigcup$(*tvars-ty* ‘ ϱ ‘ *tvars-mty* ξ)
by (*cases* ξ) (*auto simp*: *tvars-ty-subst*)

1.2 Terms

Partially-typed terms: $t ::= x_\xi | c_\xi | (t1\ t2)_\xi | (\lambda x_\xi.t)_\zeta$ We do not quotient by alpha-equivalence.

datatype (*'tv*, *'k*, *'v*, *'c*) *trm* =
Vr 'v (*'tv*,*'k*) *mty*
| *Ct 'c* (*'tv*,*'k*) *mty*
| *App* (*'tv*,*'k*,*'v*,*'c*) *trm* (*'tv*,*'k*,*'v*,*'c*) *trm* (*'tv*,*'k*) *mty*
| *Lam 'v* (*'tv*,*'k*) *mty* (*'tv*,*'k*,*'v*,*'c*) *trm* (*'tv*,*'k*) *mty*

1.2.1 Type variables of a term

tvars-trm: set of type variables occurring in all annotations of a term.

fun *tvars-trm* :: (*'tv*,*'k*,*'v*,*'c*) *trm* ⇒ *'tv set* **where**
tvars-trm (*Vr x* ξ) = *tvars-mty* ξ
| *tvars-trm* (*Ct c* ξ) = *tvars-mty* ξ

| *tvars-trm* (*App t1 t2* ξ) = *tvars-trm t1* $\cup$ *tvars-trm t2* $\cup$ *tvars-mty* ξ
| *tvars-trm* (*Lam x* ξ *t* ζ) = *tvars-mty* ξ $\cup$ *tvars-trm t* $\cup$ *tvars-mty* ζ

lemma *tvars-trm-finite*[*simp*, *intro*]: *finite* (*tvars-trm t*)
by (*induction t*) *auto*

1.2.2 Substitution on terms

subst-trm ϱ *t*: apply type substitution ϱ to all type annotations in *t*.

fun *subst-trm* :: ('*tv* $\Rightarrow$ ('*tv*,'*k*) *ty*) $\Rightarrow$ ('*tv*,'*k*,'*v*,'*c*) *trm* $\Rightarrow$ ('*tv*,'*k*,'*v*,'*c*) *trm* **where**
subst-trm ϱ (*Vr x* ξ) = *Vr x* (*subst-mty* ϱ ξ)
| *subst-trm* ϱ (*Ct c* ξ) = *Ct c* (*subst-mty* ϱ ξ)
| *subst-trm* ϱ (*App t1 t2* ξ) = *App* (*subst-trm* ϱ *t1*) (*subst-trm* ϱ *t2*) (*subst-mty* ϱ ξ)
| *subst-trm* ϱ (*Lam x* ξ *t* ζ) = *Lam x* (*subst-mty* ϱ ξ) (*subst-trm* ϱ *t*) (*subst-mty* ϱ ζ)

lemma *subst-trm-comp*:
subst-trm ϱ (*subst-trm* ϱ' *t*) = *subst-trm* (ϱ $\circ\circ$ ϱ') *t*
by (*induction t*) (*auto simp*: *subst-mty-comp*)

lemma *subst-trm-id*[*simp*]: *subst-trm TyVar t* = *t*
by (*induction t*) *auto*

lemma *subst-trm-cong*:
($\bigwedge\alpha$. α $\in$ *tvars-trm t* $\Longrightarrow$ ϱ α = ϱ' α) $\Longrightarrow$ *subst-trm* ϱ *t* = *subst-trm* ϱ' *t*
proof (*induction t*)
case (*Vr x* ξ)
then show *?case* **by** (*cases* ξ) (*auto intro*: *subst-ty-cong*)
next
case (*Ct c* ξ)
then show *?case* **by** (*cases* ξ) (*auto intro*: *subst-ty-cong*)
next
case (*App t1 t2* ξ)
then show *?case* **by** (*cases* ξ) (*auto cong*: *subst-ty-cong*)
next
case (*Lam x* ξ *t* ζ)
then show *?case* **by** (*cases* ξ; *cases* ζ) (*auto intro*: *subst-ty-cong*)
qed

lemma *subst-trm-agree*:
subst-trm ϱ *t* = *subst-trm* ϱ' *t* $\longleftrightarrow$ ($\forall\,\alpha \in$ *tvars-trm t*. ϱ α = ϱ' α)
proof (*induction t*)
case (*Vr x* ξ)
then show *?case* **by** (*cases* ξ) (*auto simp*: *subst-ty-agree*)
next
case (*Ct c* ξ)
then show *?case* **by** (*cases* ξ) (*auto simp*: *subst-ty-agree*)
next

case (*App t1 t2* ξ)
then show *?case* **by** (*cases* ξ) (*auto simp*: *subst-ty-agree*)
next
case (*Lam x* ξ *t* ζ)
then show *?case* **by** (*cases* ξ; *cases* ζ) (*auto simp*: *subst-ty-agree*)
qed

If *subst-trm* ϱ *v* = *v*, then ϱ acts as identity on *tvars-trm v*.

lemma *subst-trm-id-on-tvars*:
assumes *subst-trm* ϱ *v* = *v* $\alpha \in$ *tvars-trm v*
shows ϱ α = *TyVar* α
proof −
from *assms*(*1*) **have** *subst-trm* ϱ *v* = *subst-trm TyVar v* **by** *simp*
then have $\forall \alpha \in$ *tvars-trm v*. ϱ α = *TyVar* α **by** (*simp only*: *subst-trm-agree*)
with *assms*(*2*) **show** *?thesis* **by** *blast*
qed

lemma *subst-trm-swap*:
assumes $\beta \notin$ *tvars-trm t*
shows *subst-trm* (*single-subst* (*TyVar* α) β) (*subst-trm* (*single-subst* (*TyVar* β) α) *t*) = *t*
using *assms*
proof (*induction t*)
case (*Vr x* ξ)
then show *?case* **by** (*cases* ξ) (*auto simp*: *subst-ty-swap*)
next
case (*Ct c* ξ)
then show *?case* **by** (*cases* ξ) (*auto simp*: *subst-ty-swap*)
next
case (*App t1 t2* ξ)
then show *?case* **by** (*cases* ξ) (*auto simp*: *subst-ty-swap*)
next
case (*Lam x* ξ *t* ζ)
then show *?case* **by** (*cases* ξ; *cases* ζ) (*auto simp*: *subst-ty-swap*)
qed

lemma *tvars-trm-subst*:
tvars-trm (*subst-trm* ϱ *t*) = $\bigcup$(*tvars-ty* ‘ ϱ ‘ *tvars-trm t*)
by (*induction t*) (*auto simp*: *tvars-mty-subst*)

1.2.3 Term predicates: F-term, U-term, C-term

F-term: all annotations are types *None*. U-term: all annotations are *None*. C-term: variable and constant annotations are types, application and abstraction outer annotations are *None*, and binding variable annotations are types.

fun *fterm* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *bool* **where**
fterm (*Vr x* ξ) ⟷ $\xi \neq$ *None*

| *fterm* (*Ct c* ξ) $\longleftrightarrow$ $\xi \neq$ *None*
| *fterm* (*App t1 t2* ξ) $\longleftrightarrow$ *fterm t1* $\wedge$ *fterm t2* $\wedge$ $\xi \neq$ *None*
| *fterm* (*Lam x* ξ *t* ζ) $\longleftrightarrow$ $\xi \neq$ *None* $\wedge$ *fterm t* $\wedge$ $\zeta \neq$ *None*

fun *uterm* :: ('*tv*,'*k*,'*v*,'*c*) *trm* $\Rightarrow$ *bool* **where**
  uterm (*Vr x* ξ) $\longleftrightarrow$ ξ = *None*
| *uterm* (*Ct c* ξ) $\longleftrightarrow$ ξ = *None*
| *uterm* (*App t1 t2* ξ) $\longleftrightarrow$ *uterm t1* $\wedge$ *uterm t2* $\wedge$ ξ = *None*
| *uterm* (*Lam x* ξ *t* ζ) $\longleftrightarrow$ ξ = *None* $\wedge$ *uterm t* $\wedge$ ζ = *None*

fun *cterm* :: ('*tv*,'*k*,'*v*,'*c*) *trm* $\Rightarrow$ *bool* **where**
  cterm (*Vr x* ξ) $\longleftrightarrow$ $\xi \neq$ *None*
| *cterm* (*Ct c* ξ) $\longleftrightarrow$ $\xi \neq$ *None*
| *cterm* (*App t1 t2* ξ) $\longleftrightarrow$ *cterm t1* $\wedge$ *cterm t2* $\wedge$ ξ = *None*
| *cterm* (*Lam x* ξ *t* ζ) $\longleftrightarrow$ $\xi \neq$ *None* $\wedge$ *cterm t* $\wedge$ ζ = *None*

lemma *fterm-subst*[*simp*]: *fterm v* $\Longrightarrow$ *fterm* (*subst-trm* ϱ *v*)
  by (*induction v*) *auto*

1.2.4 Unambiguity

A term is unambiguous if no variable appears as a binding variable at two different positions. We define the set of binding variables and then unambiguity.

fun *bvars* :: ('*tv*,'*k*,'*v*,'*c*) *trm* $\Rightarrow$ '*v set* **where**
  bvars (*Vr x* ξ) = {}
| *bvars* (*Ct c* ξ) = {}
| *bvars* (*App t1 t2* ξ) = *bvars t1* $\cup$ *bvars t2*
| *bvars* (*Lam x* ξ *t* ζ) = {*x*} $\cup$ *bvars t*

lemma *subst-trm-bvars*[*simp*]: *bvars* (*subst-trm* ϱ *t*) = *bvars t*
  by (*induction t*) *auto*

fun *unambiguous* :: ('*tv*,'*k*,'*v*,'*c*) *trm* $\Rightarrow$ *bool* **where**
  unambiguous (*Vr x* ξ) $\longleftrightarrow$ *True*
| *unambiguous* (*Ct c* ξ) $\longleftrightarrow$ *True*
| *unambiguous* (*App t1 t2* ξ) $\longleftrightarrow$
    unambiguous t1 $\wedge$ *unambiguous t2* $\wedge$ *bvars t1* $\cap$ *bvars t2* = {}
| *unambiguous* (*Lam x* ξ *t* ζ) $\longleftrightarrow$
    unambiguous t $\wedge$ *x* $\notin$ *bvars t*

lemma *subst-trm-unambiguous*[*simp*]: *unambiguous* (*subst-trm* ϱ *t*) = *unambiguous t*
  by (*induction t*) *auto*

1.3 Positions

Positions are lists of naturals in $\{1, 2::'a\}$. *poss t*: the set of positions of a term. *mtp-of t p*: the maybe-type at position *p* in *t*.

fun *poss* :: $('tv,'k,'v,'c)$ *trm* ⇒ *nat list set* **where**
  poss (*Vr x ξ*) = {[]}
| *poss* (*Ct c ξ*) = {[]}
| *poss* (*App t1 t2 ξ*) = {[]} ∪ ((λ*p*. *1* # *p*) ‘ *poss t1*) ∪ ((λ*p*. *2* # *p*) ‘ *poss t2*)
| *poss* (*Lam x ξ t ζ*) = {[]} ∪ {[*1*]} ∪ ((λ*p*. *2* # *p*) ‘ *poss t*)

lemma *poss-finite*[*simp*, *intro*]: *finite* (*poss t*)
  by (*induction t*) *auto*

lemma *nil-in-poss*[*simp*]: [] ∈ *poss t*
  by (*cases t*) *auto*

lemma *subst-trm-poss*[*simp*]: *poss* (*subst-trm ϱ t*) = *poss t*
  by (*induction t*) *auto*

fun *mtp-of* :: $('tv,'k,'v,'c)$ *trm* ⇒ *nat list* ⇒ $('tv,'k)$ *mty* **where**
  mtp-of (*Vr x ξ*) *p* = *ξ*
| *mtp-of* (*Ct c ξ*) *p* = *ξ*
| *mtp-of* (*App t1 t2 ξ*) *p* = (*case p of*
    [] ⇒ *ξ*
  | (*Suc 0* # *p'*) ⇒ *mtp-of t1 p'*
  | (*Suc* (*Suc 0*) # *p'*) ⇒ *mtp-of t2 p'*
  | - ⇒ *None*)
| *mtp-of* (*Lam x ξ t ζ*) *p* = (*case p of*
    [] ⇒ *ζ*
  | [*Suc 0*] ⇒ *ξ*
  | (*Suc* (*Suc 0*) # *p'*) ⇒ *mtp-of t p'*
  | - ⇒ *None*)

mtpOf t is the maybe-type of the root of *t*.

abbreviation *mtp-of-root* :: $('tv,'k,'v,'c)$ *trm* ⇒ $('tv,'k)$ *mty* **where**
  mtp-of-root t ≡ *mtp-of t* []

lemma *mtp-of-subst*:
  assumes *p* ∈ *poss t*
  shows *mtp-of* (*subst-trm ϱ t*) *p* = *subst-mty ϱ* (*mtp-of t p*)
  using *assms* **by** (*induction t arbitrary*: *p*) (*auto split*: *list.splits nat.splits*)

lemma *tvars-trm-eq-UN-mtp*:
  tvars-trm t = (⋃ *p* ∈ *poss t*. *tvars-mty* (*mtp-of t p*))
  by (*induction t*) (*auto simp*: *image-iff*)

lemma *fterm-mtp-of-Some*:
  assumes *fterm v p* ∈ *poss v*
  shows ∃*σ*. *mtp-of v p* = *Some σ*

using *assms* **by** (*induction v arbitrary*: *p*) (*auto split*: *list.splits nat.splits*)

1.3.1 Removing annotations

fun *remove-annot* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *nat list* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* **where**
  remove-annot (*Vr x ξ*) *p* = *Vr x None*
| *remove-annot* (*Ct c ξ*) *p* = *Ct c None*
| *remove-annot* (*App t1 t2 ξ*) *p* = (*case p of*
    [] ⇒ *App t1 t2 None*
  | *Suc 0* # *p'* ⇒ *App* (*remove-annot t1 p'*) *t2 ξ*
  | *Suc* (*Suc 0*) # *p'* ⇒ *App t1* (*remove-annot t2 p'*) *ξ*
  | - ⇒ *App t1 t2 None*)
| *remove-annot* (*Lam x ξ t ζ*) *p* = (*case p of*
    [] ⇒ *Lam x ξ t None*
  | [*Suc 0*] ⇒ *Lam x None t ζ*
  | *Suc* (*Suc 0*) # *p'* ⇒ *Lam x ξ* (*remove-annot t p'*) *ζ*
  | - ⇒ *Lam x ξ t None*)

lemma *mtp-of-remove-self*:
  assumes *p* ∈ *poss t*
  shows *mtp-of* (*remove-annot t p*) *p* = *None*
  using *assms* **by** (*induction t arbitrary*: *p*) *auto*

lemma *mtp-of-remove-other*:
  assumes *p* ∈ *poss t q* ∈ *poss t q* ≠ *p*
  shows *mtp-of* (*remove-annot t p*) *q* = *mtp-of t q*
  using *assms*
  by (*induction t arbitrary*: *p q*) *fastforce+*

lemma *remove-annot-poss*:
  assumes *p* ∈ *poss t*
  shows *poss* (*remove-annot t p*) = *poss t*
  using *assms* **by** (*induction t arbitrary*: *p*) *auto*

lemma *remove-annot-bvars*:
  assumes *p* ∈ *poss t*
  shows *bvars* (*remove-annot t p*) = *bvars t*
  using *assms* **by** (*induction t arbitrary*: *p*) *auto*

lemma *remove-annot-unambiguous*:
  assumes *p* ∈ *poss t unambiguous t*
  shows *unambiguous* (*remove-annot t p*)
  using *assms* **by** (*induction t arbitrary*: *p*) (*auto simp*: *remove-annot-bvars*)

Erase: remove all type annotations.

fun *erase* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* **where**
  erase (*Vr x ξ*) = *Vr x None*
| *erase* (*Ct c ξ*) = *Ct c None*
| *erase* (*App t1 t2 ξ*) = *App* (*erase t1*) (*erase t2*) *None*

| *erase* (*Lam x* ξ *t* ζ) = *Lam x None* (*erase t*) *None*

lemma *erase-uterm*[*simp*]: *uterm* (*erase t*)
by (*induction t*) *auto*

lemma *erase-poss*[*simp*]: *poss* (*erase t*) = *poss t*
by (*induction t*) *auto*

lemma *erase-bvars*[*simp*]: *bvars* (*erase t*) = *bvars t*
by (*induction t*) *auto*

lemma *erase-unambiguous*[*simp*]: *unambiguous* (*erase t*) = *unambiguous t*
by (*induction t*) *auto*

1.4 Completion relations for types and terms

definition *mcompl* :: ('*tv*,'*k*) *mty* ⇒ ('*tv*,'*k*) *mty* ⇒ *bool* (**infix** $\sqsubseteq_m$ *50*) **where**
$\xi \sqsubseteq_m \zeta \longleftrightarrow (\xi = None \vee \xi = \zeta)$

lemma *mcompl-None*[*simp*, *intro*]: *None* $\sqsubseteq_m$ ζ
by (*simp add*: *mcompl-def*)

lemma *mcompl-refl*[*simp*, *intro*]: $\xi \sqsubseteq_m \xi$
by (*simp add*: *mcompl-def*)

lemma *mcompl-Some-iff*[*simp*]: *Some* $\sigma \sqsubseteq_m \zeta \longleftrightarrow \zeta$ = *Some* σ
by (*auto simp*: *mcompl-def*)

lemma *mcompl-antisym*: $[\![\xi \sqsubseteq_m \zeta; \zeta \sqsubseteq_m \xi]\!] \Longrightarrow \xi = \zeta$
by (*auto simp*: *mcompl-def*)

lemma *mcompl-trans*: $[\![\xi \sqsubseteq_m \zeta; \zeta \sqsubseteq_m \chi]\!] \Longrightarrow \xi \sqsubseteq_m \chi$
by (*auto simp*: *mcompl-def*)

inductive *compl* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *bool* (**infix** $\sqsubseteq$ *50*) **where**
compl-Vr: $\xi \sqsubseteq_m \zeta \Longrightarrow Vr\ x\ \xi \sqsubseteq Vr\ x\ \zeta$
| *compl-Ct*: $\xi \sqsubseteq_m \zeta \Longrightarrow Ct\ c\ \xi \sqsubseteq Ct\ c\ \zeta$
| *compl-App*: $[\![s1 \sqsubseteq s1'; s2 \sqsubseteq s2'; \xi \sqsubseteq_m \xi']\!] \Longrightarrow App\ s1\ s2\ \xi \sqsubseteq App\ s1'\ s2'\ \xi'$
| *compl-Lam*: $[\![\zeta \sqsubseteq_m \zeta'; t \sqsubseteq t'; \xi \sqsubseteq_m \xi']\!] \Longrightarrow Lam\ x\ \zeta\ t\ \xi \sqsubseteq Lam\ x\ \zeta'\ t'\ \xi'$

1.4.1 Inversion rules

Inversion rules for ($\sqsubseteq$). These follow from the cases rule.

lemma *compl-Vr-leftE*:
assumes *Vr x* $\xi \sqsubseteq t$
obtains ξ' **where** $\xi \sqsubseteq_m \xi'$ *t* = *Vr x* ξ'
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-Ct-leftE*:
assumes $Ct\ c\ \xi \sqsubseteq t$
obtains ξ' **where** $\xi \sqsubseteq_m \xi'\ t = Ct\ c\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-App-leftE*:
assumes $App\ s1\ s2\ \xi \sqsubseteq t$
obtains $s1'\ s2'\ \xi'$ **where** $s1 \sqsubseteq s1'\ s2 \sqsubseteq s2'\ \xi \sqsubseteq_m \xi'\ t = App\ s1'\ s2'\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-Lam-leftE*:
assumes $Lam\ x\ \zeta\ s\ \xi \sqsubseteq t$
obtains $s'\ \zeta'\ \xi'$ **where** $s \sqsubseteq s'\ \zeta \sqsubseteq_m \zeta'\ \xi \sqsubseteq_m \xi'\ t = Lam\ x\ \zeta'\ s'\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-Vr-rightE*:
assumes $t \sqsubseteq Vr\ x\ \xi$
obtains ξ' **where** $\xi' \sqsubseteq_m \xi\ t = Vr\ x\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-Ct-rightE*:
assumes $t \sqsubseteq Ct\ c\ \xi$
obtains ξ' **where** $\xi' \sqsubseteq_m \xi\ t = Ct\ c\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-App-rightE*:
assumes $t \sqsubseteq App\ s1\ s2\ \xi$
obtains $s1'\ s2'\ \xi'$ **where** $s1' \sqsubseteq s1\ s2' \sqsubseteq s2\ \xi' \sqsubseteq_m \xi\ t = App\ s1'\ s2'\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

lemma *compl-Lam-rightE*:
assumes $t \sqsubseteq Lam\ x\ \zeta\ s\ \xi$
obtains $s'\ \zeta'\ \xi'$ **where** $s' \sqsubseteq s\ \zeta' \sqsubseteq_m \zeta\ \xi' \sqsubseteq_m \xi\ t = Lam\ x\ \zeta'\ s'\ \xi'$
using *assms* **by** (*cases rule*: *compl.cases*) *auto*

1.4.2 Basic properties

lemma *compl-refl*[*simp*, *intro*]: $t \sqsubseteq t$
by (*induction t*) (*auto intro*: *compl.intros*)

lemma *compl-poss-eq*: $t \sqsubseteq s \Longrightarrow poss\ t = poss\ s$
by (*induction t s rule*: *compl.induct*) *auto*

lemma *compl-bvars-eq*: $t \sqsubseteq s \Longrightarrow bvars\ t = bvars\ s$
by (*induction t s rule*: *compl.induct*) *auto*

lemma *compl-unambiguous*: $t \sqsubseteq s \Longrightarrow unambiguous\ t = unambiguous\ s$
by (*induction t s rule*: *compl.induct*) (*auto simp*: *compl-bvars-eq*)

lemma *erase-compl*[*simp*, *intro*]: *erase t* $\sqsubseteq$ *t*
by (*induction t*) (*auto intro*: *compl.intros*)

lemma *remove-annot-compl*:
assumes *p* $\in$ *poss t*
shows *remove-annot t p* $\sqsubseteq$ *t*
using *assms* **by** (*induction t arbitrary*: *p*) (*auto split*: *list.splits nat.splits intro*: *compl.intros*)

lemma *compl-trans*: $[\![s \sqsubseteq t; t \sqsubseteq u]\!] \Longrightarrow s \sqsubseteq u$
by (*induction s t arbitrary*: *u rule*: *compl.induct*)
(*auto elim*!: *compl-Vr-leftE compl-Ct-leftE compl-App-leftE compl-Lam-leftE intro*: *compl.intros dest*: *mcompl-trans*)

lemma *compl-antisym*: $[\![s \sqsubseteq t; t \sqsubseteq s]\!] \Longrightarrow s = t$
by (*induction s t rule*: *compl.induct*) (*auto elim*!: *compl-Vr-rightE compl-Ct-rightE compl-App-rightE compl-Lam-rightE dest*: *mcompl-antisym*)

lemma *compl-mtp-of-root*: *t* $\sqsubseteq$ *s* $\Longrightarrow$ *mtp-of t* [] $\sqsubseteq_m$ *mtp-of s* []
by (*erule compl.cases*) *auto*

1.4.3 Positions and completions

lemma *compl-mtp-of*:
assumes *t* $\sqsubseteq$ *s*
shows *mtp-of t p* $\sqsubseteq_m$ *mtp-of s p*
using *assms*
by (*induction t s arbitrary*: *p rule*: *compl.induct*) (*auto split*: *list.splits nat.splits*)

erase t $\sqsubseteq$ *s* is preserved by *remove-annot* (erased term has *None* everywhere).

lemma *erase-compl-remove-annot*:
assumes *erase t* $\sqsubseteq$ *s*
shows *erase t* $\sqsubseteq$ *remove-annot s p*
using *assms*
proof (*induction t arbitrary*: *s p*)
case (*Vr x* ξ)
then show *?case* **by** (*auto elim*!: *compl-Vr-leftE intro*: *compl.intros*)
next
case (*Ct c* ξ)
then show *?case* **by** (*auto elim*!: *compl-Ct-leftE intro*: *compl.intros*)
next
case (*App t1 t2* ξ)
from *App.prems*(*1*) **obtain** *s1 s2* ξs **where** *seq*: *s* = *App s1 s2* ξs
erase t1 $\sqsubseteq$ *s1 erase t2* $\sqsubseteq$ *s2*
by (*auto elim*!: *compl-App-leftE*)
show *?case* **using** *App.prems seq*
by (*cases p*) (*auto split*: *nat.splits intro*: *compl.intros App.IH*)

next
case (*Lam x ξ t ζ*)
from *Lam.prems*(*1*) **obtain** *s0 ζs ξs* **where** *seq*: *s* = *Lam x ζs s0 ξs*
erase t ⊑ *s0*
by (*auto elim*!: *compl-Lam-leftE*)
show *?case* **using** *Lam.prems seq*
by (*cases p*) (*auto split*: *nat.splits list.splits intro*: *compl.intros Lam.IH*)
qed

Same shape extensionality: If $t \sqsubseteq u$ and $t \sqsubseteq u'$, and u, u' are F-terms with the same type at every position, then $u = u'$.

lemma *fterm-extensionality*:
assumes *t* ⊑ *u t* ⊑ *u' fterm u fterm u'* ⋀*p*. *p* ∈ *poss t* ⟹ *mtp-of u p* = *mtp-of u' p*
shows *u* = *u'*
using *assms*
proof (*induction t arbitrary*: *u u'*)
case (*Vr x ξ*)
from *Vr.prems*(*1*) **obtain** *ξu* **where** *u* = *Vr x ξu* **by** (*auto elim*!: *compl-Vr-leftE*)
moreover from *Vr.prems*(*2*) **obtain** *ξu'* **where** *u'* = *Vr x ξu'* **by** (*auto elim*!: *compl-Vr-leftE*)
ultimately show *u* = *u'* **using** *Vr.prems*(*5*)[*of* []] **by** *auto*
next
case (*Ct c ξ*)
from *Ct.prems*(*1*) **obtain** *ξu* **where** *u* = *Ct c ξu* **by** (*auto elim*!: *compl-Ct-leftE*)
moreover from *Ct.prems*(*2*) **obtain** *ξu'* **where** *u'* = *Ct c ξu'* **by** (*auto elim*!: *compl-Ct-leftE*)
ultimately show *u* = *u'* **using** *Ct.prems*(*5*)[*of* []] **by** *auto*
next
case (*App t1 t2 ξ*)
from *App.prems*(*1*) **obtain** *u1 u2 ξu* **where** *ueq*: *u* = *App u1 u2 ξu*
t1 ⊑ *u1 t2* ⊑ *u2* **by** (*auto elim*!: *compl-App-leftE*)
from *App.prems*(*2*) **obtain** *u1' u2' ξu'* **where** *u'eq*: *u'* = *App u1' u2' ξu'*
t1 ⊑ *u1' t2* ⊑ *u2'* **by** (*auto elim*!: *compl-App-leftE*)
have *fu*: *fterm u1 fterm u2* **using** *App.prems*(*3*) *ueq*(*1*) **by** *auto*
have *fu'*: *fterm u1' fterm u2'* **using** *App.prems*(*4*) *u'eq*(*1*) **by** *auto*
have *eq1*: ⋀*p*. *p* ∈ *poss t1* ⟹ *mtp-of u1 p* = *mtp-of u1' p*
proof −
fix *p* **assume** *p* ∈ *poss t1*
then have *Suc 0* # *p* ∈ *poss* (*App t1 t2 ξ*) **by** *auto*
from *App.prems*(*5*)[*OF this*] *ueq*(*1*) *u'eq*(*1*) **show** *mtp-of u1 p* = *mtp-of u1' p* **by** *auto*
qed
have *eq2*: ⋀*p*. *p* ∈ *poss t2* ⟹ *mtp-of u2 p* = *mtp-of u2' p*
proof −
fix *p* **assume** *p* ∈ *poss t2*
then have *2* # *p* ∈ *poss* (*App t1 t2 ξ*) **by** *auto*
from *App.prems*(*5*)[*OF this*] *ueq*(*1*) *u'eq*(*1*) **show** *mtp-of u2 p* = *mtp-of u2' p* **by** *auto*

```
  qed
  have u1 = u1' using App.IH(1)[OF ueq(2) u'eq(2) fu(1) fu'(1) eq1] .
 moreover have u2 = u2' using App.IH(2)[OF ueq(3) u'eq(3) fu(2) fu'(2) eq2]
.
  moreover have ξu = ξu' using App.prems(5)[of []] ueq(1) u'eq(1) by auto
  ultimately show u = u' using ueq(1) u'eq(1) by auto
next
  case (Lam x ξ t ζ)
  from Lam.prems(1) obtain u0 ζu ξu where ueq: u = Lam x ζu u0 ξu
    t ⊑ u0 by (auto elim!: compl-Lam-leftE)
  from Lam.prems(2) obtain u0' ζu' ξu' where u'eq: u' = Lam x ζu' u0' ξu'
    t ⊑ u0' by (auto elim!: compl-Lam-leftE)
  have fu: fterm u0 using Lam.prems(3) ueq(1) by auto
  have fu': fterm u0' using Lam.prems(4) u'eq(1) by auto
  have eq0: ⋀p. p ∈ poss t ⟹ mtp-of u0 p = mtp-of u0' p
  proof −
    fix p assume p ∈ poss t
    then have 2 # p ∈ poss (Lam x ξ t ζ) by auto
    from Lam.prems(5)[OF this] ueq(1) u'eq(1) show mtp-of u0 p = mtp-of u0'
p by auto
  qed
  have u0 = u0' using Lam.IH[OF ueq(2) u'eq(2) fu fu' eq0] .
  moreover have ζu = ζu' using Lam.prems(5)[of [Suc 0]] ueq(1) u'eq(1) by
auto
  moreover have ξu = ξu' using Lam.prems(5)[of []] ueq(1) u'eq(1) by auto
  ultimately show u = u' using ueq(1) u'eq(1) by auto
qed
```

The crucial fact for completeness: If $t \sqsubseteq u$ and u is an instance of v ($u \leq v$), and every type variable of v is "witnessed" at some non-bot position of t, then u is the unique F-term u' with $t \sqsubseteq u'$ and $u' \leq v$.

```
theorem crucial-uniqueness:
  assumes compl-u: t ⊑ u and fu: fterm u
    and inst-u: ∃ϱ. u = subst-trm ϱ v and fv: fterm v
   and cover: ∀α ∈ tvars-trm v. ∃p ∈ poss v. α ∈ tvars-mty (mtp-of v p) ∧ mtp-of
t p ≠ None
    and compl-u': t ⊑ u' and fu': fterm u'
    and inst-u': ∃ϱ'. u' = subst-trm ϱ' v
  shows u = u'
proof −
  from inst-u obtain ϱ where u-eq: u = subst-trm ϱ v by auto
  from inst-u' obtain ϱ' where u'-eq: u' = subst-trm ϱ' v by auto
```

By $[\![?t \sqsubseteq ?u;\ ?t \sqsubseteq ?u';\ fterm\ ?u;\ fterm\ ?u';\ \bigwedge p.\ p \in poss\ ?t \Longrightarrow mtp\text{-}of\ ?u\ p = mtp\text{-}of\ ?u'\ p]\!] \Longrightarrow ?u = ?u'$, suffices to show $\forall p{\in}poss\ t.\ mtp\text{-}of\ u\ p = mtp\text{-}of\ u'\ p$.

```
  have poss-eq: poss t = poss v
    using compl-poss-eq[OF compl-u] u-eq by simp
  show u = u'
```

proof (*rule fterm-extensionality*[*OF compl-u compl-u′ fu fu′*])
fix p **assume** *p-in*: $p \in$ *poss* t
then have *pv*: $p \in$ *poss* v **using** *poss-eq* **by** *auto*
have *u-mtp*: *mtp-of* $u\ p =$ *subst-mty* ϱ (*mtp-of* $v\ p$)
using *mtp-of-subst*[*OF pv*] *u-eq* **by** *auto*
have *u′-mtp*: *mtp-of* $u'\ p =$ *subst-mty* ϱ' (*mtp-of* $v\ p$)
using *mtp-of-subst*[*OF pv*] *u′-eq* **by** *auto*

Since v is an F-term, $\exists \sigma.$ *mtp-of* $v\ p =$ *Some* σ.

have $\exists \sigma.$ *mtp-of* $v\ p =$ *Some* σ **using** *fterm-mtp-of-Some*[*OF fv pv*] **.**
then obtain σ **where** *vsig*: *mtp-of* $v\ p =$ *Some* σ **by** *auto*

Need to show *subst-ty* $\varrho\ \sigma =$ *subst-ty* $\varrho'\ \sigma$. By (*subst-ty ?ϱ ?τ = subst-ty ?ϱ′ ?τ*) = ($\forall \alpha \in$ *tvars-ty ?τ*. *?ϱ* $\alpha =$ *?ϱ′* α), suffices to show $\forall \alpha \in$ *tvars-ty* σ. $\varrho\ \alpha = \varrho'\ \alpha$.

have *subst-ty* $\varrho\ \sigma =$ *subst-ty* $\varrho'\ \sigma$
proof (*rule subst-ty-cong*)
fix α **assume** *alpha-in*: $\alpha \in$ *tvars-ty* σ
then have $\alpha \in$ *tvars-trm* v
using *vsig pv tvars-trm-eq-UN-mtp* **by** *fastforce*
from *cover*[*rule-format*, *OF this*]
obtain q **where** *qv*: $q \in$ *poss* v **and** *alpha-q*: $\alpha \in$ *tvars-mty* (*mtp-of* $v\ q$)
and *t-q*: *mtp-of* $t\ q \neq$ *None* **by** *auto*
have *qt*: $q \in$ *poss* t **using** *poss-eq qv* **by** *auto*

Since *mtp-of* $t\ q$ is not *None* and $t \sqsubseteq u, u'$, the annotation is preserved.

have *mtp-of* $t\ q \sqsubseteq_m$ *mtp-of* $u\ q$ **using** *compl-mtp-of*[*OF compl-u*] **.**
moreover have *mtp-of* $t\ q \sqsubseteq_m$ *mtp-of* $u'\ q$ **using** *compl-mtp-of*[*OF compl-u′*]
.
ultimately have *eq-at-q*: *mtp-of* $u\ q =$ *mtp-of* $t\ q \wedge$ *mtp-of* $u'\ q =$ *mtp-of* $t\ q$
using *t-q* **by** (*auto simp*: *mcompl-def*)
then have *mtp-of* $u\ q =$ *mtp-of* $u'\ q$ **by** *simp*

Now use *?p* $\in$ *poss ?t* $\Longrightarrow$ *mtp-of* (*subst-trm ?ϱ ?t*) *?p* = *subst-mty ?ϱ* (*mtp-of ?t ?p*) to relate to substitutions.

have *subst-mty* ϱ (*mtp-of* $v\ q$) = *subst-mty* ϱ' (*mtp-of* $v\ q$)
using ‹*mtp-of* $u\ q =$ *mtp-of* $u'\ q$› *mtp-of-subst*[*OF qv*] *u-eq u′-eq* **by** *auto*
then show $\varrho\ \alpha = \varrho'\ \alpha$ **using** *alpha-q*
by (*cases mtp-of* $v\ q$) (*auto simp*: *subst-ty-agree*)
qed
then show *mtp-of* $u\ p =$ *mtp-of* $u'\ p$ **using** *u-mtp u′-mtp vsig* **by** *simp*
qed
qed

Strict ($\sqsubset$) witnesses a position difference.

lemma *compl-strict-witness*:
assumes $s' \sqsubseteq s\ s' \neq s$
shows $\exists p \in$ *poss s*. *mtp-of* $s'\ p =$ *None* $\wedge$ *mtp-of* $s\ p \neq$ *None*

```
  using assms
proof (induction s' s rule: compl.induct)
  case (compl-App s1 s1' s2 s2' ξ ξ')
  show ?case
  proof (cases s1 = s1')
    case True
    then show ?thesis
    proof (cases s2 = s2')
      case True
      with ‹s1 = s1'› compl-App.prems compl-App.hyps
      show ?thesis by (auto simp: mcompl-def)
    next
      case False
       with compl-App.IH(2) obtain p where p ∈ poss s2' mtp-of s2 p = None
mtp-of s2' p ≠ None
        by auto
      then show ?thesis by (intro bexI[of - 2 # -]) auto
    qed
  next
    case False
     with compl-App.IH(1) obtain p where p ∈ poss s1' mtp-of s1 p = None
mtp-of s1' p ≠ None
      by auto
    then show ?thesis by (intro bexI[of - Suc 0 # p]) auto
  qed
next
  case (compl-Lam ζ ζ' t t' ξ ξ' x)
  show ?case
  proof (cases t = t')
    case True
    then show ?thesis
    proof (cases ζ = ζ')
      case True
      with ‹t = t'› compl-Lam.prems compl-Lam.hyps
      show ?thesis by (auto simp: mcompl-def)
    next
      case False
      with compl-Lam.hyps show ?thesis
        by (auto simp: mcompl-def intro!: bexI[of - [Suc 0]])
    qed
  next
    case False
    with compl-Lam.IH obtain p where p ∈ poss t' mtp-of t p = None mtp-of t'
p ≠ None
      by auto
    then show ?thesis by auto
  qed
qed (auto simp: mcompl-def)
```

Multiple completions from agreeing substitutions.

lemma *multiple-compl*:
assumes $s \sqsubseteq$ *subst-trm* $\varrho\ v$
$\bigwedge\alpha\ p.\ [\![\ p \in$ *poss* $v;\ \alpha \in$ *tvars-mty* (*mtp-of* $v\ p$); *mtp-of* $s\ p \neq$ *None* $]\!] \Longrightarrow \varrho\ \alpha = \varrho'\ \alpha$
shows $s \sqsubseteq$ *subst-trm* $\varrho'\ v$
using *assms*
proof (*induction* v *arbitrary*: s)
case (*Vr* $x\ \xi$)
show *?case*
proof (*cases* ξ)
case *None* **then show** *?thesis* **using** *Vr* **by** (*auto elim*!: *compl-Vr-rightE intro*: *compl.intros*)
next
case (*Some* σ)
from *Vr.prems*(*1*) *Some* **obtain** ξs **where** *seq*: $s = Vr\ x\ \xi s\ \xi s \sqsubseteq_m$ *Some* (*subst-ty* $\varrho\ \sigma$)
by (*auto elim*!: *compl-Vr-rightE*)
show *?thesis*
proof (*cases* ξs)
case *None* **then show** *?thesis* **using** *seq*(*1*) *Some* **by** (*auto intro*: *compl.intros*)
next
case (*Some* τ)
then have $\tau =$ *subst-ty* $\varrho\ \sigma$ **using** *seq*(*2*) **by** (*auto simp*: *mcompl-def*)
moreover have $\forall\,\alpha{\in}$*tvars-ty* $\sigma.\ \varrho\ \alpha = \varrho'\ \alpha$
using *Vr.prems*(*2*)[*of* []] *seq*(*1*) *Some* ‹$\xi =$ *Some* σ› **by** *auto*
then have *subst-ty* $\varrho\ \sigma =$ *subst-ty* $\varrho'\ \sigma$ **by** (*intro subst-ty-cong*) *auto*
ultimately have $\tau =$ *subst-ty* $\varrho'\ \sigma$ **by** *simp*
then show *?thesis* **using** *seq*(*1*) *Some* ‹$\xi =$ *Some* σ› **by** (*auto intro*: *compl.intros simp*: *mcompl-def*)
qed
qed
next
case (*Ct* $c\ \xi$)
then show *?case*
proof (*cases* ξ)
case *None* **then show** *?thesis* **using** *Ct* **by** (*auto elim*!: *compl-Ct-rightE intro*: *compl.intros*)
next
case (*Some* σ)
from *Ct.prems*(*1*) *Some* **obtain** ξs **where** *seq*: $s = Ct\ c\ \xi s\ \xi s \sqsubseteq_m$ *Some* (*subst-ty* $\varrho\ \sigma$)
by (*auto elim*!: *compl-Ct-rightE*)
show *?thesis*
proof (*cases* ξs)
case *None* **then show** *?thesis* **using** *seq*(*1*) *Some* **by** (*auto intro*: *compl.intros*)
next
case (*Some* τ)
then have $\tau =$ *subst-ty* $\varrho\ \sigma$ **using** *seq*(*2*) **by** (*auto simp*: *mcompl-def*)
moreover have $\forall\,\alpha{\in}$*tvars-ty* $\sigma.\ \varrho\ \alpha = \varrho'\ \alpha$

```
      using Ct.prems(2)[of []] seq(1) Some ‹ξ = Some σ› by auto
    then have subst-ty ϱ σ = subst-ty ϱ' σ by (intro subst-ty-cong) auto
    ultimately have τ = subst-ty ϱ' σ by simp
      then show ?thesis using seq(1) Some ‹ξ = Some σ› by (auto intro:
compl.intros simp: mcompl-def)
  qed
 qed
next
 case (App t1 t2 ξ)
 from App.prems(1) obtain s1 s2 ξs where seq: s = App s1 s2 ξs
  s1 ⊑ subst-trm ϱ t1 s2 ⊑ subst-trm ϱ t2 ξs ⊑m subst-mty ϱ ξ
  by (auto elim!: compl-App-rightE)
 have s1 ⊑ subst-trm ϱ' t1
 proof (rule App.IH(1)[OF seq(2)])
  fix α p assume p ∈ poss t1 α ∈ tvars-mty (mtp-of t1 p) mtp-of s1 p ≠ None
  then show ϱ α = ϱ' α using App.prems(2)[of Suc 0 # p α] seq(1) by auto
 qed
 moreover have s2 ⊑ subst-trm ϱ' t2
 proof (rule App.IH(2)[OF seq(3)])
  fix α p assume p ∈ poss t2 α ∈ tvars-mty (mtp-of t2 p) mtp-of s2 p ≠ None
  then show ϱ α = ϱ' α using App.prems(2)[of 2 # p α] seq(1) by auto
 qed
 moreover have ξs ⊑m subst-mty ϱ' ξ
 proof (cases ξ)
  case None then show ?thesis using seq(4) by auto
 next
  case (Some σ)
  show ?thesis
  proof (cases ξs)
   case None then show ?thesis by auto
  next
   case (Some τ)
    then have τ = subst-ty ϱ σ using seq(4) ‹ξ = Some σ› by (auto simp:
mcompl-def)
    moreover have ∀α ∈ tvars-ty σ. ϱ α = ϱ' α
     using App.prems(2)[of []] seq(1) Some ‹ξ = Some σ› by auto
    then have subst-ty ϱ σ = subst-ty ϱ' σ by (intro subst-ty-cong) auto
   ultimately show ?thesis using Some ‹ξ = Some σ› by (auto simp: mcompl-def)
  qed
 qed
 ultimately show ?case using seq(1) by (auto intro: compl.intros)
next
 case (Lam x ξ t ζ)
 from Lam.prems(1) obtain s0 ζs ξs where
  seq: s = Lam x ζs s0 ξs
  s0 ⊑ subst-trm ϱ t ζs ⊑m subst-mty ϱ ξ ξs ⊑m subst-mty ϱ ζ
  by (auto elim!: compl-Lam-rightE)
 have s0 ⊑ subst-trm ϱ' t
 proof (rule Lam.IH[OF seq(2)])
```

```
    fix α p assume p ∈ poss t α ∈ tvars-mty (mtp-of t p) mtp-of s0 p ≠ None
    then show ϱ α = ϱ' α using Lam.prems(2)[of 2 # p α] seq(1) by auto
  qed
  moreover have ζs ⊑m subst-mty ϱ' ξ
  proof (cases ξ)
    case None then show ?thesis using seq(3) by auto
  next
    case (Some σ)
    show ?thesis
    proof (cases ζs)
      case None then show ?thesis by auto
    next
      case (Some τ)
       then have τ = subst-ty ϱ σ using seq(3) ‹ξ = Some σ› by (auto simp:
mcompl-def)
    have ∀α' ∈ tvars-ty σ. ϱ α' = ϱ' α'
    proof (intro ballI)
      fix α' assume α' ∈ tvars-ty σ
      then show ϱ α' = ϱ' α'
        using Lam.prems(2)[of [Suc 0] α'] ‹ξ = Some σ› seq(1) Some by auto
    qed
    then have eq: subst-ty ϱ σ = subst-ty ϱ' σ by (intro subst-ty-cong) auto
    from ‹τ = subst-ty ϱ σ› eq have τ = subst-ty ϱ' σ by simp
    then show ?thesis using Some ‹ξ = Some σ› by (auto simp: mcompl-def)
    qed
  qed
  moreover have ξs ⊑m subst-mty ϱ' ζ
  proof (cases ζ)
    case None then show ?thesis using seq(4) by auto
  next
    case (Some σ)
    show ?thesis
    proof (cases ξs)
      case None then show ?thesis by auto
    next
      case (Some τ)
       then have τ = subst-ty ϱ σ using seq(4) ‹ζ = Some σ› by (auto simp:
mcompl-def)
      moreover have ∀α ∈ tvars-ty σ. ϱ α = ϱ' α
        using Lam.prems(2)[of []] seq(1) Some ‹ζ = Some σ› by auto
      then have subst-ty ϱ σ = subst-ty ϱ' σ by (intro subst-ty-cong) auto
    ultimately show ?thesis using Some ‹ζ = Some σ› by (auto simp: mcompl-def)
    qed
  qed
  ultimately show ?case using seq(1) by (auto intro: compl.intros)
qed
```

1.4.4 Well-typedness and type inference

Well-typedness predicate on F-terms. We work in a locale fixing the constant-typing function *ctpOf*. We also use the abbreviation *tpOf* for the type of an F-term.

abbreviation *tpOf* :: $('tv,'k,'v,'c)$ *trm* $\Rightarrow$ *nat list* $\Rightarrow$ $('tv,'k)$ *ty* **where**
  tpOf u p $\equiv$ *the* (*mtp-of u p*)

abbreviation *tpOfR* :: $('tv,'k,'v,'c)$ *trm* $\Rightarrow$ $('tv,'k)$ *ty* **where**
  tpOfR u $\equiv$ *the* (*mtp-of u* [])

Free typed variables of a term: pairs (x, σ) where variable x occurs free with type annotation *Some* σ.

fun *fvars* :: $('tv,'k,'v,'c)$ *trm* $\Rightarrow$ $('v \times ('tv,'k)$ *ty*$)$ *set* **where**
  fvars (*Vr x* (*Some* σ)) = $\{(x, \sigma)\}$
| *fvars* (*Vr x None*) = {}
| *fvars* (*Ct c* ξ) = {}
| *fvars* (*App t1 t2* ξ) = *fvars t1* $\cup$ *fvars t2*
| *fvars* (*Lam x* ξ *t* ζ) = $\{(y, \sigma) \in$ *fvars t*. $y \neq x\}$

lemma *fvars-subst*: *fvars* (*subst-trm* ϱ *t*) = $(\lambda(x, \sigma).\ (x,$ *subst-ty* $\varrho\ \sigma))$ ‘ *fvars t*
proof (*induction t*)
  case (*Vr x* ξ) **then show** *?case* **by** (*cases* ξ) *auto*
next
  case (*Ct c* ξ) **then show** *?case* **by** *auto*
next
  case (*App t1 t2* ξ) **then show** *?case* **by** *auto*
next
  case (*Lam x* ξ *t* ζ) **then show** *?case* **by** *auto*
qed

locale *signature* =
  fixes *ctpOf* :: $'c \Rightarrow ('tv,'k)$ *ty*
begin

inductive *wt* :: $('tv,'k,'v,'c)$ *trm* $\Rightarrow$ *bool* ($\vdash$ - [50] 50) **where**
  wt-Vr: $\vdash$ *Vr x* (*Some* σ)
| *wt-Ct*: $\exists\,\varrho.\ \sigma =$ *subst-ty* ϱ (*ctpOf c*) $\Longrightarrow \vdash$ *Ct c* (*Some* σ)
| *wt-App*: $[\![\vdash u; \vdash v;$ *tpOfR u* = *Arrow* (*tpOfR v*) $\sigma\ ]\!] \Longrightarrow \vdash$ *App u v* (*Some* σ)
| *wt-Lam*: $[\![\vdash u; \forall\tau.\ (x, \tau) \in$ *fvars u* $\longrightarrow\ \tau = \sigma\ ]\!] \Longrightarrow \vdash$ *Lam x* (*Some* σ) *u* (*Some* (*Arrow* σ (*tpOfR u*)))

Inversion rules for *wt*.

lemma *wt-Vr-inv*: $\vdash$ *Vr x* $\xi \Longrightarrow \exists\,\sigma.\ \xi =$ *Some* σ
  by (*erule wt.cases*) *auto*

lemma *wt-Ct-inv*: $\vdash$ *Ct c* (*Some* σ) $\Longrightarrow \exists\,\varrho.\ \sigma =$ *subst-ty* ϱ (*ctpOf c*)
  by (*erule wt.cases*) *auto*

lemma *wt-App-inv*: $\vdash$ *App t1 t2* (*Some* σ) $\Longrightarrow$ $\vdash$ *t1* $\wedge$ $\vdash$ *t2* $\wedge$ *tpOfR t1* = *Arrow* (*tpOfR t2*) σ
by (*erule wt.cases*) *auto*

lemma *wt-Lam-inv*: $\vdash$ *Lam x* (*Some* σ) *t* (*Some* τ) $\Longrightarrow$ $\vdash$ *t* $\wedge$ τ = *Arrow* σ (*tpOfR t*)
by (*erule wt.cases*) *auto*

lemma *wt-App-invE*:
assumes $\vdash$ *App u1 u2* ξ
obtains σ **where** $\vdash$ *u1* $\vdash$ *u2* ξ = *Some* σ *tpOfR u1* = *Arrow* (*tpOfR u2*) σ
using *assms* **by** (*auto elim*: *wt.cases*)

lemma *wt-Lam-invE*:
assumes $\vdash$ *Lam x* ξ *u* ζ
obtains σ **where** $\vdash$ *u* ξ = *Some* σ ζ = *Some* (*Arrow* σ (*tpOfR u*))
$\forall \tau$. (x, τ) $\in$ *fvars u* $\longrightarrow$ τ = σ
using *assms* **by** (*auto elim*: *wt.cases*)

lemma *wt-fterm*: $\vdash$ *u* $\Longrightarrow$ *fterm u*
by (*induction u rule*: *wt.induct*) *auto*

Substitution lemma for well-typedness. Since $\vdash$ *Vr ?x* (*Some ?*σ) has no conditions, any type substitution preserves *wt*.

lemma *subst-wt*:
assumes $\vdash$ *v*
shows $\vdash$ *subst-trm* ϱ *v*
using *assms*
proof (*induction v rule*: *wt.induct*)
case (*wt-Vr x* σ)
show *?case* **by** (*simp*, *rule wt.wt-Vr*)
next
case (*wt-Ct* σ *c*)
then obtain $\varrho 0$ **where** σ = *subst-ty* $\varrho 0$ (*ctpOf c*) **by** *auto*
then have *subst-ty* ϱ σ = *subst-ty* (ϱ $\circ\circ$ $\varrho 0$) (*ctpOf c*)
by (*simp add*: *subst-ty-comp*)
then show *?case* **by** (*simp*, *intro wt.wt-Ct*, *auto*)
next
case (*wt-App u v* σ)
from *wt-App* **have** *fu*: *fterm u fterm v* **by** (*auto dest*: *wt-fterm*)
then obtain $\tau 1$ $\tau 2$ **where** *mtp-of u* [] = *Some* $\tau 1$ *mtp-of v* [] = *Some* $\tau 2$
using *fterm-mtp-of-Some* **by** (*metis nil-in-poss*)
with *wt-App* **have** *tp*: *tpOfR* (*subst-trm* ϱ *u*) = *Arrow* (*tpOfR* (*subst-trm* ϱ *v*)) (*subst-ty* ϱ σ)
by (*simp add*: *mtp-of-subst*[*of* [] *u* ϱ, *simplified*] *mtp-of-subst*[*of* [] *v* ϱ, *simplified*])
from *wt-App tp* **show** *?case* **by** (*simp*, *intro wt.wt-App*) *auto*
next

```
  case (wt-Lam u x σ)
  from wt-Lam have fu: fterm u by (auto dest: wt-fterm)
  then obtain τ where mt: mtp-of u [] = Some τ
    using fterm-mtp-of-Some by (metis nil-in-poss)
  from wt-Lam mt show ?case
  proof −
    have eq: mtp-of (subst-trm ϱ u) [] = Some (subst-ty ϱ τ)
      using mt by (simp add: mtp-of-subst[of [] u ϱ, simplified])
    have tpR: tpOfR (subst-trm ϱ u) = subst-ty ϱ τ using eq by simp
    have tpU: tpOfR u = τ using mt by simp
    have fv-cond: ∀ τ'. (x, τ') ∈ fvars (subst-trm ϱ u) ⟶ τ' = subst-ty ϱ σ
      using wt-Lam(2) by (auto simp: fvars-subst)
    from wt-Lam have ⊢ Lam x (Some (subst-ty ϱ σ)) (subst-trm ϱ u)
      (Some (Arrow (subst-ty ϱ σ) (tpOfR (subst-trm ϱ u))))
      by (intro wt.wt-Lam) (use fv-cond in auto)
    then show ?thesis using tpR tpU by simp
  qed
qed
```

If t is a typeable unambiguous C-term, then its well-typed completion is unique (there is exactly one F-term u such that $t \sqsubseteq u$ and $\vdash u$).

```
lemma cterm-unique-completion:
  assumes cterm t unambiguous t
    t ⊑ u ⊢ u t ⊑ v ⊢ v
  shows u = v
  using assms
proof (induction t arbitrary: u v)
  case (Vr x ξ)
  then show ?case by (auto elim!: compl-Vr-leftE simp: mcompl-def)
next
  case (Ct c ξ)
  then show ?case by (auto elim!: compl-Ct-leftE simp: mcompl-def)
next
  case (App t1 t2 ξ)
  from App.prems have ct1: cterm t1 cterm t2 ξ = None by auto
  from App.prems have ua1: unambiguous t1 unambiguous t2 by auto
  from App.prems(3) obtain u1 u2 ξu where ueq: u = App u1 u2 ξu
    t1 ⊑ u1 t2 ⊑ u2 ξ ⊑m ξu
    by (auto elim!: compl-App-leftE)
  from App.prems(5) obtain v1 v2 ξv where veq: v = App v1 v2 ξv
    t1 ⊑ v1 t2 ⊑ v2 ξ ⊑m ξv
    by (auto elim!: compl-App-leftE)
  from App.prems(4) ueq(1) obtain σu where
    wtu: ⊢ u1 ⊢ u2 and xiu: ξu = Some σu and
    tpu: tpOfR u1 = Arrow (tpOfR u2) σu
    by (auto elim!: wt-App-invE)
  from App.prems(6) veq(1) obtain σv where
    wtv: ⊢ v1 ⊢ v2 and xiv: ξv = Some σv and
    tpv: tpOfR v1 = Arrow (tpOfR v2) σv
```

by (*auto elim*!: *wt-App-invE*)
have *u1* = *v1* **using** *App.IH*(*1*)[*OF ct1*(*1*) *ua1*(*1*) *ueq*(*2*) *wtu*(*1*) *veq*(*2*) *wtv*(*1*)]
.
moreover have *u2* = *v2* **using** *App.IH*(*2*)[*OF ct1*(*2*) *ua1*(*2*) *ueq*(*3*) *wtu*(*2*) *veq*(*3*) *wtv*(*2*)] .
ultimately have $\sigma u = \sigma v$ **using** *tpu tpv* **by** *simp*
with *ueq*(*1*) *veq*(*1*) ‹*u1* = *v1*› ‹*u2* = *v2*› *xiu xiv* **show** *u* = *v* **by** *simp*
next
case (*Lam x* ξ *t* ζ)
from *Lam.prems* **have** *ct*: *cterm t* $\xi \neq$ *None* ζ = *None* **by** *auto*
from *Lam.prems* **have** *ua*: *unambiguous t* **by** *auto*
obtain σ **where** *sig*: ξ = *Some* σ **using** *ct*(*2*) **by** (*cases* ξ) *auto*
from *Lam.prems*(*3*) *sig* **obtain** *u'* ξu **where** *ueq*: *u* = *Lam x* (*Some* σ) *u'* ξu *t* $\sqsubseteq$ *u'*
by (*auto elim*!: *compl-Lam-leftE simp*: *mcompl-def*)
from *Lam.prems*(*5*) *sig* **obtain** *v'* ξv **where** *veq*: *v* = *Lam x* (*Some* σ) *v'* ξv *t* $\sqsubseteq$ *v'*
by (*auto elim*!: *compl-Lam-leftE simp*: *mcompl-def*)
from *Lam.prems*(*4*) *ueq*(*1*) **obtain** τu **where**
wtu: $\vdash$ *u'* **and** *xiusig*: *Some* σ = *Some* σ **and** *xiu*: ξu = *Some* (*Arrow* σ (*tpOfR u'*))
by (*auto elim*!: *wt-Lam-invE*)
from *Lam.prems*(*6*) *veq*(*1*) **obtain** τv **where**
wtv: $\vdash$ *v'* **and** *xivsig*: *Some* σ = *Some* σ **and** *xiv*: ξv = *Some* (*Arrow* σ (*tpOfR v'*))
by (*auto elim*!: *wt-Lam-invE*)
have *u'* = *v'* **using** *Lam.IH*[*OF ct*(*1*) *ua ueq*(*2*) *wtu veq*(*2*) *wtv*] .
with *ueq veq xiu xiv* **show** *u* = *v* **by** *simp*
qed

1.4.5 Well-Typed Completions

definition *is-wt-completion* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *bool* **where**
is-wt-completion t u ⟷ *t* ⊑ *u* ∧ *fterm u* ∧ ⊢ *u*

definition *is-mgen* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *bool* **where**
is-mgen t v ⟷ *is-wt-completion t v* ∧
(∀ *u*. *is-wt-completion t u* ⟶ (∃ ϱ. *u* = *subst-trm* ϱ *v*))

definition *typeable* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *bool* **where**
typeable t ⟷ (∃ *u*. *is-wt-completion t u*)

end

We work in an extended locale that assumes the existence of *is-mgen*: For any typeable unambiguous term, there exists a most general well-typed completion.

locale *signature-with-mgen* = *signature ctpOf*
for *ctpOf* :: '*c* ⇒ ('*tv*,'*k*) *ty* +

assumes *mgen-exists*:
⟦ typeable (t::('tv,'k,'v,'c) trm); unambiguous t ⟧ ⟹ ∃ v. is-mgen t v **and**
tyvars-inf: *infinite (UNIV :: 'tv set)*
begin

definition *mgen* :: *('tv,'k,'v,'c) trm ⇒ ('tv,'k,'v,'c) trm* **where**
mgen t = (SOME v. is-mgen t v)

lemma *mgen-is-mgen*:
assumes *typeable t unambiguous t*
shows *is-mgen t (mgen t)*
proof −
from *assms* **have** *∃ v. is-mgen t v*
by (*intro mgen-exists*)
then show *?thesis* **unfolding** *mgen-def* **by** (*rule someI-ex*)
qed

lemma *mgen-compl*:
assumes *typeable t unambiguous t*
shows *t ⊑ mgen t*
using *mgen-is-mgen[OF assms]* **by** (*auto simp*: *is-mgen-def is-wt-completion-def*)

lemma *mgen-fterm*:
assumes *typeable t unambiguous t*
shows *fterm (mgen t)*
using *mgen-is-mgen[OF assms]* **by** (*auto simp*: *is-mgen-def is-wt-completion-def*)

lemma *mgen-wt*:
assumes *typeable t unambiguous t*
shows *⊢ mgen t*
using *mgen-is-mgen[OF assms]* **by** (*auto simp*: *is-mgen-def is-wt-completion-def*)

lemma *mgen-most-general*:
assumes *typeable t unambiguous t is-wt-completion t u*
shows *∃ ϱ. u = subst-trm ϱ (mgen t)*
using *mgen-is-mgen[OF assms(1,2)] assms(3)* **by** (*auto simp*: *is-mgen-def*)

1.5 Correct Printings

A correct printing of a typable unambiguous C-term t is a term s such that *mgen t* is the unique most general well-typed completion of s:

definition ‹*correct-printing t s ⟷ (∀ t'. is-mgen s t' ⟷ t' = mgen t)*›

definition ‹*strong-correct-printing t s ⟷ (∀ t'. is-wt-completion s t' ⟷ t' = mgen t)*›

Extra tyvars yield distinct most general completions.

lemma *two-mgen*:

assumes *ua*: *unambiguous s* **and** *mg*: *is-mgen s v* **and** *extra*: *tvars-trm v* − *tvars-trm s* ≠ {}
shows ∃ *v′*. *v′* ≠ *v* ∧ *is-mgen s v′*
proof −
from *mg* **have** *sv*: *s* ⊑ *v* **and** *fv*: *fterm v* **and** *wtv*: ⊢ *v*
and *mg-prop*: ⋀*u*. *is-wt-completion s u* ⟹ ∃ *ϱ*. *u* = *subst-trm ϱ v*
by (*auto simp*: *is-mgen-def is-wt-completion-def*)
from *extra* **obtain** *α* **where** *alpha-v*: *α* ∈ *tvars-trm v* **and** *alpha-ns*: *α* ∉ *tvars-trm s* **by** *auto*

Property (a): *α* only appears at *None* positions of *s*.

have *prop-a*: *mtp-of s p* = *None* **if** *p* ∈ *poss v α* ∈ *tvars-mty* (*mtp-of v p*) **for** *p*
proof (*rule ccontr*)
assume *mtp-of s p* ≠ *None*
have *p* ∈ *poss s* **using** *that*(*1*) *compl-poss-eq*[*OF sv*] **by** *simp*
have *mtp-of s p* ⊑$_m$ *mtp-of v p* **using** *compl-mtp-of*[*OF sv*] **.**
then have *mtp-of s p* = *mtp-of v p* **using** ‹*mtp-of s p* ≠ *None*› **by** (*auto simp*: *mcompl-def*)
then have *α* ∈ *tvars-mty* (*mtp-of s p*) **using** *that*(*2*) **by** *simp*
then have *α* ∈ *tvars-trm s* **using** *tvars-trm-eq-UN-mtp* ‹*p* ∈ *poss s*› **by** *fastforce*
with *alpha-ns* **show** *False* **by** *contradiction*
qed
obtain *β* **where** *beta-nv*: *β* ∉ *tvars-trm v*
using *ex-new-if-finite*[*OF tyvars-inf tvars-trm-finite*[*of v*]] **by** *blast*
define *v′* **where** *v′* = *subst-trm* (*single-subst* (*TyVar β*) *α*) *v*

v != v'.

have *v-neq*: *v* ≠ *v′*
proof
assume *v* = *v′*
then have *subst-trm* (*single-subst* (*TyVar β*) *α*) *v* = *v* **by** (*simp add*: *v′-def*)
then have *single-subst* (*TyVar β*) *α α* = *TyVar α*
using *subst-trm-id-on-tvars*[*of single-subst* (*TyVar β*) *α v α*] *alpha-v* **by** *simp*
then show *False* **using** *beta-nv alpha-v* **by** *simp*
qed
have *wtv′*: ⊢ *v′* **unfolding** *v′-def* **using** *subst-wt*[*OF wtv*] **.**
have *fv′*: *fterm v′* **unfolding** *v′-def* **using** *fv* **by** (*induction v*) *auto*

s ⊑ *v′* using ⟦*?s* ⊑ *subst-trm ?ϱ ?v*; ⋀*α p*. ⟦*p* ∈ *poss ?v*; *α* ∈ *tvars-mty* (*mtp-of ?v p*); *mtp-of ?s p* ≠ *None*⟧ ⟹ *?ϱ α* = *?ϱ′ α*⟧ ⟹ *?s* ⊑ *subst-trm ?ϱ′ ?v*.

have *sv′*: *s* ⊑ *v′*
proof −
have *s* ⊑ *subst-trm TyVar v* **using** *sv* **by** *simp*
then show *s* ⊑ *v′* **unfolding** *v′-def*
proof (*rule multiple-compl*)
fix *α′ p* **assume** *p* ∈ *poss v α′* ∈ *tvars-mty* (*mtp-of v p*) *mtp-of s p* ≠ *None*
then have *α* ∉ *tvars-mty* (*mtp-of v p*) **using** *prop-a* **by** *auto*

then have $\alpha' \neq \alpha$ **using** ‹$\alpha' \in$ *tvars-mty* (*mtp-of v p*)› **by** *auto*
then show *TyVar* α' = *single-subst* (*TyVar* β) α α' **by** (*simp add*: *single-subst-def*)
qed
qed

v' is most general.

have $\bigwedge u$. *is-wt-completion s u* $\Longrightarrow \exists \varrho$. u = *subst-trm* ϱ v'
proof −
fix u **assume** *is-wt-completion s u*
then obtain ϱ **where** u = *subst-trm* ϱ v **using** *mg-prop* **by** *auto*
have v = *subst-trm* (*single-subst* (*TyVar* α) β) v'
unfolding *v'-def* **using** *subst-trm-swap*[*OF beta-nv*] **by** *simp*
then have u = *subst-trm* ($\varrho \circ\circ$ *single-subst* (*TyVar* α) β) v'
using ‹u = *subst-trm* ϱ v› *subst-trm-comp* **by** *metis*
then show $\exists \varrho$. u = *subst-trm* ϱ v' **by** *blast*
qed
then have *is-mgen s v'*
using *sv' fv' wtv'* **by** (*auto simp*: *is-mgen-def is-wt-completion-def*)
with *v-neq* **show** *?thesis* **by** *auto*
qed

Lift distinct completions to distinct most general completions.

lemma *lift-to-most-general*:
fixes s :: ‹('tv,'k,'v,'c) *trm*›
assumes *unambiguous s is-wt-completion s u is-wt-completion s u'* $u \neq u'$
obtains v v' **where** ‹$v \neq v'$› ‹*is-mgen s v*› ‹*is-mgen s v'*›
proof −
have *ty-s*: *typeable s* **using** *assms*(*2*) **by** (*auto simp*: *typeable-def*)
have *mg*: *is-mgen s* (*mgen s*) **using** *mgen-is-mgen*[*OF ty-s assms*(*1*)] **.**
obtain ϱ **where** *u-eq*: u = *subst-trm* ϱ (*mgen s*)
using *mgen-most-general*[*OF ty-s assms*(*1*,*2*)] **by** *auto*
obtain ϱ' **where** *u'-eq*: u' = *subst-trm* ϱ' (*mgen s*)
using *mgen-most-general*[*OF ty-s assms*(*1*,*3*)] **by** *auto*
from *assms*(*4*) *u-eq u'-eq* **have** $\exists \alpha \in$ *tvars-trm* (*mgen s*). $\varrho\ \alpha \neq \varrho'\ \alpha$
by (*auto simp*: *subst-trm-agree*)
then obtain α **where** *alpha-v*: $\alpha \in$ *tvars-trm* (*mgen s*) **and** *rho-neq*: $\varrho\ \alpha \neq \varrho'\ \alpha$ **by** *auto*
have *sv*: $s \sqsubseteq$ *mgen s* **using** *mgen-compl*[*OF ty-s assms*(*1*)] **.**
have $\alpha \notin$ *tvars-trm s*
proof
assume $\alpha \in$ *tvars-trm s*
then obtain p **where** *ps*: $p \in$ *poss s* **and** *alpha-s*: $\alpha \in$ *tvars-mty* (*mtp-of s p*)
using *tvars-trm-eq-UN-mtp* **by** *fastforce*
then have *snone*: *mtp-of s p* $\neq$ *None* **by** (*cases mtp-of s p*) *auto*
have *pv*: $p \in$ *poss* (*mgen s*) **using** *ps compl-poss-eq*[*OF sv*] **by** *simp*
have *mtp-of s p* = *mtp-of* (*mgen s*) *p*
using *compl-mtp-of*[*OF sv*, *of p*] *snone* **by** (*auto simp*: *mcompl-def*)
then have *alpha-vp*: $\alpha \in$ *tvars-mty* (*mtp-of* (*mgen s*) *p*) **using** *alpha-s* **by** *simp*

have *u-at*: *mtp-of u p* = *subst-mty ϱ* (*mtp-of* (*mgen s*) *p*)
using *mtp-of-subst*[*OF pv*] *u-eq* **by** *auto*
have *u′-at*: *mtp-of u′ p* = *subst-mty ϱ′* (*mtp-of* (*mgen s*) *p*)
using *mtp-of-subst*[*OF pv*] *u′-eq* **by** *auto*
have *s* ⊑ *u* **using** *assms*(*2*) **by** (*auto simp*: *is-wt-completion-def*)
then have *mtp-of u p* = *mtp-of s p*
using *compl-mtp-of*[*of - - p*] *snone* **by** (*fastforce simp*: *mcompl-def*)
moreover have *s* ⊑ *u′* **using** *assms*(*3*) **by** (*auto simp*: *is-wt-completion-def*)
then have *mtp-of u′ p* = *mtp-of s p*
using *compl-mtp-of*[*of - - p*] *snone* **by** (*fastforce simp*: *mcompl-def*)
ultimately have *subst-mty ϱ* (*mtp-of* (*mgen s*) *p*) = *subst-mty ϱ′* (*mtp-of* (*mgen s*) *p*)
using *u-at u′-at* **by** *simp*
then have *ϱ α* = *ϱ′ α* **using** *alpha-vp*
by (*cases mtp-of* (*mgen s*) *p*) (*auto simp*: *subst-ty-agree*)
with *rho-neq* **show** *False* **by** *contradiction*
qed
with *alpha-v* **have** *tvars-trm* (*mgen s*) − *tvars-trm s* ≠ {} **by** *auto*
from *two-mgen*[*OF assms*(*1*) *mg this*]
obtain *v′* **where** *v′* ≠ *mgen s is-mgen s v′* **by** *auto*
with *mg that* **show** *?thesis*
by *auto*
qed

corollary *strong-correct-printing-iff*:
assumes ‹*unambiguous s*›
shows ‹*strong-correct-printing t s* ⟷ *correct-printing t s*›
using *assms*
unfolding *strong-correct-printing-def correct-printing-def*
using *is-mgen-def typeable-def mgen-is-mgen*[*of s*] *lift-to-most-general*[*of s* ‹*mgen t*›]
by *metis*

1.6 The Algorithm

The *test* predicate, *decrease* function, and the algorithm *smobla*. We work in a locale that also fixes a position-picking function *pickPos*.

definition *non-bot-poss* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *nat list set* **where**
non-bot-poss s = {*p* ∈ *poss s*. *mtp-of s p* ≠ *None*}

definition *test* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ *nat list* ⇒ *bool* **where**
test v s p ⟷
p ∈ *poss s* ∩ *poss v* ∧ *mtp-of s p* ≠ *None* ∧
(∀ *α* ∈ *tvars-mty* (*mtp-of v p*).
∃ *q* ∈ *poss s* − {*p*}. *α* ∈ *tvars-mty* (*mtp-of v q*) ∧ *mtp-of s q* ≠ *None*)

end

locale *algorithm* = *signature-with-mgen ctpOf*

```
  for ctpOf :: 'c ⇒ ('tv,'k) ty +
  fixes pickPos :: ('tv,'k,'v,'c) trm ⇒ ('tv,'k,'v,'c) trm ⇒ nat list
  assumes compat: test v s p ⟹ test v s (pickPos v s)
begin

function decrease :: ('tv,'k,'v,'c) trm ⇒ ('tv,'k,'v,'c) trm ⇒ ('tv,'k,'v,'c) trm where
  decrease v s = (if ∃ p. test v s p
    then decrease v (remove-annot s (pickPos v s))
    else s)
  by auto

termination
proof (relation measure (λ(v, s). card (non-bot-poss s)))
  show wf (measure (λ(v, s). card (non-bot-poss s))) by simp
next
  fix v s :: ('tv,'k,'v,'c) trm
  assume ∃ p. test v s p
  then obtain p0 where t0: test v s p0 by auto
  then have t: test v s (pickPos v s) using compat by auto
  then have pp: pickPos v s ∈ poss s and nb: mtp-of s (pickPos v s) ≠ None
    by (auto simp: test-def)
  have rm: mtp-of (remove-annot s (pickPos v s)) (pickPos v s) = None
    using mtp-of-remove-self pp by auto
  have ps: poss (remove-annot s (pickPos v s)) = poss s
    using remove-annot-poss pp by auto
  have sub: non-bot-poss (remove-annot s (pickPos v s)) ⊂ non-bot-poss s
  proof (intro psubsetI subsetI)
    fix q assume q-in: q ∈ non-bot-poss (remove-annot s (pickPos v s))
    then have qp: q ∈ poss s and qnb: mtp-of (remove-annot s (pickPos v s)) q ≠
None
      by (auto simp: non-bot-poss-def ps)
    have q ≠ pickPos v s using qnb rm by auto
   then have mtp-of s q ≠ None using mtp-of-remove-other[OF pp - ‹q ≠ pickPos
v s›] qp qnb
      by (auto simp: compl-poss-eq)
    then show q ∈ non-bot-poss s using qp by (auto simp: non-bot-poss-def)
  next
    show non-bot-poss (remove-annot s (pickPos v s)) ≠ non-bot-poss s
    proof
      assume eq: non-bot-poss (remove-annot s (pickPos v s)) = non-bot-poss s
    have pickPos v s ∈ non-bot-poss s using pp nb by (auto simp: non-bot-poss-def)
      then have pickPos v s ∈ non-bot-poss (remove-annot s (pickPos v s)) using
eq by simp
      then show False using rm by (auto simp: non-bot-poss-def)
    qed
  qed
  then show ((v, remove-annot s (pickPos v s)), v, s)
    ∈ measure (λ(v, s). card (non-bot-poss s))
    using psubset-card-mono[OF finite-subset[OF - poss-finite[of s]]]
```

by (*auto simp*: *non-bot-poss-def*)
qed

lemmas *decrease.simps*[*simp del*]

definition *smobla* :: ('*tv*,'*k*,'*v*,'*c*) *trm* ⇒ ('*tv*,'*k*,'*v*,'*c*) *trm* **where**
smobla t = *decrease* (*mgen* (*erase t*)) (*mgen t*)

1.7 Completeness

Auxiliary: decrease only removes annotations, so *decrease v s* ⊑ *s*.

lemma *decrease-compl*:
decrease v s ⊑ *s*
proof (*induction v s rule*: *decrease.induct*)
case (*1 v s*)
show *?case*
using *compat compl-trans*[*OF 1 remove-annot-compl*]
by (*force simp*: *test-def decrease.simps*[*of v s*])
qed

Auxiliary: *erase t* ⊑ *decrease v s* if *erase t* ⊑ *s*.

lemma *erase-compl-decrease*:
erase t ⊑ *s* ⟹ *erase t* ⊑ *decrease v s*
proof (*induction v s rule*: *decrease.induct*)
case (*1 v s*)
show *?case*
using *compat erase-compl-remove-annot*[*OF 1*(*2*)]
by (*auto simp*: *test-def decrease.simps*[*of v s*] *intro*: *1*)
qed

The decrease invariant: coverage is preserved by *decrease*.

lemma *decrease-coverage*:
assumes *fterm v s* ⊑ *subst-trm* ϱ *v*
∀ α ∈ *tvars-trm v*. ∃ *p* ∈ *poss v*. α ∈ *tvars-mty* (*mtp-of v p*) ∧ *mtp-of s p* ≠ *None*
shows ∀ α ∈ *tvars-trm v*. ∃ *p* ∈ *poss v*. α ∈ *tvars-mty* (*mtp-of v p*) ∧
mtp-of (*decrease v s*) *p* ≠ *None*
using *assms*(*2*,*3*)
proof (*induction v s rule*: *decrease.induct*)
case (*1 v s*)
show *?case*
proof (*cases* ∃ *p*. *test v s p*)
case *False*
then have *decrease v s* = *s* **by** (*metis decrease.simps*)
with *1*(*3*) **show** *?thesis* **by** *simp*
next
case *True*
then have *t*: *test v s* (*pickPos v s*) **using** *compat* **by** *auto*

```
    then have pp: pickPos v s ∈ poss v by (auto simp: test-def)
    have rm-compl: remove-annot s (pickPos v s) ⊑ subst-trm ϱ v
      using compl-trans[OF remove-annot-compl 1(2)] t by (auto simp: test-def)
    have rm-cov: ∀ α ∈ tvars-trm v. ∃ p ∈ poss v. α ∈ tvars-mty (mtp-of v p) ∧
      mtp-of (remove-annot s (pickPos v s)) p ≠ None
    proof (intro ballI)
      fix α assume α ∈ tvars-trm v
      with 1(3) obtain q where qv: q ∈ poss v α ∈ tvars-mty (mtp-of v q)
        mtp-of s q ≠ None by auto
      have ps-eq: poss s = poss v
        using compl-poss-eq[OF 1(2)] by simp
      show ∃ p ∈ poss v. α ∈ tvars-mty (mtp-of v p) ∧
        mtp-of (remove-annot s (pickPos v s)) p ≠ None
      proof (cases q = pickPos v s)
        case False
        then have mtp-of (remove-annot s (pickPos v s)) q = mtp-of s q
          using mtp-of-remove-other t qv(1) ps-eq by (auto simp: test-def)
        with qv show ?thesis by auto
      next
        case True
```

$q = pickPos\ v\ s$, so $mtp\text{-}of\ s\ q$ is removed. But by the test condition, α is covered by some other position.

```
        from t True have ∃ q' ∈ poss s − {q}. α ∈ tvars-mty (mtp-of v q') ∧ mtp-of
s q' ≠ None
          using qv(2) by (auto simp: test-def)
        then obtain q' where q'v: q' ∈ poss v q' ≠ pickPos v s
          α ∈ tvars-mty (mtp-of v q') mtp-of s q' ≠ None
          using ps-eq True by auto
        have mtp-of (remove-annot s (pickPos v s)) q' = mtp-of s q'
          using mtp-of-remove-other t q'v(1,2) ps-eq by (auto simp: test-def)
        with q'v show ?thesis by auto
      qed
    qed
    from 1(1)[OF True rm-compl rm-cov] True
    show ?thesis by (metis decrease.simps)
  qed
qed

theorem completeness:
  assumes ty: typeable t and ua: unambiguous t and ct: cterm t
  shows ‹correct-printing t (smobla t)›
proof −
  let ?v = mgen (erase t)
  let ?s = smobla t
```

Basic setup: $erase\ t$ is typeable and unambiguous.

```
  have er-ty: typeable (erase t)
    using ty by (auto simp: typeable-def is-wt-completion-def
```

intro: *compl-trans*[*OF erase-compl*])
have *er-ua*: *unambiguous* (*erase t*) **using** *ua* **by** *simp*
have *t-ty-ua*: *typeable t unambiguous t* **using** *ty ua* **by** *auto*

Part 1: *erase t* $\sqsubseteq$ *smobla t* and *smobla t* $\sqsubseteq$ *mgen t*.

have *s-def*: *?s* = *decrease ?v* (*mgen t*) **by** (*simp add*: *smobla-def*)
have *compl1*: *erase t* $\sqsubseteq$ *mgen t* **using** *erase-compl mgen-compl*[*OF ty ua*] *compl-trans* **by** *blast*
have *s-compl-mgen*: *?s* $\sqsubseteq$ *mgen t* **using** *decrease-compl s-def* **by** *simp*
have *er-compl-s*: *erase t* $\sqsubseteq$ *?s* **using** *erase-compl-decrease*[*OF compl1*] *s-def* **by** *simp*

mgen t is a well-typed F-term completion of *smobla t*.

have *wt-completion*: *is-wt-completion ?s* (*mgen t*)
unfolding *is-wt-completion-def* **using** *s-compl-mgen mgen-fterm*[*OF ty ua*] *mgen-wt*[*OF ty ua*] **by** *auto*

Part 2: uniqueness. Any well-typed completion of *smobla t* must equal *mgen t*.

have *unique*: *u* = *mgen t* **if** *u-wc*: ‹*is-wt-completion ?s u*› **for** *u*
proof −
from *u-wc* **have** *su*: *?s* $\sqsubseteq$ *u* **and** *fu*: *fterm u* **and** *wtu*: ⊢ *u*
by (*auto simp*: *is-wt-completion-def*)

u is a completion of *erase t*, hence *u* is an instance of *mgen* (*erase t*).

have *er-u*: *erase t* $\sqsubseteq$ *u* **using** *compl-trans*[*OF er-compl-s su*] **.**
have *is-wt-completion* (*erase t*) *u*
unfolding *is-wt-completion-def* **using** *er-u fu wtu* **by** *auto*
then obtain *ϱ′* **where** *u-inst*: *u* = *subst-trm ϱ′ ?v*
using *mgen-most-general*[*OF er-ty er-ua*] **by** *auto*

Similarly, *mgen t* is a completion of *erase t*, hence *mgen t* is an instance of *mgen* (*erase t*).

have *is-wt-completion* (*erase t*) (*mgen t*)
unfolding *is-wt-completion-def* **using** *compl1 mgen-fterm*[*OF ty ua*] *mgen-wt*[*OF ty ua*] **by** *auto*
then obtain *ϱ* **where** *mgen-inst*: *mgen t* = *subst-trm ϱ ?v*
using *mgen-most-general*[*OF er-ty er-ua*] **by** *auto*

The coverage condition: every tyvar of *mgen* (*erase t*) is witnessed at a non-bot position of *smobla t*.

have *fv*: *fterm ?v* **using** *mgen-fterm*[*OF er-ty er-ua*] **.**
have *cov*: ∀ *α* ∈ *tvars-trm ?v*. ∃ *p* ∈ *poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧ *mtp-of* (*mgen t*) *p* ≠ *None*
proof (*intro ballI*)
fix *α* **assume** *α* ∈ *tvars-trm ?v*
then obtain *p* **where** *p* ∈ *poss ?v α* ∈ *tvars-mty* (*mtp-of ?v p*)

using *tvars-trm-eq-UN-mtp* **by** *fastforce*
moreover have *mtp-of* (*mgen t*) *p* ≠ *None*
proof −
have *p* ∈ *poss* (*mgen t*) **using** ‹*p* ∈ *poss ?v*› *compl-poss-eq*[*OF compl1*]
compl-poss-eq[*OF mgen-compl*[*OF er-ty er-ua*]] **by** *simp*
then show *?thesis* **using** *fterm-mtp-of-Some*[*OF mgen-fterm*[*OF ty ua*]] **by** *fastforce*
qed
ultimately show ∃ *p* ∈ *poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧ *mtp-of* (*mgen t*) *p* ≠ *None*
by *auto*
qed
have *s-compl-subst*: *?s* ⊑ *subst-trm ϱ ?v* **using** *s-compl-mgen mgen-inst* **by** *simp*
have *cov-mgen*: ∀ *α* ∈ *tvars-trm ?v*. ∃ *p* ∈ *poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧
mtp-of (*mgen t*) *p* ≠ *None* **using** *cov* **.**
have *cov-s*: ∀ *α* ∈ *tvars-trm ?v*. ∃ *p* ∈ *poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧
mtp-of ?s p ≠ *None*
proof (*intro ballI*)
fix *α* **assume** *α* ∈ *tvars-trm ?v*
have *dec-cov*: ∀ *α* ∈ *tvars-trm ?v*. ∃ *p* ∈ *poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧
mtp-of (*decrease ?v* (*mgen t*)) *p* ≠ *None*
proof −
have *mgen t* ⊑ *subst-trm ϱ ?v* **using** *mgen-inst* **by** *simp*
then show *?thesis* **using** *decrease-coverage*[*OF fv*] *cov-mgen* **by** *blast*
qed
with ‹*α* ∈ *tvars-trm ?v*› **obtain** *p* **where** *p* ∈ *poss ?v*
α ∈ *tvars-mty* (*mtp-of ?v p*) *mtp-of* (*decrease ?v* (*mgen t*)) *p* ≠ *None*
by *auto*
then show ∃ *p* ∈ *poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧ *mtp-of ?s p* ≠ *None*
unfolding *s-def* **by** *blast*
qed

Apply ⟦*?t* ⊑ *?u*; *fterm ?u*; ∃ *ϱ*. *?u* = *subst-trm ϱ ?v*; *fterm ?v*; ∀ *α*∈*tvars-trm ?v*. ∃ *p*∈*poss ?v*. *α* ∈ *tvars-mty* (*mtp-of ?v p*) ∧ *mtp-of ?t p* ≠ *None*; *?t* ⊑ *?u′*; *fterm ?u′*; ∃ *ϱ′*. *?u′* = *subst-trm ϱ′ ?v*⟧ ⟹ *?u* = *?u′*.

have ∃ *ϱ*. *mgen t* = *subst-trm ϱ ?v* **using** *mgen-inst* **by** *auto*
moreover have ∃ *ϱ′*. *u* = *subst-trm ϱ′ ?v* **using** *u-inst* **by** *auto*
show *u* = *mgen t*
proof −
have *mgen t* = *u*
by (*rule crucial-uniqueness*[*OF s-compl-mgen mgen-fterm*[*OF ty ua*]
‹∃ *ϱ*. *mgen t* = *subst-trm ϱ ?v*› *fv cov-s su fu* ‹∃ *ϱ′*. *u* = *subst-trm ϱ′ ?v*›])
then show *?thesis* **by** *simp*
qed
qed

show *?thesis*
using *assms wt-completion unique mgen-is-mgen*
unfolding *correct-printing-def is-mgen-def*
by *fastforce+*
qed

1.8 Minimality

Supporting lemmas for minimality.

After decrease, no position passes the test.

lemma *decrease-no-test*:
$\neg$*test v* (*decrease v s*) *p*
proof (*induction v s rule*: *decrease.induct*)
case (*1 v s*)
show *?case*
proof (*cases* $\exists$ *p*. *test v s p*)
case *False*
then show *?thesis* **by** (*metis decrease.simps*)
next
case *True*
then have *decrease v s* = *decrease v* (*remove-annot s* (*pickPos v s*))
by (*metis decrease.simps*)
with *1 True* **show** *?thesis* **by** *simp*
qed
qed

Decrease preserves unambiguity.

lemma *decrease-unambiguous*:
unambiguous s $\Longrightarrow$ *unambiguous* (*decrease v s*)
proof (*induction v s rule*: *decrease.induct*)
case (*1 v s*)
show *?case*
proof (*cases* $\exists$ *p*. *test v s p*)
case *False*
then show *?thesis* **using** *1*(*2*) **by** (*metis decrease.simps*)
next
case *True*
then have *pp*: *pickPos v s* $\in$ *poss s* **using** *compat* **by** (*auto simp*: *test-def*)
have *unambiguous* (*remove-annot s* (*pickPos v s*))
using *remove-annot-unambiguous*[*OF pp*] *1*(*2*) **by** *auto*
then have *unambiguous* (*decrease v* (*remove-annot s* (*pickPos v s*)))
using *1*(*1*)[*OF True*] **by** *auto*
with *True* **show** *?thesis* **by** (*metis decrease.simps*)
qed
qed

Coverage minimality.

theorem *smobla-distinct-completions*:

assumes *ty*: *typeable t* **and** *ua*: *unambiguous t* **and** *ct*: *cterm t*
and *ua'*: *unambiguous s'* **and** *s'-compl*: $s' \sqsubseteq$ *smobla t* **and** *s'-neq*: $s' \neq$ *smobla t*
obtains *v v'* **where** $v \neq v'$ *is-mgen s' v is-mgen s' v'*
proof −
let *?s* = *smobla t*
let *?v* = *mgen* (*erase t*)
have *s-ua*: *unambiguous ?s*
using *decrease-unambiguous compl-unambiguous*[*OF mgen-compl*[*OF ty ua*]] *ua*
unfolding *smobla-def* **by** *metis*

Get a position where s' has *None* but *smobla t* has non-*None*.

from *compl-strict-witness*[*OF s'-compl s'-neq*]
obtain *p* **where** *ps*: $p \in$ *poss ?s* **and** *s'-none*: *mtp-of s' p* = *None* **and** *s-some*: *mtp-of ?s p* $\neq$ *None*
by *auto*

Since not *test* (*mgen* (*erase t*)) (*smobla t*) *p*, there exists a tyvar uncovered by *smobla t*.

have *no-test*: ¬ *test ?v ?s p* **using** *decrease-no-test* **unfolding** *smobla-def* **by** *blast*

From completeness, *mgen t* is the unique well-typed completion of *smobla t*. $s' \sqsubseteq$ *smobla t* and *smobla t* $\sqsubseteq$ *mgen t*, so *mgen t* is a completion of s'. Any other completion of s' gives two distinct completions.

have *s-compl-mgen*: *?s* $\sqsubseteq$ *mgen t*
using *decrease-compl smobla-def* **by** *metis*
have *s'-compl-mgen*: *is-wt-completion s'* (*mgen t*)
using *compl-trans*[*OF s'-compl s-compl-mgen*] *mgen-fterm*[*OF ty ua*] *mgen-wt*[*OF ty ua*]
by (*auto simp*: *is-wt-completion-def*)

Construct a second distinct completion using the uncovered tyvar.

have *er-ty*: *typeable* (*erase t*)
using *ty* **by** (*auto simp*: *typeable-def is-wt-completion-def intro*: *compl-trans*[*OF erase-compl*])
have *er-ua*: *unambiguous* (*erase t*) **using** *ua* **by** *simp*
obtain *ϱ* **where** *mgen-inst*: *mgen t* = *subst-trm ϱ ?v*
using *mgen-most-general*[*OF er-ty er-ua*]
compl-trans[*OF erase-compl mgen-compl*[*OF ty ua*]] *mgen-fterm*[*OF ty ua*] *mgen-wt*[*OF ty ua*]
by (*auto simp*: *is-wt-completion-def*)
have *fv*: *fterm ?v* **using** *mgen-fterm*[*OF er-ty er-ua*] **.**
have *wtv*: ⊢ *?v* **using** *mgen-wt*[*OF er-ty er-ua*] **.**
have *poss-sv*: *poss ?s* = *poss ?v*
using *compl-poss-eq*[*OF s-compl-mgen*] *mgen-inst* **by** *simp*
have *pv*: $p \in$ *poss ?v* **using** *ps poss-sv* **by** *simp*

Extract tyvar from the negated test.

from *no-test ps pv s-some* **obtain** α **where** *alpha-vp*: $\alpha \in$ *tvars-mty* (*mtp-of ?v p*)
and *uncov*: $\forall\, q \in$ *poss ?s* $-\ \{p\}.\ \alpha \in$ *tvars-mty* (*mtp-of ?v q*) $\longrightarrow$ *mtp-of ?s q* = *None*
unfolding *test-def* **by** *blast*

Property (c): α is uncovered in s'.

have *poss-s'*: *poss* s' = *poss ?s* **using** *compl-poss-eq*[*OF s'-compl*] **.**
have *prop-c*: ‹*mtp-of* s' *q* = *None*› **if** ‹$q \in$ *poss* s'› **and** ‹$\alpha \in$ *tvars-mty* (*mtp-of ?v q*)› **for** *q*
using *compl-mtp-of*[*OF s'-compl*, *of q*] *that uncov*
by (*auto simp*: *mcompl-def s'-none poss-s'*)
have *s'-compl-vrho*: $s' \sqsubseteq$ *subst-trm* ϱ *?v*
using *compl-trans*[*OF s'-compl s-compl-mgen*] *mgen-inst* **by** *simp*

Construct type differing from ϱ at α. Simplest: *TyVar* with a fresh variable, always well-typed.

obtain *w* :: (*'tv*,*'k*) *ty* **where** *w-neq*: $w \neq \varrho\ \alpha$
using *ty.distinct*(*1*) **by** *metis*
define ϱ' **where** $\varrho' = (\lambda\beta.$ *if* $\beta = \alpha$ *then w else* $\varrho\ \beta)$

(2) ϱ and ϱ' agree on non-*None* positions of s'.

have *agree*: $\varrho\ \beta = \varrho'\ \beta$ **if** $q \in$ *poss ?v* $\beta \in$ *tvars-mty* (*mtp-of ?v q*) *mtp-of* s' $q \neq$ *None*
for β *q* **using** *that* ϱ'*-def poss-s' poss-sv prop-c* **by** *auto*
have *s'-compl-vrho'*: $s' \sqsubseteq$ *subst-trm* ϱ' *?v*
using *multiple-compl*[*OF s'-compl-vrho agree*] **.**
have *wtvrho'*: $\vdash$ *subst-trm* ϱ' *?v* **using** *subst-wt*[*OF wtv*] **.**
have $\alpha \in$ *tvars-trm ?v* **using** *alpha-vp tvars-trm-eq-UN-mtp pv* **by** *fastforce*
then have *vrho-neq*: *subst-trm* ϱ *?v* $\neq$ *subst-trm* ϱ' *?v*
using *w-neq* **by** (*auto simp*: *subst-trm-agree* ϱ'*-def*)

Two distinct completions of s'.

have *wc1*: *is-wt-completion* s' (*subst-trm* ϱ *?v*)
using *s'-compl-vrho fv subst-wt*[*OF wtv*] **by** (*auto simp*: *is-wt-completion-def*)
have *wc2*: *is-wt-completion* s' (*subst-trm* ϱ' *?v*)
using *s'-compl-vrho' fv wtvrho'* **by** (*auto simp*: *is-wt-completion-def*)
from *lift-to-most-general*[*OF ua' wc1 wc2 vrho-neq*] **show** *?thesis* **using** *that* **by** *metis*
qed

corollary *minimality*:
assumes *typeable t unambiguous t cterm t* ‹$s \sqsubseteq$ *smobla t*› ‹*correct-printing t s*›
shows ‹s = *smobla t*›
using *assms smobla-distinct-completions*
unfolding *correct-printing-def*
by (*metis mgen-compl decrease-unambiguous compl-unambiguous smobla-def*)

end

end
theory *Smolka-AI*
imports *Main*
begin

2 AI-Authored proof formalization of the roundtrip algorithm

Formalization of the Smolka-Blanchette printing–parsing roundtrip algorithm for Isabelle.

The algorithm takes a fully-typed term and selects a locally minimal and complete set of type annotations so the term can be unambiguously re-parsed via Hindley–Milner type inference.

2.1 Types and Type Substitutions

Types are built from type variables and type constructors with a fixed arity (implicit in the list length).

datatype *ty* =
TVar string
| *TCons string ty list*

We define the function type constructor as a distinguished binary constructor.

definition *fun-ty* :: *ty* ⇒ *ty* ⇒ *ty* (**infixr** → *65*) **where**
fun-ty τ_1 τ_2 = *TCons* ″*fun*″ [τ_1, τ_2]

Type variables occurring in a type.

fun *tvars-ty* :: *ty* ⇒ *string set* **where**
tvars-ty (*TVar v*) = {*v*}
| *tvars-ty* (*TCons* - *ts*) = ⋃(*set* (*map tvars-ty ts*))

Definition (Type Substitution). A type substitution is a function from variable names to types. It extends homomorphically to types.

fun *subst-ty* :: (*string* ⇒ *ty*) ⇒ *ty* ⇒ *ty* **where**
subst-ty σ (*TVar v*) = σ *v*
| *subst-ty* σ (*TCons k ts*) = *TCons k* (*map* (*subst-ty* σ) *ts*)

The arrow type $\tau \to \tau$ is strictly larger than τ, hence $\tau \to \tau \neq \tau$. This is used in the minimality proof. The proof uses the built-in *size* function from the datatype package.

lemma *arrow-neq-self*: $\tau \to \tau \neq \tau$

proof
  assume $\tau \to \tau = \tau$
  then have *size* $(\tau \to \tau) =$ *size* τ **by** *simp*
  then show *False* **by** (*simp add*: *fun-ty-def*)
qed

Lemma (Uniqueness of Type Matching). If two substitutions agree when applied to a type, they agree on all type variables of that type.

lemma *unique-type-match*:
  subst-ty $\sigma_1\ \tau =$ *subst-ty* $\sigma_2\ \tau \Longrightarrow \alpha \in$ *tvars-ty* $\tau \Longrightarrow \sigma_1\ \alpha = \sigma_2\ \alpha$
  by (*induction* τ) *auto*

Substitution on type variables: the domain is precisely *tvars-ty* τ.

lemma *tvars-subst-ty*: *tvars-ty* (*subst-ty* $\sigma\ \tau$) $= \bigcup$(*tvars-ty* ‘ σ ‘ *tvars-ty* τ)
  by (*induction* τ) *auto*

lemma *subst-ty-id*:
  $(\bigwedge v.\ v \in$ *tvars-ty* $\tau \Longrightarrow \sigma\ v =$ *TVar* $v) \Longrightarrow$ *subst-ty* $\sigma\ \tau = \tau$
  by (*induction* τ) (*auto intro*: *map-idI*)

lemma *subst-ty-agree*:
  $(\bigwedge v.\ v \in$ *tvars-ty* $\tau \Longrightarrow \sigma_1\ v = \sigma_2\ v) \Longrightarrow$ *subst-ty* $\sigma_1\ \tau =$ *subst-ty* $\sigma_2\ \tau$
  by (*induction* τ) *auto*

lemma *subst-ty-compose*:
  subst-ty σ_1 (*subst-ty* $\sigma_2\ \tau$) $=$ *subst-ty* $(\lambda v.$ *subst-ty* $\sigma_1\ (\sigma_2\ v))\ \tau$
  by (*induction* τ) *auto*

2.2 Contexts

A context is a finite partial function from variable names to types.

type-synonym *ctx* $=$ *string* $\rightharpoonup$ *ty*

Type variables of a context.

definition *tvars-ctx* :: *ctx* $\Rightarrow$ *string set* **where**
  tvars-ctx $\Gamma = \bigcup$(*tvars-ty* ‘ *ran* Γ)

Applying a type substitution to a context.

definition *subst-ctx* :: (*string* $\Rightarrow$ *ty*) $\Rightarrow$ *ctx* $\Rightarrow$ *ctx* **where**
  subst-ctx $\sigma\ \Gamma =$ *map-option* (*subst-ty* σ) $\circ\ \Gamma$

lemma *subst-ctx-dom* [*simp*]: *dom* (*subst-ctx* $\sigma\ \Gamma$) $=$ *dom* Γ
  by (*auto simp*: *subst-ctx-def dom-def*)

lemma *subst-ctx-app*: $x \in$ *dom* $\Gamma \Longrightarrow$ *subst-ctx* $\sigma\ \Gamma\ x =$ *Some* (*subst-ty* σ (*the* (Γ x)))
  by (*auto simp*: *subst-ctx-def dom-def*)

lemma *subst-ctx-update*:
subst-ctx σ $(\Gamma(x \mapsto \tau)) = (\textit{subst-ctx}\ \sigma\ \Gamma)(x \mapsto \textit{subst-ty}\ \sigma\ \tau)$
by (*auto simp*: *subst-ctx-def*)

lemma *subst-ctx-id*:
$(\bigwedge v.\ v \in \textit{tvars-ctx}\ \Gamma \Longrightarrow \sigma\ v = \textit{TVar}\ v) \Longrightarrow \textit{subst-ctx}\ \sigma\ \Gamma = \Gamma$
using *ranI* **by** (*fastforce intro*!: *map-option-idI subst-ty-id simp*: *subst-ctx-def tvars-ctx-def*)

2.3 Terms

2.3.1 Raw Terms

Raw terms include type constraints and binder constraints as annotations.

datatype *raw-term* =
RConst string
| *RVar string*
| *RAbs string raw-term*
| *RAbsT string ty raw-term*
| *RApp raw-term raw-term*
| *RConstrain raw-term ty*

The function *strip* removes all constraints from a raw term.

fun *strip* :: *raw-term* ⇒ *raw-term* **where**
strip (*RConst c*) = *RConst c*
| *strip* (*RVar x*) = *RVar x*
| *strip* (*RAbs x t*) = *RAbs x* (*strip t*)
| *strip* (*RAbsT x - t*) = *RAbs x* (*strip t*)
| *strip* (*RApp* t_1 t_2) = *RApp* (*strip* t_1) (*strip* t_2)
| *strip* (*RConstrain t -*) = *strip t*

A term is constraint-free if stripping is the identity.

fun *constraint-free* :: *raw-term* ⇒ *bool* **where**
constraint-free (*RConst -*) = *True*
| *constraint-free* (*RVar -*) = *True*
| *constraint-free* (*RAbs - t*) = *constraint-free t*
| *constraint-free* (*RAbsT - - -*) = *False*
| *constraint-free* (*RApp* t_1 t_2) = (*constraint-free* t_1 ∧ *constraint-free* t_2)
| *constraint-free* (*RConstrain - -*) = *False*

lemma *strip-idem* [*simp*]: *strip* (*strip t*) = *strip t*
by (*induction t*) *auto*

lemma *strip-constraint-free*: *constraint-free* (*strip t*)
by (*induction t*) *auto*

2.3.2 Annotated Terms

Definition (Annotated Terms). Every node carries exactly one type annotation.

```
datatype aterm =
  AConst string ty
| AVar string ty
| AAbs string ty aterm ty
| AApp aterm aterm ty
```

Definition (Type of an Annotated Term). Reads the outermost type.

```
fun typeof-aterm :: aterm ⇒ ty where
  typeof-aterm (AConst - τ) = τ
| typeof-aterm (AVar - τ) = τ
| typeof-aterm (AAbs - - - τ′) = τ′
| typeof-aterm (AApp - - τ) = τ
```

Definition (Erasure). Strips all types from an annotated term, producing a constraint-free raw term.

```
fun erase :: aterm ⇒ raw-term where
  erase (AConst c -) = RConst c
| erase (AVar x -) = RVar x
| erase (AAbs x - a -) = RAbs x (erase a)
| erase (AApp a₁ a₂ -) = RApp (erase a₁) (erase a₂)

lemma erase-constraint-free: constraint-free (erase a)
  by (induction a) auto

lemma strip-erase [simp]: strip (erase a) = erase a
  by (induction a) auto
```

Definition (Type Substitution on Annotated Terms).

```
fun subst-aterm :: (string ⇒ ty) ⇒ aterm ⇒ aterm where
  subst-aterm σ (AConst c τ) = AConst c (subst-ty σ τ)
| subst-aterm σ (AVar x τ) = AVar x (subst-ty σ τ)
| subst-aterm σ (AAbs x τ a τ′) = AAbs x (subst-ty σ τ) (subst-aterm σ a) (subst-ty
σ τ′)
| subst-aterm σ (AApp a₁ a₂ τ) = AApp (subst-aterm σ a₁) (subst-aterm σ a₂)
(subst-ty σ τ)
```

Key property: erasure is invariant under type substitution.

```
lemma erase-subst [simp]: erase (subst-aterm σ a) = erase a
  by (induction a) auto
```

Key property: *typeof-aterm* commutes with type substitution.

```
lemma typeof-subst [simp]: typeof-aterm (subst-aterm σ a) = subst-ty σ (typeof-aterm
a)
```

by (*cases a*) *auto*

Type variables of an annotated term.

fun *tvars-aterm* :: *aterm* ⇒ *string set* **where**
  tvars-aterm (*AConst* - *τ*) = *tvars-ty τ*
| *tvars-aterm* (*AVar* - *τ*) = *tvars-ty τ*
| *tvars-aterm* (*AAbs* - *τ a τ′*) = *tvars-ty τ* ∪ *tvars-aterm a* ∪ *tvars-ty τ′*
| *tvars-aterm* (*AApp* a_1 a_2 *τ*) = *tvars-aterm* a_1 ∪ *tvars-aterm* a_2 ∪ *tvars-ty τ*

If two substitutions agree on the type variables of a term, they give the same result.

lemma *subst-aterm-agree*:
  (⋀*v*. *v* ∈ *tvars-aterm a* ⟹ σ_1 *v* = σ_2 *v*) ⟹ *subst-aterm* σ_1 *a* = *subst-aterm* σ_2 *a*
  by (*induction a*) (*auto intro*: *subst-ty-agree*)

Converse: if two substitutions give the same annotated term, they agree on the type variables of that term. This is "injectivity" of substitution on annotated terms.

lemma *subst-aterm-injective*:
  subst-aterm σ_1 *a* = *subst-aterm* σ_2 *a* ⟹ *α* ∈ *tvars-aterm a* ⟹ σ_1 *α* = σ_2 *α*
  using *unique-type-match* **by** (*induction a arbitrary*: *α*) *auto*

lemma *subst-ty-TVar* [*simp*]: *subst-ty TVar τ* = *τ*
  by (*induction τ*) (*auto simp*: *map-idI*)

lemma *subst-aterm-id*:
  (⋀*v*. *v* ∈ *tvars-aterm a* ⟹ *σ v* = *TVar v*) ⟹ *subst-aterm σ a* = *a*
  by (*induction a*) (*auto intro*: *subst-ty-id*)

2.3.3 Well-Typedness

We parameterize the development by a function giving the declared type scheme of each constant. This is the type scheme that must be instantiated at each use site.

An annotated term is well-typed in a context if each node satisfies the appropriate typing constraint.

inductive *well-typed* :: (*string* ⇒ *ty*) ⇒ *ctx* ⇒ *aterm* ⇒ *bool* **where**
  wt-const: ∃ *ϱ*. *τ* = *subst-ty ϱ* (*const-type c*) ⟹
    well-typed const-type Γ (*AConst c τ*)
| *wt-var*: Γ *x* = *Some τ* ⟹
    well-typed const-type Γ (*AVar x τ*)
| *wt-abs*: *well-typed const-type* (Γ(*x* ↦ *τ*)) *a* ⟹
    τ′ = *τ* → *typeof-aterm a* ⟹
    well-typed const-type Γ (*AAbs x τ a τ′*)
| *wt-app*: *well-typed const-type* Γ a_1 ⟹

well-typed const-type Γ a_2 ⟹
typeof-aterm a_1 = *typeof-aterm* a_2 → τ ⟹
well-typed const-type Γ (*AApp* a_1 a_2 τ)

Lemma (Substitution Preserves Well-Typedness). If a is well-typed in Γ, then *subst-aterm* σ a is well-typed in *subst-ctx* σ Γ.

lemma *subst-preserves-wt*:
well-typed ct Γ a ⟹ *well-typed ct* (*subst-ctx* σ Γ) (*subst-aterm* σ a)
proof (*induction rule*: *well-typed.induct*)
case (*wt-const* τ *ct c* Γ)
then obtain ϱ **where** τ = *subst-ty* ϱ (*ct c*) **by** *auto*
then have *subst-ty* σ τ = *subst-ty* (λ*v*. *subst-ty* σ (ϱ *v*)) (*ct c*)
by (*simp add*: *subst-ty-compose*)
then show *?case* **by** (*auto intro*: *well-typed.wt-const*)
next
case (*wt-var* Γ *x* τ *ct*)
then show *?case* **by** (*auto simp*: *subst-ctx-def intro*: *well-typed.wt-var*)
next
case (*wt-abs ct* Γ *x* τ *a* τ′)
then have *ih*: *well-typed ct* ((*subst-ctx* σ Γ)(*x* ↦ *subst-ty* σ τ)) (*subst-aterm* σ *a*)
using *subst-ctx-update* **by** *metis*
from *wt-abs.hyps*(*2*) **have** *subst-ty* σ τ′ = *subst-ty* σ τ → *typeof-aterm* (*subst-aterm* σ *a*)
by (*simp add*: *fun-ty-def*)
with *ih* **show** *?case* **by** (*auto intro*: *well-typed.wt-abs*)
next
case (*wt-app ct* Γ a_1 a_2 τ)
from *wt-app.hyps*(*3*) **have**
typeof-aterm (*subst-aterm* σ a_1) = *typeof-aterm* (*subst-aterm* σ a_2) → *subst-ty* σ τ
by (*simp add*: *fun-ty-def*)
with *wt-app.IH* **show** *?case* **by** (*auto intro*: *well-typed.wt-app*)
qed

Corollary: if σ is the identity on *tvars-ctx* Γ, then *subst-ctx* σ Γ = Γ, so *subst-aterm* σ a is well-typed in Γ.

corollary *subst-wt-identity-ctx*:
well-typed ct Γ a ⟹ (⋀*v*. *v* ∈ *tvars-ctx* Γ ⟹ σ *v* = *TVar v*) ⟹
well-typed ct Γ (*subst-aterm* σ a)
using *subst-preserves-wt*[*of ct* Γ a σ] *subst-ctx-id*[*of* Γ σ] **by** *simp*

2.4 Positions and Post-Order Enumeration

Definition (Post-Order Enumeration). For a fully annotated term a, the function *enum-aterm a* returns a list of triples (p, s, τ) where p is a position number (in post-order), s is the raw subterm, and τ is the type at that position.

The helper *shift-enum k L* adds k to all position numbers in L.

definition *shift-enum* :: *nat* ⇒ (*nat* × *raw-term* × *ty*) *list* ⇒ (*nat* × *raw-term* × *ty*) *list* **where**
shift-enum k L = *map* (λ(*p*, *s*, τ). (*k* + *p*, *s*, τ)) *L*

fun *enum-aterm* :: *aterm* ⇒ (*nat* × *raw-term* × *ty*) *list* **where**
enum-aterm (*AConst c* τ) = [(*0*, *RConst c*, τ)]
| *enum-aterm* (*AVar x* τ) = [(*0*, *RVar x*, τ)]
| *enum-aterm* (*AAbs x* τ *a*$_1$ τ′) =
(*let L* = *enum-aterm a*$_1$; *n* = *length L in*
[(*0*, *RVar x*, τ)] @ *shift-enum 1 L* @ [(*1* + *n*, *RAbs x* (*erase a*$_1$), τ′)])
| *enum-aterm* (*AApp a*$_1$ *a*$_2$ τ) =
(*let L*$_1$ = *enum-aterm a*$_1$; *n*$_1$ = *length L*$_1$;
L$_2$ = *enum-aterm a*$_2$; *n*$_2$ = *length L*$_2$ *in*
L$_1$ @ *shift-enum n*$_1$ *L*$_2$ @ [(*n*$_1$ + *n*$_2$, *RApp* (*erase a*$_1$) (*erase a*$_2$), τ)])

The set of positions.

definition *pos-set* :: *aterm* ⇒ *nat set* **where**
pos-set a = *fst* ‘ *set* (*enum-aterm a*)

The type at a given position in the enumeration.

definition *type-at-pos* :: *aterm* ⇒ *nat* ⇒ *ty* **where**
type-at-pos a p = (*THE* τ. ∃ *s*. (*p*, *s*, τ) ∈ *set* (*enum-aterm a*))

The subterm at a given position in the enumeration.

definition *subterm-at-pos* :: *aterm* ⇒ *nat* ⇒ *raw-term* **where**
subterm-at-pos a p = (*THE s*. ∃ τ. (*p*, *s*, τ) ∈ *set* (*enum-aterm a*))

2.4.1 Basic Properties of Enumeration

lemma *length-shift-enum* [*simp*]: *length* (*shift-enum k L*) = *length L*
by (*simp add*: *shift-enum-def*)

lemma *set-shift-enum*: *set* (*shift-enum k L*) = (λ(*p*, *s*, τ). (*k* + *p*, *s*, τ)) ‘ *set L*
by (*auto simp*: *shift-enum-def*)

lemma *in-shift-enum-iff*:
(*p*, *s*, τ) ∈ *set* (*shift-enum k L*) ⟷ (∃ *p*′. *p* = *k* + *p*′ ∧ (*p*′, *s*, τ) ∈ *set L*)
by (*auto simp*: *shift-enum-def image-iff split*: *prod.splits*)

lemma *shift-enum-shift-enum*:
shift-enum k$_1$ (*shift-enum k*$_2$ *L*) = *shift-enum* (*k*$_1$ + *k*$_2$) *L*
by (*induction L*) (*auto simp*: *shift-enum-def*)

The length of the enumeration equals the number of nodes in the term. For an annotated term, each node produces one entry, with abstractions additionally producing a binder position.

fun *aterm-node-count* :: *aterm* ⇒ *nat* **where**

aterm-node-count (*AConst* - -) = *1*
| *aterm-node-count* (*AVar* - -) = *1*
| *aterm-node-count* (*AAbs* - - *a* -) = *2* + *aterm-node-count a*
| *aterm-node-count* (*AApp* a_1 a_2 -) = *1* + *aterm-node-count* a_1 + *aterm-node-count* a_2

lemma *length-enum-aterm*: *length* (*enum-aterm a*) = *aterm-node-count a*
by (*induction a*) (*auto simp*: *Let-def*)

Every annotated term has at least one position.

lemma *enum-aterm-nonempty*: *enum-aterm a* ≠ []
by (*cases a*) (*auto simp*: *Let-def*)

lemma *aterm-node-count-pos*: *aterm-node-count a* ≥ *1*
by (*cases a*) *auto*

2.4.2 Substitution Distributes to Positions

Lemma (Substitution Distributes to Positions). The enumeration of *subst-aterm* σ *a* is obtained from the enumeration of *a* by applying σ to each type, leaving position numbers and subterms unchanged.

We first define the operation that applies a substitution to the types in an enumeration list.

definition *map-enum-ty* ::
(*string* ⇒ *ty*) ⇒ (*nat* × *raw-term* × *ty*) *list* ⇒ (*nat* × *raw-term* × *ty*) *list* **where**
map-enum-ty σ *L* = *map* (λ(*p*, *s*, τ). (*p*, *s*, *subst-ty* σ τ)) *L*

lemma *length-map-enum-ty* [*simp*]: *length* (*map-enum-ty* σ *L*) = *length L*
by (*simp add*: *map-enum-ty-def*)

lemma *map-enum-ty-shift*: *map-enum-ty* σ (*shift-enum k L*) = *shift-enum k* (*map-enum-ty* σ *L*)
by (*induction L*) (*auto simp*: *map-enum-ty-def shift-enum-def*)

lemma *map-enum-ty-append*: *map-enum-ty* σ (L_1 @ L_2) = *map-enum-ty* σ L_1 @ *map-enum-ty* σ L_2
by (*simp add*: *map-enum-ty-def*)

The main lemma: the enumeration commutes with type substitution.

lemma *enum-subst-aterm*:
enum-aterm (*subst-aterm* σ *a*) = *map-enum-ty* σ (*enum-aterm a*)
by (*induction a*) (*auto simp*: *map-enum-ty-def shift-enum-def Let-def split*: *prod.splits*)

corollary *type-at-pos-subst-iff*:
(*p*, *s*, τ') ∈ *set* (*enum-aterm* (*subst-aterm* σ *a*)) ⟷
(∃ τ. (*p*, *s*, τ) ∈ *set* (*enum-aterm a*) ∧ τ' = *subst-ty* σ τ)
by (*auto simp*: *enum-subst-aterm map-enum-ty-def image-iff split*: *prod.splits*)

corollary *type-at-pos-subst*:
(*p*, *s*, *τ*) ∈ *set* (*enum-aterm a*) ⟹ (*p*, *s*, *subst-ty σ τ*) ∈ *set* (*enum-aterm* (*subst-aterm σ a*))
using *type-at-pos-subst-iff* **by** *auto*

The node count is invariant under type substitution.

lemma *aterm-node-count-subst* [*simp*]:
aterm-node-count (*subst-aterm σ a*) = *aterm-node-count a*
by (*induction a*) *auto*

The position set is invariant under type substitution, and more generally, is the same for any two terms with the same erasure. We prove the substitution case first.

lemma *pos-set-subst* [*simp*]: *pos-set* (*subst-aterm σ a*) = *pos-set a*
unfolding *pos-set-def*
by (*force simp*: *map-enum-ty-def enum-subst-aterm*)

All type variables of *a* appear in some type annotation at a position.

lemma *tvars-aterm-subset-enum*:
α ∈ *tvars-aterm a* ⟹ ∃ *p s τ*. (*p*, *s*, *τ*) ∈ *set* (*enum-aterm a*) ∧ *α* ∈ *tvars-ty τ*
by (*induction a*) (*auto simp*: *Let-def in-shift-enum-iff*)

Converse: every type at an enumeration position contributes to the term's type variables.

lemma *enum-tvars-subset-aterm*:
(*p*, *s*, *τ*) ∈ *set* (*enum-aterm a*) ⟹ *tvars-ty τ* ⊆ *tvars-aterm a*
by (*induction a arbitrary*: *p s τ*) (*fastforce simp*: *Let-def in-shift-enum-iff*)+

2.4.3 Distinctness and Range of Position Numbers

Position numbers in the enumeration form a contiguous range starting from 0. This gives distinctness of positions as an immediate corollary.

lemma *map-fst-shift-enum*: *map fst* (*shift-enum k L*) = *map* ((+) *k*) (*map fst L*)
by (*induction L*) (*auto simp*: *shift-enum-def*)

lemma *map-fst-enum-aterm*: *map fst* (*enum-aterm a*) = [*0*..<*aterm-node-count a*]
using *upt-add-eq-append*[*symmetric*] *map-add-upt*
by (*induction a*)
(*auto simp*: *upt-conv-Cons Let-def map-fst-shift-enum length-enum-aterm add.commute*)

Corollary: position numbers are distinct.

lemma *distinct-enum-fst*: *distinct* (*map fst* (*enum-aterm a*))
by (*simp add*: *map-fst-enum-aterm*)

Corollary: the position set is exactly the range {*0*..<*aterm-node-count a*}.

lemma *pos-set-range*: *pos-set a* = {*0*..<*aterm-node-count a*}
unfolding *pos-set-def*

by (*metis map-fst-enum-aterm set-map set-upt*)

A useful corollary: if $(p,\ s,\ \tau)$ and $(p,\ s',\ \tau')$ are both in the enumeration, then $s = s'$ and $\tau = \tau'$ (positions are unique keys).

lemma *enum-aterm-unique*:
assumes $(p,\ s,\ \tau) \in$ *set* (*enum-aterm a*) $(p,\ s',\ \tau') \in$ *set* (*enum-aterm a*)
shows $s = s'\ \tau = \tau'$
using *assms eq-key-imp-eq-value distinct-enum-fst*
by *fastforce+*

Note: The coverage lemma is proved below in *annotation-problem* locale as *coverage-initial*. The locale instantiation connecting the reverse-greedy algorithm to *annotation-selection* is also done below via the *rg* interpretation.

2.5 The Annotation Algorithm

2.5.1 Type Inference Assumption

Assumption (Type Inference). We work in a locale that fixes: - *const-type*: the type scheme of each constant - Γ: a fixed context - *a*: a fully annotated, well-typed term in Γ - *a-star*: the principal typing (generalized term) - σ: the matching substitution with σ *a-star* = *a*:

locale *annotation-problem* =
fixes *const-type* :: *string* ⇒ *ty*
and Γ :: *ctx*
and *a* :: *aterm*
and *a-star* :: *aterm*
and σ :: *string* ⇒ *ty*
assumes *a-wt*: *well-typed const-type* Γ *a*
and *a-star-wt*: *well-typed const-type* Γ *a-star*
and *same-erasure*: *erase a-star* = *erase a*
and *matching*: *subst-aterm* σ *a-star* = *a*
and *freshness*: $\bigwedge\alpha.\ \alpha \in$ *tvars-ctx* $\Gamma \Longrightarrow \sigma\ \alpha =$ *TVar* α
begin

Definition (Inference Variables). The type variables introduced by inference that are not present in the context.

definition *V* :: *string set* **where**
V = *tvars-aterm a-star* − *tvars-ctx* Γ

Definition (Key). The inference variables visible at a position. We define *key p* as the set of inference variables that appear in any type annotation at position *p* in the enumeration of *a-star*, intersected with *V*. This avoids the need to show that position numbers are unique in the enumeration.

definition *key* :: *nat* ⇒ *string set* **where**
key p = $\{\alpha.\ \exists s\ \tau.\ (p,\ s,\ \tau) \in$ *set* (*enum-aterm a-star*) $\wedge\ \alpha \in$ *tvars-ty* $\tau\} \cap V$

The σ is the identity on *tvars-ctx* Γ, hence moves only variables in V.

lemma *sigma-id-on-ctx*: $\alpha \in$ *tvars-ctx* $\Gamma \Longrightarrow \sigma\ \alpha = TVar\ \alpha$
using *freshness* **by** *simp*

Positions of *a-star* and a coincide.

lemma *pos-set-eq*: *pos-set a* = *pos-set a-star*
using *matching pos-set-subst*[*of* σ *a-star*] **by** *simp*

Lemma (Coverage). Every inference variable appears in the key of some position. This follows directly from the fact that every type variable of *a-star* appears at some position in the enumeration, and V is a subset of *tvars-aterm a-star*.

lemma *coverage-initial*:
assumes $\alpha \in V$
shows $\exists\, p \in$ *pos-set a-star*. $\alpha \in$ *key p*
using *assms tvars-aterm-subset-enum* **unfolding** *pos-set-def key-def V-def* **by** *force*

Consistency of a term a' with annotations at a set of positions P': for every position $p \in P'$, the type at position p in a' agrees with the type at position p in a (i.e., with *subst-ty* $\sigma\ \tau$ where τ is the type at position p in *a-star*).

This directly follows the paper's definition: since *erase* a' = *erase a-star*, position sets coincide, and consistency means that for each $p \in P'$ the type at position p in a' equals *subst-ty* $\sigma\ \tau$ where $(p,\ s,\ \tau) \in$ *set* (*enum-aterm a-star*).

definition *consistent-with* :: *aterm* $\Rightarrow$ *nat set* $\Rightarrow$ *bool* **where**
consistent-with $a'\ P' \equiv$
$\forall\, p\ s\ \tau.\ p \in P' \longrightarrow (p,\ s,\ \tau) \in$ *set* (*enum-aterm a-star*) $\longrightarrow$
$(\exists\, \tau'.\ (p,\ s,\ \tau') \in$ *set* (*enum-aterm* a') $\wedge\ \tau' =$ *subst-ty* $\sigma\ \tau$)

lemma *consistent-withI*:
$(\bigwedge p\ s\ \tau.\ p \in P' \Longrightarrow (p,\ s,\ \tau) \in$ *set* (*enum-aterm a-star*) $\Longrightarrow$
$\exists\, \tau'.\ (p,\ s,\ \tau') \in$ *set* (*enum-aterm* a') $\wedge\ \tau' =$ *subst-ty* $\sigma\ \tau$)
$\Longrightarrow$ *consistent-with* $a'\ P'$
unfolding *consistent-with-def* **by** *auto*

lemma *consistent-withE*:
assumes *consistent-with* $a'\ P'\ p \in P'\ (p,\ s,\ \tau) \in$ *set* (*enum-aterm a-star*)
obtains τ' **where** $(p,\ s,\ \tau') \in$ *set* (*enum-aterm* a') $\tau' =$ *subst-ty* $\sigma\ \tau$
using *assms* **unfolding** *consistent-with-def* **by** *auto*

lemma *consistent-with-mono*:
consistent-with $a'\ P' \Longrightarrow Q \subseteq P' \Longrightarrow$ *consistent-with* $a'\ Q$
unfolding *consistent-with-def* **by** *auto*

When a' is a substitution instance of *a-star*, i.e., $a' =$ *subst-aterm* σ' *a-star*, consistency at P' reduces to type agreement: *subst-ty* $\sigma'\ \tau =$ *subst-ty* $\sigma\ \tau$ for all positions $p \in P'$. This is the form used in the completeness proof.

lemma *consistent-with-substI*:
assumes $\bigwedge p\ s\ \tau.\ p \in P' \Longrightarrow (p,\ s,\ \tau) \in set\ (enum\text{-}aterm\ a\text{-}star) \Longrightarrow$
$subst\text{-}ty\ \sigma'\ \tau = subst\text{-}ty\ \sigma\ \tau$
shows *consistent-with* $(subst\text{-}aterm\ \sigma'\ a\text{-}star)\ P'$
proof (*rule consistent-withI*)
fix $p\ s\ \tau$
assume $p \in P'\ (p,\ s,\ \tau) \in set\ (enum\text{-}aterm\ a\text{-}star)$
then have $subst\text{-}ty\ \sigma'\ \tau = subst\text{-}ty\ \sigma\ \tau$ **using** *assms* **by** *auto*
moreover have $(p,\ s,\ subst\text{-}ty\ \sigma'\ \tau) \in set\ (enum\text{-}aterm\ (subst\text{-}aterm\ \sigma'\ a\text{-}star))$
using $\langle (p,\ s,\ \tau) \in set\ (enum\text{-}aterm\ a\text{-}star)\rangle$ *type-at-pos-subst* **by** *auto*
ultimately show $\exists \tau'.\ (p,\ s,\ \tau') \in set\ (enum\text{-}aterm\ (subst\text{-}aterm\ \sigma'\ a\text{-}star)) \wedge$
$\tau' = subst\text{-}ty\ \sigma\ \tau$
by *auto*
qed

lemma *consistent-with-substD*:
assumes *consistent-with* $(subst\text{-}aterm\ \sigma'\ a\text{-}star)\ P'$
$p \in P'\ (p,\ s,\ \tau) \in set\ (enum\text{-}aterm\ a\text{-}star)$
shows $subst\text{-}ty\ \sigma'\ \tau = subst\text{-}ty\ \sigma\ \tau$
using *consistent-withE*[*OF assms*] *assms enum-aterm-unique type-at-pos-subst*
by *metis*

end

An annotation selection extends the annotation problem with a finite set of positions P that covers all inference variables and where each position has a witness variable (a variable in its key not appearing in any other position's key in P). We separate the proof of the main theorems from the algorithm that produces P.

locale *annotation-selection* = *annotation-problem* +
fixes P :: *nat set*
assumes *P-subset*: $P \subseteq pos\text{-}set\ a\text{-}star$
and *coverage*: $\bigcup (key\ `\ P) = V$
and *witness*: $\bigwedge p.\ p \in P \Longrightarrow \exists \alpha \in key\ p.\ \forall p' \in P.\ p' \neq p \longrightarrow \alpha \notin key\ p'$
begin

Every inference variable appears in the key of some kept position.

lemma *coverage-mem*: $\alpha \in V \Longrightarrow \exists p \in P.\ \alpha \in key\ p$
using *coverage* **by** *blast*

2.5.2 Annotations determine the substitution

lemma *sigma-agree-on-V*:
assumes *agreement*: $\bigwedge p\ s\ \tau.\ p \in P \Longrightarrow (p,\ s,\ \tau) \in set\ (enum\text{-}aterm\ a\text{-}star) \Longrightarrow$
$subst\text{-}ty\ \sigma'\ \tau = subst\text{-}ty\ \sigma\ \tau$
and $\langle \alpha \in V\rangle$
shows $\sigma'\ \alpha = \sigma\ \alpha$
proof −
obtain p **where** $p \in P\ \alpha \in key\ p$ **using** *coverage-mem* $\langle \alpha \in V\rangle$ **by** *blast*

from ‹$\alpha \in$ *key p*› **obtain** *s* τ **where**
mem: $(p, s, \tau) \in$ *set* (*enum-aterm a-star*) **and** *var*: $\alpha \in$ *tvars-ty* τ
unfolding *key-def* **by** *auto*
from ‹$p \in P$› *mem* **have** *subst-ty* σ' τ = *subst-ty* σ τ **using** *agreement* **by** *auto*
with *var* **show** σ' α = σ α **using** *unique-type-match* **by** *blast*
qed

The main result:

lemma *annotations-determine-subst*:
assumes *agreement*: $\bigwedge p\ s\ \tau.\ p \in P \Longrightarrow (p, s, \tau) \in$ *set* (*enum-aterm a-star*) $\Longrightarrow$
subst-ty σ' τ = *subst-ty* σ τ
and *fresh'*: $\bigwedge\alpha.\ \alpha \in$ *tvars-ctx* $\Gamma \Longrightarrow \sigma'\ \alpha$ = *TVar* α
shows *subst-aterm* σ' *a-star* = *a*
proof −

We show $\sigma' = \sigma$ on *tvars-aterm a-star*, then conclude by ($\bigwedge v.\ v \in$ *tvars-aterm ?a* $\Longrightarrow$ *?*σ_1 *v* = *?*σ_2 *v*) $\Longrightarrow$ *subst-aterm ?*σ_1 *?a* = *subst-aterm ?*σ_2 *?a* and *subst-aterm* σ *a-star* = *a*.

have *agree*: σ' α = σ α **if** $\alpha \in$ *tvars-aterm a-star* **for** α
proof (*cases* $\alpha \in V$)
case *True*
then show *?thesis*
using *sigma-agree-on-V*[*OF agreement*] **by** *simp*
next
case *False*
with *that* **have** $\alpha \in$ *tvars-ctx* Γ **unfolding** *V-def* **by** *auto*
then have σ' α = *TVar* α **using** *fresh'* **by** *auto*
moreover have σ α = *TVar* α **using** ‹$\alpha \in$ *tvars-ctx* Γ› *freshness* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed
have *subst-aterm* σ' *a-star* = *subst-aterm* σ *a-star*
by (*intro subst-aterm-agree agree*)
also have ... = *a* **using** *matching* **by** *simp*
finally show *?thesis* **.**
qed

2.5.3 Completeness

Theorem (Completeness). Any well-typed a' with *erase* a' = *erase* a and consistent with the constraints at P satisfies $a' = a$.

The principality assumption gives: any well-typed term with the same erasure is a substitution instance of *a-star*, where the substitution is identity on context variables.

theorem *completeness*:
assumes *a'-wt*: *well-typed const-type* Γ a'
and *a'-erase*: *erase* a' = *erase a*
and *principality*: $\exists\sigma'.$ *subst-aterm* σ' *a-star* = a'

```
                    ∧ (∀ α ∈ tvars-ctx Γ. σ' α = TVar α)
    and consist: consistent-with a' P
  shows a' = a
proof −
  from principality obtain σ' where
    inst: subst-aterm σ' a-star = a' and
    fresh': ∀ α ∈ tvars-ctx Γ. σ' α = TVar α
    by blast
  have a' = subst-aterm σ' a-star using inst by simp
  also have ... = a
  proof (rule annotations-determine-subst)
    fix p s τ
    assume p ∈ P (p, s, τ) ∈ set (enum-aterm a-star)
    then show subst-ty σ' τ = subst-ty σ τ
      using consistent-with-substD[OF consist[folded inst] ‹p ∈ P› ‹(p, s, τ) ∈ set
(enum-aterm a-star)›] by auto
  next
    fix α
    assume α ∈ tvars-ctx Γ
    then show σ' α = TVar α using fresh' by auto
  qed
  finally show a' = a .
qed
```

2.5.4 Local Minimality

Theorem (Local Minimality). For every $p \in P$, removing the annotation at position p makes the typing non-unique: there exists a' different from a that is well-typed in Γ with *erase* $a' =$ *erase* a and consistent with the annotations at $P - \{p\}$.

```
theorem local-minimality:
  assumes p ∈ P
  shows ∃ a'. a' ≠ a
    ∧ well-typed const-type Γ a'
    ∧ erase a' = erase a
    ∧ consistent-with a' (P − {p})
proof −
```

Step 1: Obtain witness variable.

```
  from witness[OF ‹p ∈ P›] obtain α where
    α-in-key: α ∈ key p and
    α-unique: ∀ p' ∈ P. p' ≠ p ⟶ α ∉ key p'
    by blast
```

Step 2: Define the altered substitution.

```
  define σ-star where σ-star = (λv. if v = α then σ α → σ α else σ v)
```

Step 3: Define a' and show it satisfies all four properties.

define a' **where** $a' = \mathit{subst\text{-}aterm}\ \sigma\text{-}\mathit{star}\ a\text{-}\mathit{star}$

Property 1: a' differs from a.

have $\sigma\text{-}\mathit{star}\text{-}\mathit{diff}$: $\sigma\text{-}\mathit{star}\ \alpha \neq \sigma\ \alpha$
unfolding $\sigma\text{-}\mathit{star}\text{-}\mathit{def}$ **using** *arrow-neq-self* **by** *simp*

have $\alpha \in V$ **using** $\alpha\text{-}\mathit{in}\text{-}\mathit{key}$ **unfolding** *key-def* **by** *auto*
then have $\alpha \in \mathit{tvars\text{-}aterm}\ a\text{-}\mathit{star}$ **unfolding** *V-def* **by** *auto*

have $a' \neq a$
proof
assume $a' = a$
then have $\mathit{subst\text{-}aterm}\ \sigma\text{-}\mathit{star}\ a\text{-}\mathit{star} = \mathit{subst\text{-}aterm}\ \sigma\ a\text{-}\mathit{star}$ **using** *matching* $a'\text{-}\mathit{def}$ **by** *simp*
then have $\sigma\text{-}\mathit{star}\ \alpha = \sigma\ \alpha$
using ‹$\alpha \in \mathit{tvars\text{-}aterm}\ a\text{-}\mathit{star}$› *subst-aterm-injective* **by** *blast*
with $\sigma\text{-}\mathit{star}\text{-}\mathit{diff}$ **show** *False* **by** *contradiction*
qed

Property 2: a' is well-typed.

have *fresh-star*: $\alpha' \in \mathit{tvars\text{-}ctx}\ \Gamma \Longrightarrow \sigma\text{-}\mathit{star}\ \alpha' = \mathit{TVar}\ \alpha'$ **for** α'
proof −
assume $\alpha' \in \mathit{tvars\text{-}ctx}\ \Gamma$
then have $\alpha' \notin V$ **unfolding** *V-def* **by** *auto*
then have $\alpha' \neq \alpha$ **using** ‹$\alpha \in V$› **by** *auto*
then have $\sigma\text{-}\mathit{star}\ \alpha' = \sigma\ \alpha'$ **unfolding** $\sigma\text{-}\mathit{star}\text{-}\mathit{def}$ **by** *simp*
also have $\ldots = \mathit{TVar}\ \alpha'$ **using** ‹$\alpha' \in \mathit{tvars\text{-}ctx}\ \Gamma$› *freshness* **by** *auto*
finally show $\sigma\text{-}\mathit{star}\ \alpha' = \mathit{TVar}\ \alpha'$ **.**
qed
have $\mathit{well\text{-}typed}\ \mathit{const\text{-}type}\ \Gamma\ a'$
unfolding $a'\text{-}\mathit{def}$ **using** $\mathit{subst\text{-}wt\text{-}identity\text{-}ctx}[\mathit{OF}\ a\text{-}\mathit{star}\text{-}\mathit{wt}\ \mathit{fresh\text{-}star}]$ **.**

Property 3: $\mathit{erase}\ a' = \mathit{erase}\ a$.

have $\mathit{erase}\ a' = \mathit{erase}\ a$
unfolding $a'\text{-}\mathit{def}$ **using** *erase-subst same-erasure* **by** *simp*

Property 4: a' is consistent with $P - \{p\}$.

have *consistent-minus-p*: $\mathit{consistent\text{-}with}\ a'\ (P - \{p\})$
unfolding $a'\text{-}\mathit{def}$
proof (*rule consistent-with-substI*)
fix $p'\ s\ \tau$
assume $p' \in P - \{p\}$ $(p', s, \tau) \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ a\text{-}\mathit{star})$
then have $p' \in P\ p' \neq p$ **by** *auto*

Step 1: α is not in $\mathit{tvars\text{-}ty}\ \tau$.

have $\alpha \notin \mathit{tvars\text{-}ty}\ \tau$
proof
assume $\alpha \in \mathit{tvars\text{-}ty}\ \tau$

```
    then have α ∈ key p'
      using ‹(p', s, τ) ∈ set (enum-aterm a-star)› ‹α ∈ V›
      unfolding key-def by auto
    with α-unique ‹p' ∈ P› ‹p' ≠ p› show False by auto
  qed
```

Step 2: σ-*star* $= \sigma$ on *tvars-ty* τ.

```
  have σ-star β = σ β if β ∈ tvars-ty τ for β
  proof −
    from that ‹α ∉ tvars-ty τ› have β ≠ α by auto
    then show ?thesis unfolding σ-star-def by simp
  qed
  then show subst-ty σ-star τ = subst-ty σ τ
    by (rule subst-ty-agree)
 qed

 from ‹a' ≠ a› ‹well-typed const-type Γ a'› ‹erase a' = erase a› consistent-minus-p
 show ?thesis by blast
qed

end
```

2.6 Annotation Insertion

Definition (Annotation Insertion). Given the matching substitution σ, the generalized term *a-star*, the selected positions P, and a starting position counter k, the function *ins* traverses the raw term and the annotated term in lockstep, inserting type constraints at positions in P.

Returns a pair (annotated raw term, number of positions traversed).

```
fun ins :: (string ⇒ ty) ⇒ aterm ⇒ nat set ⇒ nat ⇒ raw-term × nat where
  ins σ (AConst c τ) P k =
    (if k ∈ P then RConstrain (RConst c) (subst-ty σ τ) else RConst c, 1)
| ins σ (AVar x τ) P k =
    (if k ∈ P then RConstrain (RVar x) (subst-ty σ τ) else RVar x, 1)
| ins σ (AAbs x τ a₁ τ') P k =
    (let (t₁', n₁) = ins σ a₁ P (k + 1);
         t-binder = (if k ∈ P then RAbsT x (subst-ty σ τ) t₁'
                     else RAbs x t₁');
         t' = (if k + 1 + n₁ ∈ P then RConstrain t-binder (subst-ty σ τ')
               else t-binder)
     in (t', n₁ + 2))
| ins σ (AApp a₁ a₂ τ) P k =
    (let (t₁', n₁) = ins σ a₁ P k;
         (t₂', n₂) = ins σ a₂ P (k + n₁);
         t-app = RApp t₁' t₂';
         t' = (if k + n₁ + n₂ ∈ P then RConstrain t-app (subst-ty σ τ)
               else t-app)
     in (t', n₁ + n₂ + 1))
```

The top-level annotation function.

definition *annotate* :: (*string* ⇒ *ty*) ⇒ *aterm* ⇒ *nat set* ⇒ *raw-term* **where**
annotate σ a-star P = *fst* (*ins σ a-star P 0*)

The number of positions traversed equals the node count.

lemma *ins-count*: *snd* (*ins σ a P k*) = *aterm-node-count a*
by (*induction a arbitrary*: *k*) (*auto simp*: *Let-def case-prod-unfold*)

Stripping constraints from the output of ins recovers the erasure.

lemma *strip-ins*: *strip* (*fst* (*ins σ a P k*)) = *erase a*
by (*induction a arbitrary*: *k*) (*auto simp*: *Let-def case-prod-unfold*)

Corollary: annotate preserves the raw term under stripping.

corollary *strip-annotate*: *strip* (*annotate σ a P*) = *erase a*
unfolding *annotate-def* **using** *strip-ins* **by** *simp*

2.7 The Reverse-Greedy Algorithm

Definition (Reverse-Greedy Selection). The algorithm processes candidate positions in decreasing cost order. A position is dropped if every variable in its key has count > 1 (i.e., is covered by at least one other remaining candidate); otherwise it is kept.

We model the algorithm as a fold over the candidate list (sorted by decreasing cost). The state is a count function tracking how many undropped candidates cover each variable.

The fold processes from head (highest cost) to tail (lowest cost). When a position is kept, the count is unchanged. When a position is dropped, the count is decremented for each variable in its key.

One step: a position with key K is kept iff some variable in K has count ≤ 1.

definition *rg-keep* :: (*string* ⇒ *nat*) ⇒ *string set* ⇒ *bool* **where**
rg-keep cnt K = (∃ α ∈ *K*. *cnt α* ≤ *1*)

The fold: processes a list of (position, key) pairs. Returns (kept set, final count).

```
fun rg-fold :: (nat × string set) list ⇒ (string ⇒ nat) ⇒ nat set × (string ⇒ nat)
where
  rg-fold [] cnt = ({}, cnt)
| rg-fold ((p, K) # rest) cnt =
    (if rg-keep cnt K then
       let (P', cnt') = rg-fold rest cnt in (insert p P', cnt')
     else
       let cnt' = (λα. if α ∈ K then cnt α − 1 else cnt α) in
       let (P', cnt'') = rg-fold rest cnt' in (P', cnt''))
```

Initialize the count for each variable.

definition *init-count* :: (*nat* × *string set*) *list* ⇒ *string* ⇒ *nat* **where**
init-count cands α = *length* (*filter* (λ(-, *K*). *α* ∈ *K*) *cands*)

The full algorithm.

definition *reverse-greedy* :: (*nat* × *string set*) *list* ⇒ *nat set* **where**
reverse-greedy cands = *fst* (*rg-fold cands* (*init-count cands*))

2.7.1 Basic Properties of the Fold

The kept set is a subset of the candidates.

lemma *rg-fold-subset*: *fst* (*rg-fold cands cnt*) ⊆ *fst* ‘ *set cands*
by (*induction cands arbitrary*: *cnt*) (*fastforce split*: *if-splits simp*: *case-prod-unfold Let-def*)+

If a variable is not in any candidate's key, the count is unchanged by the fold.

lemma *rg-fold-cnt-unchanged*:
∀(*p*, *K*) ∈ *set cands*. *α* ∉ *K* ⟹ *snd* (*rg-fold cands cnt*) *α* = *cnt α*
by (*induction cands arbitrary*: *cnt*) (*auto simp*: *case-prod-unfold Let-def*)

The count never increases during the fold.

lemma *rg-fold-cnt-mono*: *snd* (*rg-fold cands cnt*) *α* ≤ *cnt α*
proof (*induction cands arbitrary*: *cnt*)
case (*Cons pc rest*)
thus *?case*
using *order.trans*[*OF Cons*]
by (*cases pc*) (*auto simp*: *Let-def case-prod-unfold*)
qed *simp*

If no kept position has *α* in its key, then the final count for *α* equals *0* (assuming the initial count equals the number of candidates covering *α*).

lemma *rg-fold-no-kept-zero*:
assumes *no-kept*: ∀ *p K*. (*p*, *K*) ∈ *set cands* ⟶ *α* ∈ *K* ⟶ *p* ∉ *fst* (*rg-fold cands cnt*)
and *cnt-eq*: *cnt α* = *length* (*filter* (λ(-, *K*). *α* ∈ *K*) *cands*)
shows *snd* (*rg-fold cands cnt*) *α* = *0*
using *assms*
by (*induction cands arbitrary*: *cnt*) (*auto simp*: *case-prod-unfold Let-def*)

2.7.2 Coverage

Lemma (Reverse-Greedy Preserves Full Coverage). The key invariant: the fold maintains *1* ≤ *cnt α* for every *α* that appears in some candidate's key. This is because: - At a keep step: count is unchanged (passes to recursive call as-is). - At a drop step: we only drop when all counts in *key p* are > *1*,

so after decrementing they are ≥ 1. For variables NOT in *key p*, the count is unchanged.

At termination, $1 \leq cnt\ \alpha$ means at least one undropped candidate covers α. But there are no more candidates to process, so every undropped candidate is a kept position. Hence α is covered by a kept position.

The formal proof combines this invariant with the fact that the final count equals the number of kept positions covering α.

The key invariant: if all counts are non-zero, they remain non-zero after the fold. This is because a position is only dropped when all variables in its key have count > 1.

```
lemma rg-fold-preserves-ge1:
  assumes ⋀α. cnt α ≥ 1
  shows snd (rg-fold cands cnt) α ≥ 1
  using assms
proof (induction cands arbitrary: cnt)
next
  case (Cons pc rest)
  thus ?case
   by (cases pc) (fastforce simp: case-prod-unfold Let-def rg-keep-def intro!: Cons[simplified])
qed simp
```

The fold result depends only on the count values for variables in candidate keys.

```
lemma rg-fold-cnt-agree:
  assumes ∀α. (∃(p, K) ∈ set cands. α ∈ K) ⟶ cnt1 α = cnt2 α
  shows fst (rg-fold cands cnt1) = fst (rg-fold cands cnt2)
    and ∀α. (∃(p, K) ∈ set cands. α ∈ K) ⟶
          snd (rg-fold cands cnt1) α = snd (rg-fold cands cnt2) α
proof −
  have both: fst (rg-fold cands cnt1) = fst (rg-fold cands cnt2) ∧
    (∀α. (∃(p, K) ∈ set cands. α ∈ K) ⟶
          snd (rg-fold cands cnt1) α = snd (rg-fold cands cnt2) α)
    using assms
  proof (induction cands arbitrary: cnt1 cnt2)
    case Nil then show ?case by simp
next
  case (Cons pc rest)
  obtain p K where [simp]: pc = (p, K) by (cases pc)
  have key-eq: ∀α ∈ K. cnt1 α = cnt2 α
    using Cons.prems by auto
  then have keep-eq: rg-keep cnt1 K = rg-keep cnt2 K
    unfolding rg-keep-def by auto
  show ?case
  proof (cases rg-keep cnt1 K)
    case True
    have rest-agree: ∀α. (∃(p, K) ∈ set rest. α ∈ K) ⟶ cnt1 α = cnt2 α
```

```
    using Cons.prems by auto
  from Cons.IH[OF rest-agree] have IH:
    fst (rg-fold rest cnt1) = fst (rg-fold rest cnt2)
    ∀α. (∃(p, K) ∈ set rest. α ∈ K) ⟶
          snd (rg-fold rest cnt1) α = snd (rg-fold rest cnt2) α
    by auto
  have snd-eq: ∀β. (∃(p, K) ∈ set (pc # rest). β ∈ K) ⟶
        snd (rg-fold (pc # rest) cnt1) β = snd (rg-fold (pc # rest) cnt2) β
  proof (intro allI impI)
    fix β assume ex: ∃(p, K) ∈ set (pc # rest). β ∈ K
    obtain P1 c1 P2 c2 where
      r1: rg-fold rest cnt1 = (P1, c1) and r2: rg-fold rest cnt2 = (P2, c2)
      by (cases rg-fold rest cnt1; cases rg-fold rest cnt2)
    from IH(1) r1 r2 have P1 = P2 by simp
    have fold1: rg-fold (pc # rest) cnt1 = (insert p P1, c1)
      using ‹rg-keep cnt1 K› r1 by (simp add: Let-def)
    have fold2: rg-fold (pc # rest) cnt2 = (insert p P2, c2)
      using ‹rg-keep cnt1 K› keep-eq r2 by (simp add: Let-def)
    show snd (rg-fold (pc # rest) cnt1) β = snd (rg-fold (pc # rest) cnt2) β
    proof (cases ∃(p, K) ∈ set rest. β ∈ K)
      case True
      from IH(2) True have c1 β = c2 β using r1 r2 by auto
      then show ?thesis using fold1 fold2 by simp
    next
      case False
      then have nk1: ∀(p, K) ∈ set rest. β ∉ K by auto
       have snd (rg-fold rest cnt1) β = cnt1 β using rg-fold-cnt-unchanged[OF
nk1] by simp
      then have c1 β = cnt1 β using r1 by simp
     moreover have snd (rg-fold rest cnt2) β = cnt2 β using rg-fold-cnt-unchanged[OF
nk1] by simp
      then have c2 β = cnt2 β using r2 by simp

      moreover from ex False have β ∈ K by auto
      then have cnt1 β = cnt2 β using key-eq by auto
      ultimately show ?thesis using fold1 fold2 by simp
    qed
  qed
  from IH(1) snd-eq True keep-eq
  show ?thesis by (auto simp: Let-def split: prod.splits)

next
  case False
  define cnt1' where cnt1' = (λα. if α ∈ K then cnt1 α − 1 else cnt1 α)
  define cnt2' where cnt2' = (λα. if α ∈ K then cnt2 α − 1 else cnt2 α)
  have rest-agree: ∀α. (∃(p, K) ∈ set rest. α ∈ K) ⟶ cnt1' α = cnt2' α
    using Cons.prems key-eq unfolding cnt1'-def cnt2'-def by auto
  from Cons.IH[OF rest-agree] have IH:
    fst (rg-fold rest cnt1') = fst (rg-fold rest cnt2')
```

```
    ∀α. (∃(p, K) ∈ set rest. α ∈ K) ⟶
          snd (rg-fold rest cnt1') α = snd (rg-fold rest cnt2') α
    by auto
  have snd-eq: ∀α. (∃(p, K) ∈ set (pc # rest). α ∈ K) ⟶
        snd (rg-fold (pc # rest) cnt1) α = snd (rg-fold (pc # rest) cnt2) α
  proof (intro allI impI)
    fix β assume ∃(p, K) ∈ set (pc # rest). β ∈ K
    then have β ∈ K ∨ (∃(p, K) ∈ set rest. β ∈ K) by auto
   then show snd (rg-fold (pc # rest) cnt1) β = snd (rg-fold (pc # rest) cnt2)
β
    proof
      assume β ∈ K
      show ?thesis
      proof (cases ∃(p, K) ∈ set rest. β ∈ K)
        case True
        then show ?thesis using IH(2) False keep-eq cnt1'-def cnt2'-def
          by (simp add: Let-def rg-keep-def split: prod.splits)
      next
        case Falser: False
        then have nk2: ∀(p, K) ∈ set rest. β ∉ K by auto
        have snd (rg-fold rest cnt1') β = cnt1' β
          using rg-fold-cnt-unchanged[OF nk2] by simp
        moreover have snd (rg-fold rest cnt2') β = cnt2' β
          using rg-fold-cnt-unchanged[OF nk2] by simp
        moreover have cnt1' β = cnt2' β
          using key-eq ‹β ∈ K› unfolding cnt1'-def cnt2'-def by auto
        ultimately show ?thesis using False keep-eq cnt1'-def cnt2'-def
          by (simp add: Let-def rg-keep-def split: prod.splits)

      qed
    next
      assume ∃(p, K) ∈ set rest. β ∈ K
      then show ?thesis using IH(2) False keep-eq cnt1'-def cnt2'-def
        by (simp add: Let-def rg-keep-def split: prod.splits)
    qed
  qed
  from IH(1) snd-eq False keep-eq cnt1'-def cnt2'-def
  show ?thesis by (simp add: Let-def rg-keep-def split: prod.splits)
 qed
 qed
 from both show fst (rg-fold cands cnt1) = fst (rg-fold cands cnt2) by simp
 from both show ∀α. (∃(p, K) ∈ set cands. α ∈ K) ⟶
       snd (rg-fold cands cnt1) α = snd (rg-fold cands cnt2) α by simp
qed
```

Corollary: the fold preserves count ≥ 1 for variables in candidate keys. We prove this by using $\forall \alpha.\ (\exists (p,\ K) \in set\ ?cands.\ \alpha \in K) \longrightarrow ?cnt1.0\ \alpha = ?cnt2.0\ \alpha \Longrightarrow fst\ (rg\text{-}fold\ ?cands\ ?cnt1.0) = fst\ (rg\text{-}fold\ ?cands\ ?cnt2.0)$ $\forall \alpha.\ (\exists (p,\ K) \in set\ ?cands.\ \alpha \in K) \longrightarrow ?cnt1.0\ \alpha = ?cnt2.0\ \alpha \Longrightarrow \forall \alpha.$

$(\exists\,(p,\ K){\in}\mathit{set}\ \mathit{?cands}.\ \alpha \in K) \longrightarrow \mathit{snd}\ (\mathit{rg\text{-}fold}\ \mathit{?cands}\ \mathit{?cnt1.0})\ \alpha = \mathit{snd}\ (\mathit{rg\text{-}fold}\ \mathit{?cands}\ \mathit{?cnt2.0})\ \alpha$ to relate the fold with *init-count* to the fold with a count that is ≥ 1 everywhere.

lemma *rg-fold-preserves-ge1-on-keys*:
  assumes $\forall\,\alpha.\ (\exists\,(p,\ K) \in \mathit{set\ cands}.\ \alpha \in K) \longrightarrow \mathit{cnt}\ \alpha \geq 1$
    and $(\exists\,(p,\ K) \in \mathit{set\ cands}.\ \alpha \in K)$
  shows $\mathit{snd}\ (\mathit{rg\text{-}fold\ cands\ cnt})\ \alpha \geq 1$
proof −
  define $\mathit{cnt'}$ **where** $\mathit{cnt'}\ \beta = (\mathit{if}\ \exists\,(p,\ K) \in \mathit{set\ cands}.\ \beta \in K\ \mathit{then\ cnt}\ \beta\ \mathit{else}\ 1)$
for β
  have *ge1*: $\forall\,\beta.\ \mathit{cnt'}\ \beta \geq 1$ **unfolding** *cnt′-def* **using** *assms*(*1*) **by** *auto*
  have *agree*: $\forall\,\beta.\ (\exists\,(p,\ K) \in \mathit{set\ cands}.\ \beta \in K) \longrightarrow \mathit{cnt}\ \beta = \mathit{cnt'}\ \beta$
    unfolding *cnt′-def* **by** *auto*
  from *rg-fold-cnt-agree*(*2*)[*OF agree*] *assms*(*2*)
  have $\mathit{snd}\ (\mathit{rg\text{-}fold\ cands\ cnt})\ \alpha = \mathit{snd}\ (\mathit{rg\text{-}fold\ cands\ cnt'})\ \alpha$ **by** *auto*
  then have $\mathit{snd}\ (\mathit{rg\text{-}fold\ cands\ cnt})\ \alpha = \mathit{snd}\ (\mathit{rg\text{-}fold\ cands\ cnt'})\ \alpha$ **by** *simp*
  also have $\ldots \geq 1$ **using** *rg-fold-preserves-ge1 ge1* **by** *auto*
  finally show *?thesis* **.**
qed

2.7.3 Witness Property

Lemma (Witness Variable). For every kept position p, there exists a variable α in its key such that α does not appear in the key of any other kept position.

The proof proceeds by induction on the candidate list, using a generalized invariant that tracks an extra count (representing kept positions from the prefix that are no longer in the candidate list). The key insight: in the keep case, the extra count being zero for the witness variable automatically excludes it from the current head's key, since the extra count absorbs the head's contribution.

Auxiliary generalized lemma: the count function may over-count by an extra amount. The conclusion finds a witness α whose extra contribution is zero, meaning α only appears in candidates from the current list (not from any prior kept positions).

lemma *rg-fold-witness-aux*:
  assumes *dist*: *distinct* (*map fst cands*)
    and *cnt-eq*: $\forall\,\alpha.\ \mathit{cnt}\ \alpha = \mathit{length}\ (\mathit{filter}\ (\lambda(\text{-},\ K).\ \alpha \in K)\ \mathit{cands}) + \mathit{extra}\ \alpha$
    and *mem*: $(p,\ K) \in \mathit{set\ cands}$
    and *kept*: $p \in \mathit{fst}\ (\mathit{rg\text{-}fold\ cands\ cnt})$
  shows $\exists\,\alpha \in K.\ \mathit{extra}\ \alpha = 0\ \wedge$
    $(\forall\,p'\ K'.\ (p',\ K') \in \mathit{set\ cands} \longrightarrow p' \in \mathit{fst}\ (\mathit{rg\text{-}fold\ cands\ cnt}) \longrightarrow p' \neq p \longrightarrow \alpha \notin K')$
  using *assms*
proof (*induction cands arbitrary*: *cnt extra p K*)
  case *Nil*
  then show *?case* **by** *simp*

```
next
  case (Cons pc rest)
  obtain ph Kh where pc-def: pc = (ph, Kh) by (cases pc)
  from Cons.prems(1) pc-def have dist-rest: distinct (map fst rest)
    and ph-notin: ph ∉ fst ‘ set rest by auto
  show ?case
  proof (cases p = ph)
    case True
```

Target is the head. First establish $K = Kh$ using distinctness.

```
    from Cons.prems(3) pc-def ‹p = ph› have (ph, K) ∈ set ((ph, Kh) # rest)
by simp
    then have K = Kh ∨ (ph, K) ∈ set rest by auto
    with ph-notin have K-eq: K = Kh by (auto simp: image-iff)
    show ?thesis
    proof (cases rg-keep cnt Kh)
      case True
```

Head is kept:

```
      from True obtain α where α-in: α ∈ Kh and cnt-le: cnt α ≤ 1
        unfolding rg-keep-def by auto
      from Cons.prems(2) pc-def α-in have
        cnt-eq-α: cnt α = 1 + length (filter (λ(-, K). α ∈ K) rest) + extra α
        by simp
      from cnt-le cnt-eq-α have rest-empty: length (filter (λ(-, K). α ∈ K) rest) =
0
        and extra-zero: extra α = 0 by arith+
      from rest-empty have no-α-rest: ∀(p', K') ∈ set rest. α ∉ K'
        by (auto simp: filter-empty-conv)
```

Show the witness property for α.

```
      have wit: ∀ p' K'. (p', K') ∈ set (pc # rest) ⟶
        p' ∈ fst (rg-fold (pc # rest) cnt) ⟶ p' ≠ p ⟶ α ∉ K'
      proof (intro allI impI)
        fix p' K'
        assume p'-in: (p', K') ∈ set (pc # rest)
          and p'-kept: p' ∈ fst (rg-fold (pc # rest) cnt)
          and p'-ne: p' ≠ p
        from p'-ne ‹p = ph› have p' ≠ ph by simp
        from p'-in pc-def ‹p' ≠ ph› have (p', K') ∈ set rest by auto
        with no-α-rest show α ∉ K' by auto
      qed
      from α-in K-eq extra-zero wit show ?thesis
        by (intro bexI[of - α]) auto
    next
      case False
```

Head is dropped but $p = ph$, so p must be in the result of the recursive call. This leads to a contradiction since *ph* cannot be in *fst* of *rest*'s result.

```
    define cnt' where cnt' = (λα. if α ∈ Kh then cnt α − 1 else cnt α)
    from False pc-def have fold-eq: rg-fold (pc # rest) cnt =
      (let (P', cnt'') = rg-fold rest cnt' in (P', cnt''))
      by (simp add: Let-def cnt'-def)
    obtain P' cnt'' where rec: rg-fold rest cnt' = (P', cnt'')
      by (cases rg-fold rest cnt')
   from fold-eq rec have fst (rg-fold (pc # rest) cnt) = P' by (simp add: Let-def)
    with Cons.prems(4) ‹p = ph› have ph ∈ P' by simp
    from rg-fold-subset[of rest cnt'] rec have P' ⊆ fst ‘ set rest by simp
    with ‹ph ∈ P'› ph-notin show ?thesis by auto
  qed
 next
  case False
```

Target p is in the tail.

```
  from Cons.prems(3) pc-def ‹p ≠ ph› have p-in-rest: (p, K) ∈ set rest by auto
  show ?thesis
  proof (cases rg-keep cnt Kh)
    case True
```

Head is kept. The recursive call uses the same *cnt*. The extra count for rest includes the head's contribution.

```
    obtain P' cnt-out where rec: rg-fold rest cnt = (P', cnt-out)
      by (cases rg-fold rest cnt)
    from True pc-def rec have fold-eq:
      fst (rg-fold (pc # rest) cnt) = insert ph P'
      by (simp add: Let-def)
    from Cons.prems(4) fold-eq ‹p ≠ ph› have p-kept-rest: p ∈ P' by simp
    with rec have p-in-fold: p ∈ fst (rg-fold rest cnt) by simp
```

Define the extra count for rest:

```
    define extra' where extra' α = (if α ∈ Kh then 1 else 0) + extra α for α
    have cnt-rest: ∀α. cnt α = length (filter (λ(-, K). α ∈ K) rest) + extra' α
    proof
      fix α
      from Cons.prems(2) pc-def have
        cnt α = length (filter (λ(-, K). α ∈ K) (pc # rest)) + extra α by simp
      then show cnt α = length (filter (λ(-, K). α ∈ K) rest) + extra' α
        unfolding extra'-def pc-def by simp
    qed
```

Apply IH on rest.

```
    from Cons.IH[OF dist-rest cnt-rest p-in-rest p-in-fold]
    obtain α where α-in: α ∈ K
      and extra'-zero: extra' α = 0
      and wit-rest: ∀p' K'. (p', K') ∈ set rest ⟶
        p' ∈ fst (rg-fold rest cnt) ⟶ p' ≠ p ⟶ α ∉ K'
      by auto
```

From $extra'\ \alpha = 0$, deduce $\alpha \notin Kh$ and $extra\ \alpha = 0$.

from *extra'-zero extra'-def* **have** *α-notin-Kh*: $\alpha \notin Kh$ **and** *extra-zero*: $extra\ \alpha = 0$
by (*auto split*: *if-splits*)

Extend the witness property to the full list.

have *wit-full*: $\forall p'\ K'.\ (p', K') \in set\ (pc \# rest) \longrightarrow p' \in fst\ (rg\text{-}fold\ (pc \# rest)\ cnt) \longrightarrow p' \neq p \longrightarrow \alpha \notin K'$
proof (*intro allI impI*)
fix $p'\ K'$
assume *p'-in*: $(p', K') \in set\ (pc \# rest)$
and *p'-kept*: $p' \in fst\ (rg\text{-}fold\ (pc \# rest)\ cnt)$
and *p'-ne*: $p' \neq p$
show $\alpha \notin K'$
using *α-notin-Kh p'-in pc-def ph-notin fold-eq rec p'-kept*
by (*fastforce simp*: *image-iff p'-ne wit-rest*)
qed
from *α-in extra-zero wit-full* **show** *?thesis* **by** (*intro bexI*[*of* - α]) *auto*
next
case *False*

Head is dropped. Decrement count for variables in *Kh*.

define *cnt'* **where** $cnt' = (\lambda\alpha.\ if\ \alpha \in Kh\ then\ cnt\ \alpha - 1\ else\ cnt\ \alpha)$
obtain *P' cnt''* **where** *rec*: $rg\text{-}fold\ rest\ cnt' = (P', cnt'')$
by (*cases rg-fold rest cnt'*)
from *False pc-def cnt'-def rec* **have** *fold-eq*:
$fst\ (rg\text{-}fold\ (pc \# rest)\ cnt) = P'$
by (*simp add*: *Let-def rg-keep-def split*: *prod.splits*)
from *Cons.prems*(4) *fold-eq* **have** *p-kept-rest*: $p \in P'$ **by** *simp*
with *rec* **have** *p-in-fold*: $p \in fst\ (rg\text{-}fold\ rest\ cnt')$ **by** *simp*

Compute the *cnt'* equation for rest.

have *cnt-rest*: $\forall \alpha.\ cnt'\ \alpha = length\ (filter\ (\lambda(\text{-}, K).\ \alpha \in K)\ rest) + extra\ \alpha$
proof
fix α
from *Cons.prems*(2) *pc-def* **have**
cnt-full: $cnt\ \alpha = (if\ \alpha \in Kh\ then\ 1\ else\ 0) + length\ (filter\ (\lambda(\text{-}, K).\ \alpha \in K)\ rest) + extra\ \alpha$
by *simp*
show $cnt'\ \alpha = length\ (filter\ (\lambda(\text{-}, K).\ \alpha \in K)\ rest) + extra\ \alpha$
proof (*cases* $\alpha \in Kh$)
case *True*
then have $cnt\ \alpha \geq 1$ **using** *cnt-full* **by** *simp*
with *True* **show** *?thesis* **unfolding** *cnt'-def* **using** *cnt-full* **by** *simp*
next
case *Falseα*: *False*
then show *?thesis* **unfolding** *cnt'-def* **using** *cnt-full* **by** *simp*
qed

```
    qed
```

Apply IH on rest with extra unchanged.

```
    from Cons.IH[OF dist-rest cnt-rest p-in-rest p-in-fold]
    obtain α where α-in: α ∈ K
      and extra-zero: extra α = 0
      and wit-rest: ∀ p' K'. (p', K') ∈ set rest ⟶
        p' ∈ fst (rg-fold rest cnt') ⟶ p' ≠ p ⟶ α ∉ K'
      by auto
```

Extend witness property to full list. The head was dropped, so $ph \notin P'$.

```
    from rg-fold-subset[of rest cnt'] rec have P' ⊆ fst ‘ set rest by simp
    with ph-notin have ph-notin-P: ph ∉ P' by auto
    have wit-full: ∀ p' K'. (p', K') ∈ set (pc # rest) ⟶
      p' ∈ fst (rg-fold (pc # rest) cnt) ⟶ p' ≠ p ⟶ α ∉ K'
    proof (intro allI impI)
      fix p' K'
      assume p'-in: (p', K') ∈ set (pc # rest)
        and p'-kept: p' ∈ fst (rg-fold (pc # rest) cnt)
        and p'-ne: p' ≠ p
      from p'-kept fold-eq have p' ∈ P' by simp
      with ph-notin-P have p' ≠ ph by auto
      from p'-in pc-def ‹p' ≠ ph› have (p', K') ∈ set rest by auto
      from ‹p' ∈ P'› rec have p' ∈ fst (rg-fold rest cnt') by simp
      from wit-rest ‹(p', K') ∈ set rest› this p'-ne
      show α ∉ K' by auto
    qed
    from α-in extra-zero wit-full show ?thesis by (intro bexI[of - α]) auto
  qed
 qed
qed
```

The main witness lemma: specialization with $extra = 0$.

```
lemma rg-fold-witness:
 assumes distinct (map fst cands)
   and ∀ α. cnt α = length (filter (λ(-, K). α ∈ K) cands)
   and (p, K) ∈ set cands and p ∈ fst (rg-fold cands cnt)
 shows ∃ α ∈ K. ∀ p' ∈ fst (rg-fold cands cnt). p' ≠ p ⟶
   (∀ K'. (p', K') ∈ set cands ⟶ α ∉ K')
 using rg-fold-witness-aux[OF assms(1) - assms(3,4)] assms(2)
 by fastforce
```

2.7.4 Connecting the Algorithm to the Locale

We now connect the reverse-greedy algorithm to the locale-based proofs. The candidate list is constructed from the enumeration of *a-star*, filtering for positions with non-empty keys. The reverse-greedy produces a set P satisfying coverage and witness properties.

context *annotation-problem*
begin

Candidate list: positions paired with their keys, derived from the post-order enumeration of *a-star*. Candidates are processed in the *enum-aterm* order (generalization to other orders is easy). Positions with empty keys are included but are always dropped by the reverse-greedy (the keep condition is vacuously false).

definition *candidates* :: (*nat* × *string set*) *list* **where**
  candidates = *map* (λ(*p*, *s*, τ). (*p*, *tvars-ty* τ ∩ *V*)) (*enum-aterm a-star*)

The candidate positions have distinct first components.

lemma *candidates-distinct*: *distinct* (*map fst candidates*)
 unfolding *candidates-def* **by** (*simp add*: *comp-def case-prod-beta distinct-enum-fst*)

For $\alpha \in V$, the initial count is at least 1.

lemma *candidates-count-ge1*:
  assumes α ∈ *V*
  shows *init-count candidates* α ≥ *1*
proof −
  from *coverage-initial*[*OF assms*] **obtain** *p* **where**
    p ∈ *pos-set a-star* α ∈ *key p* **by** *auto*
  from ‹α ∈ *key p*› **obtain** *s* τ **where**
    mem: (*p*, *s*, τ) ∈ *set* (*enum-aterm a-star*) **and** *tv*: α ∈ *tvars-ty* τ **and** α ∈ *V*
    unfolding *key-def* **by** *auto*
  then have *mem-cand*: (*p*, *tvars-ty* τ ∩ *V*) ∈ *set candidates*
    unfolding *candidates-def* **by** *force*
  have α ∈ *tvars-ty* τ ∩ *V* **using** *tv assms* **by** *auto*
  with *mem-cand* **have** (*p*, *tvars-ty* τ ∩ *V*) ∈ *set* (*filter* (λ(-, *K*). α ∈ *K*) *candidates*)
    by *auto*
  then have *0* < *length* (*filter* (λ(-, *K*). α ∈ *K*) *candidates*)
    by (*rule length-pos-if-in-set*)
  then have *length* (*filter* (λ(-, *K*). α ∈ *K*) *candidates*) ≥ *1* **by** *linarith*
  then show *?thesis* **unfolding** *init-count-def* **by** *simp*
qed

The kept set from the reverse-greedy is a subset of candidate positions, which are positions in *pos-set a-star*.

lemma *rg-result-subset*: *reverse-greedy candidates* ⊆ *pos-set a-star*
proof −
  have *reverse-greedy candidates* ⊆ *fst* ‘ *set candidates*
    unfolding *reverse-greedy-def* **using** *rg-fold-subset* **by** *auto*
  also have ... ⊆ *pos-set a-star*
    unfolding *candidates-def pos-set-def* **by** *force*
  finally show *?thesis* **.**
qed

Key in the candidate list matches key in the locale.

lemma *candidates-key-eq*:
  assumes $(p,\ K) \in$ *set candidates*
  shows $K =$ *key p*
  using *assms*
  unfolding *key-def candidates-def*
  by (*fastforce dest*: *enum-aterm-unique*)

Coverage property: the kept positions cover all inference variables.

lemma *rg-coverage*: $\bigcup$(*key* ‘ *reverse-greedy candidates*) = *V*
proof (*intro equalityI subsetI*)
  fix α **assume** $\alpha \in \bigcup$(*key* ‘ *reverse-greedy candidates*)
  then obtain *p* **where** $p \in$ *reverse-greedy candidates* $\alpha \in$ *key p* **by** *auto*
  then show $\alpha \in V$ **unfolding** *key-def* **by** *auto*
next
  fix α **assume** α-*V*: $\alpha \in V$

By $[\![\forall \alpha.\ (\exists (p,\ K) \in$*set ?cands*. $\alpha \in K) \longrightarrow 1 \leq$ *?cnt* $\alpha;\ \exists (p,\ K) \in$*set ?cands*. *?*$\alpha \in K]\!] \Longrightarrow 1 \leq$ *snd* (*rg-fold ?cands ?cnt*) *?*α, the final count for α is $\geq$ *1*. This means at least one candidate covering α was not dropped.

  from *coverage-initial*[*OF* α-*V*] **obtain** *p0* **where**
    p0 $\in$ *pos-set a-star* $\alpha \in$ *key p0* **by** *auto*
  from ‹$\alpha \in$ *key p0*› **obtain** *s0* τ*0* **where**
    mem0: (*p0*, *s0*, τ*0*) $\in$ *set* (*enum-aterm a-star*) **and** $\alpha \in$ *tvars-ty* τ*0* $\alpha \in V$
    unfolding *key-def* **by** *auto*
  then have (*p0*, *tvars-ty* τ*0* $\cap$ *V*) $\in$ *set candidates*
    unfolding *candidates-def* **by** *force*
  then have *ex-cand*: $\exists (p,\ K) \in$ *set candidates*. $\alpha \in K$
    using ‹$\alpha \in$ *tvars-ty* τ*0*› ‹$\alpha \in V$› **by** (*intro bexI*[*of* - (*p0*, *tvars-ty* τ*0* $\cap$ *V*)])
auto
  have *cnt-ge1*: $\forall \beta.\ (\exists (p,\ K) \in$ *set candidates*. $\beta \in K) \longrightarrow$ *init-count candidates* $\beta \geq 1$
  proof (*intro allI impI*)
    fix β **assume** $\exists (p,\ K) \in$ *set candidates*. $\beta \in K$
    then obtain *p1 K1* **where** (*p1*, *K1*) $\in$ *set candidates* $\beta \in$ *K1* **by** *auto*
    from *candidates-key-eq*[*OF this*(*1*)] ‹$\beta \in$ *K1*› **have** $\beta \in$ *key p1* **by** *auto*
    then have $\beta \in V$ **unfolding** *key-def* **by** *auto*
    then show *init-count candidates* $\beta \geq 1$ **using** *candidates-count-ge1* **by** *auto*
  qed
  from *rg-fold-preserves-ge1-on-keys*[*OF cnt-ge1 ex-cand*]
  have *final-ge1*: *snd* (*rg-fold candidates* (*init-count candidates*)) $\alpha \geq 1$ **.**

Final count $\geq$ *1* means at least one position covering α was kept. Since final count $\geq$ *1*, at least one candidate with α was kept.

  from *rg-fold-cnt-mono*[*of candidates init-count candidates* α]
  have *snd* (*rg-fold candidates* (*init-count candidates*)) $\alpha \leq$ *init-count candidates* α **.**

have $\exists (pk, Kk) \in$ *set candidates*. $\alpha \in Kk \wedge pk \in$ *fst* (*rg-fold candidates* (*init-count candidates*))
proof (*rule ccontr*)
assume $\neg$ ($\exists (pk, Kk) \in$ *set candidates*. $\alpha \in Kk \wedge pk \in$ *fst* (*rg-fold candidates* (*init-count candidates*)))
then have *no-kept*: $\forall (pk, Kk) \in$ *set candidates*. $\alpha \in Kk \longrightarrow pk \notin$ *fst* (*rg-fold candidates* (*init-count candidates*))
by *auto*

If no kept position covers α, the count must drop to *0*:

have *nk*: $\forall p\ K.\ (p, K) \in$ *set candidates* $\longrightarrow \alpha \in K \longrightarrow$
$p \notin$ *fst* (*rg-fold candidates* (*init-count candidates*))
using *no-kept* **by** *auto*
have *snd* (*rg-fold candidates* (*init-count candidates*)) $\alpha = 0$
by (*rule rg-fold-no-kept-zero*[*OF nk*]) (*simp add*: *init-count-def*)
with *final-ge1* **show** *False* **by** *simp*
qed
then obtain *pk Kk* **where** *pk-mem*: $(pk, Kk) \in$ *set candidates*
and $\alpha \in Kk$ **and** *pk-kept*: $pk \in$ *fst* (*rg-fold candidates* (*init-count candidates*))
by *auto*
from *pk-kept* **have** $pk \in$ *reverse-greedy candidates* **unfolding** *reverse-greedy-def* **by** *simp*
from *candidates-key-eq*[*OF pk-mem*] ‹$\alpha \in Kk$› **have** $\alpha \in$ *key pk* **by** *auto*
from ‹$pk \in$ *reverse-greedy candidates*› ‹$\alpha \in$ *key pk*›
show $\alpha \in \bigcup$(*key* ‘ *reverse-greedy candidates*) **by** *auto*
qed

Witness property: each kept position has a witness variable.

lemma *rg-witness*:
assumes $p \in$ *reverse-greedy candidates*
shows $\exists \alpha \in$ *key p*. $\forall p' \in$ *reverse-greedy candidates*. $p' \neq p \longrightarrow \alpha \notin$ *key* p'
proof −
from *assms* **have** *p-in*: $p \in$ *fst* (*rg-fold candidates* (*init-count candidates*))
unfolding *reverse-greedy-def* **by** *simp*
have $p \in$ *fst* ‘ *set candidates*
using *p-in rg-fold-subset*[*of candidates init-count candidates*] **by** *auto*
then obtain *K* **where** *pK1*: $(p, K) \in$ *set candidates*
by (*force simp*: *candidates-def*)
note *pK* = *pK1 p-in*

from *rg-fold-witness*[*OF candidates-distinct - pK*]
have *wit*: $\exists \alpha \in K.\ \forall p' \in$ *fst* (*rg-fold candidates* (*init-count candidates*)). $p' \neq p \longrightarrow$
($\forall K'.\ (p', K') \in$ *set candidates* $\longrightarrow \alpha \notin K'$)
by (*simp add*: *init-count-def*)
then obtain α **where** $\alpha \in K$ **and**
excl: $\forall p' \in$ *fst* (*rg-fold candidates* (*init-count candidates*)). $p' \neq p \longrightarrow$
($\forall K'.\ (p', K') \in$ *set candidates* $\longrightarrow \alpha \notin K'$) **by** *auto*
from *candidates-key-eq*[*OF pK*(*1*)] ‹$\alpha \in K$› **have** $\alpha \in$ *key p* **by** *simp*

```
  moreover have ∀ p' ∈ reverse-greedy candidates. p' ≠ p ⟶ α ∉ key p'
  proof (intro ballI impI)
    fix p' assume p' ∈ reverse-greedy candidates p' ≠ p
    then have p'-in: p' ∈ fst (rg-fold candidates (init-count candidates))
      unfolding reverse-greedy-def by simp
    from rg-fold-subset have p' ∈ fst ` set candidates
      using p'-in by (metis in-mono)
    then obtain K' where (p', K') ∈ set candidates by auto
    from excl p'-in ‹p' ≠ p› this have α ∉ K' by auto
    from candidates-key-eq[OF ‹(p', K') ∈ set candidates›] this
    show α ∉ key p' by simp
  qed
  ultimately show ?thesis by auto
qed
```

Instantiate *annotation-selection* with $P =$ *reverse-greedy candidates*. This makes the abstract completeness and local minimality theorems available as concrete theorems about the reverse-greedy algorithm's output.

```
sublocale annotation-selection const-type Γ a a-star σ reverse-greedy candidates
proof (unfold-locales)
  show reverse-greedy candidates ⊆ pos-set a-star
    by (rule rg-result-subset)
  show ⋃(key ` reverse-greedy candidates) = V
    by (rule rg-coverage)
  fix p assume p ∈ reverse-greedy candidates
  then show ∃ α ∈ key p. ∀ p' ∈ reverse-greedy candidates. p' ≠ p ⟶ α ∉ key p'
    by (rule rg-witness)
qed
```

The output of the full algorithm: *t-out* is the annotated raw term produced by running the reverse-greedy algorithm to select positions and then inserting annotations. This is the *t-out* from the paper's Theorem 1.

```
definition t-out :: raw-term where
  t-out = annotate σ a-star (reverse-greedy candidates)
```

Key property: stripping the annotations from *t-out* recovers the erasure of *a*.

```
lemma strip-t-out: strip t-out = erase a
  unfolding t-out-def using strip-annotate same-erasure by simp
```

Theorem (Completeness), stated in terms of *t-out*:

```
theorem completeness-t-out:
  assumes well-typed const-type Γ a'
    and erase a' = strip t-out
    and ∃ σ'. subst-aterm σ' a-star = a' ∧ (∀ α ∈ tvars-ctx Γ. σ' α = TVar α)
    and consistent-with a' (reverse-greedy candidates)
  shows a' = a
  using assms strip-t-out completeness by auto
```

Theorem (Local Minimality), stated for the reverse-greedy selection:

thm *local-minimality*

end

end
theory *Set-System*
imports *Main*
begin

3 AI-Authored proof formalization via independence systems

3.1 Independence Systems

An independence system consists of a finite ground set E and a family F of subsets satisfying: (1) the empty set is in F, and (2) F is hereditary (downward closed under subsets).

locale *indep-system* =
fixes E :: *'a set* **and** *indep* :: *'a set* ⇒ *bool*
assumes *finite-E*: *finite E*
assumes *empty-indep*: *indep* {}
assumes *indep-subset-carrier*: ⋀F. *indep F* ⟹ $F \subseteq E$
assumes *hereditary*: ⋀F F'. *indep F* ⟹ $F' \subseteq F$ ⟹ *indep F'*

A maximal independent set is one where no element can be added while preserving independence.

definition (**in** *indep-system*) *maximal* :: *'a set* ⇒ *bool* **where**
maximal F ⟷ *indep F* ∧ (∀ $e \in E - F$. ¬ *indep* ($F \cup \{e\}$))

lemma (**in** *indep-system*) *maximalI*:
assumes *indep F* ⋀e. $e \in E - F$ ⟹ ¬ *indep* ($F \cup \{e\}$)
shows *maximal F*
unfolding *maximal-def* **using** *assms* **by** *blast*

lemma (**in** *indep-system*) *maximalD*:
assumes *maximal F*
shows *indep F* ⋀e. $e \in E - F$ ⟹ ¬ *indep* ($F \cup \{e\}$)
using *assms* **unfolding** *maximal-def* **by** *blast+*

3.2 Best-In-Greedy Algorithm

The Best-In-Greedy algorithm processes elements of E in a given order, greedily adding each element if independence is preserved. We model it as a fold over a list.

definition *greedy-step* :: (*'a set* ⇒ *bool*) ⇒ *'a* ⇒ *'a set* ⇒ *'a set* **where**

greedy-step indep e F = (*if indep* (*F* ∪ {*e*}) *then F* ∪ {*e*} *else F*)

definition *best-in-greedy* :: ('*a set* ⇒ *bool*) ⇒ '*a list* ⇒ '*a set* **where**
best-in-greedy indep es = *foldl* (λ*F e*. *greedy-step indep e F*) {} *es*

We define the sequence of intermediate sets produced by the greedy algorithm.

fun *greedy-partial* :: ('*a set* ⇒ *bool*) ⇒ '*a list* ⇒ *nat* ⇒ '*a set* **where**
greedy-partial indep es 0 = {} |
greedy-partial indep es (*Suc i*) =
greedy-step indep (*es* ! *i*) (*greedy-partial indep es i*)

The final result equals the partial result after processing all elements.

lemma *best-in-greedy-foldl-partial*:
assumes *n* ≤ *length es*
shows *foldl* (λ*F e*. *greedy-step indep e F*) {} (*take n es*) = *greedy-partial indep es n*
using *assms*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
thus *?case*
by (*simp add*: *take-Suc-conv-app-nth*)
qed

lemma *best-in-greedy-eq-partial*:
best-in-greedy indep es = *greedy-partial indep es* (*length es*)
unfolding *best-in-greedy-def*
using *best-in-greedy-foldl-partial*[*of length es es indep*]
by *simp*

3.2.1 Properties of *greedy-partial*

The partial result is monotonically increasing.

lemma *greedy-partial-mono*:
i ≤ *j* ⟹ *greedy-partial indep es i* ⊆ *greedy-partial indep es j*
proof (*induction j*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc j*)
then show *?case*
by (*cases i* = *Suc j*) (*auto simp*: *greedy-step-def*)
qed

If an element was skipped at step *j*, adding it to F_j-1 was not independent.

lemma *greedy-partial-skip*:
assumes $j < length\ es\ es\ !\ j \notin greedy\text{-}partial\ indep\ es\ (Suc\ j)$
shows $\neg\ indep\ (greedy\text{-}partial\ indep\ es\ j \cup \{es\ !\ j\})$
using *assms* **by** (*simp add*: *greedy-step-def split*: *if-splits*)

The partial set is always contained in the first i elements.

lemma *greedy-partial-subset*:
assumes $i \leq length\ es$
shows $greedy\text{-}partial\ indep\ es\ i \subseteq set\ (take\ i\ es)$
using *assms*
proof (*induction i*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc i*)
then have $greedy\text{-}partial\ indep\ es\ i \subseteq set\ (take\ i\ es)$ **by** *simp*
moreover have $set\ (take\ i\ es) \subseteq set\ (take\ (Suc\ i)\ es)$
by (*simp add*: *set-take-subset-set-take*)
moreover have $es\ !\ i \in set\ (take\ (Suc\ i)\ es)$
using *Suc.prems* **by** (*simp add*: *take-Suc-conv-app-nth*)
ultimately show *?case*
by (*auto simp*: *greedy-step-def*)
qed

3.2.2 Theorem: Best-In-Greedy returns a maximal independent set

Theorem 2.1 of larry2.tex (Theorem 13.9 of Korte-Vygen). If (E, indep) is an independence system and es is a permutation of E, then the Best-In-Greedy output is a maximal independent set.

First we show the greedy output is independent, by induction.

lemma (**in** *indep-system*) *greedy-partial-indep*:
assumes $i \leq length\ es\ set\ es \subseteq E$
shows $indep\ (greedy\text{-}partial\ indep\ es\ i)$
using *assms*
proof (*induction i*)
case *0*
then show *?case* **using** *empty-indep* **by** *simp*
next
case (*Suc i*)
then have *IH*: $indep\ (greedy\text{-}partial\ indep\ es\ i)$ **by** *simp*
show *?case*
proof (*cases indep* $(greedy\text{-}partial\ indep\ es\ i \cup \{es\ !\ i\})$)
case *True*
then show *?thesis* **by** (*simp add*: *greedy-step-def*)
next
case *False*
then show *?thesis* **using** *IH* **by** (*simp add*: *greedy-step-def*)

```
  qed
qed
```

Now the main theorem.

```
theorem (in indep-system) best-in-greedy-maximal:
  assumes perm: set es = E and distinct: distinct es
  shows maximal (best-in-greedy indep es)
proof (rule maximalI)
  have set-es: set es ⊆ E using perm by simp

  show indep-result: indep (best-in-greedy indep es)
    unfolding best-in-greedy-eq-partial
    using greedy-partial-indep[of length es es] set-es by simp

  fix e assume e-outside: e ∈ E − best-in-greedy indep es
  show ¬ indep (best-in-greedy indep es ∪ {e})
  proof −
    from e-outside perm have e ∈ set es by simp
    then obtain j where j: j < length es es ! j = e
      by (metis in-set-conv-nth)
    from e-outside have e ∉ greedy-partial indep es (length es)
      unfolding best-in-greedy-eq-partial by simp
    then have e ∉ greedy-partial indep es (Suc j)
      using greedy-partial-mono[of Suc j length es indep es] j by auto
    then have not-indep-j: ¬ indep (greedy-partial indep es j ∪ {e})
      using greedy-partial-skip[of j es indep] j by simp
    have sub: greedy-partial indep es j ∪ {e} ⊆ best-in-greedy indep es ∪ {e}
      unfolding best-in-greedy-eq-partial
      using greedy-partial-mono[of j length es indep es] j by auto
    show ¬ indep (best-in-greedy indep es ∪ {e})
    proof
      assume indep (best-in-greedy indep es ∪ {e})
      then show False using hereditary[OF - sub] not-indep-j by blast
    qed
  qed
qed


end
theory Smolka-AI-Independence-System
  imports Main Set-System Smolka-AI
begin
```

Formalisation of the annotation algorithm via independence systems. We prove:

- The Best-In-Greedy algorithm returns a maximal independent set.
- The annotation set system is an independence system.

- Local minimality of the kept positions.
- The witness variable corollary.

3.3 The Annotation Set System

We now define the annotation set system. We abstract away from the concrete type-theoretic setting and work with:

- A finite set S of candidate positions (the ground set).
- A finite set V of inference variables.
- A key function mapping each position to a set of variables.
- The coverage condition: S covers V.

3.3.1 Setup

locale *annotation-setup* =
  fixes S :: *nat set*
    and V :: *string set*
    and *key* :: *nat* $\Rightarrow$ *string set*
  assumes *finite-S*: *finite S*
  assumes *finite-V*: *finite V*
  assumes *key-subset*: $\bigwedge p.\ p \in S \Longrightarrow key\ p \subseteq V$
  assumes *S-covers-V*: $\bigcup (key\ `\ S) = V$
begin

Coverage

definition *covers* :: *nat set* $\Rightarrow$ *bool* **where**
  covers $Q \longleftrightarrow V \subseteq \bigcup (key\ `\ Q)$

lemma *covers-S*: *covers S*
  unfolding *covers-def* **using** *S-covers-V* **by** *blast*

lemma *covers-mono*:
  assumes *covers* Q $Q \subseteq Q'$
  shows *covers* Q'
  unfolding *covers-def* **using** *assms* **unfolding** *covers-def* **by** *blast*

3.3.2 The Independence System

The annotation set system (S, F) is defined by: $F = \{F \subseteq S \| S \setminus F$ covers $V\}$ A set F represents positions that may be dropped: it is independent precisely when the remaining positions $S \setminus F$ still cover all inference variables.

definition *ann-indep* :: *nat set* $\Rightarrow$ *bool* **where**
  ann-indep $F \longleftrightarrow F \subseteq S \wedge$ *covers* $(S - F)$

3.3.3 The annotation set system is an independence system

lemma *ann-indep-empty*: *ann-indep* {}
  unfolding *ann-indep-def* **using** *covers-S* **by** *simp*

lemma *ann-indep-subset-carrier*:
  ann-indep F $\Longrightarrow$ $F \subseteq S$
  unfolding *ann-indep-def* **by** *simp*

lemma *ann-indep-hereditary*:
  assumes *ann-indep F* $F' \subseteq F$
  shows *ann-indep* F'
proof −
  from *assms* **have** $F \subseteq S$ **and** *cov*: *covers* $(S - F)$
    unfolding *ann-indep-def* **by** *auto*
  from *assms* **have** $F' \subseteq S$ **using** ‹$F \subseteq S$› **by** *blast*
  moreover have $S - F \subseteq S - F'$ **using** ‹$F' \subseteq F$› **by** *blast*
  then have *covers* $(S - F')$ **using** *cov covers-mono* **by** *blast*
  ultimately show *?thesis* **unfolding** *ann-indep-def* **by** *blast*
qed

The annotation set system forms an independence system.

sublocale *indep-system S ann-indep*
proof *unfold-locales*
  show *finite S* **by** (*rule finite-S*)
  show *ann-indep* {} **by** (*rule ann-indep-empty*)
  show $\bigwedge F$. *ann-indep F* $\Longrightarrow$ $F \subseteq S$ **by** (*rule ann-indep-subset-carrier*)
  show $\bigwedge F\ F'$. *ann-indep F* $\Longrightarrow$ $F' \subseteq F \Longrightarrow$ *ann-indep* F'
    by (*rule ann-indep-hereditary*)
qed

3.4 Feasibility and Count Condition

For F in the independence family with $p \in S - F$, we have $F \cup \{p\}$ is independent iff $1 <$ *var-count F* α for every $\alpha \in$ *key p*, where *var-count* α counts positions in $S - F$ covering α.

definition *var-count* :: *nat set* ⇒ *string* ⇒ *nat* **where**
  var-count F α = *card* $\{q \in S - F.\ \alpha \in$ *key* $q\}$

lemma *feasibility-iff-count*:
  assumes *ann-indep F p* $\in S - F$
  shows *ann-indep* $(F \cup \{p\}) \longleftrightarrow (\forall\, \alpha \in$ *key p*. *var-count F* $\alpha > 1)$
proof
  assume *indep-Fp*: *ann-indep* $(F \cup \{p\})$
  show $\forall\, \alpha \in$ *key p*. *var-count F* $\alpha > 1$
  proof
    fix α **assume** α-*in*: $\alpha \in$ *key p*
     from *indep-Fp* **have** *covers* $(S - (F \cup \{p\}))$ **unfolding** *ann-indep-def* **by** *simp*

```
    then have V-cov: V ⊆ ⋃(key ‘ (S − (F ∪ {p}))) unfolding covers-def .
    from α-in assms(2) key-subset have α ∈ V by blast
    then obtain q where q: q ∈ S − (F ∪ {p}) α ∈ key q
      using V-cov by blast
    then have q ≠ p q ∈ S − F by auto
    then have {p, q} ⊆ {r ∈ S − F. α ∈ key r}
      using α-in q(2) assms(2) by blast
    moreover have p ≠ q using ‹q ≠ p› by simp
    moreover have finite {r ∈ S − F. α ∈ key r} using finite-S by auto
    ultimately have card {r ∈ S − F. α ∈ key r} ≥ 2
      by (metis card-2-iff card-mono)
    then show var-count F α > 1 unfolding var-count-def by simp
  qed
next
  assume count-cond: ∀ α ∈ key p. var-count F α > 1
  have F ∪ {p} ⊆ S using assms unfolding ann-indep-def by auto
  moreover have covers (S − (F ∪ {p}))
  proof −
    have V-sub: V ⊆ ⋃(key ‘ (S − F))
      using assms(1) unfolding ann-indep-def covers-def by simp
    show ?thesis unfolding covers-def
    proof
      fix α assume α ∈ V
      then obtain q where q: q ∈ S − F α ∈ key q
        using V-sub by blast
      show α ∈ ⋃(key ‘ (S − (F ∪ {p})))
      proof (cases α ∈ key p)
        case True
        then have cnt-gt: card {r ∈ S − F. α ∈ key r} > 1
          using count-cond unfolding var-count-def by simp
       moreover have p-mem: p ∈ {r ∈ S − F. α ∈ key r} using True assms(2)
by simp
        moreover have fin: finite {r ∈ S − F. α ∈ key r} using finite-S by auto
        ultimately have card ({r ∈ S − F. α ∈ key r} − {p}) > 0
          by (simp add: card-Diff-singleton)
        then have {r ∈ S − F. α ∈ key r} − {p} ≠ {}
          using card-gt-0-iff fin finite-Diff by blast
        then obtain r where r ∈ S − F α ∈ key r r ≠ p by blast
        then show ?thesis by blast
      next
        case False
        then have q ≠ p using q(2) by blast
        then have q ∈ S − (F ∪ {p}) using q(1) by blast
        then show ?thesis using q(2) by blast
      qed
    qed
  qed
  ultimately show ann-indep (F ∪ {p}) unfolding ann-indep-def by blast
qed
```

3.5 Local Minimality

lemma *local-minimality-coverage*:
assumes *ann-indep F-star*
shows *covers* (*S* − *F-star*)
using *assms* **unfolding** *ann-indep-def* **by** *simp*

lemma *local-minimality*:
assumes *greedy-maximal*: *maximal F-star*
and *p-in-P*: $p \in S - F\text{-}star$
shows ¬ *covers* (*S* − *F-star* − {*p*})
proof −
from *greedy-maximal p-in-P* **have** ¬ *ann-indep* (*F-star* ∪ {*p*})
using *maximalD*(*2*) **by** *blast*
moreover have *F-star* ∪ {*p*} ⊆ *S*
using *maximalD*(*1*)[*OF greedy-maximal*] *indep-subset-carrier p-in-P* **by** *blast*
ultimately have ¬ *covers* (*S* − (*F-star* ∪ {*p*}))
unfolding *ann-indep-def* **by** *simp*
moreover have *S* − (*F-star* ∪ {*p*}) = *S* − *F-star* − {*p*} **by** *blast*
ultimately show *?thesis* **by** *simp*
qed

corollary *witness-variable*:
assumes *greedy-maximal*: *maximal F-star*
and *p-in-P*: $p \in S - F\text{-}star$
obtains α **where** $\alpha \in key\ p\ \alpha \in V$
$\bigwedge q.\ q \in S - F\text{-}star - \{p\} \Longrightarrow \alpha \notin key\ q$
proof −
let *?P* = *S* − *F-star*
have *cov*: *covers ?P*
using *maximalD*(*1*)[*OF greedy-maximal*] *local-minimality-coverage* **by** *simp*
have *not-cov*: ¬ *covers* (*?P* − {*p*})
using *local-minimality*[*OF greedy-maximal p-in-P*] **by** *simp*
from *not-cov* **obtain** α **where** α: $\alpha \in V\ \alpha \notin \bigcup(key\ `\ (?P - \{p\}))$
unfolding *covers-def* **by** *blast*
from *cov* **have** $V \subseteq \bigcup(key\ `\ ?P)$ **unfolding** *covers-def* **by** *simp*
with α(*1*) **obtain** *q* **where** $q \in ?P\ \alpha \in key\ q$ **by** *blast*
with α(*2*) **have** *q* = *p* **by** *blast*
then have $\alpha \in key\ p$ **using** ‹α ∈ *key q*› **by** *simp*
moreover have $\bigwedge q.\ q \in ?P - \{p\} \Longrightarrow \alpha \notin key\ q$
using α(*2*) **by** *blast*
ultimately show *?thesis* **using** *that* α(*1*) **by** *blast*
qed

3.6 Reverse Greedy Correctness

definition ‹*reverse-greedy′ es* = *S* − *best-in-greedy ann-indep es*›

theorem *annotation-algorithm-correct*:
assumes *perm*: *set es* = *S* **and** *distinct*: *distinct es*

shows *covers* (*reverse-greedy′ es*)
and $\bigwedge p.\ p \in$ *reverse-greedy′ es* $\Longrightarrow$
$\exists \alpha \in$ *key p*. $\alpha \in V \wedge (\forall q \in$ *reverse-greedy′ es* $- \{p\}.\ \alpha \notin$ *key q*)
proof −
have *mx*: *maximal* (*best-in-greedy ann-indep es*)
using *best-in-greedy-maximal*[*OF perm distinct*] **.**
then have *indep-star*: *ann-indep* (*best-in-greedy ann-indep es*)
using *maximalD* **by** *blast*
show *covers* (*reverse-greedy′ es*)
using *local-minimality-coverage*[*OF indep-star*] *reverse-greedy′-def* **by** *simp*
show $\bigwedge p.\ p \in$ *reverse-greedy′ es* $\Longrightarrow$
$\exists \alpha \in$ *key p*. $\alpha \in V \wedge (\forall q \in$ *reverse-greedy′ es* $- \{p\}.\ \alpha \notin$ *key q*)
using *witness-variable*[*OF mx*] *reverse-greedy′-def* **by** *metis*
qed
end

context *annotation-problem*
begin

The annotation problem gives rise to an annotation set system.

lemma *finite-tvars-ty*: *finite* (*tvars-ty t*)
by (*induction t*) *auto*

interpretation *ann*: *annotation-setup pos-set a-star V key*
proof *unfold-locales*
show *finite* (*pos-set a-star*) **by** (*simp add*: *pos-set-def*)
have *finite* (*tvars-aterm a-star*) **by** (*induction a-star*) (*auto simp*: *finite-tvars-ty*)
then show *finite V* **unfolding** *V-def* **by** *auto*
show $\bigwedge p.\ p \in$ *pos-set a-star* $\Longrightarrow$ *key p* $\subseteq V$
unfolding *key-def* **by** *auto*
show $\bigcup$(*key* ‘ *pos-set a-star*) $= V$
proof
show $\bigcup$(*key* ‘ *pos-set a-star*) $\subseteq V$ **unfolding** *key-def* **by** *auto*
next
show $V \subseteq \bigcup$(*key* ‘ *pos-set a-star*)
proof
fix α **assume** $\alpha \in V$
then have $\alpha \in$ *tvars-aterm a-star* **unfolding** *V-def* **by** *auto*
then obtain *p s* τ **where** (*p*, *s*, τ) $\in$ *set* (*enum-aterm a-star*) $\alpha \in$ *tvars-ty* τ
using *tvars-aterm-subset-enum* **by** *blast*
then have *p* $\in$ *pos-set a-star* **unfolding** *pos-set-def* **by** *force*
moreover have $\alpha \in$ *key p* **unfolding** *key-def* **using** ‹(*p*, *s*, τ) $\in$ -› ‹$\alpha \in$ *tvars-ty* τ› ‹$\alpha \in V$› **by** *auto*
ultimately show $\alpha \in \bigcup$(*key* ‘ *pos-set a-star*) **by** *auto*
qed
qed
qed

definition *es-from-enum* :: *nat list* **where**

es-from-enum = *map fst* (*enum-aterm a-star*)

lemma *es-set*: *set es-from-enum* = *pos-set a-star*
unfolding *es-from-enum-def pos-set-def*
by (*force simp*: *image-iff*)

lemma *es-distinct*: *distinct es-from-enum*
unfolding *es-from-enum-def*
by (*simp add*: *distinct-enum-fst*)

lemma *reverse-greedy′-subs-pos*: ‹*ann.reverse-greedy′ es-from-enum* ⊆ *pos-set a-star*›
by (*simp add*: *ann.reverse-greedy′-def*)

lemma *reverse-greedy′-coverage*: ‹⋃ (*key* ‘ *ann.reverse-greedy′ es-from-enum*) = *V*›
using *ann.annotation-algorithm-correct*(*1*)[*OF es-set es-distinct*] *ann.S-covers-V*

using *ann.covers-def ann.reverse-greedy′-def* **by** *auto*

lemma *reverse-greedy′-witness*:
assumes ‹*p* ∈ *ann.reverse-greedy′ es-from-enum*›
shows ‹∃ α∈*key p*. ∀ *p′*∈*ann.reverse-greedy′ es-from-enum*. *p′* ≠ *p* ⟶ α ∉ *key p′*›
using *assms ann.annotation-algorithm-correct*(*2*)[*OF es-set es-distinct*, *of p*]
by *auto*

interpretation *ann*: *annotation-selection*
const-type
Γ *a a-star* σ ‹*ann.reverse-greedy′ es-from-enum*›
using *reverse-greedy′-subs-pos reverse-greedy′-coverage reverse-greedy′-witness*
by *unfold-locales auto*

thm *ann.local-minimality*
thm *ann.completeness*
end
end